Oracle8*i*

Application Developer's Guide - Advanced Queuing

Release 8.1.5

February 1999

Part No. A68005-01

Application Developer's Guide - Advanced Queuing, Release 8.1.5

Part No. A68005-01

Copyright © 199x, 1999, Oracle Corporation. All rights reserved.

Primary Author: Denis Raphaely

Contributors: Neerja Bhatt, Sashi Chandrasekaran, Dieter Gawlick, John Gibb, Mohan Kamath, Krishnan Meiyyappan, Bhagat Nainani, Goran Olsson, Madhu Reddy, Mary Rhodes, Ashok Saxena, Ekrem Soylemez, Alvin To, Rahim Yaseen

Graphic Designer: Valerie Moore

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and Oracle disclaims liability for any damages caused by such use of the Programs.

The Programs (which include both the software and documentation) contain proprietary information of Oracle Corporation; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Oracle Corporation.

If the Programs are delivered to the U.S. Government or anyone licensing or using the Programs on behalf of the U.S. Government, the following notice is applicable:

Restricted Rights Notice Programs delivered subject to the DOD FAR Supplement are "commercial computer software" and use, duplication, and disclosure of the Programs including documentation, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, Programs delivered subject to the Federal Acquisition Regulations are "restricted computer software" and use, duplication, and disclosure of the Programs shall be subject to the restrictions in FAR 52.227-19, Commercial Computer Software - Restricted Rights (June, 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

Oracle, Pro*Ada, Pro*COBOL, Pro*FORTRAN, SQL*Loader, SQL*Net and SQL*Plus are registered trademarks of Oracle Corporation, Redwood City, California.

Designer/2000, Developer/2000, Net8, Oracle Call Interface, Oracle7, Oracle8, Oracle8*i*, Oracle Forms, Oracle Parallel Server, PL/SQL, Pro*C, Pro*C/C++ and Trusted Oracle are trademarks of Oracle Corporation, Redwood City, California.

All other products or company names are used for identification purposes only, and may be trademarks of their respective owners.

Contents

1 Introduction

3 Managing Oracle AQ

4 Administrative Interface: Basic Operations

5 Administrative Interface: Views

6 Operational Interface: Basic Operations

7 Advanced Queuing — Java API

8 Oracle Advanced Queuing by Example

A Scripts for Implementing ’BooksOnLine’

Index

Send Us Your Comments

Application Developer's Guide - Advanced Queuing, Release 8.1.5

Part No. A68005-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the chapter, section, and page number (if available). You can send comments to us in the following ways:

- electronic mail - infodev@us.oracle.com
- FAX - (650) 506-7228
- postal service:

  Oracle Corporation
  Oracle Server Documentation Manager
  500 Oracle Parkway
  Redwood Shores, CA 94065
  USA

If you would like a reply, please give your name, address, and telephone number below.

Preface

This Guide describes features of application development on the Oracle Server having to do with *Oracle Advanced Queuing*, Release 8.1.5. Information in this Guide applies to versions of the Oracle Server that run on all platforms, and does not include system-specific information.

The Preface includes the following sections:

- Information in This Guide
- Feature Coverage and Availability
- New Features Introduced with Oracle 8.1
- Other Guides
- How This Book Is Organized
- Visual Modelling
- Conventions Used in this Guide
- Your Comments Are Welcome

Information in This Guide

Oracle Advanced Queueing (Oracle AQ) provides message queuing as an integrated part of the Oracle server. Oracle AQ provides this functionality by integrating the queuing system with the database, thereby creating a *message-enabled database*. By providing an integrated solution Oracle AQ frees application developers to devote their efforts to their specific business logic rather than having to construct a messaging infrastructure.

The *Oracle8i Application Developer's Guide - Advanced Queuing* is intended for programmers developing new applications that use Oracle Advanced Queuing, as well as those who have already implemented this technology and now wish to take advantage of new features.

The increasing importance of Oracle AQ has led to its being presented as an independent volume within the Oracle Application Developers documentation set.

Feature Coverage and Availability

For information about the differences between Oracle8 and the Oracle8 Enterprise Edition and the features and options that are available to you, see *Getting to Know Oracle8i*.

New Features Introduced with Oracle 8.1

- Queue Level Access Control
- Non-Persistent Queues
- Support for OPS Environments
- Rule-based Subscribers forPublish / Subscribe
- Asynchronous Notification
- Sender Identification
- Listen Capability (Wait on Multiple Queues)
- Propagation of Messages with LOBs
- Enhanced Propagation Scheduling Capabilities
- Dequeue Message Header Only With No Payload
- Support for Statistics Views

- Separate storage of history management information

For more information about Oracle AQ features, see:

- Chapter 2, "Implementing AQ — A Sample Application"

Other Guides

Use the *PL/SQL User's Guide and Reference* to learn PL/SQL and to get a complete description of this high-level programming language, which is Oracle Corporation's procedural extension to SQL.

The Oracle Call Interface (OCI) is described in:

- *Oracle Call Interface Programmer's Guide*

You can use the OCI to build third-generation language (3GL) applications that access the Oracle Server.

Oracle Corporation also provides the Pro* series of precompilers, which allow you to embed SQL and PL/SQL in your application programs. If you write 3GL application programs in Ada, C, C++, COBOL, or FORTRAN that incorporate embedded SQL, refer to the corresponding precompiler manual. For example, if you program in C or C++, refer to the *Pro*C/C++ Precompiler Programmer's Guide*.

For SQL information, see the *Oracle8i SQL Reference* and *Oracle8i Administrator's Guide*. For basic Oracle concepts, see *Oracle8i Concepts*.

How This Book Is Organized

The *Application Developer's Guide* - *Advanced Queuing* contains seven chapters and an appendix. A brief summary of what you will find in each chapter follows:

Chapter 1, "Introduction"

This chapter 'sets the bar' by describing the requirements for optimal messaging systems. Although Oracle AQ is a relatively new technology, and not all these goals have been realized, you can get an overview of the underlying design and a clear idea of the intended direction.

Chapter 2, "Implementing AQ — A Sample Application"

This chapter describes features already present in Oracle AQ under three headings: General Features, Enqueue Features, and Dequeue Features.

Chapter 3, "Managing Oracle AQ"

This chapter describes the primary queuing entities (message, queue, queue table, agent, queue monitor), and the basics of connecting single/multiple producers of messages with single/multiple consumers of messages. Of particular interest is the way messages can directed toward specific subscribers implicitly, explicitly or on the basis of rules.

Chapter 4, "Administrative Interface: Basic Operations"

As its title indicates, this chapter presents the basic operations underlying the Administrative interface, such as `Create Queue Table`, `Create Queue`, `Grant Queue Privilege`, `Add a Subscriber`, and `Schedule a Propagation`. We have introduced a new way of presenting this information that utilizes the Unified Modelling Language (detailed notes are included below). On-line users will additionally be able to make use of hypertext links and image-based hot links

Chapter 5, "Administrative Interface: Views"

This chapter is dedicated to the various views that Oracle has provided for administrators and users that are projected as a result of queries, such as `Select All Queue Tables in the Database`, `Select Messages in a Queue Table`, and `Select Queue Subscribers and their Rules`.

Chapter 6, "Operational Interface: Basic Operations"

We here describe the essentials of the operational interface in terms of the basic operations concerned with enqueuing a message, dequeuing a message, registering for messages based on defined rules, and listening to one or more queues for messages.

Chapter 7, "Advanced Queuing — Java API"

This chapter introduces and details the Java Application Programmer's Interface for Advanced Queuing.

Chapter 8, "Oracle Advanced Queuing by Example"

As you can see by examining the Table of Contents, small examples are interspersed throughout the text, but this chapter is dedicated solely to providing examples in both PL/SQL and OCI.

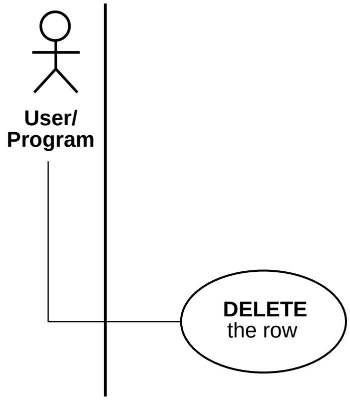
Chapter A, "Scripts for Implementing 'BooksOnLine'"

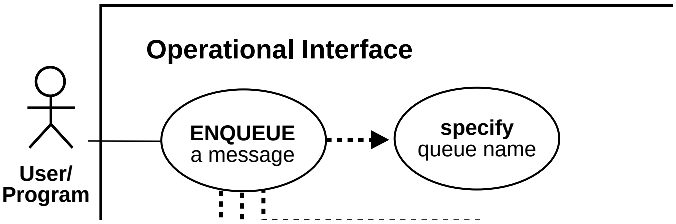
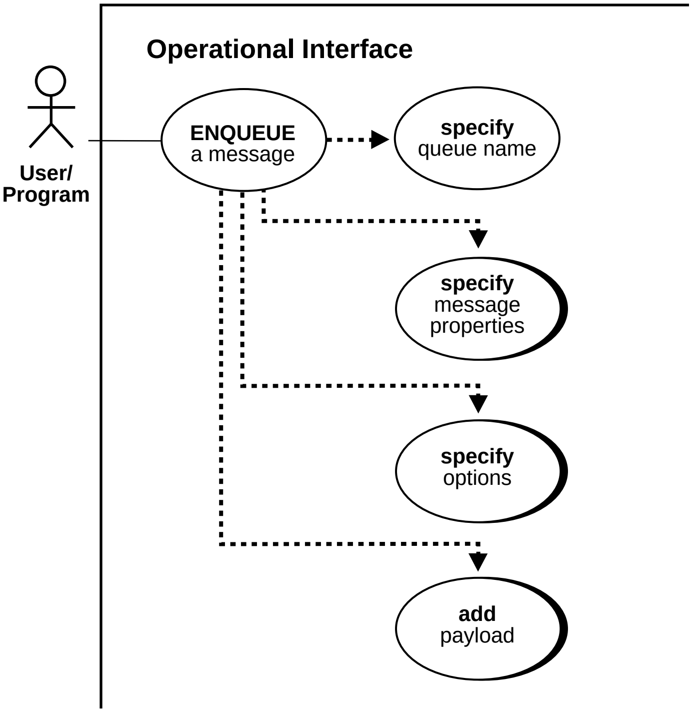
This appendix has the scripts for implementing the sample application, `BooksOnLine`.

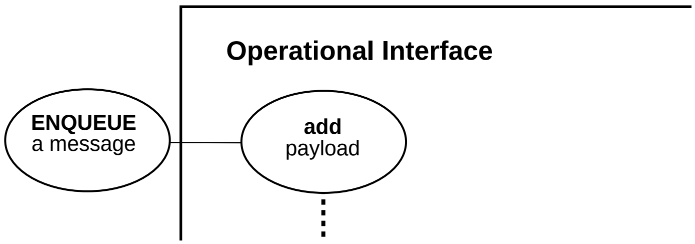
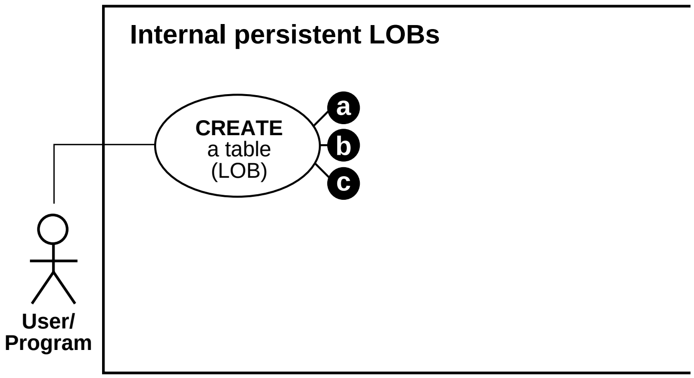
Visual Modelling

This release introduces the Universal Modeling Language (UML) as a way of explaining the technology that we hope will help you develop applications. A full presentation of the UML is beyond the scope of this documentation set, however we do provide a description of the subset of UML notation that we use in a chapter devoted to visual modelling in*Oracle8i Application Developer's Guide - Fundamentals*. What follows here is a selection from that chapter of those elements that are used in this book.

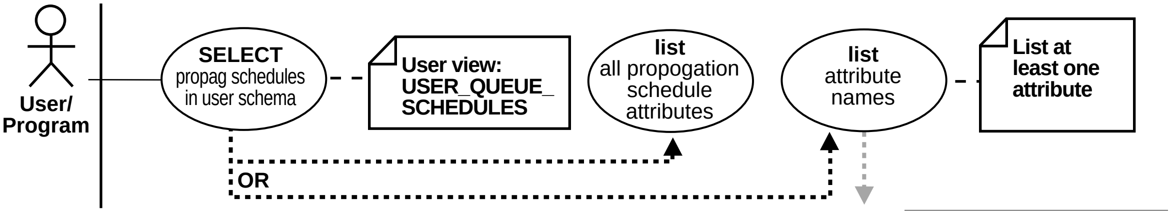
Use Case Diagrams

Graphic Element	Description
	This release of the documentation introduces and makes heavy use of the *Use Case Diagram*. Each primary use case is instigated by an *actor* ('stickman') that could be a human user, an application, or a sub-program. The actor is connected to the primary use case which is depicted as an oval (bubble) enclosing the use case action. The totality of primary use cases is described by means of a *Use Case Model Diagram*.

Graphic Element	Description
	Primary use cases may require other operations to complete them. In this diagram fragment ■ **`specify`** `queue name` is one of the sub-operations, or secondary use cases, needed to complete ■ **`ENQUEUE`** `a message` The downward lines from the primary use case lead to the other required operations (not shown).
	Secondary use cases that have drop shadows 'expand' in that they are described by means of their own use case diagrams. There are two reasons for doing this: (a) it makes it easier to understand the logic of the operation; (b) it would not have been possible to place all the operations and sub-operations on the same page. In this example ■ **`specify`** `message properties,` ■ **`specify`** `options` ■ **`add`** `payload` are all expanded in separate use case diagrams.

Graphic Element	Description
	This diagram fragment shows the use case diagram expanded. While the standard diagram has the actor as the initiator), here the use case itself is the point of departure for the sub-operation. In this example, the expanded view of ■ **`add`** `payload` represents a constituent operation of ■ **`ENQUEUE`** `a message`
	This convention (a, b, c) shows that there are three different ways of creating a table that contains LOBs.
	This fragment shows one of the uses of a NOTE box, here distinguishing which of the three ways of creating a table containing LOBs is being presented.

Graphic Element

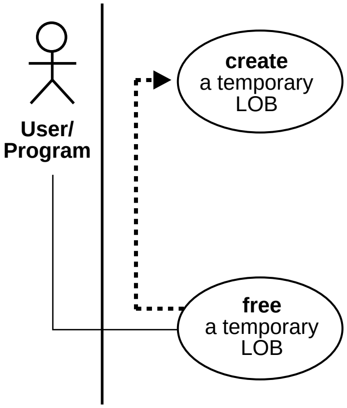
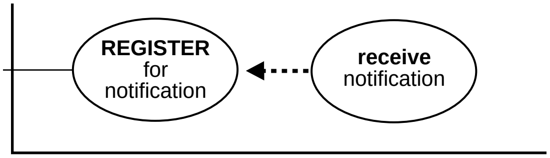


Description

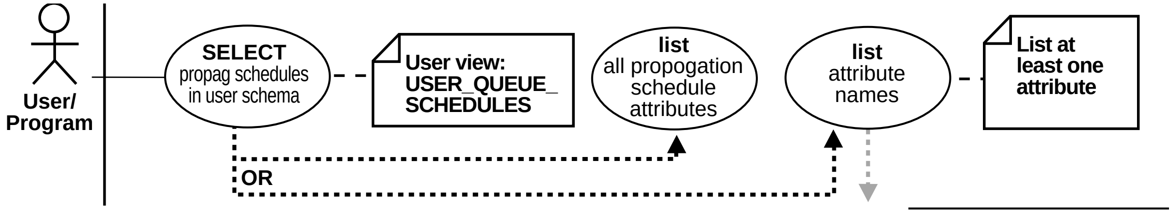
This drawing shows two other common use of NOTE boxes:

(a) as a way of presenting an alternative name, as in this case the action **`SELECT`** `propagation schedules` in `the user schema` is represented by the view `USER_QUEUE_SCHEDULES`

(b) the action **`list`** `attribute names` is qualified by the note to the user that you must list at least one attribute if you elect not to list all the propagation schedule attributes.

Graphic Element	Description
	The dotted arrow in the use case diagram indicates dependency. In this example ■ **`free`** `a temporary LOB` requires that you first ■ **`create`** `a temporary LOB` Put another way: you should not execute the **`free`** operation on a LOB that is not temporary. What you need to remember is that the target of the arrow shows the operation that must be performed first.
	Use cases and their sub-operations can be linked in complex relationships. In this example of a callback, you must earlier ■ **`REGISTER`** `for notification` in order to later ■ **`receive`** `a notification`

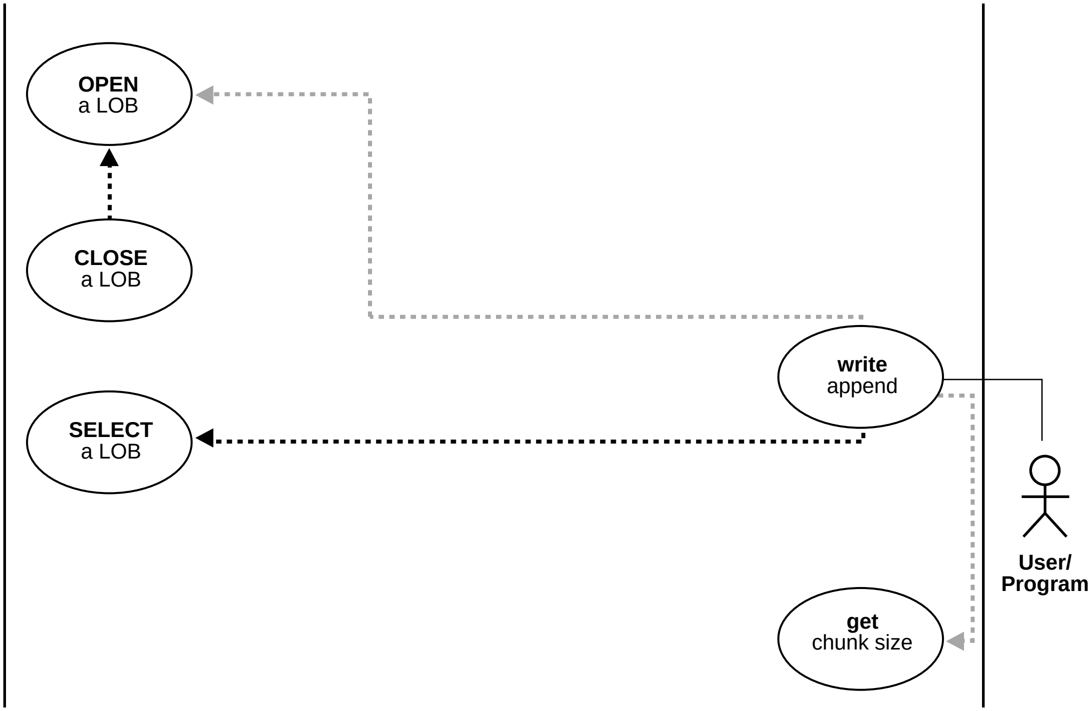
Graphic Element



Description

In this case the branching paths of an OR condition are shown. In invoking the view, you may choose either to list all the attributes or you may view one or more attributes. The fact that you may stipulate which of the attributes you wish made visible is indicated by the grayed arrow.

Graphic Element



Description

Not all linked operations are mandatory. While the black dashed-line and arrow indicate that you must perform the targeted operation to complete the use case, actions that are optional are shown by the grey dashed-line and arrow. In this example, executing

- **`write`** append

on a `LOB` requires that you first

- **`SELECT`** a `LOB`

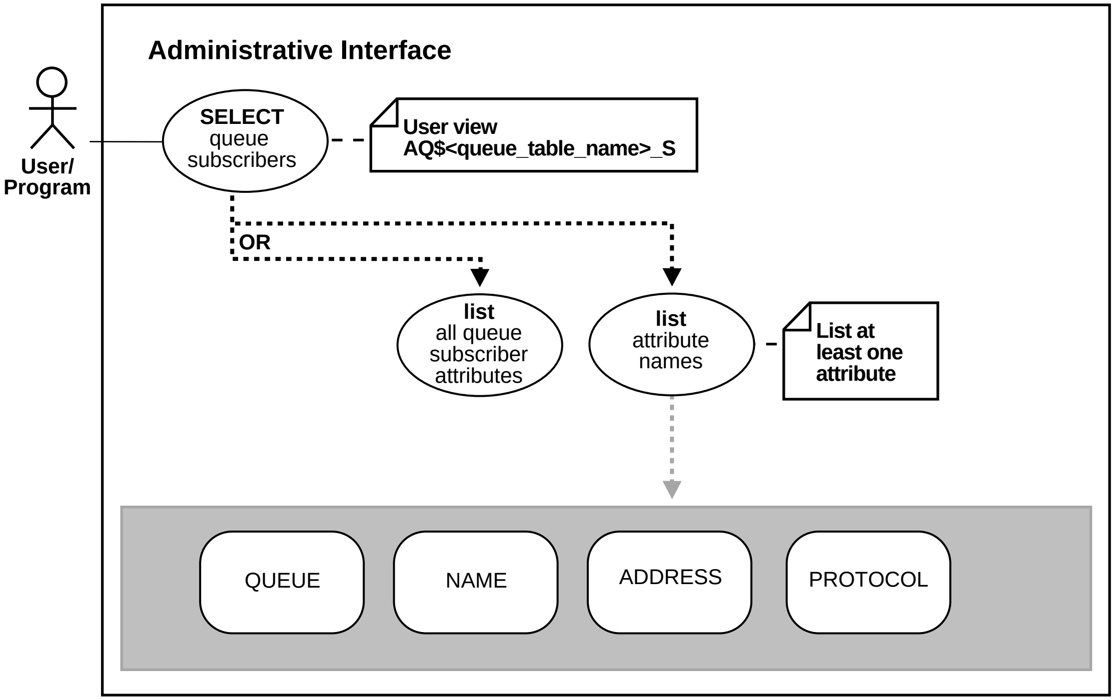
As a facilitating operations, you may choose to

- **`OPEN`** a `LOB` and / or **`get`** `chunk` size

However, note that if you do **`OPEN`** a `LOB`, you will later have to **`CLOSE`** `it`.

State Diagrams

Graphic Element



Description

All the previous notes have dealt with *use case diagrams*. Here we introduce the very basic application of a *state diagram* that we utilize in this book to present the attributes of view. In fact, attributes of a view have only two states — visible or invisible. We are not interested in showing the permutations of state but in showing what you might make visible in invoking a view. Accordingly, we have extended the UML to join a partial state diagram onto a use case diagram to show the totality of attributes, and thereby all the view sub-states of the view that you can see. We have demarcated the use case from the view state by coloring the background of the state diagram grey.

In this example, the view `AQ$<queue_table_name>_S` allows you to query queue subscribers. You can stipulate one attribute, or some combination of the four attributes, or all of the four attributes.

Graphic Element

Description

Internal temporary LOBs (part 1 of 2)

Use Case Model Diagrams summarize all the use cases in a particular domain, such as `Internal temporary LOBs`. Often these diagrams are too complex to contain within a single page. When that happens we have resorted to dividing the diagram into two parts. Please note that there is no sequence implied in this division.

continued on next page

In some cases we have had to split a diagram simply because it is too long for the page. In such cases, we have included this marker.

Conventions Used in this Guide

The following notational and text formatting conventions are used in this guide:

[ ]
Square brackets indicate that the enclosed item is optional. Do not type the brackets.

{ }
Braces enclose items of which only one is required.

|
A vertical bar separates items within braces, and may also be used to indicate that multiple values are passed to a function parameter.

...
In code fragments, an ellipsis means that code not relevant to the discussion has been omitted.

`font change`
SQL or C code examples are shown in monospaced font.

italics
Italics are used for OCI parameters, OCI routines names, file names, and data fields.

UPPERCASE
Uppercase is used for SQL keywords, like `SELECT` or `UPDATE`.

This guide uses special text formatting to draw the reader's attention to some information. A paragraph that is indented and begins with a bold text label may have special meaning. The following paragraphs describe the different types of information that are flagged this way.

> **Note:** The "Note" flag indicates that the reader should pay particular attention to the information to avoid a common problem or increase understanding of a concept.

> **Warning:** An item marked as "Warning" indicates something that an OCI programmer must be careful to do or not do in order for an application to work correctly.

> **See Also:** Text marked "See Also" points you to another section of this guide, or to other documentation, for additional information about the topic being discussed.

Your Comments Are Welcome

We value and appreciate your comment as an Oracle user and reader of our manuals. As we write, revise, and evaluate our documentation, your opinions are the most important feedback we receive.

You can send comments and suggestions about this manual to the following e-mail address:

infodev@us.oracle.com

If you prefer, you can send letters or faxes containing your comments to the following address:

Server Technologies Documentation Manager

Oracle Corporation

500 Oracle Parkway

Redwood Shores, CA 94065

Fax: (650) 506-7228

1

Introduction

In this chapter we introduce Oracle Advanced Queuing (AQ) by considering the requirements for complex information handling in a distributed environment under the following headings:

- The Need for Queuing in Messaging Systems
- Features of Advanced Queuing (AQ)
- Primary Components of Advanced Queuing (AQ)
- Modeling Queue Entities
- Programmatic Environments for Working with AQ
- AQ and XA
- Compatibility
- Restrictions
- Reference to Demos

The Need for Queuing in Messaging Systems

Consider the following application scenario.

The operations of a large bookseller, `BooksOnLine`, are based on an online book ordering system which automates activities across the various departments involved in the entire sale process.The front end of the system is an order entry application which is used to enter new orders.These incoming orders are processed by an order processing application which validates and records the order. Shipping departments located at regional warehouses are then responsible for ensuring that these orders are shipped in a timely fashion. There are three regional warehouses: one serving the East Region, one serving the West Region, and a third warehouse for shipping International orders. Once an order has been shipped, the order information is routed to a central billing department which handles payment processing.The customer service department, located at its own site, is responsible for maintaining order status and handling inquiries about orders.

Message Systems

This scenario describes an application in which **messages** come from and are disbursed to multiple clients (nodes) in a distributed computing environment. Messages are not only passed back and forth between clients and servers but are also intricately interleaved between processes on different servers. The integration of the various component applications consist of multi-step processes in which each step is triggered by one or more messages, and which may then give rises to one or more messages.

These applications can be viewed as message systems. For instance, the application should be able to implement content-based routing, content-based subscription, and content-based querying.

Such message systems need to exhibit high performance characteristics as might be measured by the following metrics:

- Number of messages enqueued / dequeued per second.
- Time to evaluate a complex query on a message warehouse.
- Time to recover / restart the messaging process after a failure.

Message systems should also exhibit high scalability. A system should continue to exhibit high performance as the number of programs using the application increase, as the number of messages increase, and as the size of the message warehouse increases.

Synchronous Communication as an Application Model

One way of modeling this intercommunication of messages, termed *synchronous*, *on-line* or *connected,* is based on the request-reply paradigm. In this model a program sends a request to another program and waits (blocks) until the reply arrives. This close coupling of the sender and receiver of the message is suitable for programs that need to get a reply before they can proceed.

Traditional client/server architectures are based on this model.Its major drawback is that all the component programs must be available and running for the application to work. In the event of network or machine failure, or even if the needed program is busy, the entire application grinds to a halt.

Asynchronous Messaging as an Application Model

In the *asynchronous, disconnected* or *deferred* model programs in the role of producers place messages in a **queue** and then proceed with their work. Programs in the role of consumers retrieve requests from the queue and act on them. This model is well suited for applications that can continue with their work after placing a request in the queue because they are not blocked waiting for a reply. It is also suited to applications that can continue with their work until there is a message to retrieve.This decoupling of ’requests for service’ from ’supply of services’ increases efficiency, and provides the infrastructure for complex scheduling.

Message Persistence

Handling an intricate scheduling of message-passing is not the only challenge. Unfortunately, networks, computing hardware, and software applications will all fail from time to time. For deferred execution to work correctly in the presence of network, machine and application failures, messages that constitute requests for service must be stored persistently, and processed exactly once.In other words, messaging must be persistent.

Being able to preserve messages is fundamental. Applications may have to deal with multiple unprocessed messages arriving simultaneously from external clients or from programs internal to the application, and in such situations they may not have the necessary resources. Similarly, the communication links between databases may not be available all the time or may be reserved for some other purpose. If the system falls short in its capacity to deal with these messages immediately, the application must be able to store the messages until they can be processed. By the same token, external clients or internal programs may not be ready to receive messages that have been processed.

Even more importantly, messaging systems need message persistence so they can deal with priorities: messages arriving later may be of higher priority than messages arriving earlier; messages arriving earlier may have to wait for messages arriving later before actions are executed; the same message may have to be accessed by different processes; and so on. Such priorities may not be fixed. One crucial dimension of handling the dynamic aspect of message persistence has to do with windows of opportunity that grow and shrink.It may be that messages in a specific queue become more important than messages in other queues, and so need to be processed with less delay or interference from messages in other queues. Similarly, it may be more pressing to send messages to some destinations than to others.

Finally, message persistence is crucial because the control component of the message can be as important as the payload data. For instance, the time that messages are received or dispatched can be a crucial part of the message. It may be central to analyzing periods of greatest demand, or for evaluating the lag between receiving and completing an order, and son on. Put more formally: the message may need to retain importance as a business asset after it has been executed.Tracking and documentation should be the responsibility of the messaging system, not the developer.

Features of Advanced Queuing (AQ)

By integrating transaction processing with queuing technology, persistent messaging in the form of **Advanced Queuing** is made possible. The following overview considers the features of Oracle AQ under four headings:

- "General Features" on page 1-5
- "ENQUEUE Features" on page 1-8
- "DEQUEUE Features" on page 1-11
- "Propagation Features" on page 1-13

General Features

The following features apply to all aspects of Oracle AQ.

SQL Access

Messages are placed in normal rows in a database table, and so can be queried using standard SQL. This means that you can use SQL to access the message properties, the message history and the payload. All available SQL technology, such as indexes, can be used to optimize the access to messages.

Integrated Database Level Operational Support

Standard database features such as recovery, restart and enterprise manager are supported. Oracle AQ queues are implemented in database tables, hence all the operational benefits of high availability, scalability and reliability are applicable to queue data. In addition, database development and management tools can be used with queues. For instance, queue tables can be imported and exported.

Structured Payload

Users can use object types to structure and manage message payloads. RDBMSs in general have had a far richer typing system than messaging systems. Since Oracle8*i* is an object-relational DBMS, it supports both traditional relational types as well as user-defined types. Many powerful features are enabled as a result of having strongly typed content i.e. content whose format is defined by an external type system. These include:

- Content-based routing: an external agent can examine the content and route the message to another queue based on the content.

- Content-based subscription: a publish and subscribe system built on top of a messaging system which can offer content based on subscription.
- Querying: the ability to execute queries on the content of the message enables message warehousing.

To see this feature applied in the context of the BooksOnLine scenario, refer to Structured Payload on page 2-7 in Chapter 2, "Implementing AQ — A Sample Application")

Retention and Message History

Users of AQ can specify that messages be retained after consumption. The systems administrator can specify the duration for which messages will be retained. Oracle AQ stores information about the history of each message, preserving the queue and message properties of delay, expiration, and retention for messages destined for local or remote recipients. The information contains the `ENQUEUE`/`DEQUEUE` time and the identification of the transaction that executed each request. This allows users to keep a history of relevant messages. The history can be used for tracking, data warehouse and data mining operations.

To see this feature applied in the context of the BooksOnLine scenario, refer to Retention and Message History on page 2-20)

Tracking and Event Journals

If messages are retained they can be related to each other. For example: if a message `m2` is produced as a result of the consumption of message `m1`, `m1` is related to `m2`. This allows users to track sequences of related messages. These sequences represent ’event journals’ which are often constructed by applications. Oracle AQ is designed to let applications create event journals automatically.

Integrated Transactions

The integration of control information with content (data payload) simplifies application development and management.

Queue Level Access Control

With Oracle 8*i*, an owner of an 8.1 style queue can grant or revoke queue level privileges on the queue. DBAs can grant or revoke new AQ system level privileges to any database user. DBAs can also make any database user an AQ administrator.

To see this feature applied in the context of the BooksOnLine scenario, refer to Queue Level Access Control on page 2-9).

Non-Persistent Queues

AQ can deliver non-persistent messages asynchronously to subscribers. These messages can be event-driven and do not persist beyond the failure of the system (or instance). AQ supports persistent and non-persistent messages with a common API.

To see this feature applied in the context of the BooksOnLine scenario, refer to Non-Persistent Queues on page 2-10).

Publish/ Subscribe Support

A combination of features are introduced to allow a publish/subscribe style of messaging between applications. These features include rule-based subscribers, message propagation, the listen feature and notification capabilities.

Support for OPS Environments

With Oracle8*i* release 8.1.5, an application can specify the instance affinity for a queue-table. When AQ is used with parallel server and multiple instances, this information is used to partition the queue-tables between instances for queue-monitor scheduling. The queue-table is monitored by the queue-monitors of the instance specified by the user. If an instance affinity is not specified, the queue-tables will be arbitrarily partitioned among the available instances. There can be ’pinging’ between the application accessing the queue-table and the queue-monitor monitoring it. Specifying the instance-affinity does not prevent the application from accessing the queue-table and its queues from other instances.

This feature prevents ’pinging’ between queue monitors and AQ propagation jobs running in different instances. In Oracle8*i* release 8.1.5 an instance affinity (primary and secondary) can be specified for a queue table. When AQ is used with parallel server and multiple instances, this information is used to partition the queue-tables between instances for queue-monitor scheduling as well as for propagation. At any time, the queue table is affiliated to one instance. In the absence of an explicitly specified affinity, any available instance is made the owner of the queue table. If the owner of the queue table dies, the secondary instance or some available instance takes over the ownership for the queue table.

To see this feature applied in the context of the BooksOnLine scenario, refer to Support for Oracle Parallel Server (OPS) on page 2-24).

Support for Statistics Views

Basic statistics about queues in the database are available via the GV$AQ view.

Reliability and Recoverability

The standard database reliability and recoverability characteristics apply to queue data.

ENQUEUE Features

The following features apply to the process of producing messages by enqueuing them into a queue.

Correlation Identifier

Users can assign an identifier to each message, thus providing a means to retrieve specific messages at a later time.

Subscription & Recipient Lists

A single message can be designed to be consumed by multiple consumers. A queue administrator can specify the list of subscribers who can retrieve messages from a queue. Different queues can have different subscribers, and a consumer program can be a subscriber to more than one queue. Further, specific messages in a queue can be directed toward specific recipients who may or may not be subscribers to the queue, thereby overriding the subscriber list.

You can design a single message for consumption by multiple consumers in a number of different ways. The consumers who are allowed to retrieve the message are specified as explicit recipients of the message by the user or application that enqueues the message. Every explicit recipient is an agent identified by name, address and protocol.

A queue administrator may also specify a default list of recipients who can retrieve all the messages from a specific queue. These implicit recipients become subscribers to the queue by being specified in s default list. If a message is enqueued without specifying any explicit recipients, the message is delivered to all the designated subscribers.

A rule-based subscriber is one that has a rule associated with it in the default recipient list. A rule based subscriber will be sent a message with no explicit recipients specified only if the associated rule evaluated to TRUE for the message. Different queues can have different subscribers, and the same recipient can be a subscriber to more than one queue. Further, specific messages in a queue can be directed toward specific recipients who may or may not be subscribers to the queue, thereby over-riding the subscriber list.

A recipient may be specified only by its name, in which case the recipient must dequeue the message from the queue in which message was enqueued. It may be specified by its name and an address with a protocol value of 0. The address should be the name of another queue in the same database or another Oracle8*i* database (identified by the database link) in which case the message is propagated to the specified queue and can be dequeued by a consumer with the specified name. If the recipient's name is NULL, the message is propagated to the specified queue in the address and can be dequeued by the subscribers of the queue specified in the address. If the protocol field is nonzero, the name and address field is not interpreted by the system and the message can be dequeued by special consumer (see third party support in the propagation section).

To see this feature applied in the context of the BooksOnLine scenario, refer to Subscriptions and Recipient Lists on page 2-29).

Priority and Ordering of Messages in Enqueuing

It is possible to specify the priority of the enqueued message. An enqueued message can also have its exact position in the queue specified. This means that users have three options to specify the order in which messages are consumed: (a) a sort order specifies which properties are used to order all message in a queue; (b) a priority can be assigned to each message; (c) a sequence deviation allows you to position a message in relation to other messages. Further, if several consumers act on the same queue, a consumer will get the first message that is available for immediate consumption. A message that is in the process of being consumed by another consumer will be skipped.

To see this feature applied in the context of the BooksOnLine scenario, refer to Priority and Ordering of Messages on page 2-31).

Message Grouping

Messages belonging to one queue can be grouped to form a set that can only be consumed by one user at a time. This requires the queue be created in a queue table that is enabled for message grouping. All messages belonging to a group have to be created in the same transaction and all messages created in one transaction belong to the same group. This feature allows users to segment complex messages into simple messages, e.g., messages directed to a queue containing invoices could be constructed as a group of messages starting with the header message, followed by messages representing details, followed by the trailer message.

To see this feature applied in the context of the BooksOnLine scenario, refer to Message Grouping on page 2-37).

Propagation

This feature enables applications to communicate with each other without having to be connected to the same database or to the same Queue. Messages can be propagated from one Oracle AQ to another, irrespective of whether these are local or remote. The propagation is done using database links, and Net8.

To see this feature applied in the context of the BooksOnLine scenario, refer to Asynchronous Notifications on page 2-39).

Sender Identification

Applications can mark the messages they send with a custom identification. Oracle also automatically identifies the queue from which a message was dequeued. This allows applications to track the pathway of a propagated message, or of a string messages within the same database.

Time Specification and Scheduling

Delay interval and/or expiration intervals can be specified for an enqueued message, thereby providing windows of execution. A message can be marked as available for processing only after a specified time elapses (a delay time) and has to be consumed before a specified time limit expires.

Rule-based Subscribers

A message can be delivered to multiple recipients based on message properties or message content. Users define a rule based subscription for a given queue as the mechanism to specify interest in receiving messages of interest. Rules can be specified based on message properties and message data (for object and raw payloads). Subscriber rules are then used to evaluate recipients for message delivery.

To see this feature applied in the context of the BooksOnLine scenario, refer to Rule-based Subscription on page 2-69).

Asynchronous Notification

OCI clients can use the new call `OCISubscriptionRegister` to register a callback for message notification. The client issues a registration call which specifies a subscription name and a callback. When messages for the subscription are received, the callback is invoked. The callback may then issue an explicit dequeue to retrieve the message.

To see this feature applied in the context of the BooksOnLine scenario, refer to Asynchronous Notifications on page 2-39).

DEQUEUE Features

Multiple Recipients

A message in queue can be retrieved by multiple recipients without there being multiple copies of the same message.

To see this feature applied in the context of the BooksOnLine scenario, refer to Multiple Recipients on page 2-50).

Local and Remote Recipients

Designated recipients can be located locally and / or at remote sites.

To see this feature applied in the context of the BooksOnLine scenario, refer to Local and Remote Recipients on page 2-52).

Navigation of Messages in Dequeuing

Users have several options to select a message from a queue. They can select the first message or once they have selected a message and established a position, they can retrieve the next. The selection is influenced by the ordering or can be limited by specifying a correlation identifier. Users can also retrieve a specific message using the message identifier.

To see this feature applied in the context of the BooksOnLine scenario, refer to Message Navigation in Dequeue on page 2-54).

Modes of Dequeuing

A `DEQUEUE` request can either browse or remove a message. If a message is browsed it remains available for further processing, if a message is removed, it is not available any more for `DEQUEUE` requests. Depending on the queue properties a removed message may be retained in the queue table.

To see this feature applied in the context of the BooksOnLine scenario, refer to Modes of Dequeuing on page 2-57).

Optimization of Waiting for the Arrival of Messages

A `DEQUEUE` could be issued against an empty queue. To avoid polling for the arrival of a new message a user can specify if and for how long the request is allowed to wait for the arrival of a message.

To see this feature applied in the context of the BooksOnLine scenario, refer to Optimization of Waiting for Arrival of Messages on page 2-61).

Retries with Delays

A message has to be consumed exactly once. If an attempt to dequeue a message fails and the transaction is rolled back, the message will be made available for reprocessing after some user specified delay elapses. Reprocessing will be attempted up to the user-specified limit.

To see this feature applied in the context of the BooksOnLine scenario, refer to Retry with Delay Interval on page 2-63).

Optional Transaction Protection

`ENQUEUE`/`DEQUEUE` requests are normally part of a transaction that contains the requests, thereby providing the desired transactional behavior. Users can, however, specify that a specific request is a transaction by itself making the result of that request immediately visible to other transactions. This means that messages can be made visible to the external world either as soon as the `ENQUEUE` or `DEQUEUE` statement is issued, or only after the transaction is committed.

Exception Handling

A message may not be consumed within given constraints, i.e. within the window of execution or within the limits of the retries. If such a condition arises, the message will be moved to a user-specified exception queue.

To see this feature applied in the context of the BooksOnLine scenario, refer to Exception Handling on page 2-65).

Listen Capability (Wait on Multiple Queues)

The listen call is a blocking call that can be used to wait for messages on multiple queues. It can be used by a gateway application to monitor a set of queues. An application can also use it to wait for messages on a list of subscriptions. If the listen returns successfully, a dequeue must be used to retrieve the message.

To see this feature applied in the context of the BooksOnLine scenario, refer to Listen Capability on page 2-72).

Dequeue Message Header with No Payload

The new dequeue mode REMOVE_NODATA can be used to remove a message from a queue without retrieving the payload. This mode will be useful for applications that want to delete messages with huge payloads and aren't interested in the payload contents.

Propagation Features

Automated Coordination of Enqueuing and Dequeuing

As already noted, recipients can be local or remote. Oracle8*i* does not support distributed object types, hence remote enqueuing or dequeuing using a standard database link does not work. However, you can use AQ's message propagation to enqueue to a remote queue.

For example, you can connect to database X and enqueue the message in a queue, say "`DROPBOX`" located in database X. You can configure AQ so that all messages enqueued in queue "`DROPBOX`" will be automatically propagated to another queue in a database Y, regardless whether database Y is local or remote. AQ will automatically check if the type of the remote queue in database Y is structurally equivalent to the type of the local queue in database X, and propagate the message.

Recipients of propagated messages can be either applications or queues. If the recipient is a queue, the actual recipients will be determined by the subscription list associated with the recipient queue.If the queues are remote, messages will be propagated using the specified database link. Only AQ to AQ message propagation is supported.

Propagation of Messages with LOBs

Propagation handles payloads with `LOB` attributes.

To see this feature applied in the context of the BooksOnLine scenario, refer to Propagation of Messages with LOB Attributes on page 2-82).

Propagation Scheduling

Messages can be scheduled to propagate from a queue to local or remote destinations. Administrators can specify the start time, the propagation window and a function to determine the next propagation window (for periodic schedules).

Enhanced Propagation Scheduling Capabilities

Detailed run-time information about propagation is gathered and stored in the `DBA_QUEUE_SCHEDULES` view for each propagation schedule. This information can be used by queue designers and administrators to fix problems or tune performance. For example, available statistics about the total and average number of message/bytes propagated can be used to tune schedules. Similarly, errors reported by the view can be used to diagnose and fix problems. The view also describes additional information such as the session ID of the session handling the propagation, and the process name of the job queue process handling the propagation. To see this feature applied in the context of the BooksOnLine scenario, refer to Enhanced Propagation Scheduling Capabilities on page 2-85).

Third Party Support

Advanced Queueing allows messages to be enqueued in queues that can then be propagated to different messaging systems by third party propagators. If the protocol number for a recipient is in the range 128 - 255, the address of the recipient is not interpreted by AQ and so the message is not propagated by the Advanced Queuing system. Instead a third party propagator can then dequeue the message by specifying a reserved consumer name in the dequeue operation. The reserved consumer names are of the form `AQ$_P#` where # is the protocol number in the range 128 - 255. For example, the consumer name `AQ$_P128` can be used to dequeue messages for recipients with protocol number 128. The list of recipients for a message with the specific protocol number is returned in the `recipient_list` message property on dequeue.

Primary Components of Advanced Queuing (AQ)

By integrating transaction processing with queuing technology, persistent messaging in the form of **Advanced Queuing** is made possible.

Message

A message is the smallest unit of information inserted into and retrieved from a queue. A message consists of

- Control information (metadata), and
- Payload (data).

The control information represents message properties used by AQ to manage messages. The payload data is the information stored in the queue and is transparent to Oracle AQ. A message can reside in only one queue. A message is created by the enqueue call and consumed by the dequeue call.

Queue

A queue is a repository for messages. There are two types of queues: user queues, also known as normal queues, and exception queues. The user queue is for normal message processing. Messages are transferred to an exception queue if they can not be retrieved and processed for some reason. Queues can be created, altered, started, stopped, and dropped by using the Oracle AQ administrative interfaces (see Chapter 4, "Administrative Interface: Basic Operations").

Queue Table

Queues are stored in queue tables. Each queue table is a database table and contains one or more queues. Each queue table contains a default exception queue. Figure 1–1, "Basic Queues" on page 1-18 shows the relationship between messages, queues, and queue tables.

Agent

An agent is a queue user. This could be an end user or an application.There are two types of agents:

- Producers who place messages in a queue (enqueuing), and
- Consumers who retrieve messages (dequeuing).

Any number of producers and consumers may be accessing the queue at a given time. Agents insert messages into a queue and retrieve messages from the queue by using the Oracle AQ operational interfaces (see Chapter 6, "Operational Interface: Basic Operations")

An agent is identified by its `name`, `address` and `protocol` (see "Agent" on page 3-5 in Chapter 3, "Managing Oracle AQ" for formal description of this data structure).

- The name of the agent may be the name of the application or a name assigned by the application. As will be described below, a queue may itself be an agent — enqueuing or dequeuing from another queue.
- The address field is a character field of up to 1024 bytes that is interpreted in the context of the protocol. For instance, the default value for the protocol is 0, signifying a database link addressing. In this case, the address for this protocol is of the form

  `queue_name@dblink`

  where `queue_name` is of the form `[schema.]queue` and `dblink` may either be a fully qualified database link name or the database link name without the domain name.

Recipient

The recipient of a message may be specified by its name only, in which case the recipient must dequeue the message from the queue in which the message was enqueued. The recipient may be specified by name and an address with a protocol value of 0. The address should be the name of another queue in the same database or another Oracle8 database (identified by the database link) in which case the message is propagated to the specified queue and can be dequeued by a consumer with the specified name. If the recipient's name is `NULL`, the message is propagated to the specified queue in the address and can be dequeued by the subscribers of the queue specified in the address. If the protocol field is nonzero, the name and address field is not interpreted by the system and the message can be dequeued by special consumer (see third party support in the propagation section).

Recipient and Subscription Lists

A single message can be designed for consumption by multiple consumers. There are two ways to do this.

- The enqueuer can explicitly specify the consumers who may retrieve the message as recipients of the message. A recipient is an agent identified by a name, address and protocol.
- A queue administrator can specify a default list of recipients who can retrieve messages from a queue. The recipients specified in the default list are known as subscribers. If a message is enqueued without specifying the recipients the message is implicitly sent to all the subscribers.

Different queues can have different subscribers, and the same recipient can be a subscriber to more than one queue. Further, specific messages in a queue can be directed toward specific recipients who may or may not be subscribers to the queue, thereby over-riding the subscriber list.

Rule

A rule is used to define one or more subscribers' interest in subscribing to messages that conform to that rule. The messages that meet this criterion are then delivered to the interested subscribers. Put another way: a rule filters for messages in a queue on a topic in which a subscriber is interested.

A rule is specified as a boolean expression (one that evaluates to true or false) using syntax similar to the `WHERE` clause of a SQL query. This boolean expression can include conditions on

- message properties (currently `priority` and `corrid`),
- user data properties (object payloads only), and
- functions (as specified in the where clause of a SQL query).

Rule Based Subscriber

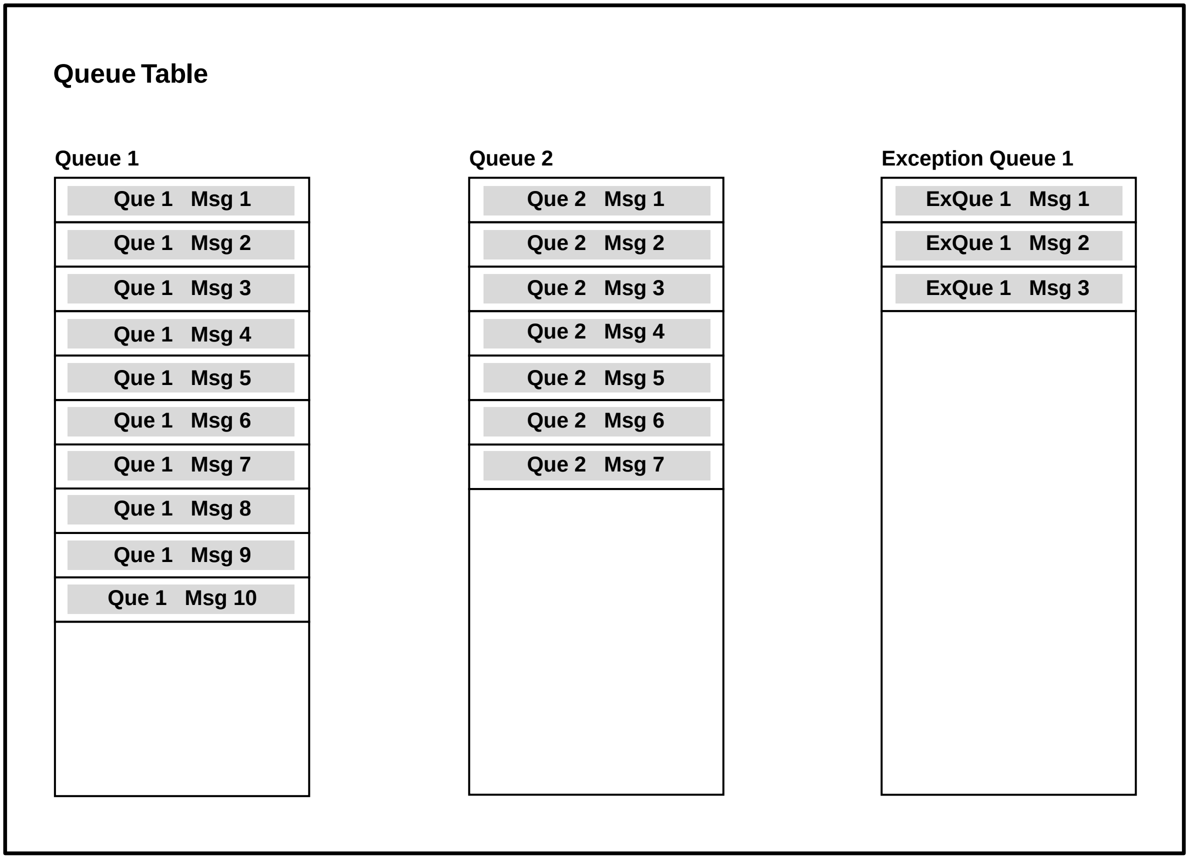
A rule-based subscriber is a subscriber that has rule associated with it in the default recipient list. A rule-based subscriber is sent a message that has no explicit recipients specified if the associated rule evaluates to `TRUE` for the message.

Queue Monitor

The queue monitor (QMNn) is a background process that monitors the messages in the queues. It provides the mechanism for message delay, expiration and retry delay. The also QMNn also performs garbage collection for the queue table and its indexes and index-organized tables. It is possible to start a maximum of 10 multiple queue monitors at the same time. You start the desired number of queue monitors by setting the dynamic init.ora parameter `aq_tm_processes`. The queue monitor wakes up every minute, or whenever there is work to be done, for instance, if a message is to be marked as expired or as ready to be processed.

Modeling Queue Entities

Figure 1–1 Basic Queues



The preceding figure portrays a queue table that contains two queues, and one exception queue:

- Queue1 — contains 10 messages.
- Queue2 — contains 7 messages.
- ExceptionQueue1 — contains 3 messages.

Basic Queuing

Basic Queuing — One Producer, One Consumer

At its most basic, one producer may enqueue different messages into one queue. Each message will be dequeued and processed once by one of the consumers. A message will stay in the queue until a consumer dequeues it or the message expires. A producer may stipulate a delay before the message is available to be consumed, and a time after which the message expires. Likewise, a consumer may wait when trying to dequeue a message if no message is available. Note that an agent program, or application, can act as both a producer and a consumer.

Basic Queuing — Many Producers, One Consumer

At a slightly higher level of complexity, many producers may enqueue messages into a queue, all of which are processed by one consumer.

Basic Queuing — Many Producers, Many Consumers of Discrete Messages

In this next stage, many producers may enqueue messages, each message being processed by a different consumer depending on type and correlation identifier. The figure below shows this scenario.

Illustrating Basic Queuing

Figure Figure 1–2, "Modeling Basic Queuing" (below) portrays a queue table that contains one queue into which messages are being enqueued and from which messages are being dequeued.

Producers

The figure indicates that there are 6 producers of messages, although only four are shown. This assumes that two other producers (P4 and P5) have the right to enqueue messages even though there are no messages enqueued by them at the moment portrayed by the figure. The figure shows:

- that a single producer may enqueue one or more messages.

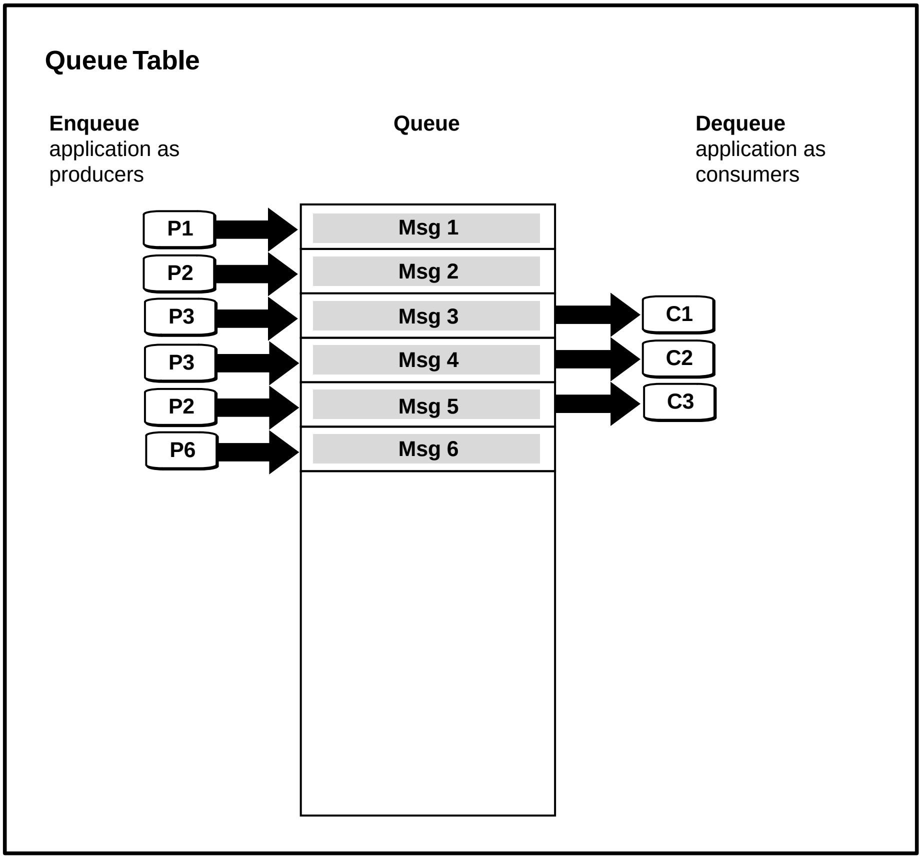
- that producers may enqueue messages in any sequence.

Consumers

According to the figure, there are 3 consumers of messages, representing the total population of consumers. The figure shows:

- messages are not necessarily dequeued in the order in which they are enqueued.
- messages may be enqueued without being dequeued.

Figure 1–2 Modeling Basic Queuing



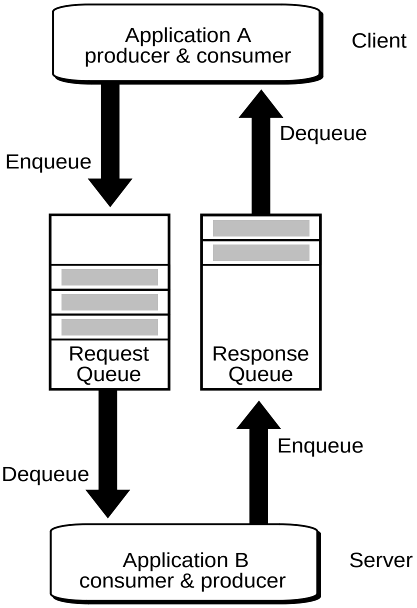
Illustrating Client-Server Communication Using AQ

The previous figure portrayed the enqueuing of multiple messages by a set of producers, and the dequeuing of messages by a set of consumers. What may not be readily evident in that sketch is the notion of *time*, and the advantages offered by Oracle AQ.

Client-Server applications normally execute in a synchronous manner, with all the disadvantages of that tight coupling described above. Figure 1–3, "Client-Server Communication Using AQ" demonstrates the asynchronous alternative using AQ.

In this example *Application B* (a server) provides service to *Application A* (a client) using a request/response queue.

Figure 1–3 Client-Server Communication Using AQ



1. *Application A* enqueues a request into the request queue.
2. *Application B* dequeues the request.
3. *Application B* processes the request.
4. *Application B* enqueues the result in the response queue.
5. *Application A* dequeues the result from the response queue.

In this way the client does not have to wait to establish a connection with the server, and the server dequeues the message at its own pace. When the server is finished processing the message, there is no need for the client to be waiting to receive the result. In this way a process of double-deferral frees both client and server.

Note: The various enqueue and dequeue operations are part of different transactions.

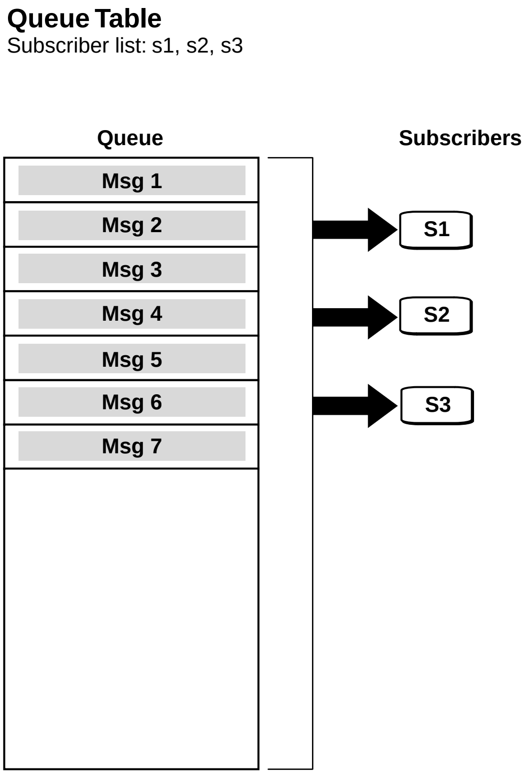
Multiple-Consumer Dequeuing of the Same Message

A message can only be enqueued into one queue at a time. If a producer had to insert the same message into several queues in order to reach different consumers, this would require management of a very large number of queues. Oracle AQ provides two mechanisms to allow for multiple consumers to dequeue the same message: *queue subscribers* and *message recipients*. The queue must reside in a queue table that is created with multiple consumer option to allow for subscriber and recipient lists. Each message remains in the queue until it is consumed by all its intended consumers.

Queue Subscribers Using this approach, multiple consumer-subscribers are associated with a queue. This will cause all messages enqueued in the queue to be made available to be consumed by each of the queue subscribers. The subscribers to the queue can be changed dynamically without any change to the messages or message producers. Subscribers to the queue are added and removed by using the Oracle AQ administrative package. The diagram below shows multiple producers enqueuing messages into queue, each of which is consumed by multiple consumer-subscribers.

Message Recipients A message producer can submit a list of recipients at the time a message is enqueued. This allows for a unique set of recipients for each message in the queue. The recipient list associated with the message overrides the subscriber list associated with the queue, if there is one. The recipients need not be in the subscriber list. However, recipients may be selected from among the subscribers.

Figure 1–4 Multiple-Consumer Dequeuing of the Same Message



Illustrating Multiple-Consumer Dequeuing of the Same Message

Figure 1–4 describes the case in which three consumers are all listed as subscribers of a queue. This is the same as saying that they all subscribe to all the messages that might ever be enqueued into that queue. The drawing illustrates a number of important points:

- The figure portrays the situation in which the 3 consumers are subscribers to 7 messages that have already been enqueued, and that they might become subscribers to messages that have not yet been enqueued.

- Every message will eventually be dequeued by every subscriber.
- There is no priority among subscribers. This means that there is no way of saying which subscriber will dequeue which message first, second, and so on. Or, put more formally: the order of dequeuing by subscribers is undetermined.
- We have no way of knowing from the figure about messages they might already have been dequeued, and which were then removed from the queue.

Figure 1–5 Communication Using a Multi-Consumer Queue

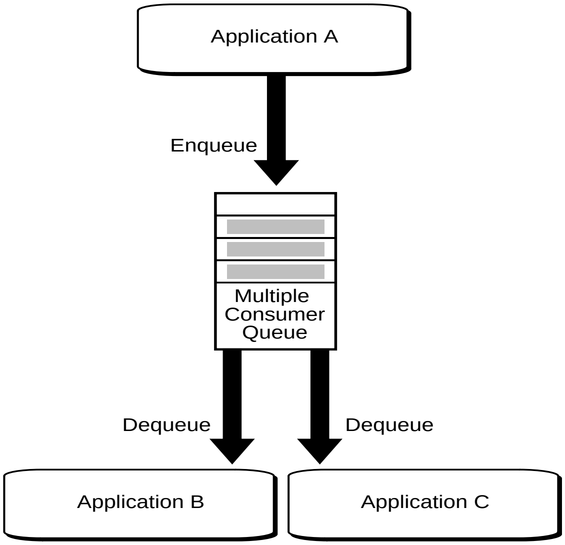
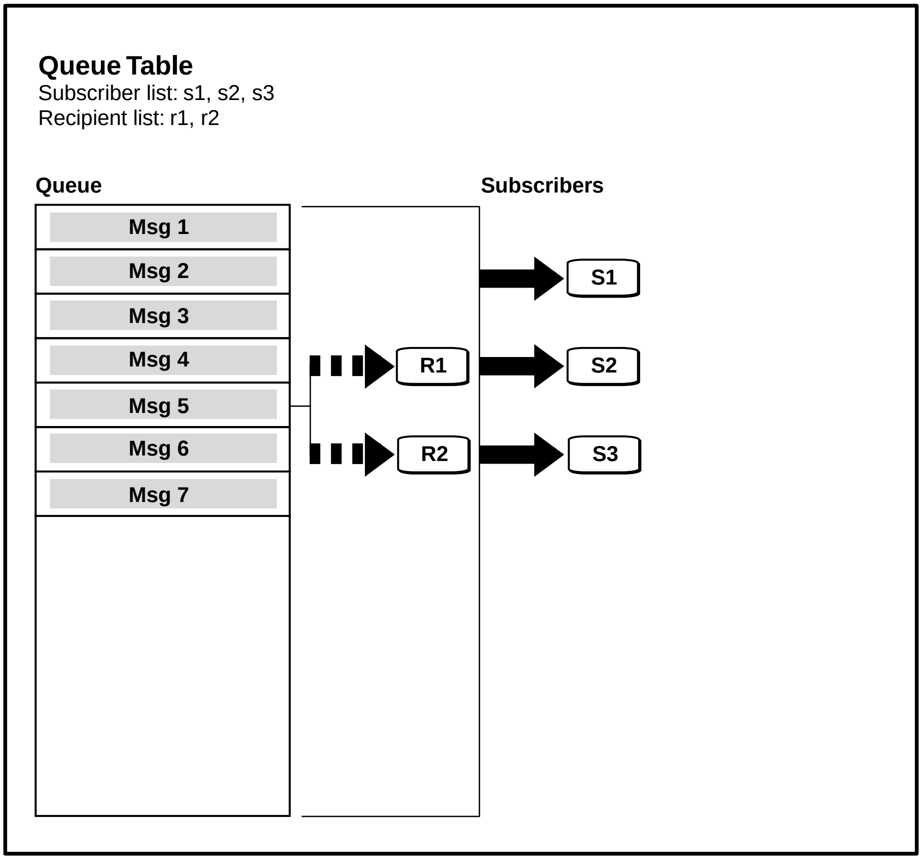


Figure 1–5 illustrates the same technology from a dynamic perspective. This examples concerns a scenario in which more than one application needs the result produced by an application. Every message enqueued by *Application A* is dequeued by *Application B* and *Application C*. To make this possible, the multiple consumer queue is specially configured with *Application B* and *Application C* as queue subscribers. Consequently, they are implicit recipients of every message placed in the queue.

Note: Queue subscribers can be applications or other queues.

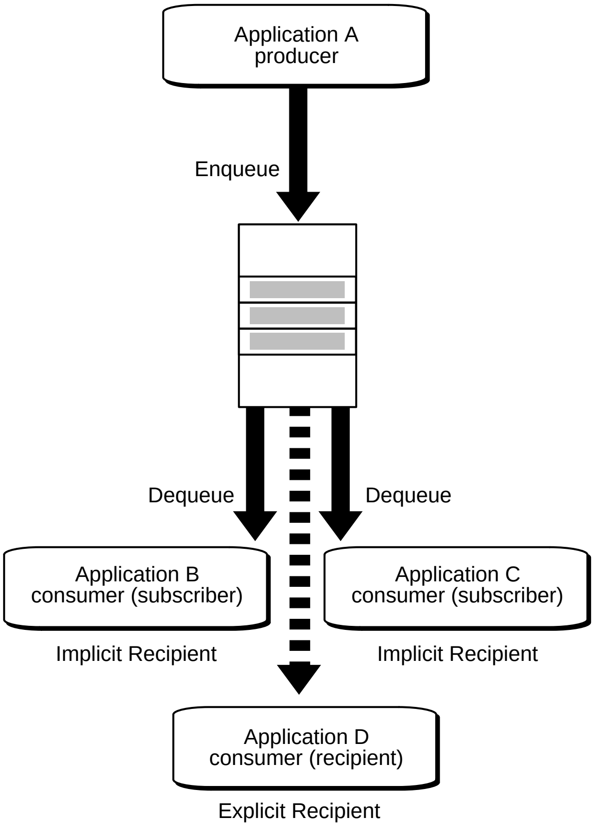
Figure 1–6 Dequeuing of Specified Messages by Specified Recipients



Illustrating Dequeuing of Specified Messages by Specified Recipients

Figure 1–6 shows how a message can be specified for one or more recipients. In this case, *Message 5* is specified to be dequeued by *Recipient-1* and *Recipient-2*. As described by the drawing, neither of the recipients is one of the 3 subscribers to the queue.

Figure 1–7 Explicit and Implicit Recipients of Messages



We earlier referred to *subscribers* as "implicit recipients" in that they are able to dequeue all the messages placed into a specific queue. This is like subscribing to a magazine and thereby implicitly gaining access to all its articles. The category of consumers that we have referred to as *recipients* may also be viewed as "explicit recipients" in that they are designated targets of particular messages.

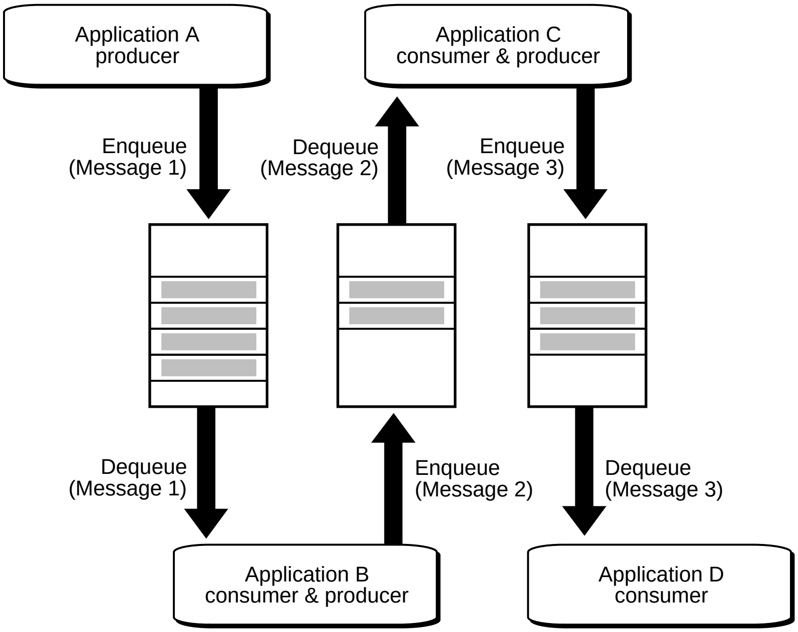
Figure 1–7 shows how Oracle AQ can adjust dynamically to accommodate both kinds of consumers. In this scenario *Application B* and *Application C* are implicit recipients (subscribers). But messages can also be explicitly directed toward specific consumers (recipients) who may or may not be subscribers to the queue. The list of such recipients is specified in the enqueue call for that message and overrides the list of subscribers for that queue. In the figure, *Application D* is specified as the sole recipient of a message enqueued by *Application A*.

Note: Multiple producers may simultaneously enqueue messages aimed at different targeted recipients.

Illustrating the Implementation of Workflows using AQ

Figure 1–8 illustrates the use of AQ for implementing workflows, also knows as chained application transactions. It shows a workflow consisting of 4 steps performed by *Applications A*, *B*, *C* and *D*. The queues are used to buffer the flow of information between different processing stages of the business process. By specifying delay interval and expiration time for a message, a window of execution can be provided for each of the applications.

Figure 1–8 Implementing Workflows using AQ



From a workflow perspective, the passing of messages is a business asset above and beyond the value of the payload data. Hence, AQ supports the optional retention of messages for analysis of historical patterns and prediction of future trends. For instance, two of the three application scenarios at the head of the chapter are founded in an implementation of workflow analysis.

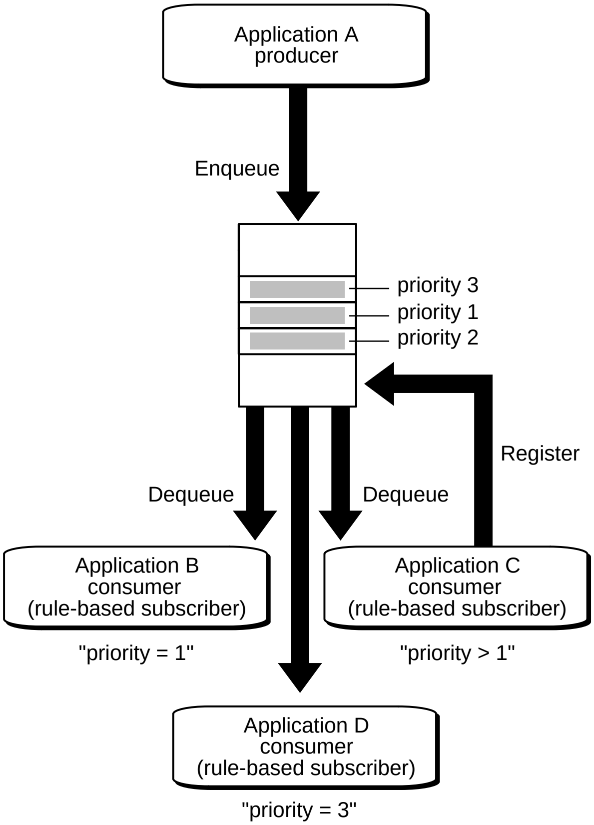
Note: The contents of the messages 1, 2 and 3 can be the same or different. Even when they are different, messages may contain parts of the of the contents of previous messages.

Illustrating the Implementation of Publish/Subscribe using AQ

Figure 1–9 illustrates the use of AQ for implementing a publish / subscribe messaging scheme between applications. Application A is a publisher application which is publishing messages to a queue. Applications B, C, D are subscriber applications. Application A publishes messages anonymously to a queue. These messages are then delivered to subscriber applications based on the rules specified by each application. Subscriber applications can specify interest in messages by defining a rule on message properties and message data content.

In the example shown, applications B has subscribed with rule "priority=1", application C has subscribed with rule "priority > 1" and application D has subscribed with rule "priority = 3". Application A enqueues 3 messages (priority 3, 1, 2). Application B receives a single message (priority 1), application C receives two messages (priority 2, 3) and application D receives a single message (priority 3). Thus, message recipients are computed dynamically based on message properties and content. Additionally, the figure also illustrates how application D uses asynchronous notification for message delivery. Application D registers for messages on the queue. When messages arrive, application D is notified and can then dequeue the messages.

Figure 1–9 Implementing Publish/Subscribe using AQ



From a workflow perspective, the passing of messages is a business asset above and beyond the value of the payload data. Hence, AQ supports the optional retention of messages for analysis of historical patterns and prediction of future trends. For instance, two of the three application scenarios at the head of the chapter are founded in an implementation of workflow analysis.

Message Propagation

Fanning-Out of Messages

In AQ, message recipients can be either consumers or other queues. If the message recipient is a queue, the actual recipients are determined by the subscribers to the queue (which may in turn be other queues). Thus it is possible to fan-out messages to a large number of recipients without requiring them all to dequeue messages from a single queue.

For example: A queue, *Source,* may have as its subscribers queues *dispatch1@dest1* and *dispatch2@dest2.* Queue *dispatch1@dest1* may in turn have as its subscribers the queues *outerreach1@dest3* and *outerreach2@dest4,* while queue *dispatch2@dest2* has as subscribers the queue *outerreach3@dest21* and *outerreach4@dest4*. In this way, messages enqueued in *Source* will be propagated to all the subscribers of four different queues.

Funneling-in of Messages

Another use of queues as a message recipient is the ability to combine messages from different queues into a single queue. This process is sometimes described as "compositing"

For example, if queue *composite@endpoint* is a subscriber to both queues *funnel1@source1* and *funnel2@source2* then the subscribers to queue *composite@endpoint* can get all messages enqueued in those queues as well as messages enqueued directly into itself.

Figure 1–10   Message Propagation

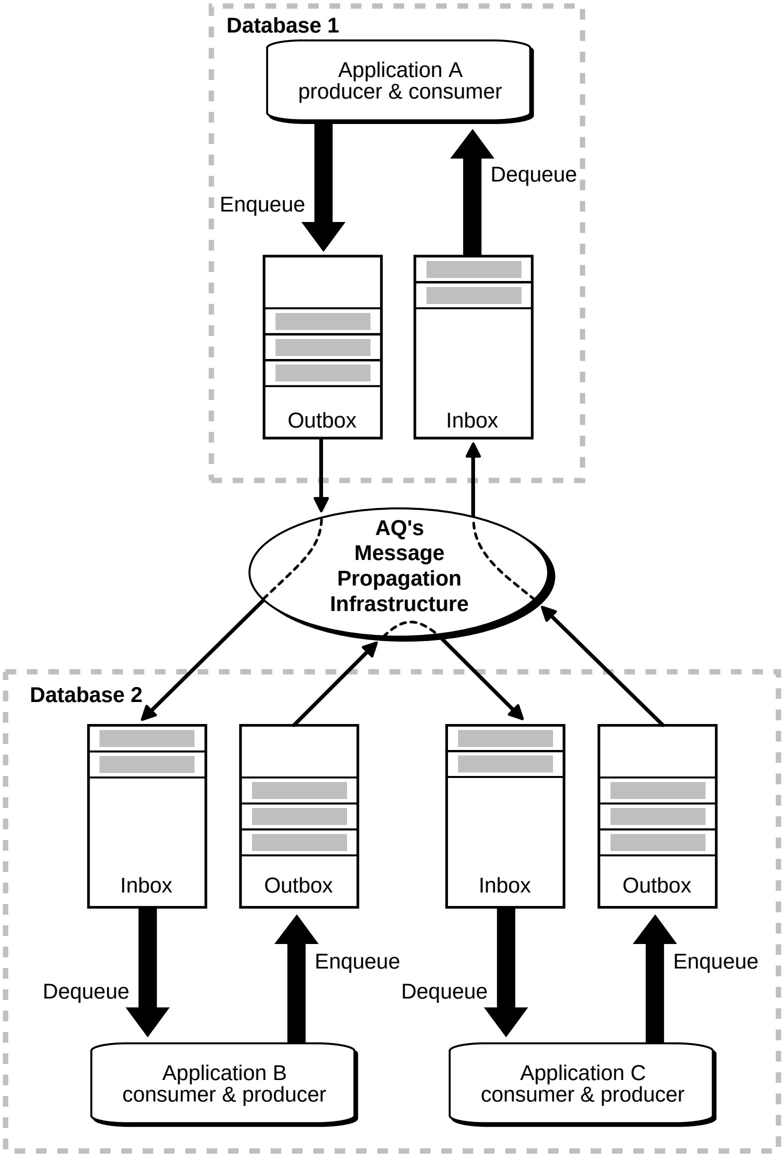


Illustration of Message Propagation

Figure 1–10 illustrates applications on different databases communicating via AQ. Each application has an inbox and an outbox for handling incoming and outgoing messages. An application enqueues a message into its outbox irrespective of whether the message has to be sent to an application that is local (on the same node) or remote (on a different node).

