WebSphere MQ

Publish/Subscribe User's Guide

Version 7.0

SC34-6950-01

WebSphere MQ

Publish/Subscribe User's Guide

Version 7.0

SC34-6950-01

Note
Before using this information and the product it supports, be sure to read the general information under notices at the back of this book.

Second edition (January 2009)

This edition of the book applies to the following:

- IBM WebSphere MQ for Windows, Version 7.0

and to any subsequent releases and modifications until otherwise indicated in new editions.

Contents

Figures

Tables

Chapter 1. What's new in publish/subscribe in WebSphere MQ Version 7.0?

Publish/subscribe has been changed significantly for WebSphere® MQ Version 7.0. In previous versions, publish/subscribe messaging was controlled using a command message interface. This interface is deprecated in Version 7.0 publish/subscribe. Instead, publish/subscribe messaging is now controlled using new function in the WebSphere MQ API and as a result, publish/subscribe messaging is much more consistent with point-to-point messaging. This new way of doing publish/subscribe messaging is documented in the main body of this book.

Applications written using previous versions of WebSphere MQ publish/subscribe and make use of the command message interface are encouraged to move to the new WebSphere MQ publish/subscribe API – however, the command message interface continues to be supported by means of a process which runs on all platforms (including z/OS®). As such, if you are already a user of publish/subscribe you can continue to use your current configuration after installing WebSphere MQ Version 7.0 without making extensive changes to your applications or configuration.

Similarly, JMS applications do not have to be modified, although if you do not chose to use the new Version 7.0 publish/subscribe you will not benefit from the simplified administration that is now available when using WebSphere MQ as the provider. Since the command message interface method of doing publish/subscribe is still supported in Version 7.0 using the PSMODE function, this interface continues to be documented in the WebSphere MQ Version 7.0 library. This information is grouped together here Version 6 (queued) publish/subscribe.

Benefits of WebSphere Version 7.0 publish/subscribe

Publish/subscribe messaging is now performed using the WebSphere MQ API, there are a number of benefits of using this method.

Benefits of the new method of publish/subscribe include:

- In WebSphere MQ Version 7.0, support has been added for publish/subscribe messaging on z/OS.
- To perform some WebSphere MQ publish/subscribe functions in previous versions you required WebSphere Event Broker, WebSphere Message Broker or the MA0C WebSphere MQ SupportPac™ (if you were using WebSphere MQ Version 5.3), none of these applications are required now unless you want to route messages according to their content, in which case you can use WebSphere MQ in combination with WebSphere Message Broker.
- In WebSphere MQ Version 6.0, if you were using WebSphere MQ publish/subscribe you had to use PCFs or RFH1 headers, this is no longer the case.
- In WebSphere MQ Version 6.0 if you were using WebSphere Event Broker or WebSphere Message Broker you had to use RFH1 or RFH2 headers, this is no longer the case.

- In WebSphere MQ Version 7.0, the way you publish and subscribe is consistent with the rest of the WebSphere MQ API, making publish/subscribe more intuitive and easier to use.
- Native support for non-durable subscriptions has been added, allowing unconsumed messages and unnecessary subscriptions to be cleaned up at disconnection. This removes the need that existed in WebSphere MQ Version 6.0 for JMS to tidy up non-durable subscriptions in order to meet JMS specification requirements.
- By using non-durable subscriptions with JMS publish/subscribe messaging performance can be improved and resource usage made more efficient.
- The interlock between JMS and the queue manager is improved by using the new WebSphere MQ publish/subscribe API (rather than a command message interface).

Chapter 2. Introduction to WebSphere MQ publish/subscribe messaging

Publish/subscribe messaging allows you to decouple the provider of information, from the consumers of that information. The sending application and receiving application do not need to know anything about each other for the information to be sent and received.

Before a point-to-point WebSphere MQ application can send a message to another application, it needs to know something about that application. For example, it needs to know the name of the queue to which to send the information, and might also specify a queue manager name.

WebSphere MQ publish/subscribe removes the need for your application to know anything about the target application. All the sending application has to do, is put a WebSphere MQ message, containing the information that it wants, and assign it a topic, that denotes the subject of the information, and let WebSphere MQ handle the distribution of that information. Similarly, the target application does not have to know anything about the source of the information it receives.

Figure 1 shows the simplest publish/subscribe system. There is one publisher, one queue manager, and one subscriber. A subscription is sent from the subscriber to the queue manager, a publication is sent from the publisher to the queue manager, and the publication is then forwarded by the queue manager to the subscriber.

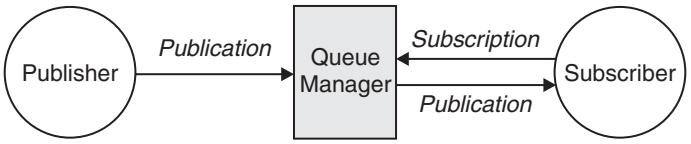


Figure 1. Simple publish/subscribe configuration

A typical publish/subscribe system has more than one publisher and more than one subscriber, and often, more than one queue manager. An application can be both a publisher and a subscriber.

Overview of publish/subscribe components

Publish/subscribe is the mechanism by which subscribers can receive information, in the form of messages, from publishers. The interactions between publishers and subscribers are controlled by queue managers, using standard WebSphere MQ facilities.

A typical publish/subscribe system has more than one publisher and more than one subscriber, and often, more than one queue manager. An application can be both a publisher and a subscriber.

The provider of information is called a *publisher*. Publishers supply information about a subject, without needing to know anything about the applications that are

interested in that information. Publishers generate this information in the form of messages, called *publications* that they want to publish and define the topic of these messages.

The consumer of the information is called a *subscriber*. Subscribers create *subscriptions* that describe the topic that the subscriber is interested in. Thus, the subscription determines which publications are forwarded to the subscriber. Subscribers can make multiple subscriptions and can receive information from many different publishers.

Published information is sent in a WebSphere MQ message, and the subject of the information is identified by its *topic*. The publisher specifies the topic when it publishes the information, and the subscriber specifies the topics about which it wants to receive publications. The subscriber is sent information about only those topics it subscribes to.

It is the existence of topics that allows the providers and consumers of information to be decoupled in publish/subscribe messaging by removing the need to include a specific destination in each message as is required in point-to-point messaging.

Interactions between publishers and subscribers are all controlled by a queue manager. The queue manager receives messages from publishers, and subscriptions from subscribers (to a range of topics). The queue manager's job is to route the published messages to the subscribers that have registered an interest in the topic of the messages.

Standard WebSphere MQ facilities are used to distribute messages, so your applications can use all the features that are available to existing WebSphere MQ applications. This means that you can use persistent messages to get once-only assured delivery, and that your messages can be part of a transactional unit-of-work to ensure that messages are delivered to the subscriber only if they are committed by the publisher.

Example of a single queue manager publish/subscribe configuration

Figure 2 on page 5 illustrates a basic single queue manager publish/subscribe configuration. The example shows the configuration for a news service, where information is available from publishers about several topics:

- Publisher 1 is publishing information about sports results using a topic of Sport
- Publisher 2 is publishing information about stock prices using a topic of Stock
- Publisher 3 is publishing information about film reviews using a topic of Films, and about television listings using a topic of TV

Three subscribers have registered an interest in different topics, so the queue manager sends them the information that they are interested in:

- Subscriber 1 receives the sports results and stock prices
- Subscriber 2 receives the film reviews
- Subscriber 3 receives the sports results

None of the subscribers have registered an interest in the television listings, so these are not distributed.

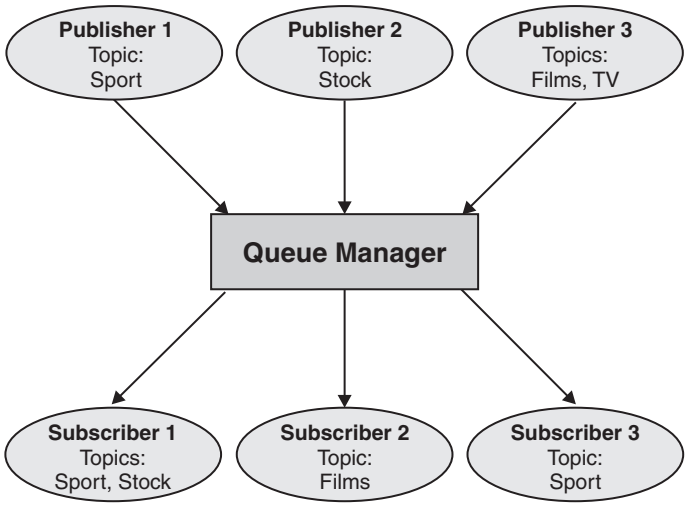


Figure 2. Single queue manager publish/subscribe example. This shows the relationship between publishers, subscribers, and queue managers.

Publishers and publications

In WebSphere MQ publish/subscribe a publisher is an application that makes information about a specified topic available to a queue manager in the form of a standard WebSphere MQ message called a publication. A publisher can publish information about more than one topic.

Publishers use the MQPUT verb to put a message to a previously opened topic, this message is a publication. The local queue manager then routes the publication to any subscribers who have subscriptions to the topic of the publication. A published message can be consumed by more than one subscriber.

In addition to distributing publications to all local subscribers that have appropriate subscriptions, a queue manager can also distribute the publication to any other queue managers connected to it, either directly or through a network of queue managers that have subscribers to the topic.

In a WebSphere MQ publish/subscribe network, a publishing application can also be a subscriber.

State and event information

Publications can be categorized as either state publications, such as the current price of a stock, or event publications, such as a trade in that stock.

State publications

State publications contain information about the current state of something, such as the price of stock or the current score in a soccer match. When something happens (for example, the stock price changes or the soccer score changes), the previous state information is no longer required because it is superseded by the new information.

A subscriber will want to receive the current version of the state information when it starts up, and be sent new information whenever the state changes.

If a publication contains state information, it is often published as a retained publication. A new subscriber typically wants the current state information immediately; the subscriber does not want to wait for an event that causes the information to be republished. Subscribers will automatically receive a topic's retained publication when it subscribes unless the subscriber uses the MQSO_PUBLICATIONS_ON_REQUEST or MQSO_NEW_PUBLICATIONS_ONLY options.

Event publications

Event publications contain information about individual events that occur, such as a trade in some stock or the scoring of a particular goal. Each event is independent of other events.

A subscriber will want to receive information about events as they happen.

Retained publications

By default, after a publication is sent to all interested subscribers it is discarded. However, a publisher can specify that a copy of a publication should be retained so that it can be sent to future subscribers who register an interest in the topic.

Deleting publications after they have been sent to all interested subscribers is suitable for event information, but is not always suitable for state information. By retaining a message, new subscribers do not have to wait for information to be published again before they receive initial state information. For example, a subscriber with a subscription to a stock price would receive the current price straight away, without waiting for the stock price to change (and hence be re-published).

The queue manager can retain only one publication for each topic, so a topic's existing retained publication is deleted when a new retained publication arrives at the queue manager. Wherever possible, have no more than one publisher sending retained publications on any topic.

Subscribers can specify that they do not want to receive retained publications by using the MQSO_NEW_PUBLICATIONS_ONLY subscription option. Existing subscribers can ask for duplicate copies of retained publications to be sent to them.

There are times when you might not want to retain publications, even for state information:

- If all subscriptions to a topic are made before any publications are made on that topic, and you do not expect, or will not allow, new subscriptions, there is no need to retain publications because they will be delivered to the complete set of subscribers the first time they are published.
- If publications occur very frequently, such as every second, a new subscriber (or a subscriber recovering from a failure) receives the current state almost immediately after their initial subscription, so there is no need to retain these publications.
- If the publications are quite large, you could end up needing a considerable amount of storage space to store the retained publication for each topic. In a multiple queue manager environment, retained publications are stored by all queue managers in the network that have a matching subscription.

When deciding whether to use retained publications, consider how subscribing applications recover from a failure. If the publisher does not use retained publications, the subscriber application might need to store its current state locally.

To ensure that a publication is retained use the MQPMO_RETAIN put-message option. If this option is used and the publication cannot be retained, the message will not be published and the call will fail with MQRC_PUT_NOT_RETAINED.

If a message is a retained publication this will be indicated by the MQIsRetained message property.

Subscribers and subscriptions

In WebSphere MQ publish/subscribe, a subscriber is an application that requests information about a specific topic from a queue manager in a publish/subscribe network. A subscriber can receive messages, about the same or different topics, from more than one publisher.

Subscriptions can be created manually using an MQSC command or by applications. These subscriptions are issued to the local queue manager and contain information about the publications the subscriber wants to receive:

- The topic the subscriber is interested in; this can resolve to multiple topics if wildcards are used.
- An optional selection string to be applied to published messages.
- A handle to a queue (known as the *subscriber queue*), on which selected publications should be placed, and the optional CorrelId.

The local queue manager stores subscription information and when it receives a publication, scans the information to determine whether there is a subscription that matches the publication's topic and selection string. For each matching subscription, the queue manager directs the publication to the subscriber's subscriber queue. The information that a queue manager stores about subscriptions can be viewed by using the DIS SBSTATUS command.

A subscription is deleted only when one of the following events occurs:

- The subscriber unsubscribes using the MQCLOSE call (if the subscription was made non-durably).
- The subscription expires.
- The subscription is deleted by the system administrator using the DELETE SUB command.
- The subscriber application ends (if the subscription was made non-durably).
- The queue manager is stopped or restarted (if the subscription was made non-durably).

Managed queues and publish/subscribe

When you create a subscription you can choose to use managed queuing. If you use managed queueing a subscription queue is automatically created when you create a subscription. Managed queues are tidied up automatically in accordance with the durability of the subscription. Using managed queues means that you do not have to worry about creating queues to receive publications and any unconsumed publications are removed from subscriber queues automatically if a non-durable subscription connection is closed.

If an application has no need to use a particular queue as its subscriber queue, the destination for the publications it receives, it can make use of the *managed subscriptions* using the MQSO_MANAGED subscription option. If you create a managed subscription, the queue manager returns an object handle to the subscriber for a subscriber queue that the queue manager creates where publications will be received. The queue's object handle will be returned allowing you to browse, get or inquire on the queue (it is not possible to put to or set attributes of a managed queue unless you have been explicitly given access to temporary dynamic queues).

The durability of the subscription determines whether the managed queue remains when the subscribing application's connection to the queue manager is broken.

Managed subscriptions are particularly useful when used with non-durable subscriptions because when the application's connection is ended, unconsumed messages would otherwise remain on the subscriber queue taking up space in your queue manager indefinitely. If you are using a managed subscription, the managed queue will be a temporary dynamic queue and as such will be deleted along with any unconsumed messages when the connection is broken for any of the following reasons:

- MQCLOSE with MQCO_REMOVE_SUB is used and the managed Hobj is closed.
- a connection is lost to an application using a non-durable subscription (MQSO_NON_DURABLE).
- a subscription is removed because it has expired and the managed Hobj is closed.

Managed subscriptions can also be used with durable subscriptions but it is possible that you would want to leave unconsumed messages on the subscriber queue so that they can be retrieved when the connection is reopened. For this reason, managed queues for durable subscriptions take the form of a permanent dynamic queue and will remain when the subscribing application's connection to the queue manager is broken.

You can set an expiry on your subscription if you want to use permanent dynamic managed queue so that although the queue will still exist after the connection is broken, it will not continue to exist indefinitely.

If you delete the managed queue you will receive an error message.

The managed queues that are created are named with numbers at the end (timestamps) so that each is unique.

Subscription durability

Subscriptions can be configured to be durable or non-durable. Subscription durability determines what happens to subscriptions when subscribing applications disconnect for a queue manager.

Durable subscriptions

Durable subscriptions continue to exist when a subscribing application's connection to the queue manager is closed. If a subscription is durable, when the subscribing application disconnects, the subscription remains in place and can be used by the subscribing application when it reconnects requesting the subscription again using the SubName that was returned when the subscription was created.

When subscribing durably, a subscription name (SubName) is required. Subscription names must be unique within a queue manager so that it can be used to identify a subscription. This means of identification is necessary when specifying a subscription you want to resume, if you have either deliberately closed the handle to the subscription (using the MQCO_KEEP_SUB option) or have been disconnected from the queue manager. You can resume an existing subscription by using the MQSUB call with the MQSO_RESUME option. Subscription names are also displayed if you use the DISPLAY SBSTATUS command with SUBTYPE ALL or ADMIN.

When an application no longer requires a durable subscription it can be removed using the MQCLOSE function call with the MQCO_REMOVE_SUB option or it can be deleted manually use the MQSC command DELETE SUB.

Whether durable subscriptions can be made to a topic can be controlled using the **DURSUB** topic attribute.

On return from an MQSUB call using the MQSO_RESUME option, subscription expiry will be set to the original expiry of the subscription and not the remaining expiry time.

A queue manager will continue to send publications to satisfy a durable subscription even if that subscriber application is not connected. This will lead to a build up of messages on the subscriber queue. The easiest way to avoid this problem is to use a non-durable subscription wherever appropriate. However, where it is necessary to use durable subscriptions, a build up of messages can be avoided if the subscriber subscribes using the MQSO_PUBLICATIONS_ON_REQUEST option. A subscriber can then control when it receives publications by using the MQSUBRQ call.

Non-durable subscriptions

Non-durable subscriptions exist only as long as the subscribing application's connection to the queue manager remains open. The subscription is removed when the subscribing application disconnects from the queue manager either deliberately or by loss of connection. When the connection is closed, the information about the subscription is removed from the queue manager, and will no longer be shown if you display subscriptions using the DISPLAY SBSTATUS command. No more messages will be put to the subscriber queue.

What happens to any unconsumed publications on the subscriber queue for non-durable subscriptions is determined as follows.

- If a subscribing application is using a managed destination, any publications that have not been consumed are automatically removed.
- If the subscribing application provides a handle to its own subscriber queue when it subscribes, unconsumed messages are not removed automatically. It is the responsibility of the application to clear the queue if that is appropriate. If the queue is shared by more than one subscriber, or other point-to-point applications, it might not be appropriate to clear the queue completely.

Although not required for non durable subscriptions, a subscription name if provided, will be used by the queue manager. Subscription names must be unique within the queue manager so that it can be used to identify a subscription.

Selection strings

A *selection string* is an expression that is applied to a publication to determine whether it matches a subscription. Selection strings can include wildcard characters.

When you subscribe, in addition to specifying a topic, you can specify a selection string to select publications according to their message properties.

Topics

A topic is the subject of the information that is published in a publish/subscribe message.

Instead of including a specific destination address in each message, messages in publish/subscribe systems are forwarded to subscribers based on either the subject that describes the contents of the message, or on the contents of the message itself.

The WebSphere MQ publish/subscribe system is a subject based publish/subscribe system. A publisher creates a message, and publishes it with a topic string that best fits the subject of the publication. To receive publications, a subscriber creates a subscription with a pattern matching topic string to select publication topics. The queue manager delivers publications to subscribers that have subscriptions that match the publication topic, and are authorized to receive the publications. The article, “Topic strings” on page 11, describes the syntax of topics strings and the two alternative wild card schemes,that subscribers use to create pattern matching topic strings are described in “Wild card schemes” on page 12.

In subject-based publish/subscribe, publishers, or administrators, are responsible for classifying subjects into topics. Typically subjects are organized hierarchically, into topic trees, using the ‘/’ character to create subtopics in the topic string. See “Topic trees” on page 17 for examples of topic trees. Topics are nodes in the topic tree. Topics can be leaf-nodes with no further subtopics, or intermediate nodes with sub-topics.

In parallel with organizing subjects into a hierarchical topic tree, you can associate topics with administrative topic objects. You assign attributes to a topic, such as whether the topic is distributed in a cluster, by associating it with an administrative topic object. The association is made by naming the topic using the **TOPICSTR** attribute of the administrative topic object. If you do not explicitly associate an administrative topic object to a topic, the topic either inherits the attributes of its closest ancestor in the topic tree that you *have* associated with an administrative topic object, or inherits from **SYSTEM.BASE.TOPIC**. Administrative topic objects are described in “Administrative topic objects” on page 18.

When you refer to a topic as a publisher or subscriber, you have a choice of supplying a topic string or referring to a topic object. You can also do both, in which case the topic string you supply defines a subtopic. The queue manager identifies the topic by adding the subtopic to the ″master″ topic named in the topic object, inserting an additional ‘/’ in between the two topic strings. This is described further in “Constructing topic names” on page 16. The resulting topic string is used to identify the topic and associate it with an administrative topic object. The administrative topic object is not necessarily the same topic object as the topic object corresponding to the master topic.

In content based publish/subscribe, you define what messages you want to receive by providing selection strings that search the contents of every message. WebSphere MQ provides a intermediate form of content based publish/subscribe using message selectors that scan message properties rather than the full content of the message, see Selectors. The archetypal use of message selectors is to subscribe to a topic and then qualify the selection with a numerical property. The selector enables you to specify you are interested in values only in a certain range; something you cannot do using either character or topic based wild cards. If you do need to filter based on the full content of the message, you need to use WebSphere Message Broker.

Topic strings

Label information you publish as a topic using a topic string. Subscribe to groups of topics using either character or topic based wild card topic strings.

Topics

A *topic string* is a character string that identifies the topic of a publish/subscribe message. You can use any characters you like when you construct a topic string.

Topic string

Three characters have special meaning in version 7 publish/subscribe. They are allowed anywhere in a topic string, but use them with caution. The use of the special characters is explained in "Topic based wild card scheme" on page 12.

A forward slash (/)

The topic level separator. Use the '/' character to structure the topic into a topic tree.

Avoid empty topic levels, '//', if you can. These correspond to nodes in the topic hierarchy with no topic string. A leading or trailing '/' in a topic string corresponds to a leading or trailing empty node and should be avoided too.

The hash sign (#)

Used in combination with '/' to construct a multilevel wild card in subscriptions. Take care using '#' adjacent to '/' in topic strings used to name published topics. "Examples of topic strings" on page 12 shows a sensible use of '#'.

The strings '.../#/...', '#/...' and '.../#' have a special meaning in subscription topic strings. The strings match all topics at one or more levels in the topic hierarchy. Thus if you created a topic with one of those sequences, you could not subscribe to it, without also subscribing to all topics at multiple levels in the topic hierarchy.

The plus sign (+)

Used in combination with '/' to construct a single-level wildcard in subscriptions. Take care using '+' adjacent to '/' in topic strings used to name published topics.

The strings '.../+/...', '+/...' and '.../+' have a special meaning in subscription topic strings. The strings match all topics at one level in the topic hierarchy. Thus if you created a topic with one of those sequences, you could not subscribe to it, without also subscribing to all topics at one level in the topic hierarchy.

Examples of topic strings

```
IBM/Business Area#/Results
IBM/Diversity/%African American
```

Wild card schemes

There are two wild card schemes used to subscribe to multiple topics. The choice of scheme is a subscription option.

MQSO_WILDCARD_TOPIC
Select topics to subscribe to using the topic based wild card scheme.

MQSO_WILDCARD_CHAR
Select topics to subscribe to using the character based wild card scheme.

Subscriptions that were created prior to WebSphere MQ Version 7.0 use the character based wild card scheme.

Examples

```
IBM/+/Results
#/Results
IBM/Software/Results
IBM/*ware/Results
```

Topic based wild card scheme:

Topic based wild cards allow subscribers to subscribe to more than one topic at a time.

Topic based wildcards are a powerful feature of the topic system in WebSphere MQ publish/subscribe. The multilevel wildcard and single level wildcard can be used for subscriptions, but they cannot be used within a topic by the publisher of a message.

The topic based wild card scheme allows you to select publications grouped by topic level. You can choose for *each level in the topic hierarchy*, whether the string in the subscription for that topic level must exactly match the string in the publication or not. For example the subscription, `IBM/+/Results` selects all the topics,

```
IBM/Software/Results
IBM/Services/Results
IBM/Hardware/Results
```

There are two types of wild card.

Multilevel wild card

- The multilevel wildcard is used in subscriptions. When used in a publication it is treated as a literal.
- The multilevel wildcard character "#" is used to match any number of levels within a topic. For example, using the example topic tree shown above, if you subscribe to "USA/Alaska/#", you receive messages on topics "USA/Alaska" and "USA/Alaska/Juneau".

- The multilevel wildcard can represent zero or more levels. Therefore, ″USA/#″ can also match the singular ″USA″, where # represents zero levels. The topic level separator is meaningless in this context, because there are no levels to separate.
- The multilevel wildcard is only effective when specified on its own or next to the topic level separator character. Therefore, ″#″ and ″USA/#″ are valid topics where the ″#″ character is treated as a wildcard. However, although ″USA#″ is also a valid topic string, the ″#″ character is not regarded as a wildcard and does not have any special meaning. See “When topic based wild cards are not wild” on page 15 for more information.

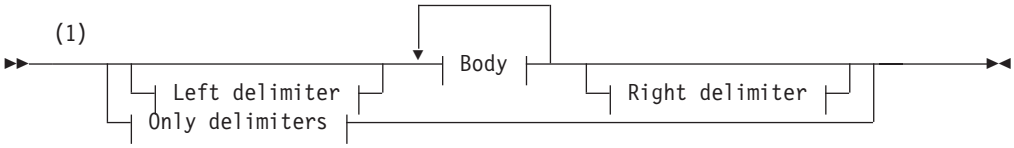
Single level wild card

- The multilevel wildcard in used in subscriptions. When used in a publication it is treated as a literal.
- The single-level wildcard character ″+″ matches one, and only one, topic level. For example, ″USA/+″ matches ″USA/Alabama″, but not ″USA/Alabama/Auburn″. Because the single-level wildcard matches only a single level, ″USA/+″ does not match ″USA″.
- The single-level wildcard can be used at any level in the topic tree, and in conjunction with the multilevel wildcard. The single-level wildcard must be specified next to the topic level separator, except when it is specified on its own. Therefore, ″+″ and ″USA/+″ are valid topics where the ″+″ character is treated as a wildcard. However, although ″USA+″ is also a valid topic string, the ″+″ character is not regarded as a wildcard and does not have any special meaning. See “When topic based wild cards are not wild” on page 15 for more information.

The syntax for the topic based wild card scheme, described in Topic based wild card string, has no escape characters. Whether ‘#’ and ‘+’ are treated as wild cards or not depends on their context. See “When topic based wild cards are not wild” on page 15 for more information.

Note: The beginning and end of a topic string is treated in a special way. Using $ to denote the end of the string, then `$#/...` is a multilevel wild card, and `$/#/...` is an empty node at the root, followed by a multilevel wild card.

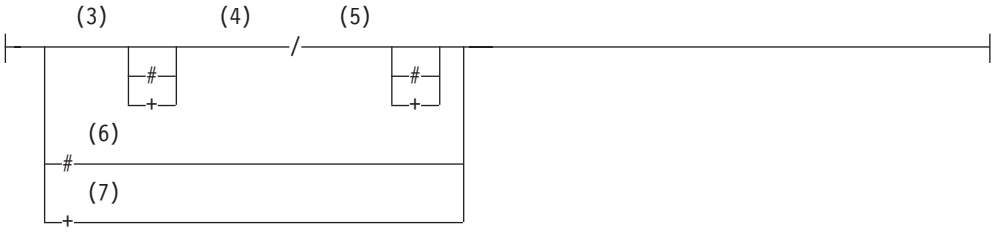
Topic based wild card string



Body:

Topic level name:

Only delimiters:



Notes:

1 A null or zero length topic string is invalid

2 You are advised not to use any of `*`, `?`, `%` in level name strings for compatibility between char based and topic based wildcard schemes.

3 These cases are equivalent to the pattern.

4 / with no wild cards matches a single empty topic.

5 These cases are equivalent to the pattern.

6 Match every topic.

7 Match every topic where there is only one level.

Left delimiter:

Body delimiter:

Right delimiter:

Notes:

1 The topic string starts with an empty topic

2 Matches zero or more levels. Multiple multi-level match strings have the same affect as one multi-level match string.

3 Matches exactly one level.

4 // is an empty topic - a topic object with no topic string.

5 The topic string ends with an empty topic

When topic based wild cards are not wild

The wildcard characters ″+″ and ″#″ have no special meaning when they are mixed with other characters (including themselves) in a topic level.

This means that topics that contain ″+″ or ″#″ together with other characters in a topic level can be published.

For example, consider the following two topics:

1. level0/level1/+/level4/#
2. level0/level1/#+/level4/level#

In the first example, the characters ″+″ and ″#″ are treated as wildcards and are therefore not valid in a topic string that is to be published to but are valid in a subscription.

In the second example, the characters ″+″ and ″#″ are not treated as wildcards and therefore the topic string can be both published and subscribed to.

Examples

```
IBM/+/Results
#/Results
IBM/Software/Results
```

Character based wild card scheme:

The character based wild card scheme allows you to select topics based on traditional character matching.

You can select all topics at multiple levels in a topic hierarchy using the string ‘*’. This is equivalent to using the topic based wild card string ‘#’

x/*/*y* is equivalent to *x*/#/*y* in the topic based scheme, and selects all topics in the topic hierarchy between levels *x* and *y*, where *x* and *y* are topic names that are not in the set of levels returned by the wild card.

/+/ in the topic based scheme has no exact equivalent in the character based scheme. IBM/*/Results would also select IBM/Patents/Software/Results. Only if

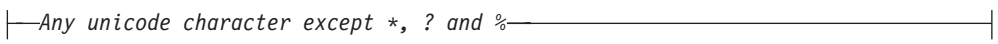
the set of topic names at each level of the hierarchy are unique, can you always construct queries with the two schemes that yield identical matches.

Used in a general way, * and ? in the character based scheme have no equivalents in the topic based scheme. The topic based scheme does not perform partial matching using wild cards. The character based wild card subscription IBM/*ware/Results has no topic based equivalent.

Note: Matches using character wild card subscriptions are slower than matches using topic based subscriptions.

Character based wild card string

V6 literal:



Notes:

1 Means "Escape the following character", so that it is treated as a literal. % must be followed by either *, ? or %. See "Examples of topic strings" on page 12.

2 Means "Match zero or more characters" in a subscription.

3 Means "Match exactly one character" in a subscription.

Examples

```
IBM/*/Results
IBM/*ware/Results
```

Constructing topic names

A topic is constructed from the subtopic identified in a topic object, and a subtopic provided by an application. You can use either subtopic as the topic name, or combine them to form a new topic name.

In an MQI program the full topic name is created by MQOPEN. It is composed of two fields:

1. The **TOPICSTR** attribute of the topic object.
2. The **ObjectString** parameter defining the subtopic provided by the application.

The resulting topic string is returned in the **ResObjectString** parameter.

These fields are considered to be present if the first character of each field is neither a blank nor a null character, and the field length is greater than zero. If only one of the fields is present, it is used unchanged as the topic name. If neither field has a value the call fails with reason code MQRC_TOPIC_STRING_ERROR.

If both fields are present, a '/' character is inserted between the two elements of the resultant combined topic name.

Table 1 shows examples of topic string concatenation:

Table 1. Topic string concatenation examples

TOPICSTR	ObjectString	Full topic name	Comment
Football/Scores	''	Football/Scores	The TOPICSTR is used alone
''	Football/Scores	Football/Scores	The ObjectString is used alone
Football	Scores	Football/Scores	A '/' character is added at the concatenation point
Football	/Scores	Football//Scores	An 'empty node' is produced between the two strings
/Football	Scores	/Football/Scores	The topic starts with an 'empty node'

Example code snippet

This code snippet, extracted from the example program, Simple WebSphere MQ publisher to a variable topic, combines a topic object with a variable topic string.

```
MQOD     td = {MQOD_DEFAULT};  /* Topic Descriptor             */
td.ObjectType = MQOT_TOPIC;    /* Object is a topic            */
td.Version = MQOD_VERSION_4;   /* Descriptor needs to be V4    */
strncpy(td.ObjectName, topicName,  MQ_TOPIC_NAME_LENGTH);
td.ObjectString.VSPtr = topicString;
td.ObjectString.VSLength = (MQLONG)strlen(topicString);
td.ResObjectString.VSPtr = resTopicStr;
td.ResObjectString.VSBufSize = sizeof(resTopicStr)-1;
MQOPEN(Hconn, &td, MQOO_OUTPUT | MQOO_FAIL_IF_QUIESCING, &Hobj, &CompCode, &Reason);
```

Topic trees

Each topic that you define is an element, or node, in the topic tree. The topic tree can either be empty to start with or contain topics that have been defined previously using MQSC or PCF commands. You can define a new topic either by using the create topic commands or by specifying the topic for the first time in a publication or subscription.

Although you can use any character string to define a topic's topic string, it is advisable to choose a topic string that fits into a hierarchical tree structure. Thoughtful design of topic stings and topic trees can help you with the following operations:

- Subscribing to multiple topics.
- Establishing security policies.

Although you can construct a topic tree as a flat, linear structure, it is better to build a topic tree in a hierarchical structure with one or more root topics.

Figure 3 shows an example of a topic tree with one root topic.

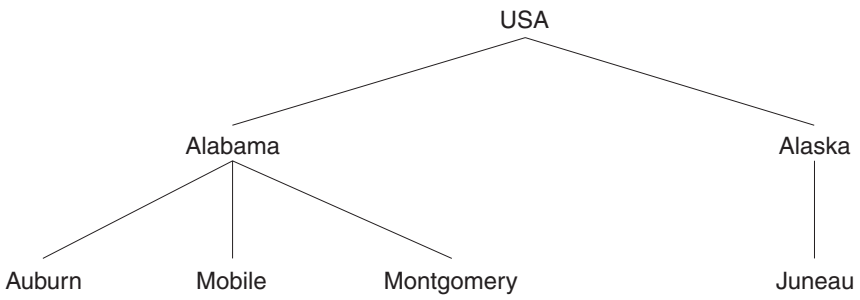


Figure 3. Example of a topic tree

Each character string in the figure represents a node in the topic tree. A complete topic string is created by aggregating nodes from one or more levels in the topic tree. Levels are separated by the ″/″ character. The format of a fully specified topic string is: ″root/level2/level3″.

The valid topics in the topic tree shown in Figure 3 are:
″USA″
″USA/Alabama″
″USA/Alaska″
″USA/Alabama/Auburn″
″USA/Alabama/Mobile″
″USA/Alabama/Montgomery″
″USA/Alaska/Juneau″

When you design topic strings and topic trees, remember that the queue manager does not interpret, or attempt to derive meaning from, the topic string itself. It simply uses the topic string to send selected messages to subscribers of that topic.

The following principles apply to the construction and content of a topic tree:

- There is no limit to the number of levels in a topic tree.
- There is no limit to the length of the name of a level in a topic tree.
- There can be any number of ″root″ nodes; that is, there can be any number of topic trees.

Administrative topic objects

An *administrative topic object* is a WebSphere MQ object that allows you to assign specific, non-default attributes to *topics*.

Figure 4 on page 19 shows how a high-level topic of 'Sport' divided into separate topics covering different sports can be visualized as a topic tree:

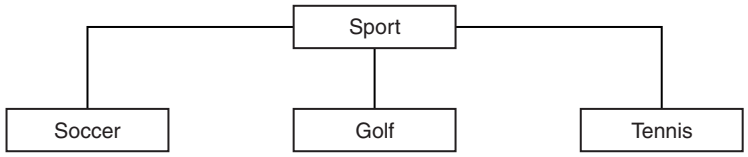


Figure 4. Visualization of a topic tree

Figure 5 shows how the topic tree can be divided further, to separate different types of information about each sport:

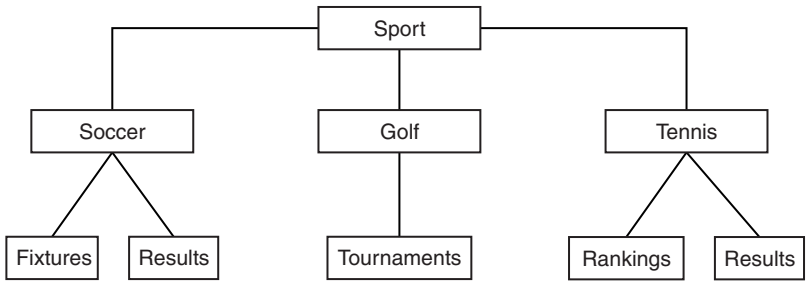


Figure 5. Extended topic tree

To create the topic tree illustrated, no administrative topic objects need be defined. If each of the nodes in this tree are defined by a topic string created in a publish or subscribe operation, each topic in the tree inherits its attributes from its parent. Attributes are inherited from the parent topic object because by default all attributes are set to ASPARENT. In this example, therefore, every topic has the same attributes as the 'Sport' topic, which again, assuming no administrative topic object exists for this node, inherits its attributes from SYSTEM.BASE.TOPIC.

Administrative topic objects can be used to define specific attributes for particular nodes in the topic tree. In the following example, the administrative topic object is defined to set the durable subscriptions attribute (DURSUB) of the soccer topic to NO:

```
DEFINE TOPIC(FOOTBALL.EUROPEAN)
       TOPICSTR('Sport/Soccer')
       DURSUB(NO)
       DESCR('Administrative topic object to disallow durable subscriptions')
```

The topic tree can now be visualized as:

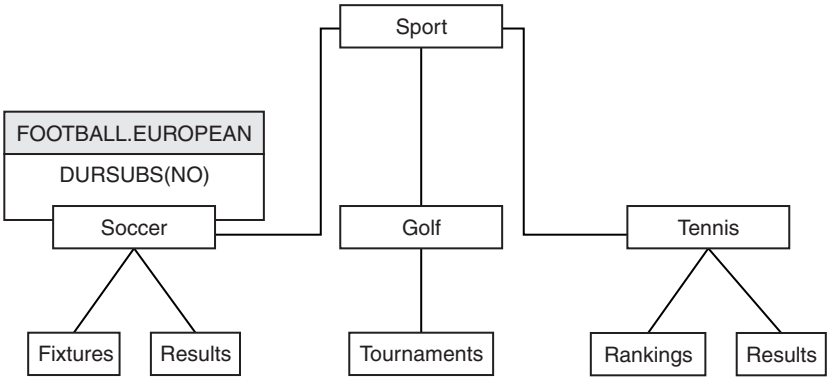


Figure 6. Visualization of an administrative topic object associated with the Sport/Soccer topic

Any applications subscribing to topics beneath Soccer in the tree can still use the topic strings they used before the administrative topic object was added. However, an application can now be written to subscribe using the object name `FOOTBALL.EUROPEAN`, instead of the string /Sport/Soccer. For example, to subscribe to /Sport/Soccer/Results, an application can specify `MQSD.ObjectName` as `FOOTBALL.EUROPEAN` and `MQSD.ObjectString` as `Results`.

This feature allows you to hide part of the topic tree from application developers. If you define an administrative topic object at a particular node in the topic tree, application developers can define their own topics below this, without needing to have knowledge of topics above the administrative topic object.

Inheriting attributes

If a topic tree has many administrative topic objects, each administrative topic object, by default, inherits its attributes from its closest parent administrative topic node. The previous example has been extended in Figure 7:

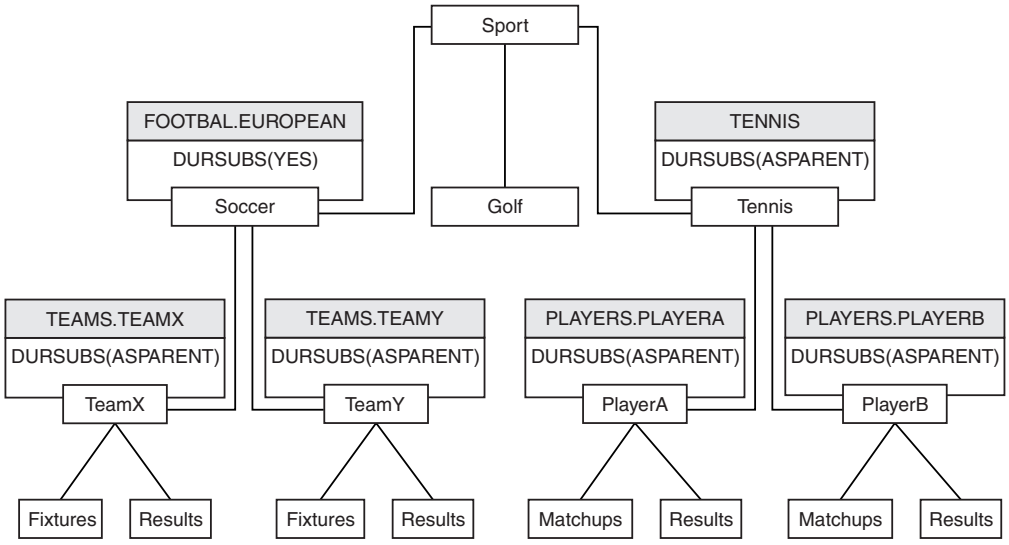


Figure 7. Topic tree with several administrative topic objects

If all topics at and below `/Sport/Soccer` need to have the attribute `DURSUB` set to `NO`, the only change that needs to be made is to alter the `DURSUB` attribute of `FOOTBALL.EUROPEAN` to `NO`.