Likewise, an application is not concerned as to whether a message originates locally or remotely. In all cases, an application dequeues messages from its inbox.

Oracle AQ facilitates all this interchange, treating messages on the same basis.

Programmatic Environments for Working with AQ

Oracle now offers you different environments for working with AQ:

- The PL/SQL language by means of the **DBMS_AQADM** and the **DBMS_AQ packages** as described in the *Oracle8i Supplied Packages Reference*
- The C++ language by means of the **Oracle Call Interface (OCI)** described in the *Oracle Call Interface Programmer's Guide*
- The Visual Basic language by means of **Oracle Objects For OLE (OO4O)** as described in its accompanying online help.
- The Java language by means of the **Java Application Programmer's Interface** as described in the Chapter 7, "Advanced Queuing — Java API".

AQ and XA

You must specify "Objects=T" in the `xa_open` string if you want to use the AQ OCI interface. This forces XA to initialize the client side cache in Objects mode. You do not need to do this if you plan to use AQ through PL/SQL wrappers from OCI or Pro*C.

You must use AQ navigation option carefully when you are using AQ from XA. XA cancels cursor fetch state after an `xa_end`. Hence, if you want to continue dequeuing between services (i.e. `xa_start`/`xa_end` boundaries) you must reset the dequeue position by using the `FIRST_MESSAGE` navigation option. Otherwise you will get an `ORA-25237` (navigation used out of sequence).

For more information about deploying AQ with XA, see:

- "Using XA with AQ" on page 3-23 in Chapter 3, "Managing Oracle AQ"
- "Deploy AQ with XA" on page 8-54 in Chapter 8, "Oracle Advanced Queuing by Example"

Compatibility

Certain features only will function if compatibility is set to ’8.1’. As shown in Table 1–1, you may have to set the `compatible` parameter of the `init.ora` and / or the `compatible` parameter of the queue table.

Table 1–1 Compatibility Settings Required to Make Use of New Features

Feature	Init.ora compatible = ’ 8.1.x’	queue table compatible = ’ 8.1’
Queue Level Access Control	X	X
Non-Persistent Queues	X	automatically created
Support for OPS Environments	X	
Rule-based Subscribers for Publish / Subscribe	X	X
Asynchronous Notification	X	
Sender Identification	X	X
Separate storage of history management information	X	X

For more information, see:

- Appendix A, "Migrating Queue Tables"
- *Oracle8i Migration*

Restrictions

The following restrictions currently apply.

Auto-commit features in DBMS_AQADM package

The auto_commit parameters in `CREATE_QUEUE_TABLE`, `DROP_QUEUE_TABLE`, `CREATE_QUEUE`, `DROP_QUEUE` and `ALTER_QUEUE` calls in `DBMS_AQADM` package are deprecated for 8.1.5 and subsequent releases. Oracle continues to support this parameter in the interface for backward compatibility purpose.

Collection Types in Message Payloads

You cannot construct a message payload using a collection type that is not itself contained within an object. You also cannot currently use a nested table even as an embedded object within a message payload. However, you can create an object type that contains one or more VARRAYs, and create a queue table that is founded on this object type.

For example, the following operations are allowed:

```
CREATE TYPE number_varray AS VARRAY(32) OF NUMBER;
CREATE TYPE embedded_varray AS OBJECT (col1 number_varray);
EXECUTE DBMS_AQADM.CREATE_QUEUE_TABLE(
  queue_table          =>     'QT',
  queue_payload_type   =>     'embedded_varray');
```

Object Type Payload Support in AQ Java API

The AQ Java classes in release 8.1.5 does not allow enqueuing and dequeuing object type payloads, only raw type payloads are supported.

Synonyms on Queue Tables and Queue

All AQ PL/SQL calls do not resolve synonyms on queues and queue tables. Even though you can create a synonyms, you should not apply the synonym to the AQ interface.

Pluggable Tablespace does not Work For 8.0 Compatible Multiconsumer Queues

Any tablespace which contains 8.0 compatible multiconsumer queue tables should not be transported using the pluggable tablespace mechanism. The mechanism will work, however, with tablespaces that contain only single consumer queues as well as 8.1 compatible multiconsumer queues. Before you can export a tablespace in pluggable mode, you have to alter the tablespace to read-only mode. If you try to import a read-only tablespace which contain 8.0 compatible multiconsumer queues, you will get an Oracle error indicating that you cannot update the queue table index at import time.

Tablespace point-in-time recovery

AQ currently does not support tablespace point in time recovery. Creating a queue table in a tablespace will disable that particular tablespace for point-in-time recovery.

Propagation from Object Queues

Note that AQ does not support propagation from Object queues that have BFILE or REF attributes in the payload.

Non-Persistent Queues

Currently you can create only non-persistent queues of `RAW` type.You are limited in that you can send messages only to subscribers and explicitly specified recipients who are local. Propagation is not supported from non-persistent queues. And in retrieving messages, you cannot use the dequeue call but must instead employ the asynchronous notification mechanism, registering for the notification by mean of `OCISubcriptionRegister`.

Reference to Demos

The following demos may be found in the $ORACLE_HOME/demo directory.:

Table 1–2

Demo & Locations	Topic
`aqdemo00.sql`	Main driver of demo
`aqdemo01.sql`	Create queue tables and queues using AQ administration interface
`aqdemo02.sql`	Load the demo package
`aqdemo03.sql`	Submit the event handler as a job to Job `Queue`
`aqdemo04.sql`	Enqueue messages
`newaqdemo00.sql`	Create users, message types, tables etc.
`newaqdemo01.sql`	Set up queue_tables, queues, subscribers and set up
`newaqdemo02.sql`	Enqueue messages
`newaqdemo03.sql`	Installs dequeue procedures
`newaqdemo04.sql`	Performs ’blocking dequeue’
`newaqdemo05.sql`	Performs ’listen’ for multiple agents
`newaqdemo06.sql`	Cleans up users, queue_tables, queues, subscribers etc. (cleanup script)
`ociaqdemo00.c`	Enqueue messages
`ociaqdemo01.c`	Performs blocking dequeue
`ociaqdemo02.c`	Performs ’Listen’ for multiple agents

2

Implementing AQ — A Sample Application

In Chapter 1 we described a messaging system for an imaginary company, `BooksOnLine`. In this chapter we consider the features of AQ in the context of a sample application based on that scenario.

A Sample Application

The operations of a large bookseller, `BooksOnLine`, are based on an online book ordering system which automates activities across the various departments involved in the entire sale process.The front end of the system is an order entry application which is used to enter new orders.These incoming orders are processed by an order processing application which validates and records the order. Shipping departments located at regional warehouses are then responsible for ensuring that these orders are shipped in a timely fashion. There are three regional warehouses: one serving the East Region, one serving the West Region, and a third warehouse for shipping International orders. Once an order has been shipped, the order information is routed to a central billing department which handles payment processing.The customer service department, located at its own site, is responsible for maintaining order status and handling inquiries about orders.

In Chapter 1 we outlined a messaging system for an imaginary company, `BooksOnLine`. In this chapter we consider the features of AQ in the context of a sample application based on that scenario. This sample application has been devised for the sole purpose of demonstrating the features of Oracle AQ. Our aim in creating this integrated scenario is to make it easier to grasp the possibilities of this technology by locating our explanations within a single context. We have also provided the complete script for the code as an appendix (see Appendix A, "Scripts for Implementing ’BooksOnLine’"). However, please keep in mind that is not possible within the scope of a single relatively small code sample to demonstrate every possible application of AQ.

General Features

- System Level Access Control
- Structured Payload
- Queue Level Access Control
- Non-Persistent Queues
- Retention and Message History
- Publish/Subscribe Support
- Support for Oracle Parallel Server (OPS)
- Support for Statistics Views

System Level Access Control

Oracle 8i supports system level access control for all queueing operations. This feature allows application designer or DBA to create users as queue administrators. A queue administrator can invoke all AQ interface (both administration and operation) on any queue in the database. This simplify the administrative work as all administrative scripts for the queues in a database can be managed under one schema for more information, see "Security" on page 3-9 in Chapter 3, "Managing Oracle AQ").

Example Scenario and Code

In the `BooksOnLine` application, the DBA creates `BOLADM`, the `BooksOnLine` Administrator account, as the queue administrator of the database. This allows `BOLADM` to create, drop, manage, and monitor any queues in the database. If you decide to create PL/SQL packages in the `BOLADM` schema that can be used by any applications to enqueue or dequeue, then you should also grant `BOLADM` the `ENQUEUE_ANY` and `DEQUEUE_ANY` system privilege.

```
CREATE USER BOLADM IDENTIFIED BY BOLADM;
GRANT CONNECT, RESOURCE, aq_administrator_role TO BOLADM;
GRANT EXECUTE ON dbms_aq TO BOLADM;
GRANT EXECUTE ON dbms_aqadm TO BOLADM;
EXECUTE dbms_aqadm.grant_system_privilege('ENQUEUE_ANY','BOLADM',FALSE);
EXECUTE dbms_aqadm.grant_system_privilege('DEQUEUE_ANY','BOLADM',FALSE);
```

In the application, AQ propagators populate messages from the `OE` (Order Entry) schema to `WS` (Western Sales), `ES` (Eastern Sales) and `OS` (Worldwide Sales) schemas. `WS`, `ES` and `OS` schemas in turn populates messages to `CB` (Customer Billing) and `CS` (Customer Service) schemas. Hence the `OE`, `WS`, `ES` and `OS` schemas all host queues that serve as the source queues for the propagators.

When messages arrive at the destination queues, sessions based on the source queue schema name are used for enqueuing the newly arrived messages into the destination queues. This means that you need to grant schemas of the source queues enqueue privileges to the destination queues.

To simplify administration, all schemas that host a source queue in the `BooksOnLine` application are granted the `ENQUEUE_ANY` system privilege.

```
EXECUTE dbms_aqadm.grant_system_privilege('ENQUEUE_ANY','OE',FALSE);
EXECUTE dbms_aqadm.grant_system_privilege('ENQUEUE_ANY','WS',FALSE);
EXECUTE dbms_aqadm.grant_system_privilege('ENQUEUE_ANY','ES',FALSE);
EXECUTE dbms_aqadm.grant_system_privilege('ENQUEUE_ANY','OS',FALSE);
```

To propagate to a remote destination queue, the login user specified in the database link in the address field of the agent structure should either be granted the '`ENQUEUE ANY QUEUE`' privilege, or be granted the rights to enqueue to the destination queue. However, you do not need to grant any explicit privileges if the login user in the database link also owns the queue tables at the destination.

Structured Payload

Oracle AQ lets you use object types to structure and manage the payload of messages. Object Relational Database Systems (ORDBMSs) generally have a richer type system than messaging systems. The object-relational capabilities of Oracle 8*i* provide a rich set of data types that range from traditional relational data types to user-defined types (see "Enqueuing and Dequeuing Object Type Messages That Contain LOB Attributes Using PL/SQL" on page 8-45 inChapter 8, "Oracle Advanced Queuing by Example").

Many powerful features are enabled as a result of having strongly typed content i.e., content whose format is defined by an external type system. These features include;

- Content-based routing: an external agent can examine the content and route messages to another queue based on content.
- Content-based subscription: a publish and subscribe system can be built on top of a messaging system offers content-based subscription
- Querying: the ability to execute queries on the content of messages allows users to examine current and processed messages for various applications including message warehousing.

Example Scenario and Code

The `BooksOnLine` application uses a rich set of data types to model book orders as message content.

- Customers are modeled as a object type called `customer_typ`.

```
CREATE OR REPLACE TYPE customer_typ AS OBJECT (
        custno          NUMBER,
        name            VARCHAR2(100),
        street          VARCHAR2(100),
        city            VARCHAR2(30),
        state           VARCHAR2(2),
        zip             NUMBER,
        country         VARCHAR2(100));
```

- Books are modeled as an object type called `book_typ`.

```
CREATE OR REPLACE TYPE book_typ AS OBJECT (
        title           VARCHAR2(100),
        authors         VARCHAR2(100),
        ISBN            NUMBER,
        price           NUMBER);
```

- An order item which represents an order line item is modeled as an object type called `orderitem_typ`. An order item is a nested type which includes the book type.

  ```
  CREATE OR REPLACE TYPE orderitem_typ AS OBJECT (
          quantity        NUMBER,
          item            BOOK_TYP,
          subtotal        NUMBER);
  ```

- An order item list is used to represent a list of order line items and is modeled as a varray of order items;

  ```
   create or replace type orderitemlist_vartyp AS VARRAY (20) OF orderitem_
  typ;
  ```

- An order is modeled as a object type called order_typ. The order type is a composite type which includes nested object types defined above. The order type captures details of the order, the customer information, and the item list.

  ```
  create or replace type order_typ as object (
          orderno         NUMBER,
          status          VARCHAR2(30),
          ordertype       VARCHAR2(30),
          orderregion     VARCHAR2(30),
          customer        CUSTOMER_TYP,
          paymentmethod   VARCHAR2(30),
          items           ORDERITEMLIST_VARTYP,
          total           NUMBER);
  ```

Queue Level Access Control

Oracle 8*i* supports queue level access control for enqueue and dequeue operations. This feature allows the application designer to protect queues created in one schema from applications running in other schemas. You need to grant only minimal access privileges to the applications that run outside the queue's schema. The supported access privileges on a queue are `ENQUEUE`, `DEQUEUE` and `ALL` for more information, see "Security" on page 3-9 in Chapter 3, "Managing Oracle AQ").

Example Scenario

The `BooksOnLine` application processes customer billings in its `CB` and `CBADM` schemas. `CB` (Customer Billing) schema hosts the customer billing application, and the `CBADM` schema hosts all related billing data stored as queue tables.

To protect the billing data, the billing application and the billing data reside in different schemas. The billing application is allowed only to dequeue messages from `CBADM_shippedorders_que`, the shipped order queue. It processes the messages, and them enqueues new messages into `CBADM_billedorders_que`, the billed order queue.

To protect the queues from other illegal operations from the application, the following two grant calls are made:

Example Code

```
/* Grant dequeue privilege on the shopped orders queue to the Customer
   Billing application. The CB application retrieves orders that are shipped but
   not billed from the shipped orders queue. */
EXECUTE dbms_aqadm.grant_queue_privilege(
   'DEQUEUE','CBADM_shippedorders_que', 'CB', FALSE);

/* Grant enqueue privilege on the billed orders queue to Customer Billing
   application.The CB application is allowed to put billed orders into this
   queue after processing the orders. */

EXECUTE dbms_aqadm.grant_queue_privilege(
   'ENQUEUE', 'CBADM_billedorders_que', 'CB', FALSE);
```

Non-Persistent Queues

Messages in a non-persistent queues are not persistent in that hey are not stored in database tables.

You create a non-persistent RAW queue which can be of either single-consumer or multi-consumer type. These queues are created in a system created queue-table (`AQ$_MEM_SC` for single-consumer queues and `AQ$_MEM_MC` for multi-consumer queues) in the schema specified by the `create_np_queue` command. Subscribers can be added to the multi-consumer queues (see "Create a Non-Persistent Queue" on page 4-24 in Chapter 2, "Implementing AQ — A Sample Application"). Non-persistent queues can be destinations for propagation.

You use the enqueue interface to enqueue messages into a non-persistent queue in the normal way. You retrieve messages from a non-persistent queue through the asynchronous notification mechanism, registering for the notification (using `OCISubcriptionRegister`) for those queues in which you are interested (see "Register for Notification" on page 6-50 in Chapter 6, "Operational Interface: Basic Operations").

When a message is enqueued into a queue, it is delivered to the clients that have active registrations for the queue. The messages are then published to the interested clients without incurring the overhead of storing them in the database.

For more information see:

- OCI documentation on OCISubscriptionRegister in *Oracle Call Interface Programmer's Guide.*

Example Scenario

Assume that there are three application processes servicing user requests at the `ORDER ENTRY` system. The connection dispatcher process, which shares out the connection requests among the application processes, would like to maintain a count of the number of users logged on to the Order Entry system as well as the number of users per application process. The application process are named `APP_1`, `APP_2`, `APP_3`. To simplify things we shall not worry about application process failures.

One way to solve this requirement is to use non-persistent queues. When a user logs-on to the database, the application process enqueues to the multi-consumer non-persistent queue, `LOGIN_LOGOUT`, with the application name as the consumer name. The same process occurs when a user logs out. To distinguish between the two events, the correlation of the message is '`LOGIN`' for logins and '`LOGOUT`' for logouts.

The callback function counts the login / logout events per application process. Note that the dispatcher process only needs to connect to the database for registering the subscriptions. The notifications themselves can be received while the process is disconnected from the database.

Example Code

```
CONNECT oe/oe;

/* Create the multiconsumer nonpersistent queue in OE schema: */
EXECUTE dbms_aqadm.create_np_queue(queue_name         => 'LOGON_LOGOFF',
                                  multiple_consumers => TRUE);

/* Enable the queue for enqueue and dequeue: */
EXECUTE dbms_aqadm.start_queue(queue_name => 'LOGON_LOGOFF');

/* Non Persistent Queue Scenario - procedure to be executed upon logon: */
CREATE OR REPLACE PROCEDURE  User_Logon(app_process IN VARCHAR2)
AS
  msgprop        dbms_aq.message_properties_t;
  enqopt         dbms_aq.enqueue_options_t;
  enq_msgid      RAW(16);
  payload        RAW(1);
BEGIN
  /* visibility must always be immediate for NonPersistent queues */
  enqopt.visibility:=dbms_aq.IMMEDIATE;
  msgprop.correlation:= 'LOGON';
  msgprop.recipient_list(0) := aq$_agent(app_process, NULL, NULL);
  /* payload is NULL */
  dbms_aq.enqueue(
        queue_name         => 'LOGON_LOGOFF',
        enqueue_options    => enqopt,
        message_properties => msgprop,
        payload            => payload,
        msgid              => enq_msgid);

END;
/

/* Non Persistent queue scenario - procedure to be executed upon logoff: */
CREATE OR REPLACE PROCEDURE  User_Logoff(app_process IN VARCHAR2)
AS
```

```
  msgprop        dbms_aq.message_properties_t;
  enqopt         dbms_aq.enqueue_options_t;
  enq_msgid      RAW(16);
  payload        RAW(1);
BEGIN
  /* Visibility must always be immediate for NonPersistent queues: */
  enqopt.visibility:=dbms_aq.IMMEDIATE;
  msgprop.correlation:= 'LOGOFF';
  msgprop.recipient_list(0) := aq$_agent(app_process, NULL, NULL);
  /* Payload is NULL: */
  dbms_aq.enqueue(
        queue_name         => 'LOGON_LOGOFF',
        enqueue_options    => enqopt,
        message_properties => msgprop,
        payload            => payload,
        msgid              => enq_msgid);
 END;
/


/* If there is a login at APP1, enqueue a message into 'login_logoff' with
   correlation 'LOGIN': */
EXECUTE User_logon('APP1');

/* If there is a logout at APP13 enqueue a message into 'login_logoff' with
   correlation 'LOGOFF': */
EXECUTE User_logoff('App3');


/* The OCI program which waits for notifications: */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <oci.h>
#ifdef WIN32COMMON
#define sleep(x)   Sleep(1000*(x))
#endif

/* LOGON / password:  */
static text *username = (text *) "OE";
static text *password = (text *) "OE";

/* The correlation strings of messages: */
static char  *logon = "LOGON";
static char  *logoff = "LOGOFF";
```

```
/* The possible consumer names of queues: */
static char *applist[] = {"APP1", "APP2","APP3"};

static OCIEnv *envhp;
static OCIServer *srvhp;
static OCIError *errhp;
static OCISvcCtx *svchp;

static void checkerr(/*_ OCIError *errhp, sword status _*/);

struct process_statistics
{
  ub4  logon;
  ub4  logoff;
};

typedef struct process_statistics process_statistics;

int main(/*_ int argc, char *argv[] _*/);


/* Notify Callback: */
ub4 notifyCB(ctx, subscrhp, pay, payl, desc, mode)
dvoid *ctx;
OCISubscription *subscrhp;
dvoid *pay;
ub4    payl;
dvoid *desc;
ub4    mode;
{
 text                *subname;   /* subscription name */
 ub4                  lsub;      /* length of subscription name */
 text                *queue;     /* queue name */
 ub4                 *lqueue;    /* queue name */
 text                *consumer;  /* consumer name */
 ub4                  lconsumer;
 text                *correlation;
 ub4                  lcorrelation;
 ub4                  size;
 ub4                  appno;
 OCIRaw              *msgid;
 OCIAQMsgProperties  *msgprop;   /* message properties descriptor */
 process_statistics   *user_count = (process_statistics *)ctx;
```

```
 OCIAttrGet((dvoid *)subscrhp, OCI_HTYPE_SUBSCRIPTION,
                             (dvoid *)&subname, &lsub,
                             OCI_ATTR_SUBSCR_NAME, errhp);

 /* Extract the attributes from the AQ descriptor: */
 /* Queue name: */
 OCIAttrGet(desc, OCI_DTYPE_AQNFY_DESCRIPTOR, (dvoid *)&queue, &size,
            OCI_ATTR_QUEUE_NAME, errhp);

 /* Consumer name: */
 OCIAttrGet(desc, OCI_DTYPE_AQNFY_DESCRIPTOR, (dvoid *)&consumer, &lconsumer,
            OCI_ATTR_CONSUMER_NAME, errhp);

 /* Message properties: */
 OCIAttrGet(desc, OCI_DTYPE_AQNFY_DESCRIPTOR, (dvoid *)&msgprop, &size,
            OCI_ATTR_MSG_PROP, errhp);

 /* Get correlation from message properties: */
  checkerr(errhp, OCIAttrGet(msgprop, OCI_DTYPE_AQMSG_PROPERTIES,
                             (dvoid *)&correlation, &lcorrelation,
                             OCI_ATTR_CORRELATION, errhp));

  if (lconsumer == strlen(applist[0]))
  {
    if (!memcmp((dvoid *)consumer, (dvoid *)applist[0], strlen(applist[0])))
     appno = 0;
    else if (!memcmp((dvoid *)consumer, (dvoid *)applist[1],
strlen(applist[1])))
     appno = 1;
    else if (!memcmp((dvoid *)consumer, (dvoid *)applist[2],
strlen(applist[2])))
     appno = 2;
    else
    {
     printf("Wrong consumer in notification");
     return;
    }
  }
  else
  {  /* consumer name must be "APP1", "APP2" or "APP3"  */
    printf("Wrong consumer in notification");
    return;
  }

  if (lcorrelation == strlen(logon) &&                  /* logon event */
```

```
      !memcmp((dvoid *)correlation, (dvoid *)logon, strlen(logon)))
  {
     user_count[appno].logon++;
                       /* increment logon count for the app process */
         printf("Logon by APP%d \n", (appno+1));
   }
  else if  (lcorrelation == strlen(logoff) &&           /* logoff event */
      !memcmp((dvoid *)correlation,(dvoid *)logoff, strlen(logoff)))
  {
     user_count[appno].logoff++;
                      /* increment logoff count for the app process */
     printf("Logoff by APP%d \n", (appno+1));
  }
  else                            /* correlation is "LOGON" or "LOGOFF" */
    printf("Wrong correlation in notification");

  printf("Total  : \n");

  printf("App1 : %d \n", user_count[0].logon-user_count[0].logoff);
  printf("App2 : %d \n", user_count[1].logon-user_count[1].logoff);
  printf("App3 : %d \n", user_count[2].logon-user_count[2].logoff);

}

int main(argc, argv)
int argc;
char *argv[];
{
  OCISession *authp = (OCISession *) 0;
  OCISubscription *subscrhp[3];
  ub4 namespace = OCI_SUBSCR_NAMESPACE_AQ;
  process_statistics  ctx[3] = {{0,0}, {0,0}, {0,0}};
  ub4 sleep_time = 0;

  printf("Initializing OCI Process\n");

  /* Initialize OCI environment with OCI_EVENTS flag set: */
  (void) OCIInitialize((ub4) OCI_EVENTS|OCI_OBJECT, (dvoid *)0,
                     (dvoid * (*)(dvoid *, size_t)) 0,
                     (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                     (void (*)(dvoid *, dvoid *)) 0 );

  printf("Initialization successful\n");

  printf("Initializing OCI Env\n");
```

```
  (void) OCIEnvInit( (OCIEnv **) &envhp, OCI_DEFAULT, (size_t) 0, (dvoid **) 0
);
  printf("Initialization successful\n");

  checkerr(errhp, OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp,
OCI_HTYPE_ERROR,
                 (size_t) 0, (dvoid **) 0));

  checkerr(errhp, OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp,
OCI_HTYPE_SERVER,
                 (size_t) 0, (dvoid **) 0));

  checkerr(errhp, OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp,
OCI_HTYPE_SVCCTX,
                 (size_t) 0, (dvoid **) 0));

  printf("connecting to server\n");
  checkerr(errhp, OCIServerAttach( srvhp, errhp, (text *)"inst1_alias",
         strlen("inst1_alias"), (ub4) OCI_DEFAULT));
  printf("connect successful\n");

  /* Set attribute server context in the service context: */
  checkerr(errhp, OCIAttrSet( (dvoid *) svchp, OCI_HTYPE_SVCCTX, (dvoid *)srvhp,
                  (ub4) 0, OCI_ATTR_SERVER, (OCIError *) errhp));

  checkerr(errhp, OCIHandleAlloc((dvoid *) envhp, (dvoid **)&authp,
                     (ub4) OCI_HTYPE_SESSION, (size_t) 0, (dvoid **) 0));

  /* Set username and password in the session handle: */
  checkerr(errhp, OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                (dvoid *) username, (ub4) strlen((char *)username),
                (ub4) OCI_ATTR_USERNAME, errhp));

  checkerr(errhp, OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                (dvoid *) password, (ub4) strlen((char *)password),
                (ub4) OCI_ATTR_PASSWORD, errhp));

  /* Begin session: */
  checkerr(errhp, OCISessionBegin ( svchp,  errhp, authp, OCI_CRED_RDBMS,
                       (ub4) OCI_DEFAULT));

  (void) OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX,
                 (dvoid *) authp, (ub4) 0,
                 (ub4) OCI_ATTR_SESSION, errhp);
```

```
  /* Register for notification: */
  printf("allocating subscription handle\n");
 subscrhp[0] = (OCISubscription *)0;
 (void) OCIHandleAlloc((dvoid *) envhp, (dvoid **)&subscrhp[0],
                       (ub4) OCI_HTYPE_SUBSCRIPTION,
                       (size_t) 0, (dvoid **) 0);

 /* For application process APP1: */
 printf("setting subscription name\n");
 (void) OCIAttrSet((dvoid *) subscrhp[0], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *) "OE.LOGON_LOGOFF:APP1",
                (ub4) strlen("OE.LOGON_LOGOFF:APP1"),
                (ub4) OCI_ATTR_SUBSCR_NAME, errhp);

 printf("setting subscription callback\n");
 (void) OCIAttrSet((dvoid *) subscrhp[0], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *) notifyCB, (ub4) 0,
                (ub4) OCI_ATTR_SUBSCR_CALLBACK, errhp);

(void) OCIAttrSet((dvoid *) subscrhp[0], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *)&ctx, (ub4)sizeof(ctx),
                (ub4) OCI_ATTR_SUBSCR_CTX, errhp);

 printf("setting subscription namespace\n");
 (void) OCIAttrSet((dvoid *) subscrhp[0], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *) &namespace, (ub4) 0,
                (ub4) OCI_ATTR_SUBSCR_NAMESPACE, errhp);

printf("allocating subscription handle\n");
 subscrhp[1] = (OCISubscription *)0;
 (void) OCIHandleAlloc((dvoid *) envhp, (dvoid **)&subscrhp[1],
                       (ub4) OCI_HTYPE_SUBSCRIPTION,
                       (size_t) 0, (dvoid **) 0);

 /* For application process APP2: */
 printf("setting subscription name\n");
 (void) OCIAttrSet((dvoid *) subscrhp[1], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *) "OE.LOGON_LOGOFF:APP2",
                (ub4) strlen("OE.LOGON_LOGOFF:APP2"),
                (ub4) OCI_ATTR_SUBSCR_NAME, errhp);

 printf("setting subscription callback\n");
 (void) OCIAttrSet((dvoid *) subscrhp[1], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *) notifyCB, (ub4) 0,
                (ub4) OCI_ATTR_SUBSCR_CALLBACK, errhp);
```

```
(void) OCIAttrSet((dvoid *) subscrhp[1], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *)&ctx, (ub4)sizeof(ctx),
                (ub4) OCI_ATTR_SUBSCR_CTX, errhp);

 printf("setting subscription namespace\n");
 (void) OCIAttrSet((dvoid *) subscrhp[1], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *) &namespace, (ub4) 0,
                (ub4) OCI_ATTR_SUBSCR_NAMESPACE, errhp);

  printf("allocating subscription handle\n");
 subscrhp[2] = (OCISubscription *)0;
 (void) OCIHandleAlloc((dvoid *) envhp, (dvoid **)&subscrhp[2],
                       (ub4) OCI_HTYPE_SUBSCRIPTION,
                       (size_t) 0, (dvoid **) 0);

 /* For application process APP3: */
 printf("setting subscription name\n");
 (void) OCIAttrSet((dvoid *) subscrhp[2], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *) "OE.LOGON_LOGOFF:APP3",
                (ub4) strlen("OE.LOGON_LOGOFF:APP3"),
                (ub4) OCI_ATTR_SUBSCR_NAME, errhp);

 printf("setting subscription callback\n");
 (void) OCIAttrSet((dvoid *) subscrhp[2], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *) notifyCB, (ub4) 0,
                (ub4) OCI_ATTR_SUBSCR_CALLBACK, errhp);

(void) OCIAttrSet((dvoid *) subscrhp[2], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *)&ctx, (ub4)sizeof(ctx),
                (ub4) OCI_ATTR_SUBSCR_CTX, errhp);

 printf("setting subscription namespace\n");
 (void) OCIAttrSet((dvoid *) subscrhp[2], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *) &namespace, (ub4) 0,
                (ub4) OCI_ATTR_SUBSCR_NAMESPACE, errhp);

 printf("Registering fornotifications \n");
 checkerr(errhp, OCISubscriptionRegister(svchp, subscrhp, 3, errhp,
                                        OCI_DEFAULT));

 sleep_time = (ub4)atoi(argv[1]);
 printf ("waiting for %d s \n", sleep_time);
 sleep(sleep_time);
```

```
  printf("Exiting");
  exit(0);
}

void checkerr(errhp, status)
OCIError *errhp;
sword status;
{
  text errbuf[512];
  sb4 errcode = 0;

  switch (status)
  {
  case OCI_SUCCESS:
    break;
  case OCI_SUCCESS_WITH_INFO:
    (void) printf("Error - OCI_SUCCESS_WITH_INFO\n");
    break;
  case OCI_NEED_DATA:
    (void) printf("Error - OCI_NEED_DATA\n");
    break;
  case OCI_NO_DATA:
    (void) printf("Error - OCI_NODATA\n");
    break;
  case OCI_ERROR:
    (void) OCIErrorGet((dvoid *)errhp, (ub4) 1, (text *) NULL, &errcode,
                        errbuf, (ub4) sizeof(errbuf), OCI_HTYPE_ERROR);
    (void) printf("Error - %.*s\n", 512, errbuf);
    break;
  case OCI_INVALID_HANDLE:
    (void) printf("Error - OCI_INVALID_HANDLE\n");
    break;
  case OCI_STILL_EXECUTING:
    (void) printf("Error - OCI_STILL_EXECUTE\n");
    break;
  case OCI_CONTINUE:
    (void) printf("Error - OCI_CONTINUE\n");
    break;
  default:
    break;
  }
}

/* End of file tkaqdocn.c */
```

Retention and Message History

AQ allows users retain messages in the queue-table which means that SQL can then be used to query these message for analysis. Messages often are related to each other. For example, if a message is produced as a result of the consumption of another message, the two are related. As the application designer, you may want to keep track of such relationships. Along with retention and message identifiers, AQ lets you automatically create message journals, also referred to as tracking journals or event journals. Taken together — retention, message identifiers and SQL queries — make it possible to build powerful message warehouses.

Example Scenario

Let us suppose that the shipping application needs to determine the average processing times of orders. This includes the time the order has to wait in the `backed_order` queue. It would also like to find out the average wait time in the `backed_order` queue. Specifying the retention as `TRUE` for the shipping queues and specifying the order number in the correlation field of the message, SQL queries can be written to determine the wait time for orders in the shipping application.

For simplicity, we will only analyze orders that have already been processed. The processing time for an order in the shipping application is the difference between the enqueue time in the `WS_bookedorders_queue` and the enqueue time in the `WS_shipped_orders_queue`.

Example Code

```
SELECT  SUM(SO.enq_time - BO.enq_time) / count (*) AVG_PRCS_TIME
   FROM WS.AQ$WS_orders_pr_mqtab BO , WS.AQ$WS_orders_mqtab SO
   WHERE SO.msg_state = 'PROCESSED' and BO.msg_state = 'PROCESSED'
   AND SO.corr_id = BO.corr_id and SO.queue = 'WS_shippedorders_que';

/* Average waiting time in the backed order queue: */
SELECT SUM(BACK.deq_time - BACK.enq_time)/count (*) AVG_BACK_TIME
   FROM WS.AQ$WS_orders_mqtab BACK
   WHERE BACK.msg_state = 'PROCESSED' AND BACK.queue = 'WS_backorders_que';
```

Publish/Subscribe Support

Oracle AQ adds various features that allow you to develop an application based on a publish/subscribe model. The aim of this application model is to enable flexible and dynamic communication between applications functioning as publishers and applications playing the role of subscribers. The specific design point is that the applications playing these different roles should be decoupled in their communication, that they should interact based on messages and message content.

In distributing messages publisher applications do not have to explicitly handle or manage message recipients. This allows the dynamic addition of new subscriber applications to receive messages without changing any publisher application logic. Subscriber applications receive messages based on message content without regarding to which publisher applications are sending messages. This allows the dynamic addition of subscriber applications without changing any subscriber application logic. Subscriber applications specify interest by defining a rule-based subscription on message content (payload) and message header properties of a queue. The system automatically routes messages by computing recipients for published messages using the rule-based subscriptions.

You can implement a publish/subscribe model of communication using AQ by taking the following steps:

- Set up one or more queues to hold messages. These queues should represent an area or subject of interest. For example, a queue can be used to represent billed orders.
- Set up a set of rule based subscribers. Each subscriber may specify a rule which represents a specification for the messages that the subscriber wishes to receive. A null rule indicates that the subscriber wishes to receive all messages.
- Publisher applications publish messages to the queue by invoking an enqueue call.
- Subscriber applications may receive messages in the following manner.
- A dequeue call retrieves messages that match the subscription criteria.
- A listen call may be used to monitor multiple queues for subscriptions on different queues. This is a more scalable solution in cases in which a subscriber application has subscribed to many queues and wishes to receive messages that arrive in any of the queues.
- Use the OCI notification mechanism. This allows a "push" mode of message delivery in which the subscriber application registers the queues (and subscriptions specified as subscribing agent) from which to receive messages

from and registers a callback to be invoked when messages matching the subscriptions arrive.

Example Scenario

The `BooksOnLine` application illustrates the use of a publish/subscribe model for communicating between applications. For example,

Define queues The Order Entry application defines a queue (`OE_booked_orders_que`) to communicate orders that are booked to various applications. The Order Entry application is not aware of the various subscriber applications and thus, a new subscriber application may be added without disrupting any setup or logic in the Order Entry (publisher) application.

Set up Subscriptions The various shipping applications and the customer service application (i.e., Eastern region shipping, Western region shipping, Overseas shipping and Customer Service) are defined as subscribers to the `booked_orders` queue of the Order Entry application. Rules are used to route messages of interest to the various subscribers. Thus, Eastern Region shipping, which handles shipment of all orders for the East coast and all rush US orders, would express its subscription rule as follows;

```
rule  => 'tab.user_data.orderregion = ''EASTERN'' OR
(tab.user_data.ordertype = ''RUSH'' AND
tab.user_data.customer.country = ''USA'') '
```

Each subscriber can specify a local queue to which messages are to be delivered. The Eastern region shipping application specifies a local queue (`ES_booked_orders_que`) for message delivery by specifying the subscriber address as follows:

```
subscriber := aq$_agent('East_Shipping', 'ES.ES_bookedorders_que', null);
```

Set up propagation Enable propagation from each publisher application queue. To allow subscribed messages to be delivered to remote queues, the Order Entry application enables propagation by means of the following statement:

```
execute dbms_aqadm.schedule_propagation(queue_name => 'OE.OE_bookedorders_que');
```

Publish Messages Booked orders are published by the Order Entry application when it enqueues orders (into the `OE_booked_order_que`) that have been validated and are ready for shipping. These messages are then routed to each of the subscribing applications. Messages are delivered to local queues (if specified) at each of the subscriber applications.

Receive Messages Each of the shipping applications and the Customer Service application will then receive these messages in their local queues. For example, Eastern Region Shipping only receives booked orders that are for East Coast addresses or any US order that is marked `RUSH`. This application then dequeues messages and processes its orders for shipping.

Support for Oracle Parallel Server (OPS)

The Oracle Parallel Server facility can be used to improve AQ performance by allowing different queues to be managed by different instances. You do this by specifying different instance affinities (preferences) for the queue tables that store the queues. This allows queue operations (enqueue / dequeue) on different queues to occur in parallel.

The AQ queue monitor process continuously monitors the instance affinities of the queue tables. The queue monitor assigns ownership of a queue table to the specified primary instance if it is available, failing which it assigns it to the specified secondary instance. If the owner instance of a queue table ceases to exist at any time, the queue monitor changes the ownership of the queue table to a suitable instance — the secondary instance or some other available instance if the secondary instance is also unavailable.

AQ propagation is able to make use of OPS although it is completely transparent to the user. The affinities for jobs submitted on behalf of the propagation schedules are set to the same values as that of the affinities of the respective queue tables. Thus a job_queue_process associated with the owner instance of a queue table will be handling the propagation from queues stored in that queue table thereby minimizing ’pinging’. Additional discussion on this topic can be found under AQ propagation scheduling (see "Schedule a Queue Propagation" on page 4-56 in Chapter 4, "Administrative Interface: Basic Operations").

For information about Oracle Parallel Server (OPS) see:

- *Oracle8i Parallel Server Setup and Configuration Guide*

Example Scenario

In the `BooksOnLine` example, operations on the `new_orders_queue` and `booked_order_queue` at the order entry (OE) site can be made faster if the two queues are associated with different instances. This is done by creating the queues in different queue tables and specifying different affinities for the queue tables in the create_queue_table() command.

In the example, the queue table `OE_orders_sqtab` stores queue `new_orders_queue` and the primary and secondary are instances 1 and 2 respectively. For queue table `OE_orders_mqtab` stores queue `booked_order_queue` and the primary and secondary are instances 2 and 1 respectively. The objective is to let instances 1 & 2 manage the two queues in parallel. By default, only one instance is available in which case the owner instances of both queue tables will be set to instance 1.

However, if OPS is setup correctly and both instances 1 and 2 are available, then queue table `OE_orders_sqtab` will be owned by instance 1 and the other queue table will be owned by instance 2. The primary and secondary instance specification of a queue table can be changed dynamically using the `alter_queue_table`() command as shown in the example below. Information about the primary, secondary and owner instance of a queue table can be obtained by querying the view `USER_QUEUE_TABLES` (see "Select Queue Tables in User Schema" on page 5-25 in "Administrative Interface: Views").

Example Code

```
/* Create queue tables, queues for OE  */
CONNECT OE/OE;
EXECUTE dbms_aqadm.create_queue_table( \
        queue_table        => 'OE_orders_sqtab',\
        comment            => 'Order Entry Single-Consumer Orders queue table',\
        queue_payload_type => 'BOLADM.order_typ',\
        compatible         => '8.1',\
        primary_instance   => 1,\
        secondary_instance => 2);

EXECUTE dbms_aqadm.create_queue_table(\
        queue_table        => 'OE_orders_mqtab',\
        comment            => 'Order Entry Multi Consumer Orders queue table',\
        multiple_consumers => TRUE,\
        queue_payload_type => 'BOLADM.order_typ',\
        compatible         => '8.1',\
        primary_instance   => 2,\
        secondary_instance => 1);

EXECUTE dbms_aqadm.create_queue ( \
        queue_name         => 'OE_neworders_que',\
        queue_table        => 'OE_orders_sqtab');

EXECUTE dbms_aqadm.create_queue ( \
        queue_name         => 'OE_bookedorders_que',\
        queue_table        => 'OE_orders_mqtab');

/* Check instance affinity of OE queue tables from AQ administrative view: */
SELECT queue_table, primary_instance, secondary_instance, owner_instance
FROM user_queue_tables;

/* Alter instance affinity of OE queue tables: */
EXECUTE dbms_aqadm.alter_queue_table( \
```

```
        queue_table        => ’OE.OE_orders_sqtab’,\
        primary_instance   => 2,\
        secondary_instance => 1);

EXECUTE dbms_aqadm.alter_queue_table(  \
        queue_table        => ’OE.OE_orders_mqtab’, \
        primary_instance   => 1,\
        secondary_instance => 2);

/* Check instance affinity of OE queue tables from AQ administrative view: */
SELECT queue_table, primary_instance, secondary_instance, owner_instance
FROM user_queue_tables;
```

Support for Statistics Views

Each instance keeps its own AQ statistics information in its own SGA, and does not have knowledge of the statistics gathered by other instances. Then, when a GV$AQ view is queried by an instance, all other instances funnel their AQ statistics information to the instance issuing the query.

Example Scenario

The `gv$` view can be queried at any time to see the number of messages in waiting, ready or expired state. The view also displays the average number of seconds for which messages have been waiting to be processed. The order processing application can use this to dynamically tune the number of order processing processes (see "Select the Number of Messages in Different States for the Whole Database" on page 5-39 in Chapter 5, "Administrative Interface: Views").

Example Code

```
CONNECT oe/oe

/* Count the number as messages and the average time for which the messages have
   been waiting: */
SELECT READY, AVERAGE_WAIT FROM gv$aq Stats, user_queues Qs
  WHERE Stats.qid = Qs.qid and Qs.Name = 'OE_neworders_que';
```

ENQUEUE Features

- Subscriptions and Recipient Lists
- Priority and Ordering of Messages
- Time Specification: Delay
- Time Specification: Expiration
- Message Grouping
- Asynchronous Notifications

Subscriptions and Recipient Lists

In a single-consumer queue a message can be processed once by only one consumer. What happens when there are multiple processes or operating system threads concurrently dequeuing from the same queue? Given that a locked message cannot be dequeued by a process other than the one which has created the lock, each process will dequeue the first unlocked message that is at the head of the queue. After processing, the message is removed if the `retention_time` of the queue is 0, or retained for the specified retention time. While the message is retained the message can be either queried using SQL on the queue table view or by dequeuing using the `BROWSE` mode and specifying the message ID of the processed message.

AQ allows a single message to be processed/consumed by more than one consumer. To use this feature, you must create multi-consumer queues and enqueue the messages into these multi-consumer queues. AQ allows two methods of identifying the list of consumers for a message: subscriptions and recipient lists.

Subscriptions

You can add a subscription to a queue by using the `DBMS_AQADM.ADD_SUBSCRIBER` PL/SQL procedure (see "Add a Subscriber" on page 4-46 in Chapter 4, "Administrative Interface: Basic Operations"). This lets you specify a consumer by means of the `AQ$_AGENT` parameter for enqueued messages. You can add more subscribers by repeatedly using the `DBMS_AQADM.ADD_SUBSCRIBER` procedure up to a maximum of 1024 subscribers for a multi-consumer queue. (Note that you are limited to 32 subscriber for multi-consumer queue created using Oracle 8.0.3.)

All consumers that are added as subscribers to a multi-consumer queue must have unique values for the `AQ$_AGENT` parameter. This means that two subscribers cannot have the same values for the `NAME`, `ADDRESS` and `PROTOCOL` attributes for the `AQ$_AGENT` type. At least one of the three attributes must be different for two subscribers (see "Agent" on page 3-5 in Chapter 3, "Managing Oracle AQ" for formal description of this data structure).

you cannot add subscriptions to single-consumer queues or exception queues. A consumer that is added as a subscriber to a queue will only be able to dequeue messages that are enqueued after the `DBMS_AQADM.ADD_SUBSCRIBER` procedure is completed. In other words, messages that had been enqueued before this procedure is executed will not be available for dequeue by this consumer.

You can remove a subscription by using the `DBMS_AQADM.REMOVE_SUBSCRIBER` procedure (see "Remove a Subscriber" in Chapter 4, "Administrative Interface: Basic Operations"). AQ will automatically remove from the queue all metadata corresponding to the consumer identified by the `AQ$_AGENT` parameter. In other words, it is not an error to execute the `REMOVE_SUBSCRIBER` procedure even when there are pending messages that are available for dequeue by the consumer. These messages will be automatically made unavailable for dequeue after the `REMOVE_SUBSCRIBER` procedure is executed. In a queue table that is created with the compatible parameter set to '8.1' or higher, such messages that were not dequeued by the consumer will be shown as "`UNDELIVERABLE`" in the `AQ$<queue_table>` view. Note that a multi-consumer queue table created without the compatible parameter, or with the compatible parameter set to '8.0', does not display the state of a message on a consumer basis, but only displays the global state of the message.

Recipient Lists

You do not need to specify subscriptions for a multi-consumer queue provided that producers of messages for enqueue supply a recipient list of consumers. In some situations it may be desirable to enqueue a message that is targeted to a specific set of consumers rather than the default list of subscribers. You accomplish this by specifying a recipient list at the time of enqueuing the message.

- In PL/SQL you specify the recipient list by adding elements to the `recipient_list` field of the `message_properties` record.
- In OCI the recipient list is specified by using the `OCISetAttr` procedure to specify an array of `OCI_DTYPE_AQAGENT` descriptors as the recipient list (`OCI_ATTR_RECIPIENT_LIST` attribute) of an `OCI_DTYPE_AQMSG_PROPERTIES` message properties descriptor.

If a recipient list is specified during enqueue, it overrides the subscription list. In other words, messages that have a specified recipient list will not be available for dequeue by the subscribers of the queue. The consumers specified in the recipient list may or may not be subscribers for the queue. It is an error if the queue does not have any subscribers and the enqueue does not specify a recipient list (see "Enqueue a Message" on page 6-4 in Chapter 6, "Operational Interface: Basic Operations").

Priority and Ordering of Messages

The message ordering dictates the order in which messages will be dequeued from a queue. The ordering method for a queue is specified when a queue table is created (see "Create a Queue Table" on page 4-4 in Chapter 4, "Administrative Interface: Basic Operations"). Currently, AQ supports two types of message ordering:

- Priority ordering of messages. If priority ordering is chosen, each message will be assigned a priority at enqueue time by the enqueuer. At dequeue time, the messages will be dequeued in the order of the priorities assigned. If two messages have the same priority, the order at which they are dequeued is undetermined.
- First-In, First-Out (FIFO) ordering. A FIFO-priority queue can also be created by specifying both the priority and the enqueue time as the sort order of the messages. A FIFO-priority queue behaves like a priority queue, except if two messages are assigned the same priority, they will be dequeued according to the order of their enqueue time.

Example Scenario

In the `BooksOnLine` application, a customer can request

- FedEx shipping (priority 1),
- Priority air shipping (priority 2). or
- Regular ground shipping (priority 3).

The Order Entry application uses a FIFO-priority queue to store booked orders. Booked orders are propagated to the regional booked orders queues. At each region, orders in these regional booked orders queues are processed in the order of the shipping priorities.

The following calls create the FIFO-priority queues for the Order Entry application.

Example Code

```
/* Create a priority queue table for OE: */
EXECUTE dbms_aqadm.create_queue_table( \
   queue_table         => 'OE_orders_pr_mqtab', \
   sort_list           =>'priority,enq_time', \
   comment             => 'Order Entry Priority  \
                          MultiConsumer Orders queue table',\
   multiple_consumers  => TRUE, \
   queue_payload_type  => 'BOLADM.order_typ', \
   compatible          => '8.1', \
```

```
   primary_instance    => 2, \
   secondary_instance  => 1);

EXECUTE dbms_aqadm.create_queue ( \
   queue_name          => 'OE_bookedorders_que', \
   queue_table         => 'OE_orders_pr_mqtab');

/* When an order arrives, the order entry application can use the following
   procedure to enqueue the order into its booked orders queue. A shipping
   priority is specified for each order: */
CREATE OR REPLACE procedure order_enq(book_title        IN VARCHAR2,
                                      book_qty          IN NUMBER,
                                      order_num         IN NUMBER,
                                      shipping_priority IN NUMBER,
                                      cust_state        IN VARCHAR2,
                                      cust_country      IN VARCHAR2,
                                      cust_region       IN VARCHAR2,
                                      cust_ord_typ      IN VARCHAR2) AS

OE_enq_order_data        BOLADM.order_typ;
OE_enq_cust_data         BOLADM.customer_typ;
OE_enq_book_data         BOLADM.book_typ;
OE_enq_item_data         BOLADM.orderitem_typ;
OE_enq_item_list         BOLADM.orderitemlist_vartyp;
enqopt                   dbms_aq.enqueue_options_t;
msgprop                  dbms_aq.message_properties_t;
enq_msgid                RAW(16);

BEGIN
   msgprop.correlation := cust_ord_typ;
   OE_enq_cust_data    := BOLADM.customer_typ(NULL, NULL, NULL, NULL,
                                cust_state, NULL, cust_country);
   OE_enq_book_data    := BOLADM.book_typ(book_title, NULL, NULL, NULL);
   OE_enq_item_data    := BOLADM.orderitem_typ(book_qty,
                                OE_enq_book_data, NULL);
   OE_enq_item_list    := BOLADM.orderitemlist_vartyp(
                                BOLADM.orderitem_typ(book_qty,
                                OE_enq_book_data, NULL));
   OE_enq_order_data   := BOLADM.order_typ(order_num, NULL,
                                cust_ord_typ, cust_region,
                                OE_enq_cust_data, NULL,
                                OE_enq_item_list, NULL);

   /*Put the shipping priority into message property before enqueueing
     the message: */
```

```
   msgprop.priority    := shipping_priority;
   dbms_aq.enqueue('OE.OE_bookedorders_que', enqopt, msgprop,
                     OE_enq_order_data, enq_msgid);
        COMMIT;
  END;
  /


/* At each region, similar booked order queues are created. The orders are
   propagated from the central Order Entry's booked order queues to the regional
   booked order queues.For example, at the western region, the booked orders
   queue is created.
   Create a priority queue table for WS shipping: */
EXECUTE dbms_aqadm.create_queue_table( \
   queue_table        =>  'WS_orders_pr_mqtab',
   sort_list          =>'  priority,enq_time',  \
   comment            =>  'West Shipping Priority  \
                            MultiConsumer Orders queue table',\
   multiple_consumers => TRUE, \
   queue_payload_type => 'BOLADM.order_typ', \
   compatible         => '8.1');

/* Booked orders are stored in the priority queue table: */
EXECUTE dbms_aqadm.create_queue ( \
   queue_name         => 'WS_bookedorders_que', \
   queue_table        => 'WS_orders_pr_mqtab');

/* At each region, the shipping application dequeues orders from the regional
   booked order queue according to the orders' shipping priorities, processes
   the orders, and enqueues the processed orders into the shipped orders queues
   or the back orders queues. */
```

Time Specification: Delay

Messages can be enqueued to a queue with a delay. The delay represents a time interval after which the message becomes available for dequeuing. A message specified with a delay is in a waiting state until the delay expires and the message becomes available. Note that delay processing requires the queue monitor to be started. Note also that dequeuing by msgid overrides the delay specification.

Example Scenario

In the `BooksOnLine` application, delay can be used to implement deferred billing. A billing application can define a queue in which shipped orders that are not billed immediately can be placed in a deferred billing queue with a delay. For example, a certain class of customer accounts, such as those of corporate customers, may not be billed for 15 days. The billing application dequeues incoming shipped order messages (from the shippedorders queue) and if the order is for a corporate customer, this order is enqueued into a deferred billing queue with a delay.

Example Code

```
/* Enqueue an order to implement deferred billing so that the order is not made
   visible again until delay has expired: */
CREATE OR REPLACE PROCEDURE defer_billing(deferred_billing_order order_typ)
AS
  defer_bill_queue_name    VARCHAR2(62);
  enqopt                   dbms_aq.enqueue_options_t;
  msgprop                  dbms_aq.message_properties_t;
  enq_msgid                RAW(16);
BEGIN

/* Enqueue the order into the deferred billing queue with a delay of 15 days: */
  defer_bill_queue_name := 'CBADM.deferbilling_que';
  msgprop.delay := 15*60*60*24;
  dbms_aq.enqueue(defer_bill_queue_name, enqopt, msgprop,
                  deferred_billing_order, enq_msgid);
END;
/
```

Time Specification: Expiration

Messages can be enqueued with an expiration which specifies the interval of time the message is available for dequeuing. Note that expiration processing requires that the queue monitor be running.

Example Scenario

In the `BooksOnLine` application, expiration can be used to control the amount of time that is allowed to process a back order. The shipping application places orders for books that are not available on a back order queue. If the shipping policy is that all back orders must be shipped within a week, then messages can be enqueued into the back order queue with an expiration of 1 week. In this case, any back orders that are not processed within one week are moved to the exception queue with the message state set to `EXPIRED`. This can be used to flag any orders that have not been shipped according to the back order shipping policy.

Example Code

```
CONNECT BOLADM/BOLADM
/* Req-enqueue a back order into a back order queue and set a delay of 7 days;
   all back orders must be processed in 7 days or they are moved to the
   exception queue: */
CREATE OR REPLACE PROCEDURE requeue_back_order(sale_region varchar2,
                                               backorder order_typ)
AS
  back_order_queue_name    VARCHAR2(62);
  enqopt                   dbms_aq.enqueue_options_t;
  msgprop                  dbms_aq.message_properties_t;
  enq_msgid                RAW(16);
BEGIN
  /* Look up a back order queue based the the region by means of a directory
     service: */
  IF sale_region = 'WEST' THEN
    back_order_queue_name := 'WS.WS_backorders_que';
  ELSIF sale_region = 'EAST' THEN
    back_order_queue_name := 'ES.ES_backorders_que';
  ELSE
    back_order_queue_name := 'OS.OS_backorders_que';
  END IF;

  /* Enqueue the order with expiration set to 7 days: */
  msgprop.expiration := 7*60*60*24;
  dbms_aq.enqueue(back_order_queue_name, enqopt, msgprop,
                  backorder, enq_msgid);
```

```
END;
/
```

Message Grouping

Messages belonging to one queue can be grouped to form a set that can only be consumed by one user at a time. This requires the queue be created in a queue table that is enabled for transactional message grouping (see "Create a Queue Table" on page 4-4 in Chapter 4, "Administrative Interface: Basic Operations"). All messages belonging to a group have to be created in the same transaction and all messages created in one transaction belong to the same group. This feature allows you to segment complex messages into simple messages.

For example, messages directed to a queue containing invoices could be constructed as a group of messages starting with the header message, followed by messages representing details, followed by the trailer message. Message grouping is also very useful if the message payload contains complex large objects such as images and video that can be segmented into smaller objects.

The general message properties (priority, delay, expiration) for the messages in a group are determined solely by the message properties specified for the first message (head) of the group irrespective of which properties are specified for subsequent messages in the group.

The message grouping property is preserved across propagation. However, it is important to note that the destination queue to which messages have to be propagated must also be enabled for transactional grouping. There are also some restrictions you need to keep in mind if the message grouping property is to be preserved while dequeuing messages from a queue enabled for transactional grouping (see "Dequeue Methods" on page 2-47 and "Modes of Dequeuing" on page 2-57 for additional information).

Example Scenario

In the `BooksOnLine` application, message grouping can be used to handle new orders. Each order contains a number of books ordered one by one in succession. Items ordered over the Web exhibit similar behavior.

In the example given below, each enqueue corresponds to an individual book that is part of an order and the group/transaction represents a complete order. Only the first enqueue contains customer information. Note that the `OE_neworders_que` is stored in the table `OE_orders_sqtab` which has been enabled for transactional grouping. Refer to the example code for descriptions of procedures `new_order_enq`() and `same_order_enq`().

Example Code

```
connect OE/OE;

/* Create queue table for OE:  */
EXECUTE dbms_aqadm.create_queue_table( \
        queue_table        => 'OE_orders_sqtab',\
        comment            => 'Order Entry Single-Consumer Orders queue table',\
        queue_payload_type => 'BOLADM.order_typ',\
        message_grouping   => DBMS_AQADM.TRANSACTIONAL, \
        compatible         => '8.1',  \
        primary_instance   => 1,\
        secondary_instance => 2);

/* Create neworders queue for OE: */
EXECUTE dbms_aqadm.create_queue ( \
        queue_name         => 'OE_neworders_que', \
        queue_table        => 'OE_orders_sqtab');

/* Login into OE account :*/
CONNECT OE/OE;
SET serveroutput on;

/* Enqueue some orders using message grouping into OE_neworders_que,
   First Order Group: */
EXECUTE BOLADM.new_order_enq('My First   Book', 1, 1001, 'CA');
EXECUTE BOLADM.same_order_enq('My Second  Book', 2);
COMMIT;
/
/* Second Order Group: */
EXECUTE BOLADM.new_order_enq('My Third   Book', 1, 1002, 'WA');
COMMIT;
/
/* Third Order Group: */
EXECUTE BOLADM.new_order_enq('My Fourth  Book', 1, 1003, 'NV');
EXECUTE BOLADM.same_order_enq('My Fifth   Book', 3);
EXECUTE BOLADM.same_order_enq('My Sixth   Book', 2);
COMMIT;
/
/* Fourth Order Group: */
EXECUTE BOLADM.new_order_enq('My Seventh Book', 1, 1004, 'MA');
EXECUTE BOLADM.same_order_enq('My Eighth  Book', 3);
EXECUTE BOLADM.same_order_enq('My Ninth   Book', 2);
COMMIT;
/
```

Asynchronous Notifications

This feature allows OCI clients to receive notifications when there is a message in a queue of interest. The client can use it to monitor multiple subscriptions. The client does not have to be connected to the database to receive notifications regarding its subscriptions.

You use the OCI function, `OCISubcriptionRegister`, to register interest in messages in a queue (see "Register for Notification" in Chapter 6, "Operational Interface: Basic Operations").

For more information about the OCI operation `Register for Notification` see:

- *Oracle Call Interface Programmer's Guide*

The client can specify a callback function which is invoked for every new message that is enqueued. For non-persistent queues, the message is delivered to the client as part of the notification. For persistent queues, only the message properties are delivered as part of the notification. Consequently, in the case of persistent queues, the client has to make an explicit dequeue to access the contents of the message.

Example Scenario

In the `BooksOnLine` application, a customer can request Fed-ex shipping (priority 1), Priority air shipping (priority 2). or Regular ground shipping (priority 3).

The shipping application then ships the orders according to the user's request. It is of interest to `BooksOnLine` to find out how many requests of each shipping type come in each day. The application uses asynchronous notification facility for this purpose. It registers for notification on the `WS.WS_bookedorders_que`. When it is notified of new message in the queue, it updates the count for the appropriate shipping type depending on the priority of the message.

Example Code

This example illustrates the use of OCIRegister. At the shipping site, an OCI client program keeps track of how many orders were made for each of the shipping types, FEDEX, AIR and GROUND. The priority field of the message enables us to determine the type of shipping desired.

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

```
#include <string.h>
#include <oci.h>
#ifdef WIN32COMMON
#define sleep(x)   Sleep(1000*(x))
#endif
static text *username = (text *) "WS";
static text *password = (text *) "WS";

static OCIEnv *envhp;
static OCIServer *srvhp;
static OCIError *errhp;
static OCISvcCtx *svchp;

static void checkerr(/*_ OCIError *errhp, sword status _*/);

struct ship_data
{
  ub4  fedex;
  ub4  air;
  ub4  ground;
};

typedef struct ship_data ship_data;

int main(/*_ int argc, char *argv[] _*/);


/* Notify callback: */
ub4 notifyCB(ctx, subscrhp, pay, payl, desc, mode)
dvoid *ctx;
OCISubscription *subscrhp;
dvoid *pay;
ub4    payl;
dvoid *desc;
ub4    mode;
{
 text                *subname;
 ub4                 size;
 ship_data           *ship_stats = (ship_data *)ctx;
 text                *queue;
 text                *consumer;
 OCIRaw              *msgid;
 ub4                 priority;
 OCIAQMsgProperties  *msgprop;
```

```
 OCIAttrGet((dvoid *)subscrhp, OCI_HTYPE_SUBSCRIPTION,
                             (dvoid *)&subname, &size,
                             OCI_ATTR_SUBSCR_NAME, errhp);

 /* Extract the attributes from the AQ descriptor.
    Queue name: */
 OCIAttrGet(desc, OCI_DTYPE_AQNFY_DESCRIPTOR, (dvoid *)&queue, &size,
            OCI_ATTR_QUEUE_NAME, errhp);

 /* Consumer name: */
 OCIAttrGet(desc, OCI_DTYPE_AQNFY_DESCRIPTOR, (dvoid *)&consumer, &size,
            OCI_ATTR_CONSUMER_NAME, errhp);

 /* Msgid: */
 OCIAttrGet(desc, OCI_DTYPE_AQNFY_DESCRIPTOR, (dvoid *)&msgid, &size,
            OCI_ATTR_NFY_MSGID, errhp);

 /* Message properties: */
 OCIAttrGet(desc, OCI_DTYPE_AQNFY_DESCRIPTOR, (dvoid *)&msgprop, &size,
            OCI_ATTR_MSG_PROP, errhp);

 /* Get priority from message properties: */
 checkerr(errhp, OCIAttrGet(msgprop, OCI_DTYPE_AQMSG_PROPERTIES,
                             (dvoid *)&priority, 0,
                             OCI_ATTR_PRIORITY, errhp));

  switch (priority)
  {
  case 1:  ship_stats->fedex++;
           break;
  case 2 : ship_stats->air++;
           break;
  case 3:  ship_stats->ground++;
           break;
  default:
           printf(" Error priority %d", priority);
  }
}


int main(argc, argv)
int argc;
char *argv[];
{
  OCISession *authp = (OCISession *) 0;
```

```
  OCISubscription *subscrhp[8];
  ub4 namespace = OCI_SUBSCR_NAMESPACE_AQ;
  ship_data  ctx = {0,0,0};
  ub4 sleep_time = 0;

  printf("Initializing OCI Process\n");

  /* Initialize OCI environment with OCI_EVENTS flag set: */
  (void) OCIInitialize((ub4) OCI_EVENTS|OCI_OBJECT, (dvoid *)0,
                       (dvoid * (*)(dvoid *, size_t)) 0,
                       (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                       (void (*)(dvoid *, dvoid *)) 0 );

  printf("Initialization successful\n");

  printf("Initializing OCI Env\n");
  (void) OCIEnvInit( (OCIEnv **) &envhp, OCI_DEFAULT, (size_t) 0, (dvoid **) 0
);
  printf("Initialization successful\n");

  checkerr(errhp, OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, OCI_HTYPE_
ERROR,
                   (size_t) 0, (dvoid **) 0));

  checkerr(errhp, OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, OCI_HTYPE_
SERVER,
                   (size_t) 0, (dvoid **) 0));

  checkerr(errhp, OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, OCI_HTYPE_
SVCCTX,
                   (size_t) 0, (dvoid **) 0));


  printf("connecting to server\n");
  checkerr(errhp, OCIServerAttach( srvhp, errhp, (text *)"inst1_alias",
           strlen("inst1_alias"), (ub4) OCI_DEFAULT));
  printf("connect successful\n");

  /* Set attribute server context in the service context: */
  checkerr(errhp, OCIAttrSet( (dvoid *) svchp, OCI_HTYPE_SVCCTX, (dvoid *)srvhp,
                    (ub4) 0, OCI_ATTR_SERVER, (OCIError *) errhp));

  checkerr(errhp, OCIHandleAlloc((dvoid *) envhp, (dvoid **)&authp,
                       (ub4) OCI_HTYPE_SESSION, (size_t) 0, (dvoid **) 0));
```

```
 /* Set username and password in the session handle: */
 checkerr(errhp, OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                (dvoid *) username, (ub4) strlen((char *)username),
                (ub4) OCI_ATTR_USERNAME, errhp));

 checkerr(errhp, OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
                (dvoid *) password, (ub4) strlen((char *)password),
                (ub4) OCI_ATTR_PASSWORD, errhp));

 /* Begin session: */
 checkerr(errhp, OCISessionBegin ( svchp,  errhp, authp, OCI_CRED_RDBMS,
                        (ub4) OCI_DEFAULT));

 (void) OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX,
                 (dvoid *) authp, (ub4) 0,
                 (ub4) OCI_ATTR_SESSION, errhp);


 /* Register for notification: */
 printf("allocating subscription handle\n");
 subscrhp[0] = (OCISubscription *)0;
 (void) OCIHandleAlloc((dvoid *) envhp, (dvoid **)&subscrhp[0],
                      (ub4) OCI_HTYPE_SUBSCRIPTION,
                      (size_t) 0, (dvoid **) 0);

 printf("setting subscription name\n");
 (void) OCIAttrSet((dvoid *) subscrhp[0], (ub4) OCI_HTYPE_SUBSCRIPTION,
               (dvoid *) "WS.WS_BOOKEDORDERS_QUE:BOOKED_ORDERS",
               (ub4) strlen("WS.WS_BOOKEDORDERS_QUE:BOOKED_ORDERS"),
               (ub4) OCI_ATTR_SUBSCR_NAME, errhp);

 printf("setting subscription callback\n");
 (void) OCIAttrSet((dvoid *) subscrhp[0], (ub4) OCI_HTYPE_SUBSCRIPTION,
               (dvoid *) notifyCB, (ub4) 0,
               (ub4) OCI_ATTR_SUBSCR_CALLBACK, errhp);

(void) OCIAttrSet((dvoid *) subscrhp[0], (ub4) OCI_HTYPE_SUBSCRIPTION,
               (dvoid *)&ctx, (ub4)sizeof(ctx),
               (ub4) OCI_ATTR_SUBSCR_CTX, errhp);

 printf("setting subscription namespace\n");
 (void) OCIAttrSet((dvoid *) subscrhp[0], (ub4) OCI_HTYPE_SUBSCRIPTION,
               (dvoid *) &namespace, (ub4) 0,
               (ub4) OCI_ATTR_SUBSCR_NAMESPACE, errhp);
```

```
  printf("Registering \n");
  checkerr(errhp, OCISubscriptionRegister(svchp, subscrhp, 1, errhp,
                                        OCI_DEFAULT));

  sleep_time = (ub4)atoi(argv[1]);
  printf ("waiting for %d s", sleep_time);
  sleep(sleep_time);

  printf("Exiting");
  exit(0);
}

void checkerr(errhp, status)
OCIError *errhp;
sword status;
{
  text errbuf[512];
  sb4 errcode = 0;

  switch (status)
  {
  case OCI_SUCCESS:
    break;
  case OCI_SUCCESS_WITH_INFO:
    (void) printf("Error - OCI_SUCCESS_WITH_INFO\n");
    break;
  case OCI_NEED_DATA:
    (void) printf("Error - OCI_NEED_DATA\n");
    break;
  case OCI_NO_DATA:
    (void) printf("Error - OCI_NODATA\n");
    break;
  case OCI_ERROR:
    (void) OCIErrorGet((dvoid *)errhp, (ub4) 1, (text *) NULL, &errcode,
                    errbuf, (ub4) sizeof(errbuf), OCI_HTYPE_ERROR);
    (void) printf("Error - %.*s\n", 512, errbuf);
    break;
  case OCI_INVALID_HANDLE:
    (void) printf("Error - OCI_INVALID_HANDLE\n");
    break;
  case OCI_STILL_EXECUTING:
    (void) printf("Error - OCI_STILL_EXECUTE\n");
    break;
  case OCI_CONTINUE:
    (void) printf("Error - OCI_CONTINUE\n");
```

```
        break;
      default:
        break;
      }
    }
```

DEQUEUE Features

- Dequeue Methods
- Multiple Recipients
- Local and Remote Recipients
- Message Navigation in Dequeue
- Modes of Dequeuing
- Optimization of Waiting for Arrival of Messages
- Retry with Delay Interval
- Exception Handling
- Rule-based Subscription
- Listen Capability

Dequeue Methods

A message can be dequeued from a queue using one of two dequeue methods: a correlation identifier or a message identifier.

A correlation identifier is a user defined message property (of `VARCHAR2` datatype) while a message identifier is a system-assigned value (of `RAW` datatype). Multiple messages with the same correlation identifier can be present in a queue while only one message with a given message identifier can be present. A dequeue call with a correlation identifier will directly remove a message of specific interest rather than using a combination of locked and remove mode to first examine the content and then remove the message. Hence, the correlation identifier usually contains the most useful attribute of a payload. If there are multiple messages with the same correlation identifier, the ordering (enqueue order) between messages may not be preserved on dequeue calls. The correlation identifier cannot be changed between successive dequeue calls without specifying the first message navigation option.

Note that dequeueing a message with either of the two dequeue methods will not preserve the message grouping property (see "Message Grouping" on page 2-37 and "Message Navigation in Dequeue" on page 2-54 for further information).

Example Scenario

In the following scenario of the `BooksOnLine` example, rush orders received by the East shipping site are processed first. This is achieved by dequeueing the message using the correlation identifier which has been defined to contain the order type (rush/normal). For an illustration of dequeueing using a message identifier please refer to the `get_northamerican_orders` procedure discussed in the example under "Modes of Dequeuing" on page 2-57.

Example Code

```
CONNECT boladm/boladm;

/*  Create procedures to enqueue into single-consumer queues: */
create or replace procedure get_rushtitles(consumer in varchar2) as

deq_cust_data          BOLADM.customer_typ;
deq_book_data          BOLADM.book_typ;
deq_item_data          BOLADM.orderitem_typ;
deq_msgid              RAW(16);
dopt                   dbms_aq.dequeue_options_t;
mprop                  dbms_aq.message_properties_t;
deq_order_data         BOLADM.order_typ;
qname                  varchar2(30);
```

```
no_messages              exception;
pragma exception_init    (no_messages, -25228);
new_orders               BOOLEAN := TRUE;

begin

        dopt.consumer_name := consumer;
        dopt.wait := 1;
        dopt.correlation := 'RUSH';

        IF (consumer = 'West_Shipping') THEN
                qname := 'WS.WS_bookedorders_que';
        ELSIF (consumer = 'East_Shipping') THEN
                qname := 'ES.ES_bookedorders_que';
        ELSE
                qname := 'OS.OS_bookedorders_que';
        END IF;

        WHILE (new_orders) LOOP
          BEGIN
            dbms_aq.dequeue(
                queue_name => qname,
                dequeue_options => dopt,
                message_properties => mprop,
                payload => deq_order_data,
                msgid => deq_msgid);
            commit;

            deq_item_data := deq_order_data.items(1);
            deq_book_data := deq_item_data.item;

            dbms_output.put_line(' rushorder book_title: ' ||
                                    deq_book_data.title ||
                            ' quantity: ' || deq_item_data.quantity);
          EXCEPTION
            WHEN no_messages THEN
                 dbms_output.put_line (' ---- NO MORE RUSH TITLES ---- ');
                 new_orders := FALSE;
          END;
        END LOOP;

end;
/

CONNECT EXECUTE on get_rushtitles to ES;
```

```
/* Dequeue the orders: */
CONNECT ES/ES;

/*  Dequeue all rush order titles for East_Shipping: */
EXECUTE BOLADM.get_rushtitles('East_Shipping');
```

Multiple Recipients

A consumer can dequeue a message from a multi-consumer normal queue by supplying the name that was used in the `AQ$_AGENT` type of the `DBMS_AQADM.ADD_SUBSCRIBER` procedure or the recipient list of the message properties (see "Add a Subscriber" on page 4-46 or Enqueue a Message [Specify Message Properties] on page 6-9).

- In PL/SQL the consumer name is supplied using the `consumer_name` field of the `dequeue_options_t` record.
- In OCI the consumer name is supplied using the `OCISetAttr` procedure to specify a text string as the `OCI_ATTR_CONSUMER_NAME` of an `OCI_DTYPE_AQDEQ_OPTIONS` descriptor.

There can be multiple processes or operating system threads that use the same `consumer_name` to dequeue concurrently from a queue. In that case AQ will provide the first unlocked message that is at the head of the queue and is intended for the consumer. Unless the message ID of a specific message is specified during dequeue, the consumers can dequeue messages that are in the `READY` state.

A message is considered `PROCESSED` only when all intended consumers have successfully dequeued the message. A message is considered `EXPIRED` if one or more consumers did not dequeue the message before the `EXPIRATION` time. When a message has expired, it is moved to an exception queue.

The exception queue must also be a multi-consumer queue. Expired messages from multi-consumer queues cannot be dequeued the intended recipients of the message. However, they can be dequeued in the `REMOVE` mode exactly once by specifying a `NULL` consumer name in the dequeue options. Hence, from a dequeue perspective, multi-consumer exception queues behave like single-consumer queues because each expired message can be dequeued only once using a `NULL` consumer name. Note that expired messages can be dequeued only by specifying a message ID if the multi-consumer exception queue was created in a queue table without the compatible parameter or with the compatible parameter set to '8.0'.

In release 8.0.x when two or more processes/threads that are using different `consumer_names` are dequeuing from a queue, only one process/thread can dequeue a given message in the `LOCKED` or `REMOVE` mode at any time. What this means is that other consumers that need to the dequeue the same message will have to wait until the consumer that has locked the message commits or aborts the transaction and releases the lock on the message. However, while release 8.0.x did not support concurrency among different consumers for the same message., with release 8.1.x all consumers can access the same message concurrently. The result is that two processes/threads that are using different consumer_name to dequeue the same message do not block each other. AQ achieves this improvement by decoupling the task of dequeuing a message and the process of removing the message from the queue. In release 8.1.x only the queue monitor removes messages from multi-consumer queues. This allows dequeuers to complete the dequeue operation by not locking the message in the queue table. Since the queue monitor performs the task of removing messages that have been processed by all consumers from multi-consumer queues approximately once every minute, users may see a delay when the messages have been completely processed and when they are physically removed from the queue.

Local and Remote Recipients

Consumers of a message in multi-consumer queues (either by virtue of being a subscriber to the queue or because the consumer was a recipient in the enqueuer's recipient list) can be local or remote.