This attribute can be set using the following command:

```
ALTER TOPIC(FOOTBALL.EUROPEAN) DURSUB(NO)
```

Because all the administrative topic objects below `Sport/Soccer` have the `DURSUB` attribute set to the default value `ASPARENT`, all topics below `Sport/Soccer` will inherit the value `NO` for their `DURSUB` attribute.

All the administrative topic objects at and below `Sport/Tennis` have the value `ASPARENT` for the attribute `DURSUB`. All topics at and below `Sport/Tennis`, therefore, will inherit `DURSUB` from the `SYSTEM.BASE.TOPIC` object and will have the value of `YES`.

Trying to make a durable subscription to the topic `Sport/Soccer/TeamX/Results` would now fail; however, trying to make a durable subscription to `Sport/Tennis/PlayerB/Results` would succeed.

SYSTEM.BASE.TOPIC

Base topic for ASPARENT resolution. If a particular topic has no parent administrative topic objects, or those parent objects also have ASPARENT, any remaining ASPARENT attributes are inherited from this object.

The default values of the SYSTEM.BASE.TOPIC are:

Table 2. Default values of SYSTEM.BASE.TOPIC

Parameter	Value
TOPICSTR	″
DEFPRTY	0
DEFPRESP	SYNC
DEFPSIST	NO
DESCR	'Base topic for resolving attributes'
DURSUB	YES
MDURMDL	SYSTEM.DURABLE.MODEL.QUEUE
MNDURMDL	SYSTEM.NDURABLE.MODEL.QUEUE
MASTER	YES
NPMSGDLV	ALLAVAIL
PMSGDLV	ALLDUR
PUB	ENABLE
SUB	ENABLE

If this object does not exist, its default values are still used by WebSphere MQ for ASPARENT attributes that are not resolved by parent topics further up the topic tree.

Chapter 3. Distributed publish/subscribe

This section discusses how publish/subscribe messaging can be performed between queue managers, and the 2 different queue manager topologies that can be used to connect queue managers, clusters and hierarchies.

Queue managers can communicate with other queue managers in your WebSphere MQ publish/subscribe system, so that subscribers can subscribe to one queue manager and receive messages that were initially published to another queue manager. This is illustrated in Figure 8.

Figure 8 shows a publish/subscribe system with two queue managers.

- Queue manager 2 is used by Publisher 4 to publish weather forecast information, using a topic of Weather, and information about traffic conditions on major roads, using a topic of Traffic.
- Subscriber 4 also uses this queue manager, and subscribes to information about traffic conditions using topic Traffic.
- Subscriber 3 also subscribes to information about weather conditions, even though it uses a different queue manager from the publisher. This is possible because the queue managers are linked to each other.

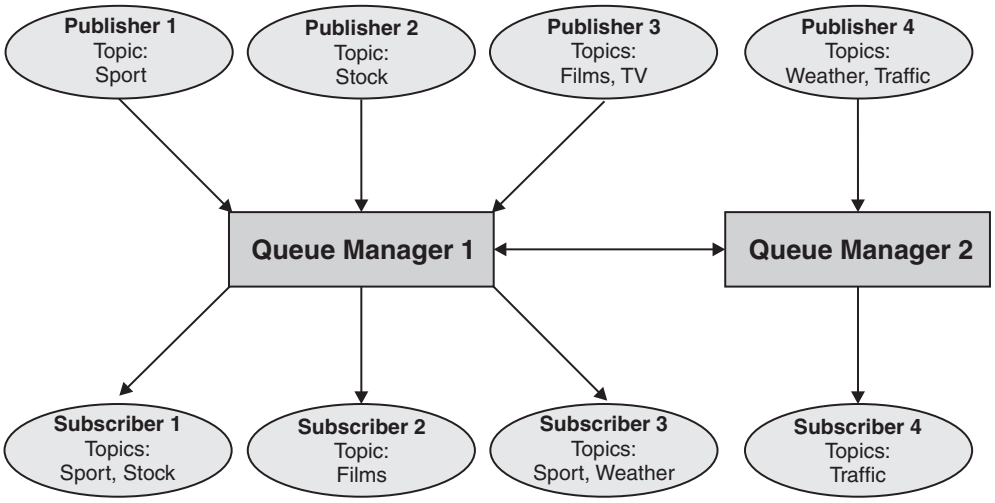


Figure 8. Publish/subscribe example with two queue managers

How does distributed publish/subscribe work?

WebSphere MQ publish/subscribe uses proxy subscriptions to ensure that subscribers can receive messages that are published to remote queue managers.

Distributed publish/subscribe uses the same components as distributed queuing to connect networks of queue managers and consequently, the applications that connect to those queue managers. To find out more about messaging between queue managers and the components involved making connections between queue managers see the *Intercommunication* documentation.

Subscribers need not do anything beyond the standard subscription operation in a distributed publish/subscribe system. When a subscription is made on a queue manager, the queue manager manages the process by which the subscription is propagated to connected queue managers. A subscriptions flows to all queue managers in the network, where proxy subscriptions are created to ensure that publications get routed back to the queue manager where the subscription was created originally. This is shown in Figure 9.

A publication is propagated to a remote queue manager only if a subscription to that topic exists on that remote queue manager.

A queue manager consolidates all the subscriptions that are created on it, whether from local applications or from remote queue managers. In turn, the queue manager creates subscriptions for these topics with its neighbors, unless a subscription already exists. This is shown in Figure 10 on page 25.

When an application publishes information, the receiving queue manager forwards it (possibly through one or more intermediate queue managers) using transmission queues to any applications that have valid subscriptions on remote queue managers. This is shown in Figure 11 on page 25.

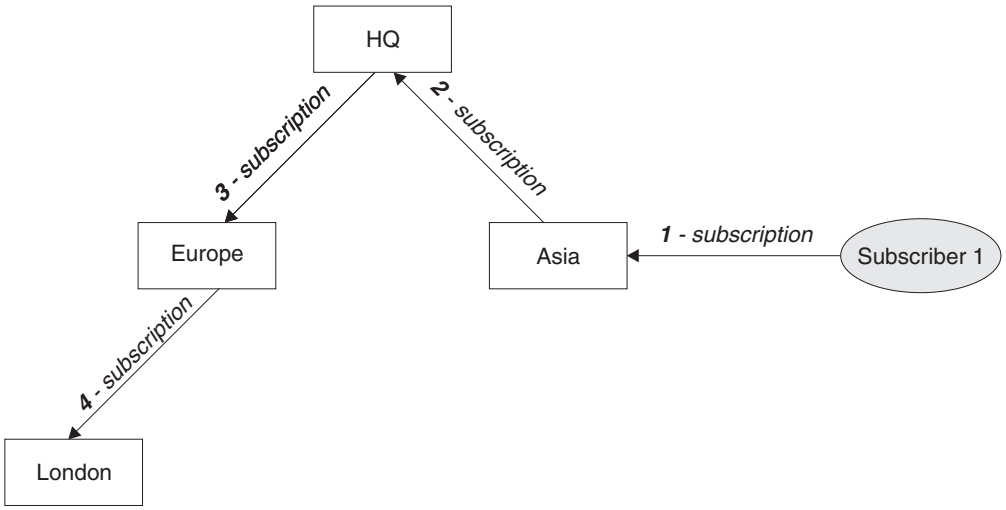


Figure 9. Propagation of subscriptions through a queue manager network. Subscriber 1 registers a subscription for a particular topic on the Asia queue manager (1). The subscription for this topic is forwarded to all other queue managers in the network (2,3,4).

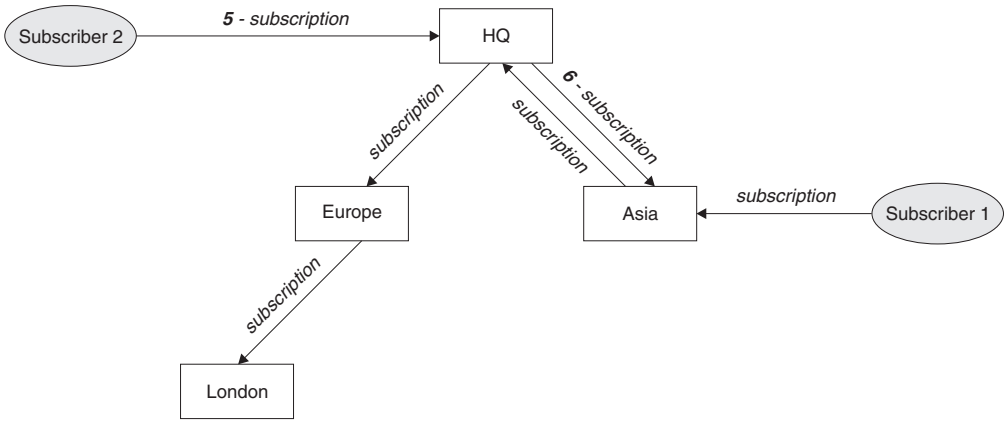


Figure 10. Multiple subscriptions. Subscriber 2 registers a subscription, to the same topic as in Figure 9 on page 24, on the HQ queue manager (5). The subscription for this topic is forwarded to the Asia queue manager, so that it is aware that subscriptions exist elsewhere on the network (6). The subscription does not have to be forwarded to the Europe queue manager, because a subscription for this topic has already been registered (step 3 in Figure 9 on page 24).

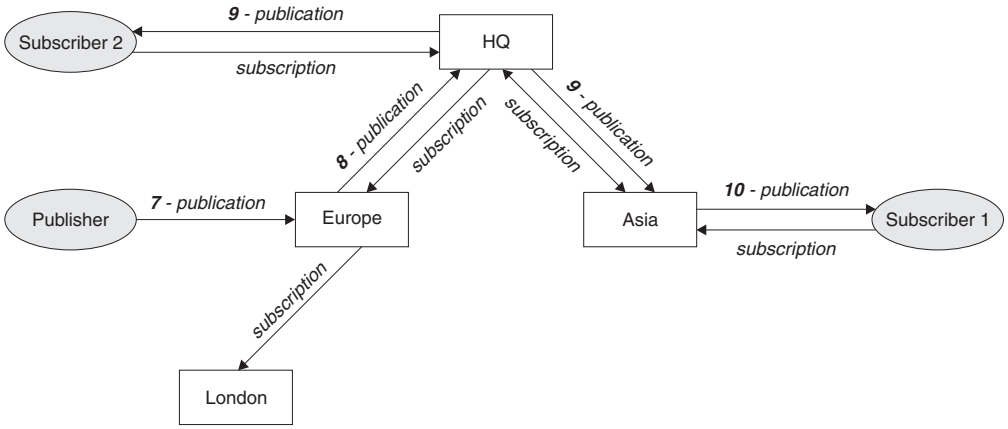


Figure 11. Propagation of publications through a queue manager network. A publisher sends a publication, on the same topic as in Figure 10, to the Europe queue manager (7). A subscription for this topic exists from HQ to Europe, so the publication is forwarded to the HQ queue manager (8). However, no subscription exists from London to Europe (only from Europe to London), so the publication is not forwarded to the London queue manager. The HQ queue manager sends the publication directly to subscriber 2 and to the Asia queue manager (9), from where it is forwarded to subscriber 1 (10).

When a queue manager sends any publications or subscriptions to another queue manager, it sets its own user ID in the message, and uses its own authority to put the message. This means that the queue manager must have the authority to put messages onto other queue managers' queues (unless the channel is set up to put incoming messages with the message channel agent's authority). This also means that all authorization checks are performed at the publisher's or subscriber's local queue manager.

The interconnected nature of publish/subscribe queue managers means that it takes some time for the proxy subscription to propagate around all nodes in the network. The consequence of this is that once a subscription has been made, remote publications are not necessarily received immediately; this can be addressed by using PROXYSUB(FORCE) as described in "More on routing mechanisms" on page 26.

The subscription operation completes when the proxy subscription has been put on the appropriate transmission queue for each directly connected queue manager, and will not include the propagation of the proxy subscription out to the rest of the topology. Proxy subscriptions are associated with the queue manager name that created them. If one queue manager is attached, by a hierarchical connection or as part of a publish/subscribe cluster, to more than one queue manager with the same queue manager name, this can result in publications failing to reach one or all of the identically named remote queue managers. To avoid this problem, as with point-to-point messaging, give queue managers unique names, especially if they are directly or indirectly connected in a WebSphere MQ network.

Within a distributed publish/subscribe network the flow of publications and subscriptions can be controlled, and if appropriate, restricted, using publication and subscription scope.

Proxy subscription aggregation and publication aggregation

Distributed publish/subscribe publications and proxy subscriptions are aggregated to minimize the quantity of messages passing between publish/subscribe queue managers.

Proxy subscription aggregation

Proxy subscriptions are aggregated using a simple duplicate elimination system. For a given resolved topic string, a proxy subscription is sent to directly connected publish/subscribe queue managers on the first local subscription or received proxy subscription.

Subsequent subscriptions make use of this existing proxy subscription. The proxy subscription is cancelled only after the last local subscription or received proxy subscription is cancelled.

Note: If PROXYSUB(FORCE) is set, a proxy subscription might be sent before the first local subscription or received proxy subscription, and will not be cancelled even after the last local subscription or received proxy subscription is cancelled.

Publication aggregation

It is possible for more than one proxy subscription to match the topic string of a single publication when the proxy subscriptions contain wildcards. If a message is published on a queue manager that matches two or more proxy subscriptions created by a single connected queue manager, only one copy of the publication is forwarded to the remote queue manager to satisfy the multiple proxy subscriptions.

More on routing mechanisms

Publish everywhere is an alternative routing mechanism to proxy subscription-forwarding. Publish everywhere works by publishing to all directly connected queue managers regardless of proxy subscriptions. Publish everywhere is not supported in publish/subscribe clusters or hierarchies, but a similar technique is available by using the PROXYSUB attribute for a high-level topic object.

PROXYSUB attribute for a high-level topic object is explained in the following comparison:

Publish everywhere

If publish everywhere routing is available in a publish/subscribe cluster,

there is no need for any proxy subscriptions and all publications are published to every member of the publish/subscribe clusters.

The advantages of publish everywhere are the removal of latency introduced by the propagation of proxy subscriptions, and the removal of the network overhead caused by proxy subscription propagation where the subscription is frequently created and deleted.

Proxy-subscription forwarding

To achieve a similar behavior to publish everywhere, alter the topic object, as follows:

```
ALTER TOPIC("SYSTEM.BASE.TOPIC") PROXYSUB(FORCE)
```

This forces the sending of a wildcard proxy subscription, for the topic string associated with this topic object, to every directly connected member of the publish/subscribe topology, regardless of whether any local subscriptions have been made.

When this forced proxy subscription has been propagated throughout the topology, any new subscriptions immediately receive any publications from other connected queue manager, without suffering latency. Proxy subscriptions for these new subscriptions are still propagated to each of the directly connected publish/subscribe queue managers; preventing a break in flow of publications if this behavior is turned off later.

Wildcard rules

Wildcards in proxy subscriptions are converted to use topic wildcards.

When a subscription for a wildcard is received, it can be either a character, as used by WebSphere MQ Version 6.0, or a topic, as used by WebSphere Message Broker Version 6.0 and WebSphere MQ Version 7.0 as follows:

- Character wildcards use '*' to represent any character (including '/').
- Topic wildcards use '#' to represent a portion of the topic space between '/' characters.

In WebSphere MQ Version 7.0, all proxy subscriptions are converted to use topic wildcards. To achieve this, if a character wildcard is found, it is replaced with a '#' character, back to the nearest '/'. For example, '/aaa/bbb/c*d' is converted to '/aaa/bbb/#'. This results in remote queue managers sending slightly more publications than were explicitly subscribed to, but these are filtered out by the local queue manager as it delivers the publications to its local subscribers.

Controlling the flow of publications and subscriptions

Scope is separated into publication and subscription scope so that queue managers can pass publications into, but not out of the publish/subscribe cluster, or out of, but not into the publish/subscribe cluster.

Publication scope

The scope of a publication controls whether queue managers distribute the publication to remote subscribers.

The PUBSCOPE topic attribute can be used to determine the scope of publications made to a specific topic. You can set the attribute to one of the following values:

QMGR
The publication is delivered only to local subscribers. These publications are called *local publications*. Local publications are not forwarded to remote queue managers and therefore are not received by remote queue managers' subscribers.

ALL The publication is delivered to local subscribers and remote subscribers through directly connected queue managers. These publications are called *global publications*.

Publishers can also specify whether a publication is local or global using the MQPMO_SCOPE_QMGR put message option, if this option is used, it overrides any behavior that has been set using the PUBSCOPE topic attribute.

Subscription scope

The scope of a subscription controls whether a subscription receives publications made on remote queue managers. You use the SUBSCOPE topic attribute to administer the scope of subscriptions.

Subscribers can decide to receive only local publications using the MQSO_SCOPE_QMGR subscription option. The MQSO_SCOPE_QMGR option determines whether a proxy subscription is created on remote queue managers in the network so that they are aware of the subscription and route publications to the local queue manager. If this option is not used, the subscriber will receive both local and global publications.

You can set the attribute to one of the following values:

QMGR
The subscription is not propagated to directly connected queue managers, and receives publications only from local publishers.

ALL The subscription is propagated to directly connected queue managers, and receives publications from local publishers and remote publishers through directly connected queue managers.

Overlapping topics

The scope of publications and subscriptions is defined in both local topic objects, as shown in the following information, and cluster topic objects.

For the following local topic definitions, a local application that subscribes using topic string '/football/#' will not receive remote publications on 'football/myteam':

```
DEFINE TOPIC(A) TOPICSTR('/football') SUBSCOPE(ALL)
DEFINE TOPIC(B) TOPICSTR('/football/myteam') SUBSCOPE(QMGR)
```

Note: Subscribers can restrict SUBSCOPE, so that remote publications are not received, by using MQSO_SCOPE_QMGR.

Retained publications

It is not good practice for two or more applications to publish retained publications to the same topic on the same or different queue managers within a single publish/subscribe topology.

It is possible that different retained publications could be active at different queue managers for the same topic, leading to unexpected behavior. As multiple proxy subscriptions are distributed, multiple retained publications could be received.

Distributed publish/subscribe security

Distributed publish/subscribe internal messages such as proxy subscriptions and publications are put to distributed publish/subscribe system queues (SYSTEM.INTER.QMGR.CONTROL, for example) by the receiving channel using normal channel security rules. The information and diagrams in this topic highlight the various processes and user IDs involved in the delivery of these messages.

Local access control

Access to topics for publication and subscriptions is governed by local security definitions and rules that are described in Topic objects authorization. On z/OS, no local topic object is required to establish access control. This is also true on distributed systems, so administrators can choose to apply access control to clustered topic objects irrespective of whether they exist in the cluster yet.

System administrators are responsible for access control on their local system and trust other members of the hierarchy or cluster collectives to which they are attached to be responsible for their own access control policy. It might not be necessary to impose any access control, or access control can be defined on high level objects in the topic tree, or fine level access control can be defined for each subdivision of the topic name space. Because access control is defined for each separate machine it is likely to be burdensome if fine level control is needed.

Making a proxy subscription

Trust for an organization to connect its queue manager to your queue manager is confirmed by normal channel authentication means. If that trusted organization is then allowed to do distributed publish/subscribe, an authority check is done when the channel puts the message to a distributed publish/subscribe queue; for example, SYSTEM.INTER.QMGR.CONTROL. The user ID for the queue authority check depends on the PUTAUT value of the receiving channel (for example, the user ID of the channel, MCAUSER, message context, and so on, depending on value and platform). For more information on channel security, see WebSphere MQ Security.

Proxy subscriptions will be made with the user ID of the distributed publish/subscribe agent on the remote queue manager (QM2 in Figure 12 on page 30) which can then easily be granted access to local topic object profiles, because that user ID is defined in the system and there are therefore no domain conflicts.

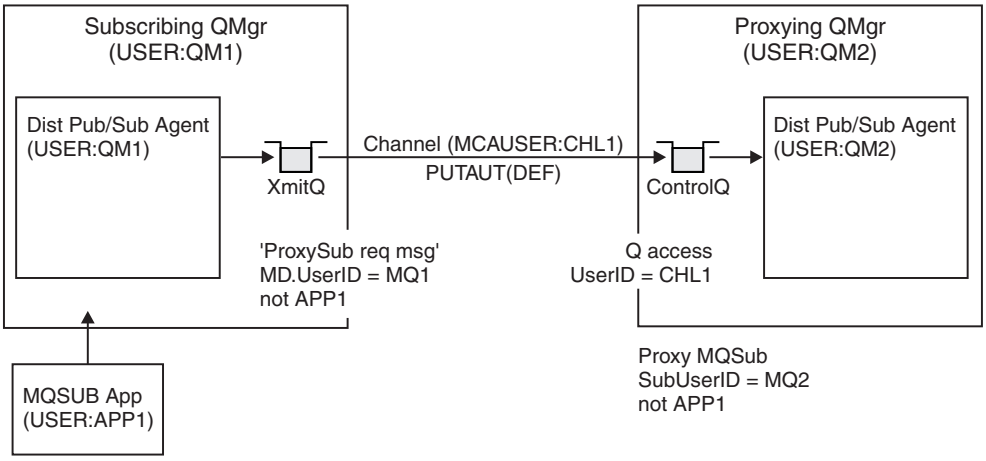


Figure 12. Proxy subscription security, making a subscription

Sending back remote publications

When a publication is made on the publishing queue manager, a copy satisfies the proxy subscription that was made, and the context of that message contains the context of the user ID which made the subscription, QM2 in Figure 13 on page 31. The proxy subscription is made with a destination queue that is a remote queue, so the publication message is resolved onto a transmission queue.

Again, trust for an organization to connect its queue manager, QM2, to another queue manager, QM1, is confirmed by normal channel authentication means. If that trusted organization is then allowed to do distributed publish/subscribe, an authority check is done when the channel puts the publication message to the distributed publish/subscribe publication queue SYSTEM.INTER.QMGR.PUBS. The user ID for the queue authority check depends on the PUTAUT value of the receiving channel (for example, the user ID of the channel, MCAUSER, message context, and so on, depending on value and platform). For more information on channel security, see WebSphere MQ Security.

When the publication message reaches the subscribing queue manager, another MQPUT to the topic is done under the authority of that queue manager and the context with the message is replaced by the context of each of the local subscribers as they are each given the message.

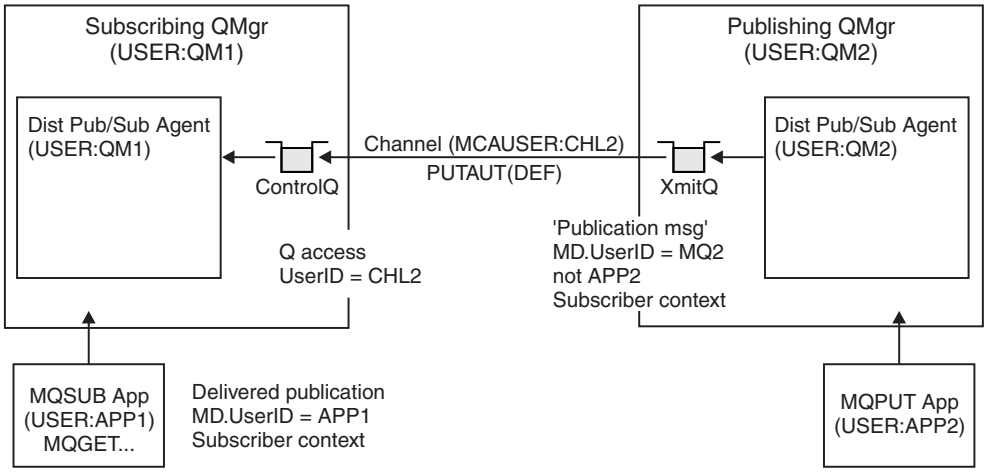


Figure 13. Proxy subscription security, forwarding publications

This means that, on a system where little has been considered regarding security, the distributed publish/subscribe processes are likely to be running under a user ID in the mqm group, the MCAUSER parameter on a channel will be blank (the default), and messages are delivered to the various system queues as required. This makes it easy to set up a proof of concept to demonstrate distributed publish/subscribe.

On a system where security is more seriously considered, these internal messages are subject to the same security controls as any message going over the channel.

If the channel is set up with a non-blank MCAUSER and a PUTAUT value specifying that MCAUSER should be checked, then the MCAUSER in question must be granted access to SYSTEM.INTER.QMGR.* queues. Where there are multiple different remote queue managers with channels running under different MCAUSER ids (for instance, when multiple hierarchical connections are configured on a single queue manager), then all those user IDs need to be granted access to the SYSTEM.INTER.QMGR.* queues.

If the channel is set up with a PUTAUT value specifying that the context of the message is used, then access to the SYSTEM.INTER.QMGR.* queues are checked based on the user ID inside the internal message. Because all these messages are put by the distributed publish/subscribe agent's user ID from the queue manager that is sending the internal message, or publication message (see Figure 13), it is not too large a set of user IDs to grant access to the various system queues (one per remote queue manager), should you want to set up your distributed publish/subscribe security in this way. It still has all of the same issues that channel context security always has; that of the different user ID domains and the fact that the user ID in the message might not be defined on the receiving system. However, it is a perfectly acceptable way to run if required.

System queue security in the *WebSphere MQ z/OS System Setup Guide* provides a list of queues and the access that is required to securely set up your distributed publish/subscribe environment. If any internal messages or publications fail to be put due to security violations, the channel writes a message to the log in the normal manner and the messages can be sent to the dead-letter queue according to normal channel error processing.

All inter-queue manager messaging for the purposes of distributed publish/subscribe runs using normal channel security. No special casing is required in the security manager on behalf of the distributed publish/subscribe component.

For information on restricting publications and proxy subscriptions at the topic level, see Topic objects authorization.

Using default user IDs with a queue manager hierarchy

If you have a hierarchy of queue managers running on different platforms and are using default user IDs, note that these default user IDs differ between platforms and might not be known on the target platform. As a result, a queue manager running on one platform rejects messages received from queue managers on other platforms with the reason code MQRC_NOT_AUTHORIZED.

To avoid this, grant user access to the queues and topic objects (as a minimum, that is: SYSTEM.BROKER.DEFAULT.STREAM and SYSTEM.BROKER.ADMIN.STREAM), to the default user IDs used on other platforms in your publish/subscribe hierarchy.

The default user IDs are as follows:

Windows	MUSR_MQADMIN
UNIX systems	mqm
i5/OS	QMQM
z/OS	The channel initiator address space user ID

User IDs can be case sensitive. The originating queue manager (if Windows, UNIX or i5/OS) will force the user ID to be all uppercase. The receiving queue manager (if Windows or UNIX) will force the user ID to be all lowercase. Therefore, all user IDs created on UNIX will need to be created in their lowercase form. However, if a message exit has been installed, the forcing to uppercase or lowercase will not take place and care must be taken to understand how the message exit will process the user ID.

- On UNIX and Windows ensure the user IDs are specified in lowercase.
- On i5/OS and z/OS ensure the user IDs are specified in uppercase.

Distributed publish/subscribe system queues

Four system queues are used by queue managers when they do publish/subscribe messaging. You normally need to be aware of their existence only for problem determination or capacity planning purposes.

Table 3. Publish/subscribe system queues

System queue	Purpose
SYSTEM.INTER.QMGR.CONTROL	WebSphere MQ distributed publish/subscribe control queue
SYSTEM.INTER.QMGR.FANREQ	WebSphere MQ distributed publish/subscribe internal proxy subscription fan-out process input queue
SYSTEM.INTER.QMGR.PUBS	WebSphere MQ distributed publish/subscribe publications

Table 3. Publish/subscribe system queues (continued)

System queue	Purpose
SYSTEM.HIERARCHY.STATE	WebSphere MQ distributed publish/subscribe hierarchy relationship state

On z/OS, you set up the necessary system objects when you create the queue manager, by including the CSQINS4R and CSQINS4G samples in the CSQINP2 initialization input data set. For more information, see the *WebSphere MQ for z/OS System Setup Guide*.

The attributes of the distributed publish/subscribe system queues are as displayed in Table 4.

Table 4. Attributes of publish/subscribe system queues

Attribute	Value
DEFPSIST	Yes
DEFSOPT	This takes the value EXCL.
MAXMSGL	On AIX®, HP-UX, Linux®, i5/OS®, Solaris and Windows® this takes the value of MAXMSGL parameter of the ALTER QMGR command. On z/OS this takes the value 100 MB (104 857 600 bytes).
MAXDEPTH	On AIX, HP-UX, Linux, i5/OS, Solaris, Windows and z/OS this takes the value 999 999 999.
SHARE	This is a keyword that specifies that the queue can be shared for GET.
STGCLASS	On z/OS this takes the value 'SYSTEM'. On other platforms this attribute is not used.

Publish/subscribe system queue errors

Errors can occur when distributed publish/subscribe queue manager queues are unavailable.

If the fan-out request queue SYSTEM.INTER.QMGR.FANREQ is unavailable, the MQSUB API receives reason codes and error messages written to the error log, on occasions where proxy subscriptions need to be delivered to directly connected queue managers.

If the hierarchy relationship state queue SYSTEM.HIERARCHY.STATE is unavailable, an error message is written to the error log and the publish/subscribe engine is put into COMPAT mode.

If any other of the SYSTEM.INTER.QMGR queues are unavailable, an error message is written to the error log, and although function is not disabled, it is likely that publish/subscribe messages will build up on queues on remote queue managers.

If the transmission queue to a parent, child or publish/subscribe cluster queue manager is unavailable:

1. The MQPUT API receives reason codes and the publications are not delivered.

2. Received inter-queue manager publications are backed out to the input queue, and subsequently re-attempted, being placed on the dead letter queue if the backout threshold is reached.
3. Proxy subscriptions are backed out to the fanout request queue, and subsequently attempted again, being placed on the dead letter queue if the backout threshold is reached; in which case the proxy subscription will not be delivered to any connected queue manager.
4. Hierarchy relationship protocol messages fail, and the connection status is marked as ERROR on the PUBSUB command.

Publish/subscribe topologies

A *publish/subscribe topology* consists of queue managers and the connections between them, that support publish/subscribe applications.

A publish/subscribe application can consist of a network of queue managers connected together. The queue managers can all be on the same physical system, or they can be distributed over several physical systems. By connecting queue managers together, publications can be received by an application using any queue manager in the network.

This provides the following benefits:

- Client applications can communicate with a nearby queue manager rather than with a distant queue manager, thereby getting better response times.
- By using more than one queue manager, more subscribers can be supported.

You can arrange queue managers that are doing publish/subscribe messaging in two different ways, clusters and hierarchies. For more information about these two topologies and to find out which is most appropriate for you, refer to the information in this chapter.

It is possible to use both topologies in combination by joining clusters together in a hierarchy.

Publish/subscribe clusters

You can improve the performance of your publish/subscribe network by arranging your queue managers in a publish/subscribe cluster. A publish/subscribe cluster consists of a set of queue managers connected together, with direct channel links between all members, to form all or part of a publish/subscribe network.

A *publish/subscribe cluster* is a set of queue managers that are fully interconnected and form part of a multi-queue manager network for publish/subscribe applications. A cluster that is used for publish/subscribe messaging is no different from a standard WebSphere MQ cluster. As such, the queue managers within the publish/subscribe cluster can exist on physically separate computers and each pair of queue managers is connected together by a pair of channels. For information about how to plan and configure a WebSphere MQ cluster refer to *WebSphere MQ Queue Manager Clusters*.

Using clusters in a publish/subscribe topology provides the following benefits:

- Messages destined for a specific queue manager in the same cluster are transported directly to that queue manager and do not need to pass through an

intermediate queue manager. This improves performance and optimizes inter-queue manager publish/subscribe traffic, in comparison with a hierarchical topology.

- There is no single point of failure in this topology. If one queue manager is not available, publications and subscriptions are still able to flow through the rest of the publish/subscribe system because each queue manager is directly connected with each other.
- If your clients are geographically dispersed, you can set up a cluster in each location, and connect the clusters (by joining a single queue manager in each cluster) to optimize the flow of publications and subscriptions through the network.
- You can group clients according to the topics to which they publish and subscribe.

  Clients that share common topics can connect to queue managers within a cluster. The common publications are transported efficiently within the cluster, because they pass through only queue managers that have at least one client with an interest in those common topics.
- A subscribing application can connect to its nearest queue manager, to improve its own performance. The queue manager receives all messages that match the subscription registration of the client from all queue managers within the cluster.

  The performance of a client application is also improved for other services that are requested from this queue manager. A client application can use both publish/subscribe and point-to-point messaging.
- The number of clients per queue manager can be reduced by adding more queue manager to the cluster to share workload. This makes a publish/subscribe cluster topology highly scalable.

When you create a cluster it is possible to create a loop causing messages to cycle forever within the network, nothing will prevent you from doing this but you will be made aware of it because of the fingerprint that is added by the queue manager (stored as a message property).

A publish/subscribe cluster is created when a clustered topic is defined. This definition is shared with all members of the cluster. This means that publications on the clustered topic are shared with all members of the cluster.

When at least one clustered topic object is defined, all queue managers within the cluster will be notified about each other.

If you have several queue managers in your publish/subscribe system, many channels are required to connect these queue managers together. However, the connections between queue managers can be created automatically to reduce the administrative work load.

Cluster topics

You can cluster topics in a similar manner to cluster queues, although an individual administrative topic object can be a member of only one cluster. Topic objects do not have an equivalent to the CLUSNL (cluster namelist) attribute.

When a cluster topic is defined, the cluster topic object is published to the full repositories. The full repositories then push all cluster topic definitions to all queue managers within the cluster.

At each queue manager a single topic space is constructed from the local and cluster topic definitions. When an application subscribes to a topic that resolves to a clustered topic, WebSphere MQ creates a proxy subscription and sends it, from the queue manager to which the subscriber connected, to all members of the cluster in which the clustered topic object is defined.

If a local and cluster topic definition exists for a single topic string, the local definition is used. Where two or more cluster topic definitions, for a single topic string, have differing attributes or exist in more than one cluster, a message is written to the log and the most recently received cluster topic definition is used. It is acceptable to define two or more cluster topic definitions with identical attributes for a single topic string.

If you are working in clusters, and a single queue manager defines a local topic object to override the behavior of a cluster topic object, this does not prevent other queue managers in the cluster from sending proxy subscriptions to the queue manager that defined the local topic object. To prevent publications being sent to those proxy subscriptions, you need to specify PUBSCOPE(QMGR) on the local topic object.

If the queue manager on which a cluster topic is defined is unavailable, you cannot alter the cluster topic definition remotely. However, you can use the RESET CLUSTER command to remove the queue manager from the cluster. You can define an additional cluster topic definition on the same topic string at a different queue manager within the cluster; if defined with differing attributes, this overrides the previous definition and a message is written to the log. If the original queue manager subsequently becomes available, its clustered topic object must either be deleted or its definition updated to match the additional cluster definition.

Cluster topic performance

The performance characteristics of cluster topics requires special consideration as it differs from cluster queues, and is potentially a source of performance problems in large or unbalanced clusters.

An important concept in asynchronous messaging performance is *balance*. Unless message consumers are balanced with message producers, there is the danger that a backlog of unconsumed messages might build up and seriously affect the performance of multiple systems. In a point-to-point messaging topology the relationship between message consumers and message producers is readily understood, and estimates of message production and consumption made, queue manager by queue manager, channel by channel. If there is a lack of balance, the bottlenecks are readily identified and then remedied.

In a clustered topology an additional consideration is the overhead of managing clustering. When a new clustered queue is defined, the destination information is pushed to the full repositories, and only sent to other cluster members when they first reference a clustered queue. Adding a smaller queue manager server to a cluster does not necessarily unbalance the performance of cluster management: the load on the new queue manager is not directly related to the size of the cluster or the other queue managers in the cluster.

Publish/subscribe topologies present performance challenges in both these respects.

1. It is harder to work out whether publishers and subscribers are balanced. Start from each subscription that resolves to a clustered topic, and work back to the queue managers having publishers on the topic. Calculate the number of publications flowing to each subscriber.
2. Publish/subscribe cluster management has much more overhead than point-to-point cluster management:
   a. When a new queue manager joins an existing cluster, it re-synchronizes with all members of the clusters, possibly causing channels to be started to each member of the cluster from the new queue manager.
   b. When a new subscription is created to a clustered topic, proxy subscriptions are sent to all members of the cluster, possibly causing channels to be started to each member of the cluster from the subscribing queue manager.
   c. When a new cluster topic is defined, it is first pushed to the full repositories. It is then immediately pushed to all queue managers in the cluster, possibly causing channels to be started to each member of the cluster from the repository.

In short, the cluster management load on any queue manager in the cluster is proportional to the size of the cluster, and not to the size of the queue manager.

To reduce the impact of publish/subscribe cluster management on the performance of a cluster consider the following two suggestions:

1. Perform cluster, topic and subscription updates at off-peak times of the day.
2. If you are thinking about adding publish/subscribe topics to an existing large cluster, just because the cluster is already there, consider if you can define a much smaller subset of queue managers involved in publish/subscribe and make that an "overlapping" cluster. Although some queue managers are now in two clusters, the overall impact of publish/subscribe is reduced:
   a. The size of the publish/subscribe cluster is smaller.
   b. Queue managers not in the publish/subscribe cluster are much less affected by the impact of management traffic in the publish/subscribe cluster.

Cluster topic names

Cluster topic names are character strings. For example, you could have high-level cluster topics named 'Sport', 'Stock', 'Films', and 'TV', and you could divide the 'Sport' cluster topic into separate, more specific cluster topics covering different sports:

```
Sport/Soccer Sport/Golf Sport/Tennis
```

These cluster topics could then be divided further, to separate different types of information about each sport:

```
Sport/Soccer/Fixtures Sport/Soccer/Results Sport/Soccer/Reports
```

WebSphere MQ publish/subscribe does not recognize that the forward slash (/) character is being used in a special way, but if you use the forward slash (/) character as a separator, you can ensure compatibility with other WebSphere business integration applications.

You can use any character in the single-byte character set for which the machine is configured in a character string. Consider, however, whether the cluster topic string might need to be translated to a different character representation, in which case you must use only those characters that are available in the configured character set of all relevant machines.

Cluster topic strings are case sensitive, and a blank character has no special meaning. As a subscriber, you can specify a cluster topic or range of cluster topics using wildcards to receive the information in which you are interested.

Key roles for publish/subscribe cluster queue managers

There are two key roles for queue managers in publish/subscribe clusters that you should consider when designing a publish/subscribe cluster.

Full repositories
You can define full repositories on any queue manager in the publish/subscribe cluster. As with a normal cluster, a publish/subscribe cluster should ideally have two full repositories, hosted in highly available machines.

Cluster topic host
A cluster topic host is a queue manager where a clustered topic object is defined. You can define clustered topic objects on any queue manager in the publish/subscribe cluster. When at least one clustered topic exists within a cluster, the cluster is a publish/subscribe cluster. Ideally, all clustered topic objects should be identically defined on two queue managers and these machines should be highly available.

If a single host of a clustered topic object is lost, for example, because of disk failure, any cluster topic cache records that are based on the clustered topic object, that already exist in the cluster cache on other queue managers, are usable within the cluster for a period of up to 30 days, or until the cache is refreshed.

You can redefine the clustered topic object on a queue manager that is working correctly. If a new object is not defined within 27 days (inclusive) after the host queue manager failure, all members of the cluster will report that an expected object update has not been received.

Full repositories and topic hosts do not need to overlap or be separated. In publish/subscribe clusters that have just two highly available computers among many computers, it is good practice to define both the highly available computers as full repositories and cluster topic hosts.

In publish/subscribe clusters with many highly available computers it is good practice to define full repositories and cluster topic hosts on separate highly available computers, so that the operation and maintenance of one function can be managed without affecting the operation of other functions.

Overlapping cluster support and publish/subscribe

With WebSphere MQ clusters, a single queue manager can be a member of more than one cluster.