- A local consumer dequeues the message from the same queue into which the producer enqueued the message. Local consumers have a non-`NULL` `NAME` and a `NULL` `ADDRESS` and `PROTOCOL` field in the `AQ$_AGENT` type (see "Agent" on page 3-5 in Chapter 3, "Managing Oracle AQ").
- A Remote consumer dequeues from a queue that is different (but has the same payload type as the source queue) from the queue in which the message was enqueued. As such, users need to be familiar with and use the AQ Propagation feature to use remote consumers. Remote consumers can fall into one of three categories:

   a. The `ADDRESS` field refers to a queue in the same database. In this case the consumer will dequeue the message from a different queue in the same database. These addresses will be of the form `[schema].queue_name` where `queue_name` (optionally qualified by the schema name) is the target queue. If the schema is not specified, the schema of the current user executing the `ADD_SUBSCRIBER` procedure or the enqueue is used (see "Add a Subscriber" on page 4-46, or "Enqueue a Message" on page 6-4 in Chapter 6, "Operational Interface: Basic Operations"). Use the `DBMS_AQADM.SCHEDULE_PROPAGATION` command with a `NULL` destination (which is the default) to schedule propagation to such remote consumers (see "Schedule a Queue Propagation" on page 4-56 in Chapter 4, "Administrative Interface: Basic Operations").

   b. The `ADDRESS` field refers to a queue in a different database. In this case the database must be reachable using database links and the `PROTOCOL` must be either `NULL` or 0. These addresses will be of the form `[schema].queue_name@dblink`. If the schema is not specified, the schema of the current user executing the `ADD_SUBSCRIBER` procedure or the enqueue is used. If the database link is not a fully qualified name (does not have a domain name specified) the default domain as specified by the `db_domain init.ora` parameter will be used. Use the `DBMS_AQADM.SCHEDULE_PROPAGATION` procedure with the database link as the destination to schedule the propagation. AQ does not support the use of synonyms to refer to queues or database links.

   c. The `ADDRESS` field refers to a destination that can be reached by a third party protocol. You will need to refer to the documentation of the third party software to determine how to specify the `ADDRESS` and the `PROTOCOL` database link, and on how to schedule propagation.

When a consumer is remote, a message will be marked as `PROCESSED` in the source queue immediately after the message has been propagated even though the consumer may not have dequeued the message at the remote queue. Similarly, when a propagated message expires at the remote queue, the message is moved to the `DEFAULT` exception queue of the remote queue's queue table, and not to the exception queue of the local queue. As can be seen in both cases, AQ does not currently propagate the exceptions to the source queue. You can use the `MSGID` and the `ORIGINAL_MSGID` columns in the queue table view (`AQ$<queue_table>`) to chain the propagated messages. When a message with message ID m1 is propagated to a remote queue, m1 is stored in the `ORIGINAL_MSGID` column of the remote queue.

The `DELAY`, `EXPIRATION` and `PRIORITY` parameters apply identically to both local and remote consumers. AQ accounts for any delay in propagation by adjusting the `DELAY` and `EXPIRATION` parameters accordingly. For example, if the `EXPIRATION` is set to one hour, and the message is propagated after 15 minutes, the expiration at the remote queue will be set to 45 minutes.

Message Navigation in Dequeue

You have several options for selecting a message from a queue. You can select the ’first message’. Alternatively, once you have selected a message and established its position in the queue (for example, as the fourth message), you can then retrieve the ’next message’.

These selections work in a slightly different way if the queue is enabled for transactional grouping.

- If the ’first message’ is requested then the dequeue position is reset to the beginning of the queue.
- If the ’next message’ is requested then the position is set to the next message of the same transaction
- If the ’next transaction’ is requested then the position is set to the first message of the next transaction.

Note that the transaction grouping property is negated if a dequeue is performed in one of the following ways: dequeue by specifying a correlation identifier, dequeue by specifying a message identifier, or dequeueing some of the messages of a transaction and committing. For additional information on dequeueing by specifying a correlation identifier or a message identifier please refer to the section on dequeue methods.

If in navigating through the queue, the program reaches the end of the queue while using the ’next message’ or’ next transaction’ option, and you have specified a non-zero wait time, then the navigating position is automatically changed to the beginning of the queue.

Example Scenario

The following scenario in the `BooksOnLine` example continues the message grouping example already discussed with regard to enqueuing (see "Dequeue Methods" on page 2-47).

The `get_orders`() procedure dequeues orders from the `OE_neworders_que`. Recall that each transaction refers to an order and each message corresponds to an individual book in the order. The `get_orders`() procedure loops through the messages to dequeue the book orders. It resets the position to the beginning of the queue using the first message option before the first dequeues. It then uses the next message navigation option to retrieve the next book (message) of an order (transaction). If it gets an error message indicating all message in the current group/transaction have been fetched, it changes the navigation option to next transaction and get the first book of the next order. It then changes the navigation option back to next message for fetching subsequent messages in the same transaction. This is repeated until all orders (transactions) have been fetched.

Example Code

```
CONNECT boladm/boladm;

create or replace procedure get_new_orders as

deq_cust_data           BOLADM.customer_typ;
deq_book_data           BOLADM.book_typ;
deq_item_data           BOLADM.orderitem_typ;
deq_msgid               RAW(16);
dopt                    dbms_aq.dequeue_options_t;
mprop                   dbms_aq.message_properties_t;
deq_order_data          BOLADM.order_typ;
qname                   VARCHAR2(30);
no_messages             exception;
end_of_group            exception;
pragma exception_init   (no_messages, -25228);
pragma exception_init   (end_of_group, -25235);
new_orders              BOOLEAN := TRUE;

begin

        dopt.wait := 1;
        dopt.navigation := DBMS_AQ.FIRST_MESSAGE;
        qname := 'OE.OE_neworders_que';
        WHILE (new_orders) LOOP
          BEGIN
            LOOP
                BEGIN
                    dbms_aq.dequeue(
                        queue_name          => qname,
                        dequeue_options     => dopt,
                        message_properties  => mprop,
                        payload             => deq_order_data,
                        msgid               => deq_msgid);

                    deq_item_data := deq_order_data.items(1);
                    deq_book_data := deq_item_data.item;
                    deq_cust_data := deq_order_data.customer;

                    IF (deq_cust_data IS NOT NULL) THEN
                      dbms_output.put_line(' **** NEXT ORDER **** ');
```

```
                      dbms_output.put_line(’order_num: ’ ||
                                deq_order_data.orderno);
                      dbms_output.put_line(’ship_state: ’ ||
                                deq_cust_data.state);
                    END IF;
                    dbms_output.put_line(’ ---- next book ---- ’);
                    dbms_output.put_line(’ book_title: ’ ||
                                deq_book_data.title ||
                                ’ quantity: ’ || deq_item_data.quantity);
                EXCEPTION
                    WHEN end_of_group THEN
                      dbms_output.put_line (’*** END OF ORDER ***’);
                      commit;
                      dopt.navigation := DBMS_AQ.NEXT_TRANSACTION;
                END;
            END LOOP;
          EXCEPTION
            WHEN no_messages THEN
                 dbms_output.put_line (’ ---- NO MORE NEW ORDERS ---- ’);
                 new_orders := FALSE;
          END;
        END LOOP;

end;
/

CONNECT EXECUTE ON get_new_orders to OE;

/*  Dequeue the orders: */
CONNECT OE/OE;
EXECUTE BOLADM.get_new_orders;
```

Modes of Dequeuing

A dequeue request can either view a message or delete a message (see "Dequeue a Message" on page 6-38 in Chapter 6, "Operational Interface: Basic Operations").

- To view a message you can use either the ’browse’ mode or ’locked’ mode.
- To delete a message you can use either the ’remove’ mode or ’remove with no data’ mode.

If a message is browsed it remains available for further processing. Similarly if a message is locked it remains available for further processing once the lock on it is released by performing a transaction commit or rollback. Once a message is deleted using either of the remove modes, it is no longer available for dequeue requests.

When a message is dequeued using `REMOVE_NODATA` mode, the payload of the message is not retrieved. This mode can be useful when the user has already examined the message payload, possibly by means of a previous `BROWSE` dequeue. In this way, you can avoid the overhead of payload retrieval which can be substantial for large payloads

A message is retained in the queue table after it has been removed only if a retention time is specified for a queue. Messages cannot be retained in exception queues (refer to the section on exceptions for further information). Removing a message with no data is generally used if the payload is known (from a previous browse/locked mode dequeue call), or the message will not be used.

Note that after a message has been browsed there is no guarantee that the message can be dequeued again since a dequeue call from a concurrent user might have removed the message. To prevent a viewed message from being dequeued by a concurrent user, you should view the message in the locked mode.

You need to take special care while using the browse mode for other reasons as well. The dequeue position is automatically changed to the beginning of the queue if a non-zero wait time is specified and the navigating position reaches the end of the queue. Hence repeating a dequeue call in the browse mode with the ’next message’ navigation option and a non-zero wait time can dequeue the same message over and over again. We recommend that you use a non-zero wait time for the first dequeue call on a queue in a session, and then use a zero wait time with the next message navigation option for subsequent dequeue calls. If a dequeue call gets an ’end of queue’ error message, the dequeue position can be explicitly set by the dequeue call to the beginning of the queue using the ’first message’ navigation option, following which the messages in the queue can be browsed again.

Example Scenario

In the following scenario from the `BooksOnLine` example, international orders destined to Mexico and Canada are to be processed separately due to trade policies and carrier discounts. Hence, a message is viewed in the locked mode (so no other concurrent user removes the message) and the customer country (message payload) is checked. If the customer country is Mexico or Canada the message be deleted from the queue using the remove with no data (since the payload is already known) mode. Otherwise, the lock on the message is released by the commit call. Note that the remove dequeue call uses the message identifier obtained from the locked mode dequeue call. The `shipping_bookedorder_deq` (refer to the example code for the description of this procedure) call illustrates the use of the browse mode.

Example Code

```
CONNECT boladm/boladm;

create or replace procedure get_northamerican_orders as

deq_cust_data            BOLADM.customer_typ;
deq_book_data            BOLADM.book_typ;
deq_item_data            BOLADM.orderitem_typ;
deq_msgid                RAW(16);
dopt                     dbms_aq.dequeue_options_t;
mprop                    dbms_aq.message_properties_t;
deq_order_data           BOLADM.order_typ;
deq_order_nodata         BOLADM.order_typ;
qname                    VARCHAR2(30);
no_messages              exception;
pragma exception_init    (no_messages, -25228);
new_orders               BOOLEAN := TRUE;

begin

        dopt.consumer_name := consumer;
        dopt.wait := DBMS_AQ.NO_WAIT;
        dopt.navigation := dbms_aq.FIRST_MESSAGE;
        dopt.dequeue_mode := DBMS_AQ.LOCKED;

        qname := 'OS.OS_bookedorders_que';

        WHILE (new_orders) LOOP
          BEGIN
            dbms_aq.dequeue(
                queue_name => qname,
```

```
                    dequeue_options => dopt,
                    message_properties => mprop,
                    payload => deq_order_data,
                    msgid => deq_msgid);

                deq_item_data := deq_order_data.items(1);
                deq_book_data := deq_item_data.item;
                deq_cust_data := deq_order_data.customer;

                IF (deq_cust_data.country = 'Canada' OR
                    deq_cust_data.country = 'Mexico' ) THEN

                    dopt.dequeue_mode := dbms_aq.REMOVE_NODATA;
                    dopt.msgid := deq_msgid;
                    dbms_aq.dequeue(
                            queue_name => qname,
                            dequeue_options => dopt,
                            message_properties => mprop,
                            payload => deq_order_nodata,
                            msgid => deq_msgid);
                    commit;

                    dbms_output.put_line(' **** next booked order **** ');
                    dbms_output.put_line('order_no: ' || deq_order_data.orderno ||
                            ' book_title: ' || deq_book_data.title ||
                            ' quantity: ' || deq_item_data.quantity);
                    dbms_output.put_line('ship_state: ' || deq_cust_data.state ||
                            ' ship_country: ' || deq_cust_data.country ||
                            ' ship_order_type: ' || deq_order_data.ordertype);

                END IF;

                commit;
                dopt.dequeue_mode := DBMS_AQ.LOCKED;
                dopt.msgid := NULL;
                dopt.navigation := dbms_aq.NEXT_MESSAGE;
              EXCEPTION
                WHEN no_messages THEN
                     dbms_output.put_line (' ---- NO MORE BOOKED ORDERS ---- ');
                     new_orders := FALSE;
              END;
            END LOOP;

end;
/
```

```
CONNECT EXECUTE on get_northamerican_orders to OS;

CONNECT ES/ES;

/*  Browse all booked orders for East_Shipping: */
EXECUTE BOLADM.shipping_bookedorder_deq('East_Shipping', DBMS_AQ.BROWSE);

CONNECT OS/OS;

/*  Dequeue all international North American orders for Overseas_Shipping: */
EXECUTE BOLADM.get_northamerican_orders;
```

Optimization of Waiting for Arrival of Messages

One of the most important features of AQ is that it allows applications to block on one or more queues waiting for the arrival of either a newly enqueued message or for a message that becomes ready. You can use the `DEQUEUE` operation to wait for arrival of a message in a queue (see "Dequeue a Message" on page 6-38) or the `LISTEN` operation to wait for the arrival of a message in more than one queue (see "Listen to One (Many) Queue(s)" on page 6-18 in Chapter 6, "Operational Interface: Basic Operations").

When the blocking `DEQUEUE` call returns, it returns the message properties and the message payload. By contrast, when the blocking `LISTEN` call returns, it discloses only the name of the queue in which a message has arrived. A subsequent `DEQUEUE` operation is needed to dequeue the message.

Applications can optionally specify a timeout of zero or more seconds to indicate the time that AQ must wait for the arrival of a message. The default is to wait forever until a message arrives in the queue. This optimization is important in two ways. It removes the burden of continually polling for messages from the application. And it saves CPU and network resource because the application remains blocked until a new message is enqueued or becomes `READY` after its `DELAY` time. In release 8.1.5 applications can also perform a blocking dequeue on exception queues to wait for arrival of `EXPIRED` messages.

A process or thread that is blocked on a dequeue is either woken up directly by the enqueuer if the new message has no `DELAY` or is woken up by the queue monitor process when the `DELAY` or `EXPIRATION` time has passed. Applications can not only wait for the arrival of a message in the queue that an enqueuer enqueues a message, but also on a remote queue, provided that propagation has been schedule to the remote queue using `DBMS_AQADM.SCHEDULE_PROPAGATION`. In this case the AQ propagator will wake-up the blocked dequeuer after a message has been propagated.

Example Scenario

In the BooksOnLine example, the `get_rushtitles` procedure discussed under dequeue methods specifies a wait time of 1 second in the `dequeue_options` argument for the dequeue call. Wait time can be specified in different ways as illustrated in the code below.

- If the wait time is specified as 10 seconds, the dequeue call is blocked with a timeout of 10 seconds until a message is available in the queue. This means that if there are no messages in the queue after 10 seconds, the dequeue call returns without a message. Predefined constants can also be assigned for the wait time.

- If the wait time is specified as `DBMS_AQ.NO_WAIT`, a wait time of 0 seconds is implemented. The dequeue call in this case will return immediately even if there are no messages in the queue.
- If the wait time is specified as `DBMS_AQ.FOREVER`, the dequeue call is blocked without a timeout until a message is available in the queue.

Example Code

```
/* dopt is a variable of type dbms_aq.dequeue_options_t.
   Set the dequeue wait time to 10 seconds: */
dopt.wait := 10;

/* Set the dequeue wait time to 0 seconds: */
dopt.wait := DBMS_AQ.NO_WAIT;

/* Set the dequeue wait time to infinite (forever): */
dopt.wait := DBMS_AQ.FOREVER;
```

Retry with Delay Interval

AQ supports delay delivery of messages by letting the enqueuer specify a delay interval on a message when enqueueing the message, that is, the time before which a message cannot be retrieved by a dequeue call. (see "Enqueue a Message [Specify Message Properties]" on page 6-9 in Chapter 6, "Operational Interface: Basic Operations"). The delay interval determines when an enqueued message is marked as available to the dequeuers after message is enqueued. The producer can also specify the time when a message expires, at which time the message is moved to an exception queue.

When a message is enqueued with a delay time set, the message is marked as in `WAIT` state. Messages in `WAIT` state are masked from the default dequeue calls.

A background time-manager daemon wakes up periodically, scans an internal index for all `WAIT` state messages, and marks messages as `READY` if their delay time has passed. The time-manager will then post to all foreground processes that are waiting on queues in which messages have just been made available.

Example Scenario

An order is placed in a back order queue at a specific shipping region if the order cannot be filled immediately. To avoid repeatedly processing an unfilled order, all unfilled orders are enqueued into the `backorder` queue with a delay time of 1 day. The shipping application will attempt to ship a backorder by dequeuing an order from the backorder queue. If the order cannot be filled, it will re-enqueue the order into the same backorder queue with delay interval of the order set to 1 day.

The following procedure re-enqueues an unfilled order. It demonstrate enqueuing a backorder with delay time set to 1 day. This guarantees that each `backorder` will be processed only once a day until the order is filled.

Example Code

```
/*  Create a package that enqueue with delay set to one day: /*
CONNECT BOLADM/BOLADM
CREATE OR REPLACE PROCEDURE requeue_unfilled_order(sale_region varchar2,
                                                   backorder order_typ)
AS
  back_order_queue_name    VARCHAR2(62);
  enqopt                   dbms_aq.enqueue_options_t;
  msgprop                  dbms_aq.message_properties_t;
  enq_msgid                RAW(16);
BEGIN
  /* Choose a back order queue based the the region: */
```

```
  IF sale_region = 'WEST' THEN
    back_order_queue_name := 'WS.WS_backorders_que';
  ELSIF sale_region = 'EAST' THEN
    back_order_queue_name := 'ES.ES_backorders_que';
  ELSE
    back_order_queue_name := 'OS.OS_backorders_que';
  END IF;

  /* Enqueue the order with delay time set to 1 day: */
  msgprop.delay := 60*60*24;
  dbms_aq.enqueue(back_order_queue_name, enqopt, msgprop,
                  backorder, enq_msgid);
END;
/
```

Exception Handling

AQ provides four integrated mechanisms to support exception handling in applications: `EXCEPTION_QUEUES`, `EXPIRATION`, `MAX_RETRIES` and `RETRY_DELAY`.

An `exception_queue` is a repository for all expired or unserviceable messages. Applications cannot directly enqueue into exception queues. Also, a multi-consumer exception queue cannot have subscribers associated with it. However, an application that intends to handle these expired or unserviceable messages can dequeue from the exception queue. The exception queue created for messages intended for a multi-consumer queue must itself be a multi-consumer queue. Like any other queue, the exception queue must be enabled for dequeue using the `DBMS_AQADM.START_QUEUE` procedure. You will get an Oracle error if you try to enable an exception queue for enqueue.

When a message has expired, it is moved to an exception queue. The exception queue for a message in multi-consumer queue must also be a multi-consumer queue. Expired messages from multi-consumer queues cannot be dequeued by the intended recipients of the message. However, they can be dequeued in the `REMOVE` mode exactly once by specifying a `NULL` consumer name in the dequeue options. Hence, from a dequeue perspective multi-consumer exception queues behave like single-consumer queues because each expired message can be dequeued only once using a `NULL` consumer name. Messages can also be dequeued from the exception queue by specifying the message ID. Note that expired messages can be dequeued only by specifying a message ID if the multi-consumer exception queue was created in a queue table without the compatible parameter or with the compatible parameter set to '8.0'.

The exception queue is a message property that can be specified during enqueue time (see "Enqueue a Message [Specify Message Properties]" on page 6-9 in Chapter 6, "Operational Interface: Basic Operations"). In PL/SQL users can use the `exception_queue` attribute of the `DBMS_AQ.MESSAGE_PROPERTIES_T` record to specify the exception queue. In OCI users can use the `OCISetAttr` procedure to set the `OCI_ATTR_EXCEPTION_QUEUE` attribute of the `OCIAQMsgProperties` descriptor.

If an exception queue is not specified, the default exception queue is used. If the queue is created in a queue table, say `QTAB`, the default exception queue will be called AQ$_QTAB_E. The default exception queue is automatically created when the queue table is created. Messages are moved to the exception queues by AQ under the following conditions.

- The message is not being dequeued within the specified expiration interval. For messages intended for more than one recipient, the message will be moved to the exception queue if one or more of the intended recipients was not able to dequeue the message within the specified expiration interval. The default expiration interval is `DBMS_AQ.NEVER`, which means the messages will not expire.
- The message is being dequeued successfully. However, because of an error that arises while processing the message, the application which dequeues the message chooses to roll back the transaction. In this case, the message is returned to the queue and will be available for any applications that are waiting to dequeue from the same queue. A dequeue is considered rolled back or undone if the application rolls back the entire transaction, or if it rolls back to a savepoint that was taken before the dequeue. If the message has been dequeued but rolled back more than the number of time specified by the retry limit, the message will be moved to the exception queue.

  For messages intended for multiple recipients, each message keeps a separate retry count for each recipient. The message is moved to the exception queue only when retry counts for all recipients of the message have exceeded the specified retry limit. The default retry limit is 5 for single consumer queues and 8.1-compatible multiconsumer queues. No retry limit is not supported for 8.0-compatible multi-consumer queues.
- The statement executed by the client contains a dequeue that succeeded but the statement itself was undone later due to an exception. To understand this case, consider a PL/SQL procedure that contains a call to `DBMS_AQ.DEQUEUE`. If the dequeue procedure succeeds but the PL/SQL procedure raises an exception, AQ will attempt to increment the `RETRY_COUNT` of the message returned by the dequeue procedure.
- The client program successfully dequeued a message but terminated before committing the transaction.

Messages intended for 8.1-compatible multiconsumer queues cannot be dequeued by the intended recipients once the messages have been moved to an exception queue. These messages should instead be dequeued in the `REMOVE` or `BROWSE` mode exactly once by specifying a `NULL` consumer name in the dequeue options. The messages can also be dequeued by their message IDs.

Messages intended for single consumer queues, or for 8.0-compatible multi-consumer queues, can only be dequeued by their message IDs once the messages have been moved to an exception queue.

Users can associate a `RETRY_DELAY` with a queue. The default value for this parameter is 0 which means that the message will be available for dequeue immediately after the `RETRY_COUNT` is incremented. Otherwise the message will be unavailable for `RETRY_DELAY` seconds. After `RETRY_DELAY` seconds the queue monitor will mark the message as `READY`.

Example Scenario

In the `BooksOnLine` application, the business rule for each shipping region is that an order will be placed in a back order queue if the order cannot be filled immediately. The back order application will try to fill the order once a day. If the order cannot be filled within 5 days, it is placed in an exception queue for special processing. You can implement this process by making use of the retry and exception handling features in AQ.

The example below shows how you can create a queue with specific maximum retry and retry delay interval.

Example Code

```
/* Example for creating a back order queue in Western Region which allows a
   maximum of 5 retries and 1 day delay between each retry. */
CONNECT BOLADM/BOLADM
BEGIN
  dbms_aqadm.create_queue (
        queue_name              => 'WS.WS_backorders_que',
        queue_table             => 'WS.WS_orders_mqtab',
        max_retries             => 5,
        retry_delay             => 60*60*24);
END;
/

/* Create an exception queue for the back order queue for Western Region. */
CONNECT BOLADM/BOLADM
BEGIN
  dbms_aqadm.create_queue (
        queue_name              => 'WS.WS_backorders_excpt_que',
        queue_table             => 'WS.WS_orders_mqtab',
        queue_type              => DBMS_AQADM.EXCEPTION_QUEUE);
end;
/

/* Enqueue a message to WS_backorders_que and specify WS_backorders_excpt_que as
the exception queue for the message: */
CONNECT BOLADM/BOLADM
```

```
CREATE OR REPLACE PROCEDURE enqueue_WS_unfilled_order(backorder order_typ)
 AS
   back_order_queue_name    varchar2(62);
   enqopt                   dbms_aq.enqueue_options_t;
   msgprop                  dbms_aq.message_properties_t;
   enq_msgid                raw(16);
 BEGIN

   /* Set back order queue name for this message: */
   back_order_queue_name := 'WS.WS_backorders_que';

   /* Set exception queue name for this message: */
   msgprop.exception_queue := 'WS.WS_backorders_excpt_que';

   dbms_aq.enqueue(back_order_queue_name, enqopt, msgprop,
                   backorder, enq_msgid);
 END;
 /
```

Rule-based Subscription

Messages may be routed to various recipients based on message properties or message content. Users define a rule-based subscription for a given queue to specify interest in receiving messages that meet particular conditions.

Rules are boolean expressions that evaluate to `TRUE` or `FALSE`. Similar in syntax to the `WHERE` clause of a SQL query, rules are expressed in terms of the attributes that represent message properties or message content. These subscriber rules are evaluated against incoming messages and those rules that match are used to determine message recipients. This feature thus supports the notions of content-based subscriptions and content-based routing of messages.

Example Scenario and Code

For the `BooksOnLine` application, we illustrate how rule-based subscriptions are used to implement a publish/subscribe paradigm utilizing content-based subscription and content-based routing of messages. The interaction between the Order Entry application and each of the Shipping Applications is modeled as follows;

- Western Region Shipping handles orders for the Western region of the US.
- Eastern Region Shipping handles orders for the Eastern region of the US.
- Overseas Shipping handles all non-US orders.
- Eastern Region Shipping also handles all US rush orders.

Each shipping application subscribes to the OE booked orders queue. The following rule-based subscriptions are defined by the Order Entry user to handle the routing of booked orders from the Order Entry application to each of the Shipping applications.

```
CONNECT OE/OE;
```

Western Region Shipping defines an agent called '`West_Shipping`' with the `WS` booked orders queue as the agent address (destination queue to which messages must be delivered). This agent subscribes to the OE booked orders queue using a rule specified on order region and `ordertype` attributes.

```
/*  Add a rule-based subscriber for West Shipping -
    West Shipping handles Western region US orders,
    Rush Western region orders are handled by East Shipping: */
DECLARE
  subscriber     aq$_agent;
BEGIN
```

```
  subscriber := aq$_agent('West_Shipping', 'WS.WS_bookedorders_que', null);
  dbms_aqadm.add_subscriber(
               queue_name => 'OE.OE_bookedorders_que',
               subscriber => subscriber,
               rule       => 'tab.user_data.orderregion =
                   ''WESTERN'' AND tab.user_data.ordertype != ''RUSH''');
END;
/
```

Eastern Region Shipping defines an agent called `East_Shipping` with the `ES` booked orders queue as the agent address (the destination queue to which messages must be delivered). This agent subscribes to the `OE` booked orders queue using a rule specified on `orderregion`, `ordertype` and customer attributes.

```
/*  Add a rule-based subscriber for East Shipping -
    East shipping handles all Eastern region orders,
    East shipping also handles all US rush orders: */
DECLARE
  subscriber     aq$_agent;
BEGIN
  subscriber := aq$_agent('East_Shipping', 'ES.ES_bookedorders_que', null);
  dbms_aqadm.add_subscriber(
        queue_name => 'OE.OE_bookedorders_que',
        subscriber => subscriber,
        rule       => 'tab.user_data.orderregion = ''EASTERN'' OR
                       (tab.user_data.ordertype = ''RUSH'' AND
                        tab.user_data.customer.country = ''USA'') ');
END;
/
```

Overseas Shipping defines an agent called `Overseas_Shipping` with the `OS` booked orders queue as the agent address (destination queue to which messages must be delivered). This agent subscribes to the `OE` booked orders queue using a rule specified on `orderregion` attribute.

```
/*  Add a rule-based subscriber for Overseas Shipping
    Intl Shipping handles all non-US orders: */
DECLARE
  subscriber     aq$_agent;
BEGIN
  subscriber := aq$_agent('Overseas_Shipping', 'OS.OS_bookedorders_que',
null);
  dbms_aqadm.add_subscriber(
        queue_name => 'OE.OE_bookedorders_que',
        subscriber => subscriber,
```

```
        rule        => 'tab.user_data.orderregion = ''INTERNATIONAL''');
END;
/
```

Listen Capability

In Oracle8*i* release 8.1.x, AQ has the capability to monitor multiple queues for messages with a single call, `listen`. An application can use `listen` to wait for messages for multiple subscriptions. It can also be used by gateway applications to monitor multiple queues. If the `listen` call returns successfully, a dequeue must be used to retrieve the message (see Listen to One (Many) Queue(s) on page 6-18 in Chapter 6, "Operational Interface: Basic Operations").

Without the `listen` call, an application which sought to dequeue from a set of queues would have to continuously poll the queues to determine if there were a message. Alternatively, you could design your application to have a separate dequeue process for each queue. However, if there are long periods with no traffic in any of the queues, these approaches will create an unacceptable overhead. The `listen` call is well suited for such applications.

Note that when there are messages for multiple agents in the agent list, `listen` returns with the first agent for whom there is a message. In that sense `listen` is not 'fair' in monitoring the queues. The application designer must keep this in mind when using the call. To prevent one agent from 'starving' other agents for messages, the application could change the order of the agents in the agent list.

Example Scenario

In the customer service component of the `BooksOnLine` example, messages from different databases arrive in the customer service queues, indicating the state of the message. The customer service application monitors the queues and whenever there is a message about a customer order, it updates the order status in the `order_status_table`. The application uses the `listen` call to monitor the different queues. Whenever there is a message in any of the queues, it dequeues the message and updates the order status accordingly.

Example Code

```
CODE (in tkaqdocd.sql)

/* Update the status of the order in the order status table: */
CREATE OR REPLACE PROCEDURE update_status(
                                new_status    IN VARCHAR2,
                                order_msg     IN BOLADM.ORDER_TYP)
IS
 old_status    VARCHAR2(30);
 dummy         NUMBER;
BEGIN
```

```
  BEGIN
    /* Query old status from the table: */
    SELECT st.status INTO old_status FROM order_status_table st
       WHERE st.customer_order.orderno = order_msg.orderno;

  /* Status can be 'BOOKED_ORDER', 'SHIPPED_ORDER', 'BACK_ORDER'
     and   'BILLED_ORDER': */

   IF new_status = 'SHIPPED_ORDER' THEN
      IF old_status = 'BILLED_ORDER' THEN
        return;             /* message about a previous state */
      END IF;
   ELSIF new_status = 'BACK_ORDER' THEN
      IF old_status = 'SHIPPED_ORDER' OR old_status = 'BILLED_ORDER' THEN
        return;             /* message about a previous state */
      END IF;
   END IF;

   /* Update the order status:  */
     UPDATE order_status_table st
        SET st.customer_order = order_msg, st.status = new_status;

   COMMIT;

  EXCEPTION
  WHEN OTHERS  THEN     /* change to no data found */
    /* First update for the order: */
    INSERT INTO order_status_table(customer_order, status)
    VALUES (order_msg, new_status);
    COMMIT;

  END;
END;
/


/* Dequeues message from 'QUEUE' for 'CONSUMER': */
CREATE OR REPLACE PROCEDURE DEQUEUE_MESSAGE(
                         queue      IN   VARCHAR2,
                         consumer   IN   VARCHAR2,
                         message    OUT  BOLADM.order_typ)
IS

dopt                     dbms_aq.dequeue_options_t;
mprop                    dbms_aq.message_properties_t;
```

```
deq_msgid                RAW(16);
BEGIN
  dopt.dequeue_mode := dbms_aq.REMOVE;
  dopt.navigation := dbms_aq.FIRST_MESSAGE;
  dopt.consumer_name := consumer;

  dbms_aq.dequeue(
                queue_name => queue,
                dequeue_options => dopt,
                message_properties => mprop,
                payload => message,
                msgid => deq_msgid);
  commit;
END;
/

/* Monitor the queues in the customer service databse for 'time' seconds: */
 CREATE OR REPLACE PROCEDURE MONITOR_STATUS_QUEUE(time  IN  NUMBER)
IS
  agent_w_message   aq$_agent;
  agent_list        dbms_aq.agent_list_t;
  wait_time         INTEGER := 120;
  no_message        EXCEPTION;
  pragma EXCEPTION_INIT(no_message, -25254);
  order_msg         boladm.order_typ;
  new_status        VARCHAR2(30);
  monitor           BOOLEAN := TRUE;
  begin_time        NUMBER;
  end_time          NUMBER;
BEGIN

 begin_time :=  dbms_utility.get_time;
 WHILE (monitor)
 LOOP
 BEGIN

  /* Construct the waiters list: */
  agent_list(1) := aq$_agent('BILLED_ORDER', 'CS_billedorders_que', NULL);
  agent_list(1) := aq$_agent('SHIPPED_ORDER', 'CS_shippedorders_que',
NULL);
  agent_list(2) := aq$_agent('BACK_ORDER', 'CS_backorders_que', NULL);
  agent_list(3) := aq$_agent('Booked_ORDER', 'CS_bookedorders_que', NULL);

   /* Wait for order status messages: */
   dbms_aq.listen(agent_list, wait_time, agent_w_message);
```

```
   dbms_output.put_line('Agent' || agent_w_message.name || ' Address '||
agent_w_message.address);
   /* Dequeue the message from the queue: */
   dequeue_message(agent_w_message.address, agent_w_message.name, order_msg);

   /* Update the status of the order depending on the type of the message,
    * the name of the agent contains the new state: */
   update_status(agent_w_message.name, order_msg);

  /* Exit if we have been working long enough: */
   end_time := dbms_utility.get_time;
   IF  (end_time - begin_time > time)   THEN
     EXIT;
   END IF;

  EXCEPTION
  WHEN  no_message  THEN
    dbms_output.put_line('No messages in the past 2 minutes');
       end_time := dbms_utility.get_time;
    /* Exit if we have done enough work: */
    IF  (end_time - begin_time > time)   THEN
      EXIT;
    END IF;
  END;

  END LOOP;
END;
/
```

Propagation Features

- Propagation
- Propagation Scheduling
- Propagation of Messages with LOB Attributes
- Enhanced Propagation Scheduling Capabilities
- Exception Handling During Propagation

Propagation

This feature enables applications to communicate with each other without having to be connected to the same database, or to the same queue. Messages can be propagated from one Oracle AQ to another, irrespective of whether these are local or remote. Propagation is performed by snapshot (`job_queue`) background processes. Propagation to remote queues is done using database links, and Net 8.

The propagation feature is used as follows. First one or more subscribers are defined for the queue from which messages are to be propagated (see "Subscriptions and Recipient Lists" on page 2-29). Second, a schedule is defined for each destination to which messages are to be propagated from the queue. Enqueued messages will now be propagated and automatically be available for dequeuing at the destination queues.

Note that two or more number of `job_queue` background processes must be running to use propagation. This is in addition to the number of `job_queue` background processes needed for handling non-propagation related jobs. Also, if you wish to deploy remote propagation, you must ensure that the database link specified for the schedule is valid and have proper privileges for enqueuing into the destination queue. For more information about the administrative commands for managing propagation schedules, see "Asynchronous Notifications" below.

Propagation also has mechanisms for handling failure. For example, if the database link specified is invalid, or if the remote database is unavailable, or if the remote queue is not enabled for enqueuing, then the appropriate error message is reported.

Finally, propagation provides detailed statistics about the messages propagated and the schedule itself. This information can be used to properly tune the schedules for best performance. Failure handling/error reporting facilities of propagation and propagation statistics are discussed under "Enhanced Propagation Scheduling Capabilities".

Propagation Scheduling

A propagation schedule is defined for a pair of source and destination queues. If a queue has messages to be propagated to several queues then a schedule has to be defined for each of the destination queues. A schedule indicates the time frame during which messages can be propagated from the source queue. This time frame may depend on a number of factors such as network traffic, load at source database, load at destination database, and so on. The schedule therefore has to be tailored for the specific source and destination. When a schedule is created, a job is automatically submitted to the `job_queue` facility to handle propagation.

The administrative calls for propagation scheduling provide great flexibility for managing the schedules (see "Schedule a Queue Propagation" in Chapter 4, "Administrative Interface: Basic Operations"). The duration or propagation window parameter of a schedule specifies the time frame during which propagation has to take place. If the duration is unspecified then the time frame is an infinite single window. If a window has to be repeated periodically then a finite duration is specified along with a `next_time` function that defines the periodic interval between successive windows.

The latency parameter for a schedule is relevant only when a queue does not have any messages to be propagated. This parameter specifies the time interval within which a queue has to be rechecked for messages. Note that if the latency parameter is to be enforced, then the `job_queue_interval` parameter for the `job_queue_processes` should be less than or equal to the `latency` parameter.

The propagation schedules defined for a queue can be changed or dropped at anytime during the life of the queue. In addition there are calls for temporarily disabling a schedule (instead of dropping the schedule) and enabling a disabled schedule. A schedule is active when messages are being propagated in that schedule. All the administrative calls can be made irrespective of whether the schedule is active or not. If a schedule is active then it will take a few seconds for the calls to be executed.

Example Scenario

In the `BooksOnLine` example, messages in the `OE_bookedorders_que` are propagated to different shipping sites. The following example code illustrates the various administrative calls available for specifying and managing schedules. It also shows the calls for enqueuing messages into the source queue and for dequeuing the messages at the destination site). The catalog view `USER_QUEUE_SCHEDULES` provides all information relevant to a schedule (see "Select Propagation Schedules in User Schema" in Chapter 5, "Administrative Interface: Views").

Example Code

```
CONNECT OE/OE;

/* Schedule Propagation from bookedorders_que to shipping: */
EXECUTE dbms_aqadm.schedule_propagation( \
   queue_name      => 'OE.OE_bookedorders_que');

/* Check if a schedule has been created: */
SELECT * FROM user_queue_schedules;

/* Enqueue some orders into OE_bookedorders_que: */
EXECUTE BOLADM.order_enq('My First   Book', 1, 1001, 'CA', 'USA', \
   'WESTERN', 'NORMAL');
EXECUTE BOLADM.order_enq('My Second  Book', 2, 1002, 'NY', 'USA', \
   'EASTERN', 'NORMAL');
EXECUTE BOLADM.order_enq('My Third   Book', 3, 1003, '',   'Canada', \
   'INTERNATIONAL', 'NORMAL');
EXECUTE BOLADM.order_enq('My Fourth  Book', 4, 1004, 'NV', 'USA', \
   'WESTERN', 'RUSH');
EXECUTE BOLADM.order_enq('My Fifth   Book', 5, 1005, 'MA', 'USA', \
   'EASTERN', 'RUSH');
EXECUTE BOLADM.order_enq('My Sixth   Book', 6, 1006, ''  , 'UK', \
   'INTERNATIONAL', 'NORMAL');
EXECUTE BOLADM.order_enq('My Seventh Book', 7, 1007, '',   'Canada', \
   'INTERNATIONAL', 'RUSH');
EXECUTE BOLADM.order_enq('My Eighth  Book', 8, 1008, '',   'Mexico', \
   'INTERNATIONAL', 'NORMAL');
EXECUTE BOLADM.order_enq('My Ninth   Book', 9, 1009, 'CA', 'USA', \
   'WESTERN', 'RUSH');
EXECUTE BOLADM.order_enq('My Tenth   Book', 8, 1010, ''  , 'UK', \
   'INTERNATIONAL', 'NORMAL');
EXECUTE BOLADM.order_enq('My Last    Book', 7, 1011, ''  , 'Mexico', \
   'INTERNATIONAL', 'NORMAL');

/* Wait for propagation to happen: */
EXECUTE dbms_lock.sleep(100);

/* Connect to shipping sites and check propagated messages: */
CONNECT WS/WS;
set serveroutput on;

/*  Dequeue all booked orders for West_Shipping: */
EXECUTE BOLADM.shipping_bookedorder_deq('West_Shipping', DBMS_AQ.REMOVE);
```

```
CONNECT ES/ES;
SET SERVEROUTPUT ON;

/* Dequeue all remaining booked orders (normal order) for East_Shipping: */
EXECUTE BOLADM.shipping_bookedorder_deq('East_Shipping', DBMS_AQ.REMOVE);

CONNECT OS/OS;
SET SERVEROUTPUT ON;

/* Dequeue all international North American orders for Overseas_Shipping: */
EXECUTE BOLADM.get_northamerican_orders('Overseas_Shipping');

/* Dequeue rest of the booked orders for Overseas_Shipping: */
EXECUTE BOLADM.shipping_bookedorder_deq('Overseas_Shipping', DBMS_AQ.REMOVE);

/* Disable propagation schedule for booked orders
EXECUTE dbms_aqadm.disable_propagation_schedule(   \
   queue_name   => 'OE_bookedorders_que');

/* Wait for some time for call to be effected: */
EXECUTE dbms_lock.sleep(30);

/*  Check if the schedule has been disabled: */
SELECT schedule_disabled FROM user_queue_schedules;

/* Alter propagation schedule for booked orders to execute every
   15 mins (900 seconds) for a window duration of 300 seconds: */
EXECUTE dbms_aqadm.alter_propagation_schedule( \
   queue_name     => 'OE_bookedorders_que', \
   duration       => 300, \
   next_time      => 'SYSDATE + 900/86400',\
   latency        => 25);

/* Wait for some time for call to be effected: */
EXECUTE dbms_lock.sleep(30);

/*  Check if the schedule parameters have changed: */
SELECT next_time, latency, propagation_window FROM user_queue_schedules;

/* Enable propagation schedule for booked orders:
EXECUTE dbms_aqadm.enable_propagation_schedule( \
   queue_name     => 'OE_bookedorders_que');

/* Wait for some time for call to be effected: */
EXECUTE dbms_lock.sleep(30);
```

```
/* Check if the schedule has been enabled: */
SELECT schedule_disabled FROM user_queue_schedules;

/* Unschedule propagation for booked orders: */
EXECUTE dbms_aqadm.unschedule_propagation(   \
   queue_name      => 'OE.OE_bookedorders_que');

/* Wait for some time for call to be effected: */
EXECUTE dbms_lock.sleep(30);

/*  Check if the schedule has been dropped
SELECT *  FROM user_queue_schedules;
```

Propagation of Messages with LOB Attributes

Large Objects can be propagated using AQ using two methods:

- Propagation from `RAW` queues. In `RAW` queues the message payload is stored as a Binary Large Object (`BLOB`). This allows users to store up to 32KB of data when using the PL/SQL interface and as much data as can be contiguously allocated by the client when using OCI. This method is supported by all releases from 8.0.4 inclusive.
- Propagation from Object queues with `LOB` attributes. The user can populate the `LOB` and read from the `LOB` using Oracle's `LOB` handling routines. The `LOB` attributes can be `BLOB`s or `CLOB`s. If the attribute is a `CLOB` AQ will automatically perform any necessary characterset conversion between the source queue and the destination queue. This method is supported by all releases from 8.1.3 inclusive.

For more information about working with LOBs, see:

- *Oracle8i Application Developer's Guide - Large Objects (LOBs)*

Note that AQ does not support propagation from Object queues that have BFILE or REF attributes in the payload.

Example Scenario

In the BooksOnLine application, the company may wish to send promotional coupons along with the book orders. These coupons are generated depending on the content of the order, and other customer preferences. The coupons are images generated from some multimedia database, and are stored as `LOB`s.

When the order information is sent to the shipping warehouses, the coupon contents are also sent to the warehouses. In the code shown below the `order_typ` is enhanced to contain a coupon attribute of LOB type. The code demonstrates how the `LOB` contents are inserted into the message that is enqueued into `OE_bookedorders_que` when an order is placed. The message payload is first constructed with an empty `LOB`. The place holder (`LOB` locator) information is obtained from the queue table and is then used in conjunction with the `LOB` manipulation routines, such as DBMS_LOB.WRITE(), to fill the LOB contents. The example has additional examples regarding for enqueue and dequeue of messages with LOBs as part the payload.

A `COMMIT` is issued only after the LOB contents are filled in with the appropriate image data. Propagation automatically takes care of moving the LOB contents along with the rest of the message contents. The code below also shows a dequeue at the destination queue for reading the LOB contents from the propagated message. The LOB contents are read into a buffer that can be sent to a printer for printing the coupon.

Example Code

```
/* Enhance the type order_typ to contain coupon field (lob field): */
CREATE OR REPLACE TYPE order_typ AS OBJECT (
        orderno         NUMBER,
        status          VARCHAR2(30),
        ordertype       VARCHAR2(30),
        orderregion     VARCHAR2(30),
        customer        customer_typ,
        paymentmethod   VARCHAR2(30),
        items           orderitemlist_vartyp,
        total           NUMBER,
        coupon          BLOB);
/

/* lob_loc is a variable of type BLOB,
   buffer is a variable of type RAW,
   length is a variable of type NUMBER. */

/* Complete the order data and perform the enqueue using the order_enq()
   procedure: */
dbms_aq.enqueue('OE.OE_bookedorders_que', enqopt, msgprop,
                OE_enq_order_data, enq_msgid);

/* Get the lob locator in the queue table after enqueue: */
SELECT t.user_data.coupon INTO lob_loc
FROM   OE.OE_orders_pr_mqtab t
WHERE  t.msgid = enq_msgid;

/* Generate a sample LOB of 100 bytes: */
buffer := hextoraw(rpad('FF',100,'FF'));

/* Fill in the lob using LOB routines in the dbms_lob package: */
dbms_lob.write(lob_loc, 90, 1, buffer);

/* Issue a commit only after filling in lob contents: */
COMMIT;

/* Sleep until propagation is complete: */
```

```
/* Perform dequeue at the Western Shipping warehouse: */
dbms_aq.dequeue(
        queue_name         => qname,
        dequeue_options    => dopt,
        message_properties => mprop,
        payload            => deq_order_data,
        msgid              => deq_msgid);

/* Get the LOB locator after dequeue: */
lob_loc := deq_order_data.coupon;

/* Get the length of the LOB: */
length := dbms_lob.getlength(lob_loc);

/* Read the LOB contents into the buffer: */
dbms_lob.read(lob_loc, length, 1, buffer);
```

Enhanced Propagation Scheduling Capabilities

Detailed information about the schedules can be obtained from the catalog views defined for propagation. Information about active schedules —such as the name of the background process handling that schedule, the SID (session, serial number) for the session handling the propagation and the Oracle instance handling a schedule (relevant if OPS is being used) — can be obtained from the catalog views. The same catalog views also provide information about the previous successful execution of a schedule (last successful propagation of message) and the next execution of the schedule.

For each schedule detailed propagation statistics are maintained. This includes the total number of messages propagated in a schedule, total number of bytes propagated in a schedule, maximum number of messages propagated in a window, maximum number of bytes propagated in a window, average number of messages propagated in a window, average size of propagated messages and the average time to propagated a message. These statistics have been designed to provide useful information to the queue administrators for tuning the schedules such that maximum efficiency can be achieved.

Propagation has built in support for handling failures and reporting errors. For example, if the database link specified is invalid, the remote database is unavailable or if the remote queue is not enabled for enqueuing then the appropriate error message is reported. Propagation uses an exponential backoff scheme for retrying propagation from a schedule that encountered a failure. If a schedule continuously encounters failures, the first retry happens after 30 seconds, the second after 60 seconds, the third after 120 seconds and so forth. If the retry time is beyond the expiration time of the current window then the next retry is attempted at the start time of the next window. A maximum of 16 retry attempts are made after which the schedule is automatically disabled. When a schedule is disabled automatically due to failures, the relevant information is written into the alert log. At anytime it is possible to check if there were failures encountered by a schedule and if so how many successive failure were encountered, the error message indicating the cause for the failure and the time at which the last failure was encountered. By examining this information, a queue administrator can fix the failure and enable the schedule. During a retry if propagation is successful then the number of failures is reset to 0.

Propagation has support built in for OPS and is completely transparent to the user and the queue administrator. The job that handles propagation is submitted to the same instance as the owner of the queue table in which the queue resides. If at anytime there is a failure at an instance and the queue table that stores the queue is migrated to a different instance, the propagation job is also automatically migrated to the new instance. This will minimize the ’pinging’ between instances and thus offer better performance. Propagation has been designed to handle any number of concurrent schedules. Note that the number of job_queue_processes is limited to a maximum of 36 and some of these may be used to handle non-propagation related jobs. Hence, propagation has built is support for multi-tasking and load balancing. The propagation algorithms are designed such that multiple schedules can be handled by a single snapshot (job_queue) process. The propagation load on a job_queue processes can be skewed based on the arrival rate of messages in the different source queues. If one process is overburdened with several active schedules while another is underloaded with many passive schedules, propagation automatically re-distributes the schedules among the processes such that they are loaded uniformly.

Example Scenario

In the `BooksOnLine` example, the `OE_bookedorders_que` is a busy queue since messages in it are propagated to different shipping sites. The following example code illustrates the calls supported by enhanced propagation scheduling for error checking and schedule monitoring.

Example Code

```
CONNECT OE/OE;

/*  get averages
select avg_time, avg_number, avg_size from user_queue_schedules;

/*  get totals
select total_time, total_number, total_bytes from user_queue_schedules;

/*  get maximums for a window
select max_number, max_bytes from user_queue_schedules;

/*  get current status information of schedule
select process_name, session_id, instance, schedule_disabled
   from user_queue_schedules;

/*  get information about last and next execution
select last_run_date, last_run_time, next_run_date, next_run_time
   from user_queue_schedules;

/*  get last error information if any
select failures, last_error_msg, last_error_date, last_error_time
   from user_queue_schedules;
```

Exception Handling During Propagation

When a system errors such as a network failure occurs, AQ will continue to attempt to propagate messages using an exponential back-off algorithm. In some situations that indicate application errors AQ will mark messages as `UNDELIVERABLE` if there is an error in propagating the message.

Examples of such errors are when the remote queue does not exist or when there is a type mismatch between the source queue and the remote queue. In such situations users must query the `DBA_SCHEDULES` view to determine the last error that occurred during propagation to a particular destination.The trace files in the `$ORACLE_HOME/log` directory can provide additional information about the error.

Example Scenario

In the BooksOnLine example, the `ES_bookedorders_que` in the Eastern Shipping region is stopped intentionally using the stop_queue() call. After a short while the propagation schedule for `OE_bookedorders_que` will display an error indicating that the remote queue ES_bookedorders_que is disabled for enqueuing. When the `ES_bookedorders_que` is started using the `start_queue`() call, propagation to that queue resumes and there is no error message associated with schedule for `OE_bookedorders_que`.

Example Scenario

```
/*  Intentionally stop the eastern shipping queue : */
connect BOLADM/BOLADM
EXECUTE dbms_aqadm.stop_queue(queue_name => 'ES.ES_bookedorders_que');

/* Wait for some time before error shows up in dba_queue_schedules: */
EXECUTE dbms_lock.sleep(100);

/* This query will return an ORA-25207 enqueue failed error: */
SELECT qname, last_error_msg from dba_queue_schedules;

/* Start the eastern shipping queue: */
EXECUTE dbms_aqadm.start_queue(queue_name => 'ES.ES_bookedorders_que');

/* Wait for Propagation to resume for eastern shipping queue: */
EXECUTE dbms_lock.sleep(100);

/* This query will indicate that there are no errors with propagation:
SELECT qname, last_error_msg from dba_queue_schedules;
```

3

Managing Oracle AQ

This chapter describes the elements you need to work with and issues you will want to take into consideration in preparing the application environment.

INIT.ORA Parameter

AQ_TM_PROCESSES

You specify the parameter `aq_tm_processes` in the `init.ora` `PARAMETER` *file* if you want to perform time monitoring on queue messages. You can set the parameter in a range from 0 to 10 depending on how many queue monitor processes you require. Setting it to any other number will result in an error. If this parameter is set to 1, one queue monitor process will be created as a background process to monitor the messages. If the parameter is not specified, or is set to 0, the queue monitor process is not created.

Since the `aq_tm_processes` parameter is dynamic, you can alter the number of queue monitors while the instance is running. You do this by means of the syntax:

```
ALTER SYSTEM SET aq_tm_processes=<integer>;
```

Parameter Name:	`aq_tm_processes`
Parameter Type:	`integer`
Parameter Class:	`Dynamic`
Allowable Values:	`0 to 10`
Syntax:	`aq_tm_processes = <0 to 10>`
Name of process:	`ora_qmon_<oracle sid>`
Example:	`aq_tm_processes = 1`

JOB_QUEUE_PROCESSES

Propagation is handled by job queue (SNP) processes. The number of job queue processes started in an instance is controlled by the `init.ora` parameter `JOB_QUEUE_PROCESSES`. The default value of this parameter is 0. In order for message propagation to take place, this parameter must be set to at least 1. The DBA can set it to higher values if there are many queues from which the messages have to be propagated, or if there are many destinations to which the messages have to be propagated, or if there are other jobs in the job queue.

Note: with release 8.1.5 you need at least two job queue processes for propagation scheduling

See Also: *Oracle8 Reference* for complete details about `JOB_QUEUE_PROCESSES`.

Common Data Structures

The following data structures are used in both the operational and administrative interfaces:

- Chapter 4, "Administrative Interface: Basic Operations"
- Chapter 6, "Operational Interface: Basic Operations"

Object Name

Purpose:

The naming of database objects. This naming convention applies to queues, queue tables and object types.

Syntax:

```
object_name := VARCHAR2
object_name := [<schema_name>.]<name>
```

Usage:

Names for objects are specified by an optional schema name and a name. If the schema name is not specified then the current schema is assumed. The name must follow object name guidelines in the *Oracle8i SQL Reference* with regard to reserved characters.The schema name, agent name and the object type name can each be up to 30 bytes long. However, queue names and queue table names can be a maximum of 24 bytes.

Type name

Purpose:

Defining queue types.

Syntax:

```
type_name := VARCHAR2
type_name := <object_type> | "RAW"
```

Usage:

Table 3–1 Type Name

Parameter	Description
<object_types>	For details on creating object types please refer to Server concepts manual. The maximum number of attributes in the object type is limited to 900.
`"RAW"`	To store payload of type `RAW`, AQ will create a queue table with a `LOB` column as the payload repository. The size of the payload is limited to 32K bytes of data. Because `LOB` columns are used for storing `RAW` payload, the AQ administrator can choose the `LOB` tablespace and configure the `LOB` storage by constructing a `LOB` storage string in the storage_clause parameter during queue table creation time.

Agent

Purpose:

To identify a producer or a consumer of a message.

Syntax:

```
TYPE aq$_agent IS OBJECT (
   name           VARCHAR2(30),
   address        VARCHAR2(1024),
   protocol       NUMBER)
```

Usage:

Table 3–2 Agent

Parameter	Description
`name` `(VARCHAR2(30))`	Name of a producer or consumer of a message.The name must follow object name guidelines in the *Oracle8i SQL Reference* with regard to reserved characters.
`address` `(VARCHAR2(1024))`	Protocol specific address of the recipient. If the protocol is 0 (default) the address is of the form [schema.]queue[@dblink]
`protocol` `(NUMBER)`	Protocol to interpret the address and propagate the message. The default value is 0.

Usage Notes

All consumers that are added as subscribers to a multi-consumer queue must have unique values for the `AQ$_AGENT` parameter. This means that two subscribers cannot have the same values for the `NAME`, `ADDRESS` and `PROTOCOL` attributes for the `AQ$_AGENT` type. At least one of the three attributes must be different for two subscribers.

AQ Recipient List Type

Purpose:

To identify the list of agents that will receive the message.

Syntax:

```
TYPE aq$_recipient_list_t IS TABLE OF aq$_agent
           INDEX BY BINARY_INTEGER;
```

AQ Agent List Type

Purpose:

To identify the list of agents for DBMS_AQ.LISTEN to listen for.

Syntax:

```
TYPE aq$_agent_list_t IS TABLE OF aq$_agent
           INDEX BY BINARY INTEGER;
```

AQ Subscriber List Type

Purpose:

To identify the list of subscribers that subscribe to this queue.

Syntax:

```
TYPE aq$_subscriber_list_t IS TABLE OF aq$_agent
          INDEX BY BINARY INTEGER;
```

Enumerated Constants in the Administrative Interface

When using enumerated constants such as `INFINITE`, `TRANSACTIONAL`, `NORMAL_QUEUE` are selected as values, the symbol needs to be specified with the scope of the packages defining it. All types associated with the administrative interfaces have to be prepended with `dbms_aqadm`. For example:

```
DBMS_AQADM.NORMAL_QUEUE
```

Table 3–3 Enumerated types in the administrative interface

Parameter	**Options**
`retention`	`0,1,2...INFINITE`
`message_grouping`	`TRANSACTIONAL`, `NONE`
`queue_type`	`NORMAL_QUEUE`, `EXCEPTION_QUEUE,NON_PERSISTENT_QUEUE`

Enumerated Constants in the Operational Interface

When using enumerated constants such as `BROWSE`, `LOCKED`, `REMOVE`, the PL/SQL constants need to be specified with the scope of the packages defining it. All types associated with the operational interfaces have to be prepended with `dbms_aq`. For example:

```
DBMS_AQ.BROWSE
```

Table 3–4 Enumerated types in the operational interface

Parameter	Options
`visibility`	`IMMEDIATE`, `ON_COMMIT`
`dequeue mode`	`BROWSE`, `LOCKED`, `REMOVE`, `REMOVE_NODATA`
`navigation`	`FIRST_MESSAGE`, `NEXT_MESSAGE`, `NEXT_TRANSACTION`
`state`	`WAITING`, `READY`, `PROCESSED`, `EXPIRED`
`sequence_deviation`	`BEFORE`, `TOP`
`wait`	`FOREVER`, `NO_WAIT`
`delay`	`NO_DELAY`
`expiration`	`NEVER`

Security

Configuration information can be managed through procedures in the `DBMS_AQADM` package. Initially, only `SYS` and `SYSTEM` have the execution privilege for the procedures in `DBMS_AQADM` and `DBMS_AQ`. Any users who have been granted the `EXECUTE` rights to these two packages will be able to create, manage, and use queues in their own schema. The user would also need the `MANAGE ANY QUEUE` privilege in order to create and manage queues in other schemas.

Security with 8.0 and 8.1 Compatible Queues

AQ administrators of an 8.1 database are allowed to create queues with 8.0 or 8.1 compatibility. All 8.1 security features are enabled for 8.1 compatible queues. However, please note that AQ 8.1 security features work only with 8.1 compatible queues; 8.0 compatible queues are protected by the 8.0 compatible security features.

To create queues in 8.1 that can make use of the new security features, the compatible parameter in `DBMS_AQADM.CREATE_QUEUE_TABLE` must be set to ’8.1’ or above. If you want to use the new security features on a queue originally created in an 8.0 database, the queue table must be converted to 8.1 compatibility by running `DBMS_AQADM.MIGRATE_QUEUE_TABLE` on the queue table.

If a database downgrade is necessary, all 8.1 compatible queue tables have to be either converted back to 8.0 compatibility or dropped before the database downgrade can be carried out. During the conversion, all 8.1 security features on the queues, like the object privileges, will be dropped. When a queue is converted to 8.0 compatibility, the 8.0 security model apply to the queue, and only 8.0 security features are supported.

The following table lists the AQ security features supported in each version of Oracle8 database and their equivalence privileges across different database version.

Table 3–5 Security with 8.0- and 8.1-Compatible Queues

Privilege	8.0.x Database	8.0.x Compatible Queues in a 8.1.x Database	8.1.x Compatible Queues in a 8.1.x Database
`AQ_USER_ROLE`	Supported. The grantee is given the execute right of `DBMS_AQ` through the role.	Supported. The grantee is given the execute right of dbms_aq through the role.	Not supported. Equivalent privileges: **1.** execute right on dbms_aq **2.** enqueue any queue system privilege **3.** dequeue any queue system privilege
`AQ_ADMINISTRATOR_ROLE`	Supported.	Supported.	Supported.
`Execute right on DBMS_AQ`	Execute right on `DBMS_AQ` should be granted to developers who write AQ applications in PL/SQL.	Execute right on `DBMS_AQ` should be granted to developers who write AQ applications in PL/SQL.	Execute right on `DBMS_AQ` should be granted to all AQ users. To enqueue/dequeue on 8.1 compatible queues, the user needs the following privileges: **1.** execute right on `DBMS_AQ` **2.** either enqueue/dequeue privileges on target queues, or `ENQUEUE ANY QUEUE/DEQUEUE ANY QUEUE` system privileges

Privileges and Access Control

With Oracle 8.1, you can grant or revoke privileges at the object level on 8.1 compatible queues. You can also grant or revoke various system level privileges. The following table lists all common AQ operations, and the privileges need to perform these operations for an 8.1-compatible queue:

Table 3–6 Operations and Required Privileges in the 8.1 Security Model

Operation(s)	Privileges Required
`CREATE/DROP/MONITOR` own queues	Must be granted execute rights on `DBMS_AQADM`. No other privileges needed.
`CREATE/DROP/MONITOR` any queues	Must be granted execute rights on `DBMS_AQADM` and be granted `AQ_ADMINISTRATOR_ROLE` by another user who has been granted this role (`SYS` and `SYSTEM` are the first granters of `AQ_ADMINISTRATOR_ROLE`)
`ENQUEUE/ DEQUEUE to own queues`	Must be granted execute rights on DBMS_AQ. No other privileges needed.
`ENQUEUE/ DEQUEUE to another's queues`	Must be granted execute rights on `DBMS_AQ` and be granted privileges by the owner using `DBMS_AQADM.GRANT_QUEUE_PRIVILEGE`.
`ENQUEUE/ DEQUEUE to any queues`	Must be granted execute rights on `DBMS_AQ` and be granted `ENQUEUE ANY QUEUE` or `DEQUEUE ANY QUEUE` system privileges by an AQ administrator using `DBMS_AQADM.GRANT_SYSTEM_PRIVILEGE`.

Roles

Access to AQ operations in Oracle 8.0 is granted to users through roles which provide execution privileges on the AQ procedures. The fact that there is no control at the database object level when using Oracle 8.0 means that in Oracle 8.0 a user with the `AQ_USER_ROLE` can enqueue and dequeue to any queue in the system. Since Oracle 8.1 offers a finer-grained access control, the function of roles changes when you develop applications in the 8.1 context.

Administrator role

Oracle 8.1 continues to support the `AQ_AQMISTRATOR_ROLE`. As in 8.0, the `AQ_ADMINISTRATOR_ROLE` has been granted all the required privileges to administer queues. The privileges granted to the role let the grantee:

- perform any queue administrative operation, including create queues and queue tables on any schema in the database
- perform enqueue and dequeue operations on any queues in the database
- access statistics views used for monitoring the queues' workload

User role

`AQ_USER_ROLE` continues to work for queues that are created with 8.0 compatibility. However, you should avoid granting `AQ_USER_ROLE` in Oracle 8.1 since this role will not provide sufficient privileges for enqueuing or dequeuing on 8.1 compatible queues.

Your database administrator has the option of granting the system privileges `ENQUEUE ANY QUEUE` and `DEQUEUE ANY QUEUE`, exercising `DBMS_AQADM.GRANT_SYSTEM_PRIVILEGE` and `DBMS_AQADM.REVOKE_SYSTEM_PRIVILEGE` directly to a database user, provided that you wish the user to have this level of control. You as the application developer give rights to a queue by granting and revoking privileges at the object level by exercising `DBMS_AQADM.GRANT_QUEUE_PRIVILEGE` and `DBMS_AQADM.REVOKE_QUEUE_PRIVILEGE`.

As a database user you do not need any explicit object level or system level privileges to enqueue or dequeue to queues in your own schema other than the execute right on `DBMS_AQ`.

Access to AQ Object Types

The procedure *grant_type_access* is made obsolete in release 8.1.5 for both 8.0-compatible and 8.1 compatible queues. All internal AQ objects are now accessible to PUBLIC.

OCI Applications

For an OCI application to access an 8.0-compatible queue, the session user has to be granted the `EXECUTE` rights of `DBMS_AQ`. For an OCI application to access an 8.1-compatible queue, the session user has to be granted either the object privilege of the queue he intends to access or the `ENQUEUE ANY QUEUE` and/or `DEQUEUE ANY QUEUE` system privileges. The `EXECUTE` right of `DBMS_AQ` will not be checked against the session user's rights, if the queue he intends to access is an 8.1-compatible queue.

Propagation

AQ propagates messages through database links. The propagation driver dequeues from the source queue as owner of the source queue; hence, no explicit access rights have to be granted on the source queue. At the destination, the login user in the database link should either be granted `ENQUEUE ANY QUEUE` privilege or be granted the rights to enqueue to the destination queue. However, if the login user in the database link also owns the queue tables at the destination, no explicit AQ privileges need to be granted either.

Performance

Queues are stored in database tables. The performance characteristics of queue operations are very similar to the underlying database operations.

Table and index structures

To understand the performance characteristics of queues it is important to understand the tables and index layout for AQ objects.

Creating a queue table creates a database table with approximately 25 columns. These columns store the AQ meta data and the user defined payload. The payload can be of an object type or `RAW`. The AQ meta data contains object types and scaler types. A view and two indexes are created on the queue table. The view allows users to query the message data. The indexes are used to accelerate access to message data. Please refer to the create queue table command for a detailed description of the objects created.

Throughput

The code path of an enqueue operation is comparable to an insert into a multi-column table with two indexes. The code path of a dequeue operation is comparable to a select and delete operation on a similar table. These operations are performed using PL/SQL functions.

Availability

Oracle Parallel Server (OPS) can be used to ensure highly available access to queue data. Queues are implemented using database tables. The *tail* and the *head* of a queue can be extreme hot spots. Since OPS does not scale well in the presence of hot spots it is recommended to limit normal access to a queue from one instance only. In case of an instance failure messages managed by the failed instance can be processed immediately by one of the surviving instances.

Scalability

Queue operation scalability is similar to the underlying database operation scalability. If a dequeue operation with wait option is issued in a Multi-Threaded Server (MTS) environment the shared server process will be dedicated to the dequeue operation for the duration of the call including the wait time. The presence of many such processes could cause severe performance and availability problems and could result in deadlocking the shared server processes. For this reason it is recommended that dequeue requests with wait option be only issued via dedicated server processes. This restriction is not enforced.

Migrating Queue Tables

Purpose:

To upgrade a 8.0-compatible queue table to an 8.1-compatible queue table or to downgrade a 8.1-compatible queue table to an 8.0-compatible queue table.

Syntax:

```
DBMS_AQADM.MIGRATE_QUEUE_TABLE(
    queue_table        IN       VARCHAR2,
    compatible         IN       VARCHAR2)
```

Usage:

Table 3–7 DBMS_AQADM_MIGRATE_QUEUE_TABLE

Parameter	Description
`queue_table` `(IN VARCHAR2)`	Specifies name of the queue table that is to be migrated.
`compatible`	Set to '8.1' to upgrade an 8.0 queue table to 8.1 compatible. Set to '8.0' to downgrade an 8.1 queue table to 8.0 compatible.

Usage Notes

For the most current information regarding the interrelationship of different releases, please refer to "Compatibility" on page 1-36 in Chapter 1, "Introduction".

Example: To Upgrade An 8.0 Queue Table To A 8.1-Compatible Queue Table

Note: You may need to set up the following data structures for certain examples to work:

```
EXECUTE DBMS_AQADM.CREATE_QUEUE_TABLE (
   queue_table            => ’qtable1’,
   multiple_consumers     => TRUE,
   queue_payload_type     => ’aq.message_typ’,
   compatible             =>’8.0’);
```

```
EXECUTE DBMS_AQADM.MIGRATE_QUEUE_TABLE(
   queue_table => ’qtable1’,
   compatible   => ’8.1’);
```

Export and Import of Queue Data

When a queue table is exported, the queue table data and anonymous blocks of PL/SQL code are written to the export dump file. When a queue table is imported, the import utility executes these PL/SQL anonymous blocks to write the metadata to the data dictionary.

Exporting Queue Table Data

Queues are implemented on tables. The export of queues entails the export of the underlying queue tables and related dictionary tables. Export of queues can only be done at queue table granularity.

Exporting queue tables with multiple recipients

For every queue table that supports multiple recipients, there is an index-organized table (IOT) and a time-management table that contain important queue metadata. For 8.1 compatible queue tables there is also a subscriber table, a history table and a rules table. This metadata is essential to the operation of the queue, so the user must export these tables as well as the queue table itself for the queues in this queue table to work after import. During full database mode and user mode export, all these tables are exported automatically.

Because these metadata tables contain rowids of some rows in the queue table, the import process will generate a note about the rowids being obsoleted when importing the metadata tables. This message can be ignored as the queuing system will automatically correct the obsolete rowids as a part of the import operation. However, if another problem is encountered while doing the import (such as running out of rollback segment space), the problem should be corrected and the import should be repeated.

Exporting Rules

Rules are associated with a queue table. When a queue table is exported, all associated rules, if any, will be exported automatically.

Supported Export Modes

Export currently operates in three modes: full database mode, user mode, and table mode. The operation of the three export modes is described as follows.

Full database mode

This mode is supported. Queue tables, all related tables, system level grants, and primary and secondary object grants are exported automatically.

User mode

This mode is supported. Queue tables, all related tables and primary object grants are exported automatically.

Table mode

This is not recommended. If there is a need to export a queue table in table mode, the user is responsible for exporting all related objects which belong to that queue table. For example, when exporting an 8.1 compatible multi-consumer queue table MCQ, you will also need to export the following tables:

```
AQ$_MCQ_I
AQ$_MCQ_H
AQ$_MCQ_S
AQ$_MCQ_T
```

Incremental export

Incremental export on queue tables is not supported.

Importing Queue Table Data

Similar to exporting queues, the import of queues entails the import of the underlying queue tables and related dictionary data. After the queue table data is imported, the import utility executes the PL/SQL anonymous blocks in the dump file to write the metadata to the data dictionary.

Importing queue tables with multiple recipients

As explained earlier, for every queue table that supports multiple recipients, there is a index-organized table (IOT), a subscriber table, a history table, and a time-management table that contain important queue metadata. All these tables as well as the queue table itself, have to be imported for the queues in this queue table to work after the import.

Because these metadata tables contain rowids of some rows in the queue table, the import process will issue a note about the rowids being obsoleted when importing the metadata table. This message can be ignored, as the queuing system will automatically correct the obsolete rowids as a part of the import operation.

However, if another problem is encountered while doing the import (such as running out of rollback segment space), the problem should be corrected and the import should be rerun.

Import IGNORE parameter

We suggest that you do not import queue data into a queue table that already contains data. We recommend that the DBA should always set the `IGNORE` parameter of the import utility to `NO` when importing queue tables. If the `IGNORE` parameter is set to `YES`, and the queue table that already exists is compatible with the table definition in the dump file, then the rows will be loaded from the dump file into the existing table. At the same time, the old queue table definition and the old queue definition will be dropped and recreated. Hence, queue table and queue definitions prior to the import will be lost, and duplicate rows will appear in the queue table.

Propagation Issues

Caution: Propagation makes use of the system queue `aq$_prop_notify_X` (where X is the instance number of the instance where the source queue of a schedule resides) for handling propagation run-time events. These messages in this queue are stored in the system table `aq$_prop_table_X` (where X is the instance number of the instance where the source queue of a schedule resides). The queue `aq$_prop_notify_X` should never be stopped or dropped and the table `aq$_prop_notify_X` should never be dropped for propagation to work correctly.

Optimizing Propagation

In setting the number of `JOB_QUEUE_PROCESSES,` the DBA should aware that this need is determined by the number of queues *from* which the messages have to be propagated and the number of destinations (rather than queues) to which messages have to be propagated.

In this release, a new scalable scheduling algorithm has been incorporated for handling propagation. It has been designed to make optimal use of the available job queue processes and also minimize the time it takes for a message to show up at a destination once it has been enqueued into the source queue, thereby providing near OLTP behavior. This algorithm is capable of simultaneously handling an unlimited number of schedules. The algorithm also has robust support for handling various types of failures. While propagation tries to make the optimal use of the available job queue processes, the number of job queue processes to be started also depends on the existence of non-propagation related jobs such as replication jobs. Hence, it is very important to use the following guidelines to get the best results from this new algorithm.

The new algorithm uses the job queue processes as follows: (for this discussion an active schedule is one which has a valid current window)

- if the number of active schedules is less than half the number of job queue processes, the number of job queue processes acquired corresponds to the number of active schedules
- if the number of active schedules is more than half the number of job queue processes, after acquiring half the number of job queue processes multiple active schedules are assigned to an acquired job queue process

- if system is overloaded (all schedules are busy propagating), depending on the availability additional job queue processes will be acquired up to one less than the total number of job queue processes
- if none of the active schedules handled by a process have messages to be propagate then that job queue process will be released
- the algorithm performs automatic load balancing by transferring schedules from a heavily loaded process to a lightly load process such that no process is excessively loaded

The scheduling algorithm places the restriction that at least 2 job queue processes be available for propagation. If there are non-propagation related jobs then more number of job queue processes is needed. If heavily loaded conditions (when there are a large number of active schedules all of which have messages to be propagated) are expected then it is recommended to start a larger number of job queue processes keeping in mind that the job queue processes will be used for non-propagation related jobs as well. In a system which only has propagation jobs, then 2 job queue processes can handle all schedules but higher the number the faster the messages get propagated. Note that, since one job queue process can propagate messages from multiple schedules, it is not necessary to have the same number of job queue processes as the number of schedules.

Handling Failures in Propagation

The new algorithm also has robust support for handling failures. It may not be able to propagate messages from a queue due to various types of failures. Some of the common reasons include failure of the database link, non-availability of the remote database, non-existence of the remote queue, remote queue not started and security violation while trying to enqueue messages into the remote queue. Under all these circumstances the appropriate error messages will be reported in the dba_queue_schedules view. When an error occurs in a schedule, propagation of messages in that schedule is attempted periodically using an exponential backoff algorithm for a maximum of 16 times after which the schedule is disabled. If the problem causing the error is fixed and the schedule is enabled, the error fields that indicate the last error date, time and message will still continue to show the error information. These fields are reset only when messages are successfully propagated in that schedule. During the later stages of the exponential backoff, the time span between propagation attempts can be large in the tune of hours or even days. This happens only when an error has been neglected for a long time. Under such circumstances it may be better to unschedule the propagation and schedule it again.

Enterprise Manager Support

Enterprise manager supports GUIs for most of the administrative functions listed in the administrative interfaces section.

These include:

1. Queues as part of schema manager to view properties.
2. Create, start, stop and drop queue.
3. Schedule and unschedule propagation.
4. Add and remove subscriber.
5. View the current propagation schedule.
6. Grant & revoke privileges.

Using XA with AQ

You must specify "Objects=T" in the xa_open string if you want to use the AQ OCI interface. This forces XA to initialize the client side cache in Objects mode. You do not need to do this if you plan to use AQ through PL/SQL wrappers from OCI or Pro*C. The LOB memory management concepts you picked up from the Pro* documentation is not relevant for AQ raw messages because AQ provides a simple RAW buffer abstraction (although they are stored as LOBs).

You must use AQ navigation option carefully when you are using AQ from XA. XA cancels cursor fetch state after an xa_end. Hence, if you want to continue dequeuing between services (i.e. xa_start/xa_end boundaries) you must reset the dequeue position by using the `FIRST_MESSAGE` navigation option. Otherwise, you will get an ORA-25237 (navigation used out of sequence).

Sample DBA Actions as Preparation for Working with AQ

Creating a User as an AQ Administrator

To set a user up as an AQ administrator, you must the following steps

```
CONNECT system/manager
CREATE USER aqadm IDENTIFIED BY aqadm;
GRANT AQ_ADMINISTRATOR_ROLE TO aqadm;
GRANT CONNECT, RESOURCE TO aqadm;
```

Additionally, you might grant execute on the AQ packages as follows:

```
GRANT EXECUTE ON DBMS_AQADM TO aqadm;
GRANT EXECUTE ON DBMS_AQ TO aqadm;
```

This allows the user to execute the procedures in the AQ packages from within a user procedure.

Creating User AQUSER1 and AQUSER2 as Two AQ Users

If you want to create an AQ user who creates and accesses queues within his/her own schema, follow the steps outlined in the previous section except do not grant the `AQ_ADMINISTRATOR_ROLE`.

```
CONNECT system/manager
CREATE USER aquser1 IDENTIFIED BY aquser1;
GRANT CONNECT, RESOURCE TO aquser1;
```

Additionally, you might grant execute on the AQ packages as follows:

```
GRANT EXECUTE ON DBMS_AQADM to aquser1;
GRANT EXECUTE ON DBMS_AQ TO aquser1;
```

If you wish to create an AQ user who does not create queues but uses a queue in another schema, first follow the steps outlined in the previous section. In addition, you must grant object level privileges. However, note that this applies only to queues defined using 8.1 compatible queue tables.

```
CONNECT system/manager
CREATE USER aquser2 IDENTIFIED BY aquser2;
GRANT CONNECT, RESOURCE TO aquser2;
```

Additionally, you might grant execute on the AQ packages as follows:

```
GRANT EXECUTE ON DBMS_AQADM to aquser2;
GRANT EXECUTE ON DBMS_AQ TO aquser2;
```

For `aquser2` to access the queue, `aquser1_q1` in `aquser1` schema, `aquser1` must execute the following statements:

```
CONNECT aquser1/aquser1
EXECUTE DBMS_AQADM.GRANT_QUEUE_PRIVILEGE(
  'ENQUEUE','aquser1_q1','aquser2',FALSE);
```

4

Administrative Interface: Basic Operations

In this chapter we describe the administrative interface to Oracle Advanced Queuing in terms of use cases. That is, we discuss each operation (such as "Create a Queue Table") as a use case by that name. The table listing all the use cases is provided at the head of the chapter (see "Use Case Model: Administrative Interface — Basic Operations" on page 4-2).

A summary figure, "Use Case Diagram: Administrator's Interface — Basic Operations", locates all the use cases in single drawing. If you are using the HTML version of this document, you can use this figure to navigate to the use case in which you are interested by clicking on the relevant use case title.

The individual use cases are themselves laid out as follows:

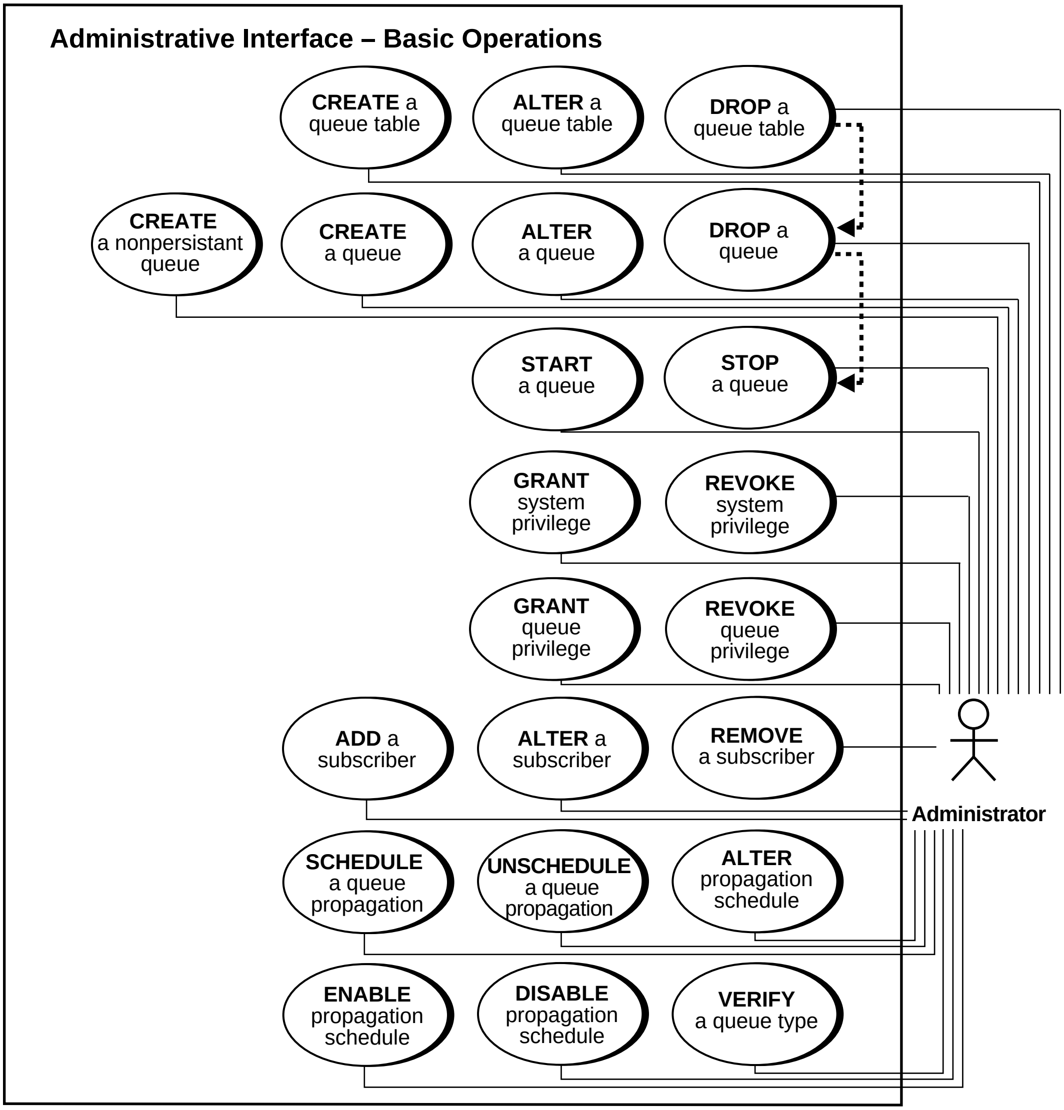
- A figure that depicts the use case (see "Preface" for a description of how to interpret these diagrams).
- A listing of the syntax.
- Basic examples
- Usage Notes, if any.

Use Case Model: Administrative Interface — Basic Operations

Table 4–1 Use Case Model: Administrative Interface — Basic Operations

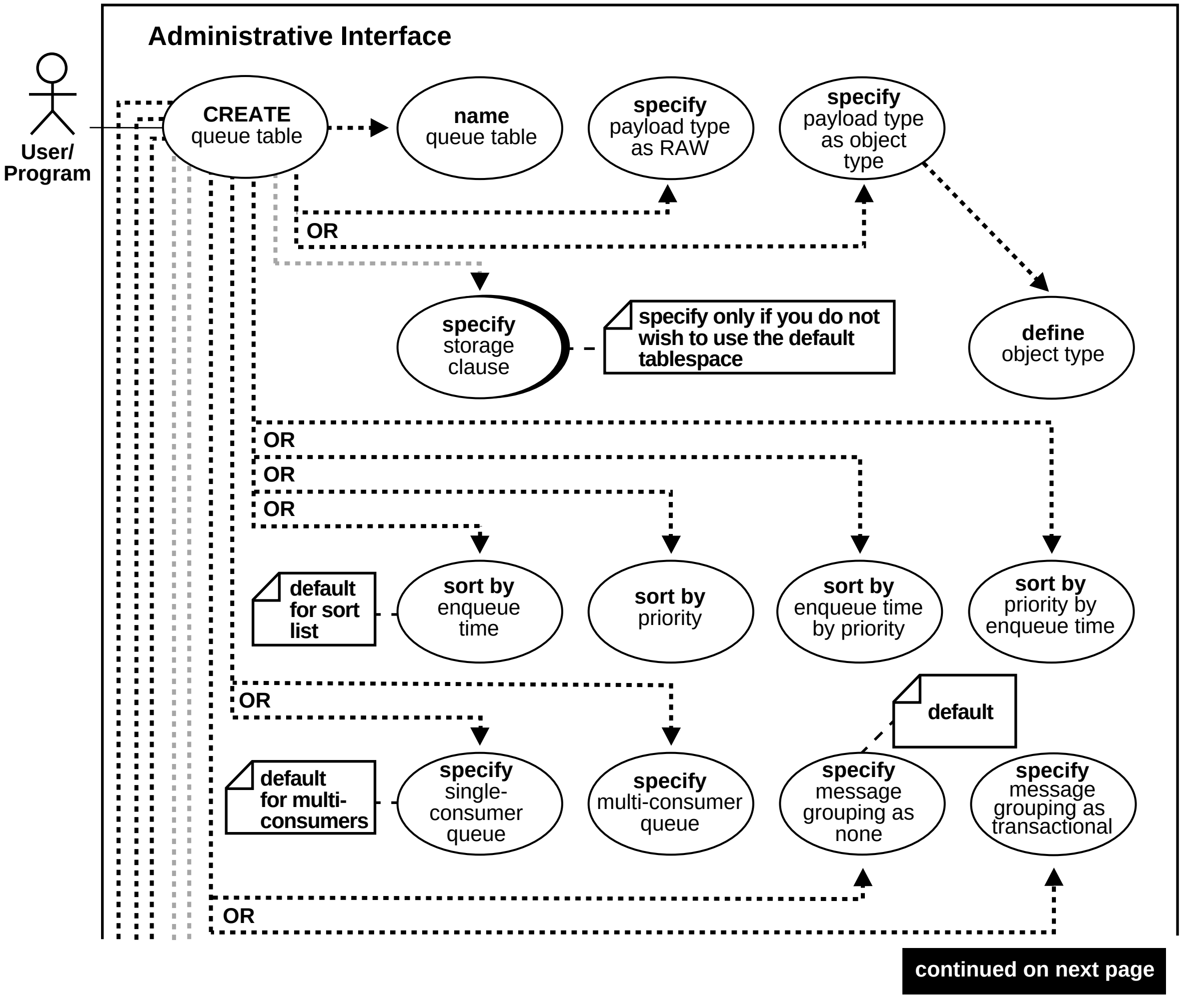
Use Case
Create a Queue Table on page 4-4
Create a Queue Table [Set Storage Clause] on page 4-11
Alter a Queue Table on page 4-12
Drop a Queue Table on page 4-15
Create a Queue on page 4-18
Create a Non-Persistent Queue on page 4-24
Alter a Queue on page 4-27
Drop a Queue on page 4-30
Start a Queue on page 4-32
Stop a Queue on page 4-34
Grant System Privilege on page 4-37
Revoke System Privilege on page 4-40
Grant Queue Privilege on page 4-42
Revoke Queue Privilege on page 4-44
Add a Subscriber on page 4-46
Alter a Subscriber on page 4-50
Remove a Subscriber on page 4-53
Schedule a Queue Propagation on page 4-56
Unschedule a Queue Propagation on page 4-60
Verify a Queue Type on page 4-62
Alter a Propagation Schedule on page 4-65
Enable a Propagation Schedule on page 4-68
Disable a Propagation Schedule on page 4-70

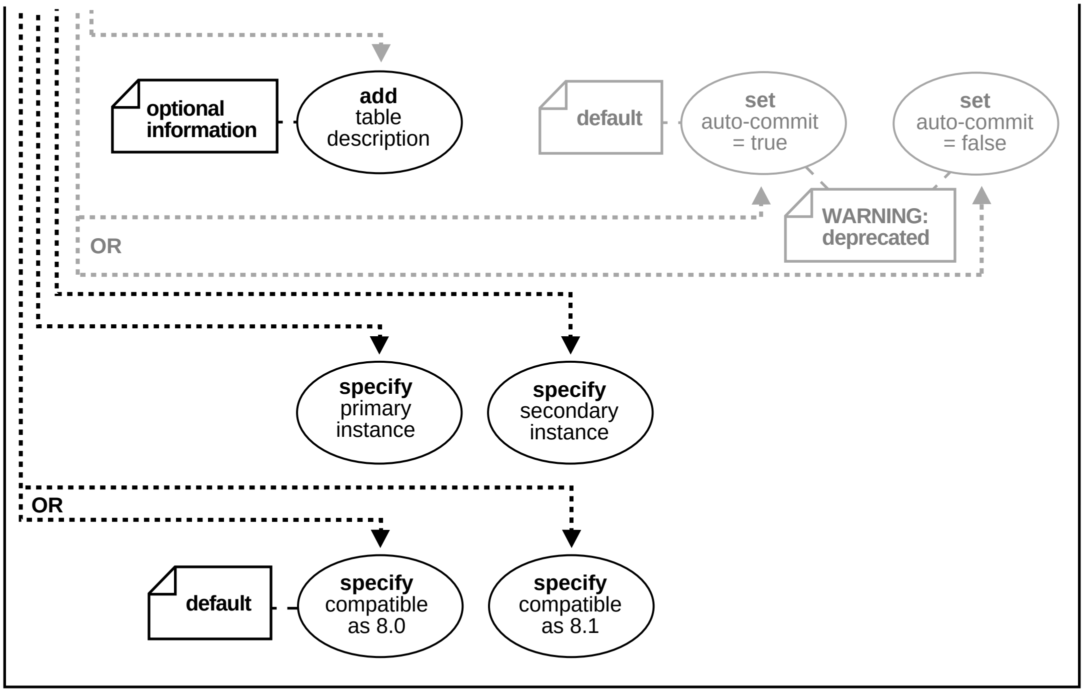
Figure 4–1 Use Case Diagram: Administrator's Interface — Basic Operations



Create a Queue Table

Figure 4–2 Use Case Diagram: Create a Queue Table





To refer to the table of all basic operations having to do with the Administrative Interface see:

- "Use Case Model: Administrative Interface — Basic Operations" on page 4-2

Purpose:

Create a queue table for messages of a pre-defined type. The sort keys for dequeue ordering, if any, need to be defined at table creation time. The following objects are created at this time:

- The default exception queue associated with the queue table called `aq$_<queue_table_name>_e`.
- A read-only view which is used by AQ applications for querying queue data called `aq$<queue_table_name>`.

- An index or an index organized table (IOT) for the queue monitor operations called `aq$_<queue_table_name>_t`.
- An index or an index organized table (IOT) in the case of multiple consumer queues for dequeue operations called `aq$_<queue_table_name>_i.`

For 8.1-compatible multiconsumer queue tables the following additional objects are created:

- A table called `aq$_<queue_table_name>_s.` This table stores information about the subscribers.
- A table called `aq$_<queue_table_name>_r.` This table stores information about rules on subscriptions.
- An index organized table (IOT) called `aq$_<queue_table_name>_h.`This table stores the dequeue history data.

Syntax

```
DBMS_AQADM.CREATE_QUEUE_TABLE (
    queue_table        IN      VARCHAR2,
    queue_payload_type IN      VARCHAR2,
    storage_clause     IN      VARCHAR2 default NULL,
    sort_list          IN      VARCHAR2 default NULL,
    multiple_consumers IN      BOOLEAN default FALSE,
    message_grouping   IN      BINARY_INTEGER default NONE,
    comment            IN      VARCHAR2 default NULL,
    auto_commit        IN      BOOLEAN default TRUE,
    primary_instance   IN      BINARY_INTEGER default 0,
    secondary_instance IN      BINARY_INTEGER default 0,
    compatible         IN      VARCHAR2 default ’8.0’);
```

Usage:

Table 4–2 DBMS_AQADM.CREATE_QUEUE_TABLE

Parameter	Description
`queue_table` `(IN VARCHAR2)`	specifies the name of a queue table to be created.
`queue_payload_type` `(IN VARCHAR2)`	specifies the type of the user data stored. Please see section entitled "Type name" on page 3-4 for valid values for this parameter.
`storage_clause` `(IN VARCHAR2)`	specifies the storage parameter. The storage parameter will be included in the 'CREATE TABLE' statement when the queue table is created. The storage parameter can be made up of any combinations of the following parameters: PCTFREE, PCTUSED, INITRANS, MAXTRANS, TABLEPSACE, LOB and a table storage clause. If tablespace is not specified in the `storage_clause` parameter, the queue table and all its related objects are created in the default user tablespace. If a tablespace is specified in the storage_clause parameter, the queue table and all its related objects are created in the tablespace specified in the storage clause. Please refer to the SQL reference guide for the usage of these parameters.
`sort_list` `(IN VARCHAR2)`	specifies the columns to be used as the sort key in ascending order. *Sort_list* has the following format: '<sort_column_1>,<sort_column_2>'. The allowed column names are *priority* and *enq_time*. If both columns are specified then <sort_column_1> defines the most significant order. Once a queue table is created with a specific ordering mechanism, all queues in the queue table inherit the same defaults. The order of a queue table cannot be altered once the queue table has been created. If no sort list is specified all the queues in this queue table will be sorted by the enqueue time in ascending order. This order is equivalent to FIFO order. Even with the default ordering defined, a dequeuer is allowed to choose a message to dequeue by specifying its *msgid* or *correlation*. *Msgid*, *correlation* and *sequence_deviation* take precedence over the default dequeueing order if they are specified.
`multiple_consumers` `(IN BOOLEAN)`	`FALSE`: Queues created in the table can only have one consumer per message. This is the default. `TRUE`: Queues created in the table can have multiple consumers per message.
`message_grouping` `(IN BINARY_INTEGER)`	specifies the message grouping behavior for queues created in the table. `NONE`: Each message is treated individually. `TRANSACTIONAL`: Messages enqueued as part of one transaction are considered part of the same group and can be dequeued as a group of related messages.

Table 4–2 DBMS_AQADM.CREATE_QUEUE_TABLE

Parameter	Description
`comment` `(IN VARCHAR2)`	specifies the user-specified description of the queue table. This user comment will be added to the queue catalog.
`auto_commit` `(IN BOOLEAN)`	`TRUE`: causes the current transaction, if any, to commit before the `CREATE_QUEUE_TABLE` operation is carried out. The `CREATE_QUEUE_TABLE` operation becomes persistent when the call returns. This is the default. `FALSE`: The operation is part of the current transaction and will become persistent only when the caller issues a commit. Caution: This parameter has been deprecated.
`primary_instance (IN BINARY_INTEGER)`	This is the primary owner of the queue table. Queue monitor scheduling and propagation for the queues in the queue table will be done in this instance. The default value for primary instance is `0`, which means queue monitor scheduling and propagation will be done in any available instance.
`secondary_instance (IN BINARY_INTEGER)`	The queue table fails over to the secondary instance if the primary instance is not available. The default value is `0`, which means that the queue-table will fail over to any available instance.
`compatible (VARCHAR2)`	specifies the lowest database version with which the queue is compatible. Currently the possible values are either ’8.0’ or ’8.1’. The default is ’8.0’.

Usage Notes:

- `CLOB`, `BLOB` or `BFILE` objects are valid attributes for an AQ object type load. However, only CLOB and BLOB can be propagated using AQ propagation in Oracle8*i* release 8.1.x.
- You can specify and modify the `primary_instance` and `secondary_instance` only when the database is in 8.1-compatible mode.
- You cannot specify a secondary instance unless there is a primary instance.

Example: Create a Queue Table Using PL/SQL (DBMS_AQADM Package)

Note: You may need to set up the following data structures for certain examples to work:

```
CONNECT system/manager;
DROP USER aqadm CASCADE;
CREATE USER aqadm IDENTIFIED BY aqadm;
GRANT CONNECT, RESOURCE TO aqadm;
GRANT EXECUTE ON DBMS_AQADM TO aqadm;
GRANT Aq_administrator_role TO aqadm;
DROP USER aq CASCADE;
CREATE USER aq IDENTIFIED BY aq;
GRANT CONNECT, RESOURCE TO aq;
GRANT EXECUTE ON dbms_aq TO aq;
```

Create queue table for queues containing messages of object type

```
CREATE type aq.Message_typ as object (
   Subject               VARCHAR2(30),
   Text                  VARCHAR2(80));

/* Note: if you do not stipulate a schema, you default to the user’s schema. */
EXECUTE dbms_aqadm.create_queue_table (
   Queue_table           => ’aq.ObjMsgs_qtab’,
   Queue_payload_type    => ’aq.Message_typ’);
```

Create queue table for queues containing messages of RAW type

```
EXECUTE dbms_aqadm.create_queue_table (
   Queue_table           => ’aq.RawMsgs_qtab’,
   Queue_payload_type    => ’RAW’);
```

Create a queue table for prioritized messages

```
EXECUTE dbms_aqadm.create_queue_table (
   Queue_table           => ’aq.PriorityMsgs_qtab’,
   Sort_list             => ’PRIORITY,ENQ_TIME’,
   Queue_payload_type    => ’aq.Message_typ’);
```

Create a queue table for multiple consumers

```
EXECUTE dbms_aqadm.create_queue_table (
   Queue_table            => ’aq.MultiConsumerMsgs_qtab’,
   Multiple_consumers     => TRUE,
   Queue_payload_type     => ’aq.Message_typ’);
```

Create a queue table for multiple consumers compatible with 8.1

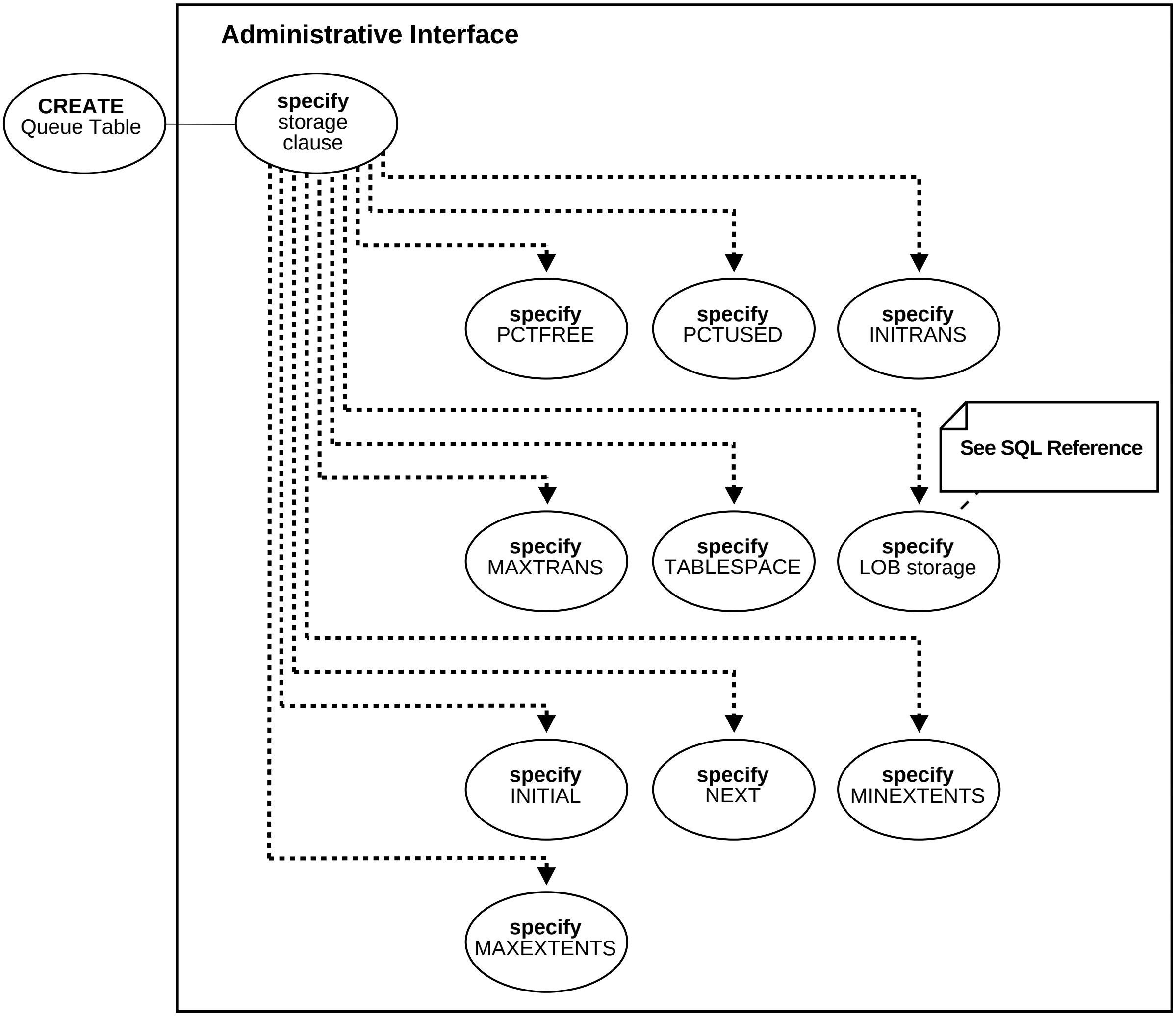
```
EXECUTE dbms_aqadm.create_queue_table (
   Queue_table            => ’aq.Multiconsumermsgs8_1qtab’,
   Multiple_consumers     =>  TRUE,
   Compatible             => ’8.1’,
   Queue_payload_type     => ’aq.Message_typ’);
```

Create a queue table in a specified tablespace

```
EXECUTE dbms_aqadm.create_queue_table(
        queue_table        => 'aq.aq_tbsMsg_qtab',
        queue_payload_type => 'aq.Message_typ',
        storage_clause     => 'tablespace aq_tbs');
```

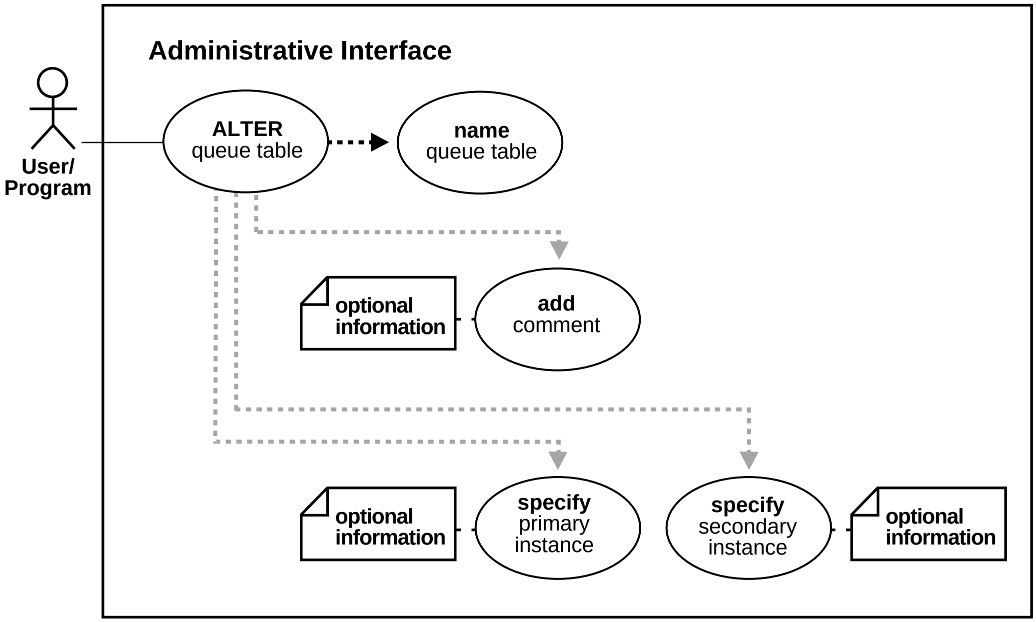
Create a Queue Table [Set Storage Clause]

Figure 4–3 Use Case Diagram: Create a Queue Table [Set Storage Clause]



Alter a Queue Table

Figure 4–4 Use Case Diagram: Alter a Queue Table



To refer to the table of all basic operations having to do with the Administrative Interface see:

- "Use Case Model: Administrative Interface — Basic Operations" on page 4-2

Purpose:

Alter the existing properties of a queue table.

Syntax:

```
DBMS_AQADM.ALTER_QUEUE_TABLE (
   queue_table          IN   VARCHAR2,
   comment              IN   VARCHAR2 default NULL,
   primary_instance     IN   BINARY_INTEGER default NULL,
   secondary_instance   IN   BINARY_INTEGER default NULL);
```

Usage:

Table 4–3 DBMS_AQADM.ALTER_QUEUE_TABLE

Parameter	Description
`queue_table` `(IN VARCHAR2)`	specifies the name of a queue table to be altered.
`comment` `(IN VARCHAR2)`	modifies the user-specified description of the queue table. This user comment will be added to the queue catalog. The default value is `NULL` which means that the value will not be changed.
`primary_instance (IN BINARY_INTEGER)`	This is the primary owner of the queue table. Queue monitor scheduling and propagation for the queues in the queue table will be done in this instance. The default value is `NULL` which means that the current value will not be changed.
`secondary_instance (IN BINARY_INTEGER)`	The queue table fails over to the secondary instance if the primary instance is not available. The default value is `NULL` which means that the current value will not be changed.

Example: Alter a Queue Table Using PL/SQL (DBMS_AQADM Package)

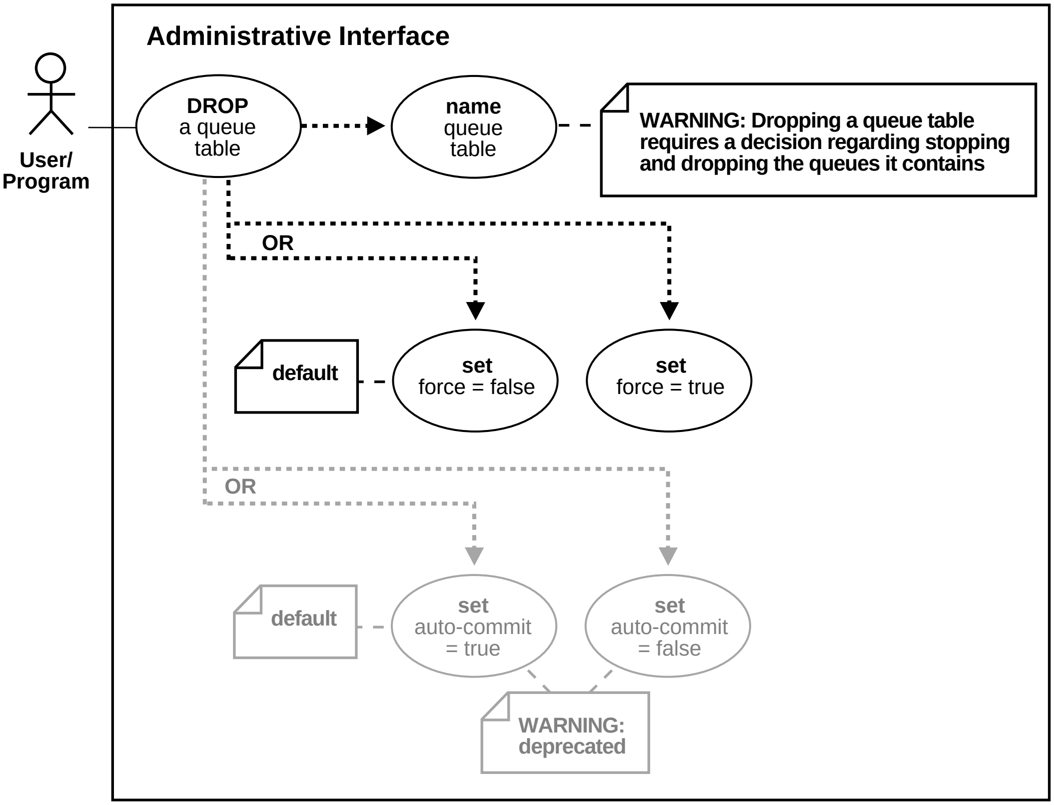
```
/* Altering the table to change the primary, secondary instances for queue owner
   (only applicable for OPS environments).The primary instance is the instance
   number of the primary owner of the queue table. The secondary instance is the
   instance number of the secondary owner of the queue table. */
 EXECUTE dbms_aqadm.alter_queue_table (
   Queue_table          => 'aq.ObjMsgs_qtab',
   Primary_instance     => 3,
   Secondary_instance   => 2);

/* Altering the table to change the comment for a queue table: */
EXECUTE dbms_aqadm.alter_queue_table (
   Queue_table          => 'aq.ObjMsgs_qtab',
   Comment              => 'revised usage for queue table');
```

Usage Notes

- You can specify and modify the primary_instance and secondary_instance only in 8.1-compatible mode.
- You cannot specify a secondary instance unless there is a primary instance.

Drop a Queue Table

Figure 4–5 Use Case Diagram: Drop a Queue Table



To refer to the table of all basic operations having to do with the Administrative Interface see:

- "Use Case Model: Administrative Interface — Basic Operations" on page 4-2

Purpose:

Drop an existing queue table. Note that you must stop and drop all the queues in a queue tables before the queue table can be dropped. You must do this explicitly unless the `force` option is used in which case this done automatically.

Syntax:

```
DBMS_AQADM.DROP_QUEUE_TABLE (
   queue_table       IN    VARCHAR2,
   force             IN    BOOLEAN default FALSE,
   auto_commit       IN    BOOLEAN default TRUE);
```

Usage:

Table 4–4 DBMS_AQADM.DROP_QUEUE_TABLE

Parameter	**Description**
`queue_table` `(IN VARCHAR2)`	specifies the name of a queue table to be dropped.
`force` `(IN BOOLEAN)`	`FALSE`: The operation will not succeed if there are any queues in the table.This is the default. `TRUE`: All queues in the table are stopped and dropped automatically.
`auto_commit` `(IN BOOLEAN)`	`TRUE`: Causes the current transaction, if any, to commit before the `DROP_QUEUE_TABLE` operation is carried out. The `DROP_QUEUE_TABLE` operation becomes persistent when the call returns. This is the default. `FALSE`: The operation is part of the current transaction and will become persistent only when the caller issues a commit. Caution: This parameter has been deprecated.

Caution: You may need to set up or drop data structures for certain examples to work:

Example: Drop a Queue Table Using PL/SQL (DBMS_AQADM Package)

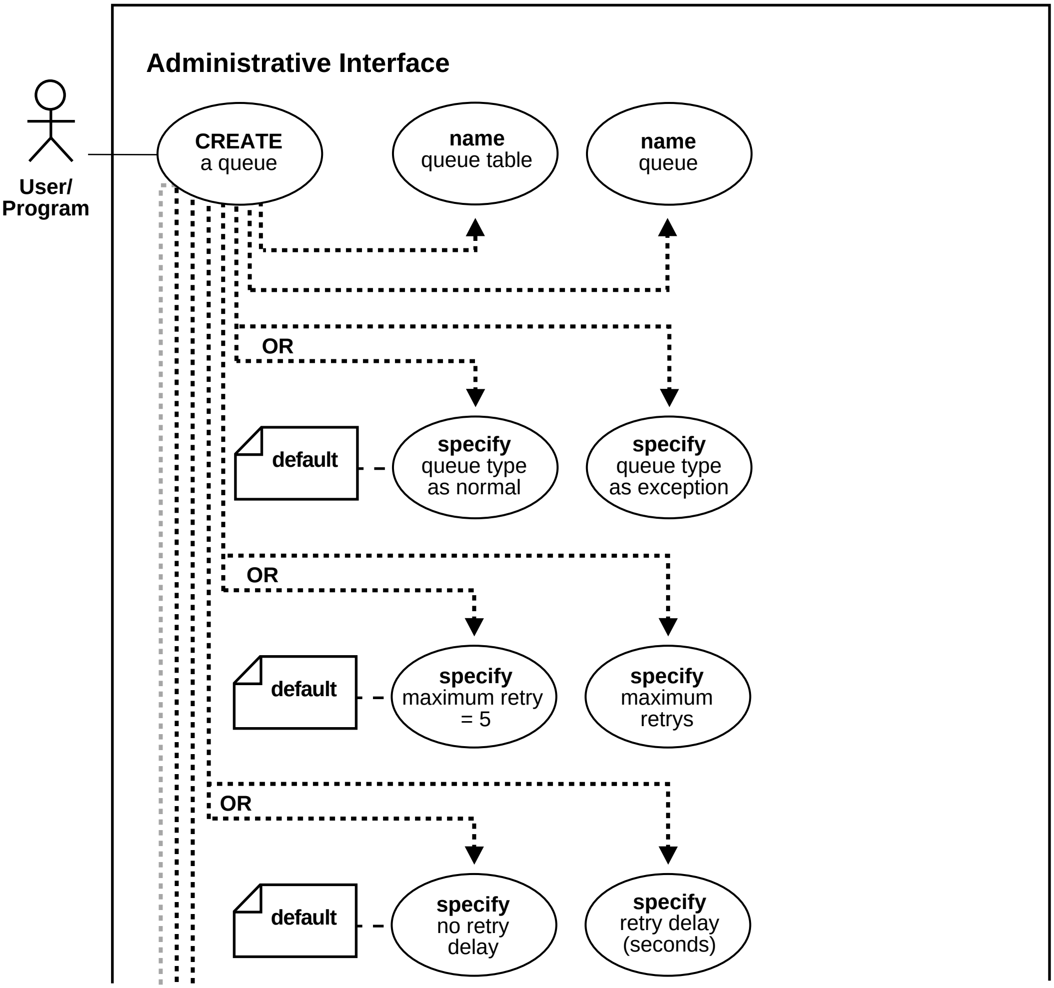
```
/* Drop the queue table (for which all queues have been previously dropped by
   the user) */
EXECUTE dbms_aqadm.drop_queue_table (
   queue_table        => 'aq.Objmsgs_qtab');
```

Caution: You may need to set up or drop data structures for certain examples to work:

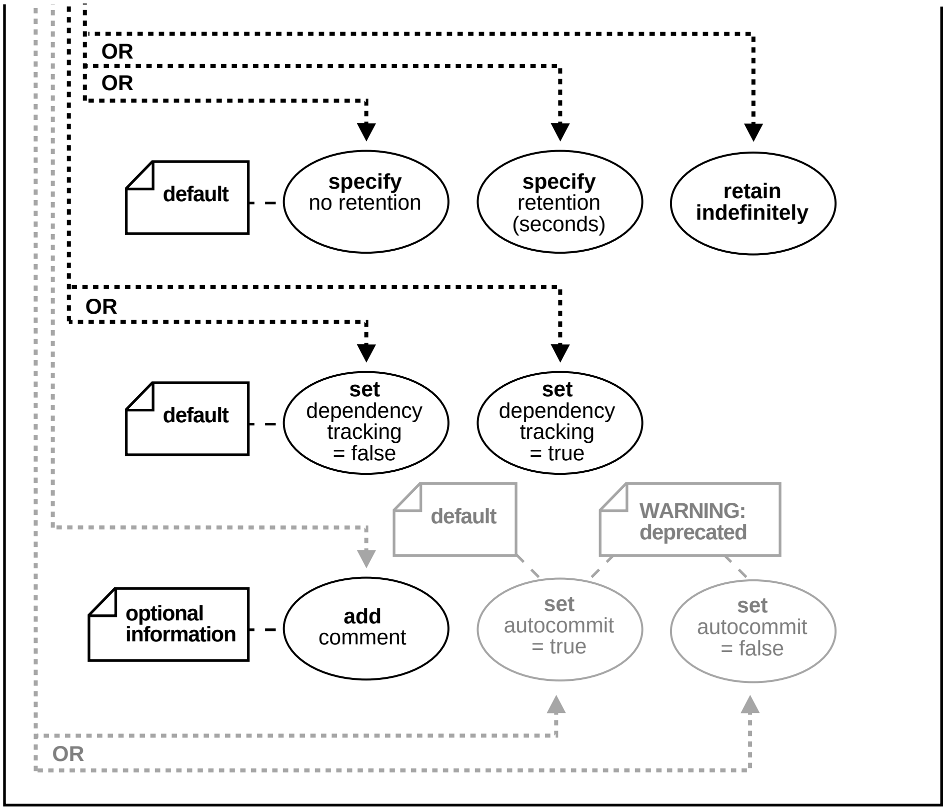
```
/* Drop the queue table and force all queues to be stopped and dropped by the
   system */
EXECUTE dbms_aqadm.drop_queue_table (
   queue_table        => 'aq.Objmsgs_qtab',
   force              => TRUE);
```

Create a Queue

Figure 4–6 Use Case Diagram: Create a Queue



continued on next page



To refer to the table of all basic operations having to do with the Administrative Interface see:

- "Use Case Model: Administrative Interface — Basic Operations" on page 4-2

Purpose:

Create a queue in the specified queue table.

Syntax:

```
DBMS_AQADM.CREATE_QUEUE (
   queue_name          IN       VARCHAR2,
```

```
queue_table         IN       VARCHAR2,
queue_type          IN       BINARY_INTEGER default NORMAL_QUEUE,
max_retries         IN       NUMBER default NULL,
retry_delay         IN       NUMBER default 0,
retention_time      IN       NUMBER default 0,
dependency_tracking IN       BOOLEAN default FALSE,
comment             IN       VARCHAR2 default NULL,
auto_commit         IN       BOOLEAN default TRUE);
```

Usage:

Table 4–5 DBMS_AQADM.CREATE_QUEUE

Parameter	Description
`queue_name` `(IN VARCHAR2)`	specifies the name of the queue that is to be created. The name must be unique within a schema and must follow object name guidelines in the *Oracle8i SQL Reference* with regard to reserved characters.
`queue_table` `(IN VARCHAR2)`	specifies the name of the queue table that will contain the queue.
`queue_type` `(IN BINARY_INTEGER)`	specifies whether the queue being created is an exception queue or a normal queue. `NORMAL_QUEUE`: The queue is a normal queue. This is the default. `EXCEPTION_QUEUE`: It is an exception queue. Only the dequeue operation is allowed on the exception queue.
`max_retries` `(IN NUMBER)`	limits the number of times a dequeue with the `REMOVE` mode can be attempted on a message. The count is incremented when the application issues a rollback after executing the dequeue. The message is moved to the exception queue when it is reaches its `max_retries`. The default is `NULL` but is set internally to 5. Note that `max_retries` is supported for all single consumer queues and 8.1-compatible multiconsumer queues but not for 8.0-compatible multiconsumer queues.
`retry_delay` `(IN NUMBER)`	specifies the delay time, in seconds before this message is scheduled for processing again after an application rollback. The default is 0, which means the message can be retried as soon as possible. This parameter will have no effect if `max_retries` is set to 0. Note that `retry_delay` is supported for single consumer queues and 8.1-compatible multiconsumer queues but not for 8.0 -compatible multiconsumer queues.
`retention_time` `(IN NUMBER)`	specifies the number of seconds for which a message will be retained in the queue table after being dequeued from the queue. `INFINITE`: Message will be retained forever. number: Number of seconds for which to retain the messages. The default is 0, i.e. no retention.

Table 4–5 DBMS_AQADM.CREATE_QUEUE

Parameter	Description
`dependency_tracking` `(IN BOOLEAN)`	Reserved for future use. `FALSE`: This is the default. `TRUE`: Not permitted in this release.
`comment` `(IN VARCHAR2)`	User-specified description of the queue. This user comment will be added to the queue catalog.
`auto_commit` `(IN BOOLEAN)`	`TRUE`: Causes the current transaction, if any, to commit before the `CREATE_QUEUE` operation is carried out. The `CREATE_QUEUE` operation becomes persistent when the call returns. This is the default. `FALSE`: The operation is part of the current transaction and will become persistent only when the caller issues a commit. Caution: This parameter has been deprecated.

Usage Notes

- All queue names must be unique within a schema. Once a queue is created with `CREATE_QUEUE`, it can be enabled by calling `START_QUEUE`. By default, the queue is created with both enqueue and dequeue disabled.
- To view retained messages, you can either dequeue by message ID or use SQL.

Example: Create a Queue Using PL/SQL (DBMS_AQADM)

Create a queue within a queue table for messages of object type

```
/* Create a message type: */
CREATE type aq.Message_typ as object (
   Subject     VARCHAR2(30),
   Text        VARCHAR2(80));

/* Create a object type queue table and queue: */
EXECUTE dbms_aqadm.create_queue_table (
   Queue_table        => 'aq.ObjMsgs_qtab',
   Queue_payload_type => 'aq.Message_typ');
```

```
EXECUTE dbms_aqadm.create_queue (
   Queue_name         =>  'msg_queue',
   Queue_table        => 'aq.ObjMsgs_qtab');
```

Create a queue within a queue table for messages of RAW type

```
/* Create a RAW type queue table and queue: */
EXECUTE dbms_aqadm.create_queue_table (
   Queue_table          => 'aq.RawMsgs_qtab',
   Queue_payload_type   => 'RAW');

/* Create queue: */
EXECUTE dbms_aqadm.create_queue (
   Queue_name          => 'raw_msg_queue',
   Queue_table         => 'aq.RawMsgs_qtab');
```

Create a prioritized message queue table and queue

Caution: You may need to set up or drop data structures for certain examples to work:

```
/* Create a queue table for priortized messages: */
EXECUTE dbms_aqadm.create_queue_table (
   Queue_table        => 'aq.PriorityMsgs_qtab',
   Sort_list          => 'PRIORITY,ENQ_TIME',
   Queue_payload_type => 'aq.Message_typ');
/* Create queue: */
EXECUTE dbms_aqadm.create_queue (
   Queue_name         => 'priority_msg_queue',
   Queue_table        => 'aq.PriorityMsgs_qtab');
```

Create a queue table and queue meant for multiple consumers

Caution: You may need to set up or drop data structures for certain examples to work:

```
/* Create a queue table for multi-consumers: */
EXECUTE dbms_aqadm.create_queue_table (
```

```
   queue_table        => 'aq.MultiConsumerMsgs_qtab',
   Multiple_consumers => TRUE,
   Queue_payload_type => 'aq.Message_typ');

/* Create queue: */
EXECUTE dbms_aqadm.create_queue (
   Queue_name         => 'MultiConsumerMsg_queue',
   Queue_table        => 'aq.MultiConsumerMsgs_qtab');
```

Create a queue table and queue to demonstrate propagation

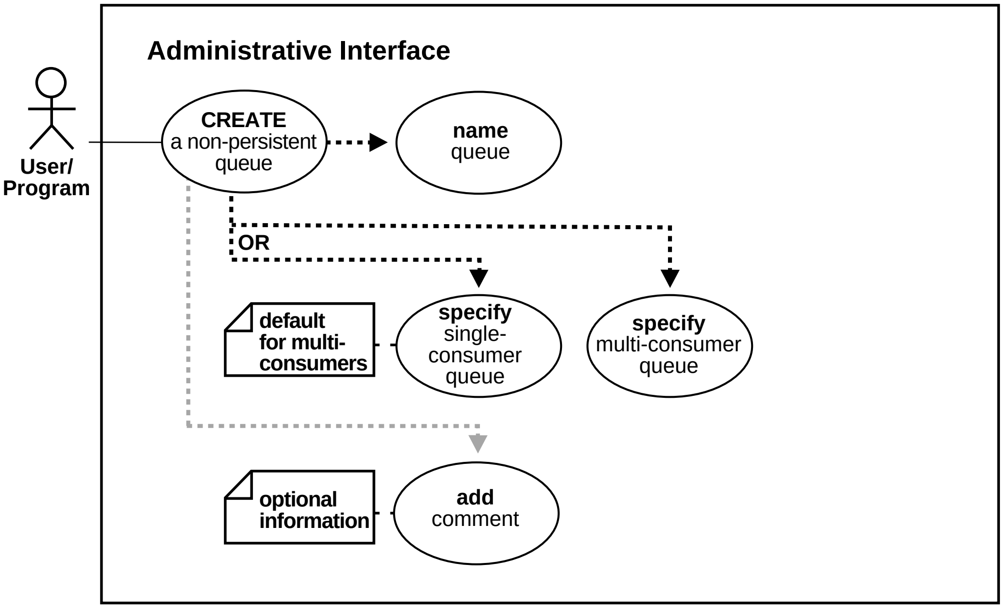
```
/* Create queue: */
EXECUTE dbms_aqadm.create_queue (
   Queue_name        => 'AnotherMsg_queue',
   queue_table       => 'aq.MultiConsumerMsgs_qtab');
```

Create a queue table and queue for multiple consumers compatible with 8.1

```
/* Create a queue table for multi-consumers compatible with Release 8.1: */
EXECUTE dbms_aqadm.create_queue_table (
   Queue_table        => 'aq.MultiConsumerMsgs81_qtab',
   Multiple_consumers => TRUE,
   Compatible         => '8.1',
   Queue_payload_type => 'aq.Message_typ');

EXECUTE dbms_aqadm.create_queue (
   Queue_name         => 'MultiConsumerMsg81_queue',
   Queue_table        => 'aq.MultiConsumerMsgs81_qtab');
```

Create a Non-Persistent Queue

Figure 4–7 Use Case Diagram: Create a Non-Persistent Queue



To refer to the table of all basic operations having to do with the Administrative Interface see:

- "Use Case Model: Administrative Interface — Basic Operations" on page 4-2

Purpose

Create a non-persistent `RAW` queue.

Syntax

```
DBMS_AQADM.CREATE_NP_QUEUE (
   queue_name              IN       VARCHAR2,
   multiple_consumers      IN       BOOLEAN default FALSE,
   comment                 IN       VARCHAR2 default NULL);
```

Usage:

Table 4–6 DBMS_AQADM.CREATE_NP_QUEUE

Parameter	Description
`queue_name` `(IN VARCHAR2)`	specifies the name of the non-persistent queue that is to be created. The name must be unique within a schema and must follow object name guidelines in the *Oracle8i SQL Reference* with regard to reserved characters.
`multiple_consumers` `(IN BOOLEAN)`	`FALSE`: Queues created in the table can only have one consumer per message. This is the default. `TRUE`: Queues created in the table can have multiple consumers per message. Note that the `multi_consumers` parameter is distinguished at the queue level because a non-persistent queue does not inherit this characteristic from any user-created queue table
`comment` `(IN VARCHAR2)`	User-specified description of the queue. This user comment will be added to the queue catalog.

Usage Notes

- The queue may be either single-consumer or multiconsumer queue. All queue names must be unique within a schema. The queues are created in a 8.1-compatible system-created queue table (`AQ$_MEM_SC` or `AQ$_MEM_MC`) in the same schema as that specified by the queue name. If the queue name does not specify a schema name, the queue is created in the login user's schema. Once a queue is created with `CREATE_NP_QUEUE`, it can be enabled by calling `START_QUEUE`. By default, the queue is created with both enqueue and dequeue disabled.
- You cannot dequeue from a non-persistent queue. The only way to retrieve a message from a non-persistent queue is by using the OCI notification mechanism (see Register for Notification on page 6-50).
- You cannot invoke the `listen` call on a non-persistent queue (see Listen to One (Many) Queue(s) on page 6-18).
- You cannot have rule based subscriptions on non-persistent queues.

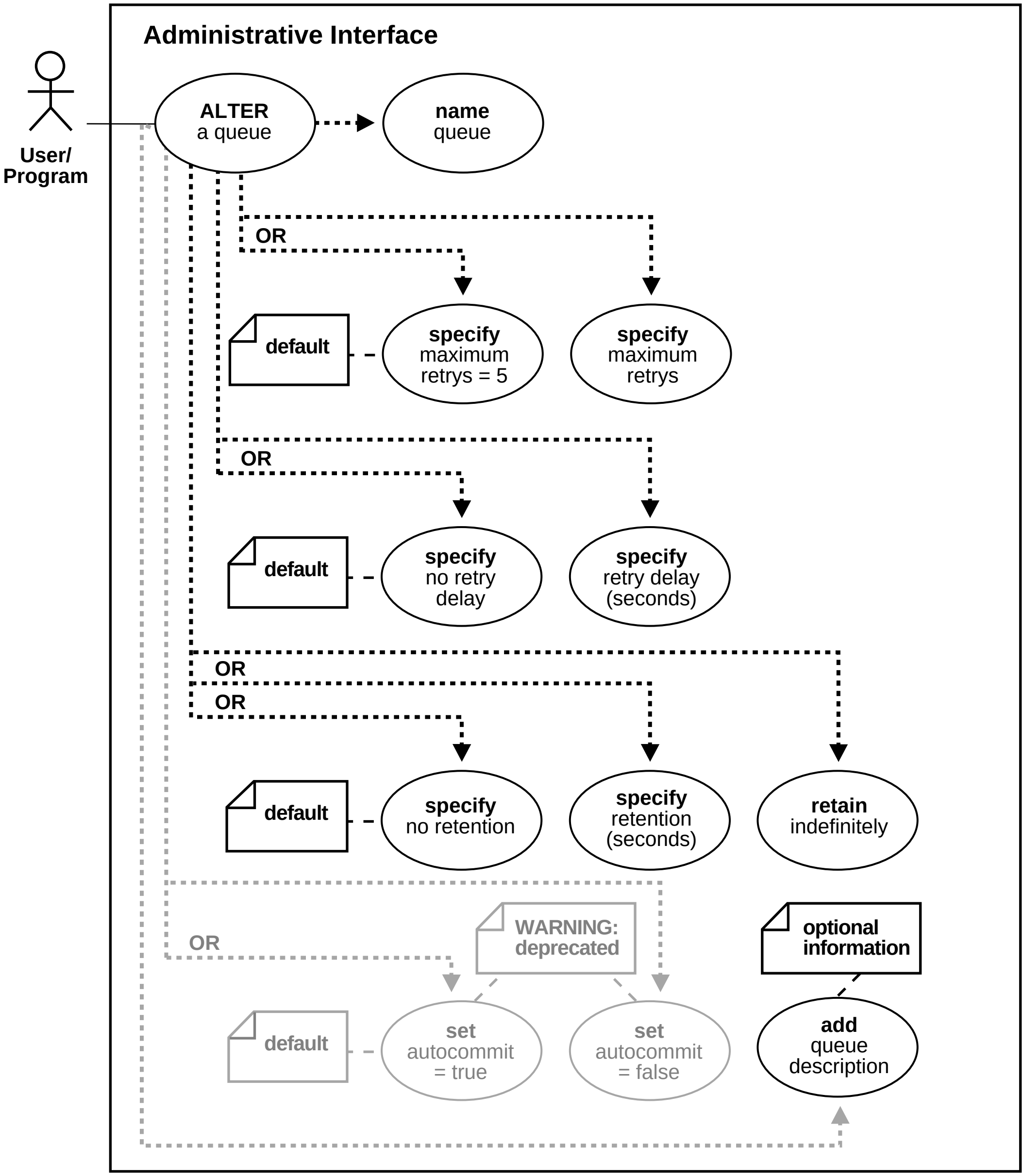
Example: Create a Non-Persistent Queue Using PL/SQL (DBMS_AQADM)

```
/* Create a non-persistent single-consumer queue (Note: this is not preceded by
```

```
   creation of a queue table) */
EXECUTE dbms_aqadm.create_np_queue(
   Queue_name          => 'Singleconsumersmsg_npque',
   Multiple_consumers   => FALSE);

/* Create a non-persistent multi-consumer queue (Note: this is not preceded by
   creation of a queue table) */
EXECUTE dbms_aqadm.create_np_queue(
   Queue_name          => 'Multiconsumersmsg_npque',
   Multiple_consumers   => TRUE);
```

Alter a Queue

Figure 4–8 Use Case Diagram: Alter a Queue



To refer to the table of all basic operations having to do with the Administrative Interface see:

- "Use Case Model: Administrative Interface — Basic Operations" on page 4-2

Purpose:

Alter existing properties of a queue. Only max_retries, retry_delay, and retention_time can be altered.

Syntax:

```
DBMS_AQADM.ALTER_QUEUE (
   queue_name        IN    VARCHAR2,
   max_retries       IN    NUMBER default NULL,
   retry_delay       IN    NUMBER default NULL,
   retention_time    IN    NUMBER default NULL,
   auto_commit       IN    BOOLEAN default TRUE,
   comment           IN    VARCHAR2 default NULL);
```

Usage:

Table 4–7 DBMS_AQADM.ALTER_QUEUE

Parameter	Description
`queue_name` `(IN VARCHAR2)`	specifies the name of the queue that is to be altered.
`max_retries` `(IN NUMBER)`	Limits the number of times a dequeue with `REMOVE` mode can be attempted on a message. The count is incremented when the application issues a rollback after executing the dequeue. If the time at which one of the retries has passed the expiration time, no further retries will be attempted. The default is `NULL` which means that existing value will not be changed. Note that `max_retries` is supported for all single consumer queues and 8.1-compatible multiconsumer queues but not for 8.0-compatible multiconsumer queues.
`retry_delay` `(IN NUMBER)`	specifies the delay time in seconds before this message is scheduled for processing again after an application rollback. The default is `NULL` which means that existing value will not be changed. Note that `retry_delay` is supported for single consumer queues and 8.1-compatible multiconsumer queues but not for 8.0-compatible multiconsumer queues.

Table 4–7 DBMS_AQADM.ALTER_QUEUE

Parameter	Description
`retention_time` `(IN NUMBER)`	specifies the retention time in seconds for which a message will be retained in the queue table after being dequeued. The default is `NULL` which means that the value will not be altered.
`auto_commit` `(IN BOOLEAN)`	`TRUE`: Causes the current transaction, if any, to commit before the `ALTER_QUEUE` operation is carried out. The `ALTER_QUEUE` operation become persistent when the call returns. This is the default. `FALSE`: The operation is part of the current transaction and will become persistent only when the caller issues a commit. Caution: This parameter has been deprecated.
`comment` `(IN VARCHAR2)`	User-specified description of the queue. This user comment will be added to the queue catalog. The default value is `NULL` which means that the value will not be changed.

Usage Notes

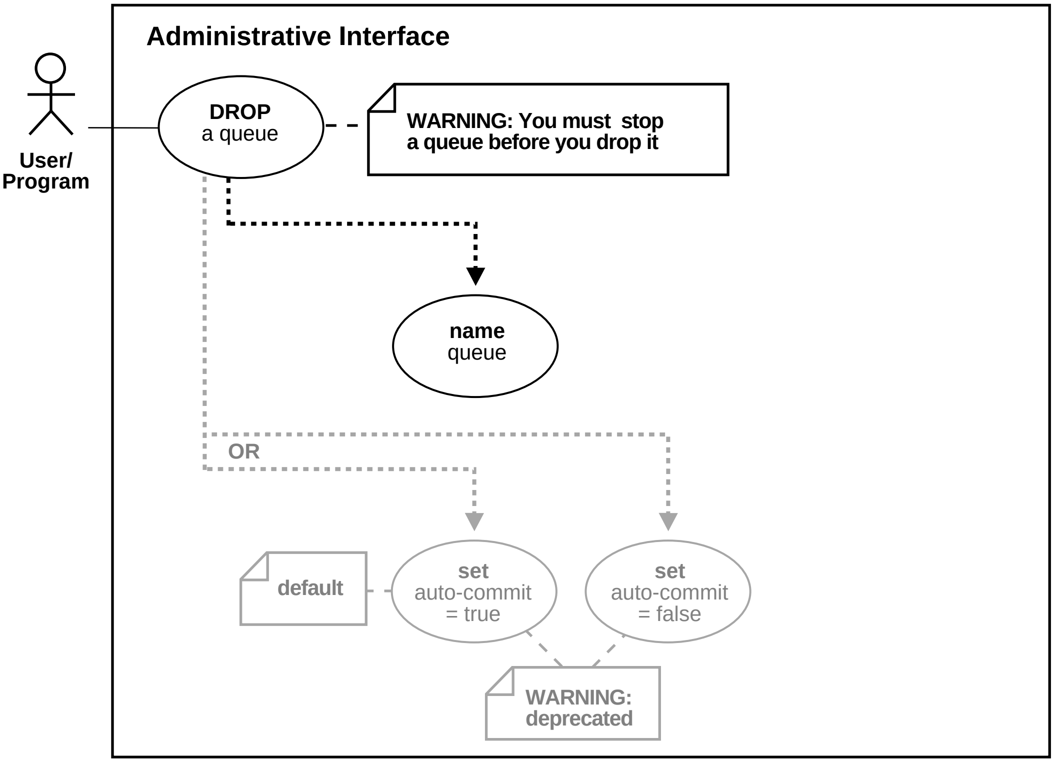
- To view retained messages, you can either dequeue by message ID or use SQL.
- You can only alter the comment field of a non-persistent queues.
- Note that `max_retries`, `retention`, `retry_delay` and `retry_count` are not supported for non-persistent queues.

Example: Alter a Queue Using PL/SQL (DBMS_AQADM)

```
/* Alter queue to change retention time, saving messages for 1 day after
   dequeueing:  */
EXECUTE dbms_aqadm.alter_queue (
   queue_name        => 'aq.Anothermsg_queue',
   retention_time    => 86400);
```

Drop a Queue

Figure 4–9 Use Case Diagram: Drop a Queue



To refer to the table of all basic operations having to do with the Administrative Interface see:

- "Use Case Model: Administrative Interface — Basic Operations" on page 4-2

Purpose:

Drops an existing queue. `DROP_QUEUE` is not allowed unless `STOP_QUEUE` has been called to disable the queue for both enqueuing and dequeuing. All the queue data is deleted as part of the drop operation.

Syntax:

```
DBMS_AQADM.DROP_QUEUE (
   queue_name       IN    VARCHAR2,
   auto_commit      IN    BOOLEAN default TRUE);
```

Usage:

Table 4–8 DBMS_AQADM.DROP_QUEUE

Parameter	Description
`queue_name` `(IN VARCHAR2)`	specifies the name of the queue that is to be dropped.
`auto_commit` `(IN BOOLEAN)`	`TRUE`: Causes the current transaction, if any, to commit before the `DROP_QUEUE` operation is carried out. The `DROP_QUEUE` operation becomes persistent when the call returns. This is the default. `FALSE`: The operation is part of the current transaction and will become persistent only when the caller issues a commit. Caution: This parameter has been deprecated.

Example: Drop a Queue Using PL/SQL (DBMS_AQADM)

Drop a Standard Queue

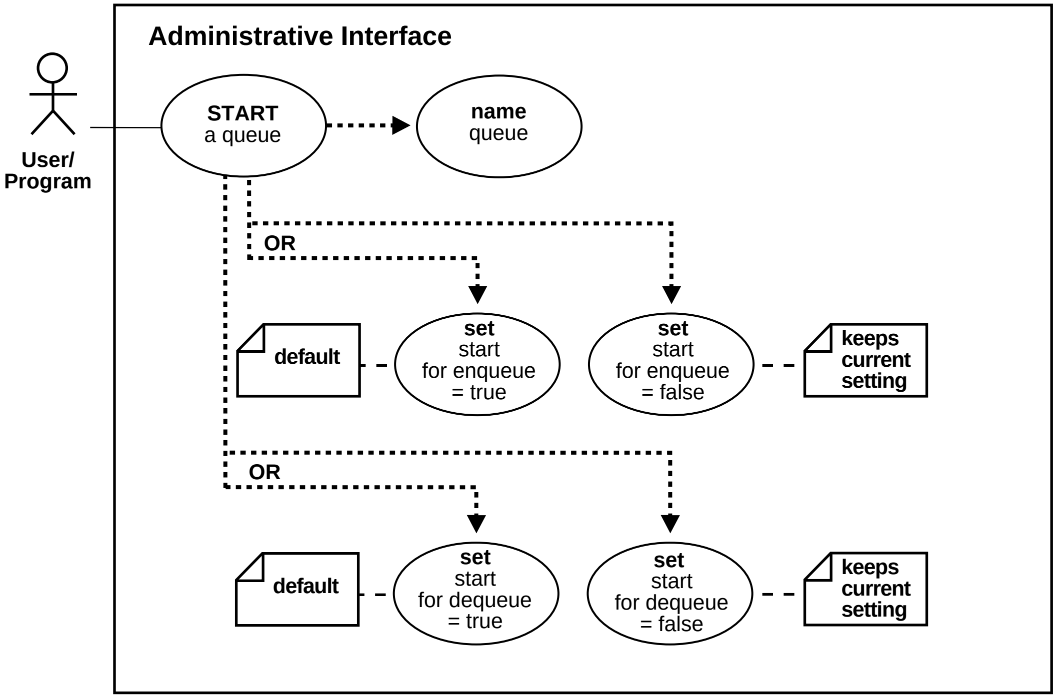
```
/* Stop the queue preparatory to dropping it (a queue may be dropped only after
   it has been succesfully stopped for enqueing and dequeing): */
EXECUTE dbms_aqadm.stop_queue (
   Queue_name       => 'aq.Msg_queue');

/* Drop queue: */
EXECUTE dbms_aqadm.drop_queue (
   Queue_name         => 'aq.Msg_queue');
```

Drop a Non-Persistent Queue

```
EXECUTE DBMS_AQADM.DROP_QUEUE( queue_name => 'Nonpersistent_
singleconsumerq1');
EXECUTE DBMS_AQADM.DROP_QUEUE( queue_name => 'Nonpersistent_multiconsumerq1');
```

Start a Queue

Figure 4–10 Use Case Diagram: Start a Queue



To refer to the table of all basic operations having to do with the Administrative Interface see:

- "Use Case Model: Administrative Interface — Basic Operations" on page 4-2

Purpose:

Enables the specified queue for enqueuing and/or dequeueing.

Syntax:

```
DBMS_AQADM.START_QUEUE (
   queue_name       IN      VARCHAR2,
```

```
   enqueue        IN     BOOLEAN default TRUE,
   dequeue        IN     BOOLEAN default TRUE)
```

Usage:

Table 4–9 DBMS_AQADM.START_QUEUE

Parameter	Description
`queue_name` `(IN VARCHAR2)`	specifies the name of the queue to be enabled.
`enqueue` `(IN BOOLEAN)`	specifies whether `ENQUEUE` should be enabled on this queue. `TRUE`: Enable `ENQUEUE`. This is the default. `FALSE`: Do not alter the current setting.
`dequeue` `(IN BOOLEAN)`	specifies whether `DEQUEUE` should be enabled on this queue. `TRUE`: Enable `DEQUEUE`. This is the default. `FALSE`: Do not alter the current setting.

Usage Notes

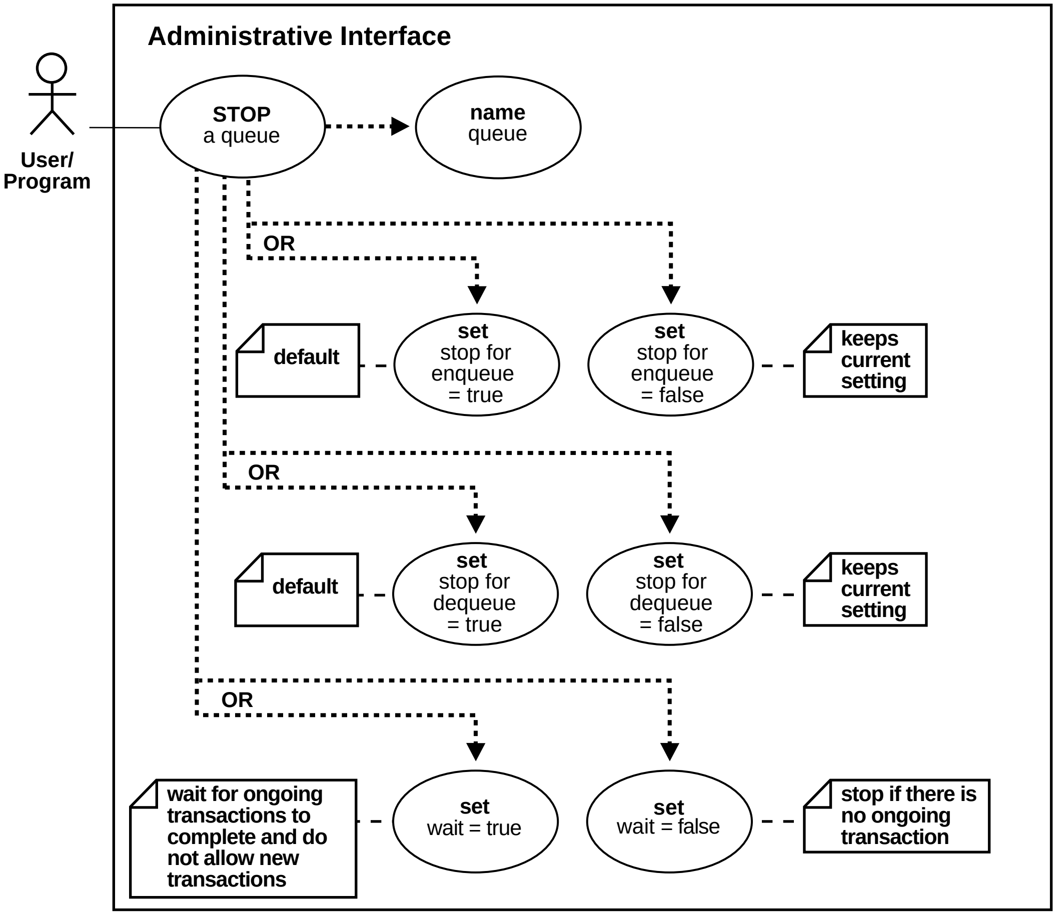
After creating a queue the administrator must use `START_QUEUE` to enable the queue. The default is to enable it for both `ENQUEUE` and `DEQUEUE`. Only dequeue operations are allowed on an exception queue. This operation takes effect when the call completes and does not have any transactional characteristics.

Example: Start a Queue using PL/SQL (DBMS_AQADM Package)

```
/* Start a queue and enable both enqueue and dequeue: */
EXECUTE dbms_aqadm.start_queue (
   queue_name         => 'Msg_queue');

/* Start a previously stopped queue for dequeue only */
EXECUTE dbms_aqadm.start_queue (
   queue_name         => 'aq.msg_queue',
   dequeue            => TRUE,
   enqueue            => FALSE);
```

Stop a Queue

Figure 4–11 Use Case Diagram: Stop a Queue



To refer to the table of all basic operations having to do with the Administrative Interface see:

- "Use Case Model: Administrative Interface — Basic Operations" on page 4-2

Purpose:

Disables enqueuing and/or dequeuing on the specified queue.

Syntax:

```
DBMS_AQADM.STOP_QUEUE (
   queue_name      IN   VARCHAR2,
   enqueue         IN   BOOLEAN default TRUE,
   dequeue         IN   BOOLEAN default TRUE,
   wait            IN   BOOLEAN default TRUE);
```

Usage:

Table 4–10 DBMS_AQADM.STOP_QUEUE

Parameter	Description
`queue_name` `(IN VARCHAR2)`	specifies the name of the queue to be disabled.
`enqueue` `(IN BOOLEAN)`	specifies whether `ENQUEUE` should be disabled on this queue. `TRUE`: Disable `ENQUEUE`. This is the default. `FALSE`: Do not alter the current setting.
`dequeue` `(IN BOOLEAN)`	specifies whether `DEQUEUE` should be disabled on this queue. `TRUE`: Disable `DEQUEUE`. This is the default. `FALSE`: Do not alter the current setting.
`wait` `(IN BOOLEAN)`	The *wait* parameter allows you to specify whether to wait for the completion of outstanding transactions. `TRUE`: Wait if there are any outstanding transactions. In this state no new transactions are allowed to enqueue to or dequeue from this queue. `FALSE`: Return immediately either with a success or an error.

Usage Notes

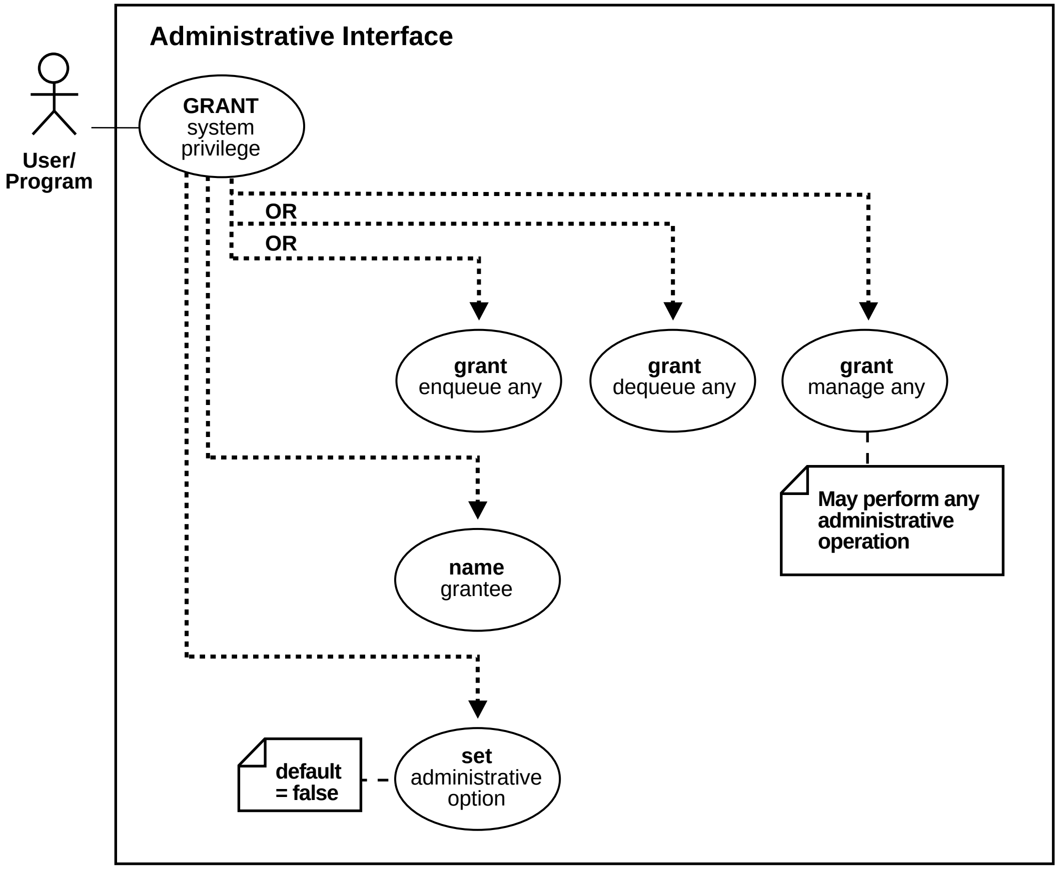
By default, this call disables both `ENQUEUE`s or `DEQUEUE`s. A queue cannot be stopped if there are outstanding transactions against the queue. This operation takes effect when the call completes and does not have any transactional characteristics.

Example: Stop a Queue Using PL/SQL (DBMS_AQADM)

```
/* Stop the queue: */
EXECUTE dbms_aqadm.stop_queue (
   queue_name        => 'aq.Msg_queue');
```

Grant System Privilege

Figure 4–12 Use Case Diagram: Grant System Privilege



To refer to the table of all basic operations having to do with the Administrative Interface see:

- "Use Case Model: Administrative Interface — Basic Operations" on page 4-2

Purpose:

To grant AQ system privileges to users and roles. The privileges are `ENQUEUE_ANY`, `DEQUEUE_ANY`, `MANAGE_ANY`. Initially, only `SYS` and `SYSTEM` can use this procedure successfully.

Syntax:

```
DBMS_AQADM.GRANT_SYSTEM_PRIVILEGE(
   privilege         IN    VARCHAR2,
   grantee           IN    VARCHAR2,
   admin_option      IN    BOOLEAN := FALSE);
```

Usage:

Table 4–11 DBMS_AQADM.GRANT_SYSTEM_PRIVILEGE

Parameter	Description
`privilege` `(IN VARCHAR2)`	specifies the AQ system privilege to grant. Options are: `ENQUEUE_ANY`, `DEQUEUE_ANY`, `MANAGE_ANY`. The operations allowed for each system privilege are specified as follows: `ENQUEUE_ANY`: users granted with this privilege are allowed to enqueue messages to any queues in the database. `DEQUEUE_ANY`: users granted with this privilege are allowed to dequeue messages from any queues in the database. `MANAGE_ANY`: users granted with this privilege are allowed to execute `DBMS_AQADM` calls on any schemas in the database.
grantee (`IN VARCHAR2`)	specifies the grantee(s). The grantee(s) can be a user, a role, or the `PUBLIC` role.
admin_option (`IN BOOLEAN`)	specifies if the system privilege is granted with the ADMIN option or not. If the privilege is granted with the `ADMIN` option, the grantee is allowed to use this procedure to grant the system privilege to other users or roles. Default:`FALSE`

Example: Grant System Privilege Using PL/SQL (DBMS_AQADM)

```
/* User AQADM grants the rights to enqueue and dequeue to ANY queues: */
```

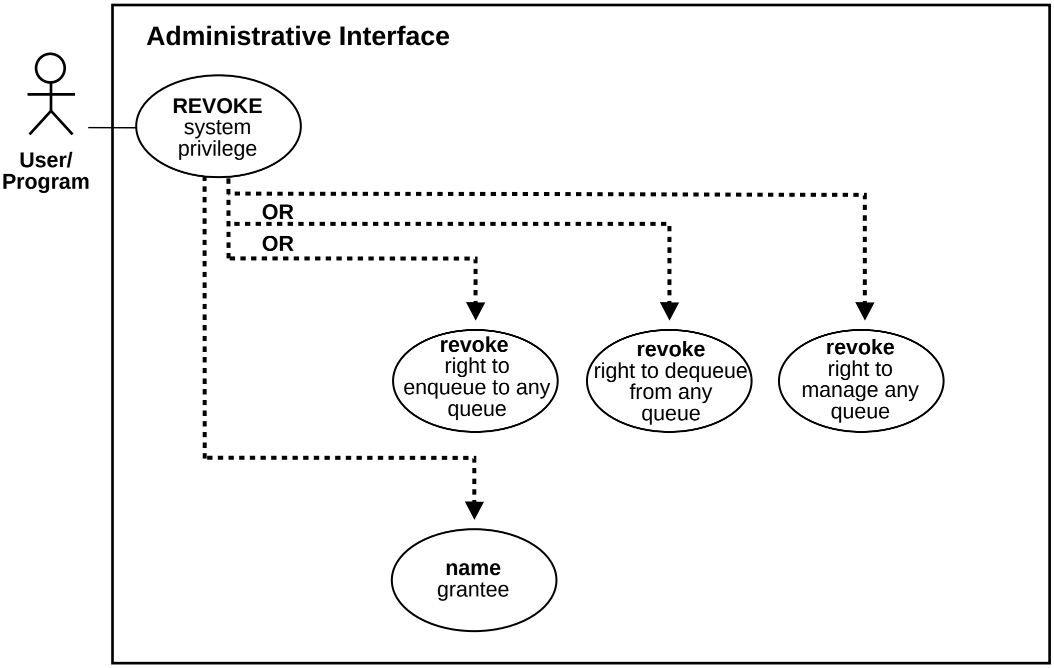
Note: You may need to set up the following data structures for certain examples to work:

```
CONNECT system/manager;
CREATE USER aqadm IDENTIFIED BY aqadm;
GRANT CONNECT, RESOURCE TO aqadm;
GRANT EXECUTE ON DBMS_AQADM TO aqadm;
GRANT Aq_administrator_role TO aqadm;
```

```
CONNECT aqadm/aqadm;
EXECUTE DBMS_AQADM.GRANT_SYSTEM_PRIVILEGE(
   privilege         =>    'ENQUEUE_ANY',
   grantee           =>    'Jones',
   admin_option      =>     FALSE);
EXECUTE DBMS_AQADM.GRANT_SYSTEM_PRIVILEGE(
   privilege         =>     'DEQUEUE_ANY',
   grantee           =>     'Jones',
   admin_option      =>      FALSE);
```

Revoke System Privilege

Figure 4–13 Use Case Diagram: Revoke System Privilege



To refer to the table of all basic operations having to do with the Administrative Interface see:

- "Use Case Model: Administrative Interface — Basic Operations" on page 4-2

Purpose:

To revoke AQ system privileges from users and roles. The privileges are `ENQUEUE_ANY`, `DEQUEUE_ANY` and `MANAGE_ANY`. The `ADMIN` option for a system privilege cannot be selectively revoked.