A reason for making a single queue manager a member of more than one cluster is to create a cluster gateway between two clusters, so that messages originating in one cluster can be routed to another cluster. Although a WebSphere MQ queue manager can be a member of more than one cluster and more than one publish/subscribe cluster, publications are not passed from one cluster to another by means of overlapping clusters. The scope of proxy subscriptions is limited to the single cluster in which the clustered topic is defined.

In Figure 14 on page 39, an application connected to queue manager QM3, subscribing on a topic that resolves to topic object T_B (which exists only in CLUSTER 1) results in proxy subscriptions being sent from queue manager QM3 to

both queue managers QM1 and QM2. An application connected to queue manager QM3, subscribing on a topic that resolved to topic object T_C (which exists only in the CLUSTER 2) results in proxy subscriptions being sent from queue manager QM3 to both queue managers QM4 and QM5.

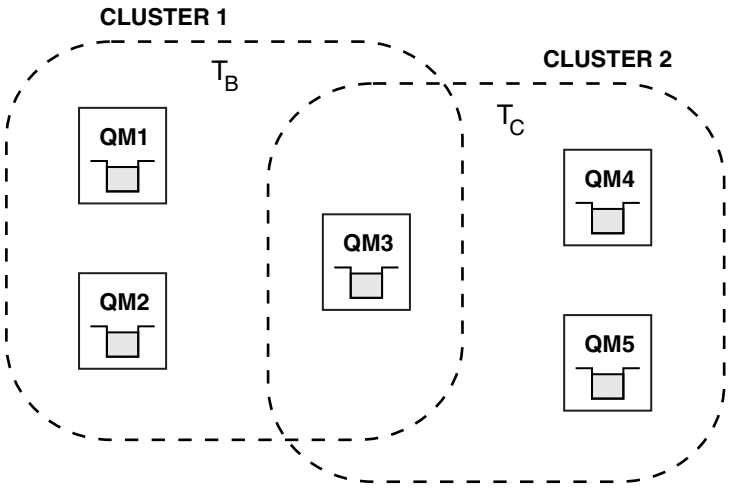


Figure 14. Overlapping clusters: two clusters each subscribing to different topics

In Figure 15 messages are output to the log of queue manager QM3 informing users that the topic object T_A exists in two clusters. An application connected to queue manager QM3, subscribing on a topic that resolved to topic object T_A (which exists in both CLUSTER 1 and CLUSTER 2) results in proxy subscriptions being sent to one cluster only – so either to queue managers QM1 and QM2 or to queue managers QM4 and QM5. The cluster chosen depends on which cluster topic object was added last to the cluster cache in queue manager QM3.

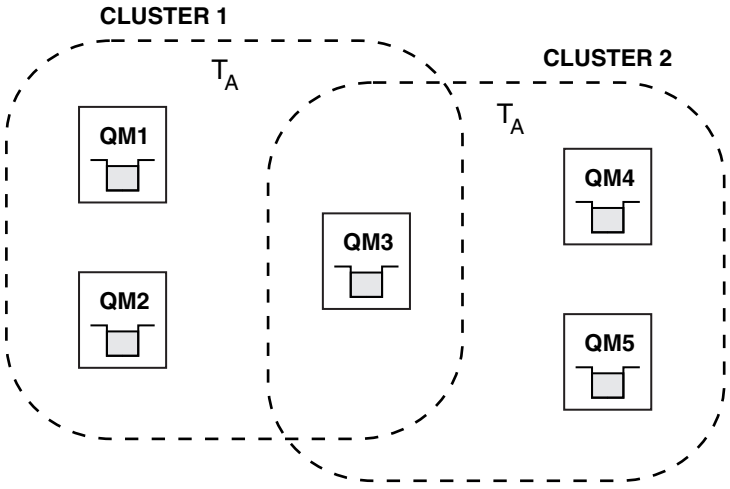


Figure 15. Overlapping clusters: two clusters each subscribing to the same topic

Publish/subscribe messages, for example, application publications and proxy subscriptions, are transmitted only over cluster channels that are part of the publish/subscribe cluster in which the cluster topic that the message relates to is defined.

For example, for the following topic definitions:

- Topic: T_A in CLUSTER 1 with TopicString: /football
- Topic: T_A in CLUSTER 2 with TopicString: /tennis

A subscription for T_A made on queue manager QM3 resolves to TopicString /tennis, assuming that this was the latest definition to be made, and causes receipt of publications on CLUSTER 2 for topic /tennis.

If any queue manager receives multiple definitions on the same topic string which differ in any detail, including cluster name, the behavior of publications and subscriptions on those topics or topic string is undefined. An informational message is issued to alert the administrator to the duplicate definition.

Subscription scope and publication scope in publish/subscribe clusters

The scope of publications and subscriptions is defined in the cluster topic object.

If a cluster topic object is defined with SUBSCOPE(QMGR), the definition is shared with the cluster, but the scope of subscriptions based on that topic is local only and publications are not received from the cluster.

If a cluster topic object is defined with PUBSCOPE(QMGR), the definition is shared with the cluster, but the scope of publications based on that topic is local only and they are not sent to other queue managers in the cluster.

These two attributes are commonly used together to isolate a queue manager from interacting with other members of the cluster on particular topics. The queue manager neither publishes or receives publications on those topics to and from other members of the cluster. Note that this does not prevent publication or subscription if topic objects are defined on subtopics.

REFRESH CLUSTER considerations

The REFRESH CLUSTER command can cause temporary disruption to publish/subscribe traffic in a publish/subscribe cluster. We therefore recommend only to run REFRESH CLUSTER command when under the guidance of your IBM Support Center.

The disruption can occur as follows:

- Up to 10 second pauses in message delivery.
- MQOPEN and MQPUT failures, for example, MQRC_NO_DESTINATIONS_AVAILABLE.

Publish/subscribe hierarchies

Queue managers can be grouped together in a hierarchy, where the hierarchy contains one or more queue managers that are directly connected. Queue managers are connected together using a connection-time parent and child relationship. When two queue managers are connected together for the first time, the child queue manager is connected to the parent queue manager.

When the parent and child queue managers are connected in a hierarchy there is no functional difference between them until you disconnect queue managers from the hierarchy.

Note: WebSphere MQ hierarchical connections require that the queue manager attribute PSMODE is set to ENABLED.

Connect a queue manager to a broker hierarchy

You can connect a local queue manager to a parent queue manager to modify a broker hierarchy.

Before you begin

1. You need to enable queued publish/subscribe mode. See "Starting queued publish/subscribe" on page 119.
2. The change is propagated to the parent queue manager using a WebSphere MQ connection. There are two ways to establish the connection.
   a. Connect the queue managers to a WebSphere MQ cluster.
   b. Establish a point-to-point channel connection using a transmission queue, or queue manager alias, with the same name as the parent queue manager.

      For example, suppose you are connecting to a queue manager called *PARENT*. Define a queue manager alias for *PARENT* that resolves to the transmission queue to parent. To place messages for *PARENT* on the transmission queue *PARENT.XMITQ*, use the following MQSC command to define the queue manager alias.

      ```
      DEFINE QREMOTE (PARENT) RNAME('') RQMNAME(PARENT) XMITQ(PARENT.XMITQ)
      ```

About this task

In WebSphere MQ Version 6.0, when the appropriate channels and queues are defined, brokers connect to one another as defined by parameters provided on the strmqbrk command.

The strmqbrk command works differently in WebSphere MQ Version 7.0 and you can no longer use it to connect children to parents. Instead you use the `ALTER QMGR PARENT (PARENT)` runmqsc command.

In WebSphere MQ Version 7, distributed publish/subscribe is typically implemented by using queue manager clusters and clustered topic definitions. For interoperability with WebSphere MQ Version 6 and WebSphere Message Broker V6.1 and WebSphere Event Broker V6.1 and earlier, you can also connect version 7 queue managers to a broker hierarchy as long as queued publish/subscribe mode is enabled.

ALTER QMGR PARENT(*PARENT*)

Example

The first example shows how to attach QM2 as a child of QM1, and then querying QM2 for its connection.

```
C:>runmqsc QM2
5724-H72 (C) Copyright IBM Corp. 1994, 2008.  ALL RIGHTS RESERVED.
Starting MQSC for queue manager QM2
alter qmgr parent(QM1)
     1 : alter qmgr parent(QM1)
AMQ8005: WebSphere MQ queue manager changed.
```

```
display pubsub type(All)
    14 : display pubsub type(All)
AMQ8723: Display pub/sub status details.
   QMNAME(QM2)                             TYPE(LOCAL)
AMQ8723: Display pub/sub status details.
   QMNAME(QM1)                             TYPE(PARENT)
```

The next example shows the result of querying QM1 for its connections

```
C:\Documents and Settings\Admin>runmqsc QM1
5724-H72 (C) Copyright IBM Corp. 1994, 2008.  ALL RIGHTS RESERVED.
Starting MQSC for queue manager QM1.
display pubsub type(all)
     1 : display pubsub type(all)
AMQ8723: Display pub/sub status details.
   QMNAME(QM1)                             TYPE(LOCAL)
AMQ8723: Display pub/sub status details.
   QMNAME(QM2)                             TYPE(CHILD)
```

What to do next

You can define topics on one broker or queue manager that are available to publishers and subscribers on the connected queue managers.

Disconnect a queue manager from a broker hierarchy

Disconnect a child queue manager from a parent queue manager in a broker hierarchy.

About this task

In WebSphere MQ Version 6.0, queue managers were disconnected from one another using the dltmqbrk command, and required that all child queue managers were disconnected first. In WebSphere MQ Version 7, the dltmqbrk command is used to discard WebSphere MQ Version 6 broker resources after migration to version 7 using the strmqbrk command.

You disconnect a version 7 queue manager from a broker hierarchy using the ALTER QMGR command. Unlike version 6, you can disconnect version 7 queue managers in any order and at any time.

The corresponding request to update the parent is sent when the connection between the queue managers is running.

```
ALTER QMGR PARENT(' ')
```

Example

```
C:\Documents and Settings\Admin>runmqsc QM2
5724-H72 (C) Copyright IBM Corp. 1994, 2008.  ALL RIGHTS RESERVED.
Starting MQSC for queue manager QM2.
    1 : alter qmgr parent('')
AMQ8005: WebSphere MQ queue manager changed.
    2 : display pubsub type(child)
AMQ8147: WebSphere MQ object  not found.
display pubsub type(parent)
    3 : display pubsub type(parent)
AMQ8147: WebSphere MQ object  not found.
```

What to do next

You can delete any streams, queues and manually defined channels that are no longer needed.

Chapter 4. Writing publish/subscribe applications

Start writing publish/subscribe WebSphere MQ applications. We assume you have already written point to point WebSphere MQ applications before.

Writing publisher applications

Get started with writing publisher applications by studying two examples. The first is modelled as closely as possible on a point to point application putting messages on a queue, and the second demonstrates creating topics dynamically - a more common pattern for publisher applications.

Writing a simple WebSphere MQ publisher application is just like writing a WebSphere MQ point to point application that puts messages to a queue (Table 5). The difference is you MQPUT messages to a topic, not to a queue.

Table 5. Point to point vs. publish/subscribe WebSphere MQ program pattern.

Step	Point to point MQ Call	Publish MQ Call
Connect to a queue manager	MQCONN	MQCONN
Open queue	MQOPEN	
Open topic		MQOPEN
Put message(s)	MQPUT	MQPUT
Close topic		MQCLOSE
Close queue	MQCLOSE	
Disconnect from queue manager	MQDISC	MQDISC

To make that concrete, there are two examples of applications to publish stock prices. In the first example ("Example 1: Publisher to a fixed topic"), that is modelled very closely on putting messages to a queue, the administrator creates a topic definition in a similar way to creating a queue. The programmer codes MQPUT to write messages to the topic instead of writing them to a queue. In the second example ("Example 2: Publisher to a variable topic" on page 49), the pattern of interaction of the program with WebSphere MQ is similar. The difference is the programmer provides the topic to which the message is written, rather than the administrator. In practice this usually means the topic string is content defined, or provided "out of band", that is, provided by human input, or by another source of information.

Related concepts

"Writing subscriber applications" on page 52
There are many more patterns of subscriber application than publisher. Three are illustrated: a WebSphere MQ application consuming messages from a queue, an application that creates a subscription and requires no knowledge of queuing, and finally an example that uses both queuing and subscriptions.

Example 1: Publisher to a fixed topic

A WebSphere MQ program to illustrate publishing to an administratively defined topic.

Note: The compact coding style is intended for readability not production use.

See the output in Figure 17 on page 47

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cmqc.h>
int main(int argc, char **argv)
{
    char     topicNameDefault[]   = "IBMSTOCKPRICE";
    char     publicationDefault[] = "129";
    MQCHAR48 qmName = "";

    MQHCONN  Hconn = MQHC_UNUSABLE_HCONN; /* connection handle                    */
    MQHOBJ   Hobj = MQHO_NONE;            /* object handle sub queue              */
    MQLONG   CompCode = MQCC_OK;          /* completion code                      */
    MQLONG   Reason = MQRC_NONE;          /* reason code                          */
    MQOD     td = {MQOD_DEFAULT};         /* Topic Descriptor                     */
    MQMD     md = {MQMD_DEFAULT};         /* Message Descriptor                   */
    MQPMO   pmo = {MQPMO_DEFAULT};        /* put message options                  */
    MQCHAR   resTopicStr[151];            /* Returned vale of topic string        */
    char *   topicName   = topicNameDefault;
    char *   publication = publicationDefault;
    memset   (resTopicStr, 0 , sizeof(resTopicStr));

    switch(argc){                         /* replace defaults with args if provided */
        default:
            publication = argv[2];
        case(2):
            topicName = argv[1];
        case(1):
            printf("Optional parameters:  TopicObject Publication\n");
    }
    do {
        MQCONN(qmName, &Hconn, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        td.ObjectType = MQOT_TOPIC;        /* Object is a topic                    */
        td.Version = MQOD_VERSION_4;       /* Descriptor needs to be V4            */
        strncpy(td.ObjectName, topicName, MQ_TOPIC_NAME_LENGTH);
        td.ResObjectString.VSPtr = resTopicStr;
        td.ResObjectString.VSBufSize = sizeof(resTopicStr)-1;
        MQOPEN(Hconn, &td, MQOO_OUTPUT | MQOO_FAIL_IF_QUIESCING, &Hobj, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        pmo.Options = MQPMO_FAIL_IF_QUIESCING | MQPMO_RETAIN;
        MQPUT(Hconn, Hobj, &md, &pmo, (MQLONG)strlen(publication)+1, publication, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        MQCLOSE(Hconn, &Hobj, MQCO_NONE, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        MQDISC(&Hconn, &CompCode, &Reason);
    } while (0);
    if (CompCode == MQCC_OK)
        printf("Published \"%s\" using topic \"%s\" to topic string \"%s\"\n",
               publication, td.ObjectName, resTopicStr);
    printf("Completion code %d and Return code %d\n", CompCode, Reason);
}
```

Figure 16. Simple WebSphere MQ publisher to a fixed topic.

```
X:\Publish1\Debug>PublishStock
Optional parameters:  TopicObject Publication
Published "129" using topic "IBMSTOCKPRICE" to topic string "NYSE/IBM/PRICE"
Completion code 0 and Return code 0

X:\Publish1\Debug>PublishStock IBMSTOCKPRICE 155
Optional parameters:  TopicObject Publication
Published "155" using topic "IBMSTOCKPRICE" to topic string "NYSE/IBM/PRICE"
Completion code 0 and Return code 0
```

Figure 17. Sample output from first publisher example

The lines of code selected below illustrate aspects of writing a publisher application for WebSphere MQ.

char topicNameDefault[] = "IBMSTOCKPRICE";
: A default topic name is defined in the program. You can override it by providing the name of a different topic object as the first argument to the program.

MQCHAR resTopicStr[151];
: `resTopicStr` is pointed at by `td.ResObjectString.VSPtr` and is used by `MQOPEN` to return the resolved topic string. Make the length of `resTopicStr` one larger than the length passed in `td.ResObjectString.VSBufSize` to give space for null termination.

memset (resTopicStr, 0, sizeof(resTopicStr));
: Initialize `resTopicStr` to nulls to ensure the resolved topic string returned in an `MQCHARV` is null terminated.

td.ObjectType = MQOT_TOPIC
: There is a new type of object for publish/subscribe: the *topic object*.

td.Version = MQOD_VERSION_4;
: To use the new type of object, you must use at least *version 4* of the object descriptor.

strncpy(td.ObjectName, topicName, MQ_OBJECT_NAME_LENGTH);
: The `topicName` is the name of a topic object, sometimes called an administrative topic object. In the example the topic object needs to be created beforehand, using WebSphere MQ Explorer or this MQSC command,

```
DEFINE TOPIC(IBMSTOCKPRICE) TOPICSTR(NYSE/IBM/PRICE) REPLACE;
```

td.ResObjectString.VSPtr = resTopicStr;
: The resolved topic string is echoed in the final `printf` in the program. Set up the `MQCHARV ResObjectString` structure for WebSphere MQ to return the resolved string back to the program.

MQOPEN(Hconn, &td, MQOO_OUTPUT | MQOO_FAIL_IF_QUIESCING, &Hobj, &CompCode, &Reason);
: Open the topic for output; just like opening a queue for output.

pmo.Options = MQPMO_FAIL_IF_QUIESCING | MQPMO_RETAIN;
: You want new subscribers to be able receive the publication, and by specifying `MQPMO_RETAIN` in the publisher, when we start a subscriber it receives the latest publication, published before the subscriber started, as its first matching publication. The alternative is to provide subscribers with publications published only after the subscriber started. In addition a subscriber has the option to decline to receive a retained publication by specifying `MQSO_NEW_PUBLICATIONS_ONLY` in its subscription.

MQPUT(Hconn, Hobj, &md, &pmo, (MQLONG)strlen(publication)+1, publication, &CompCode, &Reason);

Add 1 to the length of the string passed to MQPUT to pass the null termination character to WebSphere MQ as part of the message buffer.

What does the first example demonstrate? The example imitates as closely as possible the tried and tested traditional pattern for writing point to point WebSphere MQ programs. An important feature of the WebSphere MQ programming pattern is that the programmer is not concerned where messages are sent. The programmer's task is to connect to a queue manager, and pass to it the messages that are to be distributed to recipients. In the point-to-point paradigm, the programmer opens a queue (probably an alias queue) that the administrator has configured. The alias routes messages to a target queue, either on the local queue manager, or to a remote queue manager. Whilst the messages are waiting to be delivered, they are stored on queues somewhere between the source and the destination.

In the publish/subscribe pattern, instead of opening a queue, the programmer opens a topic. In our example, the topic is associated with a topic string by an administrator. The queue manager forwards the publication, using queues, to local or remote subscribers that have subscriptions that match the publication's topic string. In the case of retained publications the queue manager keeps the latest copy of the publication, even if it has no subscribers at present. The retained publication is available to forward to future subscribers. The publisher application plays no part in selecting or routing the publication to a destination; its task is to create and put publications to the topics defined by the administrator.

This fixed topic example is atypical of many publish/subscribe applications: it is static. It requires an administrator to define the topic strings and change the topics that are published on. Commonly publish/subscribe applications need to have knowledge of some or all of the topic tree. Perhaps topics change frequently, or perhaps although the topics do not change much, the number of topic combinations is very large and it is too onerous for an administrator to define a topic node for every topic string that might need to be published on. Perhaps topic strings are not known in advance of publication; a publisher application might use information from the publication content to specify a topic string, or it might have out of band information about topic strings to publish on, such as input from a browser. To cater for more dynamic styles of publishing, the next example shows how to create topics dynamically, as part of the publisher application.

Topics couple publishers and subscribers together. Designing the rules, or architecture, for naming topics, and organizing them in topic trees is a very important step in developing a publish/subscribe solution. Look carefully at the extent to which organization of the topic tree binds of publisher and subscriber programs together, and binds them to the content of the topic tree. Ask yourself the question whether changes in the topic tree will impact publisher and subscriber applications, and how you can minimize the impact. Built into the architecture of the WebSphere MQ publish/subscribe model is the notion of an administrative topic object that provides the root part, or root subtree, of a topic. The topic object gives you the option of defining the root part of the topic tree administratively that simplifies application programming and operations, and consequently improves maintainability. For example, if you are deploying multiple publish/subscribe applications that have isolated topic trees, then by administratively defining the root part of the topic tree, you can guarantee the isolation of topic trees, even if there is no consistency in the topic naming conventions adopted by the different applications.

In practice, publisher applications cover a spectrum from solely using fixed topics, as in this example, and variable topics, as in the next. “Example 2: Publisher to a variable topic” also demonstrates combining the use of topics and topic strings.

Related concepts

“Example 2: Publisher to a variable topic”
A Websphere MQ program to illustrate publishing to a programmatically defined topic.

“Writing subscriber applications” on page 52
There are many more patterns of subscriber application than publisher. Three are illustrated: a WebSphere MQ application consuming messages from a queue, an application that creates a subscription and requires no knowledge of queuing, and finally an example that uses both queuing and subscriptions.

Example 2: Publisher to a variable topic

A Websphere MQ program to illustrate publishing to a programmatically defined topic.

Note: The compact coding style is intended for readability not production use.

See the output in Figure 19 on page 51.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cmqc.h>
int main(int argc, char **argv)
{
    char     topicNameDefault[]   = "STOCKS";
    char     topicStringDefault[] = "IBM/PRICE";
    char     publicationDefault[] = "130";
    MQCHAR48 qmName = "";

    MQHCONN  Hconn = MQHC_UNUSABLE_HCONN; /* connection handle        */
    MQHOBJ   Hobj     = MQHO_NONE;   /* object handle sub queue       */
    MQLONG   CompCode = MQCC_OK;     /* completion code               */
    MQLONG   Reason   = MQRC_NONE;   /* reason code                   */
    MQOD     td = {MQOD_DEFAULT};    /* Topic Descriptor              */
    MQMD     md = {MQMD_DEFAULT};    /* Message Descriptor            */
    MQPMO   pmo = {MQPMO_DEFAULT};   /* put message options           */
    MQCHAR   resTopicStr[151];       /* Returned value of topic string */
    char *   topicName   = topicNameDefault;
    char *   topicString = topicStringDefault;
    char *   publication = publicationDefault;
    memset   (resTopicStr, 0 , sizeof(resTopicStr));

    switch(argc){            /* Replace defaults with args if provided */
        default:
            publication = argv[3];
        case(3):
            topicString = argv[2];
        case(2):
            if (strcmp(argv[1],"/"))  /* "/" invalid = No topic object  */
                topicName = argv[1];
            else
                *topicName = '\0';
        case(1):
            printf("Provide parameters:  TopicObject TopicString Publication\n");
    }

    printf("Publish \"%s\" to topic \"%-.48s\" and topic string \"%s\"\n", publication, topicName, topicString);
    do {
        MQCONN(qmName, &Hconn, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        td.ObjectType = MQOT_TOPIC;    /* Object is a topic             */
        td.Version = MQOD_VERSION_4;   /* Descriptor needs to be V4     */
        strncpy(td.ObjectName, topicName,  MQ_Q_NAME_LENGTH);
        td.ObjectString.VSPtr = topicString;
        td.ObjectString.VSLength = (MQLONG)strlen(topicString);
        td.ResObjectString.VSPtr = resTopicStr;
        td.ResObjectString.VSBufSize = sizeof(resTopicStr)-1;
        MQOPEN(Hconn, &td, MQOO_OUTPUT | MQOO_FAIL_IF_QUIESCING, &Hobj, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        pmo.Options = MQPMO_FAIL_IF_QUIESCING | MQPMO_RETAIN;
        MQPUT(Hconn, Hobj, &md, &pmo, (MQLONG)strlen(publication)+1, publication, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        MQCLOSE(Hconn, &Hobj, MQCO_NONE, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        MQDISC(&Hconn, &CompCode, &Reason);
    } while (0);
    if (CompCode == MQCC_OK)
        printf("Published \"%s\" to topic string \"%s\"\n",  publication, resTopicStr);
    printf("Completion code %d and Return code %d\n", CompCode, Reason);
}
```

Figure 18. Simple WebSphere MQ publisher to a variable topic.

```
X:\Publish2\Debug>PublishStock
Provide parameters:  TopicObject TopicString Publication
Publish "130" to topic "STOCKS" and topic string "IBM/PRICE"
Published "130" to topic string "NYSE/IBM/PRICE"
Completion code 0 and Return code 0

X:\Publish2\Debug>PublishStock / NYSE/IBM/PRICE 131
Provide parameters:  TopicObject TopicString Publication
Publish "131" to topic "" and topic string "NYSE/IBM/PRICE"
Published "131" to topic string "NYSE/IBM/PRICE"
Completion code 0 and Return code 0
```

Figure 19. Sample output from second publisher example

There are a few points to note about this example.

char topicNameDefault[] = "STOCKS";
: The default topic name STOCKS defines part of the topic string. You can override this topic name by providing it as the first argument to the program, or eliminate the use of the topic name by supplying / as the first parameter.

char topicString[101] = "IBM/PRICE";
: IBM/PRICE is the default topic string. You can override this topic string by providing it as the second argument to the program.

  The queue manager combines the topic string provided by the STOCKS topic object, "NYSE", with the topic string provided by the program "IBM/PRICE" and inserts a "/"between the two yielding "NYSE/IBM/PRICE" as the resolved topic string. The resulting topic string is the same as the one defined in the IBMSTOCKPRICE topic object, and has precisely the same effect.

  The administrative topic object associated with the resolved topic string is not necessarily the same topic object as passed to MQOPEN by the publisher. WebSphere MQ uses the tree implicit in the resolved topic string to work out which administrative topic object defines the attributes associated with the publication.

  Suppose there are two topic objects A and B, and A defines topic "a", and B defines topic "a/b" (Figure 20 on page 52). If the publisher program refers to topic object A and provides topic string "b", resolving the topic to the topic string "a/b", then the publication inherits its properties from topic object B because the topic matches the topic string "a/b" defined for B.

if (strcmp(argv[1],"/"))
: argv[1] is the optionally provided topicName. "/" is invalid as a topic name; here it signifies that there is no topic name, and the topic string is provided entirely by the program. The output in Figure 19 shows the whole topic string being supplied dynamically by the program.

strncpy(td.ObjectName, topicName, MQ_OBJECT_NAME_LENGTH);
: For the default case, the optional topicName needs to be created beforehand, using WebSphere MQ Explorer or this MQSC command:

  ```
  DEFINE TOPIC(STOCKS) TOPICSTR(NYSE) REPLACE;
  ```

td.ObjectString.VSPtr = topicString;
: The topic string is a MQCHARV field in the topic descriptor

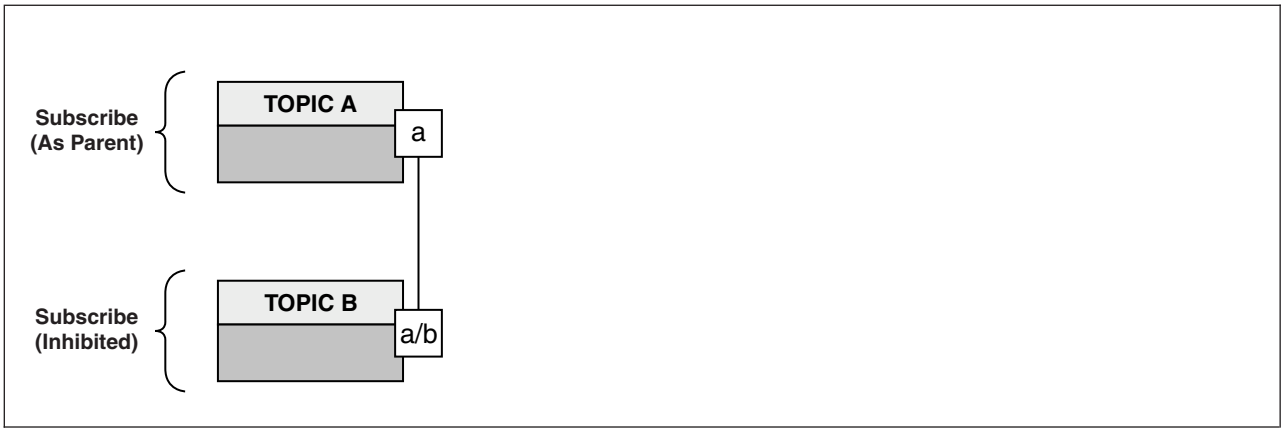


Figure 20. Topic object associations

What does the second example demonstrate? Although the code is very similar to the first example - effectively there are only two lines difference - the result is a significantly different program to the first. The programmer controls the destinations to which publications are sent. In conjunction with an ″administrator-light″ style used to design subscriber applications, no topics or queues need to be predefined to route publications from publishers to subscribers.

In the point-to-point messaging paradigm, queues have to be defined before messages are able to flow. For publish/subscribe, they do not, although WebSphere MQ implements publish/subscribe using its underlying queuing system; the benefits of guaranteed delivery, transactionality and loose coupling associated with messaging and queueing are inherited by publish/subscribe applications.

A designer has to decide whether publisher, and subscriber, programs are to be aware of the underlying topic tree or not, and also whether subscriber programs are aware of queueing or not. Study the subscriber example applications next. They are designed to be used with the publisher examples, typically publishing and subscribing to `NYSE/IBM/PRICE`.

Related concepts

“Example 1: Publisher to a fixed topic” on page 45
A WebSphere MQ program to illustrate publishing to an administratively defined topic.

“Writing subscriber applications”
There are many more patterns of subscriber application than publisher. Three are illustrated: a WebSphere MQ application consuming messages from a queue, an application that creates a subscription and requires no knowledge of queuing, and finally an example that uses both queuing and subscriptions.

Writing subscriber applications

There are many more patterns of subscriber application than publisher. Three are illustrated: a WebSphere MQ application consuming messages from a queue, an application that creates a subscription and requires no knowledge of queuing, and finally an example that uses both queuing and subscriptions.

In Table 6 on page 53 the three styles of consumer or subscriber are listed, together with the sequences of WebSphere MQ function calls that characterize them.

1. The first style, MQ Publication Consumer, is identical to a point to point MQ program that only does MQGET. The application has no knowledge that it is consuming publications - it is simply reading messages from a queue. The subscription that causes publications to get routed to the queue is created administratively using WebSphere MQ Explorer or a command.
2. The second style is the preferred pattern for most subscriber applications. The subscriber application creates the subscription, and then gets publications. The queue management is all performed by the queue manager.
3. In the third style, the subscriber application elects to open and close the underlying queue that is used for publications as well as issue subscriptions to fill the queue with publications.

One way to understand these styles is to study the example `C` programs listed in Table 6 for each of the styles. The examples are designed to be run in conjunction with the publisher example found in publisher examples.

Table 6. Point to point vs. subscribe WebSphere MQ program patterns.

Step	MQ message consumer	"Example 1: MQ Publication consumer" on page 54	"Example 2: Managed MQ subscriber" on page 56	"Example 3: Unmanaged MQ subscriber" on page 63
Connect to a queue manager	MQCONN	MQCONN	MQCONN	MQCONN
Open queue	MQOPEN	MQOPEN		MQOPEN
Subscribe			MQSUB	MQSUB
Put message(s)	MQGET	MQGET	MQGET	MQGET
Close queue	MQCLOSE	MQCLOSE	(MQCLOSE)	MQCLOSE
Close subscription			MQCLOSE	MQCLOSE
Disconnect from queue manager	MQDISC	MQDISC	MQDISC	MQDISC

Using MQCLOSE is always optional, either to release resources, pass MQCLOSE options, or just for symmetry with MQOPEN. Since you are unlikely to need to specify the MQCLOSE options when the subscription queue is closed in the Managed MQ subscriber case, and the symmetry argument is not relevant, the subscription queue is not explicitly closed in the Managed MQ subscriber example below.

Another way to understand publish/subscribe application patterns is too look at the interactions between the different entities involved. Lifeline, or UML sequence diagrams are a good way to study interactions. Three lifeline examples are described in "Publish/subscribe lifecycles" on page 73.

Related concepts

"Writing publisher applications" on page 45
Get started with writing publisher applications by studying two examples. The first is modelled as closely as possible on a point to point application putting messages on a queue, and the second demonstrates creating topics dynamically - a more common pattern for publisher applications.

"Publish/subscribe lifecycles" on page 73
Consider the lifecycles of topics, subscriptions, subscribers, publications, publishers and queues in designing publish/subscribe applications.

Example 1: MQ Publication consumer

The MQ Publication consumer is a WebSphere MQ message consumer that does not subscribe to topics itself.

To create the subscription and publication queue for this example run the following commands, or define the objects using WebSphere MQ Explorer.

```
DEFINE QLOCAL(STOCKTICKER) REPLACE;
DEFINE SUB(IBMSTOCKPRICESUB) DEST(STOCKTICKER) TOPICOBJ(IBMSTOCKPRICE) REPLACE;
```

The `IBMSTOCKPRICESUB` subscription references the `IBMSTOCK` topic object created for the publisher example and the local queue `STOCKTICKER`. The topic object `IBMSTOCK` defines the topic string that is used in the subscription, `NYSE/IBM/PRICE`. Note that the topic object and the queue used to receive publications need to be defined before the subscription is created.

There are a number of valuable facets to the MQ publication consumer pattern:

1. Multiprocessing: sharing out of the work of reading publications. The publications all go onto the single queue associated with the subscription topic. Multiple consumers can open the queue using `MQOO_INPUT_SHARED`.
2. Centrally managed subscriptions. Applications do not construct their own subscription topics or subscriptions; the administrator is responsible for where publications are sent.
3. Subscription concentration: multiple different subscriptions can be sent to a single queue.
4. Subscription durability: the queue receives all publications whether or not consumers are active.
5. Migration and coexistence: the consumer code works equally well for a point-to-point and a publish/subscribe scenario.

The subscription creates a relationship between the topic string `NYSE/IBM/PRICE` and the queue `STOCKTICKER`. Publications, including any currently retained publication, are forwarded to `STOCKTICKER` from the moment the subscription is created.

An administratively created subscription can be managed or unmanaged. A managed subscription takes effect as soon as it has been created, just like an unmanaged subscription. Not all the pattern facets are available to a managed subscription. See "Example 3: Unmanaged MQ subscriber" on page 63

Note: The compact coding style is intended for readability not production use.

The results are shown in Figure 22 on page 56

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cmqc.h>
int main(int argc, char **argv)
{
    MQCHAR   publicationBuffer[101];
    MQCHAR48 subscriptionQueueDefault = "STOCKTICKER";
    MQCHAR48 qmName = "";                       /* Use default queue manager    */

    MQHCONN  Hconn = MQHC_UNUSABLE_HCONN;       /* connection handle            */
    MQHOBJ   Hobj = MQHO_NONE;                  /* object handle sub queue      */
    MQLONG   CompCode = MQCC_OK;                /* completion code              */
    MQLONG   Reason = MQRC_NONE;                /* reason code                  */
    MQLONG   messlen = 0;
    MQOD     od = {MQOD_DEFAULT};               /* Unmanaged subscription queue */
    MQMD     md = {MQMD_DEFAULT};               /* Message Descriptor           */
    MQGMO   gmo = {MQGMO_DEFAULT};              /* Put message options          */
    char *   publication=publicationBuffer;
    char *   subscriptionQueue = subscriptionQueueDefault;

    switch(argc){                     /* Replace defaults with args if provided */
        default:
            subscriptionQueue = argv[1]
        case(1):
            printf("Optional parameter: subscriptionQueue\n");
    }

    do {
        MQCONN(qmName, &Hconn, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        strncpy(od.ObjectName, subscriptionQueue, MQ_Q_NAME_LENGTH);
        MQOPEN(Hconn, &od, MQOO_INPUT_AS_Q_DEF | MQOO_FAIL_IF_QUIESCING , &Hobj, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        gmo.Options = MQGMO_WAIT | MQGMO_NO_SYNCPOINT | MQGMO_CONVERT;
        gmo.WaitInterval = 10000;
        printf("Waiting %d seconds for publications from %s\n", gmo.WaitInterval/1000, subscriptionQueue);
        do {
            memcpy(md.MsgId, MQMI_NONE, sizeof(md.MsgId));
            memcpy(md.CorrelId, MQCI_NONE, sizeof(md.CorrelId));
            md.Encoding       = MQENC_NATIVE;
            md.CodedCharSetId = MQCCSI_Q_MGR;
            memset(publication, 0, sizeof(publicationBuffer));
            MQGET(Hconn, Hobj, &md, &gmo, sizeof(publicationBuffer)-1, publication, &messlen,
                  &CompCode, &Reason);
            if (Reason == MQRC_NONE)
                printf("Received publication \"%s\"\n", publication);
        }
        while (CompCode == MQCC_OK);
        if (CompCode != MQCC_OK && Reason != MQRC_NO_MSG_AVAILABLE) break;
        MQCLOSE(Hconn, &Hobj, MQCO_NONE, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        MQDISC(&Hconn, &CompCode, &Reason);
    } while (0);
    printf("Completion code %d and Return code %d\n", CompCode, Reason);
}
```

Figure 21. MQ publication consumer.