Syntax:

```
DBMS_AQADM.REVOKE_SYSTEM_PRIVILEGE(
```

```
privilege         IN   VARCHAR2,
grantee           IN   VARCHAR2);
```

Usage:

Table 4–12 DBMS_AQADM.REVOKE_SYSTEM_PRIVILEGE

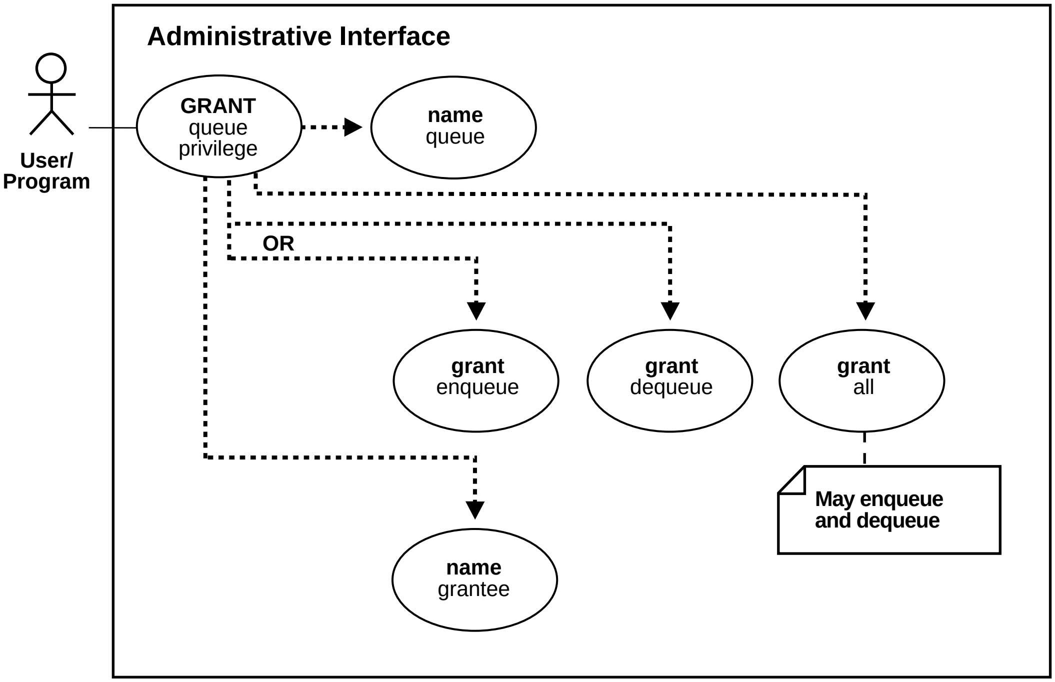
Parameter	Description
`privilege` `(IN VARCHAR2)`	specifies the AQ system privilege to revoke. Options are: `ENQUEUE_ANY`, `DEQUEUE_ANY`, `MANAGE_ANY`. The `ADMIN` option for a system privilege cannot be selectively revoked.
grantee (`IN VARCHAR2`)	specifies the grantee(s). The grantee(s) can be a user, a role, or the `PUBLIC` role.

Example: Revoke System Privilege Using PL/SQL (DBMS_AQADM)

```
/* To revoke the DEQUEUE_ANY system privilege from Jones. */
CONNECT system/manager;
   execute DBMS_AQADM.REVOKE_SYSTEM_PRIVILEGE(privilege=>'DEQUEUE_ANY',
                                  grantee=>'Jones');
```

Grant Queue Privilege

Figure 4–14 Use Case Diagram: Grant Queue Privilege



To refer to the table of all basic operations having to do with the Administrative Interface see:

- "Use Case Model: Administrative Interface — Basic Operations" on page 4-2

Purpose:

To grant privileges on a queue to users and roles. The privileges are `ENQUEUE` or `DEQUEUE`. Initially, only the queue table owner can use this procedure to grant privileges on the queues.

Syntax:

```
DBMS_AQADM.GRANT_QUEUE_PRIVILEGE(
   privilege        IN    VARCHAR2,
   queue_name       IN    VARCHAR2,
   grantee          IN    VARCHAR2,
   grant_option     IN    BOOLEAN := FALSE);
```

Usage:

Table 4–13 DBMS_AQADM.GRANT_QUEUE_PRIVILEGE

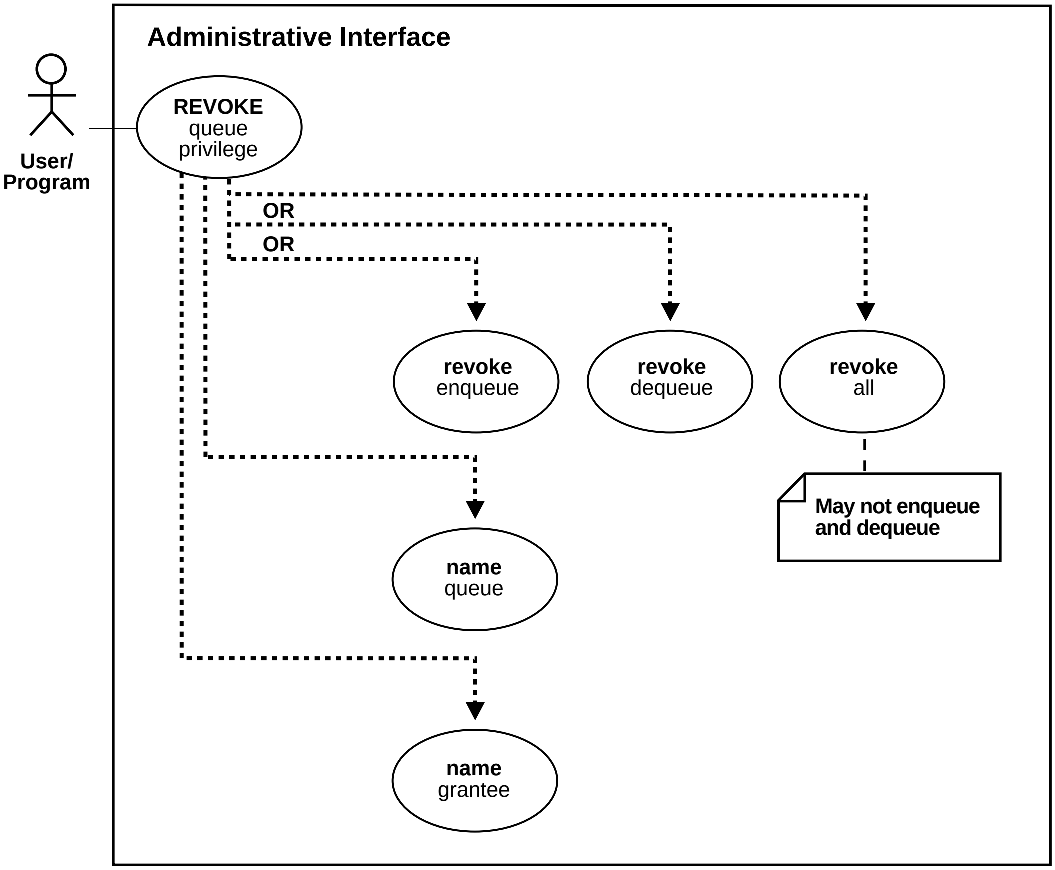
Parameter	Description
`privilege` `(IN VARCHAR2)`	specifies the AQ queue privilege to grant. Options are: `ENQUEUE`, `DEQUEUE` and `ALL`. `ALL` means both `ENQUEUE` and `DEQUEUE`.
`queue_name` `(IN VARCHAR2)`	specifies the name of the queue.
`grantee` `(IN VARCHAR2)`	specifies the grantee(s). The grantee(s) can be a user, a role, or the `PUBLIC` role.
`grant_option` `(IN BOOLEAN)`	specifies if the access privilege is granted with the `GRANT` option or not. If the privilege is granted with the `GRANT` option, the grantee is allowed to use this procedure to grant the access privilege to other users or roles, regardless of the ownership of the queue table. Default:`FALSE`

Example: Grant Queue Privilege Using PL/SQL (DBMS_AQADM)

```
/* User grants the access right for both enqueue and dequeue rights using
   DBMS_AQADM.GRANT. */
EXECUTE DBMS_AQADM.GRANT_QUEUE_PRIVILEGE (
   privilege    =>     'ALL',
   queue_name   =>     'aq.multiconsumermsg81_queue',
   grantee      =>     'Jones',
   grant_option =>      TRUE);
```

Revoke Queue Privilege

Figure 4–15 Use Case Diagram: Revoke Queue Privilege



To refer to the table of all basic operations having to do with the Administrative Interface see:

- "Use Case Model: Administrative Interface — Basic Operations" on page 4-2

Purpose:

To revoke privileges on a queue from users and roles. The privileges are `ENQUEUE` or `DEQUEUE`.

Syntax:

```
DBMS_AQADM.REVOKE_QUEUE_PRIVILEGE(
   privilege        IN      VARCHAR2,
   queue_name       IN      VARCHAR2,
   grantee          IN      VARCHAR2);
```

Usage:

Table 4–14 DBMS_AQADM.REVOKE_QUEUE_PRIVILEGE

Parameter	Description
`privilege` `(IN VARCHAR2)`	specifies the AQ queue privilege to revoke. Options are: `ENQUEUE`, `DEQUEUE` and `ALL`. `ALL` means both `ENQUEUE` and `DEQUEUE`.
`queue_name` `(IN VARCHAR2)`	specifies the name of the queue.
`grantee` `(IN VARCHAR2)`	specifies the grantee(s). The grantee(s) can be a user, a role, or the `PUBLIC` role. If the privilege has been propagated by the grantee through the `GRANT` option, the propagated privilege is also revoked.

Usage Notes

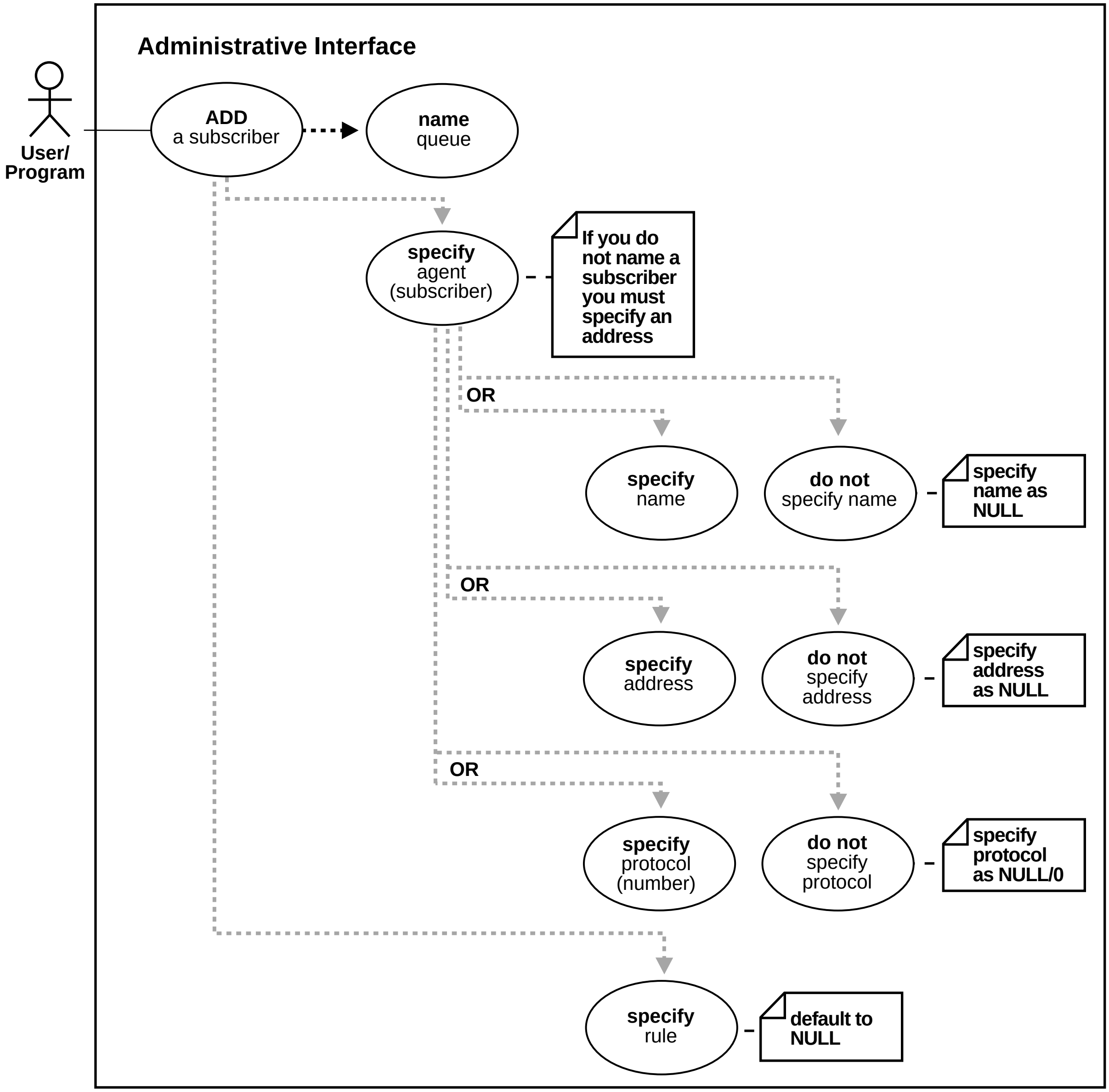
To revoke a privilege, the revoker must be the original grantor of the privilege. The privileges propagated through the `GRANT` option are revoked if the grantor's privileges are revoked.

Example: Revoke Queue Privilege Using PL/SQL (DBMS_AQADM)

```
/* User can revoke the dequeue right of a grantee on a specific queue
   leaving the grantee with only the enqueue right: */
CONNECT scott/tiger;
EXECUTE DBMS_AQADM.REVOKE_QUEUE_PRIVILEGE(
   privilege     =>     'DEQUEUE',
   queue_name    =>     'scott.ScottMsgs_queue',
   grantee       =>     'Jones');
```

Add a Subscriber

Figure 4–16 Use Case Diagram: Add a Subscriber



To refer to the table of all basic operations having to do with the Administrative Interface see:

- "Use Case Model: Administrative Interface — Basic Operations" on page 4-2

Purpose:

Adds a default subscriber to a queue.

Syntax:

```
DBMS_AQADM.ADD_SUBSCRIBER(
   queue_name     IN    VARCHAR2,
   subscriber     IN    aq$_agent,
   rule           IN    VARCHAR2 default NULL);
```

Usage:

Table 4–15 DBMS_AQADM.ADD_SUBSCRIBER

Parameter	Description
`queue_name` `(IN VARCHAR2)`	specifies the name of the queue.
`subscriber` `(IN aq$_agent)`	The agent on whose behalf the subscription is being defined (see definition of "Agent" on page 3-5).
`rule` `(IN VARCHAR2)`	A conditional expression based on the message properties, the message data properties and PL/SQL functions. A rule is specified as a boolean expression using syntax similar to the `WHERE` clause of a SQL query. This boolean expression can include conditions on message properties, user data properties (object payloads only) and PL/SQL or SQL functions (as specified in the where clause of a SQL query). Currently supported message properties are `priority` and `corrid`. To specify rules on a message payload (object payload), use attributes of the object type in clauses. You must prefix each attribute with `tab.user_data` as a qualifier to indicate the specific column of the queue table that stores the payload. The rule parameter cannot exceed 4000 characters.

Usage Note:

- A program can enqueue messages to a specific list of recipients or to the default list of subscribers. This operation will only succeed on queues that allow multiple consumers. This operation takes effect immediately and the containing transaction is committed. Enqueue requests that are executed after the completion of this call will reflect the new behavior.

- Note that any string within the rule has to be quoted as shown below;

  ```
  rule   => 'PRIORITY <= 3 AND CORRID =  ''FROM JAPAN'''
  ```

  Note that these are all single quotation marks.

Example: Add Subscriber Using PL/SQL (DBMS_AQADM)

```
/* Anonymous PL/SQL block for adding a subscriber at a designated queue in a
designated schema at a database link: */
DECLARE
   subscriber          aq$_agent;
BEGIN
   subscriber := aq$_agent ('subscriber1', 'aq2.msg_queue2@london', null);
   DBMS_AQADM.ADD_SUBSCRIBER(
      queue_name          => 'aq.multi_queue',
      subscriber          =>  subscriber);
 END;

/* Add a subscriber with a rule: */
DECLARE
   subscriber       aq$_agent;
BEGIN
   subscriber := aq$_agent('subscriber2', 'aq2.msg_queue2@london', null);
   DBMS_AQADM.ADD_SUBSCRIBER(
      queue_name          =>  'aq.multi_queue',
      subscriber          =>   subscriber,
      rule                =>   'priority < 2');
END;
```

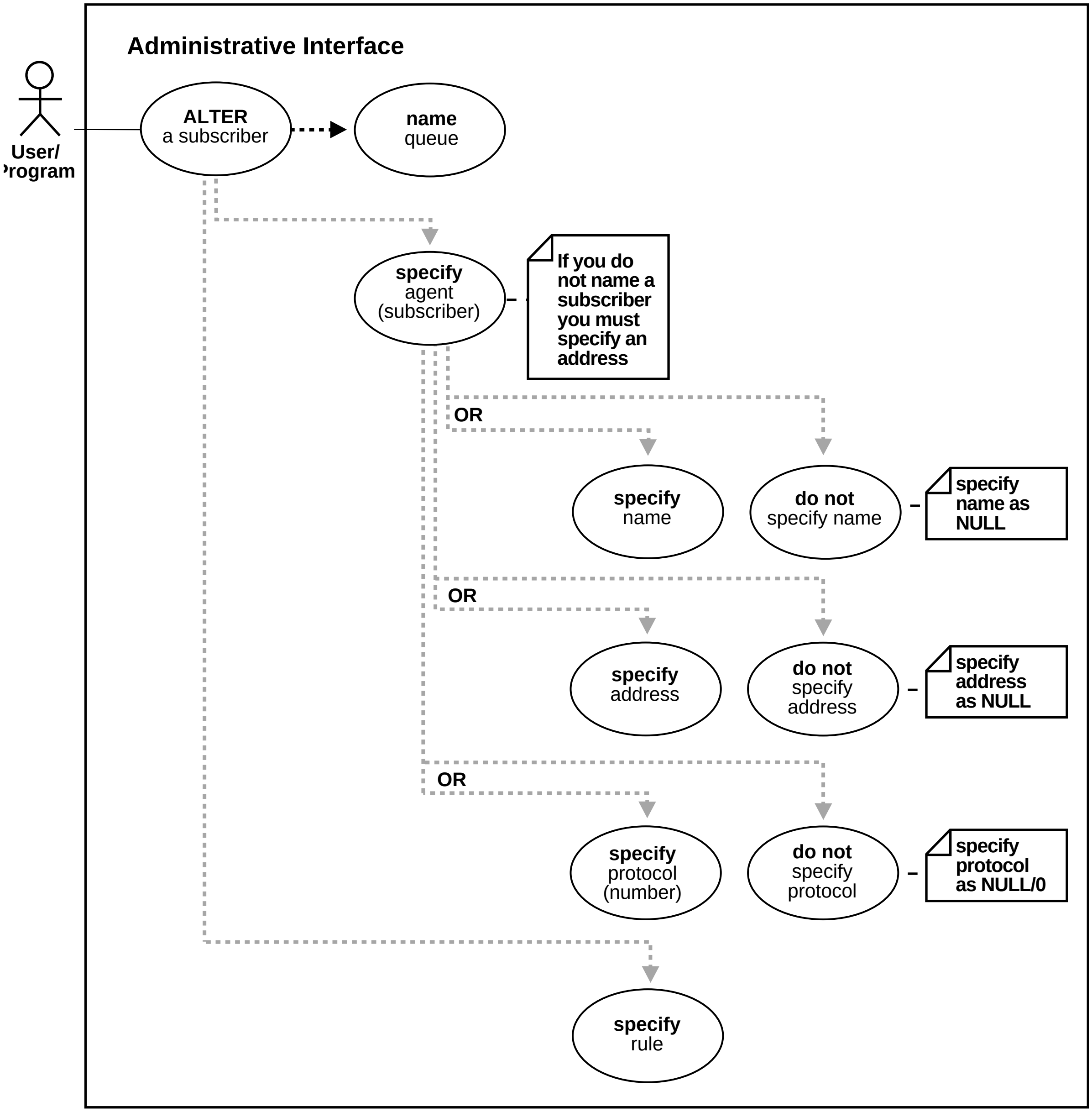
Example: Add Rule-Based Subscriber Using PL/SQL (DBMS_AQADM)

```
DECLARE
   subscriber          aq$_agent;
BEGIN
   subscriber :=  aq$_agent('East_Shipping','ES.ES_bookedorders_que',null);
   DBMS_AQADM.ADD_SUBSCRIBER(
```

```
          queue_name             => 'OE.OE_bookedorders_que',
          subscriber             => subscriber,
          rule                   => 'tab.user_data.orderregion =  ''EASTERN'' OR
                                    (tab.user_data.ordertype =  ''RUSH'' AND
                                     tab.user_data.customer.country = ''USA'') ');
   END;
```

Alter a Subscriber

Figure 4–17 Use Case Diagram: Alter a Subscriber



To refer to the table of all basic operations having to do with the Administrative Interface see:

- "Use Case Model: Administrative Interface — Basic Operations" on page 4-2

Purpose:

Alter existing properties of a subscriber to a specified queue. Only the rule can be altered.

Syntax:

```
DBMS_AQADM.ALTER_SUBSCRIBER(
   queue_name     IN    VARCHAR2,
   subscriber     IN    aq$_agent
   rule           IN    VARCHAR2);
```

Usage:

Table 4–16 DBMS_AQADM.ALTER_SUBSCRIBER

Parameter	Description
`queue_name` `(IN VARCHAR2)`	specifies the name of the queue.
`subscriber` `(IN aq$_agent)`	The agent on whose behalf the subscription is being altered (see definition of "Agent" on page 3-5).
rule `(IN VARCHAR2)`	A conditional expression based on the message properties, the message data properties and PL/SQL functions.The rule parameter cannot exceed 4000 characters.To eliminate the rule, set the rule parameter to `NULL`.

Example: Alter Subscriber Using PL/SQL (DBMS_AQADM)

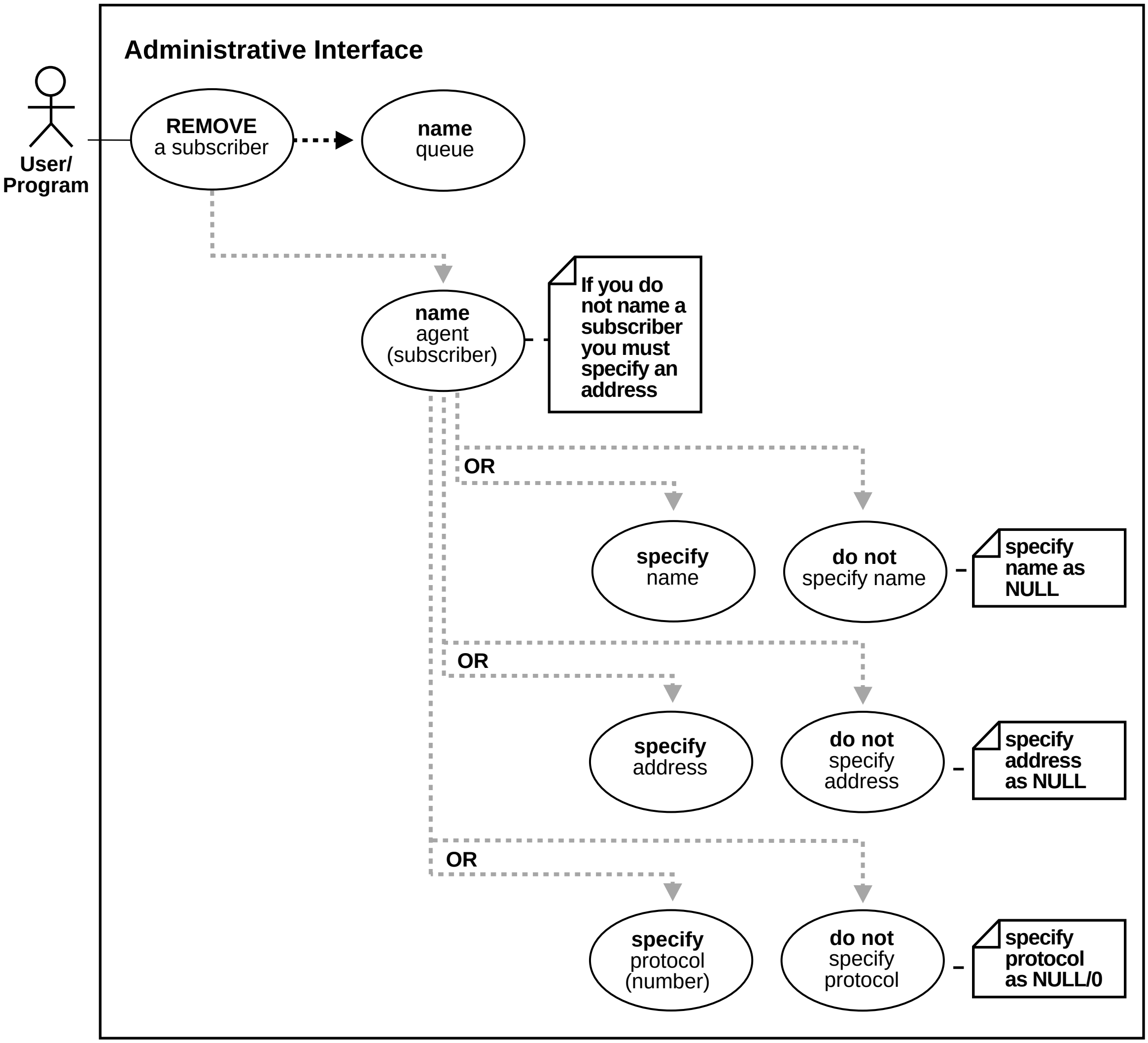
Note: You may need to set up the following data structures for certain examples to work:

```
EXECUTE DBMS_AQADM.CREATE_QUEUE_TABLE (
   queue_table            => 'aq.multi_qtab',
   multiple_consumers     => TRUE,
   queue_payload_type     => 'aq.message_typ',
   compatible             => '8.1.5');
EXECUTE DBMS_AQADM.CREATE_QUEUE (
   queue_name             =>  'multi_queue',
   queue_table            => 'aq.multi_qtab');
```

```
/* Add a subscriber with a rule: */
DECLARE
   subscriber       aq$_agent;
BEGIN
   subscriber := aq$_agent('SUBSCRIBER1', 'aq2.msg_queue2@london', null);
   DBMS_AQADM.ADD_SUBSCRIBER(
      queue_name         =>    'aq.msg_queue',
      subscriber         =>     subscriber,
      rule               =>    'priority < 2');
END;
/* Change rule for subscriber: */
DECLARE
   subscriber          aq$_agent;
BEGIN
   subscriber := aq$_agent('SUBSCRIBER1', 'aq2.msg_queue2@london', null);
   DBMS_AQADM.ALTER_SUBSCRIBER(
      queue_name         =>    'aq.msg_queue',
      subscriber         =>     subscriber,
      rule              =>     'priority = 1');
END;
```

Remove a Subscriber

Figure 4–18 Use Case Diagram: Remove a Subscriber



To refer to the table of all basic operations having to do with the Administrative Interface see:

- "Use Case Model: Administrative Interface — Basic Operations" on page 4-2

Purpose:

Remove a default subscriber from a queue.

Syntax:

```
DBMS_AQADM.REMOVE_SUBSCRIBER(
    queue_name          IN          VARCHAR2,
    subscriber          IN          aq$_agent);
```

Usage:

Table 4–17

Parameter	Description
`queue_name` `(IN VARCHAR2)`	specifies the name of the queue.
`subscriber` `(IN aq$_agent)`	The agent who is being removed from the (see definition of "Agent" on page 3-5).

Usage Notes

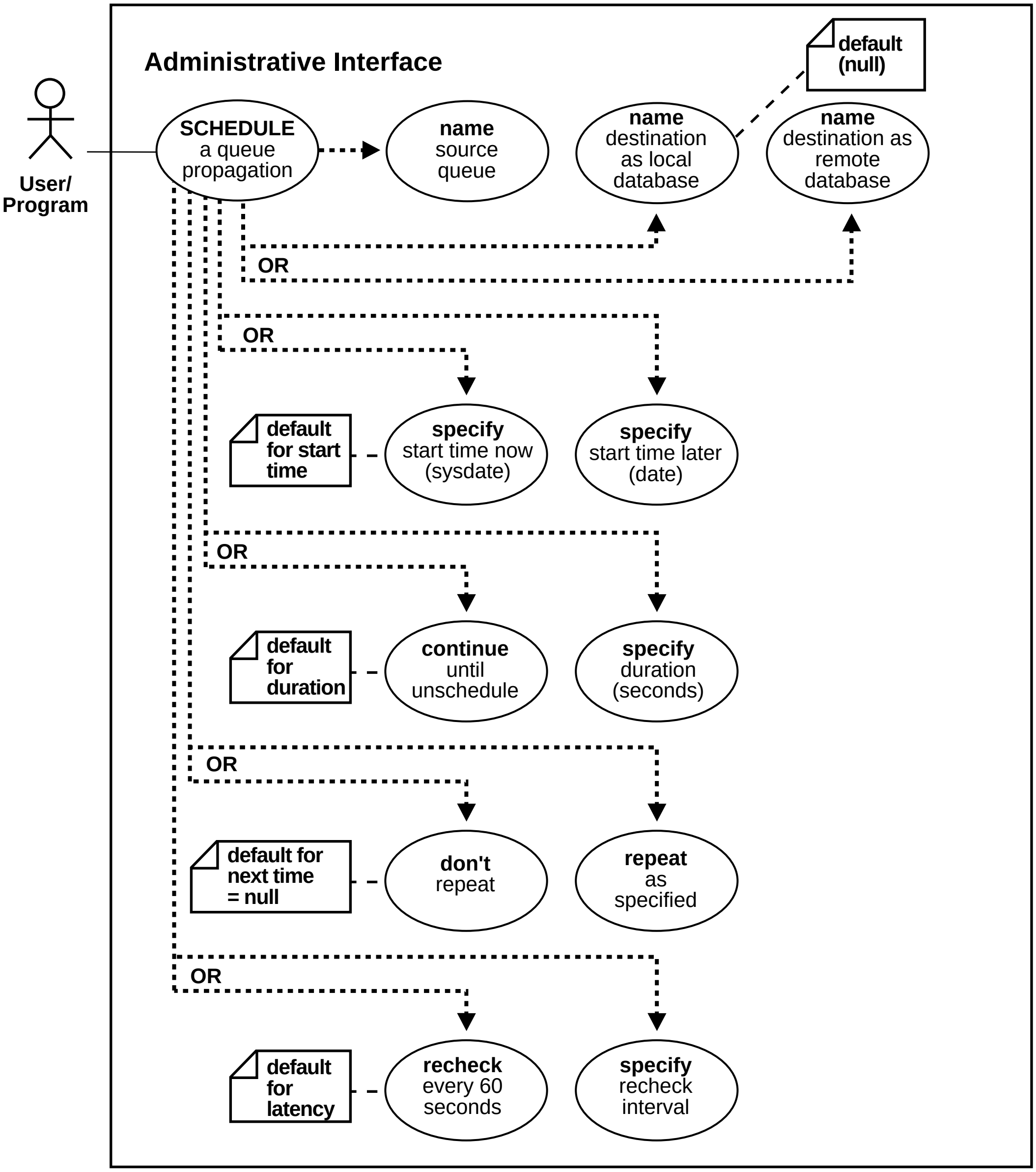
This operation takes effect immediately and the containing transaction is committed. All references to the subscriber in existing messages are removed as part of the operation.

Example: Remove Subscriber Using PL/SQL (DBMS_AQADM)

```
DECLARE
   subscriber       aq$_agent;
BEGIN
   subscriber := aq$_agent('subscriber1','aq2.msg_queue2', NULL);
   DBMS_AQADM.REMOVE_SUBSCRIBER(
      queue_name => 'aq.multi_queue',
      subscriber => subscriber);
END;
```

Schedule a Queue Propagation

Figure 4–19 Use Case Diagram: Schedule a Queue Propagation



To refer to the table of all basic operations having to do with the Administrative Interface see:

- "Use Case Model: Administrative Interface — Basic Operations" on page 4-2

Purpose:

Schedule propagation of messages from a queue to a destination identified by a specific dblink.

Syntax:

```
DBMS_AQADM.SCHEDULE_PROPAGATION(
   queue_name      IN    VARCHAR2,
   destination     IN    VARCHAR2 default NULL,
   start_time      IN    DATE default SYSDATE,
   duration        IN    NUMBER default NULL,
   next_time       IN    VARCHAR2 default NULL,
   latency         IN    NUMBER default 60);
```

Usage:

Table 4–18 DBMS_AQADM.SCHEDULE_PROPAGATION

Parameter	Description
`queue_name` `(IN VARCHAR2)`	specifies the name of the source queue whose messages are to be propagated, including the schema name. If the schema name is not specified, it defaults to the schema name of the administrative user.
`destination` `(IN VARCHAR2)`	specifies the destination dblink. Messages in the source queue for recipients at this destination will be propagated. If it is `NULL`, the destination is the local database and messages will be propagated to other queues in the local database. The length of this field is currently limited to 128 bytes and if the name is not fully qualified the default domain name is used.
`start_time` `(IN DATE)`	specifies the initial start time for the propagation window for messages from the source queue to the destination.

Table 4–18 DBMS_AQADM.SCHEDULE_PROPAGATION

Parameter	Description
`duration` `(IN NUMBER)`	specifies the duration of the propagation window in seconds. A `NULL` value means the propagation window is forever or until the propagation is unscheduled.
`next_time` `(IN VARCHAR2)`	date function to compute the start of the next propagation window from the end of the current window. If this value is NULL, propagation will be stopped at the end of the current window. For example, to start the window at the same time every day, next_time should be specified as 'SYSDATE + 1 - duration/86400'.
`latency` `(IN NUMBER)`	maximum wait, in seconds, in the propagation window for a message to be propagated after it is enqueued. For example, if the latency is 60 seconds, then during the propagation window, if there are no messages to be propagated, messages from that queue for the destination will not be propagated for at least 60 more seconds. It will be at least 60 seconds before the queue will be checked again for messages to be propagated for the specified destination. If the latency is 600, then the queue will not be checked for 10 minutes and if the latency is 0, then a job queue process will be waiting for messages to be enqueued for the destination and as soon as a message is enqueued it will be propagated.

Usage Notes

Messages may also be propagated to other queues in the same database by specifying a `NULL` destination. If a message has multiple recipients at the same destination in either the same or different queues the message will be propagated to all of them at the same time.

Example: Schedule a Propagation Using PL/SQL (DBMS_AQADM)

Note: You may need to set up the following data structures for certain examples to work:

```
EXECUTE DBMS_AQADM.CREATE_QUEUE_TABLE (
   queue_table        => 'aq.objmsgs_qtab',
   queue_payload_type => 'aq.message_typ',
   multiple_consumers => TRUE);
EXECUTE DBMS_AQADM.CREATE_QUEUE (
   queue_name         => 'aq.q1def',
   queue_table        => 'aq.objmsgs_qtab');
```

Schedule a Propagation from a Queue to other Queues in the Same Database

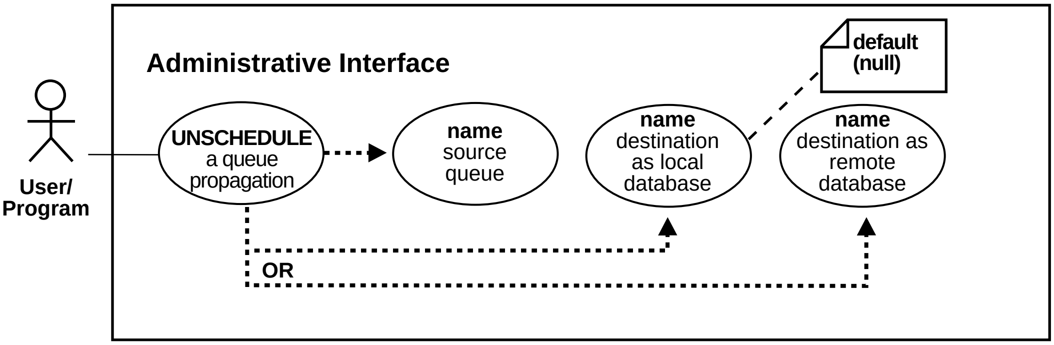
```
/* Schedule propagation from queue aq.q1def to other queues in the same
   database  */
EXECUTE DBMS_AQADM.SCHEDULE_PROPAGATION(
   Queue_name    =>    'aq.q1def');
```

Schedule a Propagation from a Queue to other Queues in Another Database

```
/* Schedule a propagation from queue aq.q1def to other queues in another
   database */
EXECUTE DBMS_AQADM.SCHEDULE_PROPAGATION(
   Queue_name    =>    'aq.q1def',
   Destination   =>    'another_db.world');
```

Unschedule a Queue Propagation

Figure 4–20 Use Case Diagram: Unschedule a Queue Propagation



To refer to the table of all basic operations having to do with the Administrative Interface see:

- "Use Case Model: Administrative Interface — Basic Operations" on page 4-2

Purpose:

Unschedule a previously scheduled propagation of messages from a queue to a destination identified by a specific `dblink`.

Syntax:

```
DBMS_AQADM.UNSCHEDULE_PROPAGATION(
   queue_name        IN   VARCHAR2,
   destination       IN   VARCHAR2 default NULL);
```

Usage:

Table 4–19 DBMS_AQADM.UNSCHEDULE_PROPAGATION

Parameter	Description
`queue_name` `(IN VARCHAR2)`	specifies the name of the source queue whose messages are to be propagated, including the schema name. If the schema name is not specified, it defaults to the schema name of the administrative user.
`destination` `(IN VARCHAR2)`	specifies the destination dblink. Messages in the source queue for recipients at this destination will be propagated. If it is `NULL`, the destination is the local database and messages will be propagated to other queues in the local database. The length of this field is currently limited to 128 bytes and if the name is not fully qualified the default domain name is used.

Example: Unschedule a Propagation Using PL/SQL (DBMS_AQADM)

Unschedule Propagation from Queue To Other Queues in the Same Database

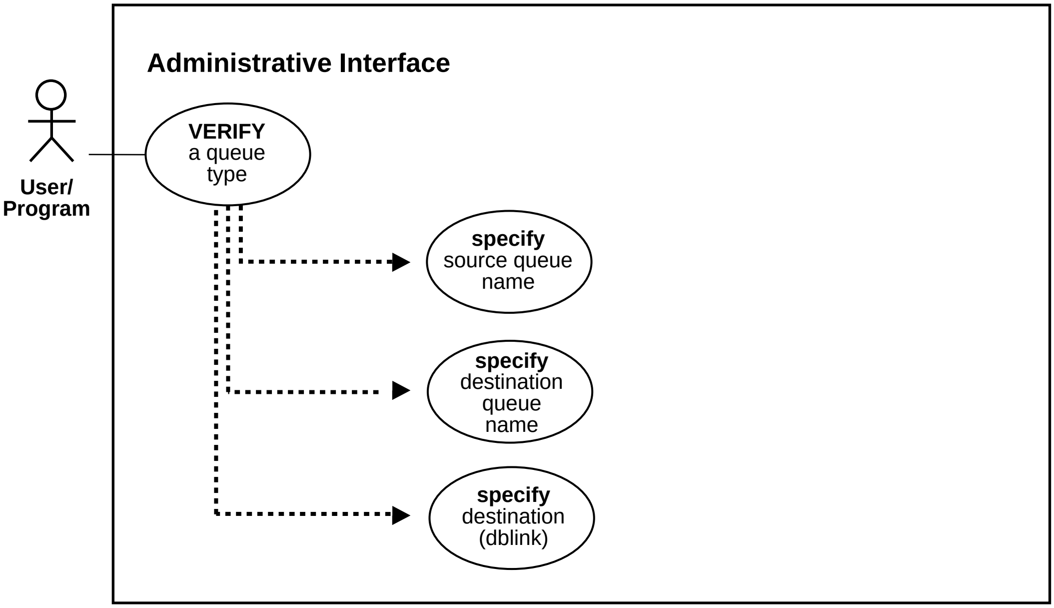
```
/* Unschedule propagation from queue aq.q1def to other queues in the same
   database: */
EXECUTE DBMS_AQADM.UNSCHEDULE_PROPAGATION(queue_name => 'aq.q1def');
```

Unschedule Propagation from a Queue to other Queues in Another Database

```
/* Unschedule propagation from queue aq.q1def to other queues in another
   database reached by the database link another_db.world */
EXECUTE DBMS_AQADM.UNSCHEDULE_PROPAGATION(
   Queue_name    =>   'aq.q1def',
   Destination   =>   'another_db.world');
```

Verify a Queue Type

Figure 4–21 Use Case Diagram: Verify a Queue Type



To refer to the table of all basic operations having to do with the Administrative Interface see:

- "Use Case Model: Administrative Interface — Basic Operations" on page 4-2

Purpose:

Verify that the source and destination queues have identical types. The result of the verification is stored in `aq$_Message_types tables`, overwriting all previous output of this command.

Syntax:

```
DBMS_AQADM.VERIFY_QUEUE_TYPES(
   src_queue_name    IN    VARCHAR2,
   dest_queue_name   IN    VARCHAR2,
```

```
destination       IN    VARCHAR2 default NULL,
rc                OUT   BINARY_INTEGER);
```

Usage:

Table 4–20 DBMS_AQADM.VERIFY_QUEUE_TYPES

Parameter	Description
`src_queue_name` `(IN VARCHAR2)`	specifies the name of the source queue whose messages are to be propagated, including the schema name. If the schema name is not specified, it defaults to the schema name of the user.
`dest_queue_name` `(IN VARCHAR2)`	specifies the name of the destination queue where messages are to be propagated, including the schema name. If the schema name is not specified, it defaults to the schema name of the user.
`destination` `(IN VARCHAR2)`	specifies the destination dblink. The destination queue is in the database that is specified by the dblink. If the destination is `NULL`, the destination queue is the same database as the source queue. The length of this field is currently limited to 128 bytes and if the name is not fully qualified the default domain name is used.
`rc` `(OUT BINARY_ INTEGER)`	return code for the result of the procedure. If there is no error and if the source and destination queue types match the result is 1, if they do not match the result is 0. If an Oracle error is encountered it is returned in rc.

Example: Verify a Queue Type Using PL/SQL (DBMS_AQADM)

Note: You may need to set up the following data structures for certain examples to work:

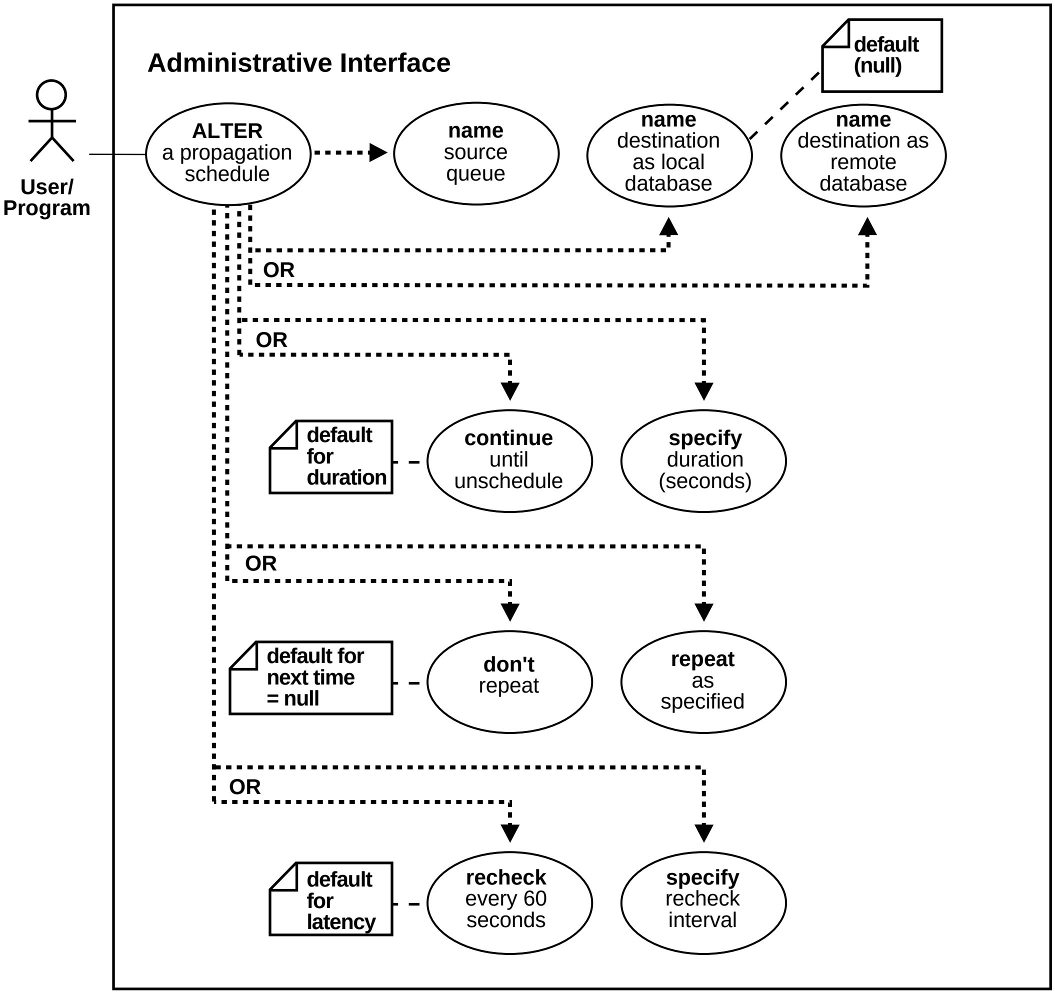
```
EXECUTE DBMS_AQADM.CREATE_QUEUE (
   queue_name         => 'aq.q2def',
   queue_table        => 'aq.objmsgs_qtab');
```

```
/*  Verify if the source and destination queues have the same type. The
    function has the side effect of inserting/updating the entry for the source
    and destination queues in the dictionary table AQ$_MESSAGE_TYPES */
DECLARE
rc      BINARY_INTEGER;
```

```
BEGIN
/* Verify if the queues aq.q1def and aq.q2def in the local database
   have the same payload type */
   DBMS_AQADM.VERIFY_QUEUE_TYPES(
      src_queue_name  => 'aq.q1def',
      dest_queue_name => 'aq.q2def',
      rc              => rc);
   DBMS_OUTPUT.PUT_LINE(rc);
END;
```

Alter a Propagation Schedule

Figure 4–22 Use Case Diagram: Alter a Propagation Schedule



To refer to the table of all basic operations having to do with the Administrative Interface see:

- "Use Case Model: Administrative Interface — Basic Operations" on page 4-2

Purpose:

To alter parameters for a propagation schedule.

Syntax:

```
DBMS_AQADM.ALTER_PROPAGATION_SCHEDULE (
   queue_name          IN     VARCHAR2,
   destination         IN     VARCHAR2 default NULL,
   duration            IN     NUMBER default NULL,
   next_time           IN     VARCHAR2 default NULL,
   latency             IN     NUMBER default 60);
```

Usage:

Table 4–21 DBMS_AQADM.ALTER_PROPAGATION_SCHEDULE

Parameter	Description
`queue_name` `(IN VARCHAR2)`	specifies the name of the source queue whose messages are to be propagated, including the schema name. If the schema name is not specified, it defaults to the schema name of the user.
`destination` `(IN VARCHAR2)`	specifies the destination dblink. The destination queue is in the database that is specified by the dblink. If the destination is `NULL`, the destination queue is the same database as the source queue. The length of this field is currently limited to 128 bytes and if the name is not fully qualified the default domain name is used.

Table 4–21 (Cont.) DBMS_AQADM.ALTER_PROPAGATION_SCHEDULE

Parameter	Description
`duration` `(IN NUMBER)`	specifies the duration of the propagation window in seconds. A `NULL` value means the propagation window is forever or until the propagation is unscheduled.
`next_time` `(IN VARCHAR2)`	the date function to compute the start of the next propagation window from the end of the current window. If this value is NULL, propagation will be stopped at the end of the current window. For example, to start the window at the same time every day, next_time should be specified as ’SYSDATE + 1 - duration/86400’.
`latency` `(IN NUMBER)`	the maximum wait, in seconds, in the propagation window for a message to be propagated after it is enqueued. The default value is 60. Caution: if latency is not specified for this call, latency will over-write any existing value with the default value. For example, if the latency is 60 seconds, then during the propagation window, if there are no messages to be propagated, messages from that queue for the destination will not be propagated for at least 60 more seconds. It will be at least 60 seconds before the queue will be checked again for messages to be propagated for the specified destination. If the latency is 600, then the queue will not be checked for 10 minutes and if the latency is 0, then a job queue process will be waiting for messages to be enqueued for the destination and as soon as a message is enqueued it will be propagated.

Example: Alter a Propagation Schedule Using PL/SQL (DBMS_AQADM)

Alter a Schedule from a Queue to Other Queues in the Same Database

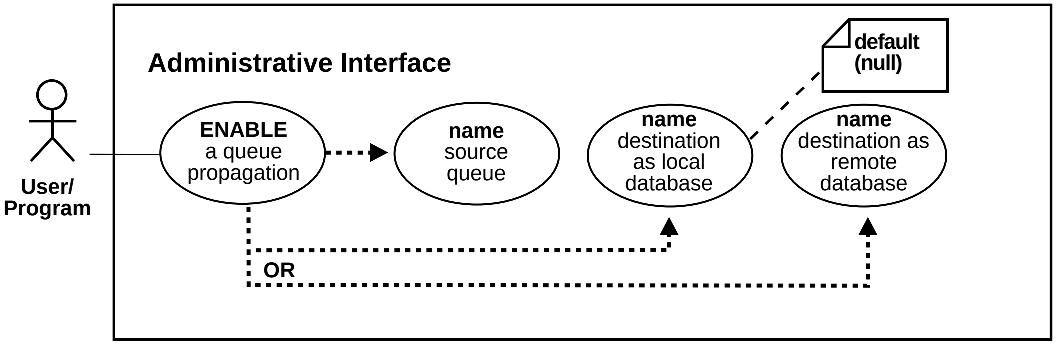
```
/* Alter schedule from queue aq.q1def to other queues in the same database */
EXECUTE DBMS_AQADM.ALTER_PROPAGATION_SCHEDULE(
   Queue_name    =>    'aq.q1def',
   Duration      =>    '2000',
   Next_time     =>    'SYSDATE + 3600/86400',
   Latency       =>    '32');
```

Alter a Schedule from a Queue to Other Queues in Another Database

```
/* Alter schedule from queue aq.q1def to other queues in another database
reached by the database link another_db.world */
EXECUTE DBMS_AQADM.ALTER_PROPAGATION_SCHEDULE(
   Queue_name    =>    'aq.q1def',
   Destination   =>    'another_db.world',
   Duration      =>    '2000',
   Next_time     =>    'SYSDATE + 3600/86400',
   Latency       =>    '32');
```

Enable a Propagation Schedule

Figure 4–23 Use Case Diagram: Enable a Propagation Schedule



To refer to the table of all basic operations having to do with the Administrative Interface see:

- "Use Case Model: Administrative Interface — Basic Operations" on page 4-2

Purpose:

To enable a previously disabled propagation schedule.

Syntax:

```
DBMS_AQADM.ENABLE_PROPAGATION_SCHEDULE (
   queue_name          IN     VARCHAR2,
   destination         IN     VARCHAR2 default NULL);
```

Usage:

Table 4–22 DBMS_AQADM.ENABLE_ PROPAGATION_SCHEDULE

Parameter	Description
`queue_name` `(IN VARCHAR2)`	specifies the name of the source queue whose messages are to be propagated, including the schema name. If the schema name is not specified, it defaults to the schema name of the user.
`destination` `(IN VARCHAR2)`	specifies the destination dblink. The destination queue is in the database that is specified by the dblink. If the destination is `NULL`, the destination queue is the same database as the source queue. The length of this field is currently limited to 128 bytes and if the name is not fully qualified the default domain name is used.

Example: Enable a Propagation Using PL/SQL (DBMS_AQADM)

Enable Propagation from a Queue to Other Queues in the Same Database

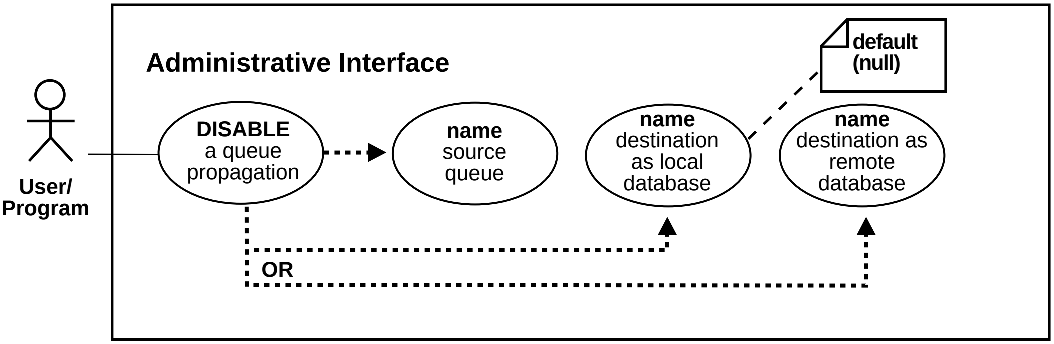
```
/* Enable propagation from queue aq.q1def to other queues in the same
   database */
EXECUTE DBMS_AQADM.ENABLE_PROPAGATION_SCHEDULE(
   Queue_name   =>   'aq.q1def');
```

Enable Propagation from a Queue to Queues in Another Database

```
/* Enable propagation from queue aq.q1def to other queues in another
  database reached by the database link another_db.world */
EXECUTE DBMS_AQADM.ENABLE_PROPAGATION_SCHEDULE(
   Queue_name    =>    'aq.q1def',
   Destination   =>    'another_db.world');
```

Disable a Propagation Schedule

Figure 4–24 Use Case Diagram: Disable a Propagation Schedule



To refer to the table of all basic operations having to do with the Administrative Interface see:

- "Use Case Model: Administrative Interface — Basic Operations" on page 4-2

Purpose:

To disable a previously disabled propagation schedule.

Syntax:

```
DBMS_AQADM.DISABLE_PROPAGATION_SCHEDULE (
   queue_name      IN     VARCHAR2,
   destination     IN     VARCHAR2 default NULL);
```

Usage:

Table 4–23 DBMS_AQADM.DISABLE_PROPAGATION_SCHEDULE

Parameter	Description
`queue_name` `(IN VARCHAR2)`	specifies the name of the source queue whose messages are to be propagated, including the schema name. If the schema name is not specified, it defaults to the schema name of the user.
`destination` `(IN VARCHAR2)`	specifies the destination dblink; the destination queues are in the database that is specified by the dblink. If the destination is `NULL`, the destination queue is the same database as the source queue. The length of this field is currently limited to 128 bytes and if the name is not fully qualified the default domain name is used.

Example: Disable a Propagation Using PL/SQL (DBMS_AQADM)

Disable Propagation from a Queue to Other Queues in the Same Database

```
/* Disable a propagation from queue aq.q1def to other queues in the same
   database */
EXECUTE DBMS_AQADM.DISABLE_PROPAGATION_SCHEDULE(
   Queue_name   =>   'aq.q1def');
```

Disable Propagation from a Queue to Queues in Another Database

```
/* Disable a propagation from queue aq.q1def to other queues in another
  database reached by the database link another_db.world */
EXECUTE DBMS_AQADM.DISABLE_PROPAGATION_SCHEDULE(
   Queue_name    =>    'aq.q1def',
   Destination   =>    'another_db.world');
```

Usage Notes

This section describes some troubleshooting tips to diagnose problems with message propagation.

Message history

AQ updates the message history when a message has been successfully propagated to a destination. The message history is stored as a collection in the queue table. An administrator can execute a SQL query to determine if a message has been propagated. For example, to check if a message with msgid

`105E7A2EBFF11348E03400400B40F149`'

in queue table *`aqadmn.queue_tab`* has been propagated to destination '*`boston`*', the following query can be executed:

```
SELECT consumer, transaction_id, deq_time, deq_user, propagated_msgid
   FROM THE(select cast(history as aq$_dequeue_history_t)
   FROM adadmn.queue_tab
      WHERE msgid='105E7A2EBFF11348E03400400B40F149')
      WHERE consumer LIKE '%BOSTON%';
```

A non-`NULL transaction_id` indicates that the message was successfully propagated. Further, the `deq_time` indicates the time of propagation, the `deq_user` indicates the userid used for propagation, and the `propagated_msgid` indicates the msgid of the message that was enqueued at the destination. If the message with the msgid cannot be found in the queue table, an administrator can check the exception queue (if the exception queue is in a different queue table) for the message history.

Propagation Schedules

To verify that propagation is working successfully, examine the schedule information using the `DBA_QUEUE_SCHEDULES` view. Check the error message field to discover if any error occurred during propagation. If there was an error, the error time and error date field display when the error last occurred. After you have corrected the problem, propagation should resume.

You should also determine if the schedule has been disabled (`DISABLED` field is `Y`). Propagation should resume once you have enabled the schedule by invoking `ENABLE_PROPAGATION_SCHEDULE`. If the schedule is already enabled, check if the schedule is active. A schedule is active if a `PROCESS_NAME` exists for that schedule. If one does not exist, which means that the schedule is inactive, check the time of the last successful execution and when the schedule will be next executed. If the next scheduled execution is too far away, change the `NEXT_TIME` parameter of the schedule so that schedules are executed more frequently (assuming that the window is not set to be infinite).

Parameters of a schedule can be changed using the `ALTER_PROPAGATION_SCHEDULE` call. If a schedule is active then the source queue may not have any messages to be propagated.

Database link

There are a number of points at which propagation may break down:

- You may want to determine if the destination is reachable with regard to whether the network connection to the destination is available. You do this by executing a simple distributed query, or by creating a connection descriptor that has the same connect string, and then by trying to connect to the remote database.
- You need to ensure that the userid that scheduled the propagation (using `dbms_aqadm.schedule_propagation`) has access to the database link for the destination.
- Verify that the userid used to login to the destination through the database link has been granted privileges to use the AQ.
- Check if the queue name specified in the address attribute of the `aq$_agent` type (in the subscriber list for the source queue or in the recipient list of the enqueuer) both (a) exists at the specified destination, and (b) has been enabled for enqueuing. All these and other errors that the propagator encounters are logged into trace file(s) generated by the job_queue processes in `$ORACLE_HOME/`log directory.

Type checking

AQ will not propagate messages from one queue to another if the payload-types of the two queues are not equivalent. An administrator can verify if the source and destination's payload types match by executing the `DBMS_AQADM.VERIFY_QUEUE_TYPES` procedure. The results of the type checking will be stored in the aq$_message_types table. This table can be accessed using the `OID` of the source queue and the address of the destination queue (i.e. [`schema.`]`queue_name`[`@destination`]).

5

Administrative Interface: Views

In this chapter we describe the administrative interface with respect to views in terms of a hybrid of use cases and state diagrams. That is, we describe each view as a use case in terms of the operations that represents it (such as "Select All Queue Tables in Database"). We describe each view as a state diagram in that each attribute of the view is represented as a possible state of the view, the implication being that any attribute (column) can be visible or invisible.

The table listing all the use cases is provided at the head of the chapter (see "Use Case Model: Administrative Interface — Views" on page 5-2). A summary figure, "Use Case Diagram: Administrator's Interface — Views", locates all the use cases in single drawing. If you are using the HTML version of this document, you can use this figure to navigate to the use case in which you are interested by clicking on the relevant use case title.

The individual use cases are themselves laid out as follows:

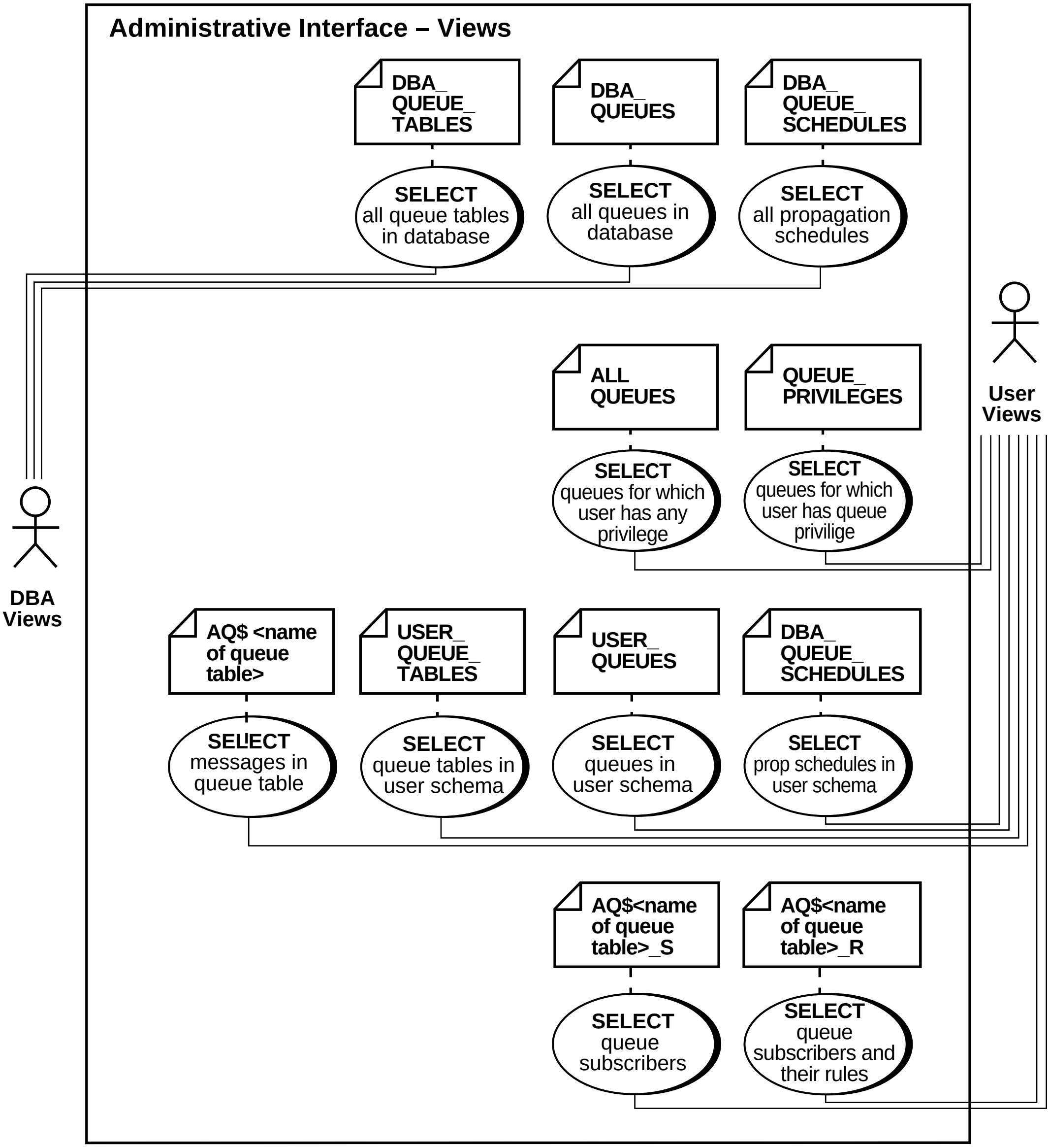
- A figure that depicts the use case (see "Preface" for a description of how to interpret these diagrams).
- A listing of the syntax.

Use Case Model: Administrative Interface — Views

Table 5–1 Use Case Model: Administrative Interface — Views

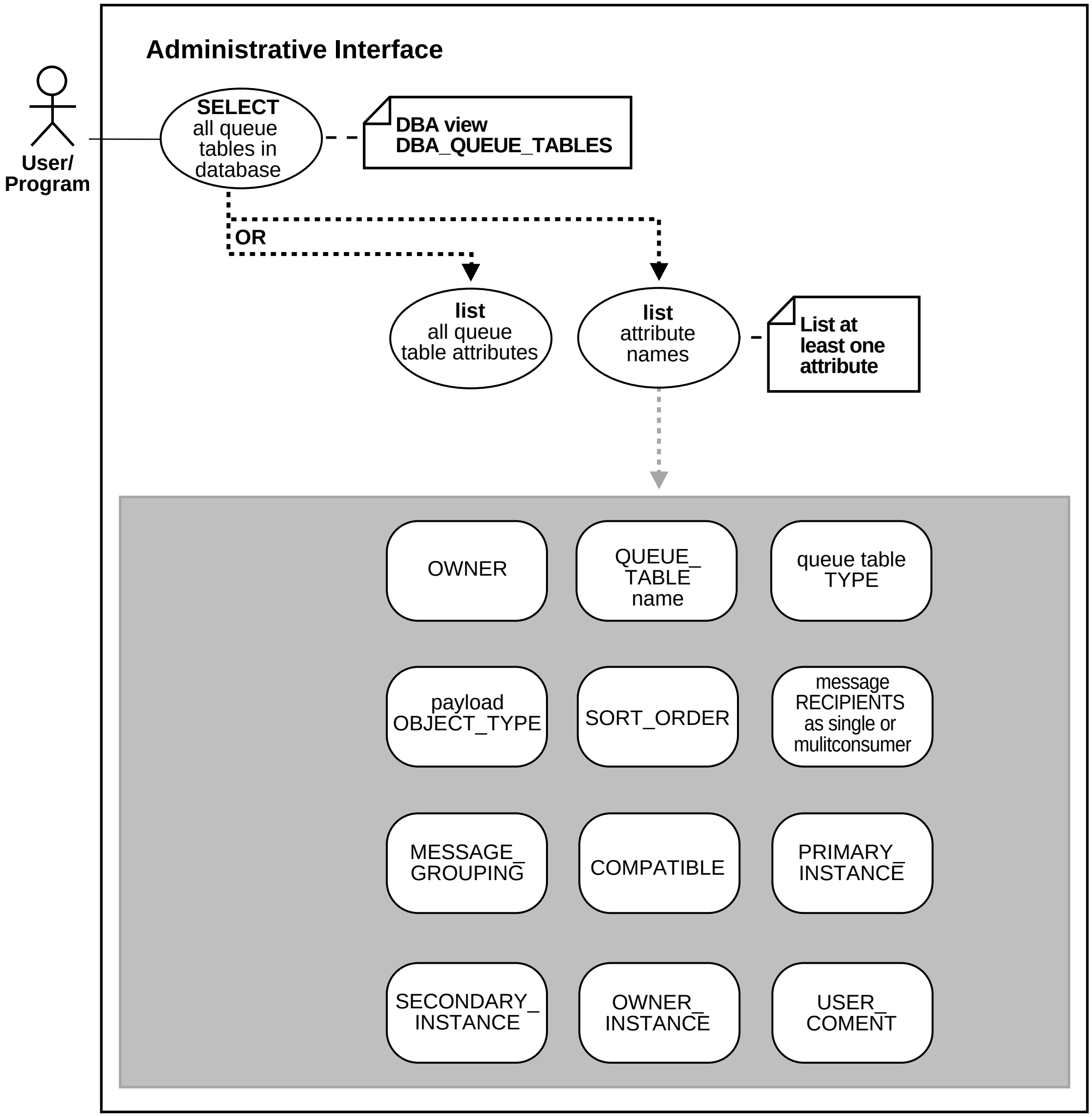
Use Case	Name of View
Select All Queue Tables in Database on page 5-4	DBA_QUEUE_TABLES
Select User Queue Tables on page 5-7	ALL_QUEUE_TABLES
Select All Queues in Database on page 5-10	DBA_QUEUES
Select All Propagation Schedules on page 5-12	DBA_QUEUE_SCHEDULES
Select Queues for which User has Any Privilege on page 5-17	ALL_QUEUES
Select Queues for which User has Queue Privilege on page 5-19	QUEUE_PRIVILEGES
Select Messages in Queue Table on page 5-21	AQ$<name of queue table>
Select Queue Tables in User Schema on page 5-25	USER_QUEUE_TABLES
Select Queues In User Schema on page 5-28	USER_QUEUES
Select Propagation Schedules in User Schema on page 5-30	USER_QUEUE_SCHEDULES
Select Queue Subscribers on page 5-35	AQ$<name of queue table>_S
Select Queue Subscribers and their Rules on page 5-37	AQ$<name of queue table>_R
Select the Number of Messages in Different States for the Whole Database on page 5-39	GV$AQ
Select the Number of Messages in Different States for Specific Instances on page 5-41	VSAQ

Figure 5–1 Use Case Model: Administrative Interface — Views



Select All Queue Tables in Database

Figure 5–2 Use Case Diagram: Select All Queue Tables in Database



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Administrative Interface — Views" on page 5-2

Name of View:

`DBA_QUEUE_TABLES`

Purpose:

This view describes the names and types of all queue tables created in the database.

Table 5–2 DBA_QUEUE_TABLES

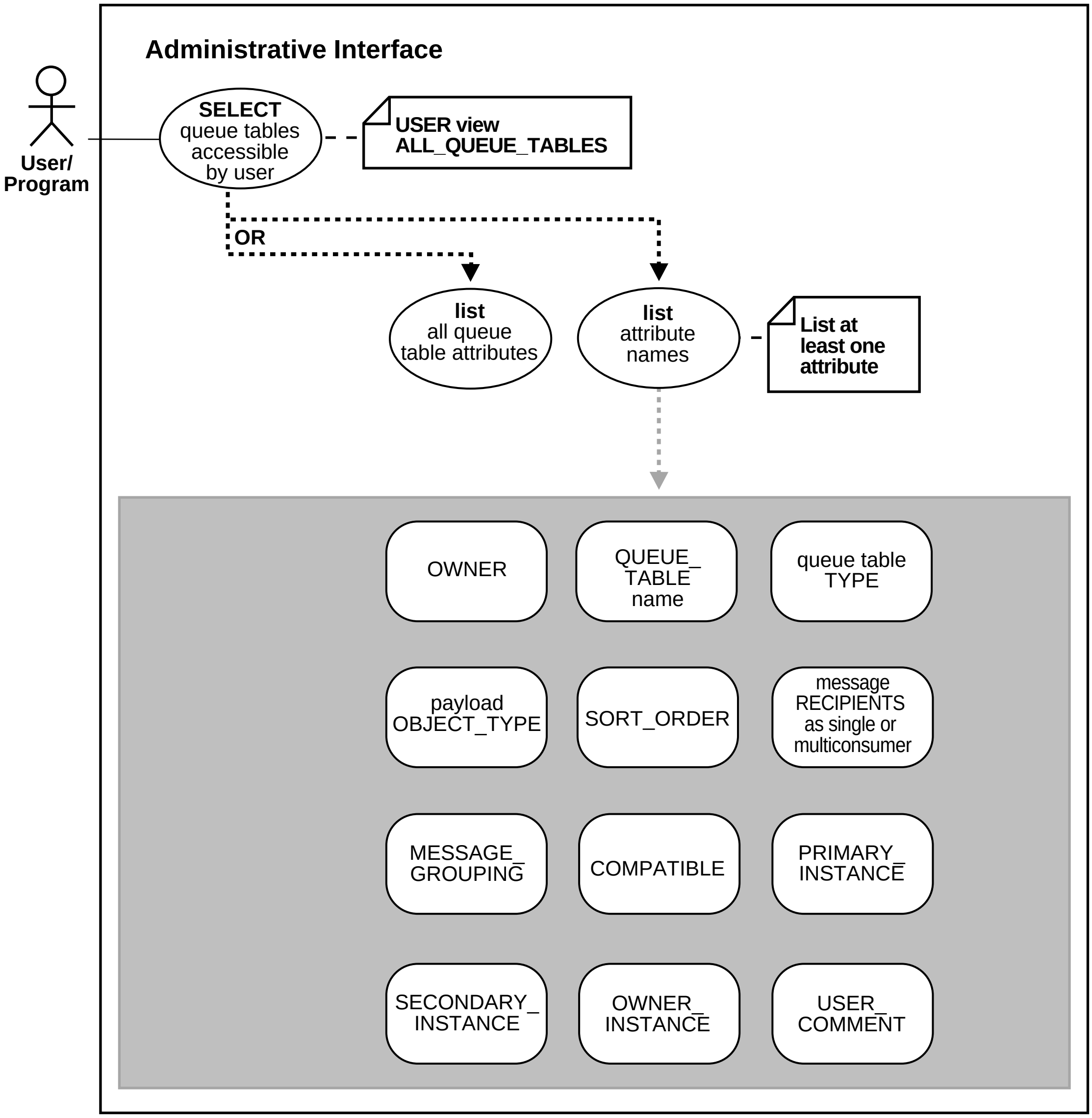
Column Name & Description	Null?	Type
`OWNER — queue table schema`		`VARCHAR2(30)`
`QUEUE_TABLE - queue table name`		`VARCHAR2(30)`
`TYPE — payload type`		`VARCHAR2(7)`
`OBJECT_TYPE — name of object type, if any`		`VARCHAR2(61)`
`SORT_ORDER — user specified sort order`		VARCHAR2(22)
`RECIPIENTS — SINGLE or MULTIPLE`		`VARCHAR2(8)`
`MESSAGE_GROUPING — NONE or TRANSACTIONAL`		VARCHAR2(13)
`COMPATIBLE — indicates the lowest version with which the queue table is compatible`		`VARCHAR2(5)`
`PRIMARY_INSTANCE — indicates which instance is the primary owner of the queue table; a value of 0 indicates that there is no primary owner`		`NUMBER`

Table 5–2 DBA_QUEUE_TABLES

Column Name & Description	Null?	Type
`SECONDARY_INSTANCE` — indicates which owner is the secondary owner of the queue table; this instance becomes the owner of the queue table if the primary owner is not up; a value of 0 indicates that there is no secondary owner		`NUMBER`
`OWNER_INSTANCE` — indicates which instance currently owns the queue table		`NUMBER`
`USER_COMMENT` — user comment for the queue table		`VARCHAR2(50)`

Select User Queue Tables

Figure 5–3 Use Case Diagram: Select User Queue Tables



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Administrative Interface — Views" on page 5-2

Name of View:

`ALL_QUEUE_TABLES`

Purpose:

This view describes queue tables accessible to a user.

Table 5–3 DBA_QUEUE_TABLES

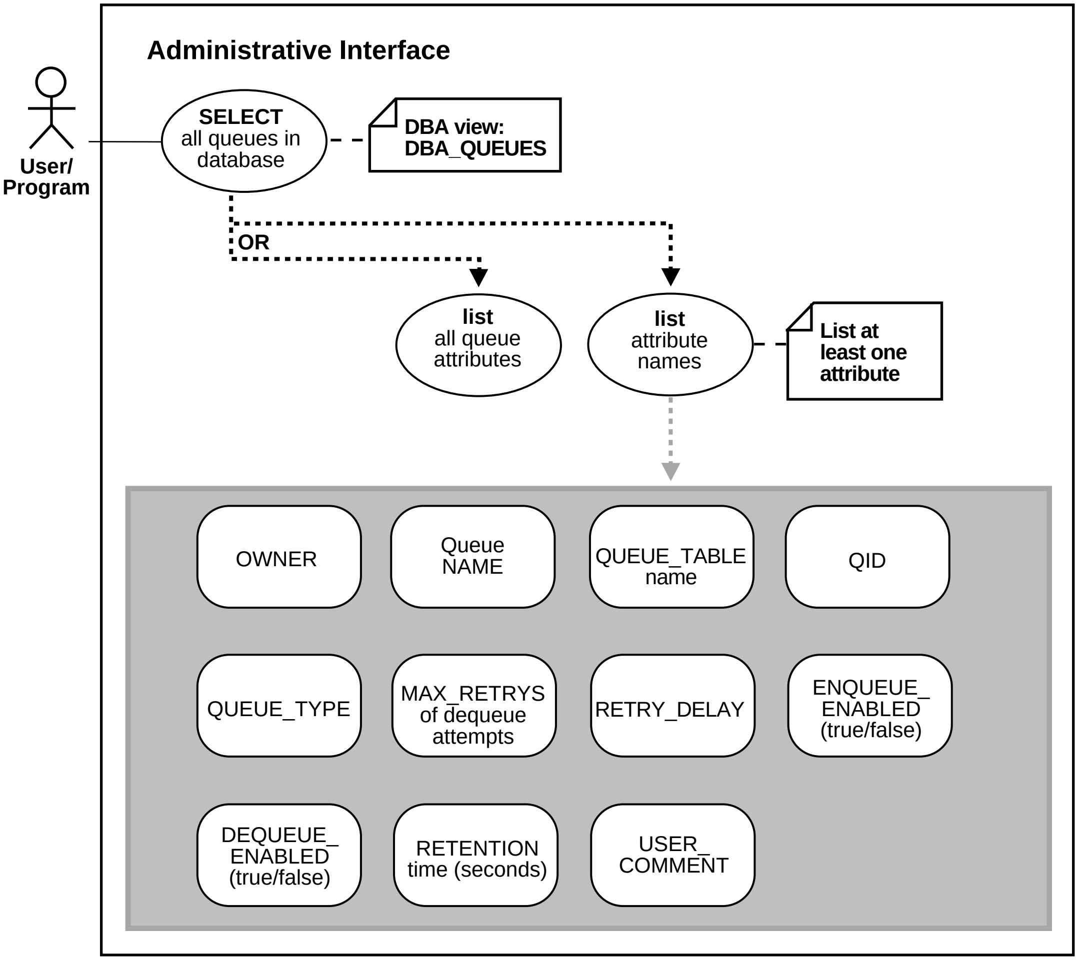
Column Name & Description	Null?	Type
`OWNER — owner of the queue table`		`VARCHAR2(30)`
`QUEUE_TABLE - queue table name`		`VARCHAR2(30)`
`TYPE — payload type`		`VARCHAR2(7)`
`OBJECT_TYPE — object type, if any`		`VARCHAR2(61)`
`SORT_ORDER — user-specified sort order`		`VARCHAR2(22)`
`RECIPIENTS — SINGLE or MULTIPLE recipient queue`		`VARCHAR2(8)`
`MESSAGE_GROUPING — NONE or TRANSACTIONAL`		`VARCHAR2(13)`
`COMPATIBLE — indicates the lowest version with which the queue table is compatible`		`VARCHAR2(5)`
`PRIMARY_INSTANCE — indicates which instance is the primary owner of the queue table; a value of 0 indicates that there is no primary owner`		`NUMBER`

Table 5–3 DBA_QUEUE_TABLES

Column Name & Description	Null?	Type
SECONDARY_INSTANCE — indicates which owner is the secondary owner of the queue table; this instance becomes the owner of the queue table if the primary owner is not up; a value of 0 indicates that there is no secondary owner		NUMBER
OWNER_INSTANCE — indicates which instance currently owns the queue table		NUMBER
USER_COMMENT — user comment for the queue table		VARCHAR2(50)

Select All Queues in Database

Figure 5–4 Use Case Diagram: Select All Queues in Database



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Administrative Interface — Views" on page 5-2

Name of View:

`DBA_QUEUES`

Purpose:

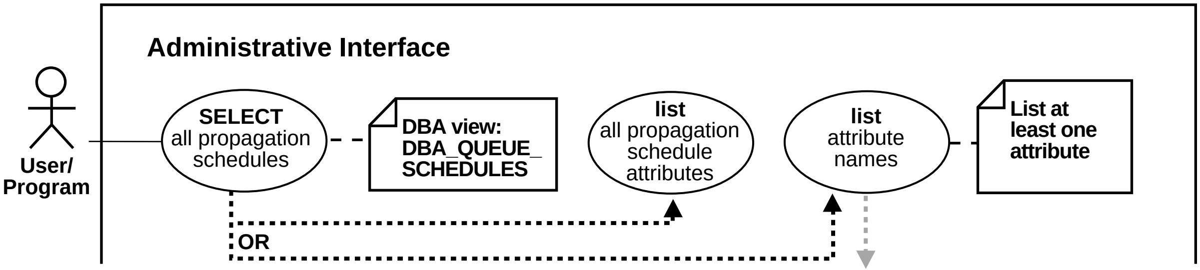
Users can specify operational characteristics for individual queues. `DBA_QUEUES` contains the view which contains relevant information for every queue in a database.

Table 5–4 DBA_QUEUES

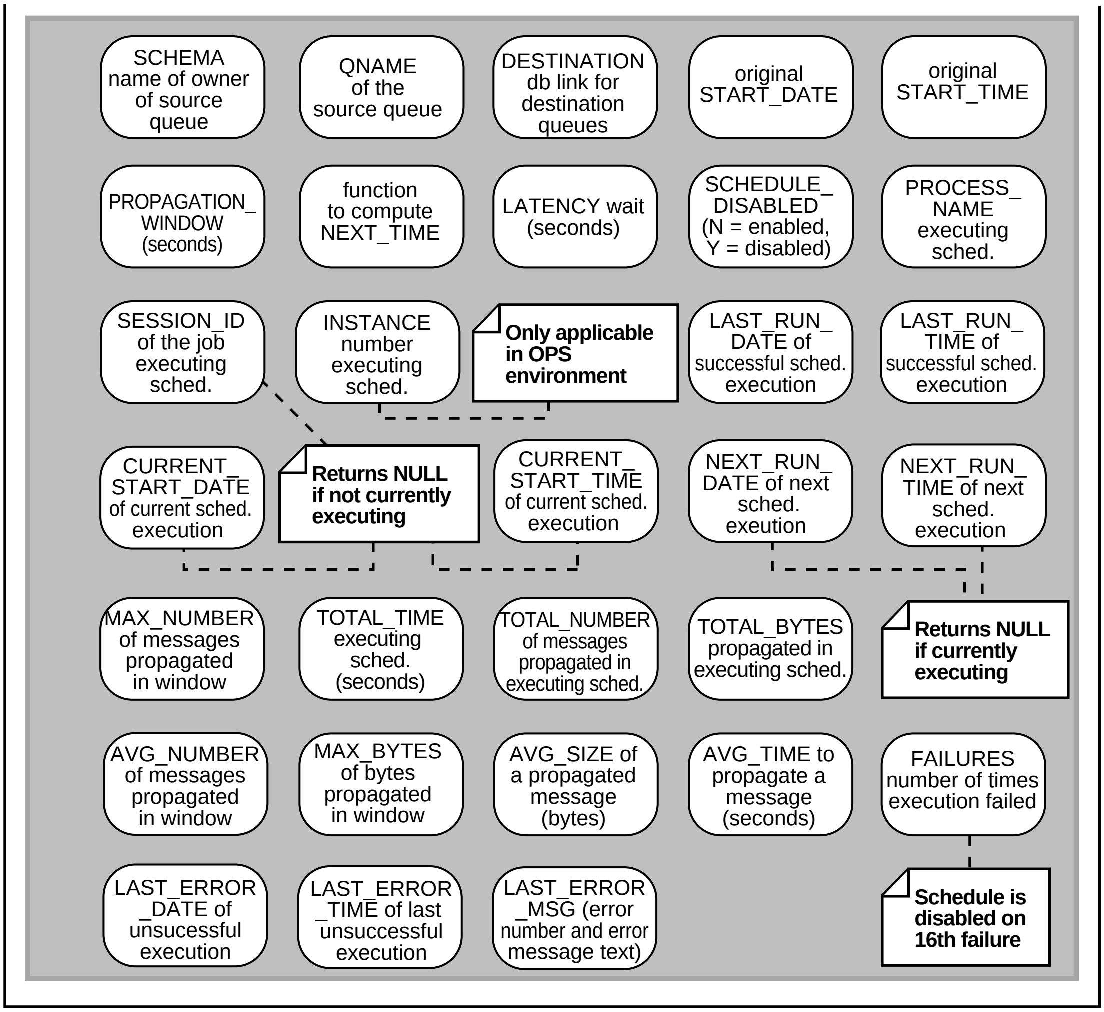
Column Name & Description	Null?	Type
`OWNER — queue schema name`	`NOT NULL`	`VARCHAR2(30)`
`NAME — queue name`	`NOT NULL`	`VARCHAR2(30)`
`QUEUE_TABLE — queue table where this queue resides`	`NOT NULL`	`VARCHAR2(30)`
`QID — unique queue identifier`	`NOT NULL`	`NUMBER`
`QUEUE_TYPE — queue type`		`VARCHAR2(15)`
`MAX_RETRIES — number of dequeue attempts allowed`		`NUMBER`
`RETRY_DELAY — number of seconds before retry can be attempted`		`NUMBER`
`ENQUEUE_ENABLED — YES/NO`		`VARCHAR2(7)`
`DEQUEUE_ENABLED — YES/NO`		`VARCHAR2(7)`
`RETENTION — number of seconds message is retained after dequeue`		`VARCHAR2(40)`
`USER_COMMENT — user comment for the queue`		`VARCHAR2(50)`

Select All Propagation Schedules

Figure 5–5 Use Case Diagram: Select All Propagation Schedules



continued on next page



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Administrative Interface — Views" on page 5-2

Name of View:

`DBA_QUEUE_SCHEDULES`

Purpose:

This view describes the current schedules for propagating messages.

Table 5–5 DBA_QUEUE_SCHEDULES

Column Name & Description	Null?	Type
SCHEMA — schema name for the source queue	NOT NULL	VARCHAR2(30)
QNAME — source queue name	NOT NULL	VARCHAR2(30)
DESTINATION — destination name, currently limited to be a DBLINK name	NOT NULL	VARCHAR2(128)
START_DATE — date to start propagation in the default date format		DATE
START_TIME — time of day at which to start propagation in HH:MI:SS format		VARCHAR2(8)
PROPAGATION_WINDOW — duration in seconds for the propagation window		NUMBER
NEXT_TIME — function to compute the start of the next propagation window		VARCHAR2(200)
LATENCY — maximum wait time to propagate a message during the propagation window.		NUMBER
SCHEDULE_DISABLED — N if enabled Y if disabled and schedule will not be executed		VARCHAR(1)
PROCESS_NAME — The name of the SNP background process executing this schedule. NULL if not currently executing		VARCHAR2(8)
SESSION_ID — The session ID (SID, SERIAL#) of the job executing this schedule. NULL if not currently executing		NUMBER

Table 5–5 DBA_QUEUE_SCHEDULES

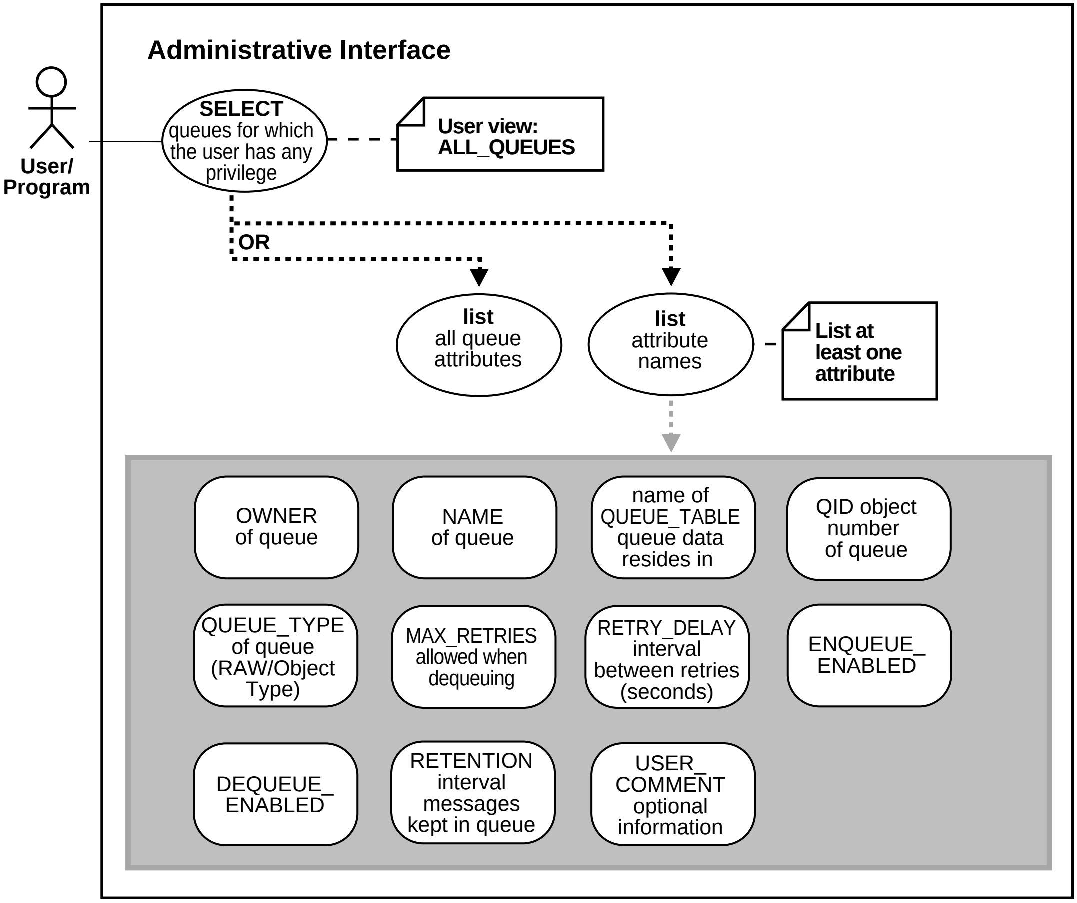
Column Name & Description	Null?	Type
INSTANCE — The OPS instance number executing this schedule		NUMBER
LAST_RUN_DATE — The date on the last successful execution		DATE
LAST_RUN_TIME — The time of the last successful execution in HH:MI:SS format		VARCHAR2(8)
CURRENT_START_DATE — Date at which the current window of this schedule was started		DATE
CURRENT_START_TIME — Time of day at which the current window of this schedule was started in HH:MI:SS format		VARCHAR2(8)
NEXT_RUN_DATE — Date at which the next window of this schedule will be started		DATE
NEXT_RUN_TIME — Time of day at which the next window of this schedule will be started in HH:MI:SS format		VARCHAR2(8)
TOTAL_TIME — Total time in seconds spent in propagating messages from the schedule		NUMBER
TOTAL_NUMBER — Total number of messages propagated in this schedule		NUMBER
TOTAL_BYTES — Total number of bytes propagated in this schedule		NUMBER
MAX_NUMBER — The maximum number of messages propagated in a propagation window		NUMBER
MAX_BYTES — The maximum number of bytes propagated in a propagation window		NUMBER
AVG_NUMBER — The average number of messages propagated in a propagation window		NUMBER

Table 5–5 DBA_QUEUE_SCHEDULES

Column Name & Description	Null?	Type
AVG_SIZE — The average size of a propagated message in bytes		NUMBER
AVG_TIME — The average time, in seconds, to propagate a message		NUMBER
FAILURES — The number of times the execution failed. If 16, the schedule will be disabled		NUMBER
LAST_ERROR_DATE — The date of the last unsuccessful execution		DATE
LAST_ERROR_TIME — The time of the last unsuccessful execution		VARCHAR2(8)
LAST_ERROR_MSG — The error number and error message text of the last unsuccessful execution		VARCHAR2(4000)

Select Queues for which User has Any Privilege

Figure 5–6 Use Case Diagram: Select Queues for which User has Any Privilege



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Administrative Interface — Views" on page 5-2

Name of View:

`ALL_QUEUES`

Purpose:

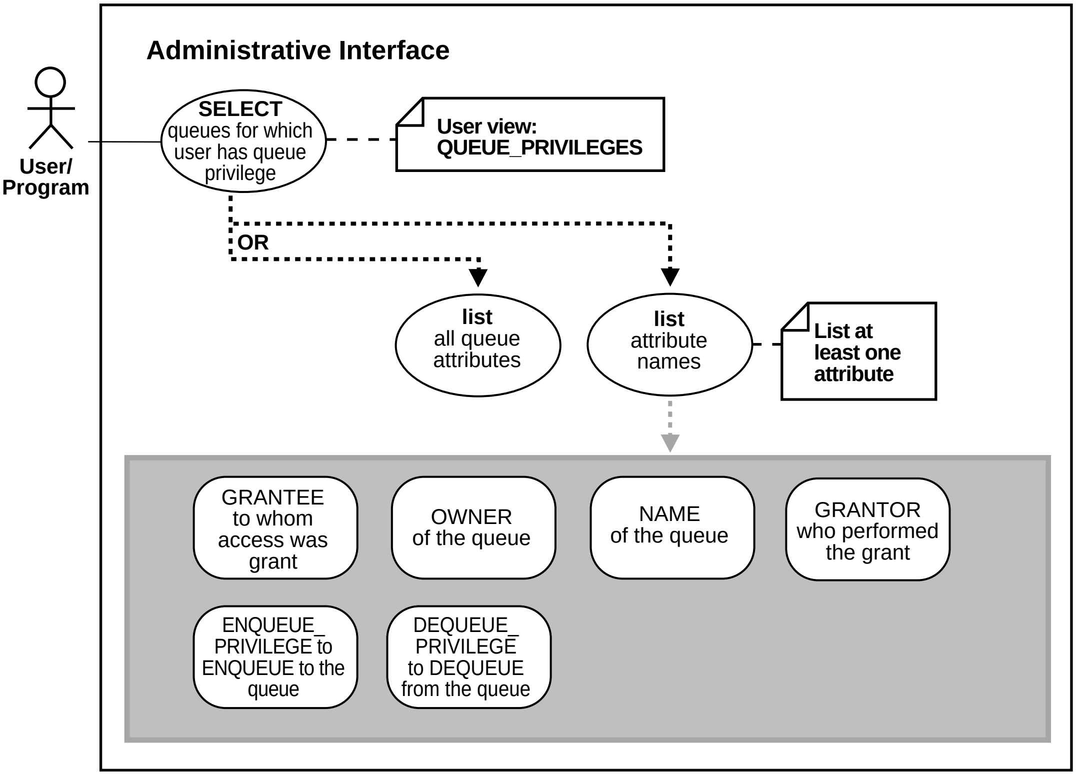
This view describes all queues accessible to the user.

Table 5–6 ALL_QUEUES

Column Name & Description	Null?	Type
`OWNER — Owner of the queue`	`NOT NULL`	`VARCHAR2(30)`
`NAME — Name of the queue`	`NOT NULL`	`VARCHAR2(30)`
`QUEUE_TABLE — Name of the table the queue data resides in`	`NOT NULL`	`VARCHAR2(30)`
`QID — Object number of the queue`	`NOT NULL`	`NUMBER`
`QUEUE_TYPE — Type of the queue`		`VARCHAR2(15)`
`MAX_RETRIES — Maximum number of retries allowed when dequeuing from the queue`		`NUMBER`
`RETRY_DELAY — Time interval between retries`		`NUMBER`
`ENQUEUE_ENABLED — Queue is enabled for enqueue`		`VARCHAR2(7)`
`DEQUEUE_ENABLED — Queue is enabled for dequeue`		`VARCHAR2(7)`
`RETENTION — Time interval processed messages retained in the queue`		`VARCHAR2(40)`
`USER_COMMENT — User specified comment`		`VARCHAR2(50)`

Select Queues for which User has Queue Privilege

Figure 5–7 Use Case Diagram: Select Queues for which User has Queue Privilege



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Administrative Interface — Views" on page 5-2

Name of View:

`QUEUE_PRIVILEGES`

Purpose:

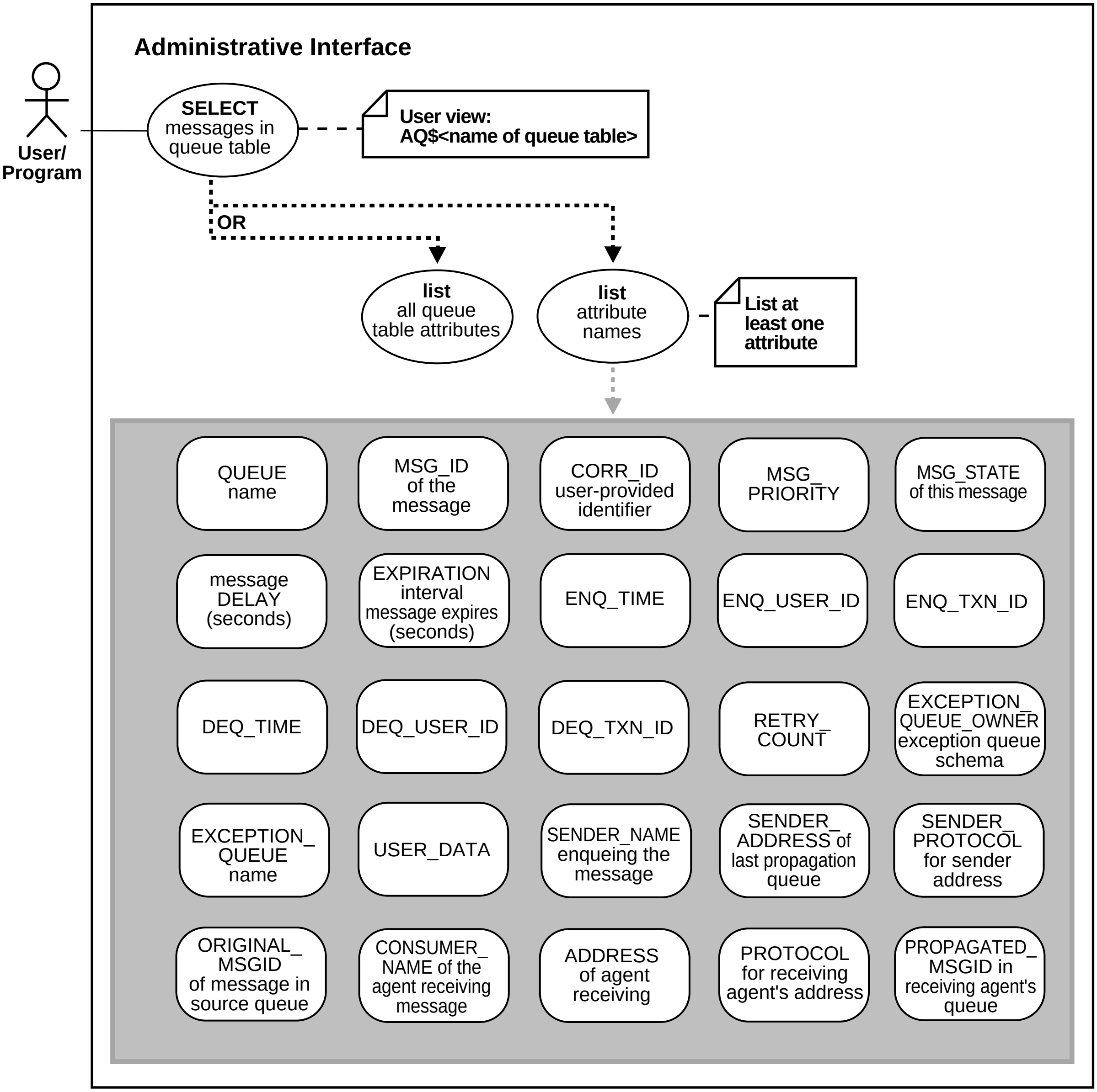
This view describes queues for which the user is the grantor, or grantee, or owner, or an enabled role or the queue is granted to `PUBLIC`.

Table 5–7 QUEUE_PRIVILEGES

Column Name & Description	Null?	Type
`GRANTEE — Name of the user to whom access was granted`	`NOT NULL`	`VARCHAR2(30)`
`OWNER — Owner of the queue`	`NOT NULL`	`VARCHAR2(30)`
`NAME — Name of the queue`	`NOT NULL`	`VARCHAR2(30)`
`GRANTOR — Name of the user who performed the grant`	`NOT NULL`	`VARCHAR2(30)`
`ENQUEUE_PRIVILEGE — Permission to ENQUEUE to the queue`		`NUMBER(1 if granted, 0 if not)`
`DEQUEUE_PRIVILEGE — Permission to DEQUEUE to the queue`		`NUMBER(1 if granted, 0 if not)`

Select Messages in Queue Table

Figure 5–8 Use Case Diagram: Select Messages in Queue Table



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Administrative Interface — Views" on page 5-2

Name of View:

`AQ$<name of queue table>`

Purpose:

This view describes the queue table in which message data is stored. This view is automatically created with each queue table and should be used for querying the queue data. The dequeue history data (time, user identification and transaction identification) is only valid for single consumer queues.

Table 5–8 AQ$<name of queue table>

Column Name & Description	Null?	Type
QUEUE — queue name		VARCHAR2(30)
MSG_ID — unique identifier of the message		RAW(16)
CORR_ID — user-provided correlation identifier		VARCHAR2(128)
MSG_PRIORITY — message priority		NUMBER
MSG_STATE — state of this message		VARCHAR2(9)
DELAY — number of seconds the message is delayed		DATE
EXPIRATION — number of seconds in which the message will expire after being READY		NUMBER
ENQ_TIME — enqueue time		DATE
ENQ_USER_ID — enqueue user id		NUMBER
ENQ_TXN_ID — enqueue transaction id	NOT NULL	VARCHAR2(30)
DEQ_TIME — dequeue time		DATE
DEQ_USER_ID — dequeue user id		NUMBER

Table 5–8 AQ$<name of queue table>

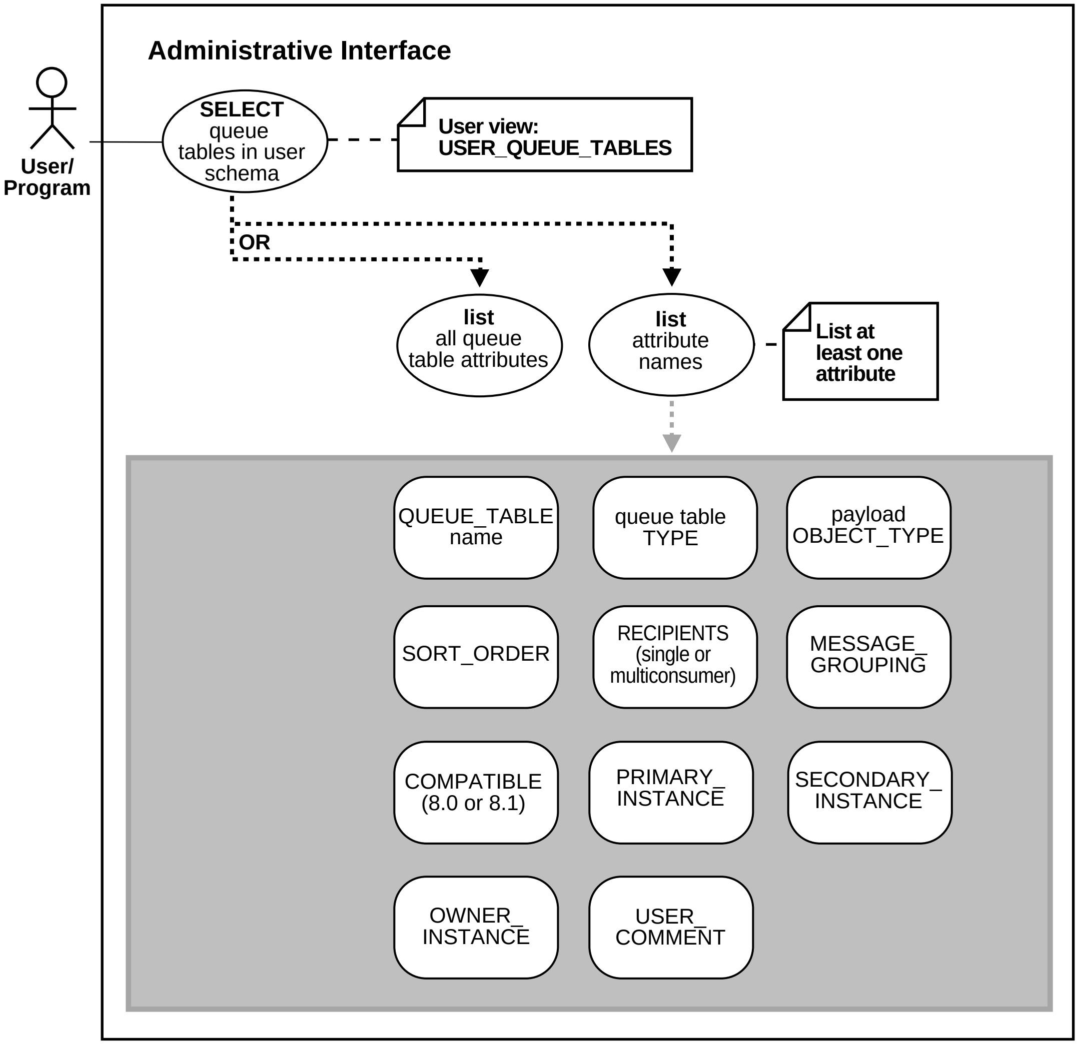
Column Name & Description	Null?	Type
DEQ_TXN_ID — dequeue transaction id		VARCHAR2(30)
RETRY_COUNT — number of retries		NUMBER
EXCEPTION_QUEUE_OWNER — exception queue schema		VARCHAR2(30)
EXCEPTION_QUEUE — exception queue name		VARCHAR2(30)
USER_DATA — user data		BLOB
SENDER_NAME — name of the Agent enqueuing the message (valid only for 8.1-compatible queue tables)		VARCHAR2(30)
SENDER_ADDRESS — queue name and database name of the source (last propagating) queue; the database name is not specified if the source queue is in the local database (valid only for 8.1-compatible queue tables)		VARCHAR2(1024)
SENDER_PROTOCOL — protocol for sender address, reserved for future use (valid only for 8.1-compatible queue tables)		NUMBER
ORIGINAL_MSGID — message id of the message in the source queue (valid only for 8.1-compatible queue tables)		RAW(16)
CONSUMER_NAME — name of the Agent receiving the message (valid ONLY for 8.1-compatible MULTICONSUMER queue tables)		VARCHAR2(30)
ADDRESS — address (queue name and database link name) of the agent receiving the message.The database link name is not specified if the address is in the local database. The address is NULL if the receiving agent is local to the queue (valid ONLY for 8.1-compatible MULTICONSUMER queue tables)		VARCHAR2(1024)

Table 5–8 AQ$<name of queue table>

Column Name & Description	Null?	Type
PROTOCOL — protocol for receiving agent’s address (valid only for 8.1-compatible queue tables)		NUMBER
PROPAGATED_MSGID — message id of the message in the receiving agent’s queue (valid only for 8.1-compatible queue tables)	NULL	RAW(16)

Select Queue Tables in User Schema

Figure 5–9 Use Case Diagram: Select Queue Tables in User Schema



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Administrative Interface — Views" on page 5-2

Name of View:

`USER_QUEUE_TABLES`

Syntax:

This view is the same as `DBA_QUEUE_TABLES` with the exception that it only shows queue tables in the user's schema. It does not contain a column for `OWNER`.

Table 5–9 USER_QUEUE_TABLES

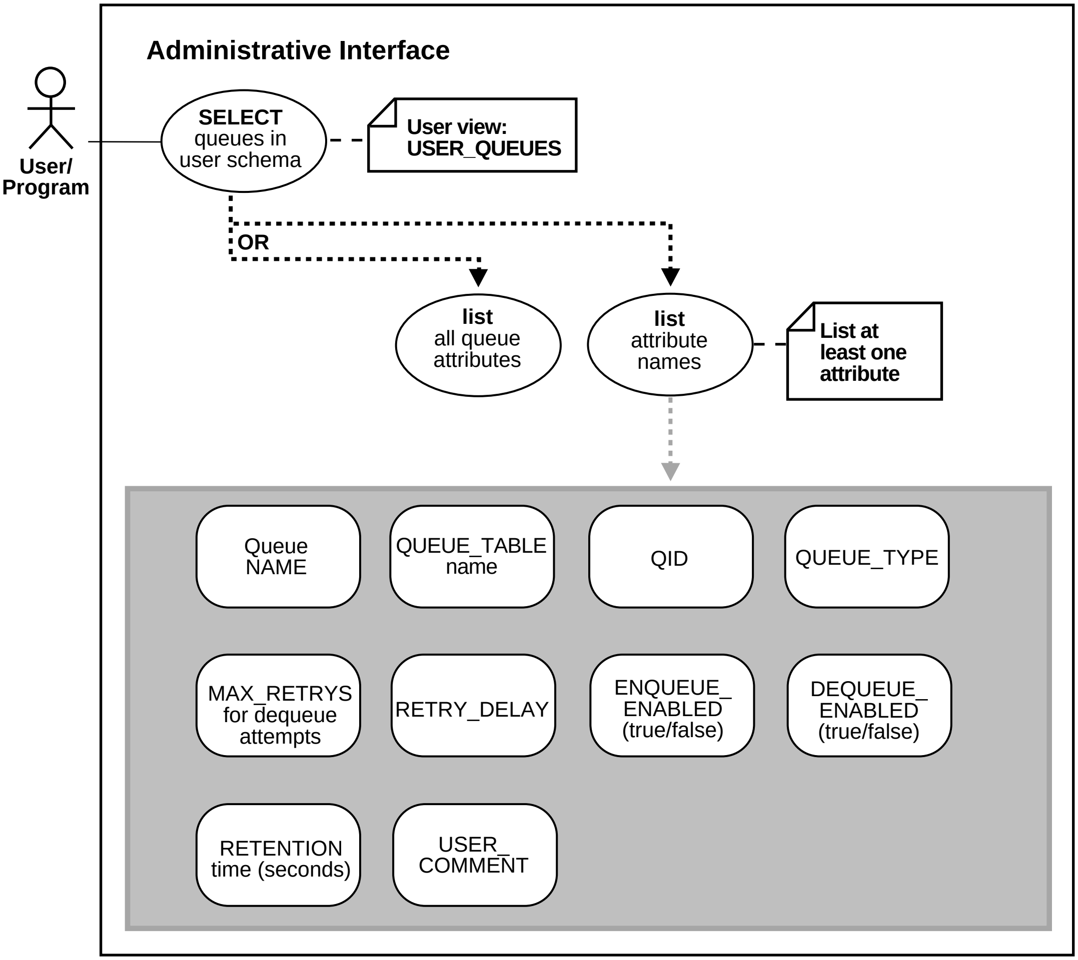
Column Name & Description	Null?	Type
QUEUE_TABLE - queue table name		VARCHAR2(30)
TYPE — payload type		VARCHAR2(7)
OBJECT_TYPE — name of object type, if any		VARCHAR2(61)
SORT_ORDER — user specified sort order		VARCHAR2(22)
RECIPIENTS — SINGLE or MULTIPLE		VARCHAR2(8)
MESSAGE_GROUPING — NONE or TRANSACTIONAL		VARCHAR2(13)
COMPATIBLE — indicates the lowest version with which the queue table is compatible		VARCHAR2(5)
PRIMARY_INSTANCE — indicates which instance is the primary owner of the queue table; a value of 0 indicates that there is no primary owner		NUMBER

Table 5–9 USER_QUEUE_TABLES

Column Name & Description	Null?	Type
SECONDARY_INSTANCE — indicates which owner is the secondary owner of the queue table; this instance becomes the owner of the queue table if the primary owner is not up; a value of 0 indicates that there is no secondary owner		NUMBER
OWNER_INSTANCE — indicates which instance currently owns the queue table		NUMBER
USER_COMMENT — user comment for the queue table		VARCHAR2(50)

Select Queues In User Schema

Figure 5–10 .Use Case Diagram: Select Queues in User Schema



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Administrative Interface — Views" on page 5-2

Name of View:

`USER_QUEUES`

Purpose:

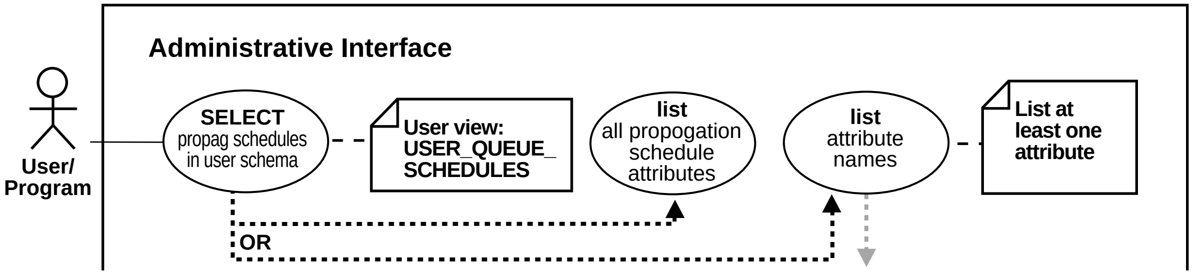
This view is the same as `DBA_QUEUES` with the exception that it only shows queues in the user's schema.

Table 5–10 USER_QUEUES

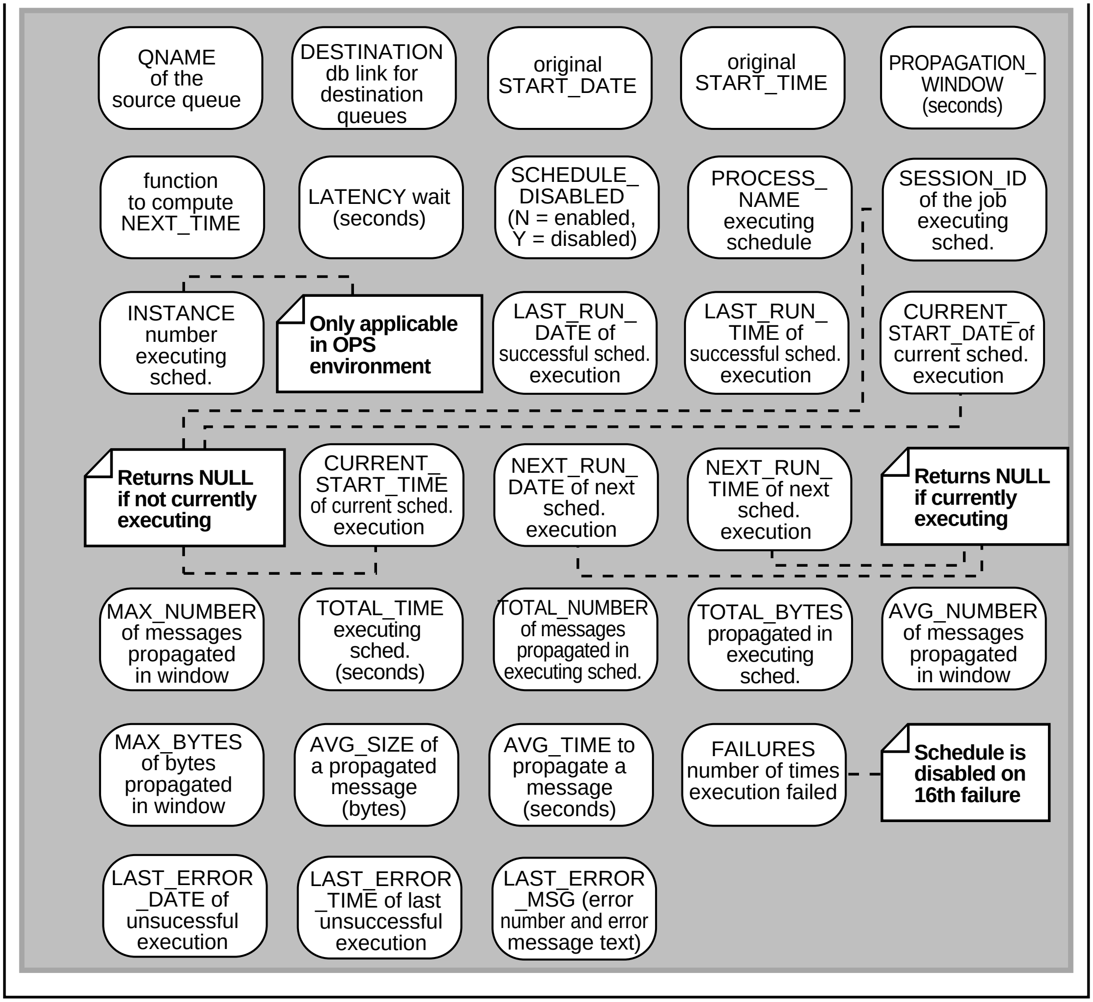
Column Name & Description	Null?	Type
`NAME — queue name`	`NOT NULL`	`VARCHAR2(30)`
`QUEUE_TABLE — queue table where this queue resides`	`NOT NULL`	`VARCHAR2(30)`
`QID — unique queue identifier`	`NOT NULL`	`NUMBER`
`QUEUE_TYPE — queue type`		`VARCHAR2(15)`
`MAX_RETRIES — number of dequeue attempts allowed`		`NUMBER`
`RETRY_DELAY — number of seconds before retry can be attempted`		`NUMBER`
`ENQUEUE_ENABLED — YES/NO`		`VARCHAR2(7)`
`DEQUEUE_ENABLED — YES/NO`		`VARCHAR2(7)`
`RETENTION — number of seconds message is retained after dequeue`		`VARCHAR2(40)`
`USER_COMMENT — user comment for the queue`		`VARCHAR2(50)`

Select Propagation Schedules in User Schema

Figure 5–11 Use Case Diagram: Select Propagation Schedules in User Schema



continued on next page



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Administrative Interface — Views" on page 5-2

Name:

USER_QUEUE_SCHEDULES

Purpose:

Table 5–11 USER_QUEUE_SCHEDULES

Column Name & Description	Null?	Type
QNAME — source queue name	NOT NULL	VARCHAR2(30)
DESTINATION — destination name, currently limited to be a DBLINK name	NOT NULL	VARCHAR2(128)
START_DATE — date to start propagation in the default date format		DATE
START_TIME — time of day at which to start propagation in HH:MI:SS format		VARCHAR2(8)
PROPAGATION_WINDOW — duration in seconds for the propagation window		NUMBER
NEXT_TIME — function to compute the start of the next propagation window		VARCHAR2(200)
LATENCY — maximum wait time to propagate a message during the propagation window.		NUMBER
SCHEDULE_DISABLED — N if enabled Y if disabled and schedule will not be executed		VARCHAR(1)
PROCESS_NAME — The name of the SNP background process executing this schedule. NULL if not currently executing		VARCHAR2(8)
SESSION_ID — The session ID (SID, SERIAL#) of the job executing this schedule. NULL if not currently executing		VARCHAR2(82)
INSTANCE — The OPS instance number executing this schedule		NUMBER

Table 5–11 USER_QUEUE_SCHEDULES

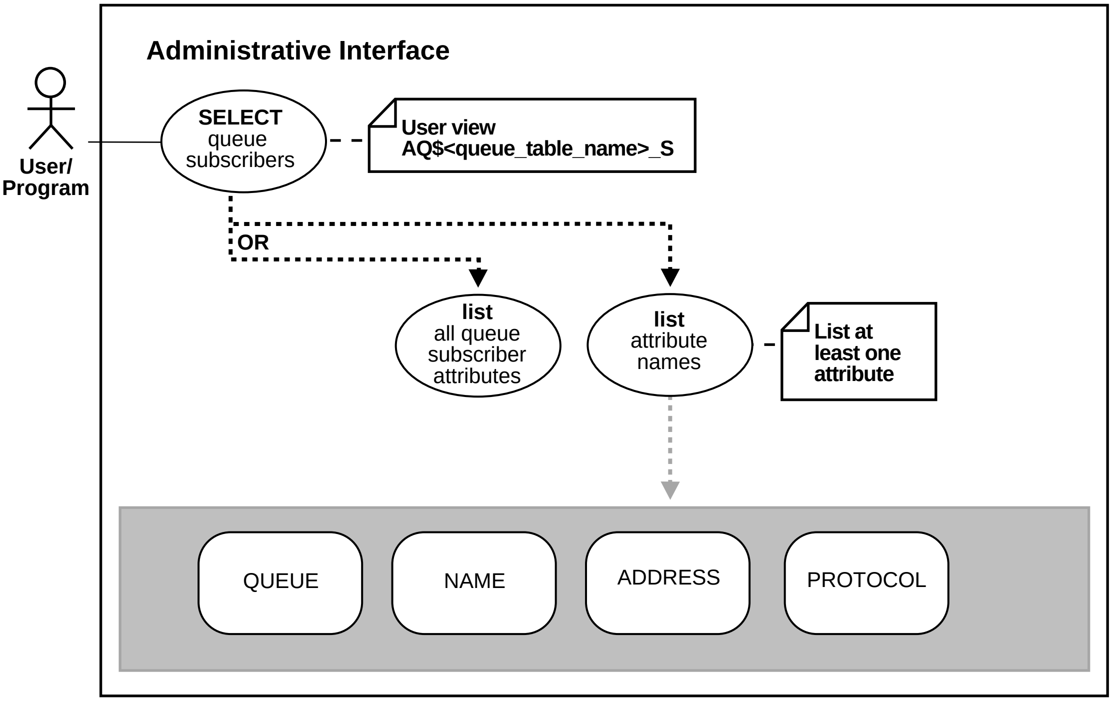
Column Name & Description	Null?	Type
LAST_RUN_DATE — The date on the last successful execution		DATE
LAST_RUN_TIME — The time of the last successful execution in HH:MI:SS format		VARCHAR2(8)
CURRENT_START_DATE — Date at which the current window of this schedule was started		DATE
CURRENT_START_TIME — Time of day at which the current window of this schedule was started in HH:MI:SS format		VARCHAR2(8)
NEXT_RUN_DATE — Date at which the next window of this schedule will be started		DATE
NEXT_RUN_TIME — Time of day at which the next window of this schedule will be started in HH:MI:SS format		VARCHAR2(8)
TOTAL_TIME — Total time in seconds spent in propagating messages from the schedule		NUMBER
TOTAL_NUMBER — Total number of messages propagated in this schedule		NUMBER
TOTAL_BYTES — Total number of bytes propagated in this schedule		NUMBER
MAX_NUMBER — The maximum number of messages propagated in a propagation window		NUMBER
MAX_BYTES — The maximum number of bytes propagated in a propagation window		NUMBER
AVG_NUMBER — The average number of messages propagated in a propagation window		NUMBER
AVG_SIZE — The average size of a propagated message in bytes		NUMBER

Table 5–11 USER_QUEUE_SCHEDULES

Column Name & Description	Null?	Type
`AVG_TIME — The average time, in seconds, to propagate a message`		`NUMBER`
`FAILURES — The number of times the execution failed. If 16, the schedule will be disabled`		`NUMBER`
`LAST_ERROR_DATE — The date of the last unsuccessful execution`		`DATE`
`LAST_ERROR_TIME — The time of the last unsuccessful execution`		`VARCHAR2(8)`
`LAST_ERROR_MSG — The error number and error message text of the last unsuccessful execution`		`VARCHAR2(4000)`

Select Queue Subscribers

Figure 5–12 Use Case Diagram: Select Queue Subscribers



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Administrative Interface — Views" on page 5-2

Name of View:

`AQ$<queue_table_name>_S`

Purpose:

This is a view of all the subscribers for all the queues in any given queue table. This view is generated when the queue table is created and is called `aq$<queue_table_name>_s`.This view is used to query subscribers for any or all the queues in this queue table. Note that this view is only created for 8.1-compatible queue tables.

Table 5–12 AQ$<queue_table_name>_S

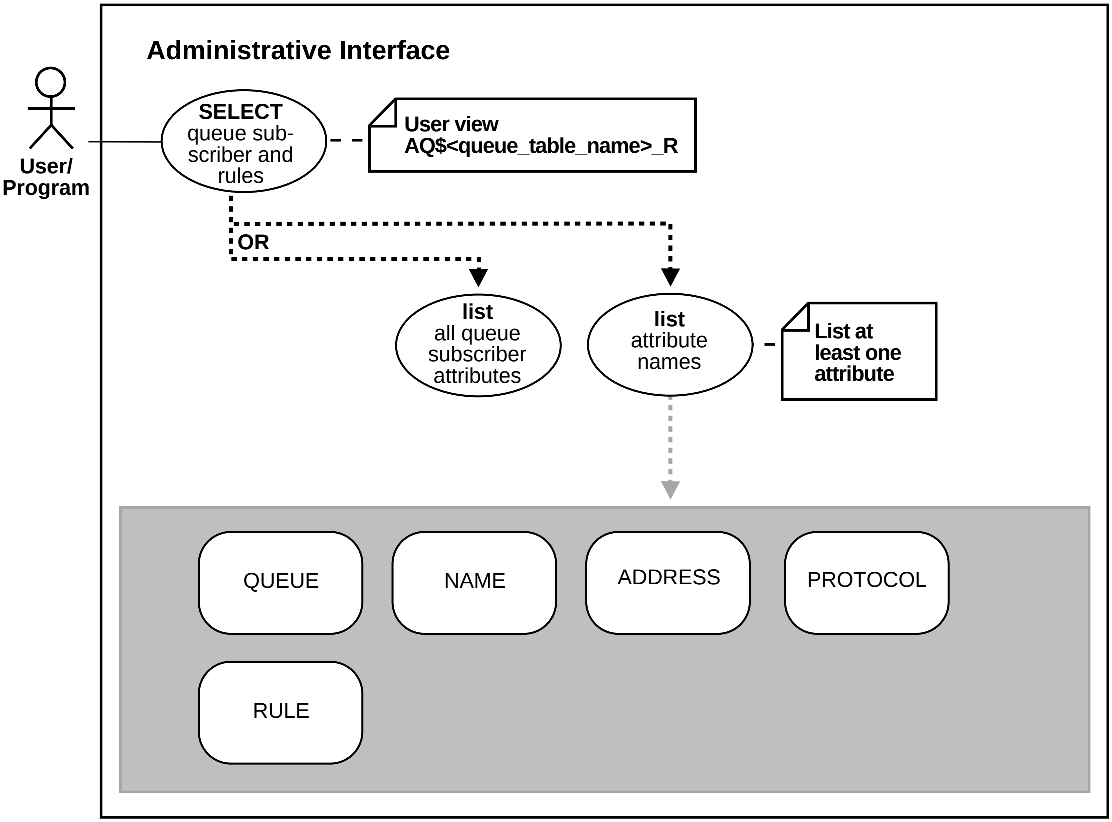
Column Name & Description	Null?	Type
`QUEUE - name of Queue for which subscriber is defined`	`NOT NULL`	`VARCHAR2(30)`
`NAME - name of Agent`		`VARCHAR2(30)`
`ADDRESS - address of Agent`		`VARCHAR2(1024)`
`PROTOCOL - protocol of Agent`		`NUMBER`

Usage Notes

For queues created in 8.1-compatible queue tables, this view provides functionality that is equivalent to the dbms_aqadm.queue_subscribers() procedure. For these queues, it is recommended that the view be used instead of this procedure to view queue subscribers.

Select Queue Subscribers and their Rules

Figure 5–13 Use Case Diagram: Select Queue Subscribers and their Rules



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Administrative Interface — Views" on page 5-2

Name of View:

`AQ$<queue_table_name>_R`

Purpose:

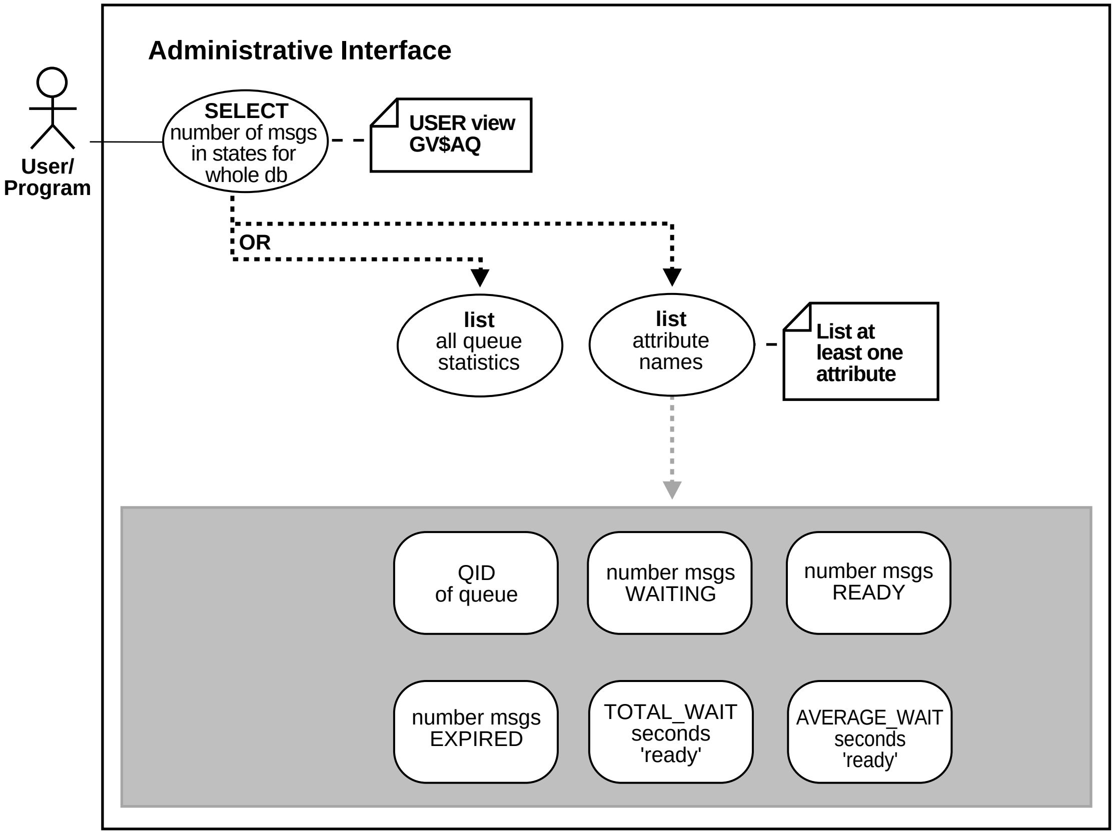
This view displays only the rule based subscribers for all queues in a given queue table including the text of the rule defined by each subscriber. This is a view of subscribers with rules defined on any queues of a given queue table. This view is generated when the queue table is created and is called aq$<queue_table_name>_r. It is used to query subscribers for any or all the queues in this queue table. Note that this view is only created for 8.1-compatible queue tables.

Table 5–13 AQ$<queue_table_name>_R

Column Name & Description	Null?	Type
`QUEUE - name of Queue for which subscriber is defined`	`NOT NULL`	`VARCHAR2(30)`
`NAME - name of Agent`		`VARCHAR2(30)`
`ADDRESS - address of Agent`		`VARCHAR2(1024)`
`PROTOCOL - protocol of Agent`		`NUMBER`
`RULE - text of defined rule`		`VARCHAR2(30)`

Select the Number of Messages in Different States for the Whole Database

Figure 5–14 GV$AQ



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Administrative Interface — Views" on page 5-2

Name of View:

`GV$AQ`

Purpose:

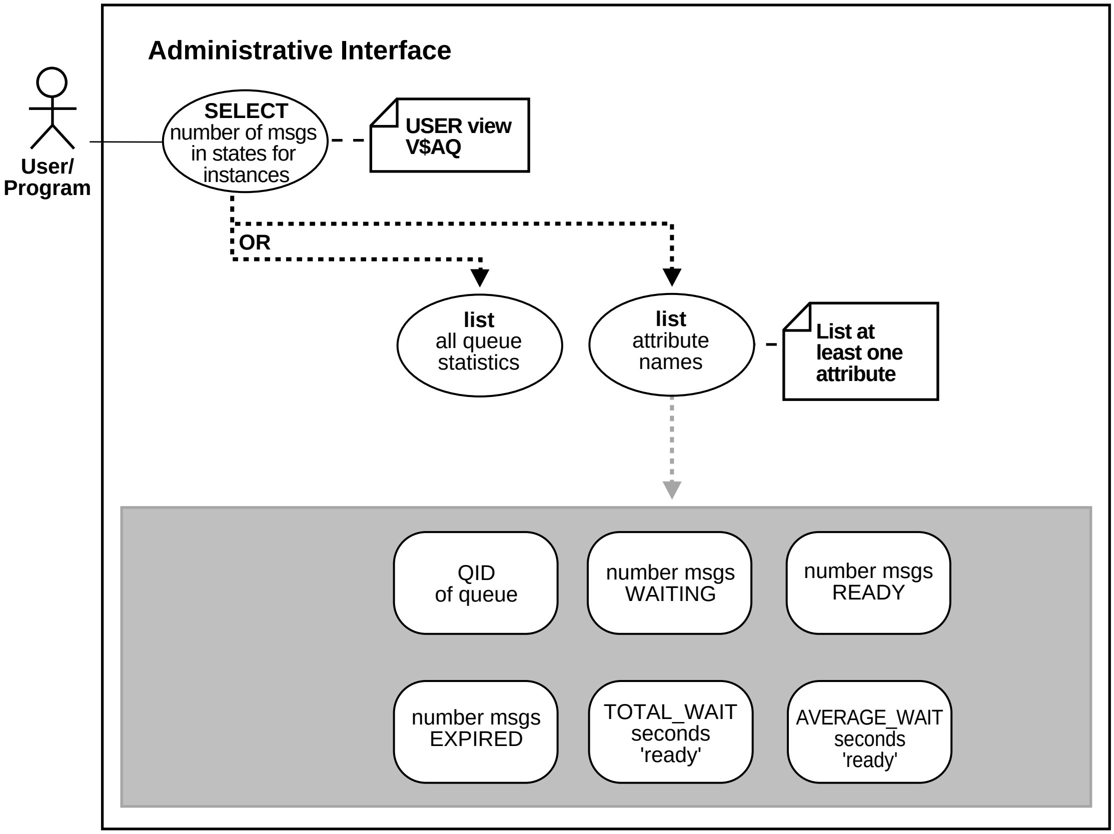
Provides information about the number of messages in different states for the whole database.

Table 5–14 AQ$<queue_table_name>_R

Column Name & Description	Null?	Type
`QID` — the identity of the queue. This is the same as the `qid` in `user_queues` and `dba_queues`.		`NUMBER`
`WAITING` — the number of messages in the state '`WAITING`'.		`NUMBER`
`READY` — the number of messages in state '`READY`'.		`NUMBER`
`EXPIRED` — the number of messages in state '`EXPIRED`'.		`NUMBER`
`TOTAL_WAIT` — the number of seconds for which messages in the queue have been waiting in state '`READY`'		`NUMBER`
`AVERAGE_WAIT` — the average number of seconds a message in state '`READY`' has been waiting to be dequeued.		`NUMBER`

Select the Number of Messages in Different States for Specific Instances

Figure 5–15 V$AQ



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Administrative Interface — Views" on page 5-2

Name of View:

`V$AQ`

Purpose:

Provides information about the number of messages in different states for specific instances.

Table 5–15 AQ$<queue_table_name>_R

Column Name & Description	Null?	Type
`QID` — the identity of the queue. This is the same as the `qid` in `user_queues` and `dba_queues`.		`NUMBER`
`WAITING` — the number of messages in the state '`WAITING`'.		`NUMBER`
`READY` — the number of messages in state '`READY`'.		`NUMBER`
`EXPIRED` — the number of messages in state '`EXPIRED`'.		`NUMBER`
`TOTAL_WAIT` — the number of seconds for which messages in the queue have been waiting in state '`READY`'		`NUMBER`
`AVERAGE_WAIT` — the average number of seconds a message in state '`READY`' has been waiting to be dequeued.		`NUMBER`

6

Operational Interface: Basic Operations

In this chapter we describe the operational interface to Oracle Advanced Queuing in terms of use cases. That is, we discuss each operation (such as "Enqueue a Message") as a use case by that name. The table listing all the use cases is provided at the head of the chapter (see "Use Case Model: Operational Interface — Basic Operations" on page 6-2).

A summary figure, "Use Case Diagram: Operational Interface — Basic Operations", locates all the use cases in single drawing. If you are using the HTML version of this document, you can use this figure to navigate to the use case in which you are interested by clicking on the relevant use case title.

The individual use cases are themselves laid out as follows:

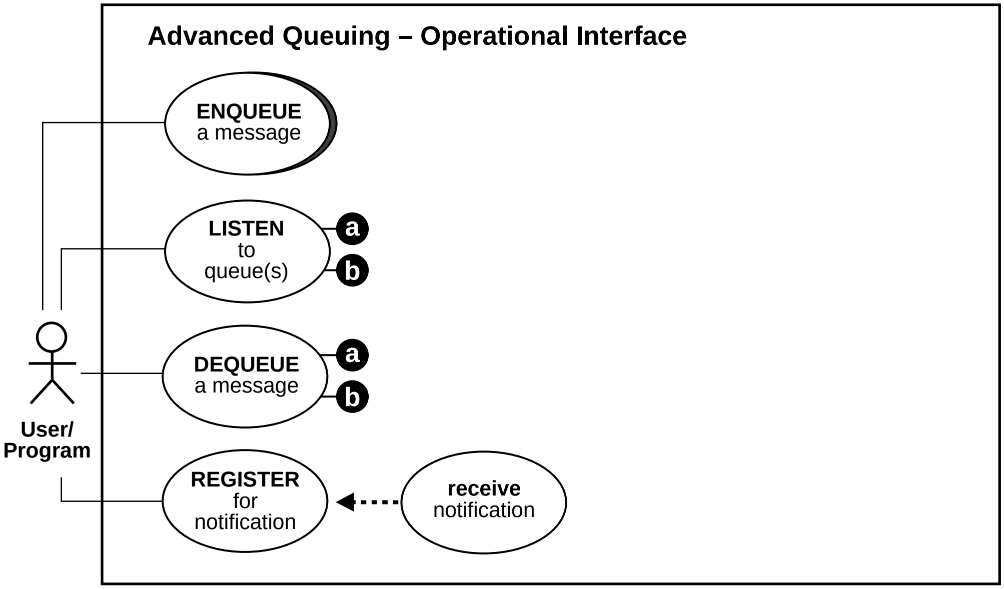
- A figure that depicts the use case (see "Preface" for a description of how to interpret these diagrams).
- A listing of the syntax.
- Basic examples
- Usage Notes, if any.

Use Case Model: Operational Interface — Basic Operations

Table 6–1 Use Case Model: Operational Interface

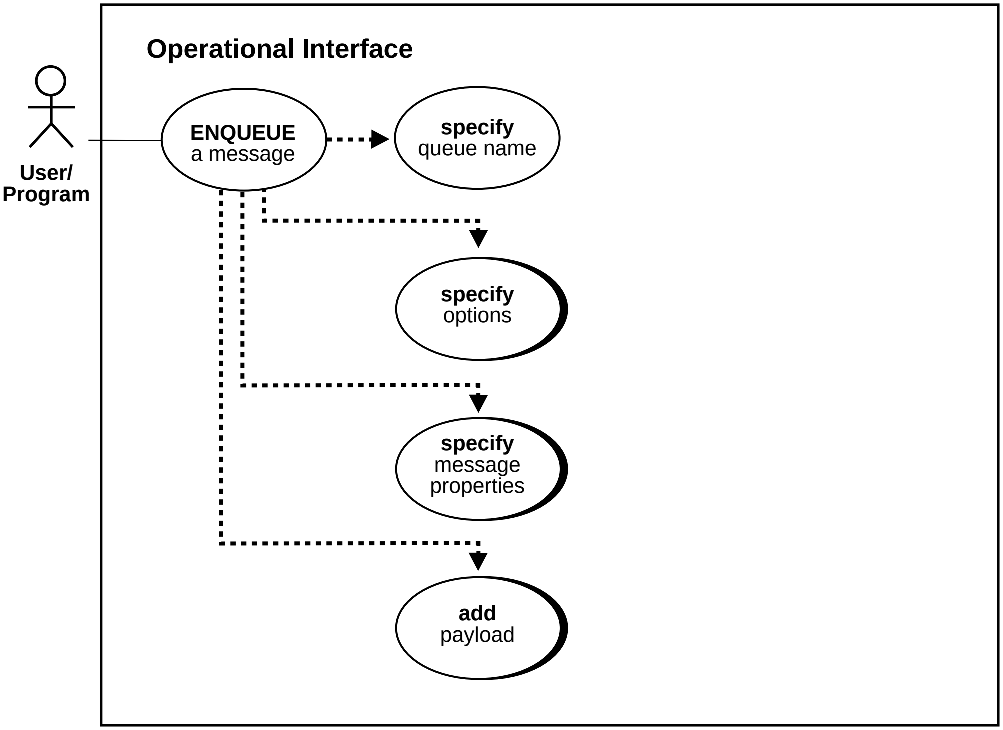
Use Case
Enqueue a Message on page 6-4
Enqueue a Message [Specify Options] on page 6-7
Enqueue a Message [Specify Message Properties] on page 6-9
Enqueue a Message [Add Payload] on page 6-15
Listen to One (Many) Queue(s) on page 6-18
Listen to One (Many) Single-Consumer Queue(s) on page 6-20
Listen to One (Many) Multi-Consumer Queue(s) on page 6-30
Dequeue a Message on page 6-38
Dequeue a Message from a Single-Consumer Queue [Specify Options] on page 6-41
Dequeue a Message from a Multi-Consumer Queue [Specify Options] on page 6-46
Register for Notification on page 6-50
Register for Notification [Specify Subscription Name — Single-Consumer Queue] on page 6-54
Register for Notification [Specify Subscription Name — Multi-Consumer Queue] on page 6-55

Figure 6–1 Use Case Model Diagram: Operational Interface



Enqueue a Message

Figure 6–2 Use Case Diagram: Enqueue a Message



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Operational Interface — Basic Operations" on page 6-2

Purpose:

Adds a message to the specified queue.

Syntax:

```
DBMS_AQ.ENQUEUE (
    Queue_name          IN      VARCHAR2,
    Enqueue_options     IN      enqueue_options_t,
    Message_properties  IN      message_properties_t,
    Payload             IN      "<type_name>",
    Msgid               OUT     RAW);
```

Usage:

Table 6–2 DBMS_AQ.ENQUEUE

Parameter	Description
`queue_name` `(IN VARCHAR2)`	Specifies the name of the queue to which this message should be enqueued. The queue cannot be an exception queue.
`enqueue_options` `(IN enqueue_option_t)`	For the definition please refer to the section titled Enqueue a Message [Specify Options]
`message_properties` `(IN message_properties_t)`	For the definition please refer to the section titled "Message Properties."
`payload` `(IN "<type_name>")`	Not interpreted by Oracle AQ. The payload must be specified according to the specification in the associated queue table. `NULL` is an acceptable parameter. For the definition of <type_name> please refer to section titled "Type name".
`msgid` `(OUT RAW)`	The system generated identification of the message. This is a globally unique identifier that can be used to identify the message at dequeue time.

Usage Notes

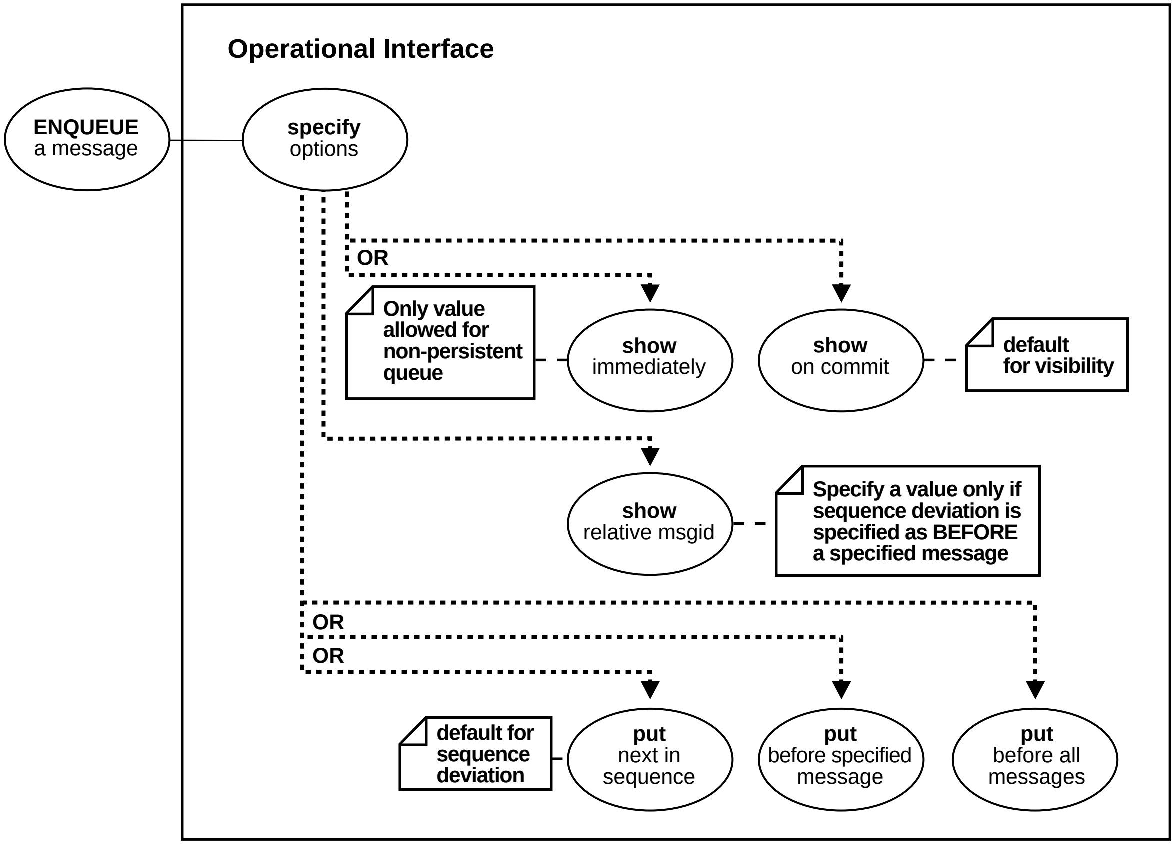
- The `sequence_deviation` parameter in enqueue_options can be used to change the order of processing between two messages. The identity of the other message, if any, is specified by the enqueue_options parameter relative_msgid. The relationship is identified by the `sequence_deviation` parameter.

  Specifying `sequence_deviation` for a message introduces some restrictions for the delay and priority values that can be specified for this message. The delay of this message has to be less than or equal to the delay of the message before which this message is to be enqueued. The priority of this message has to be greater than or equal to the priority of the message before which this message is to be enqueued.

- The visibility option must be immediate for non-persistent queues.
- Only local recipients are supported are supported for non-persistent queues.
- If a message is enqueued to a multi-consumer queue with no recipient and the queue has no subscribers (or rule-based subscribers that match this message) then, the Oracle error ORA 24033 is raised. This is a warning that the message will be discarded since there are no recipients or subscribers to whom it can be delivered.

Enqueue a Message [Specify Options]

Figure 6–3 Use Case Diagram: Enqueue a Message [Specify Options]



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Operational Interface — Basic Operations" on page 6-2

Purpose:

To specify the options available for the enqueue operation.

Syntax:

```
TYPE Enqueue_options_t IS RECORD (
   Visibility           BINARY_INTEGER DEFAULT ON_COMMIT,
   Relative_msgid       RAW(16) DEFAULT NULL,
   Sequence_deviation   BINARY_INTEGER DEFAULT NULL);
```

Usage:

Table 6–3 Enqueue a Message [Specify Options]

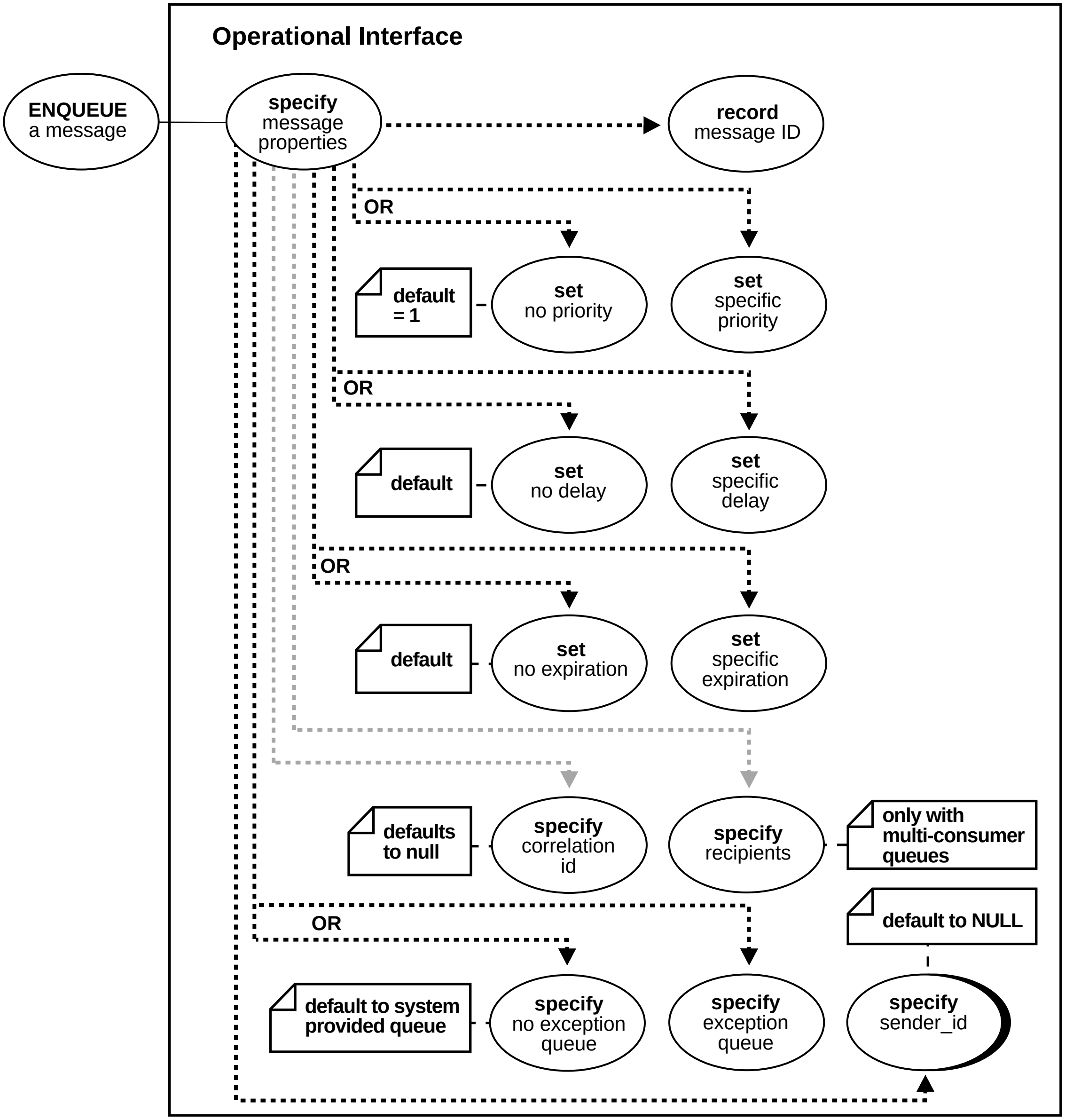
Parameter	Description
`visibility`	Specifies the transactional behavior of the enqueue request. `ON_COMMIT`: The enqueue is part of the current transaction. The operation is complete when the transaction commits. This is the default case. `IMMEDIATE`: The enqueue is not part of the current transaction. The operation constitutes a transaction on its own. This is the only value allowed when enqueuing to a non-persistent queue.
`relative_msgid`	Specifies the message identifier of the message which is referenced in the sequence deviation operation. This field is valid if and only if `BEFORE` is specified in `sequence_deviation`. This parameter will be ignored if sequence deviation is not specified.
`sequence_deviation`	Specifies if the message being enqueued should be dequeued before other message(s) already in the queue. `BEFORE`: The message is enqueued ahead of the message specified by `relative_msgid.` `TOP`: The message is enqueued ahead of any other messages. `NULL`: Default

Usage Notes

Do not use the `immediate` option when you want to use LOB locators since LOB locators are valid only for the duration of the transaction. As the `immediate` option automatically commits the transaction, your locator will not be valid.