```
X:\Subscribe1\Debug>Subscribe1
Optional parameter: subscriptionQueue
Waiting 10 seconds for publications from STOCKTICKER
Received publication "129"
Completion code 0 and Return code 0
```

Figure 22. Output from MQ publication consumer

There are a couple of standard WebSphere MQ C language programming tips to be aware of:

memset(publication, 0, sizeof(publicationBuffer));
: Ensure the message has trailing a trailing null for easy formatting using `printf`. The publisher example includes the trailing null in the message buffer passed to `MQPUT` by adding 1 to `strlen(publication)`. Setting `MQCHAR` buffers to null is good programming style for WebSphere MQ C programs that use the buffers to store strings, ensuring a null follows an array of characters that does not completely fill the buffer.

MQGET(Hconn, Hobj, &md, &gmo, sizeof(publicationBuffer)-1, publication, &messlen, &CompCode, &Reason);
: Reserve one null at the end of the message buffer to ensure the returned message has trailing null in case `"if (messlen == strlen(publication));"` is true. This tip complements the preceding one, and ensures that there is at least one null in `publicationBuffer` that is not overwritten by the contents of `publication`.

Related concepts

"Example 2: Managed MQ subscriber"
The managed MQ subscriber is the core pattern for most subscriber applications. The example requires *no* administrative definition of queues, topics or subscriptions.

"Example 3: Unmanaged MQ subscriber" on page 63
The unmanaged subscriber is an important class of subscriber application. With it, you combine the benefits of publish/subscribe with *control* of queuing and consumption of publications. The example demonstrates different ways of combining subscriptions and queues.

"Writing publisher applications" on page 45
Get started with writing publisher applications by studying two examples. The first is modelled as closely as possible on a point to point application putting messages on a queue, and the second demonstrates creating topics dynamically - a more common pattern for publisher applications.

Example 2: Managed MQ subscriber

The managed MQ subscriber is the core pattern for most subscriber applications. The example requires *no* administrative definition of queues, topics or subscriptions.

This simplest kind of managed subscriber typically makes use of a *non-durable* subscription. The example focuses on a non-durable subscription. The subscription only lasts only as long as the lifetime of the subscription handle from `MQSUB`. Any publications that match the topic string during the lifetime of the subscription are sent to the subscription queue (and possibly a retained publication if the flag `MQSO_NEW_PUBLICATIONS_ONLY` is not set or defaulted, an earlier publication matching the topic string was retained, and the publication was persistent or the queue manager has not terminated, since the publication was created).

You can also use a *durable* subscription with this pattern. Typically if a managed durable subscription is used it is done for reliability reasons, rather than to establish a subscription that, without any errors occurring, would outlive the subscriber. See the discussion of different lifecycles associated with managed, unmanaged, durable and non-durable subscriptions in the related topics section.

Durable subscriptions are often associated with persistent publications, and non-durable subscriptions with non-persistent publications, but there is no necessary relationship between subscription durability and publication persistence. All four combinations of persistence and durability are possible.

For the managed non-durable case we are considering, the queue manager creates a subscription queue that is purged and deleted when the queue is closed. The publications are removed from the queue when the non-durable subscription is closed.

The valuable facets of the managed non-durable pattern exemplified by this code are listed below.

1. On demand subscription: the subscription topic string is dynamic. It is provided by the application when it runs.
2. Self managing queue: the subscription queue is self defining and managing.
3. Self managing subscription lifecycle: *non-durable* subscriptions only exist for the duration of the subscriber application.
   - If you define a *durable* managed subscription, then it results in a permanent subscription queue and publications continue to be stored on it with no subscriber programs being active. The queue manager deletes the queue (and clears any unretrieved publications from it) only after the application or administrator has chosen to delete the subscription. The subscription can be deleted using an administrative command, or by closing the subscription with the `MQCO_REMOVE_SUB` option.
   - Consider setting `SubExpiry` for durable subscriptions so that publications cease to be sent to the queue and the subscriber can consume any remaining publications before removing the subscription and causing the queue manager to delete the queue and any remaining publications on it.
4. Flexible topic string deployment: Subscription topic management is simplified by defining the root part of the subscription using an administratively defined topic. The root part of the topic tree is then hidden from the application. By hiding the root part an application can be deployed without the application inadvertently creating a topic tree that overlaps with another topic tree created by another instance, or another application.
5. Administered topics: by using a topic string in which the first part matches an administratively defined topic object, publications are managed according to the attributes of the topic object.
   - For example, if the first part of the topic string matches the topic string associated with a clustered topic object, then the subscription can receive publications from other members of the cluster
   - The selective matching of administratively defined topic objects and programmatically defined subscriptions enables you to combine the benefits of both. The administrator provides attributes for topics, and the programmer dynamically defines "sub-topics" without being concerned about the management of topics.
   - It is the resultant topic string which is used to match the topic object that provides the attributes associated with the topic, and not necessarily the

topic object named in sd.Objectname, thought they usually turn out to be one and the same. See the discussion in "Example 2: Publisher to a variable topic" on page 49.

By making the subscription durable in the example, publications continue to be sent to the subscription queue after the subscriber has closed the subscription with the MQCO_KEEP_SUB option. The queue continues to receive publications when the subscriber is not active. You can override this behavior by creating the subscription with the MQSO_PUBLICATIONS_ON_REQUEST option and using MQSUBRQ to request the retained publication.

The subscription can be resumed later by opening the subscription with the MQCO_RESUME option.

You can use the queue handle, Hobj, returned by MQSUB in a number of ways. The queue handle is used in the example to inquire on the name of the subscription queue. Managed queues are opened using the default model queues SYSTEM.NDURABLE.MODEL.QUEUE or SYSTEM.DURABLE.MODEL.QUEUE. You can override the defaults by providing your own durable and non-durable model queues on a topic by topic basis as properties of the topic object associated with the subscription.

Regardless of the attributes inherited from the model queues, you cannot reuse a managed queue handle to create an additional subscription. Nor can you obtain another handle for the managed queue by opening the managed queue a second time using the returned queue name. The queue behaves as if it has been opened for exclusive input.

Unmanaged queues are more flexible than managed queues. You can, for example share unmanaged queues, or define multiple subscriptions on the one queue. The next example, "Example 3: Unmanaged MQ subscriber" on page 63 demonstrates how to combine subscriptions with an unmanaged subscription queue.

Note: The compact coding style is intended for readability not production use.

The results are shown in Figure 25 on page 61.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cmqc.h>

void inquireQname(MQHCONN HConn, MQHOBJ Hobj, MQCHAR48 qName);

int main(int argc, char **argv)
{
    MQCHAR48 topicNameDefault     = "STOCKS";
    char     topicStringDefault[] = "IBM/PRICE";
    MQCHAR48 qmName = "";                 /* Use default queue manager     */
    MQCHAR48 qName = "";                  /* Allocate to query queue name  */
    char     publicationBuffer[101];      /* Allocate to receive messages  */
    char     resTopicStrBuffer[151];   /* Allocate to resolve topic string */

    MQHCONN  Hconn = MQHC_UNUSABLE_HCONN; /* connection handle             */
    MQHOBJ   Hobj = MQHO_NONE;            /* publication queue handle      */
    MQHOBJ   Hsub = MQSO_NONE;            /* subscription handle           */
    MQLONG   CompCode = MQCC_OK;          /* completion code               */
    MQLONG   Reason = MQRC_NONE;          /* reason code                   */
    MQLONG   messlen = 0;
    MQSD     sd = {MQSD_DEFAULT};         /* Subscription Descriptor       */
    MQMD     md = {MQMD_DEFAULT};         /* Message Descriptor            */
    MQGMO   gmo = {MQGMO_DEFAULT};        /* put message options           */

    char *   topicName   = topicNameDefault;
    char *   topicString = topicStringDefault;
    char *   publication = publicationBuffer;
    char *   resTopicStr = resTopicStrBuffer;
    memset(resTopicStr, 0, sizeof(resTopicStrBuffer));

    switch(argc){               /* Replace defaults with args if provided */
        default:
          topicString = argv[2];
        case(2):
          if (strcmp(argv[1],"/"))       /* "/" invalid = No topic object  */
             topicName = argv[1];
          else
             *topicName = '\0';
        case(1):
             printf("Optional parameters: topicName, topicString\nValues \"%s\"  \"%s\"\n",
                   topicName, topicString);
    }
```

Figure 23. Managed MQ subscriber - part 1: declarations and parameter handling.

There are some additional comments to make about the declarations in this example.

MQHOBJ Hobj = MQHO_NONE;
You cannot explicitly open a non-durable managed subscription queue to receive publications, but you do need to allocate storage for the object handle the queue manager returns when it opens the queue for you. It is important to initialize the handle to MQHO_OBJECT. This indicates to the queue manager that it needs to return a queue handle to the subscription queue.

MQSD sd = {MQSD_DEFAULT};
The new subscription descriptor, used in MQSUB.

MQCHAR48 qName;

Although the example doesn't require knowledge of the subscription queue, we do inquire the name of the subscription queue - the MQINQ binding is a little awkward in the C language, so you may find this part of the example useful to study.

```
    do {
        MQCONN(qmName, &Hconn, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        strncpy(sd.ObjectName, topicName, MQ_Q_NAME_LENGTH);
        sd.ObjectString.VSPtr = topicString;
        sd.ObjectString.VSLength = MQVS_NULL_TERMINATED;
        sd.Options = MQSO_CREATE | MQSO_MANAGED | MQSO_NON_DURABLE | MQSO_FAIL_IF_QUIESCING ;
        sd.ResObjectString.VSPtr = resTopicStr;
        sd.ResObjectString.VSBufSize = sizeof(resTopicStrBuffer)-1;
        MQSUB(Hconn, &sd, &Hobj, &Hsub, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        gmo.Options = MQGMO_WAIT | MQGMO_NO_SYNCPOINT | MQGMO_CONVERT;
        gmo.WaitInterval = 10000;
        inquireQname(Hconn, Hobj, qName);
        printf("Waiting %d seconds for publications matching \"%s\" from \"%-0.48s\"\n",
               gmo.WaitInterval/1000, resTopicStr, qName);
        do {
            memcpy(md.MsgId, MQMI_NONE, sizeof(md.MsgId));
            memcpy(md.CorrelId, MQCI_NONE, sizeof(md.CorrelId));
            md.Encoding       = MQENC_NATIVE;
            md.CodedCharSetId = MQCCSI_Q_MGR;
            memset(publicationBuffer, 0, sizeof(publicationBuffer));
            MQGET(Hconn, Hobj, &md, &gmo, sizeof(publicationBuffer-1),
                publication, &messlen, &CompCode, &Reason);
            if (Reason == MQRC_NONE)
                printf("Received publication \"%s\"\n", publication);
        }
        while (CompCode == MQCC_OK);
        if (CompCode != MQCC_OK && Reason != MQRC_NO_MSG_AVAILABLE) break;
        MQCLOSE(Hconn, &Hsub, MQCO_REMOVE_SUB, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        MQDISC(&Hconn, &CompCode, &Reason);
    } while (0);
    printf("Completion code %d and Return code %d\n", CompCode, Reason);
 return;
}
void inquireQname(MQHCONN Hconn, MQHOBJ Hobj, MQCHAR48 qName) {
#define _selectors 1
#define _intAttrs 1

    MQLONG select[_selectors] = {MQCA_Q_NAME}; /* Array of attribute selectors */
    MQLONG intAttrs[_intAttrs];                /* Array of integer attributes  */
    MQLONG CompCode, Reason;
    MQINQ(Hconn, Hobj, _selectors, select, _intAttrs, intAttrs, MQ_Q_NAME_LENGTH, qName,
          &CompCode, &Reason);
    if (CompCode != MQCC_OK) {
        printf("MQINQ failed with Condition code %d and Reason %d\n", CompCode, Reason);
        strcpy(qName, "unknown queue");
    }
    return;
}
```

Figure 24. Managed MQ subscriber - part 2: code body.

```
W:\Subscribe2\Debug>solution2
Optional parameters: topicName, topicString
Values "STOCKS"  "IBM/PRICE"
Waiting 10 seconds for publications matching "NYSE/IBM/PRICE" from "SYSTEM.MANAGED.NDURABLE.48A0AC7403300020       "
Received publication "150"
Completion code 0 and Return code 0

W:\Subscribe2\Debug>solution2 / NYSE/IBM/PRICE
Optional parameters: topicName, topicString
Values ""  "NYSE/IBM/PRICE"
Waiting 10 seconds for publications matching "NYSE/IBM/PRICE" from "SYSTEM.MANAGED.NDURABLE.48A0AC7403310020       "
Received publication "150"
Completion code 0 and Return code 0
```

Figure 25. Output from managed MQ subscriber

There are some additional comments to make about the code in this example.

strncpy(sd.ObjectName, topicName, MQ_Q_NAME_LENGTH);
: If topicName is null or blank *(default value)*, the topic name is not used to compute the resolved topic string.

sd.ObjectString.VSPtr = topicString;
: Rather than solely use a predefined topic object, in this example the programmer provides a topic object and a topic string, that are combined by MQSUB. Notice the topic string is a MQCHARV structure.

sd.ObjectString.VSLength = MQVS_NULL_TERMINATED;
: An alternative to setting the length of a MQCHARV field.

sd.Options = MQSO_CREATE | MQSO_MANAGED | MQSO_NON_DURABLE | MQSO_FAIL_IF_QUIESCING;
: After defining the topic string, the sd.Options flags need the most careful attention. There are many options, we shall specify only the most commonly used ones in this example, the others are left to default.

1. As the subscription is *non-durable*, in other words, it has a lifetime of the open subscription in the application, set the MQSO_CREATE flag. You can also set the *(default)* MQSO_NON_DURABLE flag for readability.
2. Complementing MQSO_CREATE is MQSO_RESUME. Both flags may be set together; the queue manager either creates a new subscription or resumes an existing subscription, whichever is appropriate. However, if you do specify MQSO_RESUME you must also initialize the MQCHARV structure for sd.SubName, even if there is no subscription to resume. Failure to initialize SubName results in a return code of 2440: MQRC_SUB_NAME_ERROR from MQSUB.

   Note: MQSO_RESUME is always ignored for a non-durable managed subscription: but specifying it without initializing the MQCHARV structure for sd.SubName does cause the error.
3. In addition there is a third flag affecting how the subscription is opened, MQSO_ALTER. Given the right permissions, the properties of a resumed subscription are changed to match other attributes specified in MQSUB.

   Note: At least one of the MQSO_CREATE, MQSO_RESUME and MQSO_ALTER flags must be specified. See the discussion in MQSD Options. There are examples of using all three flags in "Example 3: Unmanaged MQ subscriber" on page 63.
4. Set MQSO_MANAGED for the queue manager to manage the subscription for you automatically.

sd.ObjectString.VSLength = MQVS_NULL_TERMINATED;

Optionally, omit setting the length of MQCHARV for null terminated strings and use the null terminator flag instead.

sd.ResObjectString.VSPtr = resTopicStr;

The resulting topic string is echoed in first printf in the program. Set up MQCHARV ResObjectString for WebSphere MQ to return the resolved string back to our program.

Note: We initialized resTopicStringBuffer to nulls in memset(resTopicStr, 0, sizeof(resTopicStrBuffer)). Returned topic strings do not end with a trailing null.

sd.ResObjectString.VSBufSize = sizeof(resTopicStrBuffer)-1;

Set the buffer size of the sd.ResObjectString to one less than its actual size. This prevents overwriting the null terminator we provided, in case the resolved topic string fills the entire buffer.

Note: No error is returned if the topic string is longer than sizeof(resTopicStrBuffer)-1. Even if VSLength > VSBufSiz the length returned in sd.ResObjectString.VSLength is the length of the complete string and not necessarily the length of the returned string. Test sd.ResObjectString.VSLength < sd.ResObjectString.VSBufSiz to confirm the topic string is complete.

MQSUB(Hconn, &sd, &Hobj, &Hsub, &CompCode, &Reason);

The MQSUB function creates a subscription. If it is non-durable you are probably not interested in its name, though you can inspect its status in WebSphere MQ Explorer. You can provide the sd.SubName parameter as input, so you know what name to look for; you obviously have to avoid name clashes with other subscriptions.

MQCLOSE(Hconn, &Hsub, MQCO_REMOVE_SUB, &CompCode, &Reason);

Closing both the subscription and the subscription queue is optional. In the example the subscription is closed, but not the queue. The MQCLOSE MQCO_REMOVE_SUB option is the default in this case anyway as the subscription is non-durable. Using MQCO_KEEP_SUB is an error.

Note: the subscription *queue* is not closed by MQSUB, and its handle, Hobj, remains valid until the queue is closed by MQCLOSE or MQDISC. If the application terminates prematurely, the queue and subscription are cleaned up by the queue manager sometime after application termination.

Related concepts

"Example 1: MQ Publication consumer" on page 54
The MQ Publication consumer is a WebSphere MQ message consumer that does not subscribe to topics itself.

"Example 3: Unmanaged MQ subscriber"
The unmanaged subscriber is an important class of subscriber application. With it, you combine the benefits of publish/subscribe with *control* of queuing and consumption of publications. The example demonstrates different ways of combining subscriptions and queues.

"Writing publisher applications" on page 45
Get started with writing publisher applications by studying two examples. The first is modelled as closely as possible on a point to point application putting messages on a queue, and the second demonstrates creating topics dynamically - a more common pattern for publisher applications.

Example 3: Unmanaged MQ subscriber

The unmanaged subscriber is an important class of subscriber application. With it, you combine the benefits of publish/subscribe with *control* of queuing and consumption of publications. The example demonstrates different ways of combining subscriptions and queues.

The unmanaged pattern is more commonly associated with *durable* subscriptions than *non-durable*. Typically the lifecycle of a subscription created by an unmanaged subscriber is independent of the lifecycle of the subscribing application itself. By making the subscription durable the subscription receives publications even when no subscribing application is active.

You can create durable *managed* subscriptions to achieve the same result, but some applications require more flexibility and control over queues and messages than is possible with a managed subscription. For a durable managed subscription, the queue manager creates a permanent queue for the publications that match the subscription topic. It deletes the queue and associated publications when the subscription is deleted.

Typically durable *managed* subscriptions are used if the lifecycle of the application and the subscription is essentially the same, but hard to guarantee. By making the subscription durable, and having the publisher create persistent publications, there are no lost messages should the queue manager or subscriber terminate prematurely and need to be recovered.

The queue manager implicitly opens the durable managed subscription queue for a subscriber in such a way that shared processing of the queue is not possible. In addition, you cannot create more than one subscription for each managed queue and you may find the queues harder to manage because you have less control over the names of the queues. For these reasons, consider whether the *unmanaged* MQ subscriber is a better fit for applications requiring durable subscriptions than the *managed* MQ subscriber.

The code in Figure 28 on page 70 demonstrates an unmanaged durable subscription pattern. For illustration the code also creates unmanaged, non-durable subscriptions. The pattern facets exemplified by this code are,

1. On demand subscriptions: the subscription topic strings are dynamic. They are provided by the application when it runs.

2. Simplified subscription topic management: subscription topic management is simplified by defining the root part of the subscription topic string using an administratively defined topic. This hides the root part of the topic tree from the application. By hiding the root part a subscriber can be deployed to different topic trees.
3. Flexible subscription management: you can define a subscription either administratively, or create it on-demand in a subscriber program. There is no difference between administratively and programmatically created subscriptions, except an attribute that shows how the subscription was created. There is a third type of subscription that is created automatically by the queue manager for distribution of subscriptions. All subscriptions are displayed in the WebSphere MQ Explorer.
4. Flexible association of subscriptions with queues: a predefined local queue is associated with a subscription by the MQSUB function. There are different ways to use MQSUB to associate subscriptions with queues:
   a. Associate a subscription with a queue having *no* existing subscriptions, MQSO_CREATE + (Hobj from MQOPEN).
   b. Associate a *new* subscription with a queue having existing subscriptions, MQSO_CREATE + (Hobj from MQOPEN).
   c. Move a existing subscription to a different queue, MQSO_ALTER + (Hobj from MQOPEN).
   d. Resume an existing subscription associated with an existing queue, MQSO_RESUME + (Hobj = MQHO_NONE), or MQSO_RESUME + (Hobj = from MQOPEN of queue with existing subscription).

   - By combining MQSO_CREATE | MQSO_RESUME | MQSO_ALTER in different combinations, you can cater for different input states of the subscription and the queue without having to code multiple versions of MQSUB with different sd.Options values.
   - Alternatively, by coding a specific choice of MQSO_CREATE | MQSO_RESUME | MQSO_ALTER the queue manager returns an error (Table 7 on page 66) if the states of the subscription and queue provided as input to MQSUB are inconsistent with the value of sd.Options. Figure 34 on page 73 shows the results of issuing MQSUB for Subscription X with different individual settings of the sd.Options flag, and passing it three different object handles.

   Explore different inputs to the example program in Figure 27 on page 68 to become familiar with these different kinds of error. One common error, RC = 2440, that is not included in the cases listed in the table, is a subscription name error. it is commonly caused by passing a null or invalid subscription name with MQSO_RESUME or MQSO_ALTER.
5. Multiprocessing: you can share out of the work of reading publications to multiple consumers. The publications all go onto the single queue associated with the subscription topic. Consumers have a choice of opening the queue directly using MQOPEN or resuming the subscription using MQSUB.
6. Subscription concentration: multiple subscriptions can be created on the same queue. Be cautious with this capability as it can lead to "overlapping" subscriptions, and receiving the same publication multiple times. The MQSO_GROUP_SUB option eliminates duplicate publications caused by overlapping subscriptions.
7. Subscriber and consumer separation: As well as the three consumer models illustrated in the examples, another model is to separate the consumer from the subscriber. It is a variation of the unmanaged MQ Subscriber, but rather than issue the MQOPEN and MQSUB in the same program, one program subscribes to publications, and another program consumes them. For example, the subscriber

might be part of a publish/subscribe cluster and the consumer attached to a queue manager outside the queue manager cluster. The consumer receives publications through standard distributed queuing by defining the subscription queue as a remote queue definition.

Understanding the behavior of MQSO_CREATE | MQSO_RESUME | MQSO_ALTER is important, especially if you plan to simplify your code by using combinations of these options. Study the table Table 7 on page 66 that shows the results of passing different queue handles to MQSUB, and the results of running the example program shown in Figure 29 on page 71 to Figure 34 on page 73.

The scenario used to construct the table has one subscription X and two queues, A and B. The subscription name parameter sd.SubName is set to X, the name of a subscription attached to queue A. Queue B has no subscription attached to it.

Examine the top left cell. MQSUB is passed subscription X and the queue handle to queue A .

1. MQSO_CREATE fails because the queue handle corresponds to the queue A which already has a subscription to X. Contrast this behavior to the cell to the right: there, the call succeeds because queue B does not have a subscription to X attached to it.
2. MQSO_RESUME succeeds because the queue handle corresponds to the queue A which already has a subscription to X. In contrast, the call fails in the cell to the right.
3. MQSO_ALTER behaves in a similar way to MQSO_RESUME with respect to opening the subscription and queue. However if the attributes contained within the subscription descriptor passed to MQSUB differ from the attributes of the subscription, MQSO_RESUME fails, whereas MQSO_ALTER succeeds as long as the program instance has permission to alter the attributes. Note that you can never change the topic string in a subscription; but rather than return an error, MQSUB ignores the topic name and topic string values in the subscription descriptor and uses the values in the existing subscription.

Next, look at the cell below. MQSUB is passed subscription X and the queue handle to queue B.

1. MQSO_CREATE succeeds and creates subscription X on queue B because this is a new subscription on queue B.
2. MQSO_RESUME fails. MQSUB looks for subscription X on queue B and does not find it, but rather than returning *RC = 2428 - subscription X does not exist*, it returns *RC = 2019 - Subscription queue does not match queue object handle*. The behavior of the third option MQSO_ALTER suggests the reason for this unexpected error. MQSUB expects the queue handle to point to a queue with a subscription. It checks this first before checking whether the subscription named in sd.SubName exists.
3. MQSO_ALTER succeeds, and moves the subscription from queue A to queue B.
4. A case that is not shown in the table is if the subscription name of the subscription on queue A does not match the subscription name in sd.SubName. That call fails with a *RC = 2428 - subscription X does not exist on Queue A*.

Table 7. Errors from MQSUB with different queue handles and subscription combinations

	Queue A Subscription X **Queue B** No subscription	**Queue A** No subscription **Queue B** No subscription
Hobj for *Queue A* passed to MQSUB	**MQSO_CREATE** RC = 2432 - Subscription X already exists on Queue A **MQSO_RESUME** Resumes subscription X on Queue A **MQSO_ALTER** Resumes subscription X on Queue A and makes permitted alterations	**MQSO_CREATE** Creates subscription X on Queue A **MQSO_RESUME** RC = 2428 - Subscription X does not exist on Queue A **MQSO_ALTER** RC = 2428 - Subscription X does not exist on Queue A
Hobj for *Queue B* passed to MQSUB	**MQSO_CREATE** Creates new subscription X on Queue B **MQSO_RESUME** RC = 2019 - Subscription queue does not match queue object handle **MQSO_ALTER** Move subscription X from Queue A to Queue B	**MQSO_CREATE** Creates new subscription X on Queue B **MQSO_RESUME** RC = 2428 - subscription X does not exist on Queue B **MQSO_ALTER** RC = 2428 - subscription X does not exist on Queue B
MQHO_NONE passed to MQSUB	**MQSO_CREATE** RC = 2019 - Bad object handle: set MQSO_MANAGED flag to create a managed subscription and create a managed queue **MQSO_RESUME** Resumes subscription X on Queue A and returns Hobj to Queue A **MQSO_ALTER** Resumes subscription X on Queue A, returns Hobj to Queue A and makes permitted alterations	**MQSO_CREATE** RC = 2019 - Bad object handle: set MQSO_MANAGED flag to create a managed subscription and create a managed queue **MQSO_RESUME** RC = 2428 - No subscription X **MQSO_ALTER** RC = 2019 - Bad object handle: No queue A or B

Note: The compact coding style is intended for readability not production use.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <cmqc.h>

void inquireQname(MQHCONN HConn, MQHOBJ Hobj, MQCHAR48 qName);

int main(int argc, char **argv)
{
    MQCHAR48 topicNameDefault            = "STOCKS";
    char     topicStringDefault[]        = "IBM/PRICE";
    char     subscriptionNameDefault[]   = "IBMSTOCKPRICESUB";
    char     subscriptionQueueDefault[]  = "STOCKTICKER";
    char     publicationBuffer[101];      /* Allocate to receive messages     */
    char     resTopicStrBuffer[151];      /* Allocate to resolve topic string */
    MQCHAR48 qmName = "";                 /* Default queue manager            */
    MQCHAR48 qName = "";                  /* Allocate storage for MQINQ       */

    MQHCONN  Hconn = MQHC_UNUSABLE_HCONN; /* connection handle                */
    MQHOBJ   Hobj = MQHO_NONE;            /* subscription queue handle        */
    MQHOBJ   Hsub = MQSO_NONE;            /* subscription handle              */
    MQLONG   CompCode = MQCC_OK;          /* completion code                  */
    MQLONG   Reason = MQRC_NONE;          /* reason code                      */
    MQLONG   messlen = 0;
    MQOD     od = {MQOD_DEFAULT};         /* Unmanaged subscription queue     */
    MQSD     sd = {MQSD_DEFAULT};         /* Subscription Descriptor          */
    MQMD     md = {MQMD_DEFAULT};         /* Message Descriptor               */
    MQGMO   gmo = {MQGMO_DEFAULT};        /* get message options              */
    MQLONG   sdOptions = MQSO_CREATE | MQSO_RESUME | MQSO_DURABLE | MQSO_FAIL_IF_QUIESCING;

    char *   topicName = topicNameDefault;
    char *   topicString = topicStringDefault;
    char *   subscriptionName =  subscriptionNameDefault;
    char *   subscriptionQueue = subscriptionQueueDefault;
    char *   publication = publicationBuffer;
    char *   resTopicStr = resTopicStrBuffer;
    memset(resTopicStrBuffer, 0, sizeof(resTopicStrBuffer));
```

Figure 26. Unmanaged MQ subscriber - part 1: declarations.

```
    switch(argc){                   /* Replace defaults with args if provided */
default:
        switch((argv[5][0])) {
case('A'):  sdOptions =  MQSO_ALTER  | MQSO_DURABLE | MQSO_FAIL_IF_QUIESCING;
            break;
case('C'):  sdOptions =  MQSO_CREATE | MQSO_DURABLE | MQSO_FAIL_IF_QUIESCING;
            break;
case('R'):  sdOptions =  MQSO_RESUME | MQSO_DURABLE | MQSO_FAIL_IF_QUIESCING;
            break;
default:    ;
        }
case(5):
    if (strcmp(argv[4],"/")) /* "/" invalid = No subscription  */
        subscriptionQueue = argv[4];
    else {
        *subscriptionQueue = '\0';
        if (argc > 5) {
            if (argv[5][0] == 'C') {
                sdOptions = sdOptions + MQSO_MANAGED;
            }
        }
        else
            sdOptions = sdOptions + MQSO_MANAGED;
    }

case(4):
    if (strcmp(argv[3],"/")) /* "/" invalid = No subscription  */
        subscriptionName = argv[3];
    else {
        *subscriptionName = '\0';
        sdOptions = sdOptions - MQSO_DURABLE;
    }
case(3):
    if (strcmp(argv[2],"/")) /* "/" invalid = No topic string  */
        topicString = argv[2];
    else
        *topicString = '\0';
case(2):
    if (strcmp(argv[1],"/")) /* "/" invalid = No topic object  */
        topicName = argv[1];
    else
        *topicName = '\0';
case(1):
    sd.Options = sdOptions;
    printf("Optional parameters: "
    printf("topicName, topicString, subscriptionName, subscriptionQueue, A(lter)|C(reate)|R(esume)\n");
    printf("Values \"%-.48s\"  \"%s\" \"%s\"  \"%-.48s\" sd.Options=%d\n",
           topicName, topicString, subscriptionName, subscriptionQueue, sd.Options);
    }
```

Figure 27. Unmanaged MQ subscriber - part 2: parameter handling.

There are some additional comments to make about the parameter handling in this example.

switch((argv[5][0]))
: You have the choice of entering Alter | Create | Resume in parameter 5, to test the effect of overriding part of the MQSUB option setting used by default in the example. The default setting used by the example is MQSO_CREATE | MQSO_RESUME | MQSO_DURABLE.

Note: Setting MQSO_ALTER or MQSO_RESUME without setting MQSO_DURABLE is an error, and sd.SubName must be set and refer to a subscription that can be resumed or altered.

```
*subscriptionQueue = '\0';
sdOptions = sdOptions + MQSO_MANAGED;
```

If the default subscription queue, STOCKTICKER is replaced by a null string then as long as MQSO_CREATE is set, the example sets the MQSO_MANAGED flag and creates a dynamic subscription queue. If Alter or Resume are set in the fifth parameter the behavior of the example will depend on the value of subscriptionName.

```
*subscriptionName = '\0';
sdOptions = sdOptions - MQSO_DURABLE;
```

If the default subscription, IBMSTOCKPRICESUB, is replaced by a null string then the example removes the MQSO_DURABLE flag. If you run the example providing the default values for the other parameters an additional temporary subscription destined to STOCKTICKER is created and receives duplicate publications. Next time you run the example, without any parameters, you receive just one publication again.

```
    do {
        MQCONN(qmName, &Hconn, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        if (strlen(subscriptionQueue)) {
            strncpy(od.ObjectName, subscriptionQueue, MQ_Q_NAME_LENGTH);
            MQOPEN(Hconn, &od, MQOO_INPUT_AS_Q_DEF | MQOO_FAIL_IF_QUIESCING | MQOO_INQUIRE,
                   &Hobj, &CompCode, &Reason);
            if (CompCode != MQCC_OK) break;
        }
        strncpy(sd.ObjectName, topicName, MQ_TOPIC_NAME_LENGTH);
        sd.ObjectString.VSPtr = topicString;
        sd.ObjectString.VSLength = MQVS_NULL_TERMINATED;
        sd.SubName.VSPtr = subscriptionName;
        sd.SubName.VSLength = MQVS_NULL_TERMINATED;
        sd.ResObjectString.VSPtr = resTopicStr;
        sd.ResObjectString.VSBufSize = sizeof(resTopicStrBuffer)-1;
        MQSUB(Hconn, &sd, &Hobj, &Hsub, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        gmo.Options = MQGMO_WAIT | MQGMO_NO_SYNCPOINT | MQGMO_CONVERT;
        gmo.WaitInterval = 10000;
        inquireQname(Hconn, Hobj, qName);
        printf("Waiting %d seconds for publications matching \"%s\" from %-0.48s\n",
                gmo.WaitInterval/1000, resTopicStr, qName);
        do {
            memcpy(md.MsgId, MQMI_NONE, sizeof(md.MsgId));
            memcpy(md.CorrelId, MQCI_NONE, sizeof(md.CorrelId));
            md.Encoding       = MQENC_NATIVE;
            md.CodedCharSetId = MQCCSI_Q_MGR;
            MQGET(Hconn, Hobj, &md, &gmo, sizeof(publication), publication, &messlen, &CompCode, &Reason);
            if (Reason == MQRC_NONE)
                printf("Received publication \"%s\"\n", publication);
        }
        while (CompCode == MQCC_OK);
        if (CompCode != MQCC_OK && Reason != MQRC_NO_MSG_AVAILABLE) break;
        MQCLOSE(Hconn, &Hsub, MQCO_NONE, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        MQCLOSE(Hconn, &Hobj, MQCO_NONE, &CompCode, &Reason);
        if (CompCode != MQCC_OK) break;
        MQDISC(&Hconn, &CompCode, &Reason);
    } while (0);
    printf("Completion code %d and Return code %d\n", CompCode, Reason);
}
void inquireQname(MQHCONN Hconn, MQHOBJ Hobj, MQCHAR48 qName) {
#define _selectors 1
#define _intAttrs 1

    MQLONG select[_selectors] = {MQCA_Q_NAME}; /* Array of attribute selectors */
    MQLONG intAttrs[_intAttrs];   /* Array of integer attributes   */
    MQLONG CompCode, Reason;
    MQINQ(Hconn, Hobj, _selectors, select, _intAttrs, intAttrs, MQ_Q_NAME_LENGTH, qName, &CompCode, &Reason);
    if (CompCode != MQCC_OK) {
        printf("MQINQ failed with Condition code %d and Reason %d\n", CompCode, Reason);
        strncpy(qName, "unknown queue", MQ_Q_NAME_LENGTH);
    }
    return;
}
```

Figure 28. Unmanaged MQ subscriber - part 3: code body.

There are some additional comments to make on the code in this example,

if (strlen(subscriptionQueue))
If there is no subscription queue name then the example uses MQHO_NONE as the value of Hobj.

MQOPEN(...);
The subscription queue is opened and the queue handle saved in Hobj.

MQSUB(Hconn, &sd, &Hobj, &Hsub, &CompCode, &Reason);
The subscription is opened using the Hobj passed from MQOPEN (or MQHO_NONE if there is no subscription queue name). An unmanaged queue can be resumed without explicitly opening it with an MQOPEN.

MQCLOSE(Hconn, &Hsub, MQCO_NONE, &CompCode, &Reason);
The subscription is closed using the subscription handle. Depending on whether the subscription is durable or not, the subscription is closed with an implicit MQCO_KEEP_SUB or MQCO_REMOVE_SUB. You can close a durable subscription with MQCO_REMOVE_SUB, but you *cannot* close a non-durable subscription with MQCO_KEEP_SUB. The action of MQCO_REMOVE_SUB is to remove the subscription which stops any further publications being sent to the subscription queue.

MQCLOSE(Hconn, &Hobj, MQCO_NONE, &CompCode, &Reason);
No special action is taken if the subscription is unmanaged. If the queue is managed and the subscription closed with either an explicit or implicit MQCO_REMOVE_SUB, then all publications are purged from the queue and queue deleted at this point.

Results from the example illustrate aspects of publish/subscribe.

1. In Figure 29 the example starts by publishing 130 on the NYSE/IBM/PRICE topic.

```
W:\Subscribe3\Debug>..\..\Publish2\Debug\publishstock
Provide parameters:  TopicObject TopicString Publication
Publish "130" to topic "STOCKS" and topic string "IBM/PRICE"
Published "130" to topic string "NYSE/IBM/PRICE"
Completion code 0 and Return code 0
```

Figure 29. Publish 130 to NYSE/IBM/PRICE

2. In Figure 30 execution of the example using default parameters receives the retained publication 130.

   The provided topic object and topic string are ignored, as shown in Figure 34 on page 73. The topic object and topic string are always taken from the subscription object, when one is provided, and the topic string is immutable. The actual behavior of the example depends on the choice or combination of MQSO_CREATE, MQSO_RESUME, and MQSO_ALTER. In this example MQSO_RESUME is the option selected.

```
W:\Subscribe3\Debug>solution3
Optional parameters: topicName, topicString, subscriptionName, subscriptionQueue, A(lter)|C(reate)|R(esume)
Values "STOCKS"  "IBM/PRICE" "IBMSTOCKPRICESUB"  "STOCKTICKER" sd.Options=8206
Waiting 10 seconds for publications matching "NYSE/IBM/PRICE" from STOCKTICKER
Received publication "130"
Completion code 0 and Return code 0
```

Figure 30. Receive the retained publication

3. In (Figure 31 on page 72) no publications are received, because the durable subscription has already received the retained publication. In this example, the subscription is resumed by providing only the subscription name without the queue name. If the queue name was provided, the queue would be opened first and the handle passed to MQSUB.

Note: The 2038 error from MQINQ is due to the implicit MQOPEN of STOCKTICKER by MQSUB not including the MQOO_INQUIRE option. Avoid the 2038 return code from MQINQ by opening the queue explicitly.

```
W:\Subscribe3\Debug>solution3 STOCKS IBM/PRICE IBMSTOCKPRICESUB / Resume
Optional parameters: topicName, topicString, subscriptionName, subscriptionQueue, A(lter)|C(reate)|R(esume)
Values "STOCKS"  "IBM/PRICE" "IBMSTOCKPRICESUB"  "" sd.Options=8204
MQINQ failed with Condition code 2 and Reason 2038
Waiting 10 seconds for publications matching "NYSE/IBM/PRICE" from unknown queue
Completion code 0 and Return code 0
```

Figure 31. Resume subscription

4. In Figure 32, the example creates a non-durable unmanaged subscription using STOCKTICKER as the destination. Because this is a new subscription, it receives the retained publication.

```
W:\Subscribe3\Debug>solution3 STOCKS IBM/PRICE / STOCKTICKER Create
Optional parameters: topicName, topicString, subscriptionName, subscriptionQueue, A(lter)|C(reate)|R(esume)
Values "STOCKS"  "IBM/PRICE" ""  "STOCKTICKER" sd.Options=8194
Waiting 10 seconds for publications matching "NYSE/IBM/PRICE" from STOCKTICKER
Received publication "130"
Completion code 0 and Return code 0
```

Figure 32. Receive retained publication with new unmanaged non durable subscription

5. To demonstrate overlapping subscriptions, another publication is sent, changing the retained publication. Next, a new non-durable, unmanaged subscription is created by not providing a subscription name (Figure 33). The retained publication is received twice, once for the new subscription, and once for the durable IBMSTOCKPRICESUB subscription that is still active on the STOCKTICKER queue.

   The example is an illustration it is the queue that has subscriptions, and not the application. Despite not referring to the IBMSTOCKPRICESUB subscription in this invocation of the application, the application receives the publication twice: once from the durable subscription that was created administratively, and once from the non-durable subscription created by the application itself.

```
W:\Subscribe3\Debug>..\..\Publish2\Debug\publishstock
Provide parameters:  TopicObject TopicString Publication
Publish "130" to topic "STOCKS" and topic string "IBM/PRICE"
Published "130" to topic string "NYSE/IBM/PRICE"
Completion code 0 and Return code 0

W:\Subscribe3\Debug>solution3 STOCKS IBM/PRICE / STOCKTICKER Create
Optional parameters: topicName, topicString, subscriptionName, subscriptionQueue, A(lter)|C(reate)|R(esume)
Values "STOCKS"  "IBM/PRICE" ""  "STOCKTICKER" sd.Options=8194
Waiting 10 seconds for publications matching "NYSE/IBM/PRICE" from STOCKTICKER
Received publication "130"
Received publication "130"
Completion code 0 and Return code 0
```

Figure 33. Overlapping subscriptions

6. In Figure 34 on page 73 the example demonstrates that providing a new topic string and an existing subscription does not result in a changed subscription.
   a. In the first case, Resume resumes the existing subscription, as you might expect, and ignores the changed topic string.
   b. In the second case, Alter causes an error, RC = 2510, Topic not alterable.

c. In the third example, `Create` causes an error `RC = 2432, Sub already exists`.