Enqueue a Message [Specify Message Properties]

Figure 6–4 Use Case Diagram: Enqueue a Message [Specify Message Properties]



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Operational Interface — Basic Operations" on page 6-2

Purpose:

The *Message Properties* describe the information that is used by AQ to manage individual messages. These are set at enqueue time and their values are returned at dequeue time.

Syntax:

```
TYPE Message_properties_t IS RECORD (
    Priority              BINARY_INTEGER DEFAULT 1,
    Delay                 BINARY_INTEGER DEFAULT NO_DELAY,
    Expiration            BINARY_INTEGER DEFAULT NEVER,
    Correlation           VARCHAR2(128) DEFAULT NULL,
    Attempts              BINARY_INTEGER,
    Recipient_list        aq$_recipient_list_t,
    Exception_queue       VARCHAR2(51) DEFAULT NULL,
    Enqueue_time          DATE,
    State                 BINARY_INTEGER,
    Sender_id             aq$_agent DEFAULT NULL,
    Original_msgid        RAW(16) DEFAULT NULL);

TYPE aq$_recipient_list_t IS TABLE OF aq$_agent
    INDEX BY BINARY_INTEGER;
```

Usage:

Table 6–4 Message properties

Parameter	Description
`priority` `(BINARY_INTEGER)`	Specifies/returns the priority of the message. A smaller number indicates higher priority. The priority can be any number, including negative numbers.
`delay` `(BINARY_INTEGER)`	Specifies/returns the delay of the enqueued message. The delay represents the number of seconds after which a message is available for dequeuing. Dequeuing by msgid overrides the delay specification. A message enqueued with delay set will be in the `WAITING` state, when the delay expires the messages goes to the `READY` state. `DELAY` processing requires the queue monitor to be started. Note that delay is set by the producer who enqueues the message. `NO_DELAY`: the message is available for immediate dequeuing. number: the number of seconds to delay the message.
`expiration` `(BINARY_INTEGER)`	Specifies/returns the expiration of the message. It determines, in seconds, the duration the message is available for dequeuing. This parameter is an offset from the delay. Expiration processing requires the queue monitor to be running. `NEVER`: message will not expire. number: number of seconds message will remain in `READY` state. If the message is not dequeued before it expires, it will be moved to the exception queue in the `EXPIRED` state.
`correlation` `(VARCHAR2(128))`	Returns the identification supplied by the producer for a message at enqueuing.
`attempts` `(BINARY_INTEGER)`	Returns the number of attempts that have been made to dequeue this message. This parameter can not be set at enqueue time.
`recipient_list` `(aq$_recipient_list_t)`	For type definition please refer to section titled "Agent". This parameter is only valid for queues which allow multiple consumers. The default recipients are the queue subscribers. This parameter is not returned to a consumer at dequeue time.
`exception_queue` `(VARCHAR2(51))`	Specifies/returns the name of the queue to which the message is moved if it cannot be processed successfully. Messages are moved in two cases: The number of unsuccessful dequeue attempts has exceeded *max_retries* or the message has expired. All messages in the exception queue are in the `EXPIRED` state. The default is the exception queue associated with the queue table. If the exception queue specified does not exist at the time of the move the message will be moved to the default exception queue associated with the queue table and a warning will be logged in the alert file. If the default exception queue is used the parameter will return a `NULL` value at dequeue time.

Table 6–4 Message properties

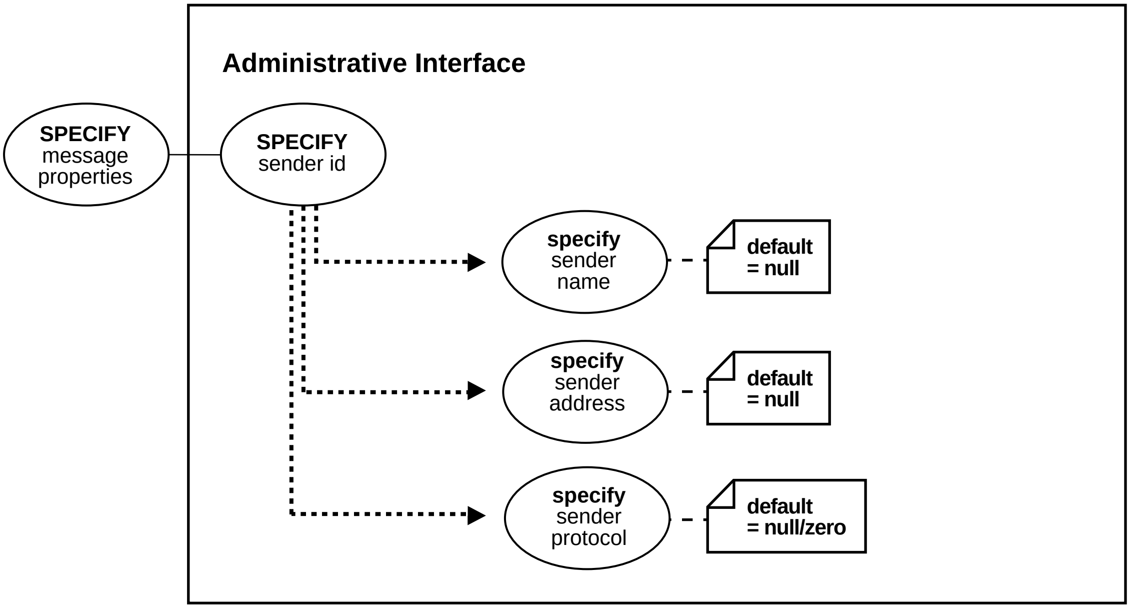
Parameter	Description
`enqueue_time` `(DATE)`	Returns the time the message was enqueued. This value is determined by the system and cannot be set by the user. This parameter can not be set at enqueue time.
`state` `(BINARY_INTEGER)`	Returns the state of the message at the time of the dequeue. This parameter can not be set at enqueue time. `0`: The message is ready to be processed. `1`: The message delay has not yet been reached. `2`: The message has been processed and is retained. `3`: The message has been moved to the exception queue.
`sender_id` `(aq$_agent)`	Specifies/returns the application-specified sender identification. `DEFAULT`: `NULL`
`original_msgid` `(RAW(16))`	This parameter is used by Oracle AQ for propagating messages. `DEFAULT`: `NULL`

Usage Notes

- To view messages in a waiting or processed state, you can either dequeue or browse by message ID, or use SELECT statements.
- Message delay and expiration are enforced by the queue monitor (QMN) background processes. You should remember to start the QMN processes for the database if you intend to use the delay and expiration features of AQ.

Enqueue a Message [Specify Message Properties [Specify Sender ID]]

Figure 6–5 Use Case Diagram: Enqueue a Message [Specify Message Properties [Specify Sender ID]]



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Operational Interface — Basic Operations" on page 6-2

Purpose:

To identify the sender (producer) of a message.

Syntax:

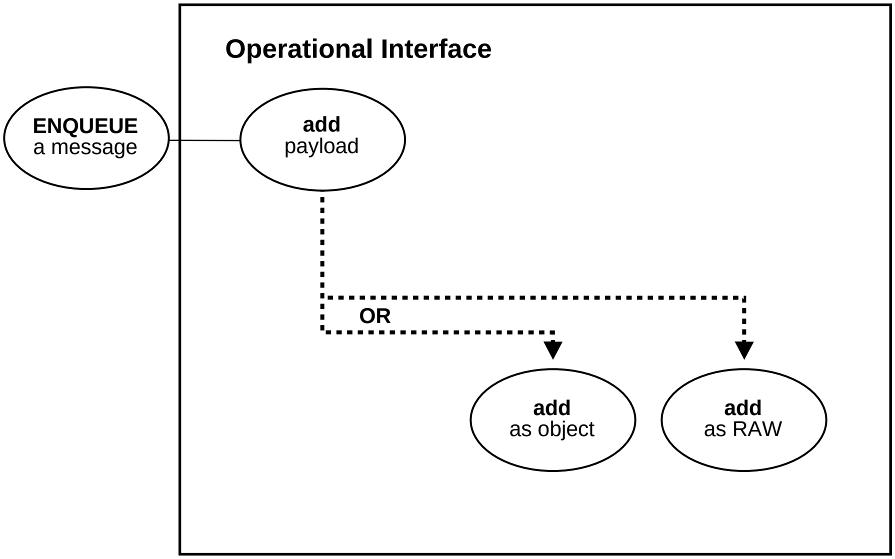
```
TYPE aq$_agent IS OBJECT (
    Name            VARCHAR2(30),
    Address         VARCHAR2(1024),
    Protocol        NUMBER);
```

For more information about Agent see:

- "Agent" on page 3-5

Enqueue a Message [Add Payload]

Figure 6–6 Use Case Diagram: Enqueue a Message [Add Payload]



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Operational Interface — Basic Operations" on page 6-2

Usage Notes

To store a payload of type `RAW`, AQ will create a queue table with `LOB` column as the payload repository. The maximum size of the payload is determined by which programmatic environment you use to access AQ. For PL/SQL, Java and precompilers the limit is 32K; for the OCI the limit is 4G.

Example: Enqueue of Object Type Messages

Note: You may need to set up the following data structures for certain examples to work:

```
CONNECT system/manager
CREATE USER aq IDENTIFIED BY aq;
GRANT Aq_administrator_role TO aq;
EXECUTE DBMS_AQADM.CREATE_QUEUE_TABLE (
   Queue_table           =>  'aq.objmsgs_qtab',
   Queue_payload_type    =>  'aq.message_typ');
EXECUTE DBMS_AQADM.CREATE_QUEUE (
   Queue_name           =>  'aq.msg_queue',
   Queue_table          =>  'aq.objmsgs_qtab');
EXECUTE DBMS_AQADM.START_QUEUE (
   Queue_name        => 'aq.msg_queue',
   Enqueue           => TRUE);
EXECUTE DBMS_AQADM.CREATE_QUEUE_TABLE (
   Queue_table           => 'aq.prioritymsgs_qtab',
   Sort_list             => 'PRIORITY,ENQ_TIME',
   Queue_payload_type    => 'aq.message_typ');
EXECUTE DBMS_AQADM.CREATE_QUEUE (
   Queue_name            => 'aq.priority_msg_queue',
   Queue_table           => 'aq.prioritymsgs_qtab');
EXECUTE DBMS_AQADM.START_QUEUE (
   Queue_name            => 'aq.priority_msg_queue',
   Enqueue               => TRUE);
```

Enqueue a Single Message and Specify the Queue Name and Payload.

```
/* Enqueue to msg_queue: */
DECLARE
   Enqueue_options     DBMS_AQ.enqueue_options_t;
   Message_properties  DBMS_AQ.message_properties_t;
   Message_handle      RAW(16);
   Message             aq.message_typ;

BEGIN
   Message := aq.message_typ('NORMAL MESSAGE',
      'enqueued to msg_queue first.');

   DBMS_AQ.ENQUEUE(queue_name => 'msg_queue',
   Enqueue_options            => enqueue_options,
```

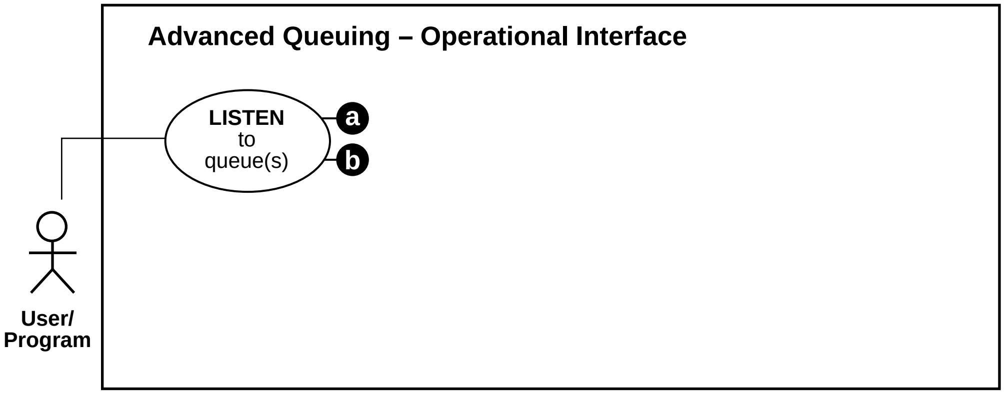
```
   Message_properties          => message_properties,
   Payload                     => message,
   Msgid                       => message_handle);

   COMMIT;
END;
```

Enqueue a Single Message and Specify the Priority

```
/* The queue name priority_msg_queue is defined as an object type queue table.
   The payload object type is message. The schema of the queue is aq.  */

 /* Enqueue a message with priority 30: */
DECLARE
   Enqueue_options       dbms_aq.enqueue_options_t;
   Message_properties    dbms_aq.message_properties_t;
   Message_handle        RAW(16);
   Message               aq.Message_typ;

BEGIN
   Message := Message_typ('PRIORITY MESSAGE', 'enqued at priority 30.');

   message_properties.priority := 30;

   DBMS_AQ.ENQUEUE(queue_name => 'priority_msg_queue',
   enqueue_options             => enqueue_options,
   message_properties          => message_properties,
   payload                     => message,
   msgid                       => message_handle);

   COMMIT;
END;
```

Listen to One (Many) Queue(s)

Figure 6–7 Use Case Diagram: Listen to One(Many) Queue(s)



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Operational Interface — Basic Operations" on page 6-2

Purpose:

To monitor one or more queues on behalf of a list of agents.

Syntax:

```
DBMS_AQ.LISTEN (
   agent_list IN   aq$_agent_list_t,
   wait       IN   BINARY_INTEGER default DBMS_AQ.FOREVER,
   agent      OUT  aq$_agent);

TYPE aq$_agent_list_t IS TABLE of aq$_agent INDEX BY BINARY_INTEGER;
```

Usage:

Table 6–5 DBMS_AQADM.LISTEN

Parameter	Description
`agent_list (aq$_agent_list)`	The list of agents for which to 'listen'.
`wait (integer default DBMS_AQ.FOREVER)`	The time-out for the listen call (seconds). By default, the call will block forever.
`agent (aq$_ agent)`	The agent with a message available for consumption.

Usage Notes

The call takes a list of agents as an argument. You specify the queue to be monitored in the address field of each agent listed. You also must specify the name of the agent when monitoring multiconsumer queues. For single-consumer queues, an agent name must not be specified. Only local queues are supported as addresses. Protocol is reserved for future use.

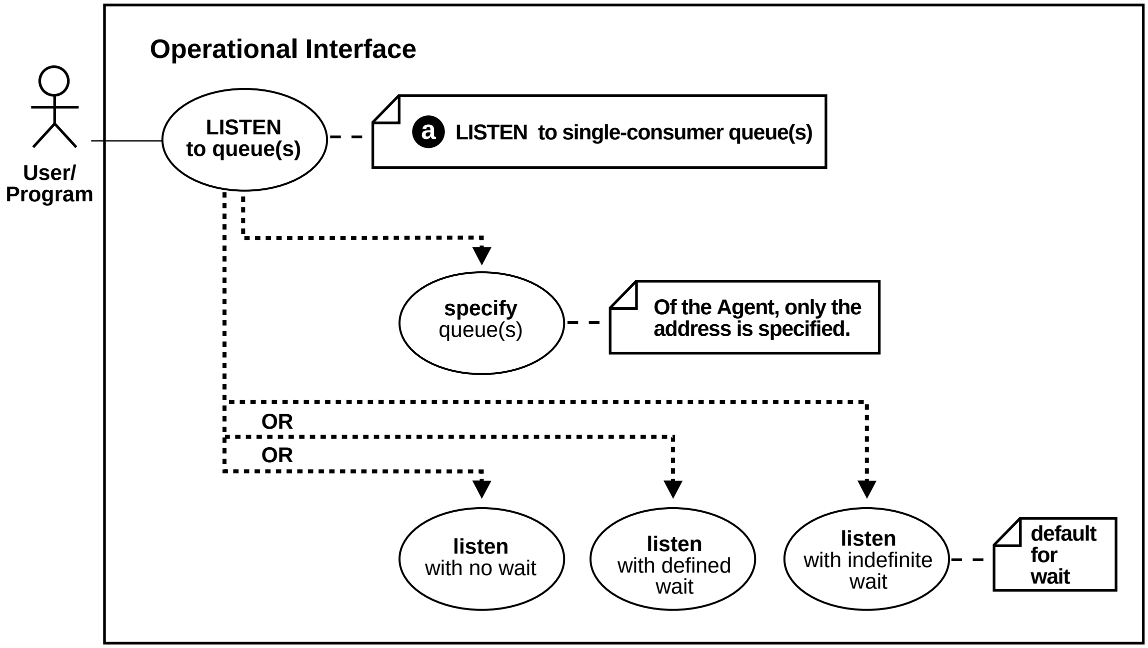
This is a blocking call that returns when there is a message ready for consumption for an agent in the list. If there are messages for more than one agent, only the first agent listed is returned. If there are no messages found when the wait time expires, an error is raised.

A successful return from the `listen` call is only an indication that there is a message for one of the listed agents in one the specified queues. The interested agent must still dequeue the relevant message.

Note that you cannot call `listen` on non-persistent queues.

Listen to One (Many) Single-Consumer Queue(s)

Figure 6–8 Use Case Diagram: Listen to One(Many) Single-Consumer Queue(s)



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Operational Interface — Basic Operations" on page 6-2

Example: Listen to Queue(s) Using PL/SQL (DBMS_AQ Package)

```
/* The listen call allows you to monitor a list of queues for messages for
   specific agents. You need to have dequeue privileges for all the queues
   you wish to monitor. */
```

Listen to Single-Consumer Queue (Timeout of Zero).

```
DECLARE
   Agent_w_msg      aq$_agent;
   My_agent_list    dbms_aq.agent_list_t;

BEGIN
   /* NOTE:  MCQ1, MCQ2, MCQ3 are multi-consumer queues  in SCOTT’s schema
   *        SCQ1, SCQ2, SCQ3 are single consumer queues in SCOTT’s schema
   */

   Qlist(1):= aq$_agent(NULL, 'scott.SCQ1',  NULL);
   Qlist(2):= aq$_agent(NULL, 'SCQ2', NULL);
   Qlist(3):= aq$_agent(NULL, 'SCQ3', NULL);

   /* Listen with a time-out of zero: */
   DBMS_AQ.LISTEN(
      Agent_list   =>   My_agent_list,
      Wait         =>   0,
      Agent        =>   agent_w_msg);
   DBMS_OUTPUT.PUT_LINE('Message in Queue :- ' ||  agent_w_msg.address);
   DBMS_OUTPUT.PUT_LINE('');
END;
```

Example: Listen to Single-Consumer Queue(s) Using C (OCI)

Listening for Single Consumer Queues with Zero Timeout

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

static void checkerr(errhp, status)
OCIError *errhp;
sword status;
{
    text errbuf[512];
```

```
    ub4 buflen;
    sb4 errcode;

    switch (status)
    {
   case OCI_SUCCESS:
       break;
   case OCI_SUCCESS_WITH_INFO:
       printf("Error - OCI_SUCCESS_WITH_INFO\n");
       break;
   case OCI_NEED_DATA:
       printf("Error - OCI_NEED_DATA\n");
       break;
   case OCI_NO_DATA:
       printf("Error - OCI_NO_DATA\n");
       break;
   case OCI_ERROR:
       OCIErrorGet ((dvoid *) errhp, (ub4) 1, (text *) NULL, &errcode,
       errbuf, (ub4) sizeof(errbuf), (ub4) OCI_HTYPE_ERROR);
       printf("Error - %s\n", errbuf);
       break;
   case OCI_INVALID_HANDLE:
       printf("Error - OCI_INVALID_HANDLE\n");
       break;
   case OCI_STILL_EXECUTING:
       printf("Error - OCI_STILL_EXECUTE\n");
       break;
   case OCI_CONTINUE:
       printf("Error - OCI_CONTINUE\n");
       break;
   default:
   break;
    }
}

/* set agent into descriptor */
void SetAgent(agent, appname, queue,errhp)

OCIAQAgent  *agent;
text        *appname;
text        *queue;
OCIError    *errhp;
{

  OCIAttrSet(agent, OCI_DTYPE_AQAGENT,
```

```
      appname ? (dvoid *)appname : (dvoid *)"",
      appname ? strlen((const char *)appname) : 0,
         OCI_ATTR_AGENT_NAME, errhp);

  OCIAttrSet(agent, OCI_DTYPE_AQAGENT,
      queue ? (dvoid *)queue : (dvoid *)"",
      queue ? strlen((const char *)queue) : 0,
         OCI_ATTR_AGENT_ADDRESS, errhp);

  printf("Set agent name to %s\n", appname ? (char *)appname : "NULL");
  printf("Set agent address to %s\n", queue ? (char *)queue : "NULL");
}

/* get agent from descriptor */
void GetAgent(agent, errhp)
OCIAQAgent *agent;
OCIError   *errhp;
{
text      *appname;
text      *queue;
ub4       appsz;
ub4       queuesz;

  if (!agent )
  {
    printf("agent was NULL \n");
    return;
  }
  checkerr(errhp, OCIAttrGet(agent, OCI_DTYPE_AQAGENT,
     (dvoid *)&appname, &appsz, OCI_ATTR_AGENT_NAME, errhp));
  checkerr(errhp, OCIAttrGet(agent, OCI_DTYPE_AQAGENT,
     (dvoid *)&queue, &queuesz, OCI_ATTR_AGENT_ADDRESS, errhp));
  if (!appsz)
     printf("agent name: NULL\n");
  else printf("agent name: %.*s\n", appsz, (char *)appname);
  if (!queuesz)
     printf("agent address: NULL\n");
  else printf("agent address: %.*s\n", queuesz, (char *)queue);
}

int main()
{
  OCIEnv *envhp;
  OCIServer *srvhp;
  OCIError *errhp;
```

```
 OCISvcCtx *svchp;
 OCISession *usrhp;
 OCIAQAgent *agent_list[3];
 OCIAQAgent *agent = (OCIAQAgent *)0;
 /* added next 2 121598 */
 int i;

/* Standard OCI Initialization */

 OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,  (dvoid * (*)()) 0,
    (dvoid * (*)()) 0,  (void (*)()) 0 );

 OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp,
           (ub4) OCI_HTYPE_ENV, 0, (dvoid **) 0);

 OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 0, (dvoid **) 0);

 OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
    0, (dvoid **) 0);

 OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
    0, (dvoid **) 0);

 OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

 OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
    0, (dvoid **) 0);

 /* set attribute server context in the service context */
 OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
    (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

 /* allocate a user context handle */
 OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp, (ub4) OCI_HTYPE_SESSION,
     (size_t) 0, (dvoid **) 0);

 /* allocate a user context handle */
 OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp, (ub4) OCI_HTYPE_SESSION,
    (size_t) 0, (dvoid **) 0);

 OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION,
    (dvoid *)"scott", (ub4)strlen("scott"), OCI_ATTR_USERNAME, errhp);

 OCIAttrSet((dvoid *) usrhp, (ub4) OCI_HTYPE_SESSION,
     (dvoid *) "tiger", (ub4) strlen("tiger"),
```

```
      (ub4) OCI_ATTR_PASSWORD, errhp);

  OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS, OCI_DEFAULT);

  OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX,
     (dvoid *)usrhp, (ub4)0, OCI_ATTR_SESSION, errhp);

  /* AQ LISTEN Initialization - allocate agent handles */
  for (i = 0; i < 3; i++)
  {
     agent_list[i] = (OCIAQAgent *)0;
     OCIDescriptorAlloc(envhp, (dvoid **)&agent_list[i],
         OCI_DTYPE_AQAGENT, 0, (dvoid **)0);
  }

  /*
   *   SCQ1, SCQ2, SCQ3 are single consumer queues in SCOTT's schema
   */

  SetAgent(agent_list[0], (text *)0, "SCOTT.SCQ1", errhp);
  SetAgent(agent_list[1], (text *)0, "SCOTT.SCQ2", errhp);
  SetAgent(agent_list[2], (text *)0, "SCOTT.SCQ3", errhp);

  checkerr(errhp,OCIAQListen(svchp, errhp, agent_list, 3, 0, &agent, 0));

  printf("MESSAGE for :- \n");
  GetAgent(agent, errhp);
  printf("\n");

}
```

Listening for Single Consumer Queues with Timeout of 120 Seconds

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

static void checkerr(errhp, status)
OCIError *errhp;
sword status;
{
    text errbuf[512];
    ub4 buflen;
```

```
    sb4 errcode;

    switch (status)
    {
   case OCI_SUCCESS:
       break;
   case OCI_SUCCESS_WITH_INFO:
       printf("Error - OCI_SUCCESS_WITH_INFO\n");
       break;
   case OCI_NEED_DATA:
       printf("Error - OCI_NEED_DATA\n");
       break;
   case OCI_NO_DATA:
       printf("Error - OCI_NO_DATA\n");
       break;
   case OCI_ERROR:
       OCIErrorGet ((dvoid *) errhp, (ub4) 1, (text *) NULL, &errcode,
       errbuf, (ub4) sizeof(errbuf), (ub4) OCI_HTYPE_ERROR);
       printf("Error - %s\n", errbuf);
       break;
   case OCI_INVALID_HANDLE:
       printf("Error - OCI_INVALID_HANDLE\n");
       break;
   case OCI_STILL_EXECUTING:
       printf("Error - OCI_STILL_EXECUTE\n");
       break;
   case OCI_CONTINUE:
       printf("Error - OCI_CONTINUE\n");
       break;
   default:
   break;
    }
}

/* set agent into descriptor */
/* void SetAgent(agent, appname, queue) */
void SetAgent(agent, appname, queue,errhp)

OCIAQAgent  *agent;
text        *appname;
text        *queue;
OCIError    *errhp;
{

  OCIAttrSet(agent, OCI_DTYPE_AQAGENT,
```

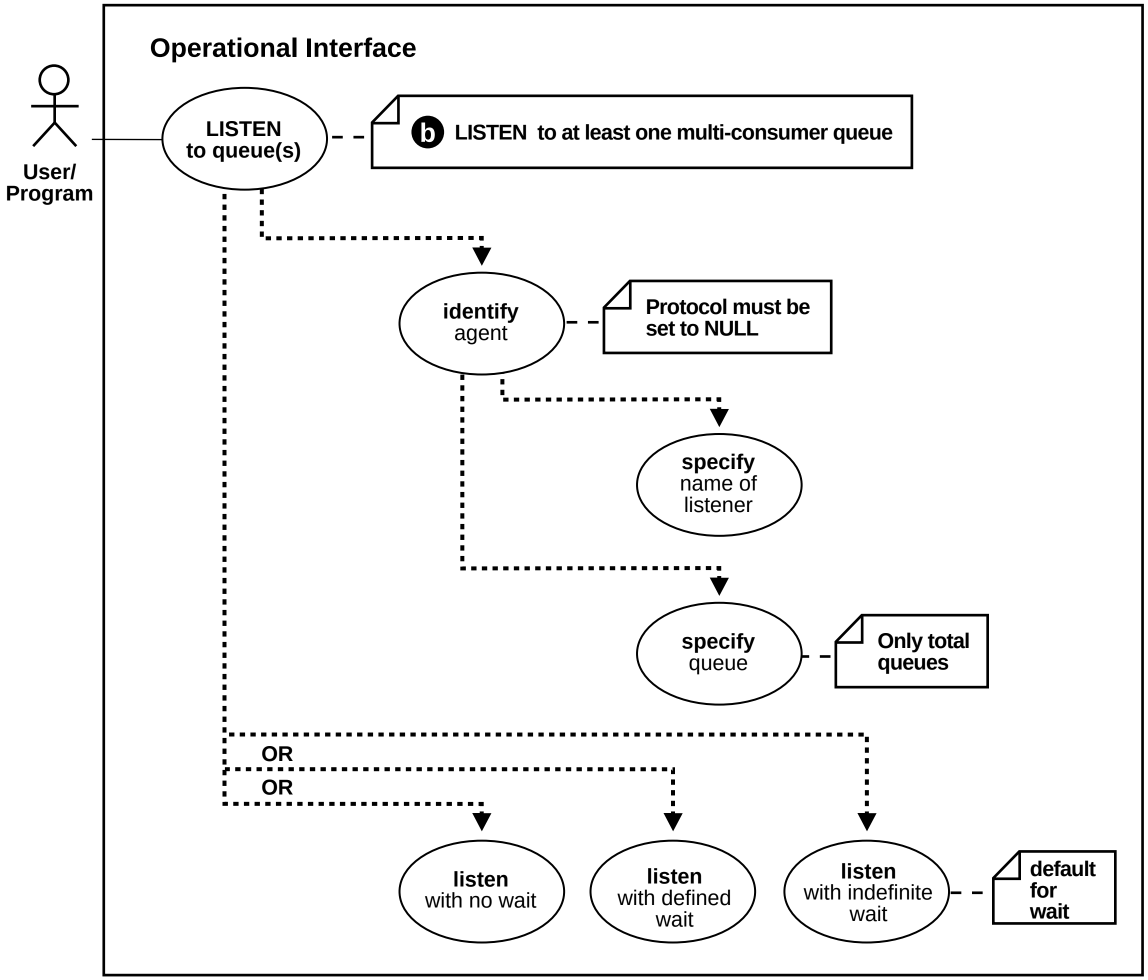
```
      appname ? (dvoid *)appname : (dvoid *)"",
      appname ? strlen((const char *)appname) : 0,
         OCI_ATTR_AGENT_NAME, errhp);

   OCIAttrSet(agent, OCI_DTYPE_AQAGENT,
      queue ? (dvoid *)queue : (dvoid *)"",
      queue ? strlen((const char *)queue) : 0,
         OCI_ATTR_AGENT_ADDRESS, errhp);

   printf("Set agent name to %s\n", appname ? (char *)appname : "NULL");
   printf("Set agent address to %s\n", queue ? (char *)queue : "NULL");
}

/* get agent from descriptor */
void GetAgent(agent, errhp)
OCIAQAgent *agent;
OCIError   *errhp;
{
text      *appname;
text      *queue;
ub4       appsz;
ub4       queuesz;

   if (!agent )
   {
     printf("agent was NULL \n");
     return;
   }
   checkerr(errhp, OCIAttrGet(agent, OCI_DTYPE_AQAGENT,
      (dvoid *)&appname, &appsz, OCI_ATTR_AGENT_NAME, errhp));
   checkerr(errhp, OCIAttrGet(agent, OCI_DTYPE_AQAGENT,
      (dvoid *)&queue, &queuesz, OCI_ATTR_AGENT_ADDRESS, errhp));
   if (!appsz)
      printf("agent name: NULL\n");
   else printf("agent name: %.*s\n", appsz, (char *)appname);
   if (!queuesz)
      printf("agent address: NULL\n");
   else printf("agent address: %.*s\n", queuesz, (char *)queue);
}

int main()
{
   OCIEnv *envhp;
   OCIServer *srvhp;
   OCIError *errhp;
```

```
 OCISvcCtx *svchp;
 OCISession *usrhp;
 OCIAQAgent *agent_list[3];
 OCIAQAgent *agent = (OCIAQAgent *)0;
 /* added next 2 121598 */
 int i;

/* Standard OCI Initialization */

 OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,  (dvoid * (*)()) 0,
    (dvoid * (*)()) 0,  (void (*)()) 0 );

 OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp,
           (ub4) OCI_HTYPE_ENV, 0, (dvoid **) 0);

 OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 0, (dvoid **) 0);

 OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
    0, (dvoid **) 0);

 OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
    0, (dvoid **) 0);

 OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

 OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
    0, (dvoid **) 0);

 /* set attribute server context in the service context */
 OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
    (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

 /* allocate a user context handle */
 OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp, (ub4) OCI_HTYPE_SESSION,
     (size_t) 0, (dvoid **) 0);

 /* allocate a user context handle */
 OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp, (ub4) OCI_HTYPE_SESSION,
    (size_t) 0, (dvoid **) 0);

 OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION,
    (dvoid *)"scott", (ub4)strlen("scott"), OCI_ATTR_USERNAME, errhp);

 OCIAttrSet((dvoid *) usrhp, (ub4) OCI_HTYPE_SESSION,
     (dvoid *) "tiger", (ub4) strlen("tiger"),
```

```
        (ub4) OCI_ATTR_PASSWORD, errhp);

    OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS, OCI_DEFAULT);

    OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX,
       (dvoid *)usrhp, (ub4)0, OCI_ATTR_SESSION, errhp);

    /* AQ LISTEN Initialization - allocate agent handles */
    for (i = 0; i < 3; i++)
    {
       agent_list[i] = (OCIAQAgent *)0;
       OCIDescriptorAlloc(envhp, (dvoid **)&agent_list[i],
           OCI_DTYPE_AQAGENT, 0, (dvoid **)0);
    }

    /*
     *   SCQ1, SCQ2, SCQ3 are single consumer queues in SCOTT’s schema
     */

    SetAgent(agent_list[0], (text *)0, "SCOTT.SCQ1", errhp);
    SetAgent(agent_list[1], (text *)0, "SCOTT.SCQ2", errhp);
    SetAgent(agent_list[2], (text *)0, "SCOTT.SCQ3", errhp);

    checkerr(errhp,OCIAQListen(svchp, errhp, agent_list, 3, 120, &agent, 0));

    printf("MESSAGE for :- \n");
    GetAgent(agent, errhp);
    printf("\n");

  }
```

Listen to One (Many) Multi-Consumer Queue(s)

Figure 6–9 Use Case Diagram: Listen to One(Many) Multi-Consumer Queue(s)



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Operational Interface — Basic Operations" on page 6-2

Example: Listen to Queue(s) Using PL/SQL (DBMS_AQ Package)

```
/* The listen call allows you to monitor a list of queues for messages for
   specific agents. You need to have dequeue privileges for all the queues
   you wish to monitor. */
```

Listen to Multi-Consumer Queue (Timeout of Zero).

```
DECLARE
   Agent_w_msg      aq$_agent;
   My_agent_list    dbms_aq.agent_list_t;

BEGIN
   /* NOTE:  MCQ1, MCQ2, MCQ3 are multi-consumer queues  in SCOTT’s schema
   *        SCQ1, SCQ2, SCQ3 are single consumer queues in SCOTT’s schema
   */
    Qlist(1):= aq$_agent('agent1', 'MCQ1',  NULL);
    Qlist(2):= aq$_agent('agent2', 'scott.MCQ2', NULL);
    Qlist(3):= aq$_agent('agent3', 'scott.MCQ3', NULL);

   /* Listen with a time-out of zero: */
   DBMS_AQ.LISTEN(
      agent_list   =>    My_agent_list,
      wait         =>    0,
      agent        =>    agent_w_msg);
   DBMS_OUTPUT.PUT_LINE('Message in Queue :- ' ||  agent_w_msg.address);
   DBMS_OUTPUT.PUT_LINE('');
END;
   /
```

Listen to Mixture of Multi-Consumer Queues (Timeout 100 Seconds).

```
DECLARE
   Agent_w_msg      aq$_agent;
   My_agent_list    dbms_aq.agent_list_t;

BEGIN
```

```
/* NOTE:  MCQ1, MCQ2, MCQ3 are multi-consumer queues  in SCOTT’s schema
*        SCQ1, SCQ2, SCQ3 are single consumer queues in SCOTT’s schema
*/
Qlist(1):= aq$_agent('agent1', 'MCQ1',  NULL);
Qlist(2):= aq$_agent(NULL, 'scott.SQ1', NULL);
Qlist(3):= aq$_agent('agent3', 'scott.MCQ3', NULL);
/* Listen with a time-out of 100 seconds */
DBMS_AQ.LISTEN(
   Agent_list   =>   My_agent_list,
   Wait         =>   100,
   Agent        =>    agent_w_msg);
   DBMS_OUTPUT.PUT_LINE('Message in Queue :- ' ||  agent_w_msg.address
                        || 'for agent' || agent_w_msg.name);
   DBMS_OUTPUT.PUT_LINE('');
END;
/
```

Example: Listen to Multi-Consumer Queue(s) Using C (OCI)

Listening to Multi-consumer Queues with a Zero Timeout, a Timeout of 120 Seconds, and a Timeout of 100 Seconds

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

static void checkerr(errhp, status)
OCIError *errhp;
sword status;
{
    text errbuf[512];
    ub4 buflen;
    sb4 errcode;

    switch (status)
    {
   case OCI_SUCCESS:
       break;
   case OCI_SUCCESS_WITH_INFO:
       printf("Error - OCI_SUCCESS_WITH_INFO\n");
       break;
   case OCI_NEED_DATA:
```

```
        printf("Error - OCI_NEED_DATA\n");
        break;
    case OCI_NO_DATA:
        printf("Error - OCI_NO_DATA\n");
        break;
    case OCI_ERROR:
        OCIErrorGet ((dvoid *) errhp, (ub4) 1, (text *) NULL, &errcode,
        errbuf, (ub4) sizeof(errbuf), (ub4) OCI_HTYPE_ERROR);
        printf("Error - %s\n", errbuf);
        break;
    case OCI_INVALID_HANDLE:
        printf("Error - OCI_INVALID_HANDLE\n");
        break;
    case OCI_STILL_EXECUTING:
        printf("Error - OCI_STILL_EXECUTE\n");
        break;
    case OCI_CONTINUE:
        printf("Error - OCI_CONTINUE\n");
        break;
    default:
    break;
     }
}

void SetAgent(OCIAQAgent *agent,
         text       *appname,
         text       *queue,
         OCIError   *errhp,
         OCIEnv     *envhp);

void GetAgent(OCIAQAgent *agent,
         OCIError   *errhp);

/*---------------------------------------------------------------*/
/* OCI Listen examples for multi-consumers                       */
/*                                                               */
void SetAgent(agent, appname, queue, errhp)
OCIAQAgent    *agent;
text          *appname;
text          *queue;
OCIError      *errhp;
{
  OCIAttrSet(agent,
        OCI_DTYPE_AQAGENT,
        appname ? (dvoid *)appname : (dvoid *)"",
```

```
        appname ? strlen((const char *)appname) : 0,
             OCI_ATTR_AGENT_NAME,
        errhp);

  OCIAttrSet(agent,
        OCI_DTYPE_AQAGENT,
        queue ? (dvoid *)queue : (dvoid *)"",
        queue ? strlen((const char *)queue) : 0,
             OCI_ATTR_AGENT_ADDRESS,
        errhp);

  printf("Set agent name to %s\n", appname ? (char *)appname : "NULL");
  printf("Set agent address to %s\n", queue ? (char *)queue : "NULL");
}

/* get agent from descriptor */
void GetAgent(agent, errhp)
OCIAQAgent *agent;
OCIError   *errhp;
{
   text      *appname;
   text      *queue;
   ub4       appsz;
   ub4       queuesz;

   if (!agent )
  {
     printf("agent was NULL \n");
     return;
  }
  checkerr(errhp, OCIAttrGet(agent, OCI_DTYPE_AQAGENT,
     (dvoid *)&appname, &appsz, OCI_ATTR_AGENT_NAME, errhp));
  checkerr(errhp, OCIAttrGet(agent, OCI_DTYPE_AQAGENT,
     (dvoid *)&queue, &queuesz, OCI_ATTR_AGENT_ADDRESS, errhp));
  if (!appsz)
     printf("agent name: NULL\n");
  else printf("agent name: %.*s\n", appsz, (char *)appname);
  if (!queuesz)
     printf("agent address: NULL\n");
  else printf("agent address: %.*s\n", queuesz, (char *)queue);
}

/* main from AQ Listen to Multi-Consumer Queue(s) */

/*  int main() */
```

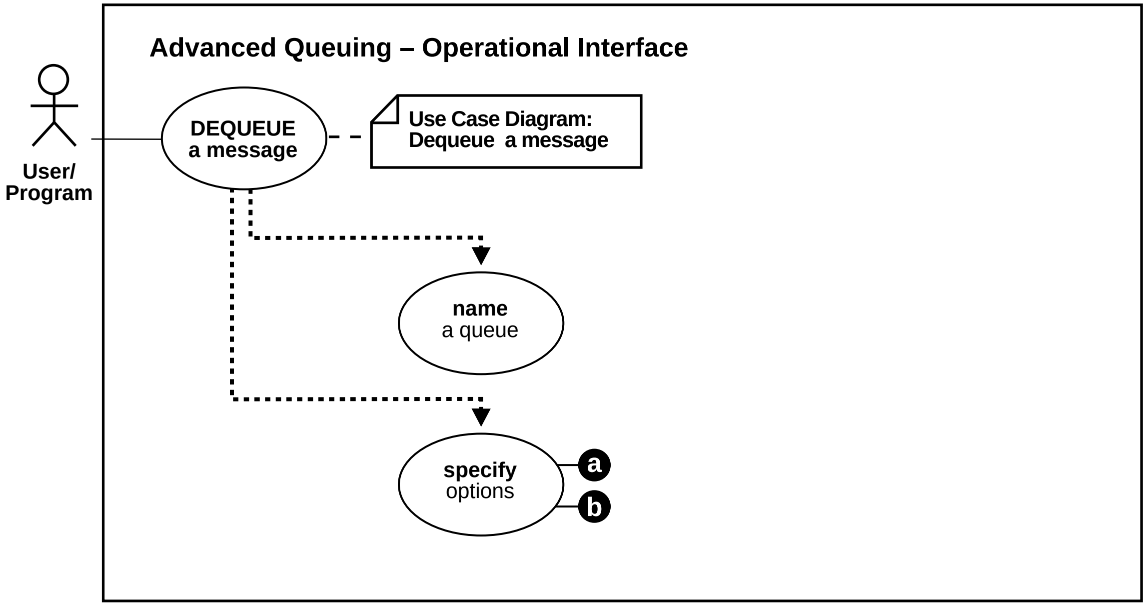
```
int main(char *argv, int argc)
{
    OCIEnv     *envhp;
    OCIServer  *srvhp;
    OCIError   *errhp;
    OCISvcCtx  *svchp;
    OCISession *usrhp;
    OCIAQAgent *agent_list[3];
    OCIAQAgent *agent;
    int         i;

 /* Standard OCI Initialization */

  OCIInitialize((ub4) OCI_OBJECT,
      (dvoid *)0,
      (dvoid * (*)()) 0,
      (dvoid * (*)()) 0,
      (void (*)()) 0 );

  OCIHandleAlloc( (dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
     0, (dvoid **) 0);

  OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 0, (dvoid **)0);

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
     0, (dvoid **) 0);

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
     0, (dvoid **) 0);

  OCIServerAttach( srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

  OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
     0, (dvoid **) 0);

  /* set attribute server context in the service context */
  OCIAttrSet( (dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
     (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

   /* allocate a user context handle */
  OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp, (ub4) OCI_HTYPE_SESSION,
     (size_t) 0, (dvoid **) 0);

   /* allocate a user context handle */
  OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp, (ub4) OCI_HTYPE_SESSION,
```

```
   (size_t) 0, (dvoid **) 0);


OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION,
   (dvoid *)"scott", (ub4)strlen("scott"), OCI_ATTR_USERNAME, errhp);


 OCIAttrSet((dvoid *) usrhp, (ub4) OCI_HTYPE_SESSION,
    (dvoid *) "tiger", (ub4) strlen("tiger"),
    (ub4) OCI_ATTR_PASSWORD, errhp);

OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS, OCI_DEFAULT);

OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX,
   (dvoid *)usrhp, (ub4)0, OCI_ATTR_SESSION, errhp);

/* AQ LISTEN Initialization - allocate agent handles */
for (i = 0; i < 3; i++)
{
   OCIDescriptorAlloc(envhp, (dvoid **)&agent_list[i],
       OCI_DTYPE_AQAGENT, 0, (dvoid **)0);
}

/*
 *  MCQ1, MCQ2, MCQ3 are multi-consumer queues in SCOTT’s schema
 */
/* Listening to Multi-consumer Queues with Zero Timeout */

SetAgent(agent_list[0], "app1", "MCQ1", errhp);
SetAgent(agent_list[1], "app2", "MCQ2", errhp);
SetAgent(agent_list[2], "app3", "MCQ3", errhp);

checkerr(errhp, OCIAQListen(svchp, errhp, agent_list, 3, 0, &agent, 0));

printf("MESSAGE for :- \n");
GetAgent(agent, errhp);
printf("\n");


/* Listening to Multi-consumer Queues with Timeout of 120 Seconds */

SetAgent(agent_list[0], "app1", "SCOTT.MCQ1", errhp);
SetAgent(agent_list[1], "app2", "SCOTT.MCQ2", errhp);
SetAgent(agent_list[2], "app3", "SCOTT.MCQ3", errhp);
```

```
  checkerr(errhp, OCIAQListen(svchp, errhp, agent_list, 3, 120, &agent, 0));

  printf("MESSAGE for :- \n");
  GetAgent(agent, errhp);
  printf("\n");


  /* Listening to a Mixture of Single and Multi-consumer Queues
   * with a Timeout of 100 Seconds
   */

  SetAgent(agent_list[0], "app1", "SCOTT.MCQ1", errhp);
  SetAgent(agent_list[1], "app2", "SCOTT.MCQ2", errhp);
  SetAgent(agent_list[2], (text *)0, "SCOTT.SCQ3", errhp);

  checkerr(errhp, OCIAQListen(svchp, errhp, agent_list, 3, 100, &agent, 0));

  printf("MESSAGE for :- \n");
  GetAgent(agent, errhp);
  printf("\n");

}
```

Dequeue a Message

Figure 6–10 Use Case Diagram: Dequeue a Message



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Operational Interface — Basic Operations" on page 6-2

Purpose:

Dequeues a message from the specified queue.

Syntax:

```
DBMS_AQ.DEQUEUE (
    queue_name            IN      VARCHAR2,
    dequeue_options       IN      dequeue_options_t,
    message_properties    OUT     message_properties_t,
    payload               OUT    "<type_name>",
    msgid                 OUT     raw);
```

Usage:

Table 6–6 DBMS_AQ.DEQUEUE

Parameter	Description
`queue_name` `(IN VARCHAR2)`	Specifies the name of the queue.
`dequeue_options` `(IN dequeue_option_t)`	For the definition please refer to the section titled "`DEQUEUE` Options."
`message_properties` `(OUT message_ properties_t)`	For the definition please refer to the section titled "Message Properties."
`payload` `(OUT "<type_name>")`	Not interpreted by Oracle AQ. The payload must be specified according to the specification in the associated queue table. For the definition of <type_name> please refer to section titled "Type name".
`msgid` `(OUT RAW)`	The system generated identification of the message.

Usage Notes

Search criteria and dequeue order for messages:

- The search criteria for messages to be dequeued is determined by the *consumer_ name*, *msgid* and *correlation* parameters in the dequeue_options. Msgid uniquely identifies the message to be dequeued. Correlation identifiers are application-defined identifiers that are not interpreted by AQ.
- Only messages in the `READY` state are dequeued unless a msgid is specified.
- The dequeue order is determined by the values specified at the time the queue table is created unless overridden by the msgid and correlation id in dequeue_ options.
- The database consistent read mechanism is applicable for queue operations. For example, a `BROWSE` call may not see a message that is enqueued after the beginning of the browsing transaction.

Navigating through a queue:

The default `NAVIGATION` parameter during dequeue is `NEXT_MESSAGE`. This means that subsequent dequeues will retrieve the messages from the queue based on the snapshot obtained in the first dequeue. In particular, a message that is enqueued after the first dequeue command will be processed only after processing all the remaining messages in the queue. This is usually sufficient when all the messages have already been enqueued into the queue, or when the queue does not have a priority-based ordering. However, applications must use the `FIRST_MESSAGE` navigation option when the first message in the queue needs to be processed by every dequeue command. This usually becomes necessary when a higher priority message arrives in the queue while messages already-enqueued are being processed.

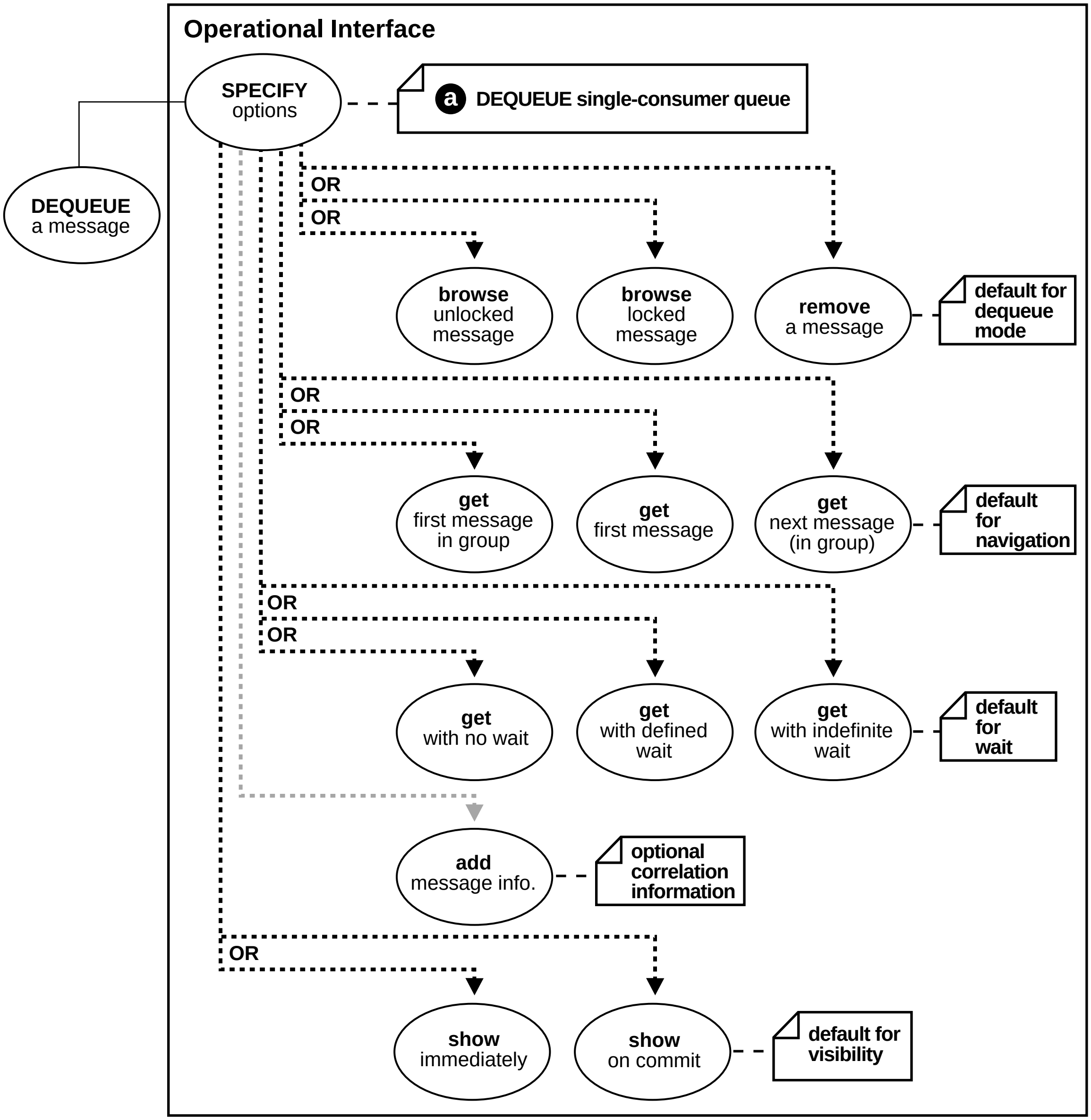
Note: It may also be more efficient to use the `FIRST_MESSAGE` navigation option when there are messages being concurrently enqueued. If the `FIRST_MESSAGE` option is not specified, AQ will have to continually generate the snapshot as of the first dequeue command, leading to poor performance. If the `FIRST_MESSAGE` option is specified, AQ will use a new snapshot for every dequeue command.

Dequeue by Message Grouping:

- Messages enqueued in the same transaction into a queue that has been enabled for message grouping will form a group. If only one message is enqueued in the transaction, this will effectively form a group of one message. There is no upper limit to the number of messages that can be grouped in a single transaction.
- In queues that have not been enabled for message grouping, a dequeue in `LOCKED` or `REMOVE` mode locks only a single message. By contrast, a dequeue operation that seeks to dequeue a message that is part of a group will lock the entire group. This is useful when all the messages in a group need to be processed as an atomic unit.
- When all the messages in a group have been dequeued, the dequeue returns an error indicating that all messages in the group have been processed. The application can then use the `NEXT_TRANSACTION` to start dequeuing messages from the next available group. In the event that no groups are available, the dequeue will time-out after the specified `WAIT` period.

Dequeue a Message from a Single-Consumer Queue [Specify Options]

Figure 6–11 Use Case Diagram: Dequeue a Message from a Single-Consumer Queue



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Operational Interface — Basic Operations" on page 6-2

Purpose:

To specify the options available for the dequeue operation.

Syntax:

```
TYPE dequeue_options_t IS RECORD (
   consumer_name    VARCHAR2(30) default NULL,
   dequeue_mode     BINARY_INTEGER default REMOVE,
   navigation       BINARY_INTEGER default NEXT_MESSAGE,
   visibility       BINARY_INTEGER default ON_COMMIT,
   wait             BINARY_INTEGER default FOREVER,
   msgid            RAW(16) default NULL,
   correlation      VARCHAR2(128) default NULL);
```

Usage:

Table 6–7 DEQUEUE options for a Singe-Consumer Queue

Parameter	Description
`consumer_name`	Name of the consumer. Only those messages matching the consumer name are accessed. If a queue is not set up for multiple consumers, this field should be set to `NULL`.
`dequeue_mode`	Specifies the locking behavior associated with the dequeue. `BROWSE`: Read the message without acquiring any lock on the message. This is equivalent to a select statement. `LOCKED`: Read and obtain a write lock on the message. The lock lasts for the duration of the transaction. This is equivalent to a select for update statement. `REMOVE`: Read the message and update or delete it. This is the default. The message can be retained in the queue table based on the retention properties. `REMOVE_NODATA`: Mark the message as updated or deleted. The message can be retained in the queue table based on the retention properties.
`navigation`	Specifies the position of the message that will be retrieved. First, the position is determined. Second, the search criterion is applied. Finally, the message is retrieved. `NEXT_MESSAGE`: Retrieve the next message which is available and matches the search criteria. If the previous message belongs to a message group, AQ will retrieve the next available message which matches the search criteria and belongs to the message group. This is the default. `NEXT_TRANSACTION`: Skip the remainder of the current transaction group (if any) and retrieve the first message of the next transaction group. This option can only be used if message grouping is enabled for the current queue. `FIRST_MESSAGE`: Retrieves the first message which is available and matches the search criteria. This will reset the position to the beginning of the queue.
`visibility`	Specifies whether the new message is dequeued as part of the current transaction.The visibility parameter is ignored when using the `BROWSE` mode. `ON_COMMIT`: The dequeue will be part of the current transaction. This is the default case. `IMMEDIATE`: The dequeued message is not part of the current transaction. It constitutes a transaction on its own.

Table 6–7 DEQUEUE options for a Singe-Consumer Queue

Parameter	Description
`wait`	Specifies the wait time if there is currently no message available which matches the search criteria. `FOREVER`: wait forever. This is the default. `NO_WAIT`: do not wait number: wait time in seconds
`msgid`	Specifies the message identifier of the message to be dequeued.
`correlation`	Specifies the correlation identifier of the message to be dequeued. Special pattern matching characters, such as the percent sign (%) and the underscore (_) can be used. If more than one message satisfies the pattern, the order of dequeuing is undetermined.

Usage Notes

Typically, you expect the consumer of messages to access messages using the dequeue interface. You can view processed messages or messages still to be processed by browsing by message id or by using `SELECT`s.

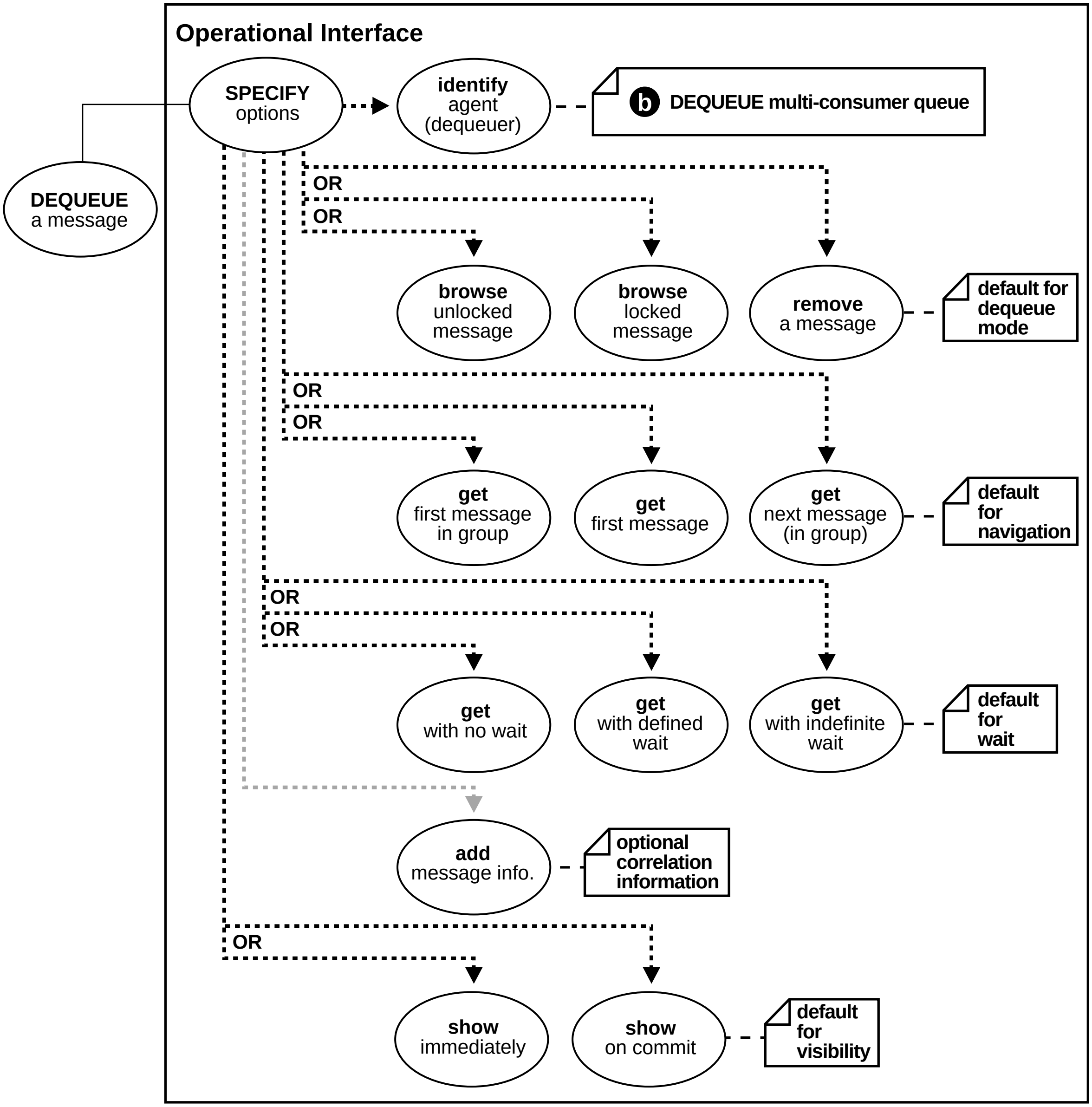
Example: Dequeue of Object Type Messages using PL/SQL (DBMS_AQ Package)

```
/* Dequeue from msg_queue: */
DECLARE
dequeue_options     dbms_aq.dequeue_options_t;
message_properties  dbms_aq.message_properties_t;
message_handle      RAW(16);
message             aq.message_typ;

BEGIN
   DBMS_AQ.DEQUEUE(
      queue_name          =>     'msg_queue',
      dequeue_options      =>    dequeue_options,
      message_properties  =>     message_properties,
      payload             =>     message,
      msgid               =>     message_handle);

   DBMS_OUTPUT.PUT_LINE ('Message: ' || message.subject ||
```

```
                                             ' ... ' || message.text );
   COMMIT;
END;
```

Dequeue a Message from a Multi-Consumer Queue [Specify Options]

Figure 6–12 Use Case Diagram: Dequeue a Message from a Multi-Consumer Queue



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Operational Interface — Basic Operations" on page 6-2

Purpose:

To specify the options available for the dequeue operation.

Syntax:

```
TYPE dequeue_options_t IS RECORD (
   consumer_name    VARCHAR2(30) default NULL,
   dequeue_mode     BINARY_INTEGER default REMOVE,
   navigation       BINARY_INTEGER default NEXT_MESSAGE,
   visibility       BINARY_INTEGER default ON_COMMIT,
   wait             BINARY_INTEGER default FOREVER,
   msgid            RAW(16) default NULL,
   correlation      VARCHAR2(128) default NULL);
```

Usage:

Table 6–8 DEQUEUE options for a Multi-Consumer Queue

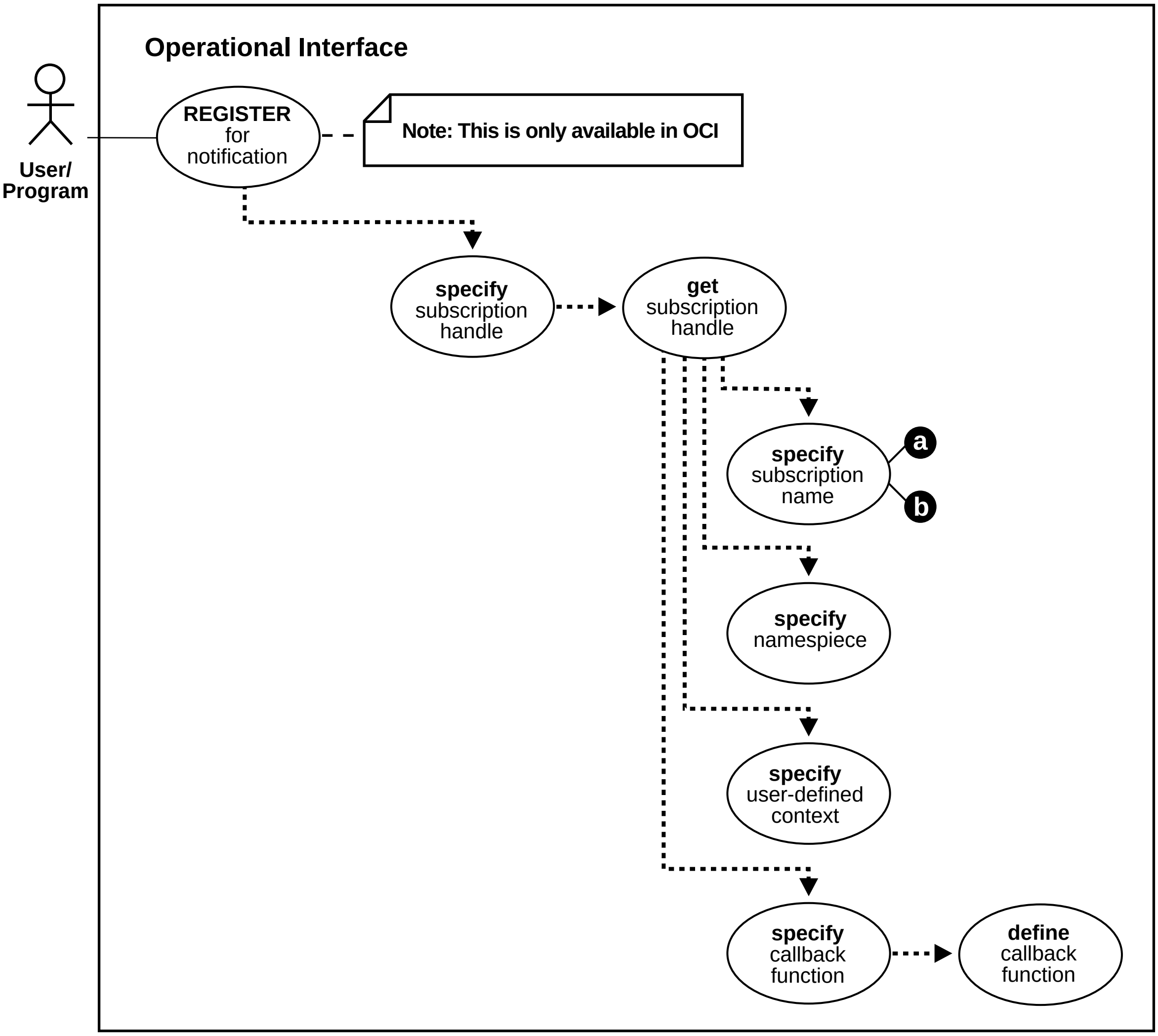
Parameter	Description
`consumer_name`	Name of the consumer. Only those messages matching the consumer name are accessed. If a queue is not set up for multiple consumers, this field should be set to `NULL`.
`dequeue_mode`	Specifies the locking behavior associated with the dequeue. `BROWSE`: Read the message without acquiring any lock on the message. This is equivalent to a select statement. `LOCKED`: Read and obtain a write lock on the message. The lock lasts for the duration of the transaction. This is equivalent to a select for update statement. `REMOVE`: Read the message and update or delete it. This is the default. The message can be retained in the queue table based on the retention properties. `REMOVE_NODATA`: Mark the message as updated or deleted. The message can be retained in the queue table based on the retention properties.
`navigation`	Specifies the position of the message that will be retrieved. First, the position is determined. Second, the search criterion is applied. Finally, the message is retrieved. `NEXT_MESSAGE`: Retrieve the next message which is available and matches the search criteria. If the previous message belongs to a message group, AQ will retrieve the next available message which matches the search criteria and belongs to the message group. This is the default. `NEXT_TRANSACTION`: Skip the remainder of the current transaction group (if any) and retrieve the first message of the next transaction group. This option can only be used if message grouping is enabled for the current queue. `FIRST_MESSAGE`: Retrieves the first message which is available and matches the search criteria. This will reset the position to the beginning of the queue.
`visibility`	Specifies whether the new message is dequeued as part of the current transaction.The visibility parameter is ignored when using the `BROWSE` mode. `ON_COMMIT`: The dequeue will be part of the current transaction. This is the default case. `IMMEDIATE`: The dequeued message is not part of the current transaction. It constitutes a transaction on its own.

Table 6–8 DEQUEUE options for a Multi-Consumer Queue

Parameter	Description
`wait`	Specifies the wait time if there is currently no message available which matches the search criteria. `FOREVER`: wait forever. This is the default. `NO_WAIT`: do not wait number: wait time in seconds
`msgid`	Specifies the message identifier of the message to be dequeued.
`correlation`	Specifies the correlation identifier of the message to be dequeued. Special pattern matching characters, such as the percent sign (%) and the underscore (_) can be used. If more than one message satisfies the pattern, the order of dequeuing is undetermined.

Register for Notification

Figure 6–13 Use Case Diagram: Register for Notification



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Operational Interface — Basic Operations" on page 6-2

Purpose:

To register a callback for message notification.

Syntax:

```
ub4 OCISubscriptionRegister (
   OCISvcCtx           *svchp,
   OCISubscription     **subscrhpp,
   ub2                 count,
   OCIError            *errhp,
   ub4                 mode);
```

Usage:

Table 6–9 DEQUEUE options for a Multi-Consumer Queue

Parameter	Description
`svchp (IN)`	A V8 OCI service context. This service context should have a valid authenticated user handle.
`subscrhpp (IN)`	An array of subscription handles. Each element of this array should be a subscription handle with the `OCI_ATTR_SUBSCR_NAME`, `OCI_ATTR_SUBSCR_NAMESPACE`, `OCI_ATTR_SUBSCR_CBACK`, and `OCI_ATTR_SUBSCR_CTX` attributes set; otherwise, an error will be returned. For information, see Subscription Handle Attributes. When a notification is received for the registration denoted by the subscrhpp[i], the user defined callback function (`OCI_ATTR_SUBSCR_CBACK`) set for subscrhpp[i] will get invoked with the context (`OCI_ATTR_SUBSCR_CTX`) set for subscrhpp[i].

Table 6–9 DEQUEUE options for a Multi-Consumer Queue

Parameter	Description
`count (IN)`	The number of elements in the subscription handle array
`errhp (OUT)`	An error handle you can pass to `OCIErrorGet`() for diagnostic information in the event of an error.
`mode (IN)`	Call-specific mode. Valid values: ■ `OCI_DEFAULT` - executes the default call which specifies that the registration is treated as disconnected ■ `OCI_NOTIFY_CONNECTED` - notifications are received only if the client is connected (not supported in this release) Whenever a new client process comes up, or an old one goes down and comes back up, it needs to register for all subscriptions of interest. If the client stays up and the server first goes down and then comes back up, the client will continue to receive notifications for registrations that are `DISCONNECTED`. However, the client will not receive notifications for `CONNECTED` registrations as they will be lost once the server goes down and comes back up.

Usage Notes

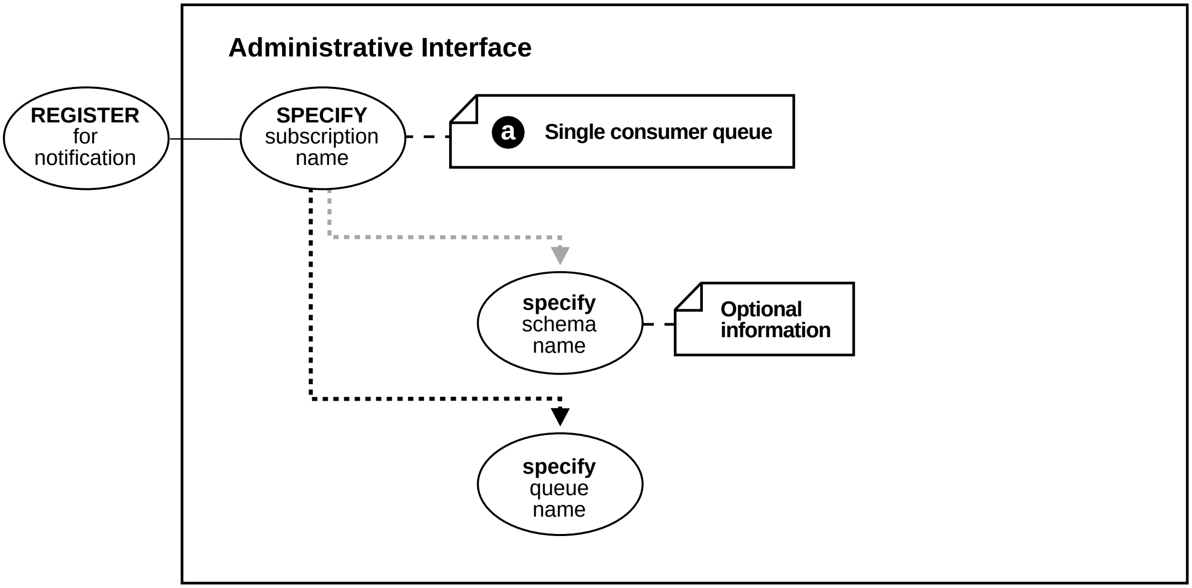
- This call is invoked for registration to a subscription which identifies the subscription name of interest and the associated callback to be invoked. Interest in several subscriptions can be registered at one time.
- This interface is only valid for the asynchronous mode of message delivery. In this mode, a subscriber issues a registration call which specifies a callback. When messages are received that match the subscription criteria, the callback is invoked. The callback may then issue an explicit `message_receive` (dequeue) to retrieve the message.
- The user must specify a subscription handle at registration time with the namespace attribute set to `OCI_SUBSCR_NAMESPACE_AQ`.
- The subscription name is the string '`schema.queue`' if the registration is for a single consumer queue and '`schema.queue:consumer_name`' if the registration is for a multiconsumer queues.
- Related Functions: `OCIAQListen`(), `OCISubscriptionDisable`(), `OCISubscriptionEnable`(), `OCISubscriptionUnRegister`()

For more information about the OCI operation `Register for Notification` see:

- *Oracle Call Interface Programmer's Guide*

Register for Notification [Specify Subscription Name — Single-Consumer Queue]

Figure 6–14 Use Case Diagram: Specify Subscription Name - Single Consumer Queue

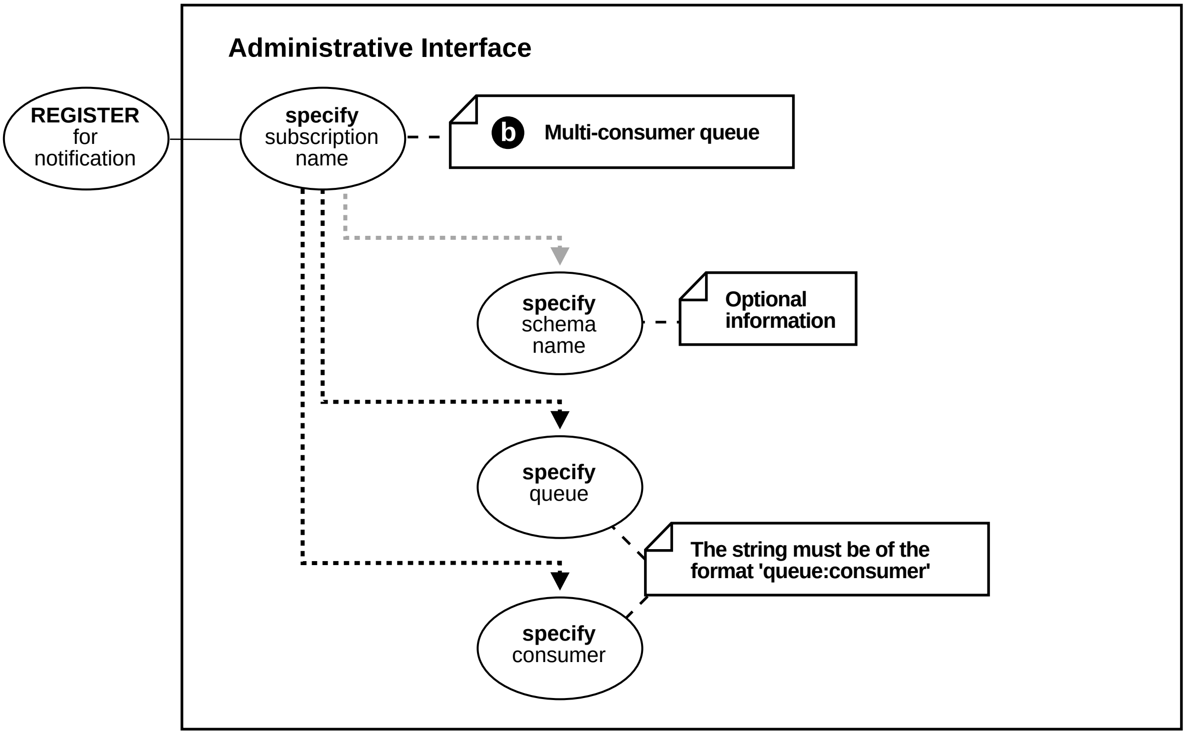


To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Operational Interface — Basic Operations" on page 6-2

Register for Notification [Specify Subscription Name — Multi-Consumer Queue]

Figure 6–15 Use Case Diagram: Specify Subscription Name - Multi-Consumer Queue



To refer to the table of all basic operations having to do with the Operational Interface see:

- "Use Case Model: Operational Interface — Basic Operations" on page 6-2

Example: Register for Notifications For Single-Consumer and Multi-Consumer Queries Using C (OCI)

```
/* OCIRegister can be used by the client to register to receive notifications
   when messages are enqueued into non-persistent and normal queues. */
    #include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <oci.h>


static OCIEnv     *envhp;
static OCIServer  *srvhp;
static OCIError   *errhp;
static OCISvcCtx  *svchp;


/* The callback that gets invoked on notification */
ub4 notifyCB(ctx, subscrhp, pay, payl, desc, mode)
dvoid *ctx;
OCISubscription *subscrhp;      /* subscription handle */
dvoid           *pay;           /* payload  */
ub4              payl;          /* payload length */
dvoid           *desc;          /* the AQ notification descriptor */
ub4              mode;
{
 text               *subname;
 ub4                 size;
 ub4                *number = (ub4 *)ctx;
 text               *queue;
 text               *consumer;
 OCIRaw             *msgid;
 OCIAQMsgProperties  *msgprop;