```
W:\Subscribe3\Debug>solution3 "" NASDAC/IBM/PRICE IBMSTOCKPRICESUB STOCKTICKER Resume
Optional parameters: topicName, topicString, subscriptionName, subscriptionQueue, A(lter)|C(reate)|R(esume)
Values ""  "NASDAC/IBM/PRICE" "IBMSTOCKPRICESUB"  "STOCKTICKER" sd.Options=8204
Waiting 10 seconds for publications matching "NYSE/IBM/PRICE" from STOCKTICKER
Received publication "130"
Completion code 0 and Return code 0

W:\Subscribe3\Debug>solution3 "" NASDAC/IBM/PRICE IBMSTOCKPRICESUB STOCKTICKER Alter
Optional parameters: topicName, topicString, subscriptionName, subscriptionQueue, A(lter)|C(reate)|R(esume)
Values ""  "NASDAC/IBM/PRICE" "IBMSTOCKPRICESUB"  "STOCKTICKER" sd.Options=8201
Completion code 2 and Return code 2510

W:\Subscribe3\Debug>solution3 "" NASDAC/IBM/PRICE IBMSTOCKPRICESUB STOCKTICKER Create
Optional parameters: topicName, topicString, subscriptionName, subscriptionQueue, A(lter)|C(reate)|R(esume)
Values ""  "NASDAC/IBM/PRICE" "IBMSTOCKPRICESUB"  "STOCKTICKER" sd.Options=8202
Completion code 2 and Return code 2432
```

Figure 34. Subscription topics cannot be changed

Related concepts

"Example 1: MQ Publication consumer" on page 54
The MQ Publication consumer is a WebSphere MQ message consumer that does not subscribe to topics itself.

"Example 2: Managed MQ subscriber" on page 56
The managed MQ subscriber is the core pattern for most subscriber applications. The example requires *no* administrative definition of queues, topics or subscriptions.

"Writing publisher applications" on page 45
Get started with writing publisher applications by studying two examples. The first is modelled as closely as possible on a point to point application putting messages on a queue, and the second demonstrates creating topics dynamically - a more common pattern for publisher applications.

Publish/subscribe lifecycles

Consider the lifecycles of topics, subscriptions, subscribers, publications, publishers and queues in designing publish/subscribe applications.

The lifecycle of an object, such as a subscription, starts with its creation and ends with its deletion. It may also include other states and changes that it goes through, such as temporary suspension, having parent and children topics, expiration and deletion.

Traditionally WebSphere MQ objects such as queues are created administratively, or by administrative programs using Programmable Command Format (PCF). Publish/subscribe is different in providing the `MQSUB` and `MQCLOSE` API verbs to create and delete subscriptions, having the concept of managed subscriptions that not only create and delete queues, but also clean up unconsumed messages, and having associations between administratively created topic objects and programmatically or administratively created topic strings.

This functional richness caters for a wide range of publish/subscribe requirements, and also simplifies designing some common patterns of publish/subscribe application. Managed subscriptions, for example, simplify both the programming and administration of a subscription that is intended to last only as long as the

program that created it. Unmanaged subscriptions simplify programming where there is a looser connection between subscribing and consuming publications. Centrally created subscriptions are useful where the pattern is one of routing publication traffic to consumers based on a centralized model of control, for example sending flight information to automated gates, whereas programmatically created subscriptions might be used if gate staff are responsible for subscribing to the passengers records for that flight, by entering a flight number at a gate.

In this last example a managed durable subscription might be appropriate: managed, because the subscriptions are being created very often, and have a clear endpoint when the gate closes and the subscription can be programmatically removed; durable, to avoid losing a passenger record due to the gate subscriber program going down for one reason or another[1]. To initiate the publication of passenger records to the gate, a possible design would be for the gate application to both subscribe to the passenger records using the gate number, and publish the gate opening event using the gate number. The publisher responds to the gate opening event by publishing the passenger records - which might then also go to other interested parties, such as billing, to record the flight is taking place, and to customer services, to text notifications to passengers' mobile phones of the gate number.

The centrally managed subscription might use a durable unmanaged model, routing passenger lists to the gate using a predefined queue for each gate.

The following three examples of publish/subscribe lifecycles illustrate how managed non-durable, managed durable, and unmanaged durable subscribers interact with subscriptions, topics, queues, publishers and the queue manager, and how the responsibilities might be divided between administration and the subscriber programs.

Managed non-durable subscriber

"Publish/subscribe lifecycles" on page 73 shows an application creating a managed non-durable subscription, getting two messages that are published to the topic identified in the subscription, and terminating. The interactions labeled in an italic grey font with dotted arrows are implicit.

There are some points to note.

1. The application creates a subscription on a topic that has already been published to twice. When the subscriber receives its first publication, it receives the *second* publication which is the currently retained publication.
2. The queue manager creates a temporary subscription queue as well as creating a subscription for the topic.
3. The subscription has an expiry. When the subscription expires no more publications on the topic are sent to this subscription, but the subscriber continues to get messages published before the subscription expired. Publication expiry is not affected by subscription expiry.
4. The fourth publication is not placed on the subscription queue and consequently the last `MQGET` does not return a publication.
5. Although the subscriber closes its subscription, it does not close its connection to the queue or the queue manager.

1. The publisher must send the passenger records as persistent messages to avoid other possible failures, of course.

6. The queue manager cleans up shortly after the application terminates. Because the subscription is managed and non-durable, the subscription queue is deleted.

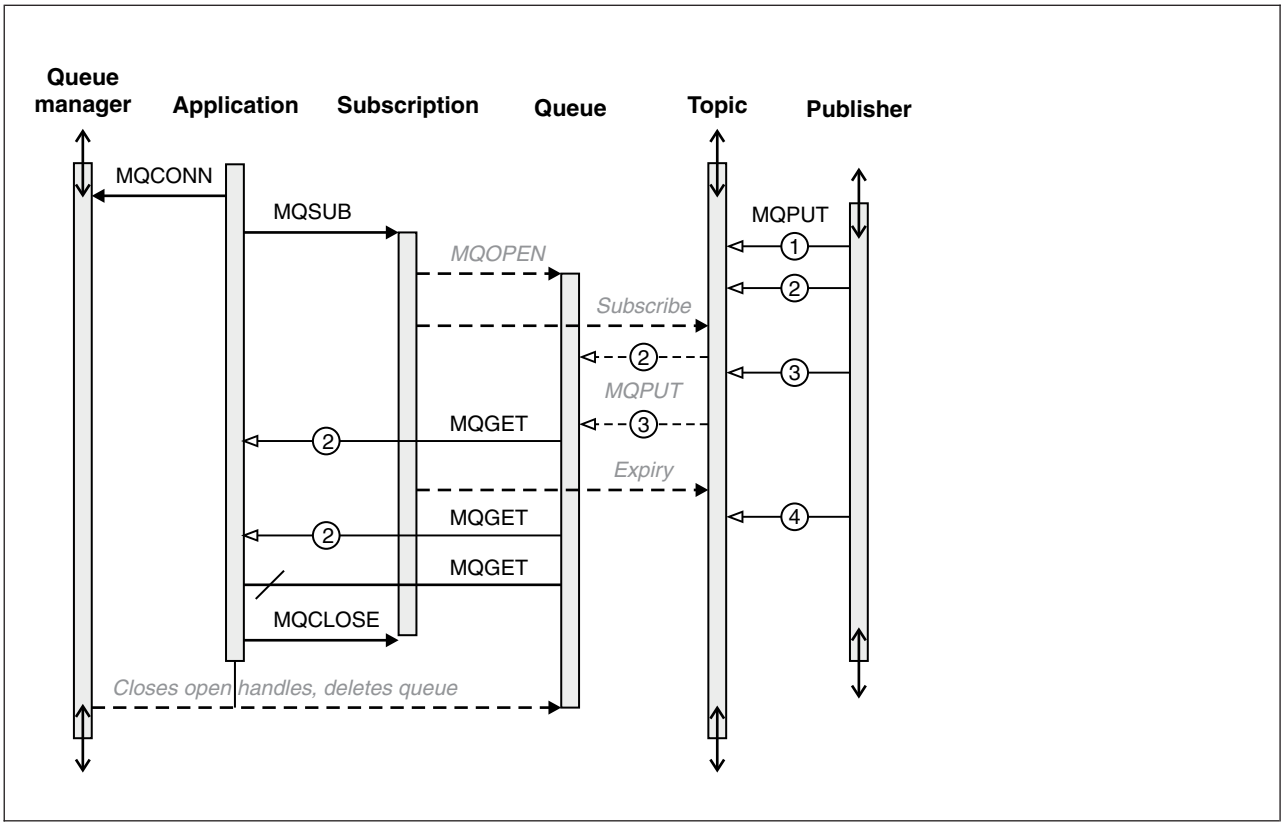


Figure 35. Managed non-durable subscriber lifelines

Managed durable subscriber

The managed durable subscriber takes the previous example a step further, and shows a managed subscription surviving the termination and restart of the subscribing application.

There are some new points to note.

1. In this example, unlike the last, the publication topic did not exist before it was defined in the subscription.
2. The first time the subscriber terminates, it closes the subscription with the option `MQCO_KEEP_SUB`. That is the default behavior for implicitly closing a managed durable subscription.
3. When the subscriber resumes the subscription, the subscription queue is reopened.
4. The new publication 2, placed on the queue before it is reopened, is available to `MQGET`, even after the subscription has been removed.

   Even though the subscription is durable, the subscriber reliably receives all messages sent by the publisher only if *both* the subscription is durable and the messages persistent. Message persistence depends on the setting of the `Persistent` field in the `MQMD` of the message sent by the publisher. A subscriber has no control over this.

5. Closing the subscription with the flag MQCO_REMOVE_SUB removes the subscription, stopping any further publications being placed on the subscription queue. When the subscription queue is closed, then the queue manager removes the unread publication 3, and then deletes the queue. The action is equivalent to administratively deleting the subscription.

   Note: Do not delete the queue manually, or issue MQCLOSE with the option MQCO_DELETE, or MQCO_PURGE_DELETE. The visible implementation details of a managed subscription is not part of the supported WebSphere MQ interface. The queue manager manage cannot manage a subscription reliably unless it has complete control.

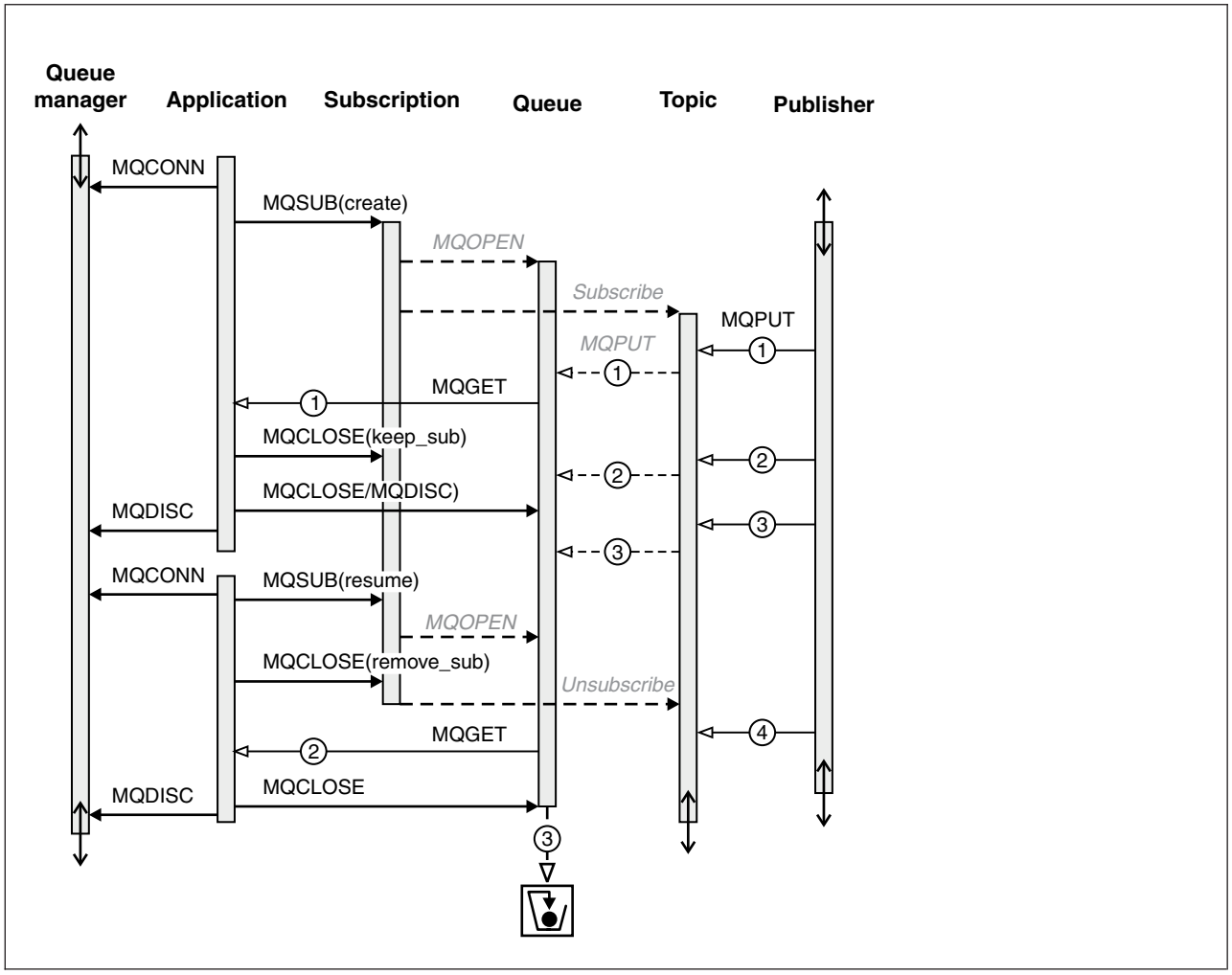


Figure 36. Managed durable subscriber lifelines

Unmanaged durable subscriber

An administrator is added in the third example: the unmanaged durable subscriber. It is a good example to show how the administrator might interact with a publish/subscribe application.

The points to note are listed.

1. The publisher puts a message, 1, to a topic that later becomes associated with the topic object that is used for subscription. The topic object defines a topic string that matches the topic that was published to by using wild-cards.
2. The topic has a retained publication.
3. The administrator creates a topic object, a queue and a subscription. The topic object and queue need to be defined before the subscription.
4. The application opens the queue associated with the subscription and passes MQSUB the handle of the queue. It could, alternatively, simply open the subscription, passing it the queue handle MQHO_NONE. The converse is not true, it cannot resume a subscription by passing it only queue handle without a subscription name - a queue might have multiple subscriptions.
5. The application opens the subscription using the option MQSO_RESUME even though it is the first time it has opened the subscription. It is resuming an administratively created subscription.
6. The subscriber receives the retained publication, 1. Publication 2, although published before any publications were received by the subscriber, was published after the subscription started, and is the second publication on the subscription queue.

   Note: If the retained publication is not published as a persistent message, then it is lost after queue manager restart.
7. In this example the subscription is durable. It is possible for a program to create an unmanaged non-durable subscription; it should be obvious this is not something an administrator can do.
8. The effect of the option MQCO_REMOVE_SUB on closing the subscription is to remove the subscription just as if the administrator had deleted it. This stops any further publications being sent to the queue, but does not affect publications that are already on the queue, even when the queue is closed, unlike a *managed* durable subscription.
9. The administrator later deletes the remaining message, 3, and deletes the queue.

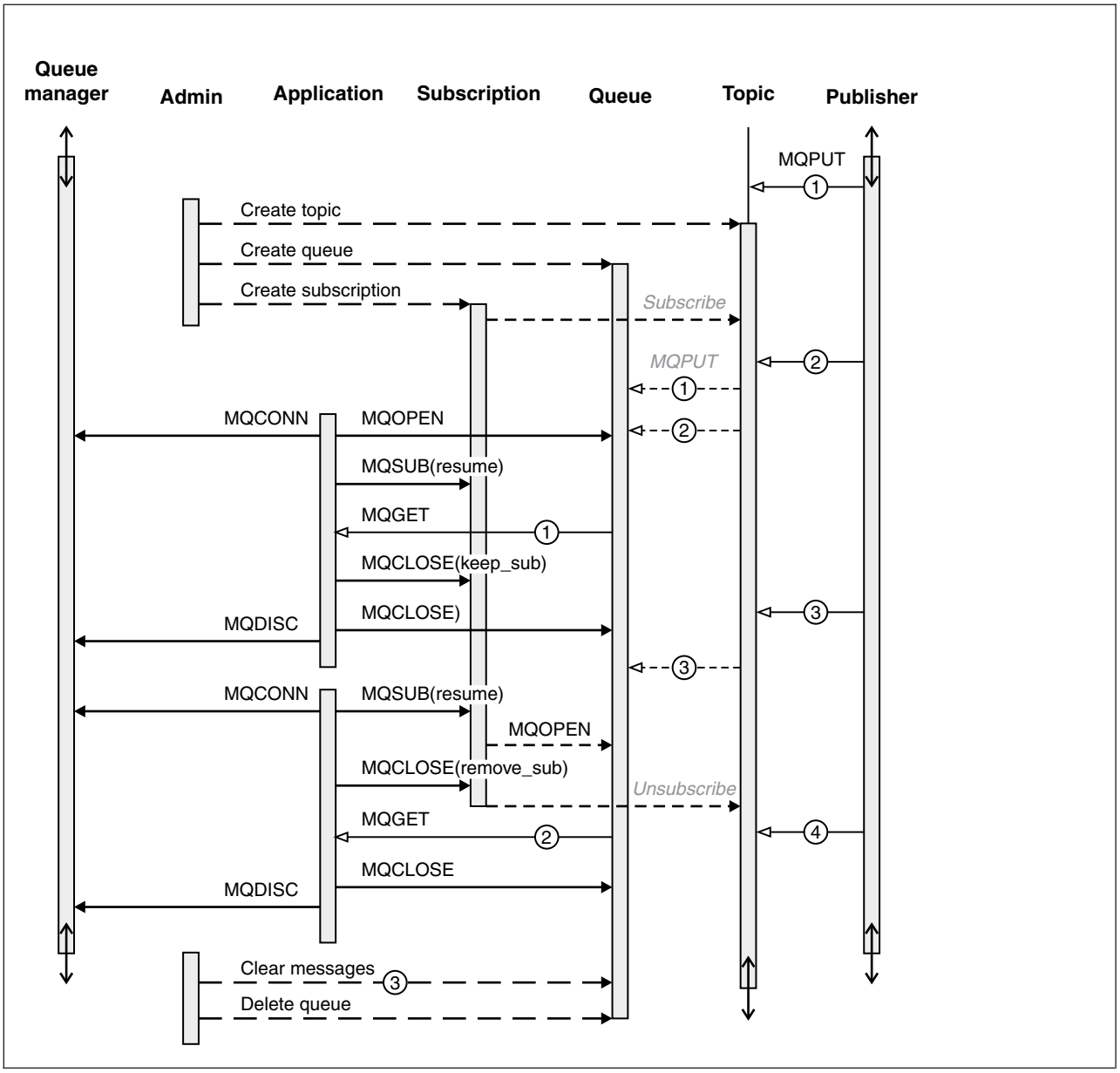


Figure 37. Unmanaged durable subscriber lifelines

A normal pattern for an unmanaged subscription is for queue and subscription housekeeping to be performed by the administrator. Typically one would not attempt to emulate the behavior of a managed subscriber and tidy up queues and subscriptions programmatically in application code. If you find yourself needing to write management logic, question whether you can achieve the same results using a managed pattern. It is not easy to write tightly synchronized, completely reliable management code. It is easier to tidy up later, either manually, or using a automated management program, when you can be sure that messages, subscriptions, and queues can be simply deleted, regardless of their state.

Publish/subscribe message properties

Several message properties relate to Websphere MQ publish/subscribe messaging.

PubAccountingToken

This is the value that will be in the AccountingToken field of the Message Descriptor (MQMD) of all publication messages matching this subscription. AccountingToken is part of the identity context of the message. For more information about message context, see the *WebSphere MQ Application Programming Guide*. For more information about the AccountingToken field in the MQMD, see the *WebSphere Application Programming Reference*.

PubApplIdentityData

This is the value that will be in the ApplIdentityData field of the Message Descriptor (MQMD) of all publication messages matching this subscription. ApplIdentityData is part of the identity context of the message. For more information about message context, see the *WebSphere MQ Application Programming Guide*. For more information about the ApplIdentityData field in the MQMD, see the *WebSphere Application Programming Reference*.

If the option MQSO_SET_IDENTITY_CONTEXT is not specified, the ApplIdentityData which will be set in each message published for this subscription is blanks, as default context information.

If the option MQSO_SET_IDENTITY_CONTEXT is specified, the PubApplIdentityData is being generated by the user and this field is an input field which contains the ApplIdentityData to be set in each publication for this subscription.

PubPriority

This is the value that will be in the Priority field of the Message Descriptor (MQMD) of all publication messages matching this subscription. For more information about the Priority field in the MQMD, see the *WebSphere Application Programming Reference*.

The value must be greater than or equal to zero; zero is the lowest priority. The following special values can also be used:

- MQPRI_PRIORITY_AS_Q_DEF - When a subscription queue is provided in the Hobj field in the MQSUB call, and is not a managed handle, then the priority for the message is taken from the DefProirity attribute of this queue. If the queue so identified is a cluster queue or there is more than one definition in the queue-name resolution path then the priority is determined when the publication message is put to the queue as described for Priority in the MQMD in the *WebSphere MQ Application Programming Reference*. If the MQSUB call uses a managed handle, the priority for the message is taken from the DefPriority attribute of the model queue associated with the topic subscribed to.
- MQPRI_PRIORITY_AS_PUBLISHED - The priority for the message is the priority of the original publication. This is the initial value of this field.

SelectionString

This variable length field will be returned on output from an MQSUB call using the MQSO_RESUME option, if a big enough buffer is provided. If the buffer provided on the call is not big enough (by the value in VSBufSize) only the length of the selection string will be returned in the VSLength field of the MQCHARV and the contents of the buffer will not be altered. The MQSUB call with the

MQSO_RESUME option can then be issued again providing a buffer long enough to fit VSLength bytes.

SubCorrelId

Attention: a correlation identifier can only be passed between queue managers in a publish/subscribe cluster, not a hierarchy.

All publications sent to match this subscription will contain this correlation identifier in the message descriptor. If multiple subscriptions use the same queue to get their publications from, using MQGET by correlation id allows only publications for a specific subscription to be obtained. This correlation identifier can either be generated by the queue manager or by the user.

If the option MQSO_SET_CORREL_ID is not specified, the correlation identifier is generated by the queue manager and this field is an output field which contains the correlation identifier which will be set in each message published for this subscription.

If the option MQSO_SET_CORREL_ID is specified, the correlation identifier is being generated by the user and this field is an input field which contains the correlation identifier to be set in each publication for this subscription. In this case, if the field contains MQCI_NONE, the correlation identifier which will be set in each message published for this subscription will be the correlation identifier created by the original put of the message.

If the option MQSO_GROUP_SUB is specified and the correlation identifier specified is the same as an existing grouped subscription using the same queue and an overlapping topic string, only the most significant subscription in the group is provided with a copy of the publication.

SubUserData

This is the subscription user data. The data provided on the subscription in this field will be included as the MQSubUserData message property of every publication sent to this subscription.

Message ordering

For a given topic, messages are published by the queue manager in the same order as they are received from publishing applications (subject to reordering based on message priority). This normally means that each subscriber receives messages from a particular queue manager, on a particular topic, from a particular publisher in the order that they are published by that publisher.

However, as with all WebSphere MQ messages, it is possible for messages, occasionally, to be delivered out of order. This can happen in the following situations:

- If a link in the network goes down and subsequent messages are rerouted along another link
- If a queue becomes temporarily full, or put-inhibited, so that a message is put to a dead-letter queue and therefore delayed, while subsequent messages pass straight through.

- If the administrator deletes a queue manager when publishers and subscribers are still operating, causing queued messages to be put to the dead-letter queue and subscriptions to be interrupted.

If these circumstances cannot occur, publications are always delivered in order.

Intercepting publications

You can intercept a publication, modify it, and then republish it before it reaches any other subscriber.

You might want to intercept a publication before it reaches a subscriber in order to do one of the following actions:

- Attach additional information to the message
- Block the message
- Transform the message

You can perform the same operation on each message or vary the operation depending on something in the message or message header.

After the message has been intercepted, it can be passed on to an application capable of doing message transformation.

To intercept a publication use the **subscription level** subscription attribute. The interceptor is simply another subscriber to the topic the messages it must intercept are being published on. The interceptor then republishes the message so that it can be received by other subscribers.

To ensure the interceptor receives the messages before any other subscribers, make sure it has the highest subscription level of all subscribers by using the *SubLevel* field in the MQSD. By default, subscribers have a *SubLevel* of 1.

- If you have one intercepting subscriber it should be configured to subscribe at a *SubLevel* of 9.
- If more than one intercepting application is required to receive the publication, set the sublevel of each interceptor's subscription appropriately to determine the order in which these intercepting applications receive the publication.

The interceptor with the highest subscription level receives the publication first, after which it is republished and received by the subscription with the next highest subscription level, and so on. When configuring multiple intercepting applications, there should be no more than one at each *SubLevel* value which republishes the message; otherwise duplicate publications will be sent to the final set of subscribing applications because more than one interceptor republishes the message to the next *SubLevel* down.

By default, applications publish to a topic using a *PubLevel* of 9. *PubLevel* is a field in the MQPMO. If there are any subscriptions with a *SubLevel* of 9, only those subscriptions are given a copy of the publication. If there are no subscriptions with a *SubLevel* of 9, the publications are given to all those subscriptions on this topic which have the highest *SubLevel*.

An intercepting application should make its subscription using the options described in Table 8 on page 82.

Table 8. Intercepting subscriber options

Subscription option	Notes
MQSO_SET_CORREL_ID and *SubCorrelId* set to MQCI_NONE	This ensures that the *CorrelId* in the publication message when it is re-published by the interceptor, is the one set by the original publisher, in case any subscriptions at a lower *SubLevel* has requested that. **Attention:** a correlation identifier can only be passed between queue managers in a publish/subscribe cluster, not a hierarchy.
PubPriority set to MQPRI_PRIORITY_AS_PUBLISHED	This ensures that the *Priority* in the publication message when it is re-published by the interceptor, is the one set by the original publisher, in case any subscriptions at a lower *SubLevel* has requested that.

An intercepting application should process the publication message (for example, transform or encrypt it) and then republish it to the same topic at a publication level one lower than the subscription level which intercepted it. For example, a subscription with a *SubLevel* of 9 should republish the message with a *PubLevel* of 8. In order to republish the message correctly, several pieces of information are required as shown in Table 9, and the intercepting application should use the same MQMD as in the original message and use MQPMO_PASS_ALL_CONTEXT to ensure all information in that MQMD is preserved and passed on to the next application (ordinary subscriber or interceptor).

Table 9. MQMD values for republished messages

Republish message using MQPUT	Information in publication message
MQOD.ObjectString	Message property MQTopicString
MQPMO.Options should OR with the information in the message	Message property MQPubOptions

An intercepting subscriber is a normal subscriber and as such can use any of the normal publish/subscribe or WebSphere MQ functions.

A maximum of 8 intercepting applications can be implemented (with subscription levels from 9 down to 2 inclusive). In this case the final recipient of the message will have a subscription level of 1.

You can have a subscriber with subscription level 0 that serves as a catchall if no other subscriber is interested in the message. This configuration can be useful because you can monitor the messages this subscriber receives and check why no other subscribers received it and whether it is correct that it not be received by anyone else.

Retained publications

If the publication is put by the original application with put-message option MQPMO_RETAIN, it will only be retained if it is received by a subscriber with a subscription level of 1 or 0. In order to ensure that the instruction to retain this publication is preserved as the publication passes through an intercepting application, the MQPMO options are carried with the publication as a message

property and must be used on the republishing MQPUT call by the intercepting application.

Publishing options

Several options are available that control the way messages are published.

Withholding reply-to information from subscribers

If you do not want subscribers to be able to reply to publications they receive, it is possible to withhold information in the ReplyToQ and ReplyToQgr fields of the MQMD by using the MQPMO_SUPPRESS_REPLYTO put-message option. If this option is used, the queue manager removes that information from the MQMD when it receives the publication before forwarding it to any subscribers.

This option cannot be used in combination with a report option that needs a ReplyToQ, if this is attempted the call with fail with MQRC_MISSING_REPLY_TO_Q.

Publication level

Using publication levels is a way of controlling which subscribers receive the publication. The publication level denotes the level of subscription targeted by the publication. Only subscriptions with the highest subscription level less than or equal to the publication's publication level, will receive the publication. This value must be in the range zero to nine; zero is the lowest publication level. The initial value of this field is 9. One of the uses of publication and subscription levels is to intercept publications.

Subscription options

Subscriptions and message persistence

Queue managers maintain the persistence of the publications they forward to subscribers as set by the publisher, unless changed by options specified when the subscription is registered. These options are:

- Nonpersistent
- Persistent
- Persistence as queue
- Persistence as publisher (the default)

The system administrator can determine which users are allowed to have publications sent persistently.

Subscriptions and retained publications

To control when retained publications are received, subscribers can use two subscription options.

Publish on request only, MQSO_PUBLICATIONS_ON_REQUEST

If you want a subscriber to have control of when it receives publications you can use the MQSO_PUBLICATIONS_ON_REQUEST subscription option. A subscriber can then control when it receives publications by using the MQSUBRQ call

(specifying the Hsub handle that was returned from the original MQSUB call) to request that it is sent a topic's retained publication. Subscribers using the MQSO_PUBLICATIONS_ON_REQUEST subscription option, do not receive any non-retained publications.

If you specify MQSO_PUBLICATIONS_ON_REQUEST you must use MQSUBRQ to retrieve any publication. If you do not use MQSO_PUBLICATIONS_ON_REQUEST you get messages as they are published.

If a subscriber uses the MQSUBRQ call and uses wildcards in the subscription's topic, the subscription might match multiple topics or nodes on a topic tree, all of whose retained messages (if any exist) will be sent to the subscriber.

This option can be particularly helpful when used with durable subscriptions because a queue manager will continue to send publications to a subscriber if it subscribed durably even if that subscriber application is not running. This could lead to a buildup of messages on the subscriber queue. This build up can be avoided if the subscriber registers using the MQSO_PUBLICATIONS_ON_REQUEST option. Alternatively, you can use non-durable subscriptions if appropriate to your application to avoid a build up of unwanted messages.

If a subscription is durable and a publisher uses retained publications the subscriber application can use the MQSUBRQ call to refresh its state information after a restart. The subscriber must then refresh its state periodically using the MQSUBRQ call.

No publications will be sent as a result of the MQSUB call using this option. A durable subscription that has been resumed following disconnection will use the MQSO_PUBLICATIONS_ON_REQUEST option if the original subscription was configured to use this option.

New publications only, MQSO_NEW_PUBLICATIONS_ONLY

If a retained publication exists on a topic, any subscribers that make a subscription after the publication was made will receive a copy of that publication. If a subscriber does not want to receive any publications that were made prior to the subscription being made, the subscriber can use the MQSO_NEW_PUBLICATIONS_ONLY subscription option.

Grouping subscriptions

You can group subscriptions to eliminate receiving duplicate publications from overlapping subscriptions.

Think about grouping subscriptions if you have set up a queue to receive publications, and have a number of overlapping subscriptions feeding publications to the queue. This situation is similar to the example in Overlapping subscriptions.

You can eliminate receiving duplicate publications by setting the option MQSO_GROUP_SUB when you subscribe to a topic. The result is that when more than one subscription in the group matches the topic of a publication, only one subscription is responsible for placing the publication on the queue. The other subscriptions that matched the publication topic are ignored.

The subscription responsible for placing the publication on the queue is chosen on the basis that it has the longest matching topic string, before encountering any wild cards. In other words, it can be thought of as the closest matching subscription. Its properties are propagated to the publication, including whether it has the MQSO_NOT_OWN_PUBS property. If it does, no publication is delivered to the queue, even though other matching subscriptions might not have the MQSO_NOT_OWN_PUBS property.

You cannot place *all* your subscriptions in a single group to eliminate duplicate publications. The basic rules for grouping subscriptions are:

1. Only applicable to non-managed subscriptions.
2. By definition, a group of subscriptions deliver publications to one queue.
3. Each subscription must be at the same subscription level.
4. The publication message for each subscription in the group has the same *CorrelId*.

   To ensure each subscription results in a publication message with the same *CorrelId*, set MQSO_SET_CORREL_ID to create your own *CorrelId* in the publication, and set the same *SubCorrelId* in each subscription. Do not set *SubCorrelId* to the value MQCI_NONE.

Fuller details of the rules for grouping subscriptions are to be found in MQSO_GROUP_SUB.

Chapter 5. Publish/subscribe security

The components and interactions that are involved in publish/subscribe are described as an introduction to the more detailed explanations and examples that follow.

There are a number of components involved in publishing and subscribing to a topic. Some of the relationships between these components are illustrated in Figure 38 and described below.

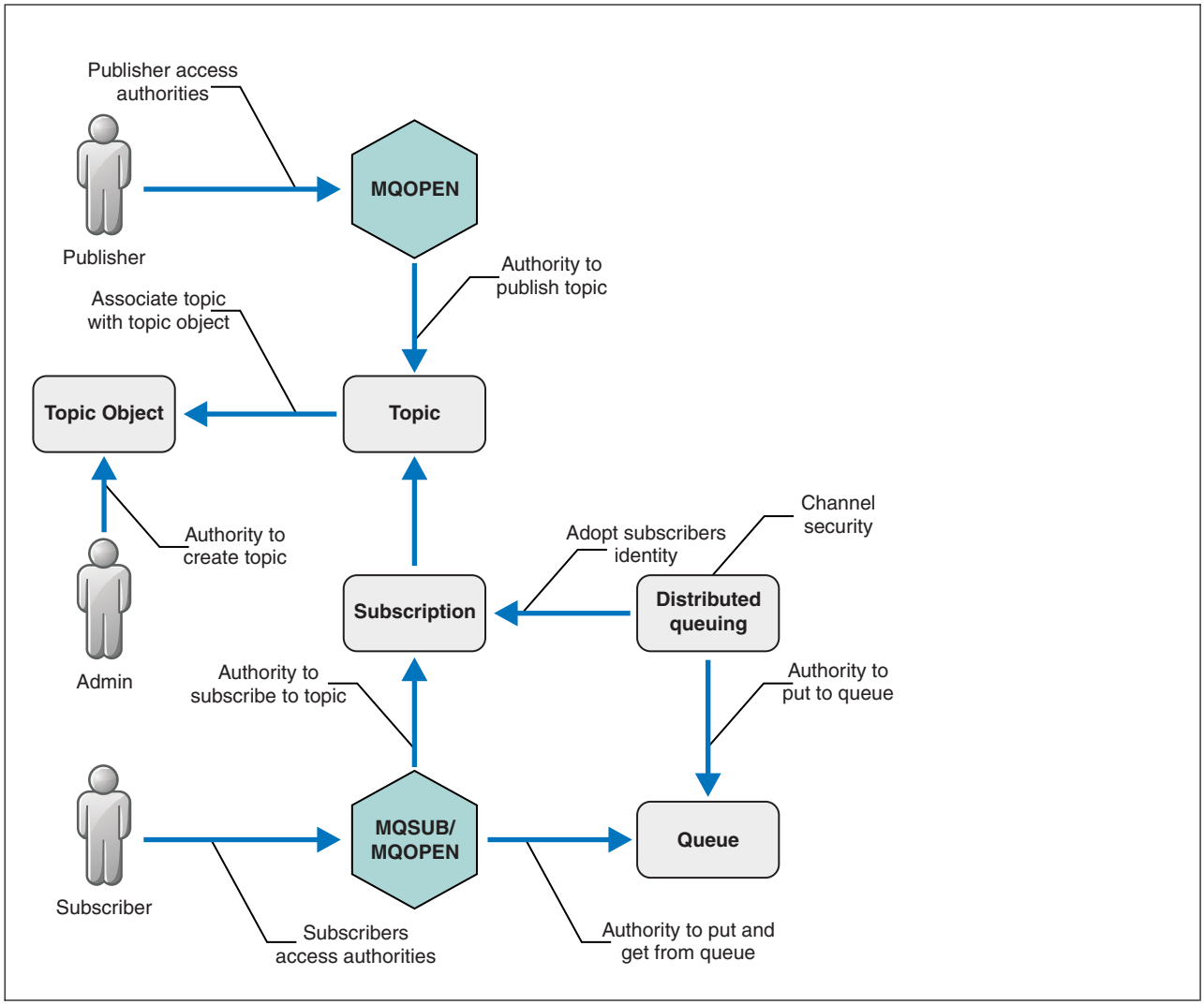


Figure 38. Publish/subscribe security relationships

"Topics" on page 10
Topics are identified by topic strings, and are usually organized into "Topic trees" on page 17. You need to associate a topic with a topic object to control access to the topic. "Topic security model" on page 90 explains how you secure topics using topic objects.

"Administrative topic objects" on page 18
You can control who has access to a topic, and for what purpose, by using

the command setmqaut against a list of administrative topic objects. See the examples, “Grant access to a user to subscribe to a topic” on page 95 and “Grant access to a user to publish to a topic” on page 101.

Subscriptions

You subscribe to one or more topics by creating a subscription supplying a topic string, which can include wild cards, to match against the topic strings of publications. “Subscribing using the topic object name” on page 91 explains how to check your authorization to use an existing topic object for the subscription, and “Subscribing using a topic string where the topic node does not exist” on page 92 the case where the topic in the subscription does not correspond to an existing topic object. If the subscription includes wild cards, “Subscribing using a topic string that contains wildcard characters” on page 92 explains the rules for associating a topic string containing wild cards with a topic object, and then applying authorization to all the matching topics that result.

A subscription also contains information about the identity of the subscriber, the identity of the destination queue onto which the publications are to be placed, and information about how the publication is to be placed on the destination queue.

As well as defining which subscribers have the authority to subscribe to certain topics, you can restrict subscriptions to being used by only one subscriber. You can also control what information about the subscriber is adopted by the queue manager when publications are placed onto the destination queue; see “Subscription security” on page 105.

Queues

The destination queue is an important queue to secure. It is local to the subscriber, and publications that matched the subscription are placed onto it. You need to consider access to the destination queue twice over.

1. From the perspective of putting a publication onto the queue
2. From the perspective of getting the publication off the queue.

In the first instance it is the queue manager using an identity provided by the subscriber that places the publication onto the queue; in the second it is either the subscriber, or a program that has been delegated the task of getting publications, that takes messages off the queue. The section, “Authority to destination queues” on page 93 provides more details.

There are no topic object aliases, but you can use an alias queue as the alias for a topic object. If you do so, then as well as checking authority to use the topic for publish or subscribe, the queue manager also checks authority to use the queue.

“Distributed publish/subscribe security” on page 29

Your permission to publish or subscribe to a topic is checked on the local queue manager using local identities and authorizations. Authorization does not depend on the whether the topic is defined or not, nor where it is defined. Consequently, you need to perform topic authorization on every queue manager in a cluster when clustered topics are used.

Note: This differs from the security model for queues; though you can achieve the same result for queues by defining a queue alias locally for every clustered queue.

Queue managers exchange subscriptions in a cluster. In most WebSphere MQ cluster configurations, channels are configured with *PUTAUT=DEF* to place messages onto target queues using the authority of the channel process. You can modify the channel configuration to use *PUTAUT=CTX* to require the subscribing user to have authority to propagate a subscription onto another queue manager in a cluster.

"Distributed publish/subscribe security" on page 29 describes how to change your channel definitions to control who is allowed to propagate subscriptions onto other servers in the cluster.

Authorization

You can apply authorization to topic objects, just like queues and other objects, and there are three authorization operations, pub, sub, and resume that you can apply only to topics. The details are described in Specifying authorities for different object types.

Function calls

In publish and subscribe programs, like in queued programs, authorization checks are made when objects are opened, created, changed or deleted in some way; checks are not made when MQPUT or MQGET MQI calls are made to put and get publications.

To publish a topic, perform an MQOPEN on the topic, which performs the authorization checks, and then publish messages to the topic using the MQPUT command, which performs no authorization checks.

To subscribe to a topic, typically you perform an MQSUB command to create or resume the subscription, and also to open the destination queue to receive publications; or you perform a separate MQOPEN to open the destination queue and then perform the MQSUB to create or resume the subscription.

Whichever calls you use, the queue manager checks that you can subscribe to the topic and get the resulting publications from the destination queue. If the destination queue is unmanaged, authorization checks are also made that the queue manager will be able to place publications on the destination queue with the identity it will adopt when a matching topic is published. It is assumed that the queue manager is always able to place publications onto unmanaged destination queues.

Roles

Users are involved in four roles in running publish/subscribe applications:

1. Publisher
2. Subscriber
3. Topic administrator
4. WebSphere MQ Administrator - member of group mqm

The WebSphere MQ administrator uses the `setmqaut -n TopicObjects -topic -g -Pubs&Subs ±pub ±sub ±resume` command to authorize the publishers and subscribers in the *Pubs&Subs* group to publish and subscribe to the topic objects listed in the *TopicObjects* profile.

The WebSphere MQ administrator might also authorize a "topic" administrator to be responsible for managing topic objects.

In addition you need to extend the security administration tasks, already performed by the WebSphere MQ administrator, to the queues and channels responsible for moving publications and subscriptions.

Topic security model

Only defined topic objects, that are specified as administration nodes, have associated security attributes. For a description of topic objects see “Administrative topic objects” on page 18. These attributes specify whether a specified user ID, or security group, is permitted to perform a subscribe or a publish operation on each topic object.

The security attributes are associated with the appropriate administration node in the topic tree. When an authority check is made for a particular user ID during a subscribe or publish operation, the authority granted is based on a set of rules dependent on the security attributes associated with the appropriate topic tree nodes.

You can represent the security attributes as an access control list, thereby indicating what authority a particular operating system user ID, or security group, has to the topic object.

Consider the following example where the topic objects have been defined with the security attributes, or authorities shown:

Table 10. Example topic object authorities

Topic name	Topic string	Authorities - not z/OS	z/OS authorities
SECROOT	SEC	none	NONE
SECGOOD	SEC/GOOD	usr1+subscribe	ALTER Hlq.SUBSCRIBE.SECGOOD
SECBAD	SEC/BAD	none	NONE Hlq.SUBSCRIBE.SECBAD
SECCOMB	SEC/COMB	none	NONE Hlq.SUBSCRIBE.SECCOMB
SECCOMBB	SEC/COMB/ GOOD/BAD	none	NONE Hlq.SUBSCRIBE.SECCOMBB
SECCOMBG	SEC/COMB/ GOOD	usr2+subscribe	ALTER Hlq.SUBSCRIBE.SECCOMBG
SECCOMBN	SEC/COMB/ BAD	none	NONE Hlq.SUBSCRIBE.SECCOMBN

The examples listed give the following authorizations:

- At the root of the tree `/SEC` no user has authority at that node.
- `usr1` has been granted subscribe authority to the object `/SEC/GOOD`
- `usr2` has been granted subscribe authority to the object `/SEC/COMB/GOOD`

The topic tree with the associated security attributes at each node can be represented as:

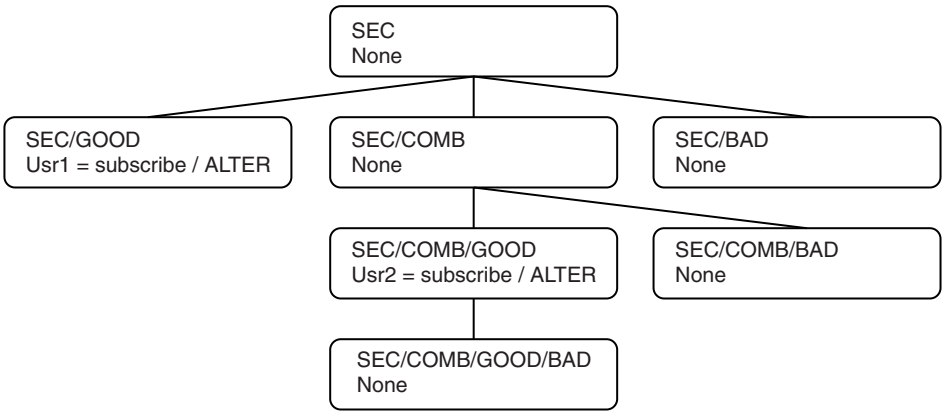


Figure 39. Example topic tree security attributes

Subscribing using the topic object name

When subscribing to a topic object by specifying the MQCHAR48 name, the corresponding node in the topic tree is located. If the security attributes associated with the topic node indicate that the user has authority to subscribe, then access is granted.

If not, the parent node in the tree is considered to determine if the user has authority to subscribe to that node. If so, then access is also granted. If not, then the parent of that node is considered, and so on, until a node is located that grants subscribe authority to the user, or until the root node is considered without authority having been granted. In the latter case, access is denied.

This means that if any node in the path grants authority to subscribe to that user or application, the subscriber is allowed to subscribe at that node, or anywhere below that node in the topic tree.

The root node in the above example is `SEC`; note, it is possible that the root node will always be a topic object.

The security attributes indicate that a particular user ID has subscribe authority, if the access control list indicates that the user ID itself has authority, or that an operating system security group of which the user ID is a member has authority.

So, for example:

- If `usr1` tries to subscribe, using a topic string of `SEC/GOOD`, the subscription would be allowed as the user ID has access to the node associated with that topic. However, if `usr1` tried to subscribe using topic string `SEC/COMB/GOOD` the subscription would not be allowed as the user ID does not have access to the node associated with it.
- If `usr2` tries to subscribe, using a topic string of `SEC/COMB/GOOD` the subscription would be allowed to as the user ID has access to the node associated with the topic. However, if `usr2` tried to subscribe to `SEC/GOOD` the subscription would not be allowed as the user ID does not have access to the node associated with it.
- If `usr2` tries to subscribe using a topic string of `SEC/COMB/GOOD/BAD` the subscription would be allowed to because the user ID has access to the parent node `SEC/COMB/GOOD`.

- If usr1 or usr2 tries to subscribe using a topic string of /SEC/COMB/BAD, neither would be allowed as they do not have access to the topic node associated with it, or the parent nodes of that topic.

A subscribe operation specifying a topic object name in the case where the topic object does not exist results in an MQRC_UNKNOWN_OBJECT_NAME error.

Subscribing using a topic string where the topic node exists

The behavior is the same as when specifying the topic by the MQCHAR48 object name.

Subscribing using a topic string where the topic node does not exist

When an application subscribes specifying a topic string representing a topic node that does not currently exist in the topic tree, the authority check is performed as outlined in the previous section, starting with the parent node of that which is represented by the topic string. If the authority is granted, a new node representing the topic string is created in the topic tree.

For example, if usr1 tries to subscribe to a topic SEC/GOOD/NEW, this would be allowed as usr1 has access to the parent node SEC/GOOD and a new topic node is created in the tree as the following diagram shows. As this is not a topic object it does not have any security attributes associated with it directly; the attributes are inherited from its parent.

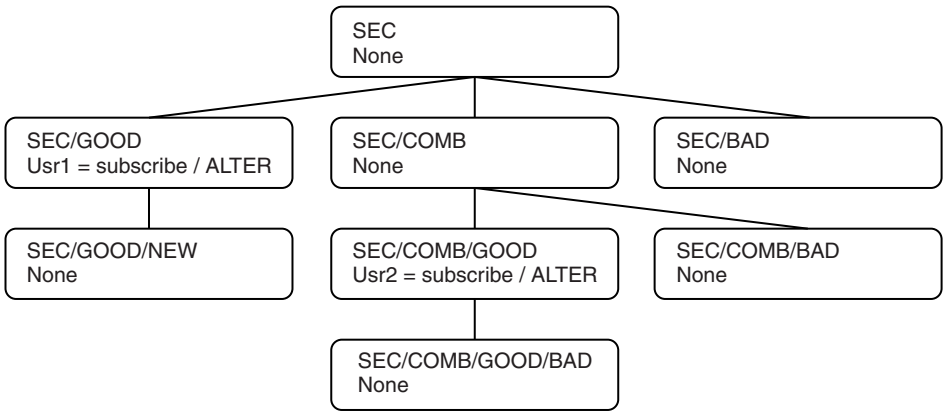


Figure 40. Example topic tree security attributes

Subscribing using a topic string that contains wildcard characters

When an application attempts to connect by specifying a topic string that contains a wildcard character, the authority check is made against the node in the topic tree that matches the fully qualified part of the topic string.

So, if an application needs to subscribe to SEC/COMB/GOOD/*, an authority check is carried out as outlined in the previous two sections on the node SEC/COMB/GOOD in the topic tree.

Similarly, if an application needs to subscribe to SEC/COMB/*/GOOD, an authority check is carried out on the node SEC/COMB.

Authority to destination queues

When subscribing to a topic, one of the parameters on the call is the handle hobj, this is:

- The hobj of a queue that has been opened for output to receive the publications.
- Blank and:
  - The MQSO_MANAGED option has been specified, and
  - The subscription does not exist, and
  - Create is specified,

  in which case a managed queue is created.
- Blank, and you are altering or resuming an existing subscription, in which the destination queue can be either a managed or non managed queue.

In each case the application or user making the MQSUB request, has to have the authority to put messages to that destination queue it has provided; in effect authority to have published messages put on that queue, in order for the subscribe request to continue. This follows the existing rules for queue security checking.

This includes Alternate user ID and Context security checks where required. In order to be able to set any of the Identity context fields you have to specify the MQSO_SET_IDENTITY_CONTEXT option as well as the MQSO_CREATE or MQSO_ALTER option. You are not allowed to set any of the Identity context fields on an MQSO_RESUME request.

If the destination is a managed queue, no security checks are performed against the managed destination. If you are allowed to subscribe to that topic the assumption is that you can use managed destinations.

Publish using the topic name or topic string where the topic node exists

The security model for the publish operation is the same as that for the subscribe operation, except that there is no wildcard character in the topic string case to consider.

The authorities to publish and subscribe are distinct; a user or group can have one authority without necessarily having the other.

When publishing to a topic object by specifying either the MQCHAR48 name or the topic string, the corresponding node in the topic tree is located. If the security attributes associated with the topic node indicates that the user has authority to publish, then access is granted.

If not, then the parent node in the tree is considered to determine if the user has authority to publish to that node. If so, then access is also granted. If not, then the parent of that node is considered, and so on until a node is located which grants publish authority to the user, or until the root node is considered without authority having been granted. In the latter case, access is denied.

This means that if any node in the path grants authority to publish to that user or application, the publisher is allowed to publish at that node or anywhere below that node in the topic tree.

Publish using the topic name or topic string where the topic node does not exist

As with the subscribe operation, when an application publishes, specifying a topic string representing a topic node that does not currently exist in the topic tree, the authority check is performed starting with the parent node of that which is represented by the topic string. If the authority is granted, a new node representing the topic string is created in the topic tree.

Publish using an alias queue that resolves to a topic object

If you publish, using an alias queue that resolves to a topic object then security checking occurs on both the open of the alias queue that you specify and the underlying topic to which it resolves.

The security check on the alias queue looks to see that the user has authority to put messages to that alias queue and the security check on the topic looks to see that the user can Publish to that topic. This is different behavior from that which takes place when an alias queue resolves to other queues.

Closing a subscription

There is additional security checking if you close a subscription using the MQCO_ SUB_ REMOVE option and you did not create the subscription under this handle.

A security check is performed to ensure that you have the correct authority to do this as the action results in the removal of the subscription.

A similar process to that used to determine if a user has the correct level of authority to subscribe to a topic is followed to determine if the user has the correct level of authority required to perform the close remove request. If the security attributes associated with the topic node indicate that the user has authority , then access is granted. If not, then the parent node in the tree is considered to determine if the user has authority to close remove that subscription and so on until either authority is granted or the root node is reached.

Note that the security check takes place during close processing.

Defining, altering, and deleting a subscription

When a subscription is created administratively, rather that through an MQSUB API request, no subscribe security checks take place to see if the subscription can be created or altered, as the administrator has already been given this authority through the command, and command resource security associated with the command.

Security checks are performed to ensure that publications can be put to the destination queue associated with the subscription in the same way they are performed for a MQSUB request.

The user ID that is used for these security checks depends upon the command being issued and the contents of the SUBUSER parameter on the command if it is

specified, as follows:

Table 11. User IDs used for security checks for commands

Command	SUBUSER specified and blank	SUBUSER specified and completed	SUBUSER not specified
DEFINE	Use the administrator ID	Use the User ID specified in SUBUSER	Use the administrator's ID
ALTER	Use the administrator ID	Use the User ID specified in SUBUSER	Use the User ID from the existing subscription

The only security check performed when deleting subscriptions using the DELETE SUB command is the command security check.

Example publish/subscribe security setup

This section describes a scenario that has access control setup on topics in a way that allows the security control to be applied as required.

Grant access to a user to subscribe to a topic

This topic is the first one in a list of tasks that tells you how to grant access to topics by more than one user.

About this task

This task assumes that no administrative topic objects exist, nor have any profiles been defined for subscription or publication. The applications are creating new subscriptions, rather than resuming existing ones, and are doing so using the topic string only.

An application can make a subscription by providing a topic object, or a topic string, or a combination of both. Whichever way the application selects, the effect is to make a subscription at a certain point in the topic tree. If this point in the topic tree is represented by an administrative topic object, a security profile is checked based on the name of that topic object.

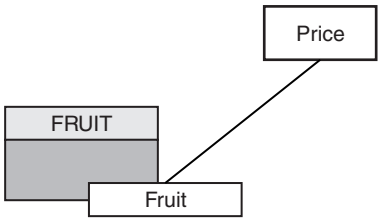


Figure 41. Topic object access example

Table 12. Example topic object access

Topic	Subscribe access required	Topic object
Price	No user	None
Price/Fruit	USER1	FRUIT

Define a new topic object as follows:

1. Issue the MQSC command `DEF TOPIC(FRUIT) TOPICSTR('Price/Fruit')`.
2. Grant access as follows:
   a. z/OS. Grant access to `USER1` to subscribe to topic `"Price/Fruit"` by granting the user access to the `hlq.SUBSCRIBE.FRUIT` profile. Do this, using the following RACF® commands:

      ```
      RDEFINE MXTOPIC hlq.SUBSCRIBE.FRUIT UACC(NONE)
      PERMIT hlq.SUBSCRIBE.FRUIT CLASS(MXTOPIC) ID(USER1) ACCESS(ALTER)
      ```

   b. Other platforms. Grant access to `USER1` to subscribe to topic `"Price/Fruit"` by granting the user access to the `FRUIT` profile. Do this, using the following setmqaut command:

      ```
      setmqaut –t topic –n FRUIT –p USER1 +sub
      ```

Results

When `USER1` attempts to subscribe to topic `"Price/Fruit"` the result is success.

When `USER2` attempts to subscribe to topic `"Price/Fruit"` the result is failure with an `MQRC_NOT_AUTHORIZED` message, together with:

- On z/OS, the following messages seen on the console that show the full security path through the topic tree that has been attempted:

  ```
  ICH408I USER(USER2   ) ...
    hlq.SUBSCRIBE.FRUIT ...

  ICH408I USER(USER2   ) ...
    hlq.SUBSCRIBE.SYSTEM.BASE.TOPIC ...
  ```

- On other platforms, the following authorization event:

  ```
  MQRC_NOT_AUTHORIZED
  ReasonQualifier   MQRQ_SUB_NOT_AUTHORIZED
  UserIdentifier     USER2
  AdminTopicNames   FRUIT, SYSTEM.BASE.TOPIC
  TopicString     "Price/Fruit"
  ```

Note that this is an illustration of what you see; not all the fields.

Grant access to a user to subscribe to a topic deeper within the tree

This topic is the second in a list of tasks that tells you how to grant access to topics by more than one user.

Before you begin

This topic uses the setup described in “Grant access to a user to subscribe to a topic” on page 95.

About this task

If the point in the topic tree where the application makes the subscription is not represented by an administrative topic object, move up the tree until the closest parent administrative topic object is located. The security profile is checked, based on the name of that topic object.

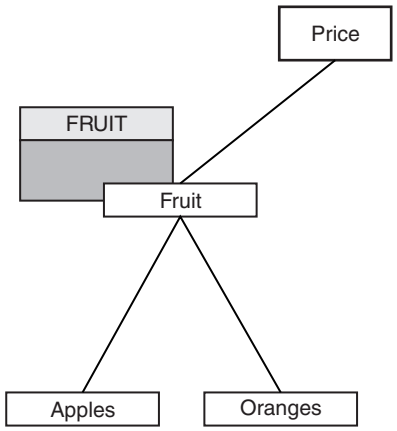


Figure 42. Example of granting access to a topic within a topic tree

Table 13. Access requirements for example topics and topic objects

Topic	Subscribe access required	Topic object
Price	No user	None
Price/Fruit	USER1	FRUIT
Price/Fruit/Apples	USER1	
Price/Fruit/Oranges	USER1	

In the previous task USER1 was granted access to subscribe to topic "Price/Fruit" by granting it access to the hlq.SUBSCRIBE.FRUIT profile on z/OS and subscribe access to the FRUIT profile on other platforms. This single profile also grants USER1 access to subscribe to "Price/Fruit/Apples", "Price/Fruit/Oranges" and "Price/Fruit/#".

When USER1 attempts to subscribe to topic "Price/Fruit/Apples" the result is success.

When USER2 attempts to subscribe to topic "Price/Fruit/Apples" the result is failure with an MQRC_NOT_AUTHORIZED message, together with:

- On z/OS, the following messages seen on the console that show the full security path through the topic tree that has been attempted:

```
ICH408I USER(USER2   ) ...
  hlq.SUBSCRIBE.FRUIT ...

ICH408I USER(USER2   ) ...
  hlq.SUBSCRIBE.SYSTEM.BASE.TOPIC ...
```

- On other platforms, the following authorization event:

```
MQRC_NOT_AUTHORIZED
ReasonQualifier   MQRQ_SUB_NOT_AUTHORIZED
UserIdentifier     USER2
AdminTopicNames   FRUIT, SYSTEM.BASE.TOPIC
TopicString     "Price/Fruit/Apples"
```

Note the following:

- The messages you receive on z/OS are identical to those received in the previous task as the same topic objects and profiles are controlling the access.

- The event message you receive on other platforms is similar to the one received in the previous task, but the actual topic string is different.

Grant another user access to subscribe to only the topic deeper within the tree

This topic is the third in a list of tasks that tells you how to grant access to subscribe to topics by more than one user.

Before you begin

This topic uses the setup described in "Grant access to a user to subscribe to a topic deeper within the tree" on page 96.

About this task

In the previous task `USER2` was refused access to topic `"Price/Fruit/Apples"`. This topic tells you how to grant access to that topic, but not to any other topics.

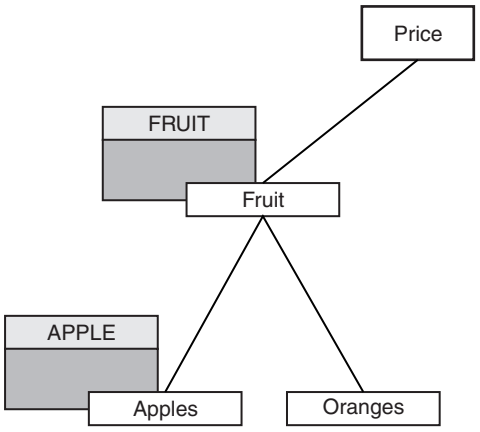


Figure 43. Granting access to specific topics within a topic tree

Table 14. Access requirements for example topics and topic objects

Topic	Subscribe access required	Topic object
Price	No user	None
Price/Fruit	USER1	FRUIT
Price/Fruit/Apples	USER1 and USER2	APPLE
Price/Fruit/Oranges	USER1	

Define a new topic object as follows:

1. Issue the MQSC command `DEF TOPIC(APPLE) TOPICSTR('Price/Fruit/Apples')`.
2. Grant access as follows:
   a. z/OS.

      In the previous task `USER1` was granted access to subscribe to topic `"Price/Fruit/Apples"` by granting the user access to the `hlq.SUBSCRIBE.FRUIT` profile.

This single profile also granted USER1 access to subscribe to "Price/Fruit/Oranges" "Price/Fruit/#" and this access remains even with the addition of the new topic object and the profiles associated with it.

Grant access to USER2 to subscribe to topic "Price/Fruit/Apples" by granting the user access to the hlq.SUBSCRIBE.APPLE profile. Do this, using the following RACF commands:

```
RDEFINE MXTOPIC hlq.SUBSCRIBE.APPLE UACC(NONE)
PERMIT hlq.SUBSCRIBE.FRUIT APPLE(MXTOPIC) ID(USER2) ACCESS(ALTER)
```

b. Other platforms.

In the previous task USER1 was granted access to subscribe to topic "Price/Fruit/Apples" by granting the user subscribe access to the FRUIT profile.

This single profile also granted USER1 access to subscribe to "Price/Fruit/Oranges" and "Price/Fruit/#", and this access remains even with the addition of the new topic object and the profiles associated with it.

Grant access to USER2 to subscribe to topic "Price/Fruit/Apples" by granting the user subscribe access to the APPLE profile. Do this, using the following setmqaut command:

```
setmqaut –t topic –n APPLE –p USER2 +sub
```

Results

On z/OS, when USER1 attempts to subscribe to topic "Price/Fruit/Apples" the first security check on the hlq.SUBSCRIBE.APPLE profile fails, but on moving up the tree the hlq.SUBSCRIBE.FRUIT profile allows USER1 to subscribe, so the subscription succeeds and no return code is sent to the MQSUB call. However, a RACF ICH message is generated for the first check:

```
ICH408I USER(USER1   ) ...
  hlq.SUBSCRIBE.APPLE ...
```

When USER2 attempts to subscribe to topic "Price/Fruit/Apples" the result is success because the security check passes on the first profile.

When USER2 attempts to subscribe to topic "Price/Fruit/Oranges" the result is failure with an MQRC_NOT_AUTHORIZED message, together with:

- On z/OS, the following messages seen on the console that show the full security path through the topic tree that has been attempted:

```
ICH408I USER(USER2   ) ...
  hlq.SUBSCRIBE.FRUIT ...

ICH408I USER(USER2   ) ...
  hlq.SUBSCRIBE.SYSTEM.BASE.TOPIC ...
```

- On other platforms, the following authorization event:

```
MQRC_NOT_AUTHORIZED
ReasonQualifier   MQRQ_SUB_NOT_AUTHORIZED
UserIdentifier     USER2
AdminTopicNames   FRUIT, SYSTEM.BASE.TOPIC
TopicString     "Price/Fruit/Oranges"
```

The disadvantage of this setup is that, on z/OS, you receive additional ICH messages on the console. You can avoid this if you secure the topic tree in a different manner.

Change access control to avoid additional messages

This topic is the fourth in a list of tasks that tells you how to grant access to subscribe to topics by more than one user and to avoid additional RACF ICH408I messages on z/OS.

Before you begin

This topic enhances the setup described in "Grant another user access to subscribe to only the topic deeper within the tree" on page 98 so that you avoid additional error messages.

About this task

This topic tells you how to grant access to topics deeper in the tree, and how to remove access to the topic lower down the tree when no user requires it.

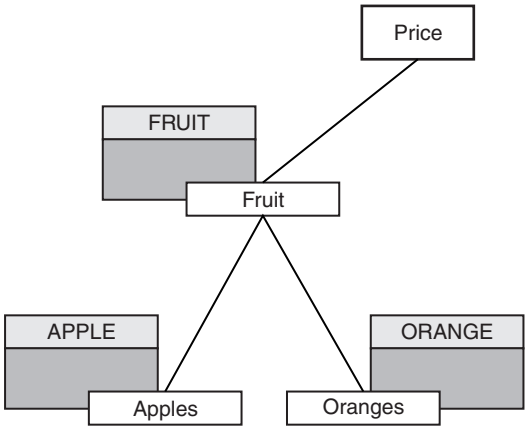


Figure 44. Example of granting access control to avoid additional messages.

Define a new topic object as follows:

1. Issue the MQSC command `DEF TOPIC(ORANGE) TOPICSTR('Price/Fruit/Oranges')`.
2. Grant access as follows:
   a. z/OS.

      Define a new profile and add access to that profile, and the existing profiles. Do this, using the following RACF commands:

      ```
      RDEFINE MXTOPIC hlq.SUBSCRIBE.ORANGE UACC(NONE)
      PERMIT hlq.SUBSCRIBE.ORANGE CLASS(MXTOPIC) ID(USER1) ACCESS(ALTER)
      PERMIT hlq.SUBSCRIBE.APPLE CLASS(MXTOPIC) ID(USER1) ACCESS(ALTER)
      ```

   b. Other platforms.

      Setup the equivalent access by using the following setmqaut commands:

      ```
      setmqaut –t topic –n ORANGE –p USER1 +sub
      setmqaut -t topic -n APPLE -p USER1 +sub
      ```

Results

On z/OS, when `USER1` attempts to subscribe to topic `"Price/Fruit/Apples"` the first security check on the `hlq.SUBSCRIBE.APPLE` profile succeeds.

Similarly, when USER2 attempts to subscribe to topic "Price/Fruit/Apples" the result is success because the security check passes on the first profile.

When USER2 attempts to subscribe to topic "Price/Fruit/Oranges" the result is failure with an MQRC_NOT_AUTHORIZED message, together with:

- On z/OS, the following messages seen on the console that show the full security path through the topic tree that has been attempted:

```
ICH408I USER(USER2   ) ...
  hlq.SUBSCRIBE.ORANGE ...

ICH408I USER(USER2   ) ...
  hlq.SUBSCRIBE.FRUIT ...

ICH408I USER(USER2   ) ...
  hlq.SUBSCRIBE.SYSTEM.BASE.TOPIC ...
```

- On other platforms, the following authorization event:

```
MQRC_NOT_AUTHORIZED
ReasonQualifier   MQRQ_SUB_NOT_AUTHORIZED
UserIdentifier     USER2
AdminTopicNames   ORANGE, FRUIT, SYSTEM.BASE.TOPIC
TopicString     "Price/Fruit/Oranges"
```

Grant access to a user to publish to a topic

This topic is the first one in a list of tasks that tells you how to grant access to publish topics by more than one user.

About this task

This task assumes that no administrative topic objects exist on the right hand side of the topic tree, nor have any profiles been defined for publication. The assumption used is that publishers are using the topic string only.

An application can publish to a topic by providing a topic object, or a topic string, or a combination of both. Whichever way the application selects, the effect is to publish at a certain point in the topic tree. If this point in the topic tree is represented by an administrative topic object, a security profile is checked based on the name of that topic object. For example:

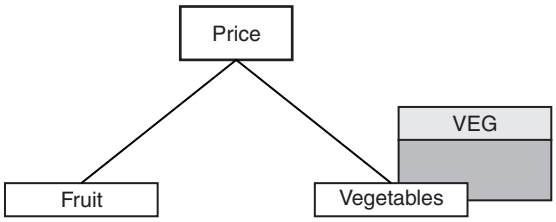


Figure 45. Granting publish access to a topic

Table 15. Example publish access requirements

Topic	Publish access required	Topic object
Price	No user	None
Price/Vegetables	USER1	VEG

Define a new topic object as follows:

1. Issue the MQSC command `DEF TOPIC(VEG) TOPICSTR('Price/Vegetables')`.
2. Grant access as follows:
   a. z/OS. Grant access to `USER1` to publish to topic `"Price/Vegetables"` by granting the user access to the `hlq.PUBLISH.VEG` profile. Do this, using the following RACF commands:

      ```
      RDEFINE MXTOPIC hlq.PUBLISH.VEG UACC(NONE)
      PERMIT hlq.PUBLISH.VEG CLASS(MXTOPIC) ID(USER1) ACCESS(UPDATE)
      ```

   b. Other platforms. Grant access to `USER1` to publish to topic `"Price/Vegetables"` by granting the user access to the `VEG` profile. Do this, using the following setmqaut command:

      ```
      setmqaut –t topic –n VEG –p USER1 +pub
      ```

Results

When `USER1` attempts to publish to topic `"Price/Vegetables"` the result is success; that is, the MQOPEN call succeeds.

When `USER2` attempts to publish to topic `"Price/Vegetables"` the MQOPEN call fails with an `MQRC_NOT_AUTHORIZED` message, together with:

- On z/OS, the following messages seen on the console that show the full security path through the topic tree that has been attempted:

  ```
  ICH408I USER(USER2   ) ...
    hlq.PUBLISH.VEG ...

  ICH408I USER(USER2   ) ...
    hlq.PUBLISH.SYSTEM.BASE.TOPIC ...
  ```

- On other platforms, the following authorization event:

  ```
  MQRC_NOT_AUTHORIZED
  ReasonQualifier   MQRQ_OPEN_NOT_AUTHORIZED
  UserIdentifier     USER2
  AdminTopicNames   VEG, SYSTEM.BASE.TOPIC
  TopicString     "Price/Vegetables"
  ```

Note that this is an illustration of what you see; not all the fields.

Grant access to a user to publish to a topic deeper within the tree

This topic is the second in a list of tasks that tells you how to grant access to publish to topics by more than one user.

Before you begin

This topic uses the setup described in “Grant access to a user to publish to a topic” on page 101.

About this task

If the point in the topic tree where the application publishes is not represented by an administrative topic object, move up the tree until the closest parent administrative topic object is located. The security profile is checked, based on the name of that topic object.

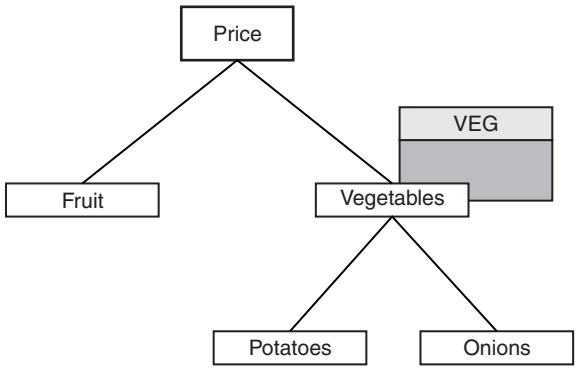


Figure 46. Granting publish access to a topic within a topic tree

Table 16. Example publish access requirements

Topic	Subscribe access required	Topic object
Price	No user	None
Price/Vegetables	USER1	VEG
Price/Vegetables/Potatoes	USER1	
Price/Vegetables/Onions	USER1	

In the previous task USER1 was granted access to publish topic "Price/Vegetables/Potatoes" by granting it access to the hlq.PUBLISH.VEG profile on z/OS or publish access to the VEG profile on other platforms. This single profile also grants USER1 access to publish at"Price/Vegetables/Onions".

When USER1 attempts to publish at topic "Price/Vegetables/Potatoes" the result is success; that is the MQOPEN call succeeds.

When USER2 attempts to subscribe to topic "Price/Vegetables/Potatoes" the result is failure; that is, the MQOPEN call fails with an MQRC_NOT_AUTHORIZED message, together with:

- On z/OS, the following messages seen on the console that show the full security path through the topic tree that has been attempted:

```
ICH408I USER(USER2   ) ...
  hlq.PUBLISH.VEG ...

ICH408I USER(USER2   ) ...
  hlq.PUBLISH.SYSTEM.BASE.TOPIC ...
```

- On other platforms, the following authorization event:

```
MQRC_NOT_AUTHORIZED
ReasonQualifier   MQRQ_OPEN_NOT_AUTHORIZED
UserIdentifier     USER2
AdminTopicNames   VEG, SYSTEM.BASE.TOPIC
TopicString     "Price/Vegetables/Potatoes"
```

Note the following:

- The messages you receive on z/OS are identical to those received in the previous task as the same topic objects and profiles are controlling the access.
- The event message you receive on other platforms is similar to the one received in the previous task, but the actual topic string is different.

Grant access for publish and subscribe

This topic is the last in a list of tasks that tells you how to grant access to publish and subscribe to topics by more than one user.

Before you begin

This topic uses the setup described in "Grant access to a user to publish to a topic deeper within the tree" on page 102.

About this task

In a previous task `USER1` was given access to subscribe to the topic `"Price/Fruit"`. This topic tells you how to grant access to that user to publish to that topic.

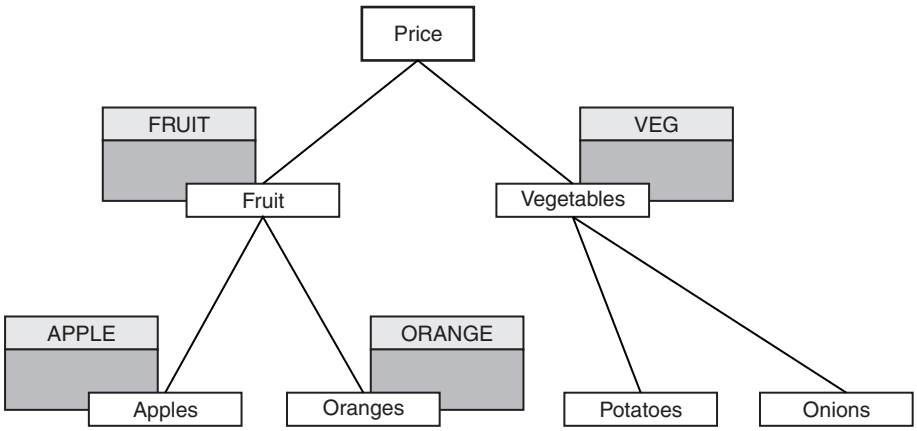


Figure 47. Granting access for publishing and subscribing

Table 17. Example publishing and subscribing access requirements

Topic	Subscribe access required	Publish access required	Topic object
Price	No user	No user	None
Price/Fruit	USER1	USER1	FRUIT
Price/Fruit/Apples	USER1 and USER2		APPLE
Price/Fruit/Oranges	USER1		ORANGE

Grant access as follows:

1. z/OS.

   In an earlier task `USER1` was granted access to subscribe to topic `"Price/Fruit"` by granting the user access to the `hlq.SUBSCRIBE.FRUIT` profile.

   In order to publish to the `"Price/Fruit"` topic, grant access to `USER1` to the `hlq.PUBLISH.FRUIT` profile. Do this, using the following RACF commands:

   ```
   RDEFINE MXTOPIC hlq.PUBLISH.FRUIT UACC(NONE)
   PERMIT hlq.PUBLISH.FRUIT CLASS(MXTOPIC) ID(USER1) ACCESS(ALTER)
   ```

2. Other platforms.

Grant access to USER1 to publish to topic "Price/Fruit" by granting the user publish access to the FRUIT profile. Do this, using the following setmqaut command:

```
setmqaut –t topic –n FRUIT –p USER1 +pub
```

Results

On z/OS, when USER1 attempts to publish to topic "Price/Fruit" the security check on the MQOPEN call passes.

When USER2 attempts to publish at topic "Price/Fruit" the result is failure with an MQRC_NOT_AUTHORIZED message, together with:

- On z/OS, the following messages seen on the console that show the full security path through the topic tree that has been attempted:

```
ICH408I USER(USER2   ) ...
  hlq.PUBLISH.FRUIT ...

ICH408I USER(USER2   ) ...
  hlq.PUBLISH.SYSTEM.BASE.TOPIC ...
```

- On other platforms, the following authorization event:

```
MQRC_NOT_AUTHORIZED
ReasonQualifier   MQRQ_OPEN_NOT_AUTHORIZED
UserIdentifier     USER2
AdminTopicNames   FRUIT, SYSTEM.BASE.TOPIC
TopicString     "Price/Fruit"
```

Following the complete set of these tasks, gives USER1 and USER2 the following access authorities for publish and subscribe to the topics listed:

Table 18. Complete list of access authorities resulting from security examples

Topic	Subscribe access required	Publish access required	Topic object
Price	No user	No user	None
Price/Fruit	USER1	USER1	FRUIT
Price/Fruit/Apples	USER1 and USER2		APPLE
Price/Fruit/Oranges	USER1		ORANGE
Price/Vegetables		USER1	VEG
Price/Vegetables/ Potatoes			
Price/Vegetables/ Onions			

Where you have different requirements for security access at different levels within the topic tree, careful planning ensures that you do not receive extraneous security warnings on the z/OS console log. Setting up security at the correct level within the tree avoids misleading security messages.

Subscription security

MQSO_ALTERNATE_USER_AUTHORITY

The AlternateUserId field contains a user identifier to use to validate this MQSUB call. The call can succeed only if this AlternateUserId is authorized to subscribe to

the topic with the specified access options, regardless of whether the user identifier under which the application is running is authorized to do so.

MQSO_SET_IDENTITY_CONTEXT

The subscription is to use the accounting token and application identity data supplied in the PubAccountingToken and PubApplIdentityData fields.

If this option is specified, the same authorization check is carried out as if the destination queue was accessed using an MQOPEN call with MQOO_SET_IDENTITY_CONTEXT, except in the case where the MQSO_MANAGED option is also used in which case there is no authorization check on the destination queue.

If this option is not specified, the publications sent to this subscriber will have default context information associated with them as follows:

Table 19. Default publication context information

Field in MQMD	Value used
UserIdentifier	The user id associated with the subscription (see SUBUSER field on DISPLAY SBSTATUS) at the time the publication is made.
AccountingToken	Determined from the environment if possible; set to MQACT_NONE otherwise.
ApplIdentityData	Set to blanks.

This option is only valid with MQSO_CREATE and MQSO_ALTER. If used with MQSO_RESUME, the PubAccountingToken and PubApplIdentityData fields are ignored, so this option has no effect.

If a subscription is altered without using this option where previously the subscription had supplied identity context information, default context information will be generated for the altered subscription.

If a subscription allowing different user ids to use it with option MQSO_ANY_USERID, is resumed by a different user ID, default identity context will be generated for the new user ID now owning the subscription and any subsequent publications will be delivered containing the new identity context.

AlternateSecurityId

This is a security identifier that is passed with the AlternateUserId to the authorization service to allow appropriate authorization checks to be performed. AlternateSecurityId is used only if MQSO_ALTERNATE_USER_AUTHORITY is specified, and the AlternateUserId field is not entirely blank up to the first null character or the end of the field.

MQSO_ANY_USERID subscription option

When MQSO_ANY_USERID is specified, the identity of the subscriber is not restricted to a single userid. This allows any user to alter or resume the subscription when they have suitable authority. Only a single user may have the subscription at any one time. An attempt to resume use of a subscription currently in use by another application will cause the call to fail with MQRC_SUBSCRIPTION_IN_USE.

To add this option to an existing subscription the MQSUB call (using MQSO_ALTER) must come from the same userid as the original subscription.

If an MQSUB call refers to an existing subscription with MQSO_ANY_USERID set, and the userid differs from the original subscription, the call succeeds only if the new userid has authority to subscribe to the topic. After successful completion, future publications to this subscriber are put to the subscriber's queue with the new userid set in the publication.

MQSO_FIXED_USERID

When MQSO_FIXED_USERID is specified, the subscription can only be altered or resumed by a single owning userid. This userid is the last userid to alter the subscription that set this option, thereby removing the MQSO_ANY_USERID option, or if no alters have taken place, it is the userid that created the subscription.

If an MQSUB verb refers to an existing subscription with MQSO_ANY_USERID set and alters the subscription (using MQSO_ALTER) to use option MQSO_FIXED_USERID, the userid of the subscription is now fixed at this new user id. The call succeeds only if the new userid has authority to subscribe to the topic.

If a userid other than the one recorded as owning a subscription trys to resume or alter an MQSO_FIXED_USERID subscription, the call will fail with MQRC_IDENTITY_MISMATCH. The owning user id of a subscription can be viewed using the DISPLAY SBSTATUS command.

If neither MQSO_ANY_USERID or MQSO_FIXED_USERID is specified, the default is MQSO_FIXED_USERID.

Chapter 6. Queued publish/subscribe compatibility

WebSphere MQ version 6 publish/subscribe and Message and Event Broker version 6 publish/subscribe coexist and interoperate with version 7 publish/subscribe, with some restrictions. WebSphere MQ version 6 publish/subscribe is deprecated in version 7.

WebSphere® MQ Version 7.0 provides publish/subscribe function that is integrated into the queue manager. Applications use new MQI verbs to define topics and subscriptions, and to publish and subscribe. There are also new administrative commands to define topics and subscriptions, and to organize publish/subscribe into clusters and hierarchies.

Earlier versions of publish/subscribe were not integrated into the queue manager, but were implemented by separate publish/subscribe brokers. Unlike integrated publish/subscribe, which extends the normal MQI verbs to perform publish/subscribe, in earlier versions of WebSphere MQ you place messages containing `MQRFH1`, `MQRFH2` or PCF commands onto special queues to communicate with the publish/subscribe broker. This is known as ″queued″ publish/subscribe.

Documentation of version 6 publish/subscribe is available with WebSphere MQ Version 6, or free to download as an information center or a PDF file from WebSphere MQ Version 6 Publish/Subscribe User's Guide.

You can continue to run queued publish/subscribe alongside integrated publish/subscribe.

- You can publish and subscribe on one system to topics defined on a different system; for example, you can write a new WebSphere MQ Version 7 publish/subscribe application that uses topics defined using the WebSphere Version 6 queued publish/subscribe system. This is known as publish/subscribe interoperability. It is discussed further in the topic, "Interoperation with queued publish/subscribe" on page 112.
- You can run new WebSphere MQ Version 7 publish/subscribe applications alongside existing queued publish/subscribe programs on a version 7 queue manager. This is called coexistance, and it is discussed further in the topic "Coexistence with queued publish/subscribe" on page 111.
- If you use the publish/subscribe broker that is part of WebSphere Message Broker V6.1 to run your queued publish/subscribe applications, or the publish/subscribe broker that is part of the earlier members of the WebSphere Message and Event broker family, you can continue to do so should you choose to run the broker on WebSphere MQ Version 7. Queued publish/subscribe programs running on WebSphere Message and Event Broker can coexist with integrated publish/subscribe programs running on the same WebSphere MQ Version 7 queue manager. This is described in the Coexistance topic.
- If you have upgraded your queue manager from version 6 to version 7, you must migrate the version 6 publish/subscribe broker to version 7 to continue to run your queued publish/subscribe applications on the version 7 queue manager. This is described in Strmqbrk (Migrate WebSphere MQ Version 6.0 broker to Version 7.0).
- The commands to manage the integrated publish/subscribe broker in WebSphere Version 7 have changed. The new commands are described in the topic "Controlling queued publish/subscribe" on page 118.

- JMS publish/subscribe applications, by default, use integrated publish/subscribe in WebSphere MQ Version 7. This is called normal mode, and selected by setting PROVIDERVERSION=7. The selection of transport for JMS publish/subscribe is described in Rules for selecting the WebSphere MQ messaging provider mode.

Queued publish/subscribe systems

The queued publish/subscribe systems that work with version 7 publish/subscribe are listed below.

SupportPac MA0C
MA0C was a fully supported SupportPac. It added publish/subscribe messaging to WebSphere MQ. It is known as "queued" publish/subscribe, or "command-based" publish/subscribe because it uses MQRFH1 and PCF commands to register publications and subscriptions. It performs these and other publish/subscribe functions by implementing a WebSphere MQ publish/subscribe Broker. MA0C was integrated into WebSphere MQ 5.3 in CSD08, and was superseded by WebSphere MQ Version 6.0.

SupportPac MA88
MA88 provided JMS support for WebSphere MQ, including its publish/subscribe messaging programming interface. MA0C was one option to implement the publish/subscribe interface. You could also use the WebSphere Event or Integration brokers as the publish/subscribe engine. MA88 was superseded by WebSphere MQ 5.3.

WebSphere MQ Version 6 (queued) publish/subscribe
Websphere MQ version 6 superseded MA0C, and supported the JMS and AMI publish/subscribe interfaces in addition to the MQRFH1 and PCF command based programming interfaces.

SupportPac MS0Q
MS0Q extended the IBM WebSphere MQ Version 6 Explorer to provide a topic based view of the WebSphere MQ V6 publish/subscribe broker. MS0Q has been superseded by the WebSphere MQ Version 7 Explorer, which browses topics defined in both version 6 and 7.

WebSphere Message Broker Version 6 publish/subscribe
WebSphere Message Broker Version 6, and previous and related products, include a variety message transports that provide publish/subscribe support. The message broker enterprise messaging transport includes support for the MA0C and WebSphere Version 6 queued publish/subscribe.

Migration

Because queued publish/subscribe coexists and interoperates with the integrated publish/subscribe in WebSphere MQ Version 7, you can undertake migration step by step. Migration is discussed in the topic, Migration to publish/subscribe on WebSphere MQ V7.0. A sequence of steps you might take are listed below.

1. Install a version 7 queue managers, enable queued mode publish/subscribe, add the new queue manager to your existing version 6 broker hierarchy and run your existing queued publish/subscribe programs on the version 7 queue manager.
2. Upgrade your existing version 6 queue managers to version 7, run strmqbrk to migrate the broker to version 7, and run your existing queued publish/subscribe applications.

3. Write new, or migrate existing publish/subscribe programs using the integrated publish/subscribe MQI interface. Guidance on migrating version 6 applications is provided in Publish/subscribe command messages. Guidance on writing new publish/subscribe applications is provided in Chapter 4, "Writing publish/subscribe applications," on page 45.
4. Migrate broker hierarchies to clusters, to remove dependence on running the queued publish/subscribe broker, and to improve performance by connecting queue managers directly, rather than through a indirect hierarchical network of brokers. Topology migration is described in WebSphere MQ publish/subscribe topology migration.

Coexistence with queued publish/subscribe

Coexistence is controlled with a new queue manager attribute, PSMODE. Using this attribute you can continue to run existing publish/subscribe applications without modification, and at the same time run new publish/subscribe programs that use the WebSphere MQ Version 7 publish/subscribe interface.

Programs written to the WebSphere MQ Version 7 publish/subscribe interface run only on the Version 7 release of WebSphere MQ, and above.

Programs written to the version 6 queued publish/subscribe interfaces run either on version 7 with the queue manager attribute **PSMODE** set to ENABLED, or on the WebSphere Message or Event Broker Version 6.1 publish/subscribe broker with the queue manager attribute **PSMODE** set to COMPAT.

Note: If the queue manager is providing enterprise messaging support for the WebSphere Message or Event Broker, then queued publish/subscribe programs must use the publish/subscribe broker provided by the WebSphere Message or Event Broker. New publish/subscribe programs can use version 7 integrated publish/subscribe on the same queue manager. This has implications for interoperability between a version 6 and version 7 publish/subscribe program running on the same queue manager as a message broker. See the topic, "Interoperation with queued publish/subscribe" on page 112, for more details.

The PSMODE queue manager attribute controls which kind of publish/subscribe programs run on a version 7 queue manager. It has three modes as described below.

PSMODE
Controls whether the publish/subscribe engine and the queued publish/subscribe interface are running, and therefore controls whether applications can publish or subscribe by using the application programming interface and the queues that are monitored by the queued publish/subscribe interface.

COMPAT
The publish/subscribe engine is running. It is therefore possible to publish or subscribe by using the application programming interface.

The queued publish/subscribe interface is not running. Any publish/subscribe messages put to the queues that are monitored by the queued publish/subscribe interface will not be acted upon.

Use this setting for compatibility with WebSphere Message Broker V6 or earlier versions that use this queue manager, because

WebSphere Message Broker needs to read the same queues from which the queued publish/subscribe interface would normally read.

DISABLED
The publish/subscribe engine and the queued publish/subscribe interface are not running. It is therefore not possible to publish or subscribe by using the application programming interface. Any publish/subscribe messages put to the queues that are monitored by the queued publish/subscribe interface will not be acted upon.

ENABLED
The publish/subscribe engine and the queued publish/subscribe interface are running. It is therefore possible to publish or subscribe by using the application programming interface and the queues that are being monitored by the queued publish/subscribe interface. This is the queue manager's initial default value.

Note: Changing the PSMODE attribute can change the PSMODE status. Use DISPLAY PUBSUB, or on i5/OS DSPMQM, to determine the current state of the publish/subscribe engine and the queued publish/subscribe interface.

Interoperation with queued publish/subscribe

The WebSphere MQ publish/subscribe brokers need to exchange publications and subscriptions when they are connected together. There are some differences in the way the queued and integrated publish/subscribe interfaces and brokers work, and what can be connected together, and you need to take these differences into account when writing applications and administering publish/subscribe brokers that interoperate.

An example of the interoperation of publish/subscribe is a results service. The subscriber, using MQSUB and running on a WebSphere MQ version 7 queue manager, *subscribes* to the `Soccer/Scores` topic in the results service that is defined on a WebSphere MQ Version 6 queue manager. The soccer results service publisher is an existing application, written to the WebSphere MQ version 6 publish/subscribe interface. The publisher is running on WebSphere MQ Version 6 and *publishes* to the `Soccer/Scores` topic. WebSphere MQ Version 6 sends the publication to the subscriber's queue on WebSphere MQ Version 7. Two different kinds of publish/subscribe broker are involved. They share the same topic space, and publications and subscriptions flow between them.

Another example would be to migrate the Version 6 queue manager to version 7. The publisher application continues to run unchanged in queued mode on the migrated queue manager. It interoperates with the version 7 subscriber that runs as a WebSphere MQ Version 7 publish/subscribe application.

Differences from WebSphere MQ Version 6 publish/subscribe

Queued publish/subscribe programs and queued broker administration differ in version 7 from version 6. The differences in program behavior are slight; the administration differences are more extensive. Many version 6 programs coexist and interoperate with version 7, without change.

There are many changes in the implementation of version 7 queued publish/subscribe from the implementation in version 6. Queued

publish/subscribe in version 7 makes use of integrated publish/subscribe. Only the changes that result, that might affect your applications or administration procedures are described here. The use of queued publish/subscribe is deprecated in version 7.

Administration

The version 6 WebSphere MQ publish/subscribe broker is integrated into version 7 publish/subscribe. The commands you used to control a version 6 publish/subscribe broker are obsolete, and replace by commands to control version 7 publish/subscribe. The new commands, as they relate to controlling queued publish/subscribe, are described in the topic "Controlling queued publish/subscribe" on page 118. Table 20 relates the old and new commands.

Table 20. Broker command differences

Operation	WebSphere MQ Version 6	WebSphere MQ Version 7
Remove broker from hierarchy	clrmqbrk	See "Disconnect a queue manager from a broker hierarchy" on page 42 for instructions how to disconnect a version 7 queue manager from a hierarchy.
Delete broker	dltmqbrk	There is no publish/susbcribe broker in version 7. The dltmqbrk command removes version 6 broker resources after running the command strmqbrk to migrate the version 6 broker to version 7 publish/subscribe.
Display broker	dspmqbrk	Use the mqsc command DISPLAY PUBSUB to display publish/subscribe status.
Stop broker	endmqbrk	See "Stopping queued publish/subscribe" on page 120 for instructions how to stop queued publish/subscribe in version 7.
Migrate broker to WebSphere Message Broker	migmqbrk	You should run the migmqbrk command on a version 6 queue manager. Once you have upgraded to Version 7 there is *no migration* from version 7 publish/subscribe to the version 6.1 WebSphere Message Broker publish/subscribe broker.
Start broker	strmqbrk	In version 7 the strmqbrk command migrates the version 6 broker to version 7. See "Starting queued publish/subscribe" on page 119 for instructions how to start queued publish/subscribe in version 7.

Configuration data

The broker stanza parameters in WebSphere MQ Version 6 publish/subscribe are described in Broker configuration stanza. They are replaced with queue manager attributes, which are described in New queue manager attributes for publish/subscribe.

Interoperation with WebSphere Message Broker

You cannot connect the WebSphere Message Broker V6.1 publish/subscribe broker to WebSphere MQ Version 7 using clusters or broker hierarchies. Messages flow between the broker and WebSphere MQ, but because no cluster or hierarchy connection can be established, subscriptions and publications are not passed directly between the broker publication node and the WebSphere MQ queue manager.

Metatopics

Metatopics are a special set of topics recognized by the WebSphere MQ Version 6 broker. See Metatopics.

Metatopics are *not provided* by WebSphere MQ Version 7. Instead you can inquire on the list of topic names, and on individual topics and subscriptions.

If you send a subscription to a metatopic, it is ignored.

Register Publisher and Deregister Publisher commands

The Register Publisher and Deregister Publisher commands do nothing in version 7, except return a successful response message to a request. Your publisher program is *not affected* by the change.

Routing exit

The WebSphere MQ Version 6 publish/subscribe has an exit for customizing and routing publications, which is described in Message broker exit. This exit is *no longer supported*.

Using the MQSD sublevel field, an intermediate subscriber can intercept publications to customize or block them, before they arrive at the ultimate subscribers, see “Intercepting publications” on page 81.

Streams

There are significant changes in how streams are implemented in WebSphere MQ version 7.

Although streams are not supported by the integrated publish/subscribe MQI interface, your version 6 queued publish/subscribe applications using streams interoperate *without change* with version 7 integrated publish/subscribe applications. This is because streams are mapped to topic space in version 7, see “Streams and topics” on page 115

Wildcards

The wildcard schemes used by version 6 and version 7 publish subscribe are different.

The earlier WebSphere MQ Version 6 publish/subscribe scheme, uses the characters described in Matching topic strings in WebSphere MQ Version 6.

An asterisk (*)
Match zero or more characters.

A question mark (?)
Match exactly one character.

The percent sign (%)
Escape "*", "?", or "%" in a topic string.

WebSphere MQ Version 7 and WebSphere Message Broker V6.1 use a different scheme to specify subscriptions. The version 7 topic string scheme is described in “Topics” on page 10.

A forward slash (/)
Topic level separator.

The hash sign (#)
Multilevel wildcard.

The plus sign (+)
Single-level wildcard.

There is no special escape character. When "#" or "+" are mixed with characters other than "/" they are treated as normal characters. Thus "/+/" and "/#/" are not valid topic strings and are single-level or multilevel wildcards respectively, whereas "/+ve/" matches only the subtopic "+ve".

To use the version 6 wildcard scheme when subscribing with a version 7 subscriber, set the MQSO_WILDCARD_CHAR option.

For a discussion of how WebSphere Message Broker handles WebSphere MQ Version 6 wildcards see the topic Wildcard characters.

Streams and topics

WebSphere MQ Version 6 publication streams are mapped to topics by WebSphere MQ Version 7. Default mapping is performed when a version 6 broker is migrated to version 7. You can add and tailor mappings for different configurations.

WebSphere MQ Version 6 publish/subscribe has the concept of a publication stream that does not exist in the WebSphere MQ Version 7 publish/subscribe model. In version 6, streams provide a way of separating the flow of information for different topics. A stream is implemented as a queue, defined at each broker that supports the stream. Each queue has the same name (the name of the stream).

The default stream `SYSTEM.BROKER.DEFAULT.STREAM` is set up automatically for all the brokers on a network, and no additional configuration is required to use the default stream. Think of the default stream as an unnamed default topic space. Topics published to the default stream are immediately available to all connected brokers, including version 7 brokers running with queued publish/subscribe enabled. Named streams are like separate, named, topic spaces. The named stream must be defined on each broker where it is used.

If you define a topic on WebSphere MQ Version 7, the topic is available to both version 6 and version 7 publishers and subscribers, with no special configuration. For example, suppose a topic is defined, with the topic string `Soccer/Results`. To receive soccer results, a version 6 application subscribes to `Soccer/Results` on `SYSTEM.BROKER.DEFAULT.STREAM`. A version 7 publisher publishes to `Soccer/Results` by opening the topic object and making MQPUT calls to it.

If the version 6 and 7 brokers are on different queue managers, then once the brokers are connected in the same broker hierarchy, no further configuration is required for the publications and subscriptions to flow between them.

The same interoperability works in reverse, too. So if the topic `Soccer/Results` is registered by a version 6 queued publish/subscribe application, then a version 7 application can subscribe to it using MQSUB.

Named streams

The version 6 solution designer might have decided to place all sports publications into a named stream called Sport. In version 6 a stream is often replicated automatically onto other brokers using the model queue, SYSTEM.BROKER.MODEL.STREAM. However, for the stream to be available to a version 7 broker running with queued publish/subscribe enabled, the stream needs to be added manually.

If you have just upgraded a queue manager from version 6 to version 7, running the command strmqbrk migrates version 6 publish/subscribe broker resources to version 7, and maps the streams that have been defined to topics. The stream Sport is mapped to the topic Sport.

Version 6 applications subscribing to Soccer/Results on stream Sport work without change. New version 7 applications subscribing to the topic Sport using MQSUB, and supplying the topic string Soccer/Results work as well too. When the topic Sport is created by strmqbrk, it is defined with the topic string Sport. A subscription to Soccer/Results is actually realized as a subscription to Sport/Soccer/Results, and so publications to the Sport stream are mapped to different place in topic space to publications to a different stream, such as Business.

There are scenarios for which the automatic migration performed by strmqbrk is not the answer, and you need to manually add streams on a version 7 queue manager. The task of adding a stream to a version 7 queue manager is described in the topic, Adding a stream. You might need to add streams manually for three reasons.

1. You continue to maintain publish/subscribe applications on version 6 queue managers, which interoperate with newly written version 7 publish/subscribe applications. You need to add new streams to version 7 to flow publications between the new streams and the topics on version 7.
2. You continue to develop your version 6 based publish/subscribe applications that are running on version 7 queue managers, rather than migrate the applications to the version 7 publish/subscribe MQI interface. You need to add new streams to the version 7 queue managers.
3. The default mapping of streams to topics leads to a "collision" in topic space, and publications on a stream have the same topic string as publications from elsewhere.

Mapping between streams and topics

A version 6 stream is mimicked in version 7 by creating a special queue for publications, giving it the same name as the version 6 stream[2]. The queue is identified to the publish/subscribe engine by adding it to the special namelist called SYSTEM.QPUBSUB.QUEUE.NAMELIST. You can add as many streams as you need, by adding additional special queues to the namelist. Finally you need to add topics, with the same names as the streams, and the same topic strings as the stream name, so you can publish and subscribe to the topics.

However, in exceptional circumstances, you can give the topics corresponding to the streams any topic strings you choose when you define the topics in version 7.

2. The special queue for publications is sometimes called the stream queue, because that is how it appears to version 6 publish/subscribe applications.

The purpose of the topic string is to give the topic a unique place in version 7 topic space. Usually the stream name serves that purpose perfectly. Sometimes, however, there may overlap between stream names and the version 7 topic space, in which case as long as you keep the topic strings unique, you can choose another string for the topic string.

The topic string defined in the topic definition is prefixed in the normal way to the topic string provided by publishers and subscribers using the MQOPEN or MQSUB MQI calls. The choice of prefix topic string has no effect on applications - which is why you can choose any topic string that keeps the publications unique in the topic space.

The remapping of different steams onto different topics relies on the prefixes used for the topic strings being unique, to separate one set of topics completely from another. You should define a universal topic naming convention that is rigidly adhered to for the mapping to work. In version 6, if topic strings collided you could use streams to separate the topic spaces. In version 7, you use the prefixing mechanism to remap a topic string to another place in topic space.

Note: When you delete a stream, delete all the subscriptions on the stream first. This is most important if any of the subscriptions originate from other brokers in the broker hierarchy.

Example

In Figure 48 on page 118, topic `'Sport'` has the topic string `'xyz'` resulting in publications originating from stream `'Sport'` being prefixed with the string `'xyz'` in the version 7 queue manager topic space. Publishing or subscribing in version 7 to the topic `'Sport'` prefixes `'xyz'` to the topic string. If the publication flows to a version 6 subscriber, the prefix `'xyz'` is removed from the publication and it is placed in the `'Sport'` stream. Conversely, when a publication flows from version 6 to version 6 from the `'Sport'` stream to the `'Sport'` topic, the prefix `'xyz'` is added to the topic string.

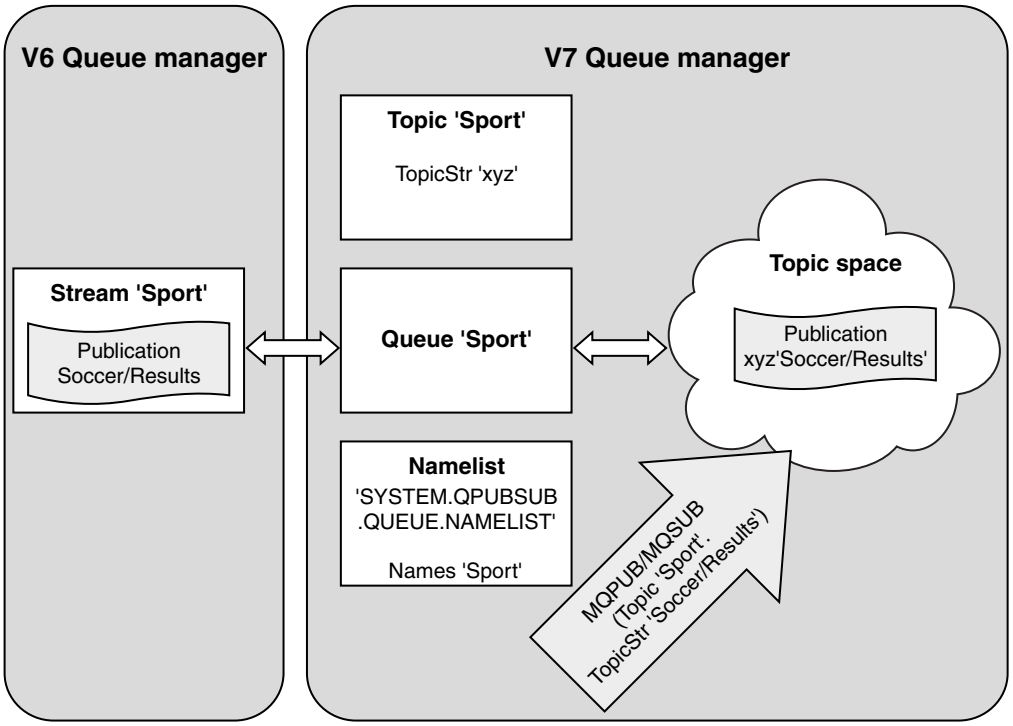


Figure 48. Version 6 streams coexisting with version 7 topics

Heterogeneous broker topologies

Version 6 WebSphere MQ publish/subscribe brokers are organized into *broker hierarchies*, WebSphere Message Brokers are organized into *collectives*, and in WebSphere MQ version 7 *clusters* or *broker hierarchies* are used to distribute publishers and subscribers over multiple queue managers.

Version 6 and version 7 WebSphere MQ publish/subscribe interoperate, using broker hierarchies. You cannot connect the WebSphere Message Broker V6.1 publish/subscribe broker to WebSphere MQ Version 7 using clusters or broker hierarchies.

You use a different procedure in WebSphere Version 7 to WebSphere MQ Version 6 to add a queue manager to a broker hierarchy.

Set the value of the queue manager attribute PARENT to the name (or alias) of the parent queue manager in the hierarchy. The procedure is described in the topic, "Publish/subscribe hierarchies" on page 40. To delete a parent: assign a different parent, or remove the value of the attribute altogether.

Note: If you are using named streams in your WebSphere MQ Version 6 broker hierarchy, you also need to add the streams manually to each version 7 queue manager added to the hierarchy.

Controlling queued publish/subscribe

Start, stop and display the status of queued publish/subscribe. Add and remove streams, and add and delete queue managers from a broker hierarchy.

New queue manager attributes for publish/subscribe

Five attributes, formerly held in the queue manager configuration file, qm.ini, are now replaced by attributes of the queue manager.

In WebSphere MQ Version 6.0, the attributes listed in the following table were held in the Brokers stanza of the qm.ini file (or the registry in Windows). In WebSphere MQ Version 7.0, they are replaced by the queue manager attributes listed, which can be set by the MQSC command ALTER QMGR or the PCF command Change Queue Manager.

Table 21.

Attribute in qm.ini	Queue manager attribute (PCF parameter name)	MQSC parameter name
MaxMsgRetryCount	PubSubMaxMsgRetryCount	PSRTYCNT
DiscardNonPersistentInputMsg	PubSubNPInputMsg	PSNPMSG
DLQNonPersistentResponse	PubSubNPResponse	PSNPRES
DiscardNonPersistentResponse	PubSubNPResponse	PSNPRES
SyncPointIfPersistent	PubSubsyncPoint	PSSYNCPT

Starting queued publish/subscribe

The task of starting the publish/subscribe broker in WebSphere MQ Version 7 has changed to enabling the (deprecated) queued publish/subscribe interface rather than running the strmqbrk command.

Before you begin

Read the description of PSMODE to understand the three modes of publish/subscribe:

- COMPAT
- DISABLED
- ENABLED

You must use strmqbrk to migrate Version 6 publish/subscribe broker state to version 7 if you are working with an upgraded queue manager.

About this task

Using the QMGR **PSMODE** attribute you can start either the queued publish/subscribe interface (also known as the broker), or the publish/subscribe engine (also known as Version 7 publish/subscribe) or both. To start queued publish/subscribe you need to set **PSMODE** to ENABLED. The default is ENABLED.

Queued publish/subscribe is deprecated.

Use WebSphere MQ Explorer or the runmqsc command to enable the queued publish/subscribe interface if the interface is not already enabled.

Example

```
ALTER QMGR PSMODE(ENABLED)
```

What to do next

WebSphere MQ now processes queued publish/subscribe commands and publish/subscribe MQI calls.

Stopping queued publish/subscribe

The task of stopping the publish/subscribe broker in WebSphere MQ Version 7 has changed to disabling the queued publish/subscribe interface rather than running the endmqbrk command.

Before you begin

Queued publish/subscribe is deprecated.

Read the description of PSMODE to understand the three modes of publish/subscribe:

- COMPAT
- DISABLED
- ENABLED

About this task

Using the QMGR **PSMODE** attribute you can stop either the queued publish/subscribe interface (also known as the broker), or the publish/subscribe engine (also known as Version 7 publish/subscribe) or both. To stop queued publish/subscribe you need to set **PSMODE** to COMPAT. To stop the publish/subscribe engine entirely, set **PSMODE** to DISABLED.

Use WebSphere MQ Explorer or the runmqsc command to disable the queued publish/subscribe interface.

Example

```
ALTER QMGR PSMODE(COMPAT)
```

What to do next

WebSphere MQ now processes only Version 7 MQI publish/subscribe calls.

Adding a stream

You can add streams manually to WebSphere MQ Version 7 queue managers to coexist with streams migrated from version 6 queue managers.

Before you begin

Familiarize yourself with the way publish/subscribe streams operate in WebSphere MQ version 7 onwards by reading the topic, “Streams and topics” on page 115.

About this task

Use PCF commands, runmqsc, or WebSphere MQ Explorer to do these steps.

1. Define a local queue with the same name as the version 6 stream.
2. Define a local topic with the same name as the version 6 stream.

3. Add the name of the queue to the namelist, SYSTEM.QPUBSUB.QUEUE.NAMELIST
4. Repeat these three steps for all queue managers at version 7 or above that are in the Publish/Subscribe hierarchy.

Adding 'Sport'

For example, if version 6 and version 7 queue managers are working in the same publish/subscribe hierarchy, and the version 6 queue managers share a stream called 'Sport', create a queue and a topic on version 7 queue managers called 'Sport', with a topic string 'Sport'.

A version 7 publish application, publishing to topic 'Sport', with topic string 'Soccer/Results', creates the resultant topic string 'Sport/Soccer/Results'. On version 7 queue managers , subscribers to topic 'Sport', with topic string 'Soccer/Results' receive the publication.

On version 6 queue managers, subscribers to stream 'Sport', with topic string 'Soccer/Results' receive the publication.

```
runmqsc QM1
5724-H72 (C) Copyright IBM Corp. 1994, 2008.  ALL RIGHTS RESERVED.
Starting MQSC for queue manager QM1.
define qlocal('Sport')
     1 : define qlocal('Sport')
AMQ8006: WebSphere MQ queue created.
define topic('Sport') topicstr('Sport')
     2 : define topic('Sport') topicstr('Sport')
AMQ8690: WebSphere MQ topic created.
alter namelist(SYSTEM.QPUBSUB.QUEUE.NAMELIST) NAMES('Sport', 'SYSTEM.BROKER.DEFAULT.STREAM', 'SYST
     3 : alter namelist(SYSTEM.QPUBSUB.QUEUE.NAMELIST) NAMES('Sport', 'SYSTEM.BROKER.DEFAULT.STREA
AMQ8551: WebSphere MQ namelist changed.
```

Note: You need to provide the existing names in a namelist as well as new names you are adding to the alter namelist command.

What to do next

Information about the stream is passed to other brokers in the hierarchy.

If a broker is version 6, administer it as a version 6 broker. That is, you have a choice of creating the stream queue manually, or letting the broker create the stream queue dynamically when it is needed. The queue is based on the model queue definition, SYSTEM.BROKER.MODEL.STREAM.

If a broker is version 7, you need to configure each version 7 queue manager in the hierarchy manually.

Deleting a stream

Before deleting a stream from a WebSphere MQ Version 7 queue manager you must ensure that there are no remaining subscriptions to the stream.

Before you begin

The use of queued publish/subscribe is deprecated in WebSphere MQ Version 7.0.

Before you delete a stream, quiesce all applications that use the stream.

You must be absolutely certain that there are no subscriptions left on the stream you are about to delete. If publications continue to flow to a deleted stream, it takes a lot of administrative effort to restore the system to a cleanly working state.

About this task

This task describes how to delete a stream from a WebSphere MQ Version 7.0, or later, queue manager. For instructions on deleting the stream from any version 6 queue managers it is connected to, see Deleting a stream.

1. Find all the connected brokers that host this stream.
2. Cancel all subscriptions to the stream on all the brokers.
3. Remove the queue (with the same name as the stream) from the namelist, `SYSTEM.QPUBSUB.QUEUE.NAMELIST`.
4. Delete or purge all the messages from the queue with the same name as the stream.
5. Delete the queue with the same name as the stream.
6. Delete the associated topic object.

What to do next

1. Repeat steps 3 to 5 on all the other connected version 7, or later, queue managers hosting the stream.
2. Remove the stream from all other connected version 6, or earlier, queue managers.

Connect a queue manager to a broker hierarchy

You can connect a local queue manager to a parent queue manager to modify a broker hierarchy.

Before you begin

1. You need to enable queued publish/subscribe mode. See "Starting queued publish/subscribe" on page 119.
2. The change is propagated to the parent queue manager using a WebSphere MQ connection. There are two ways to establish the connection.
   a. Connect the queue managers to a WebSphere MQ cluster.
   b. Establish a point-to-point channel connection using a transmission queue, or queue manager alias, with the same name as the parent queue manager.

      For example, suppose you are connecting to a queue manager called *PARENT*. Define a queue manager alias for *PARENT* that resolves to the transmission queue to parent. To place messages for *PARENT* on the transmission queue *PARENT.XMITQ*, use the following MQSC command to define the queue manager alias.

      ```
      DEFINE QREMOTE (PARENT) RNAME('') RQMNAME(PARENT) XMITQ(PARENT.XMITQ)
      ```

About this task

In WebSphere MQ Version 6.0, when the appropriate channels and queues are defined, brokers connect to one another as defined by parameters provided on the strmqbrk command.

The strmqbrk command works differently in WebSphere MQ Version 7.0 and you can no longer use it to connect children to parents. Instead you use the `ALTER QMGR PARENT (PARENT)` runmqsc command.

In WebSphere MQ Version 7, distributed publish/subscribe is typically implemented by using queue manager clusters and clustered topic definitions. For interoperability with WebSphere MQ Version 6 and WebSphere Message Broker V6.1 and WebSphere Event Broker V6.1 and earlier, you can also connect version 7 queue managers to a broker hierarchy as long as queued publish/subscribe mode is enabled.

ALTER QMGR PARENT(*PARENT*)

Example

The first example shows how to attach QM2 as a child of QM1, and then querying QM2 for its connection.

```
C:>runmqsc QM2
5724-H72 (C) Copyright IBM Corp. 1994, 2008.  ALL RIGHTS RESERVED.
Starting MQSC for queue manager QM2
alter qmgr parent(QM1)
     1 : alter qmgr parent(QM1)
AMQ8005: WebSphere MQ queue manager changed.
display pubsub type(All)
    14 : display pubsub type(All)
AMQ8723: Display pub/sub status details.
   QMNAME(QM2)                             TYPE(LOCAL)
AMQ8723: Display pub/sub status details.
   QMNAME(QM1)                             TYPE(PARENT)
```

The next example shows the result of querying QM1 for its connections

```
C:\Documents and Settings\Admin>runmqsc QM1
5724-H72 (C) Copyright IBM Corp. 1994, 2008.  ALL RIGHTS RESERVED.
Starting MQSC for queue manager QM1.
display pubsub type(all)
     1 : display pubsub type(all)
AMQ8723: Display pub/sub status details.
   QMNAME(QM1)                             TYPE(LOCAL)
AMQ8723: Display pub/sub status details.
   QMNAME(QM2)                             TYPE(CHILD)
```

What to do next

You can define topics on one broker or queue manager that are available to publishers and subscribers on the connected queue managers.

Disconnect a queue manager from a broker hierarchy

Disconnect a child queue manager from a parent queue manager in a broker hierarchy.

About this task

In WebSphere MQ Version 6.0, queue managers were disconnected from one another using the dltmqbrk command, and required that all child queue managers were disconnected first. In WebSphere MQ Version 7, the dltmqbrk command is used to discard WebSphere MQ Version 6 broker resources after migration to version 7 using the strmqbrk command.

You disconnect a version 7 queue manager from a broker hierarchy using the ALTER QMGR command. Unlike version 6, you can disconnect version 7 queue managers in any order and at any time.

The corresponding request to update the parent is sent when the connection between the queue managers is running.

```
ALTER QMGR PARENT(' ')
```

Example

```
C:\Documents and Settings\Admin>runmqsc QM2
5724-H72 (C) Copyright IBM Corp. 1994, 2008.  ALL RIGHTS RESERVED.
Starting MQSC for queue manager QM2.
    1 : alter qmgr parent('')
AMQ8005: WebSphere MQ queue manager changed.
    2 : display pubsub type(child)
AMQ8147: WebSphere MQ object  not found.
display pubsub type(parent)
    3 : display pubsub type(parent)
AMQ8147: WebSphere MQ object  not found.
```

What to do next

You can delete any streams, queues and manually defined channels that are no longer needed.

Migration to publish/subscribe on WebSphere MQ V7.0

Publish/subscribe function in WebSphere MQ Version 7.0 is performed by the queue manager, rather than by a separate publish/subscribe broker. When you migrate your systems to WebSphere MQ Version 7.0, publish/subscribe function is not automatically migrated. You must upgrade publish/subscribe information to WebSphere MQ Version 7.0 separately.

In WebSphere MQ V6, applications perform publish and subscribe operations by placing special request messages on certain queues. The WebSphere MQ V6 Publish/Subscribe Broker then reads and acts on these messages (for example by publishing messages to subscribing applications). State information such as who is subscribing to which publications is owned and maintained by the publish/subscribe broker. This broker is started and stopped independently from the queue manager.

In WebSphere MQ V7, newly written publish/subscribe applications do not communicate with the broker in order to publish or subscribe; they use the new API directly. The verb MQPUT is used to publish messages to a topic and MQSUB is used to subscribe. The queue manager itself performs the publish/subscribe function, so no separate publish/subscribe broker is required.

When you upgrade a queue manager from WebSphere MQ V6 to WebSphere MQ V7, the publish/subscribe broker is not upgraded. State information must be migrated from the WebSphere MQ publish/subscribe broker into the queue manager. Data that is migrated includes subscriptions, retained publications, hierarchy relations, and authorities. You migrate a queue manager by using the strmqbrk command, which previously started the publish/subscribe broker.

WebSphere MQ V6 publish/subscribe brokers could be connected into hierarchies so that publications and subscriptions could flow between them. After migrating (using strmqbrk) these hierarchies continue to function in WebSphere MQ V7. WebSphere MQ V7 also contains a new method of allowing publications and subscriptions flow between queue managers; publish/subscribe clusters. An advantage of using publish/subscribe clusters mean that no queue manager

provides a single point of failure to the flow of publications or subscriptions. To migrate to a publish/subscribe cluster, first migrate to a WebSphere MQ V7 hierarchy using strmqbrk and then convert it to a cluster by creating cluster topics and altering parent/child relations.

strmqbrk (Migrate WebSphere MQ Version 6.0 broker to Version 7.0)

Migrate the persistent state of a Websphere MQ publish/subscribe broker.

Purpose

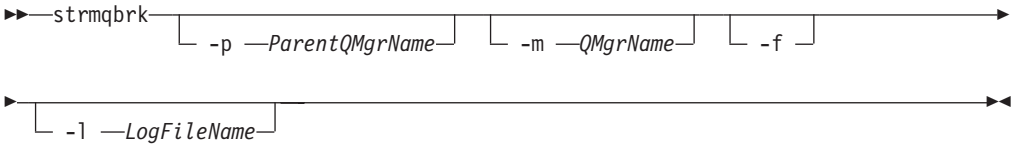
Use the **strmqbrk** command to migrate WebSphere MQ Version 6.0 publish/subscribe broker's state to WebSphere MQ Version 7.0 publish/subscribe. If the queue manager has already been migrated, no action is taken.

In WebSphere MQ Version 6.0, **strmqbrk** started a broker. The WebSphere MQ Version 7.0 publish/subscribe engine cannot be started in this manner. To enable publish/subscribe for a queue manager, use the ALTER QMGR command; for details, see ALTER QMGRin the *WebSphere MQ Script (MQSC) Command Reference*.

You can also use the runmqbrk command. This has the same parameters as strmqbrk and exactly the same effect.

Syntax

AIX, HP-UX, Linux, Solaris, and Windows



Optional parameters

AIX, HP-UX, Linux, Solaris, and Windows

-p *ParentQMgrName*

Note: This option is deprecated.
If you specify the current parent queue manager, a warning message is issued and migration continues. If you specify a different queue manger, a warning is issued and migration is not performed.

-m *QMgrName*
The name of the queue manager to be migrated. If you do not specify this parameter, the command is routed to the default queue manager.

-f Force migration. This option specifies that objects created during the migration replace existing objects with the same name. If this option is not specified, if migration would create a duplicate object, a warning is issued, the object is not created, and migration continues.

-l *LogFileName*
Log migration activity to the file specified in *LogFileName*.

Syntax

i5/OS

Optional parameters

AIX, HP-UX, Linux, Solaris, and Windows

-PARENTMQM*(ParentQMgrName)*

Note: This option is deprecated.
If you specify the current parent queue manager, a warning message is issued and migration continues. If you specify a different queue manger, a warning is issued and migration is not performed.

-MQMNAME *QMgrName*
The name of the queue manager to be migrated. If you do not specify this parameter, the command is routed to the default queue manager.

Application migration

The WebSphere MQ Version 6.0 publish/subscribe command message interface is being deprecated. If you have applications that use this interface directly, you should migrate those applications to use the new Version 7.0 publish/subscribe functions.

The following sections explain how to replace existing command messages.

Identity

In WebSphere MQ Version 6 there were two ways of identifying a subscriber. These were referred to as the traditional identity and the subscription name.

The traditional identity was also used to identify a publisher. The traditional identity was a combination of queue name, queue manager name, and optional correlation identifier.

A publisher no longer has an explicit publisher identity, but can be identified in the same way as any only WebSphere MQ application, by means of its connection to the queue manager. Since there is no explicit registration of a publisher, or his identity over and above what can be obtained by displaying the connections to the queue manager, there is no longer a need for the anonymous option on Register Publisher. Your application must now use the SubName field in the MQSD to identify a subscriber.

The correlation identifier also had a secondary use which was to allow subscribers to MQGET by CorrelId to only get publications for a particular subscription, if there were multiple subscriptions all using the same queue. This is provided by using the SubCorrelId field returned in the MQSD at MQSUB time.

Stream Name

MQPS_STREAM_NAME is deprecated since stream names are part of the full topic name. Stream names can be mapped to administrative topic objects, and then the topic name used along with the stream name can be mapped to a topic string to be concatenated with the topic string from the topic object. For example, if the application was previously using a stream queue name of SYSTEM.BROKER.RESULTS.STREAM and a topic of `Sport/Soccer/State/LatestScore/*`, then a topic object can be created whose name is SYSTEM.BROKER.RESULTS.STREAM which is defined to have a TOPICSTR of / and the new application will provide a two part topic name in the MQOD or MQSD using an ObjectName of SYSTEM.BROKER.RESULTS.STREAM and an ObjectString of `Sport/Soccer/State/LatestScore/*`.

If an administrative topic object that does not exist is used in place of a stream name, the error (effectively mapping to MQRCCF_STREAM_ERROR) which is given is MQRC_UNKNOWN_OBJECT_NAME. supported.

Application migration details

When migrating to use the MQ API to do publish/subscribe, the code within any one application program must be consistent.

The application program must not contain a mixture of these deprecated APIs and the new MQ API options. An entire application suite, such as the combination of a subscribing application program and a publishing application program, does not all need to be migrated at the same time. Interaction between a publishing application program using the deprecated APIs and a subscribing application using the new MQ API is supported.

Delete Publication - Version 7 replacement

The Delete Publication command message contains a number of parameters. This should be replaced by using the PCF ClearTopic command. This section details the equivalent options or fields in the PCF command message to show how an application would migrate from using the Delete Publication command message to using the PCF ClearTopic command message.

Required parameters

MQPS_COMMAND with value MQPS_DELETE_PUBLICATION is implied when you use the ClearTopic command.

MQPS_TOPIC is provided in a field in the ClearTopic command message. If your application provided more than one MQPS_TOPIC in a single Delete Publication command message, it must now issue a separate ClearTopic call for each separate topic string.

Optional parameters

MQPS_DELETE_OPTIONS is replaced with an attribute of the ClearTopic command message.

For MQPS_STREAM_NAME see *WebSphere MQ Publish/Subscribe User's Guide*.

Error codes

If your application checked for any of the following error codes, the equivalent MQRC error codes are shown in the following table:

Reason codes in NameValueString of the broker response message.	MQRC equivalent
MQRCCF_STREAM_ERROR	MQRC_UNKNOWN_OBJECT_NAME
MQRCCF_TOPIC_ERROR	MQRC_OBJECT_STRING_ERROR
MQRCCF_INCORRECT_STREAM	See Note 1

Notes:

1. No equivalent since there is no need to provide the stream name twice, once in the command and once by putting it to the stream queue, so you cannot have a mismatch.

Deregister publisher - Version 7 replacement

The Deregister Publisher command message contains a number of parameters. You should replace it with the MQCLOSE verb. This section details the equivalent options or fields in the MQ API to show how to migrate an application from the Deregister Publisher command message to MQCLOSE.

A difference in behaviour will be seen because a Register Publisher command could leave an application registered even when it was not connected, whereas the equivalent MQOPEN will only show a publisher's intent when the application is connected and keeps the handle from MQOPEN available. Even without issuing MQCLOSE, an application will be deregistered when the queue manager detects that the application's connection is lost.

Required parameters

MQPS_COMMAND with value MQPS_REGISTER_PUBLISHER is implied when closing a handle to a topic previously opened using MQOPEN with the MQOO_OUTPUT option.

Optional parameters

If your application provided a queue and queue manager name (either by using MQPS_Q_MGR_NAME and MQPS_Q_NAME in the command message, or from the ReplyToQ and ReplyToQMgr fields in MQMD of the command message) these attributes are now implied by the provision of the handle obtained when opening the topic.

MQPS_REGISTRATION_OPTIONS is replaced with options on the MQCLOSE call. See MQCLOSE for more details. Note that there are two ways you could have specified each of these options in your application, a string constant, MQPS_* or an integer constant, MQREGO_*. Both are replaced by the use of a single numeric constant.

String constant	Integer constant	MQCLOSE Options field constant
MQPS_CORREL_ID_AS_IDENTITY	MQREGO_CORREL_ID_AS_IDENTITY	See Version 6 (queued) publish/subscribe
MQPS_DEREGISTER_ALL	MQREGO_DEREGISTER_ALL	See Note 1

String constant	Integer constant	MQCLOSE Options field constant
Notes: 1. Since only one topic can be opened by the MQOPEN call, closing the handle closes that one topic. There is no need for an equivalent option. If many topics are opened, simply issuing MQDISC will close them all, saving the need to MQCLOSE each handle.		

For MQPS_STREAM_NAME see Version 6 (queued) publish/subscribe, although in this case, the stream name is implied by the provision of the handle obtained when opening the topic. MQPS_TOPIC is implied by the provision of the handle obtained when opening the topic.

Error codes

If your application checked for any of the following error codes, the equivalent MQRC error codes are shown in the following table:

Reason codes in NameValueString of the broker response message.	MQRC equivalent
MQRCCF_STREAM_ERROR	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_TOPIC_ERROR	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_NOT_REGISTERED	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_Q_MGR_NAME_ERROR	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_Q_NAME_ERROR	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_DUPLICATE_IDENTITY	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_UNKNOWN_STREAM	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_REG_OPTIONS_ERROR	MQRC_OPTIONS_ERROR
Notes: 1. This error code implies the same type of problem, but since all of these fields are now implied by the provision of the handle obtained when opening the topic, this is the only equivalent error.	

Deregister subscriber - Version 7 replacement

The Deregister Subscriber command message contains a number of parameters. This should be replaced by using the MQCLOSE verb. This section details the equivalent options or fields in the MQ API to show how an application would migrate from using the Deregister Subscriber command message to using MQCLOSE. If the Deregister Subscriber command message was used in a different program from that of the Register Subscriber command message, the application must now first use the MQSUB call with the MQSO_RESUME option to get a handle to the subscription, in order to deregister it.

Required parameters

MQPS_COMMAND with value MQPS_DEREGISTER_SUBSCRIBER is replaced by the use of the MQCLOSE verb with the option MQCO_REMOVE_SUB.

Optional parameters

If your application provided a queue and queue manager name (either by using MQPS_Q_MGR_NAME and MQPS_Q_NAME in the command message, or from the ReplyToQ and ReplyToQMgr fields in MQMD of the command message) these attributes are now implied by the provision of the handle obtained when subscribing to the topic.

MQPS_REGISTRATION_OPTIONS is replaced with Options on the MQCLOSE call. See MQCLOSE for more details. Note that there are two ways you could have specified each of these options in your application, a string constant, MQPS_* or an integer constant, MQREGO_*. Both are replaced by the use of a single numeric constant.

String constant	Integer constant	MQCLOSE Options field constant
MQPS_CORREL_ID_AS_IDENTITY	MQREGO_CORREL_ID_AS_IDENTITY	See Note 1
MQPS_DEREGISTER_ALL	MQREGO_DEREGISTER_ALL	See Note 2
MQPS_FULL_RESPONSE	MQREGO_FULL_RESPONSE	See Note 3
MQPS_LEAVE_ONLY	MQREGO_LEAVE_ONLY	See Note 4
MQPS_VARIABLE_USER_ID	MQREGO_VARIABLE_USER_ID	See Note 1

Notes:

1. This option is implied by the provision of the handle obtained when subscribing to the topic.
2. Since only one topic (separate topic string that is – of course wildcards can still be used within one topic string) can be subscribed to by the MQSUB call, closing the handle closes that one topic. There is no need for an equivalent option. If many topics are opened, simply issuing MQDISC will close them all, saving the need to MQCLOSE each handle.
3. Use of this option is implied in the use of the MQSUB verb. The fields returned in the response message are now populated in the MQSD structure. See MQSUB for more details. Because an MQSUB call must be made in order to obtain the handle to pass to the MQCLOSE call, this option is deprecated.
4. Use of these options are deprecated and moved to spiCONN for the one environment where they are needed.

For MQPS_STREAM_NAME see Version 6 (queued) publish/subscribe, although in this case, the stream name is implied by the provision of the handle obtained when subscribing to the topic. MQPS_SUBSCRIPTION_IDENTITY is replaced by a field in spiCONN for the one environment where it is needed. MQPS_SUBSCRIPTION_NAME is replaced by the field in the MQSD called SubName and is therefore implied by the provision of the handle obtained when subscribing to the topic. MQPS_TOPIC is provided in a field in the MQSD called ObjectString, and is therefore implied by the provision of the handle obtained when subscribing to the topic.

See MQSD for more details

Error codes

If your application checked for any of the following error codes, the equivalent MQRC error codes are shown in the following table:

Reason codes in NameValueString of the broker response message.	MQRC equivalent
MQRCCF_STREAM_ERROR	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_TOPIC_ERROR	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_NOT_REGISTERED	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_Q_MGR_NAME_ERROR	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_Q_NAME_ERROR	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_DUPLICATE_IDENTITY	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_UNKNOWN_STREAM	MQRC_HOBJ_ERROR (See note 1)
MQRCCF_REG_OPTIONS_ERROR	MQRC_OPTIONS_ERROR

Notes:

1. This error code implies the same type of problem, but because all of these fields are now implied by the provision of the handle obtained when opening the topic, this is the only equivalent error.

Publish - Version 7 replacement

The Publish command message contains a number of parameters. This should be replaced by using the MQPUT/MQPUT1 verbs. This section details the equivalent options or fields in the MQ API to show how an application would migrate from using the Publish command message to using MQPUT/MQPUT1.

Required parameters

MQPS_COMMAND with value MQPS_PUBLISH is implied when putting a message to an object handle opening a topic for MQOO_OUTPUT. If your application did not use Register Publisher, see the details in Publish for remaining unregistered.

MQPS_TOPIC is provided in a field in the MQOD called ObjectString. See MQOD - Object Descriptor for more details. If your application provided more than one MQPS_TOPIC in a single Register Publisher command message, it must now issue a separate MQOPEN call for each separate topic string.

Optional parameters

MQPS_INTEGER_DATA can be replaced with a message property.

MQPS_PUBLICATION_OPTIONS is replaced with the Options field in the MQPMO structure. See MQPMO for more details. Note that there are two ways you could have specified each of these options in your application, a string constant, MQPS_* or an integer constant, MQREGO_*. Both are replaced by the use of a single numeric constant.

String constant	Integer constant	MQCLOSE Options field constant
MQPS_CORREL_ID_AS_ IDENTITY	MQREGO_CORREL_ID_ AS_IDENTITY	See Version 6 (queued) publish/subscribe for more information
MQPS_IS_RETAINED_ PUBLICATION	MQREGO_IS_RETAINED_ PUBLICATION	See Note 1

String constant	Integer constant	MQCLOSE Options field constant
MQPS_NO_REGISTRATION	MQREGO_NO_ REGISTRATION	See Note 2
MQPS_OTHER_ SUBSCRIBERS_ONLY	MQREGO_OTHER_ SUBSCRIBERS_ONLY	See Note 3
MQPS_RETAIN_ PUBLICATION	MQREGO_RETAIN_ PUBLICATION	MQPMO_RETAIN

Notes:

1. A message property will contain this information
2. This option is deprecated because publishers are no longer registered
3. This option is deprecated. If an application does not want to receive its own publications it should subscribe using the option MQSO_NO_LOCAL on the MQSUB call.

MQPS_Q_MGR_NAME is replaced by the ReplyToQMgr in the MQMD of the publication. If the publisher specifies MQPMO_NO_DIRECT_REQUEST the ReplyToQMgr will not contain the publishers queue manager name, otherwise it will.

MQPS_Q_NAME is replaced by the ReplyToQ in the MQMD of the publication. If the publisher does not set this, it is not available.

MQPS_REGISTRATION_OPTIONS is replaced with Options in the MQPMO. See MQPMO for more details. These are exactly the same as those in the section on Register Publisher below.

MQPS_SEQUENCE_NUMBER is replaced with a message property.

For MQPS_STREAM_NAME see Version 6 (queued) publish/subscribe. MQPS_STRING_DATA is replaced with a message property.

Register publisher - Version 7 replacement

The Register Publisher command message contains a number of parameters. This should be replaced by using the MQOPEN verb. This section details the equivalent options or fields in the MQ API to show how an application would migrate from using the Register Publisher command message to using MQOPEN. A difference in behaviour will be seen because a Register Publisher command could leave an application registered even when it was not connected, whereas MQOPEN will only show a publishers intent when the application is connected and keeps the handle from MQOPEN available.

Required parameters

MQPS_COMMAND with value MQPS_REGISTER_PUBLISHER is implied when opening a topic for MQOO_OUTPUT. If your application did not use Register Publisher, see the details in Publish for remaining unregistered.

MQPS_TOPIC is provided in a field in the MQOD called ObjectString. See MQOD - Object Descriptor for more details. If your application provided more than one MQPS_TOPIC in a single Register Publisher command message, it must now issue a separate MQOPEN call for each separate topic string.

Optional parameters

If your application provided a queue and queue manager name (either by using MQPS_Q_MGR_NAME and MQPS_Q_NAME in the command message, or from the ReplyToQ and ReplyToQMgr fields in MQMD of the command message) in order for subscribing applications to be able to directly contact the publisher, then your application must now provide these details on each published message.

MQPS_REGISTRATION_OPTIONS is replaced with Options in the MQPMO. See MQPMO for more details. Note that there are two ways you could have specified each of these options in your application, a string constant, MQPS_* or an integer constant, MQREGO_*. Both are replaced by the use of a single numeric constant.

String constant	Integer constant	MQCLOSE Options field constant
MQPS_ANONYMOUS	MQREGO_ANONYMOUS	See Version 6 (queued) publish/subscribe
MQPS_CORREL_ID_ AS_IDENTITY	MQREGO_CORREL_ID_ AS_IDENTITY	See Version 6 (queued) publish/subscribe
MQPS_DIRECT_REQUEST	MQREGO_DIRECT_ REQUEST	See Note 1
MQPS_LOCAL	MQREGO_LOCAL	MQPMO_SCOPE_QMGR

Notes:

1. Use of this option is implied if the ReplyToQ and ReplyToQMgr fields are provided in the MQMD of the message put. If these fields are not provided, the queue manager will still fill in the ReplyToQMgr as the queue manager local to the publisher. To remain completely anonymous and not even provide this information to subscribers, your application should use the MQPMO_NO_DIRECT_REQUEST option.

For MQPS_STREAM_NAME see Version 6 (queued) publish/subscribe.

Register subscriber - Version 7 replacement

The Register Subscriber command message contains a number of parameters. This should be replaced by using the MQSUB verb. This section details the equivalent options or fields in the MQ API to show how an application would migrate from using the Register Subscriber command message to using MQSUB.

Required parameters

MQPS_COMMAND with value MQPS_REGISTER_SUBSCRIBER is replaced by the use of the MQSUB verb. If your application did not use Register Subscriber then the use of the MQSUB verb is not required for equivalent behaviour.

MQPS_TOPIC is provided in a field in the MQSD called ObjectString. See MQSD for more details. If your application provided more than one MQPS_TOPIC in a single Register Subscriber command message, it must now issue a separate MQSUB call for each separate topic string.

Optional parameters

If your application provided a non-local queue name and/or a queue manager name other than the one connected to (either by using MQPS_Q_MGR_NAME and MQPS_Q_NAME in the command message, or from the ReplyToQ and

ReplyToQMgr fields in MQMD of the command message) then your application must now provide an object handle, which has been returned by a MQOPEN call for that queue, in the Hobj parameter of the MQSUB verb.

If your application provided the name of a queue local to the queue manager it connected to, it now has the option to request that the queue manager manage where the publications are sent. This can be done by using the MQSO_MANAGED option in the field in the MQSD called Options.

MQPS_REGISTRATION_OPTIONS is replaced with a field in the MQSD called Options. See MQSD for more details. Note that there are two ways you could have specified each of these options in your application, a string constant, MQPS_* or an integer constant, MQREGO_*. Both are replaced by the use of a single numeric constant.

String constant	Integer constant	MQCLOSE Options field constant
MQPS_ADD_NAME	MQREGO_ADD_NAME	See Note 1
MQPS_ANONYMOUS	MQREGO_ANONYMOUS	See Identity
MQPS_CORREL_ID_AS_IDENTITY	MQREGO_CORREL_ID_AS_IDENTITY	See Identity (also see Note 7)
MQPS_DUPLICATES_OK	MQREGO_DUPLICATES_OK	See Note 2
MQPS_FULL_RESPONSE	MQREGO_FULL_RESPONSE	See Note 3
MQPS_INCLUDE_STREAM_NAME	MQREGO_INCLUDE_STREAM_NAME	See Note 4
MQPS_INFORM_IF_RETAINED	MQREGO_INFORM_IF_RETAINED	See Note 5
MQPS_JOIN_EXCLUSIVE	MQREGO_JOIN_EXCLUSIVE	See Note 6
MQPS_JOIN_SHARED	MQREGO_JOIN_SHARED	See Note 6
MQPS_LOCAL	MQREGO_LOCAL	MQSO_SCOPE_QMGR
MQPS_LOCKED	MQREGO_LOCKED	See Note 6
MQPS_NEW_PUBLICATIONS_ONLY	MQREGO_NEW_PUBLICATIONS_ONLY	MQSO_NEW_PUBLICATIONS_ONLY
MQPS_NO_ALTERATION	MQREGO_NO_ALTERATION	MQSO_RESUME
MQPS_NON_PERSISTENT	MQREGO_NON_PERSISTENT	MQSO_NON_PERSISTENT
MQPS_PERSISTENT	MQREGO_PERSISTENT	MQSO_PERSISTENT
MQPS_PERSISTENT_AS_PUBLISH	MQREGO_PERSISTENT_AS_PUBLISH	MQSO_PERSISTENT_AS_PUBLISH
MQPS_PERSISTENT_AS_Q	MQREFO_PERSISTENT_AS_Q	MQSO_PERSISTENT_AS_QUEUE_DEF
MQPS_PUBLISH_ON_REQUEST_ONLY	MQREGO_PUBLISH_ON_REQUEST_ONLY	MQSO_PUBLICATIONS_ON_REQUEST
MQPS_VARIABLE_USER_ID	MQREGO_VARIABLE_USER_ID	MQSO_ANY_USERID, (also see Note 7)

String constant	Integer constant	MQCLOSE Options field constant
Notes: 1. Use of this option is deprecated since the only identity of a subscription is the SubName. See Deprecation. 2. Use of this option is deprecated since the queued interface has been removed. 3. Use of this option is implied in the use of the MQSUB verb. The fields returned in the response message are now populated in the MQSD structure. See MQSD for more details. 4. Use of this option is deprecated since stream names are part of the full topic name. 5. Use of this option is deprecated since the information about whether a publication is a retained publication or not is a message property that is always present. 6. Use of these options are deprecated and moved to spiCONN for the one environment where they are needed. 7. A tick in this column indicates this option is also relevant for Request Update.		

For MQPS_STREAM_NAME see Version 6 (queued) publish/subscribe, although in this case, the stream name is implied by the provision of the handle obtained when subscribing to the topic.

MQPS_SUBSCRIPTION_IDENTITY is replaced by a field in spiCONN for the one environment where it is needed.

MQPS_SUBSCRIPTION_NAME is replaced by the field in the MQSD called SubName. See MQSD for more details.

MQPS_SUBSCRIPTION_USER_DATA is replaced by the field in the MQSD called SubUserData. See MQSD for more details.

Error codes

If your application checked for any of the following error codes, the equivalent MQRC error codes are shown in the following table:

Reason codes in NameValueString of the broker response message.	MQRC equivalent
MQRCCF_STREAM_ERROR	
MQRCCF_TOPIC_ERROR	
MQRCCF_Q_MGR_NAME_ERROR	
MQRCCF_Q_NAME_ERROR	
MQRCCF_DUPLICATE_IDENTITY	MQRC_IDENTITY_MISMATCH
MQRCCF_CORREL_ID_ERROR	
MQRCCF_NOT_AUTHORIZED	
MQRCCF_UNKNOWN_STREAM	
MQRCCF_REG_OPTIONS_ERROR	
MQRCCF_DUPLICATE_SUBSCRIPTION	MQRC_SUB_ALREADY_EXISTS
MQRCCF_SUB_NAME_ERROR	
MQRCCF_SUB_IDENTITY_ERROR	See note 1
MQRCCF_SUBSCRIPTION_IN_USE	MQRC_SUBSCRIPTION_IN_USE

Reason codes in NameValueString of the broker response message.	MQRC equivalent
MQRCCF_SUBSCRIPTION_LOCKED	See note 1
MQRCCF_ALREADY_JOINED	See note 1

Notes:

1. No equivalent since the use of SubIdentity is deprecated since the only identity of a subscription is the SubName. See Deprecation.

Request Update - Version 7 replacement

The Request Update command message contains a number of parameters. This should be replaced by using the MQSUBRQ verb. This section details the equivalent options or fields in the MQ API to show how an application would migrate from using the Request Update command message to using MQSUBRQ.

Required parameters

MQPS_COMMAND with value MQPS_REQUEST_UPDATE is replaced by the use of the MQSUBRQ verb.

MQPS_TOPIC is implied by the use of the Hsub handle returned from the MQSUB call which is used as a parameter on the MQSUBRQ call.

Optional parameters

QMgrName, QName and StreamName are used in exactly the same way in Request Update command messages as they are in Register Subscriber command messages.

See “Register subscriber - Version 7 replacement” on page 133 for details of how to migrate the use of these fields.

See “Register subscriber - Version 7 replacement” on page 133 for details of how to migrate your application’s use of MQPS_REGISTRATION_OPTIONS in this command message.

For MQPS_STREAM_NAME see Version 6 (queued) publish/subscribe.

MQPS_SUBSCRIPTION_NAME is implied by the use of the Hsub handle returned from the MQSUB call which is used as a parameter on the MQSUBRQ call.

New queue manager attributes for publish/subscribe

Five attributes, formerly held in the queue manager configuration file, qm.ini, are now replaced by attributes of the queue manager.

In WebSphere MQ Version 6.0, the attributes listed in the following table were held in the Brokers stanza of the qm.ini file (or the registry in Windows). In WebSphere MQ Version 7.0, they are replaced by the queue manager attributes listed, which can be set by the MQSC command ALTER QMGR or the PCF command Change Queue Manager.

Table 22.

Attribute in qm.ini	Queue manager attribute (PCF parameter name)	MQSC parameter name
MaxMsgRetryCount	PubSubMaxMsgRetryCount	PSRTYCNT
DiscardNonPersistentInputMsg	PubSubNPInputMsg	PSNPMSG
DLQNonPersistentResponse	PubSubNPResponse	PSNPRES
DiscardNonPersistentResponse	PubSubNPResponse	PSNPRES
SyncPointIfPersistent	PubSubsyncPoint	PSSYNCPT

WebSphere MQ publish/subscribe topology migration

This section contains topics that describe various scenarios for migration to WebSphere MQ Version 7.0 publish/subscribe.

Migrating a WebSphere MQ Version 6.0 publish/subscribe hierarchy to a Version 7.0 publish/subscribe cluster - all queue managers simultaneously

How to migrate an entire existing Websphere MQ Version 6.0 hierarchy, where the parent and child queue managers are on separate computers, to a Websphere Version 7.0 publish/subscribe cluster, migrating all queue managers at the same time.

Before you begin

To migrate the hierarchy, perform the following steps:

About this task

1. Install WebSphere MQ Version 7.0 on all of the computers that contain queue managers in the hierarchy, to upgrade all queue managers in the hierarchy to WebSphere MQ Version 7.0.
2. Use the **strmqbrk** control command on each queue manager to migrate all publish/subscribe configuration data into WebSphere MQ Version 7.0.
3. Create a new cluster or nominate an existing cluster, which need not be an existing publish/subscribe cluster. You can do this using WebSphere MQ Script commands (MQSC), or any other type of administration command or utility that is available on your platform, such as the WebSphere MQ Explorer. These methods are described in *WebSphere MQ Queue Manager Clusters*.
4. Ensure that each queue manager is in the cluster by using the MQSC command DISPLAY CLUSQMGR(*), described in *WebSphere MQ Script (MQSC) Command Reference*. If a queue manager that should be in the cluster is not, then add it. For more information, refer to *WebSphere MQ Queue Manager Clusters*
5. To remove the hierarchical relationship on each child queue manager within the hierarchy, execute the following MQSC command: `ALTER QMGR PARENT(' ')`
6. Before proceeding to the next step, to confirm that all the hierarchical relationships have been cancelled, use the MQSC command `DISPLAY PUBSUB TYPE(ALL)` on each queue manager.
7. On one of the queue managers within the cluster, define one cluster topic by executing the following MQSC command: `ALTER TOPIC(<topic name>) PUBSCOPE(ALL) SUBSCOPE(ALL) CLUSTER(<cluster>)` Use a high-level topic, but not the root. For information about cluster topic naming, see Cluster topics.

Example

What to do next

Alternative procedure for i5/OS:

The following steps show an alternative procedure for WebSphere MQ for i5/OS, using CL commands and panels in place of MQSC commands.

1. Install WebSphere MQ Version 7.0 on all of the computers that contain queue managers in the hierarchy, to upgrade all queue managers in the hierarchy to WebSphere MQ Version 7.0.
2. Use the **strmqbrk** command on each queue manager to migrate all publish/subscribe configuration data into WebSphere MQ Version 7.0.
3. Create a new cluster or nominate an existing cluster, which need not be an existing publish/subscribe cluster. You can do this using WebSphere MQ Script commands (MQSC), or any other type of administration command or utility that is available on your platform, such as the WebSphere MQ Explorer. These methods are described in *WebSphere MQ Queue Manager Clusters*
4. Ensure that each queue manager is in the cluster. If a queue manager that should be in the cluster is not, then add it.
5. Execute `WRKMQMPS PUBSUBNAME(<parent_queue_manager>)` to display the hierarchy.
6. On each child queue manager within the hierarchy, use **option 4=Remove** to detach from the parent, followed by **option 34=Work with Pub/Sub** to move down the sub-hierarchy. Repeat options 4 and 34 until no child queue managers are displayed.
7. Repeat step 6 for each child queue manager belonging to PUBSUBNAME(<parent_queue_manager>) until no child queue managers are displayed.
8. On one of the queue managers within the cluster, define at least one cluster topic by executing the following command: `CHGMQMTOP TOPNAME(<topic name>) PUBSCOPE(*ALL) SUBSCOPE(*ALL) CLUSTER(<cluster>) MQMNAME(<queue manager name>)` Use a high-level topic, but not the root. For information about cluster topic naming, see .

Migrating a WebSphere MQ Version 6.0 publish/subscribe hierarchy to a Version 7.0 publish/subscribe cluster - queue manager by queue manager

How to migrate an existing WebSphere MQ Version 6.0 hierarchy, where the parent and child queue managers are on separate computers, to a WebSphere Version 7.0 publish/subscribe cluster, one queue manager at a time.

Before you begin

To migrate the hierarchy, perform the following steps:

About this task

1. Create a new cluster or nominate an existing cluster, which need not be an existing publish/subscribe cluster. You can do this using WebSphere MQ Script commands (MQSC), or any other type of administration command or utility that is available on your platform, such as the WebSphere MQ Explorer. These methods are described in *WebSphere MQ Queue Manager Clusters*
2. Select the first queue manager to migrate into the publish/subscribe cluster. To cause the least disruption, select a queue manager that is a leaf node.

3. Install WebSphere MQ Version 7.0 on the computer that contains the selected queue manager in the hierarchy, to migrate the queue manager to WebSphere MQ Version 7.0.
4. Use the **strmqbrk** command on this queue manager to migrate all publish/subscribe configuration data into WebSphere MQ Version 7.0.
5. Join this queue manager into the cluster.
6. If this queue manager has a hierarchical relationship to an existing member of the cluster, use the MQSC command `ALTER QMGR PARENT(' ')` at the child queue manager to cancel the relationship. Before proceeding to the next step, to confirm that the hierarchical relationship has been cancelled, use the MQSC command `DISPLAY PUBSUB TYPE(PARENT)` at the child queue manager.
7. If this is the first queue manager to be migrated into the publish/subscribe cluster, define at least one cluster topic by executing the MQSC command `ALTER TOPIC(<topic name="">) PUBSCOPE(ALL) SUBSCOPE(ALL) CLUSTER(<cluster>)`

   Note: Use a high-level topic, but not the root. For information about cluster topic naming, see Cluster topics in *WebSphere MQ Publish/Subscribe User's Guide*.
8. To migrate the remainder of the hierarchy without introducing loops, repeat recursively from step 3 for each child queue manager that is not already in the cluster, and repeat recursively from step 3 for each parent queue manager that is not already in the cluster.

Example

What to do next

Migrating a WebSphere MQ Version 6.0 two queue manager publish/subscribe hierarchy to a Version 7.0 hierarchy - parent first

How to migrate an existing WebSphere MQ Version 6.0 hierarchy, where the parent and child queue managers are on separate computers, into a WebSphere Version 7.0 hierarchy, migrating the parent queue manager first.

Before you begin

To migrate the hierarchy, perform the following steps:

About this task

1. Install WebSphere MQ Version 7.0 on the computer that contains the parent queue manager in the hierarchy, to migrate the parent queue manager to WebSphere MQ Version 7.0.
2. Use the **strmqbrk** command on the parent queue manager to migrate all publish/subscribe configuration data into WebSphere MQ Version 7.0. At this point you can either run as a mixed WebSphere MQ Version 6.0 and Version 7.0 hierarchy or continue with the migration by upgrading the child in the next step.
3. Install WebSphere MQ Version 7.0 on the computer that contains the child queue manager in the hierarchy, to migrate the child queue manager to WebSphere MQ Version 7.0.
4. Use the **strmqbrk** command on the child queue manager to migrate all publish/subscribe configuration data into WebSphere MQ Version 7.0, and check the migration log to verify that the migration was successful. The migration log is in the queue manager directory: for example,

/var/mqm/<QMgrName> on Linux or C:\Program Files\IBM\WebSphereMQ\qmgrs\<QMgrName> on Windows, unless you have specified otherwise on the command line.

Example

What to do next

Migrating a WebSphere MQ Version 6.0 two publish/subscribe queue manager hierarchy to a Version 7.0 hierarchy - child first

How to migrate an existing WebSphere MQ Version 6.0 hierarchy, where the parent and child queue managers are on separate computers, into a WebSphere Version 7.0 hierarchy, migrating the child queue manager first.

About this task

To migrate the hierarchy, perform the following steps:

1. Install WebSphere MQ Version 7.0 on the computer that contains the child queue manager in the hierarchy, to migrate the child queue manager to WebSphere MQ Version 7.0.
2. Use the **strmqbrk** command on the child queue manager to migrate all publish/subscribe configuration data into WebSphere MQ Version 7.0. At this point you can either run as a mixed WebSphere MQ Version 6.0 and Version 7.0 hierarchy or continue with the migration by upgrading the parent in the next step.
3. Install WebSphere MQ Version 7.0 on the computer that contains the parent queue manager in the hierarchy, to migrate the parent queue manager to WebSphere MQ Version 7.0.
4. Use the **strmqbrk** command on the parent queue manager to migrate all publish/subscribe configuration data into WebSphere MQ Version 7.0, and check the migration log to verify that the migration was successful. The migration log is in the queue manager directory: for example, /var/mqm/<QMgrName> on Linux or C:\Program Files\IBM\WebSphereMQ\qmgrs\<QMgrName> on Windows, unless you have specified otherwise on the command line.

Using publish/subscribe with WebSphere MQ classes for JMS

Existing WebSphere MQ classes for JMS applications run unchanged after you upgrade your queue manager to Websphere MQ V7.0. In some circumstances, you must specify whether WebSphere MQ classes for JMS uses WebSphere MQ Version 6.0 or Version 7.0 publish/subscribe function.

The circumstances in which you must specify whether WebSphere MQ classes for JMS uses WebSphere MQ Version 6.0 or Version 7.0 publish/subscribe function are described later in this topic. The advantages of using WebSphere MQ Version 7.0 publish/subscribe function, compared with WebSphere MQ Version 6.0 Publish/Subscribe, WebSphere Event Broker, or WebSphere Message Broker, are introduced in *WebSphere MQ Using Java*.

WebSphere MQ messaging provider

The WebSphere MQ messaging provider has two modes of operation:

- *WebSphere MQ messaging provider normal mode*

- *WebSphere MQ messaging provider migration mode*

The WebSphere MQ messaging provider normal mode uses all the features of the WebSphere MQ Version 7.0 queue managers to implement JMS. This mode is used only to connect to a WebSphere MQ queue manager and can connect to WebSphere MQ Version 7.0 queue managers in either client or bindings mode. The WebSphere MQ messaging provider normal mode is optimized to use the new WebSphere MQ Version 7.0 function.

The WebSphere MQ messaging provider migration mode is based on WebSphere MQ Version 6.0 function and uses only features that were available in the WebSphere MQ Version 6.0 queue manager to implement JMS. You can connect to a WebSphere MQ Version 7.0 queue manager using WebSphere MQ messaging provider migration mode but you cannot use any of the Version 7.0 optimizations. This mode allows connections to either of the following queue manager versions:

- WebSphere MQ Version 7.0 queue manager in bindings or client mode, but this mode uses only those features that were available to a WebSphere MQ Version 6.0 queue manager
- WebSphere MQ Version 6.0 or earlier queue manager in client mode

If you want to connect to WebSphere Event Broker or WebSphere Message Broker using either WebSphere MQ Enterprise Transport or WebSphere MQ Real-Time Transport, use the WebSphere MQ messaging provider migration mode. If you use WebSphere MQ Real-Time Transport, the WebSphere MQ messaging provider migration mode is automatically selected, because you have explicitly selected properties in the connection factory object. Connection to WebSphere Event Broker or WebSphere Message Broker using the WebSphere MQ Enterprise Transport follows the general rules for mode selection described in "Rules for selecting the WebSphere MQ messaging provider mode."

Rules for selecting the WebSphere MQ messaging provider mode

If you are not using WebSphere MQ Real-Time Transport, the mode of operation used is determined primarily by the PROVIDERVERSION property of the connection factory. If you cannot change the connection factory you are using, you can use a client configuration property called com.ibm.msg.client.wmq.overrideProviderVersion, which overrides any setting on the connection factory. This override applies to all connection factories in the JVM but the actual connection factory objects are not modified. You can set PROVIDERVERSION to three possible values: 7, 6, or unspecified:

PROVIDERVERSION=7

Uses the WebSphere MQ messaging provider normal mode.

If you set PROVIDERVERSION to 7 only the WebSphere MQ messaging provider normal mode of operation is available. If the queue manager that is connected to as a result of the other settings in the connection factory is not a Version 7.0 queue manager, the createConnection() method fails with an exception.

The WebSphere MQ messaging provider normal mode uses the sharing conversations feature, and the number of conversations that can be shared is controlled by the SHARECNV() property on the server connection channel. If this property is set to 0, you cannot use WebSphere MQ messaging provider normal mode and the createConnection() method fails with an exception.

PROVIDERVERSION=6

Uses the WebSphere MQ messaging provider migration mode.

The WebSphere MQ classes for JMS use the features and algorithms supplied with WebSphere MQ Version 6.0. If you want to connect to WebSphere Event Broker or WebSphere Message Broker using WebSphere MQ Enterprise Transport, you must use this mode. You can connect to a WebSphere MQ Version 7.0 queue manager using this mode, but none of the new features of a Version 7.0 queue manager are used, for example, read ahead or streaming.

PROVIDERVERSION=unspecified

This is the default value and the actual text is ″unspecified″.

A connection factory that was created with a previous version of WebSphere MQ classes for JMS in JNDI takes this value when the connection factory is used with V7.0 of WebSphere MQ classes for JMS. The following algorithm is used to determine which mode of operation is used. This algorithm is used when the createConnection() method is called and uses other aspects of the connection factory to determine if WebSphere MQ messaging provider normal mode or WebSphere MQ messaging provider migration mode is required.

- Firstly, an attempt to use WebSphere MQ messaging provider normal mode is made.
- If the queue manager connected is not WebSphere MQ Version 7.0, the connection is closed and WebSphere MQ messaging provider migration mode is used instead.
- If the SHARECNV() property on the server connection channel is set to 0, the connection is closed and WebSphere MQ messaging provider migration mode is used instead.
- If BROKERVER is set to V1 or unspecified, WebSphere MQ messaging provider normal mode continues to be used, and therefore any publish/subscribe operations use the new WebSphere MQ V7.0 features.

  If WebSphere Event Broker or WebSphere Message Broker are used in compatibility mode (and you want to use Version 6.0 publish/subscribe function rather than the WebSphere MQ Version 7 publish/subscribe function), set PROVIDERVERSION to 6 to ensure WebSphere MQ messaging provider migration mode is used.
- If BROKERVER is set to V2 and BROKERQMGR is nonblank, this means BROKERQMGR has been explicitly changed from the default, so the assumption is the connection factory really is intended for use with WebSphere Event Broker or WebSphere Message Broker and WebSphere MQ Enterprise Transport. Therefore WebSphere MQ messaging provider migration mode is used.
- If BROKERVER is set to V2, BROKERQMGR is blank, the specified BROKERCONQ command queue exists and can be opened for output (that is, MQOPEN for output succeeds), and PSMODE on the queue manager is set to COMPAT or DISABLED, WebSphere MQ messaging provider migration mode is used.

You can find further guidance about using PROVIDERVERSION in “When to use PROVIDERVERSION ” on page 143.

When to use PROVIDERVERSION

There are two scenarios where you cannot use the algorithm described in "Rules for selecting the WebSphere MQ messaging provider mode" on page 141; consider using PROVIDERVERSION in these scenarios.

1. If WebSphere Event Broker or WebSphere Message Broker is in compatibility mode, you must specify PROVIDERVERSION for them to work correctly.
2. If you are using WebSphere Application Server Version 6.0.1, Version 6.0.2, or Version 6.1, you define connection factories using the WebSphere Application Server administrative console.

   In WebSphere Application Server the default value of the BROKERVER property on a connection factory is V2. The default BROKERVER property for connection factories created by using JMSAdmin or WebSphere MQ Explorer is V1. This property is now ″unspecified″ in WebSphere MQ Version 7.0.

If BROKERVER is set to V2 (either because it was created by WebSphere Application Server or the connection factory has been used for publish/subscribe before) and the existing queue manager has a BROKERCONQ defined (because it has been used for publish/subscribe messaging before), the WebSphere MQ messaging provider migration mode is used.

However, if you want the application to use peer-to-peer communication and the application is using an existing queue manager that has previously been used for publish/subscribe, and has a connection factory with BROKERVER set to 2 (which is the default if the connection factory was created in WebSphere Application Server), the WebSphere MQ messaging provider migration mode is used. Using WebSphere MQ messaging provider migration mode in this case is unnecessary; use WebSphere MQ messaging provider normal mode instead. You can use one of the following methods to work around this:

- Set BROKERVER to V1 or unspecified. This is dependent on your application.
- Set PROVIDERVERSION to 7; this is a custom property in WebSphere Application Server Version 6.1. The option to set custom properties in WebSphere Application Server Version 6.1 and later is not currently documented in the WebSphere Application Server Information Center.

  Alternatively, use the client configuration property (see "Rules for selecting the WebSphere MQ messaging provider mode" on page 141 for details about how you can specify this system property for all environments), or modify the queue manager connected so it does not have the BROKERCONQ, or make the queue unusable.

Subscription name migration on the JMS client

On the JMS client, if the ConnectionFactory property brokerPubQ is not the default, WebSphere MQ adds the stream name to the subscription name.

In WebSphere MQ Version 6.0, a subscription name needed to be unique only within the stream and not across the queue manager. In WebSphere MQ Version 7.0, a subscription name must be unique across the queue manager. Therefore to migrate WebSphere MQ Version 6.0 durable subscriptions to WebSphere MQ Version 7.0, the subscription names must be unique. WebSphere MQ does this when it migrates the queue manager, by appending the stream name to the existing subscription name. For any existing durable subscription that uses a stream other than the default of ″SYSTEM.BROKER.DEFAULT.STREAM″ the migration process appends the stream name to the subscription name.

On the JMS client, if the ConnectionFactory property brokerPubQ is not the default, it is assumed that a WebSphere MQ Version 6.0 durable subscription is being resumed, and WebSphere MQ Version 7.0 appends the stream name to match the action of the migration process. Subscription names that use the default stream are migrated across with the subscription name unchanged.

Migration implications of mapping an alias queue to a topic object

WebSphere MQ Version 7.0 introduces an extension to the alias queue object that allows an alias queue to be mapped to a topic object.

The new TARGTYPE attribute allows you to specify that a queue alias resolves to a queue or a topic. The TARGQ attribute, defined in WebSphere MQ Version 6.0 as the name of the queue to which the alias queue resolves, is renamed to TARGET in WebSphere MQ Version 7.0 and generalized to allow you to specify the name of either a queue or a topic. The attribute name TARGQ is retained for compatibility with your existing programs.

This feature is useful for migrating your existing applications to a publish/subscribe message model.

A useful example of this feature is the queue to which statistics messages are written. Prior to WebSphere MQ Version 7.0 there could be only a single consumer of a statistic message because a single statistics message only was written to a queue and got from a queue.

By defining a queue alias that points to a topic object, it is possible for each person interested in processing statistics messages to subscribe to the topic, rather than getting from the queue, allowing multiple consumers of the statistics information.

Within a queue sharing group it is possible to define a queue alias as a group object - this means that each queue manager in the queue sharing group will create a queue alias definition with the same name and the same properties as the QSGDISP(GROUP) object.

The new TARGTYPE attribute may be set or altered in a QSGDISP(GROUP) object by a new Version 7.0 queue manager, so that the queue alias refers to a topic object. However, any Version 6 queue managers in the queue sharing group do not understand and will ignore the new TARGTYPE attribute. A V6 queue manager will interpret the queue alias as referring to a queue object, regardless of the setting of TARGTYPE.

Defining a queue alias is described in *WebSphere MQ System Administration Guide*.

Migrated topologies

If you have a WebSphere MQ publish/subscribe broker network, you can continue to use this network unchanged. The introduction of WebSphere Message Broker or WebSphere Event Broker to your environment, and the creation of brokers in that broker domain, does not affect your WebSphere MQ publish/subscribe broker domain until you take specific action to connect the two networks.

If you want to have two separate, independent networks, you do not have to do anything. You can retain your existing WebSphere MQ publish/subscribe network, and install and configure a WebSphere Message Broker or WebSphere Event Broker network, without any interaction.

Heterogeneous networks

A *heterogeneous network* is a network of brokers, some of which form a WebSphere MQ publish/subscribe network and some of which belong to WebSphere Message Broker or WebSphere Event Broker.

With WebSphere Message Broker and WebSphere Event Broker, there are two ways in which a broker can be joined to the WebSphere MQ publish/subscribe network; it can be joined as a leaf node or as a parent node.

Leaf node

When a broker is joined as a leaf node, it is joined as a child broker of another broker in the WebSphere MQ publish/subscribe network.

Adding the broker as a leaf node rather than as a parent node causes the new broker to receive only some of the WebSphere MQ publish/subscribe message traffic that is directed to the brokers for which this new broker is a child broker.

Parent node

When a broker is joined as a parent node, it is joined as a parent broker of one or more brokers in the WebSphere MQ publish/subscribe network.

Adding the broker as a parent node rather than as a leaf node causes the new broker to receive all the WebSphere MQ publish/subscribe message traffic that is directed to the child brokers for which this new broker is the parent broker.

Notices

This information was developed for products and services offered in the United States. IBM® may not offer the products, services, or features discussed in this information in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this information. The furnishing of this information does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing,
IBM Corporation,
North Castle Drive,
Armonk, NY 10504-1785,
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation,
Licensing,
2-31 Roppongi 3-chome, Minato-k,u
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM United Kingdom Laboratories,
Mail Point 151,
Hursley Park,
Winchester,
Hampshire,
England
SO21 2JN.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AIX	DB2 Universal Database	i5/OS
IBM	IBMLink	MQSeries
OS/2	RACF	SupportPac
Tivoli	WebSphere	z/OS

Java™ and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft®, Windows, Windows NT®, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

UNIX® is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Index

U

W

Sending your comments to IBM

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book.

Please limit your comments to the information in this book and the way in which the information is presented.

To make comments about the functions of IBM products or systems, talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM , you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate, without incurring any obligation to you.

You can send your comments to IBM in any of the following ways:

- By mail, to this address:

  User Technologies Department (MP095)
  IBM United Kingdom Laboratories
  Hursley Park
  WINCHESTER,
  Hampshire
  SO21 2JN
  United Kingdom
- By fax:
  - From outside the U.K., after your international access code use 44-1962-816151
  - From within the U.K., use 01962-816151
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink™: HURSLEY(IDRCF)
  - Internet: idrcf@hursley.ibm.com

Whichever method you use, ensure that you include:

- The publication title and order number
- The topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

SC34-6950-01

Spine information:

WebSphere MQ Publish/Subscribe User's Guide Version 7.0