  (*number)++;

  /* Get the subscription name */
  OCIAttrGet((dvoid *)subscrhp, OCI_HTYPE_SUBSCRIPTION,
                             (dvoid *)&subname, &size,
                             OCI_ATTR_SUBSCR_NAME, errhp);
 printf("got notification number %d for %.*s  %d  \n",
         *number, size, subname, payl);

 /* Get the queue name from the AQ notify descriptor */
 OCIAttrGet(desc, OCI_DTYPE_AQNFY_DESCRIPTOR, (dvoid *)&queue, &size,
```

```
              OCI_ATTR_QUEUE_NAME, errhp);

 /* Get the consumer name for which this notification was received */
 OCIAttrGet(desc, OCI_DTYPE_AQNFY_DESCRIPTOR, (dvoid *)&consumer, &size,
       OCI_ATTR_CONSUMER_NAME, errhp);

 /* Get the message id of the message for which we were notified */
 OCIAttrGet(desc, OCI_DTYPE_AQNFY_DESCRIPTOR, (dvoid *)&msgid, &size,
       OCI_ATTR_NFY_MSGID, errhp);

 /* Get the message properties of the message for which we were notified */
 OCIAttrGet(desc, OCI_DTYPE_AQNFY_DESCRIPTOR, (dvoid *)&msgprop, &size,
       OCI_ATTR_MSG_PROP, errhp);

}


int main(argc, argv)
int argc;
char *argv[];
{
  OCISession *authp = (OCISession *) 0;

  /* The subscription handles */
  OCISubscription *subscrhp[5];

  /* Registrations are for AQ namespace */
  ub4 namespace = OCI_SUBSCR_NAMESPACE_AQ;

  /* The context fot the callback */
  ub4 ctx[5] = {0,0,0,0,0};

  printf("Initializing OCI Process\n");

  /* The OCI Process Environment must be initialized  with OCI_EVENTS */
  /* OCI_OBJECT flag is set to enable us dequeue */
  (void) OCIInitialize((ub4) OCI_EVENTS|OCI_OBJECT, (dvoid *)0,
                      (dvoid * (*)(dvoid *, size_t)) 0,
                      (dvoid * (*)(dvoid *, dvoid *, size_t))0,
                      (void (*)(dvoid *, dvoid *)) 0 );

  printf("Initialization successful\n");

  /* The standard OCI setup */
  printf("Initializing OCI Env\n");
```

```
(void) OCIEnvInit((OCIEnv **) &envhp, OCI_DEFAULT, (size_t) 0,
              (dvoid **) 0 );

(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &errhp, OCI_HTYPE_ERROR,
                 (size_t) 0, (dvoid **) 0);

/* Server contexts */
(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &srvhp, OCI_HTYPE_SERVER,
                 (size_t) 0, (dvoid **) 0);

(void) OCIHandleAlloc( (dvoid *) envhp, (dvoid **) &svchp, OCI_HTYPE_SVCCTX,
                 (size_t) 0, (dvoid **) 0);


printf("connecting to server\n");
(void) OCIServerAttach( srvhp, errhp, (text *)"", strlen(""), 0);
printf("connect successful\n");

/* Set attribute server context in the service context */
(void) OCIAttrSet( (dvoid *) svchp, OCI_HTYPE_SVCCTX, (dvoid *)srvhp,
        (ub4) 0, OCI_ATTR_SERVER, (OCIError *) errhp);

(void) OCIHandleAlloc((dvoid *) envhp, (dvoid **)&authp,
           (ub4) OCI_HTYPE_SESSION, (size_t) 0, (dvoid **) 0);

(void) OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
               (dvoid *) "scott", (ub4) strlen("scott"),
               (ub4) OCI_ATTR_USERNAME, errhp);

(void) OCIAttrSet((dvoid *) authp, (ub4) OCI_HTYPE_SESSION,
               (dvoid *) "tiger", (ub4) strlen("tiger"),
               (ub4) OCI_ATTR_PASSWORD, errhp);

checkerr(errhp, OCISessionBegin ( svchp,  errhp, authp, OCI_CRED_RDBMS,
         (ub4) OCI_DEFAULT));

(void) OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX,
                 (dvoid *) authp, (ub4) 0,
                 (ub4) OCI_ATTR_SESSION, errhp);

/* Setting the subscription handle for notification on
   a NORMAL single consumer queue */
printf("allocating subscription handle\n");
subscrhp[0] = (OCISubscription *)0;
(void) OCIHandleAlloc((dvoid *) envhp, (dvoid **)&subscrhp[0],
```

```
        (ub4) OCI_HTYPE_SUBSCRIPTION,
        (size_t) 0, (dvoid **) 0);

 printf("setting subscription name\n");
 (void) OCIAttrSet((dvoid *) subscrhp[0], (ub4) OCI_HTYPE_SUBSCRIPTION,
               (dvoid *) "SCOTT.SCQ1", (ub4) strlen("SCOTT.SCQ1"),
               (ub4) OCI_ATTR_SUBSCR_NAME, errhp);

 printf("setting subscription callback\n");
 (void) OCIAttrSet((dvoid *) subscrhp[0], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *) notifyCB, (ub4) 0,
                (ub4) OCI_ATTR_SUBSCR_CALLBACK, errhp);

printf("setting subscription context \n");
(void) OCIAttrSet((dvoid *) subscrhp[0], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *)&ctx[0], (ub4)sizeof(ctx[0]),
                (ub4) OCI_ATTR_SUBSCR_CTX, errhp);

 printf("setting subscription namespace\n");
 (void) OCIAttrSet((dvoid *) subscrhp[0], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *) &namespace, (ub4) 0,
                (ub4) OCI_ATTR_SUBSCR_NAMESPACE, errhp);

/* Setting the subscription handle for notification on a NORMAL multi-consumer
   consumer queue */
 subscrhp[1] = (OCISubscription *)0;
 (void) OCIHandleAlloc((dvoid *) envhp, (dvoid **)&subscrhp[1],
        (ub4) OCI_HTYPE_SUBSCRIPTION,
        (size_t) 0, (dvoid **) 0);

 (void) OCIAttrSet((dvoid *) subscrhp[1], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *) "SCOTT.MCQ1:APP1",
                (ub4) strlen("SCOTT.MCQ1:APP1"),
                (ub4) OCI_ATTR_SUBSCR_NAME, errhp);

 (void) OCIAttrSet((dvoid *) subscrhp[1], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *) notifyCB, (ub4) 0,
                (ub4) OCI_ATTR_SUBSCR_CALLBACK, errhp);

 (void) OCIAttrSet((dvoid *) subscrhp[1], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *)&ctx[1], (ub4)sizeof(ctx[1]),
                (ub4) OCI_ATTR_SUBSCR_CTX, errhp);

 (void) OCIAttrSet((dvoid *) subscrhp[1], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *) &namespace, (ub4) 0,
```

```
                 (ub4) OCI_ATTR_SUBSCR_NAMESPACE, errhp);


/* Setting the subscription handle for notification on a non-persistent
  single-consumer queue */
 subscrhp[2] = (OCISubscription *)0;
 (void) OCIHandleAlloc((dvoid *) envhp, (dvoid **)&subscrhp[2],
        (ub4) OCI_HTYPE_SUBSCRIPTION,
        (size_t) 0, (dvoid **) 0);

 (void) OCIAttrSet((dvoid *) subscrhp[2], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *) "SCOTT.NP_SCQ1",
                (ub4) strlen("SCOTT.NP_SCQ1"),
                (ub4) OCI_ATTR_SUBSCR_NAME, errhp);

 (void) OCIAttrSet((dvoid *) subscrhp[2], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *) notifyCB, (ub4) 0,
                (ub4) OCI_ATTR_SUBSCR_CALLBACK, errhp);

 (void) OCIAttrSet((dvoid *) subscrhp[2], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *)&ctx[2], (ub4)sizeof(ctx[2]),
                (ub4) OCI_ATTR_SUBSCR_CTX, errhp);

 (void) OCIAttrSet((dvoid *) subscrhp[2], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *) &namespace, (ub4) 0,
                (ub4) OCI_ATTR_SUBSCR_NAMESPACE, errhp);


/* Setting the subscription handle for notification on
   a non-persistent multi consumer queue */
/* Waiting on user specified recipient */
 subscrhp[3] = (OCISubscription *)0;
 (void) OCIHandleAlloc((dvoid *) envhp, (dvoid **)&subscrhp[3],
        (ub4) OCI_HTYPE_SUBSCRIPTION,
        (size_t) 0, (dvoid **) 0);

 (void) OCIAttrSet((dvoid *) subscrhp[3], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *) "SCOTT.NP_MCQ1",
                (ub4) strlen("SCOTT.NP_MCQ1"),
                (ub4) OCI_ATTR_SUBSCR_NAME, errhp);

 (void) OCIAttrSet((dvoid *) subscrhp[3], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *) notifyCB, (ub4) 0,
                (ub4) OCI_ATTR_SUBSCR_CALLBACK, errhp);
```

```
 (void) OCIAttrSet((dvoid *) subscrhp[3], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *)&ctx[3], (ub4)sizeof(ctx[3]),
                (ub4) OCI_ATTR_SUBSCR_CTX, errhp);

  (void) OCIAttrSet((dvoid *) subscrhp[3], (ub4) OCI_HTYPE_SUBSCRIPTION,
                (dvoid *) &namespace, (ub4) 0,
                (ub4) OCI_ATTR_SUBSCR_NAMESPACE, errhp);


  printf("Registering for all the subscriptiosn \n");
  checkerr(errhp, OCISubscriptionRegister(svchp, subscrhp, 4, errhp,
                OCI_DEFAULT));

  printf("Waiting for notifcations \n");

  /* wait for minutes for notifications */
  sleep(300);

  printf("Exiting\n");
}
```

7

Advanced Queuing — Java API

This chapter introduces and details the Java Application Programmer's Interface for Advanced Queuing under the following headings:

- Introduction
- AQDriverManager
- APIs/Classes
- AQSession
- AQConstants
- AQAgent
- AQQueueTableProperty
- AQQueueProperty
- AQQueueTable
- AQQueueAdmin
- AQQueue
- AQEnqueueOption
- AQDequeueOption
- AQMessage
- AQMessageProperty
- AQRawPayload
- AQException
- AQOracleSQLException

Introduction

The Java AQ API supports both the administrative and operational features of Oracle AQ. In developing Java programs for messaging applications, you will use JDBC to open a connection to the database and then the Java AQ API for message queuing. This means that you will no longer be required to use the PL/SQL interfaces.

The following sections describe the common interfaces and classes based on the current PL/SQL interfaces. The common interfaces are prefixed with "**AQ**". These interfaces will have different implementations in Oracle8*i* and Oracle Lite. In this document we describe the common interfaces and their corresponding Oracle8*i* implementations, which are in turn prefixed with "**AQOracle**".

The java AQ classes are located in `$ORACLE_HOME/rdbms/jlib/aqapi.jar`. These classes can be used with any Oracle8*i* JDBC driver. If your application uses the OCI8 or thin JDBC driver, you must include `$ORACLE_HOME/rdbms/jlib/aqapi.jar` in the `CLASSPATH`. If the application is using the KPRB driver and accessing the java AQ API from java stored procedures, you must first load the `aqapi.jar` file into the database using the "loadjava" utility.

Chapter 8, "Oracle Advanced Queuing by Example" contains the following examples:

- Enqueue and Dequeue of RAW Type Messages Using Java
- Dequeue of Messages Using Java
- Dequeue of Messages Using Java
- Enqueue of Messages with Priority Using Java

Set up for the `test_aqjava` class is described in "Setup for AQ Examples" on page 7-10. The way to create a multi-consumer queue is described in the "AQSession" on page 7-8.

AQDriverManager

The various implementations of the Java AQ API are managed via an `AQDriverManager`. Both OLite and Oracle8*i* will have an `AQDriver` which is registered with the `AQDriverManager`. The driver manager is used to create an `AQSession` which can be used to perform messaging tasks.

When the `AQDriverManager.createAQSession`() method is invoked, it calls the appropriate `AQDriver` (amongst the registered drivers) depending on the parameter passed to the `createAQSession`() call.

The Oracle8*i* `AQDriver` expects a valid JDBC connection to be passed in as a parameter to create an AQSession. Users must have the execute privilege on the `DBMS_AQIN` package in order to use the AQ Java interfaces. Users can also acquire these rights through the `AQ_USER_ROLE` or the `AQ_ADMINSTRATOR_ROLE`.Users will also need the appropriate system and queue privileges for 8.1 style queue tables.

Note: Currently the Oracle8i AQDriver supports only `RAW` type payloads.

getDrivers

Purpose:

This method returns the list of drivers registered with the driver manager. It returns a Vector of strings containing the names of the registered drivers.

Syntax:

```
public static java.util.Vector getDrivers()
```

getAQSession

Purpose:

This method creates an AQSession.

Syntax:

```
public static AQSession getAQSession (java.lang.Object conn) throws AQException
```

Table 7–1 getAQSession Parameters

Parameter	Meaning
`conn`	if the user is using the `AQOracleDriver` , then the object passed in must be a valid JDBC connection

Multithreaded Program Support

Currently Java AQ objects are not thread safe. Therefore, methods on `AQSession`, `AQQueueTable`, `AQQueue` and other AQ objects should not be called concurrently from different threads. You can pass these objects between threads, but the program must ensure that the methods on these AQ objects are not invoked concurrently.

We recommend that multithreaded programs create a different `AQSession` in each thread (using the same or a different JDBC connection) and get new queue table and queue handles using the `getQueueTable` and `getQueue` methods in `AQSession`.

registerDriver

Purpose:

This method is used by various implementations of the AQ driver to register themselves with the driver manager (this method is not directly called by client programs)

Syntax:

```
public static void registerDriver(AQDriver aq_driver)
```

Note: To create an `AQSession`, you must first open a JDBC connection. Then you must load the `AQDriver` that you need to use in the application. Note that the driver needs to be loaded only once (before the first `createAQSession` call). Loading the driver multiple times will have no effect. For more information, see "Setup for AQ Examples" on page 7-10.

Example

```
Connection db_conn;       /* JDBC connection */
AQSession  aq_sess;       /* AQSession */

/* JDBC setup and connection creation: */
class.forName(“oracle.jdbc.driver.OracleDriver”);
db_conn = DriverManager.getConnection (
   "jdbc:oracle:oci8:@", "aquser", "aquser");
db_conn.setAutoCommit(false);

/* Load the Oracle8i AQ driver: */
class.forName(“oracle.AQ.AQOracleDriver”);
/* Create an AQ Session: */
aq_sess = AQDriverManager.createAQSession(db_conn);
```

APIs/Classes

Table 7–2 AQ Interfaces

Interface Summary	Description
AQSession	Open a session to the queuing system
AQQueueTable	AQ Queue Table interface
AQQueueAdmin	AQ Queue administrative interfaces
AQQueue	AQ Queue operational interfaces
AQMessage	AQ message
AQRaw Payload	AQ Raw Payload
AQDriver	Interface for various AQ drivers

Table 7–3 AQ Common Classes

Class Summary	Description
AQConstants	Constants used in AQ operations
AQAgent	AQ Agent
AQDriverManager	Driver Manager for various AQ drivers
AQEnqueueOption	AQ Enqueue Options
AQDequeueOption	AQ Dequeue options
AQMessageProperty	AQ Message properties
AQQueueProperty	AQ Queue properties
AQQueueTableProperty	AQ Queue Table properties

Table 7–4 Oracle8i AQ Classes

Class Summary	Description
AQOracleSession	Oracle server implementation of AQSession
AQOracleMessage	Oracle Server implementation of AQMessage
AQOracleDriver	Oracle server implementation of AQDriver
AQOracleQueue	Oracle server implementation of AQQueue
AQOracleQueueTable	Oracle server implementation of AQQueueTable
AQOracleRawPayload	Oracle server implementation of AQRawPayload

In general use only the interfaces and classes that are common to both implementations (as described in the first two tables). This will ensure that your applications are portable between Oracle 8*i* and Olite AQ implementations.

The **AQOracle** classes should not be used unless there is a method in these classes that is not available in the common interfaces.

Note that since the `AQQueue` interface extends `AQQueueAdmin`, all queue administrative and operation functionality is available via `AQQueue`.

AQSession

createQueueTable

Purpose:

This method creates a new queue table in a particular user’s schema according to the properties specified in the `AQQueueTableProperty` object passed in.

Syntax:

```
public AQQueueTable createQueueTable(java.lang.String owner,
                                     java.lang.String name,
                                     AQQueueTableProperty property)
                                     throws AQException
```

Table 7–5 createQueueTable Parameters

Parameter	**Meaning**
`owner`	schema (user) in which to create the queue table
`q_name`	name of the queue table
`property`	queue table properties

Returns:

`AQQueueTable` object

getQueueTable

Purpose:

This method is used to get a handle to an existing queue table.

Syntax:

```
public AQQueueTable getQueueTable(java.lang.String owner,
                                  java.lang.String name)
```

Table 7–6 getQueueTable Parameters

Parameter	Meaning
`owner`	schema (user) in which the queue table resides
`name`	name of the queue table

Returns:

`AQQueueTable` object

createQueue

Purpose:

This method creates a queue in a queue_table with the specified queue properties. It uses the same schema name that was used to create the queue table.

Syntax:

```
public AQQueue createQueue(AQQueueTable q_table,
                           java.lang.String q_name,
                           AQQueueProperty q_property) throws AQException
```

Table 7–7 createQueue Parameters

Parameter	Meaning
`q_table`	queue table in which to create queue
`name`	name of the queue to be created
`q_property`	queue properties

Returns:

`AQQueue` object

getQueue

Purpose:

This method can be used to get a handle to an existing queue.

Syntax:

```
public AQQueue getQueue(java.lang.String owner,
                        java.lang.String name)
```

Table 7–8   getQueue Parameters

Parameter	Meaning
`owner`	schema (user) in which the queue table resides
`name`	name of the queue

Returns:

`AQQueue` object

Usage Note

Currently the java AQ API supports only queues with raw payloads. If you try to access queue tables that contain queues with object payloads you will get an `AQException` with the message "payload type not supported."

Setup for AQ Examples

1. Create an AQ User

Here an 'aqjava' user is setup as follows:

```
CONNECT sys/change_on_install AS sysdba

DROP USER aqjava CASCADE;
GRANT CONNECT, RESOURCE, AQ_ADMINISTRATOR_ROLE TO aqjava IDENTIFIED BY aqjava;
GRANT EXECUTE ON SYS.DBMS_AQADM TO aqjava;
GRANT EXECUTE ON SYS.DBMS_AQ TO aqjava;
CONNECT aqjava/aqjava
```

2. Set up main class

Next we set up the main class from which we will call subsequent examples and handle exceptions.

```
import java.sql.*;
import oracle.AQ.*;

public class test_aqjava
{
   public static void main(String args[])
   {
      AQSession  aq_sess = null;

      try
      {
         aq_sess = createSession(args);

        /* now run the test: */
        runTest(aq_sess);
      }
      catch (Exception ex)
      {
         System.out.println("Exception-1: " + ex);
         ex.printStackTrace();
      }
   }
}
```

3. Create an AQ Session;

`Next,  an AQ Session is created for the 'aqjava' user as shown in the AQDriverManager section above:`

```
 public static AQSession createSession(String args[])
   {
      Connection db_conn;
      AQSession  aq_sess = null;

      try
      {

         Class.forName("oracle.jdbc.driver.OracleDriver");
         /* your actual hostname, port number, and SID will
         vary from what follows. Here we use 'dlsun736,' '5521,'
         and 'test,' respectively: */
```

```
            db_conn =
                     DriverManager.getConnection(
                     "jdbc:oracle:thin:@dlsun736:5521:test",
                     "aqjava", "aqjava");

            System.out.println("JDBC Connection opened ");
            db_conn.setAutoCommit(false);

            /* Load the Oracle8i AQ driver: */
            Class.forName("oracle.AQ.AQOracleDriver");

            /* Create an AQ Session: */
            aq_sess = AQDriverManager.createAQSession(db_conn);
            System.out.println("Successfully created AQSession ");
         }
         catch (Exception ex)
         {
            System.out.println("Exception: " + ex);
            ex.printStackTrace();
         }
         return aq_sess;
      }
```

Example

1. Create a queue table and a queue

Now, with the ’ runTest’ class, called from the above main class, we will create a queue table and queue for the ’aqjava’ user.

```
public static void runTest(AQSession aq_sess) throws AQException
{
    AQQueueTableProperty     qtable_prop;
    AQQueueProperty          queue_prop;
    AQQueueTable             q_table;
    AQQueue                  queue;

    /* Create a AQQueueTableProperty object (payload type - RAW): */
    qtable_prop = new AQQueueTableProperty("RAW");

    /* Create a queue table called aq_table1 in aqjava schema: */
    q_table = aq_sess.createQueueTable ("aqjava", "aq_table1", qtable_prop);
    System.out.println("Successfully created aq_table1 in aqjava schema");

    /* Create a new AQQueueProperty object: */
```

```
    queue_prop = new AQQueueProperty();

    /* Create a queue called aq_queue1 in aq_table1: */
    queue = aq_sess.createQueue (q_table, "aq_queue1", queue_prop);
    System.out.println("Successfully created aq_queue1 in aq_table1");
}
```

2. Get a handle to an existing queue table and queue

```
public static void runTest(AQSession aq_sess) throws AQException
{
    AQQueueTable            q_table;
    AQQueue                 queue;

    /* Get a handle to queue table - aq_table1 in aqjava schema: */
    q_table = aq_sess.getQueueTable ("aqjava", "aq_table1");
    System.out.println("Successful getQueueTable");

    /* Get a handle to a queue - aq_queue1 in aqjava schema: */
    queue = aq_sess.getQueue ("aqjava", "aq_queue1");
    System.out.println("Successful getQueue");
}
```

AQConstants

This class contains some constants used in the java AQ API .

Visibility constants

`VISIBILITY_IMMEDIATE`

```
public static final int VISIBILITY_IMMEDIATE
```

`VISIBILITY_ONCOMMIT`

```
public static final int VISIBILITY_ONCOMMIT
```

Payload type

`RAW_TYPE_PAYLOAD`

```
public static final int RAW_TYPE_PAYLOAD
```

AQAgent

This object specifies the producer or a consumer of a message.

Constructor

Purpose:

There are two implementations of the constructor, each of which allocates a new `AQAgent` with the specified parameters.

Syntax:

```
public AQAgent(java.lang.String name,
               java.lang.String address,
               double protocol)

public AQAgent(java.lang.String name,
               java.lang.String address)
```

Table 7–9 AQAgent Parameters

Parameter	Meaning
`name`	agent name
`address`	agent address
`protocol`	agent protocol (required only in the first constructor); default is 0

getName

Purpose:

This method gets the agent name.

Syntax:

```
public java.lang.String getName() throws AQException
```

setName

Purpose:

This method sets the agent name.

Syntax:

```
public void setName(java.lang.String name) throws AQException
```

Table 7–10 setName Parameters

Parameter	Meaning
`name`	Agent name

getAddress

Purpose:

This method gets the agent address.

Syntax:

```
public java.lang.String getAddress() throws AQException
```

setAddress

Purpose:

This method sets the agent address.

Syntax:

```
public void setAddress(java.lang.String address) throws AQException
```

Table 7–11 setAddress Parameters

Parameter	Meaning
`address`	queue at a specific destination

getProtocol

Purpose:

This method gets the agent protocol.

Syntax:

```
public int getProtocol() throws AQException
```

setProtocol

Purpose:

This method sets the agent protocol.

Syntax:

```
public void setProtocol(int protocol) throws AQException
```

Table 7–12 setProtocol Parameters

Parameter	Meaning
`protocol`	Agent protocol

AQQueueTableProperty

This class represents queue table properties.

Constants for Message Grouping

```
public static final int NONE
public static final int TRANSACTIONAL
```

Constructor

Purpose:

This method creates an `AQQueueTableProperty` object with default property values and the specified payload type.

Syntax:

```
public AQQueueTableProperty(java.lang.String p_type)
```

Table 7–13 AQQueueTableProperty Parameters

Parameter	Meaning
`p_type`	payload type: this is "`RAW`" for queue tables that will contain raw payloads or the object type for queue tables that will contain structured payloads

Note: Currently only payloads of `RAW` type are supported.

getPayloadType

Purpose:

This method returns "`RAW`" for raw payloads or the object type for object payloads.

Syntax:

```
public java.lang.String getPayloadType() throws AQException
```

setPayloadType

Purpose:

This method is used to set the payload type.

Syntax:

```
public void setPayloadType(java.lang.String p_type) throws AQException
```

Table 7–14 setPayloadType Parameters

Parameter	Meaning
`p_type`	payload type: this is “RAW” for queue tables that will contain raw payloads or the object type for queue tables that will contain structured payloads

setStorageClause

Purpose:

This method is used to set the storage clause to be used to create the queue table.

Syntax:

```
public void setStorageClause(java.lang.String s_clause) throws AQException
```

Table 7–15 setStorageClause Parameters

Parameter	Meaning
`s_clauses`	storage parameter: this clause is used in the '`CREATE TABLE`' statement when the queue table is created

getSortOrder

Purpose:

This method gets the sort order that is used.

Syntax:

```
public java.lang.String getSortOrder() throws AQException
```

Returns:

The sort order used

setSortOrder

Purpose:

This method sets the sort order to be used.

Syntax:

```
public void setSortOrder(java.lang.String s_order) throws AQException
```

Table 7–16 setSortOrder Parameters

Parameter	Meaning
`s_order`	specifies the columns to be used as the `sort_key` in ascending order; the string has the format <sort_column1, sort_column2>; the allowed columns name are `priority` and `enq_time`.

isMulticonsumerEnabled

Purpose:

This method queries whether the queues created in the table can have multiple consumers per message or not.

Syntax:

```
public boolean isMulticonsumerEnabled() throws AQException
```

Returns:

`TRUE` if the queues created in the table can have multiple consumers per message.

`FALSE` if the queues created in the table can have only one consumer per message.

setMultiConsumer

Purpose:

This method determines whether the queues created in the table can have multiple consumers per message or not.

Syntax:

```
public void setMultiConsumer(boolean enable) throws AQException
```

Table 7–17 setMultiConsumer Parameters

Parameter	Meaning
`enable`	`FALSE` if the queues created in the table can have only one consumer per message `TRUE` if the queues created in the table can have multiple consumers per message

getMessageGrouping

Purpose:

This method is used to get the message grouping behavior for the queues in this queue table.

Syntax:

```
public int getMessageGrouping() throws AQException
```

Returns:

`NONE`: each message is treated individually

`TRANSACTIONAL`: all messages enqueued as part of one transaction are considered part of the same group and can be dequeued as a group of related messages.

setMessageGrouping

Purpose:

This method is used to set the message grouping behavior for queues created in this queue table.

Syntax:

```
public void setMessageGrouping(int m_grouping) throws AQException
```

Table 7–18 setMessageGrouping Parameters

Parameter	**Meaning**
`m_grouping`	`NONE or TRANSACTIONAL`

getComment

Purpose:

This method gets the queue table comment.

Syntax:

```
public java.lang.String getComment() throws AQException
```

setComment

Purpose:

This method sets a comment.

Syntax:

```
public void setComment(java.lang.String qt_comment) throws AQException
```

Table 7–19 setComment Parameters

Parameter	Meaning
`qt_comment`	`comment`

getCompatible

Purpose:

This method gets the compatible property.

Syntax:

```
public java.lang.String getCompatible() throws AQException
```

setCompatible

Purpose:

This method sets the compatible property.

Syntax:

```
public void setCompatible(java.lang.String qt_compatible) throws AQException
```

Table 7–20 setCompatible Parameters

Parameter	Meaning
`qt_compatible`	`compatible property`

getPrimaryInstance

Purpose:

This method gets the primary instance.

Syntax:

```
public int getPrimaryInstance() throws AQException
```

setPrimaryInstance

Purpose:

This method sets the primary instance.

Syntax:

```
public void setPrimaryInstance(int inst) throws AQException
```

Table 7–21 setPrimaryInstance Parameters

Parameter	Meaning
`inst`	`primary instance`

getSecondaryInstance

Purpose:

This method gets the secondary instance.

Syntax:

```
public int getSecondaryInstance() throws AQException
```

setSecondaryInstance

Purpose:

This method sets the secondary instance.

Syntax:

```
public void setSecondaryInstance(int inst) throws AQException
```

Table 7–22 setSecondaryInstance Parameters

Parameter	Meaning
`inst`	`secondary instance`

Examples:

Set up the test_aqjava class as described in the For more information, see "Setup for AQ Examples" on page 7-10.

1. Create a queue table property object with raw payload type

```
public static void runTest(AQSession aq_sess) throws AQException
 {
     AQQueueTableProperty qtable_prop;

     /* Create AQQueueTable Property object: */
     qtable_prop = new AQQueueTableProperty("RAW");
     qtable_prop.setSortOrder("PRIORITY");
}
```

2. Create a queue table property object with raw payload type (for 8.1 style queues)

```
 public static void runTest(AQSession aq_sess) throws AQException
 {
     AQQueueTableProperty qtable_prop;

     /* Create AQQueueTable Property object: */
     qtable_prop = new AQQueueTableProperty("RAW");
     qtable_prop.setComment("Qtable with raw payload");
     qtable_prop.setCompatible("8.1");
}
```

AQQueueProperty

This class represents queue properties.

Constants:

```
public static final int NORMAL_QUEUE
public static final int EXCEPTION_QUEUE
public static final int INFINITE   /* infinite retention */
```

Constructor:

Purpose:

This method creates a new `AQQueueProperty` object with default property values.

Syntax:

```
public AQQueueProperty()
```

getQueueType

Purpose:

This method gets the queue type .

Syntax:

```
public int getQueueType() throws AQException
```

Returns:

`NORMAL_QUEUE` or `EXCEPTION_QUEUE`

setQueueType

Purpose:

This method is used to set the queue type.

Syntax:

```
public void setQueueType(int q_type) throws AQException
```

Table 7–23 setQueueType Parameters

Parameter	Meaning
`q_type`	`NORMAL_QUEUE` or `EXCEPTION_QUEUE`

getMaxRetries

Purpose:

This method gets the maximum retries for dequeue with `REMOVE` mode.

Syntax:

```
public int getMaxRetries() throws AQException
```

setMaxRetries

Purpose:

This method sets the maximum retries for dequeue with `REMOVE` mode.

Syntax:

```
public void setMaxRetries(int retries) throws AQException
public void setMaxRetries(Integer retries) throws AQException
```

Table 7–24 setMaxRetries Parameters

Parameter	Meaning
`retries`	maximum retries for dequeue with REMOVE mode; specifying `NULL` will use the default. The default applies to single consumer queues and 8.1. compatible multiconsumer queues. `Max_retries` is not supported for 8.0 compatible multiconsumer queues.

setRetryInterval

Purpose:

This method sets the retry interval, that is the time before this message is scheduled for processing after an application rollback. Default is 0.

Syntax:

```
public void setRetryInterval(double interval) throws AQException
public void setRetryInterval(Double interval) throws AQException
```

Table 7–25 setRetryInterval Parameters

Parameter	Meaning
`interval`	retry interval; specifying `NULL` will use the default

getRetryInterval

Purpose:

This method gets the retry interval.

Syntax:

```
public double getRetryInterval() throws AQException
```

getRetentionTime

Purpose:

This method gets the retention time.

Syntax:

```
public double getRetentionTime() throws AQException
```

setRetentionTime

Purpose:

This method gets the retention time.

Syntax:

```
public void setRetentionTime(double r_time) throws AQException
public void setRetentionTime(Double r_time) throws AQException
```

Table 7–26 setRetentionTime Parameters

Parameter	Meaning
`r_time`	retention time; specifying `NULL` will use the default

getComment

Purpose:

This method gets the queue comment.

Syntax:

```
public java.lang.String getComment() throws AQException
```

setComment

Purpose:

This method sets the queue comment.

Syntax:

```
public void setComment(java.lang.String qt_comment) throws AQException
```

Table 7–27 setComment Parameters

Parameter	Meaning
`qt_comment`	queue comment

Example:

Set up the test_aqjava class as described in the Setup for AQ Examples section on on page 7-10, above.

Create a AQQueueProperty object

```
 {
     AQQueueProperty          q_prop;
     q_prop = new AQQueueProperty();
     q_prop.setRetentionTime(15); /* set retention time */
     q_prop.setRetryInterval(30); /* set retry interval  */
}
```

AQQueueTable

The AQQueueTable interface contains methods for queue table administration.

getOwner

Purpose:

This method gets the queue table owner.

Syntax:

```
public java.lang.String getOwner() throws AQException
```

getName

Purpose:

This method gets the queue table name.

Syntax:

```
public java.lang.String getName() throws AQException
```

getProperty

Purpose:

This method gets the queue table properties.

Syntax:

```
public AQQueueTableProperty getProperty() throws AQException
```

Returns:

`AQQueueTableProperty` object

drop

Purpose:

This method drops the current queue table.

Syntax:

```
public void drop(boolean force) throws AQException
```

Table 7–28 drop Parameters

Parameter	Meaning
`force`	`FALSE`: this operation will not succeed if there are any queues in the queue table (the default) `TRUE` : all queues in the queue table are stopped and dropped automatically

alter

Purpose:

This method is used to alter queue table properties.

Syntax:

```
public void alter(java.lang.String comment,
                  int primary_instance,
                  int secondary_instance) throws AQException

public void alter(java.lang.String comment) throws AQException
```

Table 7–29 alter Parameters

Parameter	Meaning
`comment`	new comment
`primary_instance`	new value for primary instance
`secondary_instance`	new value for secondary instance

createQueue

Purpose:

This method is used to create a queue in this queue table.

Syntax:

```
public AQQueue createQueue(java.lang.String queue_name,
                           AQQueueProperty q_property) throws AQException
```

Table 7–30 createQueue Parameters

Parameter	Meaning
`queue_name`	name of the queue to be created
`q_property`	queue properties

Returns:

`AQQueue` object

dropQueue

Purpose:

This method is used to drop a queue in this queue table.

Syntax:

```
public void dropQueue(java.lang.String queue_name) throws AQException
```

Table 7–31 dropQueue Parameters

Parameter	Meaning
queue_name	name of the queue to be dropped

Example:

Set up the test_aqjava class as described in the Setup for AQ Examples section on on page 7-10, above.

1. Create a queue table and a queue

```
public static void runTest(AQSession aq_sess) throws AQException
{
   AQQueueTableProperty    qtable_prop;
   AQQueueProperty         queue_prop;
   AQQueueTable            q_table;
   AQQueue                 queue;

   /* Create a AQQueueTable property object (payload type - RAW): */
   qtable_prop = new AQQueueTableProperty("RAW");

   /* Create a queue table called aq_table2 in aquser schema: */
   qtable = aq_sess.createQueueTable ("aquser", "aq_table2", qtable_prop);
   System.out.println("Successfully createQueueTable");

   /* Create a new AQQueueProperty object: */
   queue_prop = new AQQueueProperty();

   /* Create a queue called aq_queue2 in aq_table2: */
   queue = qtable.createQueue ("aq_queue2", queue_prop);
   System.out.println("Successful createQueue");
}
```

2. Alter queue table, get properties and drop the queue table

```
{
     AQQueueTableProperty    qtable_prop;
     AQQueueTable            q_table;

     /* Get a handle to the queue table called aq_table2 in aquser schema: */
     q_table = aq_sess.getQueueTable ("aqjava", "aq_table2");
```

```
        System.out.println("Successful getQueueTable");

        /* Get queue table properties: */
        qtable_prop = q_table.getProperty();

        /* Alter the queue table: */
        q_table.alter("altered queue table");

        /* Drop the queue table (and automatically drop queues inside it): */
        q_table.drop(true);
        System.out.println("Successful drop");
    }
```

Note: Queues can be created via the `AQSession.createQueue` or the `AQQueueTable.createQueue` `interfaces`. The former expects an `AQQueueTable` object as a parameter in addition to the `queue_name` and queue properties.

AQQueueAdmin

start

Purpose:

This method is used to enable enqueue and dequeue on this queue.

Syntax:

```
public void start(boolean enqueue,
                  boolean dequeue) throws AQException
```

Table 7–32 start Parameters

Parameter	Meaning
`enqueue`	`TRUE` — enable enqueue on this queue `FALSE` — leave current setting unchanged
`dequeue`	`TRUE` — enable dequeue on this queue `FALSE` — leave current setting unchanged

startEnqueue

Purpose:

This method is used to enable enqueue on this queue . This is equivalent to `start(TRUE, FALSE)`

Syntax:

```
public void startEnqueue() throws AQException
```

startDequeue

Purpose:

This method is used to enable dequeue on this queue. This is equivalent to `start(FALSE, TRUE)`.

Syntax:

```
public void startDequeue() throws AQException
```

stop

Purpose:

This method is used to disable enqueue / dequeue on this queue.

Syntax:

```
public void stop(boolean enqueue,
                 boolean dequeue,
                 boolean wait) throws AQException
```

Table 7–33 stop Parameters

Parameter	Meaning
`enqueue`	`TRUE` — disable enqueue on this queue `FALSE` — leave current setting unchanged
`dequeue`	`TRUE` — disable dequeue on this queue `FALSE` — leave current setting unchanged
`wait`	`TRUE` — wait for outstanding transactions to complete `FALSE` — return immediately either with a success or an error

stopEnqueue

Purpose:

This method is used to disable enqueue on a queue. This is equivalent to `stop(TRUE, FALSE, wait)`.

Syntax:

```
public void stopEnqueue(boolean wait) throws AQException
```

Table 7–34 stopEnqueue Parameters

Parameter	Meaning
`wait`	`TRUE` — wait for outstanding transactions to complete `FALSE` — return immediately either with a success or an error

stopDequeue

Purpose:

This method is used to disable dequeue on a queue. This is equivalent to `stop(FALSE, TRUE, wait)`.

Syntax:

```
public void stopDequeue(boolean wait) throws AQException
```

Table 7–35 stopDequeue Parameters

Parameter	Meaning
`wait`	`TRUE` — wait for outstanding transactions to complete `FALSE` — return immediately either with a success or an error

drop

Purpose:

This method is used to drop a queue

Syntax:

```
public void drop() throws AQException
```

alterQueue

Purpose:

This method is used to alter queue properties

Syntax:

```
public void alterQueue(AQQueueProperty property) throws AQException
```

Table 7–36 alterQueue Parameters

Parameter	Meaning
`property`	`AQQueueProperty` object with new property values. Note that only `max_retries`, `retry_delay`, `retention_time` and `comment` can be altered.

addSubscriber

Purpose:

This method is used to add a subscriber for this queue.

Syntax:

```
public void addSubscriber(AQAgent subscriber,
                          java.lang.String rule) throws AQException
```

Table 7–37 addSubscriber Parameters

Parameter	Meaning
`subscriber`	the `AQAgent` on whose behalf the subscription is being defined
`rule`	a conditional expression based on message properties, and the message data properties

removeSubscriber

Purpose:

This method removes a subscriber from a queue.

Syntax:

```
public void removeSubscriber(AQAgent subscriber) throws AQException
```

Table 7–38 removeSubscriber Parameters

Parameter	Meaning
`subscriber`	the `AQAgent` to be removed

alterSubscriber

Purpose:

This method alters properties for a subscriber to a queue.

Syntax:

```
public void alterSubscriber(AQAgent subscriber,
                            java.lang.String rule) throws AQException
```

Table 7–39 alterSubscriber Parameters

Parameter	Meaning
`subscriber`	the `AQAgent` whose subscription is being altered
`rule`	a conditional expression based on message properties, the message data properties

grantQueuePrivilege

Purpose:

This method is used to grant queue privileges to users and roles. The method has been overloaded. The second implementation is equivalent to to calling the first imeplementation with `grant_option` = `FALSE` .

Syntax:

```
public void grantQueuePrivilege(java.lang.String privilege,
                                java.lang.String grantee,
                                boolean grant_option) throws AQException

public void grantQueuePrivilege(java.lang.String privilege,
                                java.lang.String grantee) throws AQException
```

Table 7–40 grantQueuePrivilege Parameters

Parameter	Meaning
`privilege`	specifies the privilege to be granted`:` `ENQUEUE`, `DEQUEUE` or `ALL`
`grantee`	specifies the grantee(s); the grantee(s) can be a user, a role or the `PUBLIC` roles
`grant_option`	`TRUE` — the grantee is allowed to use this method to grant access to others `FALSE` — default

revokeQueuePrivilege

Purpose:

This method is used to revoke a queue privilege.

Syntax:

```
public void revokeQueuePrivilege(java.lang.String privilege,
                                  java.lang.String grantee) throws AQException
```

Table 7–41 revokeQueuePrivilege Parameters

Parameter	Meaning
`privilege`	specifies the privilege to be revoked: `ENQUEUE`, `DEQUEUE` or `ALL`
`grantee`	specifies the grantee(s); the grantee(s) can be a user, a role or the `PUBLIC` roles

schedulePropagation

Purpose:

This method is used to schedule propagation from a queue to a destination identified by a database link.

Syntax:

```
public void schedulePropagation(java.lang.String destination,
                                java.util.Date start_time,
                                java.lang.Double duration,
                                java.lang.String next_time,
                                java.lang.Double latency) throws AQException
```

Table 7–42 scheduleProgation Parameters

Parameter	Meaning
`destination`	specifies the destination database link. Messages in the source queue for recipients at the destination will be propagated. `NULL` => destination is the local database and messages will be propagated to all other queues in the local database. Maximum length for this field is 128 bytes. If the name is not fully qualified, the default domain name is used.
`start_time`	specifies the initial start time for the propagation window for messages from this queue to the destination. `NULL` => start time is current time.
`duration`	specifies the duration of the propagation window in seconds. `NULL` => propagation window is forever or until propagation is unscheduled
`next_time`	date function to compute the start of the next propagation window from the end of the current window. (e.g use "`SYSDATE`+ 1 - duration/86400" to start the window at the same time everyday. `NULL` => propagation will be stopped at the end of the current window
`latency`	maximum wait, in seconds, in the propagation window for the message to be propagated after it is enqueued. `NULL` => use default value (60 seconds)

unschedulePropagation

Purpose:

This method is used to unschedule a previously scheduled propagation of messages from the current queue to a destination identfied by a specific database link..

Syntax:

```
public void unschedulePropagation(java.lang.String destination)
                                  throws AQException
```

Table 7–43 unschedulePropagation Parameters

Parameter	Meaning
destination	specifies the destination database link. `NULL` => destination is the local database.

alterPropagationSchedule

Purpose:

This method is used to alter a propagation schedule.

Syntax:

```
public void alterPropagationSchedule(java.lang.String destination,
                                     java.lang.Double duration,
                                     java.lang.String next_time,
                                     java.lang.Double latency)throws AQException
```

Table 7–44 alterPropagationSchedule Parameters

Parameter	Meaning
destination	specifies the destination database link. `NULL` => destination is the local database.
duration	specifies the duration of the propagation window in seconds. `NULL` => propagation window is forever or until propagation is unscheduled
next_time	date function to compute the start of the next propagation window from the end of the current window. (e.g use "`SYSDATE`+ 1 - duration / 86400" to start the window at the same time everyday. `NULL` => propagation will be stopped at the end of the current window
latency	maximum wait, in seconds, in the propagation window for the message to be propagated after it is enqueued. `NULL` => use default value (60 seconds)

enablePropagationSchedule

Purpose:

This method is used to enable a propagation schedule.

Syntax:

```
public void enablePropagationSchedule(java.lang.String destination)
                                      throws AQException
```

Table 7–45 enablePropagationSchedule Parameters

Parameter	Meaning
destination	specifies the destination database link. `NULL` => destination is the local database.

disablePropagationSchedule

Purpose:

This method is used to disable a propagation schedule.

Syntax:

```
public void disablePropagationSchedule(java.lang.String destination)
                                       throws AQException
```

Table 7–46 disablePropagationSchedule Parameters

Parameter	Meaning
destination	specifies the destination database link. `NULL` => destination is the local database.

Examples:

Set up the test_aqjava class. For more information, see "Setup for AQ Examples" on page 7-10

1. Create a queue and start enqueue/dequeue

```
{
     AQQueueTableProperty    qtable_prop;
     AQQueueProperty         queue_prop;
     AQQueueTable            q_table;
     AQQueue                 queue;

     /* Create a AQQueueTable property object (payload type - RAW): */
     qtable_prop = new AQQueueTableProperty("RAW");
    qtable_prop.setCompatible("8.1");

     /* Create a queue table called aq_table3 in aqjava schema: */
     q_table = aq_sess.createQueueTable ("aqjava", "aq_table3", qtable_prop);
     System.out.println("Successful createQueueTable");

     /* Create a new AQQueueProperty object: */
     queue_prop = new AQQueueProperty();

     /* Create a queue called aq_queue3 in aq_table3: */
     queue = aq_sess.createQueue (q_table, "aq_queue3", queue_prop);
     System.out.println("Successful createQueue");

     /* Enable enqueue/dequeue on this queue: */
     queue.start();
     System.out.println("Successful start queue");

     /* Grant enqueue_any privilege on this queue to user scott: */
     queue.grantQueuePrivilege("ENQUEUE", "scott");
     System.out.println("Successful grantQueuePrivilege");
}
```

2. Create a multi-consumer queue and add subscribers

```
public static void runTest(AQSession aq_sess) throws AQException
{
     AQQueueTableProperty    qtable_prop;
     AQQueueProperty         queue_prop;
     AQQueueTable            q_table;
```

```
        AQQueue                  queue;
        AQAgent                   subs1, subs2;

        /* Create a AQQueueTable property object (payload type - RAW): */
        qtable_prop = new AQQueueTableProperty("RAW");
        System.out.println("Successful setCompatible");

        /* Set multiconsumer flag to true: */
        qtable_prop.setMultiConsumer(true);

        /* Create a queue table called aq_table4 in aqjava schema: */
        q_table = aq_sess.createQueueTable ("aqjava", "aq_table4", qtable_prop);
        System.out.println("Successful createQueueTable");

        /* Create a new AQQueueProperty object: */
        queue_prop = new AQQueueProperty();
        /* Create a queue called aq_queue4 in aq_table4 */
        queue = aq_sess.createQueue (q_table, "aq_queue4", queue_prop);
        System.out.println("Successful createQueue");

        /* Enable enqueue/dequeue on this queue: */
        queue.start();
        System.out.println("Successful start queue");

        /* Add subscribers to this queue: */
        subs1 = new AQAgent("GREEN", null, 0);
        subs2 = new AQAgent("BLUE", null, 0);

        queue.addSubscriber(subs1, null); /* no rule   */
        System.out.println("Successful addSubscriber 1");

        queue.addSubscriber(subs2, "priority < 2"); /* with rule */
        System.out.println("Successful addSubscriber 2");
   }
```

AQQueue

This interface supports the operational interfaces of queues. `AQQueue` extends `AQQueueAdmin`. Hence, you can also use adminstrative functions through this interface.

getOwner

Purpose:

This method gets the queue owner.

Syntax:

```
public java.lang.String getOwner() throws AQException
```

getName

Purpose:

This method gets the queue name.

Syntax:

```
public java.lang.String getName() throws AQException
```

getQueueTableName

Purpose:

This method gets the name of the queue table in which the queue resides.

Syntax:

```
public java.lang.String getQueueTableName() throws AQException
```

getProperty

Purpose:

This method is used to get the queue properties.

Syntax:

```
public AQQueueProperty getProperty() throws AQException
```

Returns:

`AQQueueProperty` object

createMessage

Purpose:

This method is used to create a new `AQMessage` object that can be populated with data to be enqueued.

Syntax:

```
public AQMessage createMessage() throws AQException
```

Returns:

`AQMessage` object

enqueue

Purpose:

This method is used to enqueue a message in a queue.

Syntax:

```
public byte[] enqueue(AQEnqueueOption enq_option,
                      AQMessage message) throws AQException
```

Table 7–47 alterPropagationSchedule Parameters

Parameter	Meaning
`enq_option`	`AQEnqueOption` object
`message`	`AQMessage` to be enqueued

Returns:

Message id of the the enqueued message. The AQMessage object's messageId field is also populated after the completion of this call.

dequeue

Purpose:

This method is used to dequeue a message from a queue.

Syntax:

```
public AQMessage dequeue(AQDequeueOption deq_option) throws AQException
```

Table 7–48 alterPropagationSchedule Parameters

Parameter	Meaning
deq_option	`AQDequeueOption` object

Returns:

`AQMessage`,the dequeued message

getSubscribers

Purpose:

This method is used to get a subscriber list for the queue.

Syntax:

```
public AQAgent[] getSubscribers() throws AQException
```

Returns:

An array of `AQAgents`

AQEnqueueOption

This class is used to specify options available for the enqueue operation.

Constants

```
public static final int DEVIATION_NONE
public static final int DEVIATION_BEFORE
public static final int DEVIATION_TOP
public static final int VISIBILITY_ONCOMMIT
public static final int VISIBILITY_IMMEDIATE
```

Constructors

Purpose:

There are two constructors available. The first creates an object with the specified options, the second creates an object with the default options.

Syntax:

```
public AQEnqueueOption(int visibility,
                       byte[] relative_msgid,
                       int sequence_deviation)

public AQEnqueueOption()
```

Table 7–49 AQEnqueueOption Parameters

Parameter	Meaning
`visibility`	`VISIBILITY_IMMEDIATE` or `VISIBILITY_ONCOMMIT` (default)
`relative_msgid`	when `DEVIATION_BEFORE` is used, this parameter identifies the message identifier of the message before which the current message is to be enqueued
`sequence_deviation`	`DEVIATION_TOP`— the message is enqueued ahead of any other messages `DEVIATION_BEFORE` — the message is enqueued ahead of the message specified by relative_msgid `DEVIATION_NONE` — default

getVisibility

Purpose:

This method gets the visibility.

Syntax:

```
public int getVisibility() throws AQException
```

Returns:

`VISIBILITY_IMMEDIATE` or `VISIBILITY_ONCOMMIT`

setVisibility

Purpose:

This method sets the visibility.

Syntax:

```
public void setVisibility(int visibility) throws AQException
```

Table 7–50 setVisibility Parameters

Parameter	Meaning
`visibility`	`VISIBILITY_IMMEDIATE` or `VISIBILITY_ONCOMMIT`

getRelMessageId

Purpose:

This method gets the relative message id.

Syntax:

```
public byte[] getRelMessageId() throws AQException
```

getSequenceDeviation

Purpose:

This method gets the sequence deviation.

Syntax:

```
public int getSequenceDeviation() throws AQException
```

setSequenceDeviation

Purpose:

This method specfies whether the message being enqueued should be dequeued before other message(s) already in the queue

Syntax:

```
public void setSequenceDeviation(int sequence_deviation,
                                 byte[] relative_msgid) throws AQException
```

Table 7–51 setSequenceDeviation Parameters

Parameter	Meaning
`sequence_deviation`	`DEVIATION_TOP`— the message is enqueued ahead of any other messages `DEVIATION_BEFORE` — the message is enqueued ahead of the message specified by relative_msgid `DEVIATION_NONE` — default
`relative_msgid`	when `DEVIATION_BEFORE` is used, this parameter identifies the message identifier of the message before which the current message is to be enqueued

AQDequeueOption

This class is used to specify the options available for the dequeue option.

Constants

```
public static final int NAVIGATION_FIRST_MESSAGE
public static final int NAVIGATION_NEXT_TRANSACTION
public static final int NAVIGATION_NEXT_MESSAGE
public static final int DEQUEUE_BROWSE
public static final int DEQUEUE_LOCKED
public static final int DEQUEUE_REMOVE
public static final int DEQUEUE_REMOVE_NODATA
public static final int WAIT_FOREVER
public static final int WAIT_NONE
public static final int VISIBILITY_ONCOMMIT
public static final int VISIBILITY_IMMEDIATE
```

Constructor

Purpose:

This method creates an object with the default options.

Syntax:

```
public AQDequeueOption()
```

getConsumerName

Purpose:

This method gets consumer name.

Syntax:

```
public java.lang.String getConsumerName() throws AQException
```

setConsumerName

Purpose:

This method sets consumer name

Syntax:

```
public void setConsumerName(java.lang.String consumer_name) throws AQException
```

Table 7–52 setConsumerName Parameters

Parameter	Meaning
`consumer_name`	Agent name

getDequeueMode

Purpose:

This method gets dequeue mode

Syntax:

```
public int getDequeueMode() throws AQException
```

Returns:

`DEQUEUE_BROWSE`, `DEQUEUE_LOCKED`, `DEQUEUE_REMOVE` or `DEQUEUE_REMOVE_NODATA`

setDequeueMode

Purpose:

This method sets the dequeue mode.

Syntax:

```
public void setDequeueMode(int dequeue_mode) throws AQException
```

Table 7–53 setDequeueMode Parameters

Parameter	Meaning
dequeue_mode	`DEQUEUE_BROWSE`, `DEQUEUE_LOCKED`, `DEQUEUE_REMOVE` or `DEQUEUE_REMOVE_NODATA`

getNavigationMode

Purpose:

This method gets the navigation mode.

Syntax:

```
public int getNavigationMode() throws AQException
```

Returns:

`NAVIGATION_FIRST_MESSAGE` or `NAVIGATION_NEXT_MESSAGE` or `NAVIGATION_NEXT_TRANSACTION`

setNavigationMode

Purpose:

This method sets the navigation mode.

Syntax:

```
public void setNavigationMode(int navigation) throws AQException
```

Table 7–54 setNavigationMode Parameters

Parameter	Meaning
`navigation`	`NAVIGATION_FIRST_MESSAGE` or `NAVIGATION_NEXT_MESSAGE` or `NAVIGATION_NEXT_TRANSACTION`

getVisibility

Purpose:

This method gets the visibility.

Syntax:

```
public int getVisibility() throws AQException
```

Returns:

`VISIBILITY_IMMEDIATE` or `VISIBILITY_ONCOMMIT`

setVisibility

Purpose:

This method sets the visibility.

Syntax:

```
public void setVisibility(int visibility) throws AQException
```

Table 7–55 setVisibility Parameters

Parameter	Meaning
`visibility`	`VISIBILITY_IMMEDIATE` or `VISIBILITY_ONCOMMIT`

getWaitTime

Purpose:

This method gets the wait time.

Syntax:

```
public int getWaitTime() throws AQException
```

Returns:

`WAIT_FOREVER` or `WAIT_NONE` or the actual time in seconds

setWaitTime

Purpose:

This method sets the wait time.

Syntax:

```
public void setWaitTime(int wait_time) throws AQException
```

Table 7–56 setWaitTime Parameters

Parameter	Meaning
`wait_time`	`WAIT_FOREVER` or `WAIT_NONE` or time in seconds

getMessageId

Purpose:

This method gets the message id.

Syntax:

```
public byte[] getMessageId() throws AQException
```

setMessageId

Purpose:

This method sets the message id.

Syntax:

```
public void setMessageId(byte[] message_id) throws AQException
```

Table 7–57 setMessageId Parameters

Parameter	Meaning
`message_id`	message id

getCorrelation

Purpose:

This method gets the correlation id.

Syntax:

```
public java.lang.String getCorrelation() throws AQException
```

setCorrelation

Purpose:

This method sets the correlation id.

Syntax:

```
public void setCorrelation(java.lang.String correlation) throws AQException
```

Table 7–58 setCorrelation Parameters

Parameter	Meaning
`correlation`	user-supplied information

AQMessage

This interface contains methods for AQ messages with raw or object payloads.

getMessageId

Purpose:

This method gets the message id.

Syntax:

```
public byte[] getMessageId() throws AQException
```

getRawPayload

Purpose:

This method gets the raw payload

Syntax:

```
public AQRawPayload getRawPayload() throws AQException
```

Returns:

`AQRawPayload` object

setRawPayload

Purpose:

This method sets the raw payload. It throws `AQException` if this is called on messages created from object type queues.

Syntax:

```
public void setRawPayload(AQRawPayload message_payload) throws AQException
```

Table 7–59 setRawPayload Parameters

Parameter	Meaning
message_payload	`AQRawPayload` object containing raw user data

getMessageProperty

Purpose:

This method gets the message properties

Syntax:

```
public AQMessageProperty getMessageProperty() throws AQException
```

Returns:

`AQMessageProperty` object

setMessageProperty

Purpose:

This method sets the message properties.

Syntax:

```
public void setMessageProperty(AQMessageProperty property) throws AQException
```

Table 7–60 setObjectPayload Parameters

Parameter	Meaning
`property`	AQMessageProperty object

AQMessageProperty

The AQMessageProperty class contains information that is used by AQ to manage individual messages. The properties are set at enqueue time and their values are returned at dequeue time.

Constants

```
public static final int DELAY_NONE
public static final int EXPIRATION_NEVER
public static final int STATE_READY
public static final int STATE_WAITING
public static final int STATE_PROCESSED
public static final int STATE_EXPIRED
```

Constructor

Purpose:

This method creates the AQMessageProperty object with default property values.

Syntax:

```
public AQMessageProperty()
```

getPriority

Purpose:

This method gets the message priority.

Syntax:

```
public int getPriority() throws AQException
```

setPriority

Purpose:

This method sets the message priority.

Syntax:

```
public void setPriority(int priority) throws AQException
```

Table 7–61 setPriority Parameters

Parameter	Meaning
`priority`	priority of the message; this can be any number, including negative number - a smaller number indicates a higher priority

getDelay

Purpose:

This method gets the delay value.

Syntax:

```
public int getDelay() throws AQException
```

setDelay

Purpose:

This method sets delay value.

Syntax:

```
public void setDelay(int delay) throws AQException
```

Table 7–62 setDelay Parameters

Parameter	Meaning
`delay`	the delay represents the number of seconds after which the message is available for dequeuing; with `NO_DELAY` the message is available for immediate dequeuing

getExpiration

Purpose:

This method gets expiration value

Syntax:

```
public int getExpiration() throws AQException
```

setExpiration

Purpose:

This method sets expiration value

Syntax:

```
public void setExpiration(int expiration) throws AQException
```

Table 7–63 setExpiration Parameters

Parameter	Meaning
`expiration`	the duration the message is available for dequeuing; this parameter is an offset from the delay; if `NEVER`, the message will not expire

getCorrelation

Purpose:

This method gets correlation

Syntax:

```
public java.lang.String getCorrelation() throws AQException
```

setCorrelation

Purpose:

This method sets correlation

Syntax:

```
public void setCorrelation(java.lang.String correlation) throws AQException
```

Table 7–64 setCorrelation Parameters

Parameter	**Meaning**
`correlation`	user-supplied information

getAttempts

Purpose:

This method gets the number of attempts.

Syntax:

```
public int getAttempts() throws AQException
```

getRecipientList

Purpose:

This method gets the recipient list.

Syntax:

```
public java.util.Vector getRecipientList() throws AQException
```

Returns:

A vector of `AQAgents`.This parameter is not returned to a consumer at dequeue time.

setRecipientList

Purpose:

This method sets the recipient list.

Syntax:

```
public void setRecipientList(java.util.Vector r_list) throws AQException
```

Table 7–65 setRecipientList Parameters

Parameter	Meaning
`r_list`	vector of `AQAgents`; the default recipients are the queue subscribers

getOrigMessageId

Purpose:

This method gets original message id.

Syntax:

```
public byte[] getOrigMessageId() throws AQException
```

getSender

Purpose:

This method gets the sender of the message.

Syntax:

```
public AQAgent getSender() throws AQException
```

setSender

Purpose:

This method sets the sender of the message.

Syntax:

```
public void setSender(AQAgent sender) throws AQException
```

Table 7–66 setSender Parameters

Parameter	Meaning
sender	AQAgent

getExceptionQueue

Purpose:

This method gets the exception queue name.

Syntax:

```
public java.lang.String getExceptionQueue() throws AQException
```

setExceptionQueue

Purpose:

This method sets the exception queue name.

Syntax:

```
public void setExceptionQueue(java.lang.String queue) throws AQException
```

Table 7–67 setExceptionQueue Parameters

Parameter	Meaning
queue	exception queue name

getEnqueueTime

Purpose:

This method gets the enqueue time.

Syntax:

```
public java.util.Date getEnqueueTime() throws AQException
```

getState

Purpose:

This method gets the message state.

Syntax:

```
public int getState() throws AQException
```

Returns:

`STATE_READY` or `STATE_WAITING` or `STATE_PROCESSED` or `STATE_EXPIRED`

AQRawPayload

This object represents the raw user data that is included in AQMessage.

getStream

Purpose:

This method reads some portion of the raw payload data into the specified byte array.

Syntax:

```
public int getStream(byte[] value, int len) throws AQException
```

Table 7–68 getStream Parameters

Parameter	Meaning
`value`	byte array to hold the raw data
`len`	number of bytes to be read

Returns:

The number of bytes read

getBytes

Purpose:

This method retrieves the entire raw payload data as a byte array.

Syntax:

```
public byte[] getBytes() throws AQException
```

Returns:

byte[] - the raw payload as a byte array

setStream

Purpose:

This method sets the value of the raw payload.

Syntax:

```
public void setStream(byte[] value,
                      int len) throws AQException
```

Table 7–69 getStream Parameters

Parameter	Meaning
`value`	byte array containing the raw payload
`len`	number of bytes to be written to the raw stream

AQException

This exception is raised when the user encounters any error while using the Java AQ Api.

public class AQException extends java.lang.RuntimeException

This interface supports all methods supported by Java exceptions and some additional methods.

getMessage

Purpose:

This method gets the error message.

getErrorCode

Purpose:

This method gets the error number (Oracle error code).

getNextException

Purpose:

This method gets the next exception in the chain if any.

AQOracleSQLException

`AQOracleSQLException` extends `AQException`.

When using Oracle8*i* AQ driver, some errors may be raised from the client side and some from the RDBMS. The Oracle8*i* driver raises `AQOracleSQLException` for all errors that ocuur while performing SQL.

For sophisticated users interested in differentiating between the two types of exceptions, this interface might be useful. In general you will only use `AQException`.

8

Oracle Advanced Queuing by Example

In this chapter we provide examples of operations using different programatic environments:

- Create Queue Tables and Queues
    - Create a Queue Table and Queue of Object Type
    - Create a Queue Table and Queue of Raw Type
    - Create a Prioritized Message Queue Table and Queue
    - Create a Multiple-Consumer Queue Table and Queue
    - Create a Queue to Demonstrate Propagation
- Enqueue and Dequeue Of Messages
    - Enqueue and Dequeue of Object Type Messages Using PL/SQL
    - Enqueue and Dequeue of Object Type Messages Using Pro*C/C++
    - Enqueue and Dequeue of Object Type Messages Using OCI
    - Enqueue and Dequeue of RAW Type Messages Using PL/SQL
    - Enqueue and Dequeue of RAW Type Messages Using Pro*C/C++
    - Enqueue and Dequeue of RAW Type Messages Using OCI
    - Enqueue and Dequeue of RAW Type Messages Using Java
    - Dequeue of Messages Using Java
    - Dequeue of Messages in Browse Mode Using Java
    - Enqueue and Dequeue of Messages by Priority Using PL/SQL

Create Queue Tables and Queues

Note: You may need to set up the following data structures for certain examples to work:

```
CONNECT system/manager;
DROP USER aqadm CASCADE;
GRANT CONNECT, RESOURCE TO aqadm;
CREATE USER aqadm IDENTIFIED BY aqadm;
GRANT EXECUTE ON DBMS_AQADM TO aqadm;
GRANT Aq_administrator_role TO aqadm;
DROP USER aq CASCADE;
CREATE USER aq IDENTIFIED BY aq;
GRANT CONNECT, RESOURCE TO aq;
GRANT EXECUTE ON dbms_aq TO aq;
```

Create a Queue Table and Queue of Object Type

```
/* Create a message type: */
CREATE type aq.Message_typ as object (
subject     VARCHAR2(30),
text        VARCHAR2(80));

/* Create a object type queue table and queue: */
EXECUTE DBMS_AQADM.CREATE_QUEUE_TABLE (
queue_table        => 'aq.objmsgs80_qtab',
queue_payload_type => 'aq.Message_typ');

EXECUTE DBMS_AQADM.CREATE_QUEUE (
queue_name         => 'msg_queue',
queue_table        => 'aq.objmsgs80_qtab');

EXECUTE DBMS_AQADM.START_QUEUE (
queue_name         => 'msg_queue');
```

Create a Queue Table and Queue of Raw Type

```
/* Create a RAW type queue table and queue: */
EXECUTE DBMS_AQADM.CREATE_QUEUE_TABLE (
queue_table          => 'aq.RawMsgs_qtab',
queue_payload_type   => 'RAW');

EXECUTE DBMS_AQADM.CREATE_QUEUE (
```

```
queue_name          => 'raw_msg_queue',
queue_table         => 'aq.RawMsgs_qtab');

EXECUTE DBMS_AQADM.START_QUEUE (
queue_name          => 'raw_msg_queue');
```

Create a Prioritized Message Queue Table and Queue

```
EXECUTE DBMS_AQADM.CREATE_QUEUE_TABLE (
queue_table        => 'aq.priority_msg',
sort_list          => 'PRIORITY,ENQ_TIME',
queue_payload_type => 'aq.Message_typ');

EXECUTE DBMS_AQADM.CREATE_QUEUE (
queue_name         => 'priority_msg_queue',
queue_table        => 'aq.priority_msg');

EXECUTE DBMS_AQADM.START_QUEUE (
queue_name         => 'priority_msg_queue');
```

Create a Multiple-Consumer Queue Table and Queue

```
EXECUTE DBMS_AQADM.CREATE_QUEUE_TABLE (
queue_table        => 'aq.MultiConsumerMsgs_qtab',
multiple_consumers => TRUE,
queue_payload_type => 'aq.Message_typ');

EXECUTE DBMS_AQADM.CREATE_QUEUE (
queue_name         => 'msg_queue_multiple',
queue_table        => 'aq.MultiConsumerMsgs_qtab');

EXECUTE DBMS_AQADM.START_QUEUE (
queue_name         => 'msg_queue_multiple');
```

Create a Queue to Demonstrate Propagation

```
EXECUTE DBMS_AQADM.CREATE_QUEUE (
queue_name        => 'another_msg_queue',
queue_table       => 'aq.MultiConsumerMsgs_qtab');

EXECUTE DBMS_AQADM.START_QUEUE (
```

```
queue_name          => 'another_msg_queue');
```

Enqueue and Dequeue Of Messages

Enqueue and Dequeue of Object Type Messages Using PL/SQL

To enqueue a single message without any other parameters specify the queue name and the payload.

```
/* Enqueue to msg_queue: */
DECLARE
   enqueue_options     dbms_aq.enqueue_options_t;
   message_properties  dbms_aq.message_properties_t;
   message_handle      RAW(16);
   message             aq.message_typ;

BEGIN
   message := message_typ('NORMAL MESSAGE',
   'enqueued to msg_queue first.');

   dbms_aq.enqueue(queue_name => 'msg_queue',
         enqueue_options      => enqueue_options,
         message_properties   => message_properties,
         payload              => message,
         msgid                => message_handle);

   COMMIT;

/* Dequeue from msg_queue: */
DECLARE
   dequeue_options     dbms_aq.dequeue_options_t;
   message_properties  dbms_aq.message_properties_t;
   message_handle      RAW(16);
   message             aq.message_typ;

BEGIN
   DBMS_AQ.DEQUEUE(queue_name => 'msg_queue',
           dequeue_options    => dequeue_options,
           message_properties => message_properties,
           payload            => message,
           msgid              => message_handle);

   DBMS_OUTPUT.PUT_LINE ('Message: ' || message.subject ||
```

```
                                            ' ... ' || message.text );
   COMMIT;
END;
```

Enqueue and Dequeue of Object Type Messages Using Pro*C/C++

Note: You may need to set up data structures similar to the following for certain examples to work:

```
$ cat >> message.typ
case=lower
type aq.message_typ
$
$ ott userid=aq/aq intyp=message.typ outtyp=message_o.typ \
code=c hfile=demo.h
$
$ proc intyp=message_o.typ iname=<program name> \
config=<config file> SQLCHECK=SEMANTICS userid=aq/aq
```

```
#include  <stdio.h>
#include  <string.h>
#include  <sqlca.h>
#include  <sql2oci.h>
/* The header file generated by processing
object type 'aq.Message_typ': */
#include  "pceg.h"

void sql_error(msg)
char *msg;
{
EXEC SQL WHENEVER SQLERROR CONTINUE;
printf("%s\n", msg);
printf("\n% .800s \n", sqlca.sqlerrm.sqlerrmc);
EXEC SQL ROLLBACK WORK RELEASE;
exit(1);
}

main()
{
Message_typ     *message = (Message_typ*)0;  /* payload */
char             user[60]="aq/AQ";  /* user logon password */
char             subject[30];  /* components of the */
```

```
char              txt[80];      /* payload type */

/* ENQUEUE and DEQUEUE to an OBJECT QUEUE */

/*  Connect to database: */
EXEC SQL CONNECT :user;

/* On an oracle error print the error number :*/
EXEC SQL WHENEVER SQLERROR DO sql_error("Oracle Error :");

/* Allocate memory for the host variable from the object cache : */
EXEC SQL ALLOCATE :message;

/* ENQUEUE */

strcpy(subject, "NORMAL ENQUEUE");
strcpy(txt, "The Enqueue was done through PLSQL embedded in PROC");

/* Initialize the components of message : */
EXEC SQL OBJECT SET subject, text OF  :message TO :subject, :txt;

/* Embedded PLSQL call to the AQ enqueue procedure : */
EXEC SQL EXECUTE
DECLARE
message_properties   dbms_aq.message_properties_t;
enqueue_options      dbms_aq.enqueue_options_t;
msgid                RAW(16);
BEGIN
/* Bind the host variable 'message' to the payload: */
dbms_aq.enqueue(queue_name => 'msg_queue',
message_properties => message_properties,
enqueue_options => enqueue_options,
payload => :message,
msgid => msgid);
END;
END-EXEC;
/* Commit work */
EXEC SQL COMMIT;

printf("Enqueued Message \n");
printf("Subject  :%s\n",subject);
printf("Text     :%s\n",txt);

/* Dequeue */
```

```
/* Embedded PLSQL call to the AQ dequeue procedure : */
EXEC SQL EXECUTE
DECLARE
message_properties  dbms_aq.message_properties_t;
dequeue_options     dbms_aq.dequeue_options_t;
msgid               RAW(16);
BEGIN
/* Return the payload into the host variable ’message’:  */
dbms_aq.dequeue(queue_name => ’msg_queue’,
message_properties => message_properties,
dequeue_options => dequeue_options,
payload => :message,
msgid => msgid);
END;
END-EXEC;
 /* Commit work :*/
EXEC SQL COMMIT;

/* Extract the components of message: */
EXEC SQL OBJECT GET SUBJECT,TEXT FROM :message INTO :subject,:txt;

printf("Dequeued Message \n");
printf("Subject  :%s\n",subject);
printf("Text     :%s\n",txt);
}
```

Enqueue and Dequeue of Object Type Messages Using OCI

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

struct message
{
  OCIString   *subject;
  OCIString   *data;
};
typedef struct message message;

struct null_message
{
  OCIInd    null_adt;
  OCIInd    null_subject;
```

```
  OCIInd    null_data;
};
typedef struct null_message null_message;

int main()
{
  OCIEnv       *envhp;
  OCIServer    *srvhp;
  OCIError     *errhp;
  OCISvcCtx    *svchp;
  dvoid        *tmp;
  OCIType      *mesg_tdo = (OCIType *) 0;
  message      msg;
  null_message nmsg;
  message      *mesg     = &msg;
  null_message *nmesg    = &nmsg;
  message      *deqmesg  = (message *)0;
  null_message *ndeqmesg = (null_message *)0;

  OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,  (dvoid * (*)()) 0,
              (dvoid * (*)()) 0,  (void (*)()) 0 );

  OCIHandleAlloc((dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
               52, (dvoid **) &tmp);

  OCIEnvInit(&envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp  );

  OCIHandleAlloc((dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
               52, (dvoid **) &tmp);
  OCIHandleAlloc((dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
               52, (dvoid **) &tmp);

  OCIServerAttach(srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

  OCIHandleAlloc((dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
               52, (dvoid **) &tmp);

  OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
      (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

  OCILogon(envhp, errhp, &svchp, "AQ", strlen("AQ"), "AQ", strlen("AQ"), 0, 0);

  /* Obtain TDO of message_typ */
  OCITypeByName(envhp, errhp, svchp, (CONST text *)"AQ", strlen("AQ"),
              (CONST text *)"MESSAGE_TYP", strlen("MESSAGE_TYP"),
```

```
                  (text *)0, 0, OCI_DURATION_SESSION, OCI_TYPEGET_ALL, &mesg_tdo);

  /* Prepare the message payload */
  mesg->subject = (OCIString *)0;
  mesg->data = (OCIString *)0;
  OCIStringAssignText(envhp, errhp,
                      (CONST text *)"NORMAL MESSAGE", strlen("NORMAL MESSAGE"),
                       &mesg->subject);


  OCIStringAssignText(envhp, errhp,
                      (CONST text *)"OCI ENQUEUE", strlen("OCI ENQUEUE"),
                      &mesg->data);
  nmesg->null_adt = nmesg->null_subject = nmesg->null_data = OCI_IND_NOTNULL;

  /* Enqueue into the msg_queue */
  OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue", 0, 0,
           mesg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, 0, 0);
  OCITransCommit(svchp, errhp, (ub4) 0);

  /* Dequeue from the msg_queue */
  OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue", 0, 0,
           mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0);
  printf("Subject: %s\n", OCIStringPtr(envhp, deqmesg->subject));
  printf("Text: %s\n", OCIStringPtr(envhp, deqmesg->data));
  OCITransCommit(svchp, errhp, (ub4) 0);
}
```

Enqueue and Dequeue of RAW Type Messages Using PL/SQL

```
DECLARE
   enqueue_options     dbms_aq.enqueue_options_t;
   message_properties  dbms_aq.message_properties_t;
   message_handle      RAW(16);
   message             RAW(4096);

BEGIN
   message :=  HEXTORAW(RPAD(’FF’,4095,’FF’));
   DBMS_AQ.ENQUEUE(queue_name => ’raw_msg_queue’,
           enqueue_options    => enqueue_options,
           message_properties => message_properties,
                     payload  => message,
                      msgid   => message_handle);
```

```
   COMMIT;
END;

/* Dequeue from raw_msg_queue: */
/* Dequeue from raw_msg_queue: */
DECLARE
   dequeue_options     DBMS_AQ.dequeue_options_t;
   message_properties  DBMS_AQ.message_properties_t;
   message_handle      RAW(16);
   message             RAW(4096);

BEGIN
   DBMS_AQ.DEQUEUE(queue_name => 'raw_msg_queue',
           dequeue_options    => dequeue_options,
           message_properties => message_properties,
           payload            => message,
           msgid              => message_handle);

   COMMIT;
END;
```

Enqueue and Dequeue of RAW Type Messages Using Pro*C/C++

Note: You may need to set up data structures similar to the following for certain examples to work:

```
$ cat >> message.typ
case=lower
type aq.message_typ
$
$ ott userid=aq/aq intyp=message.typ outtyp=message_o.typ \
code=c hfile=demo.h
$
$ proc intyp=message_o.typ iname=<program name> \
config=<config file> SQLCHECK=SEMANTICS userid=aq/aq
```

```
#include  <stdio.h>
#include  <string.h>
#include  <sqlca.h>
#include  <sql2oci.h>

void sql_error(msg)
```

```
char *msg;
{
EXEC SQL WHENEVER SQLERROR CONTINUE;
printf("%s\n", msg);
printf("\n% .800s \n", sqlca.sqlerrm.sqlerrmc);
EXEC SQL ROLLBACK WORK RELEASE;
exit(1);
}

main()
{
OCIEnv          *oeh;   /* OCI Env handle */
OCIError        *err;   /* OCI Err handle */
OCIRaw          *message= (OCIRaw*)0;   /* payload */
ub1             message_txt[100];   /* data for payload */
char            user[60]="aq/AQ";   /* user logon password */
int             status;   /* returns status of the OCI call */

/* Enqueue and dequeue to a RAW queue */

/* Connect to database: */
EXEC SQL CONNECT :user;

/* On an oracle error print the error number: */
EXEC SQL WHENEVER SQLERROR DO sql_error("Oracle Error :");

/* Get the OCI Env handle: */
if (SQLEnvGet(SQL_SINGLE_RCTX, &oeh) != OCI_SUCCESS)
{
printf(" error in SQLEnvGet \n");
exit(1);
}
/* Get the OCI Error handle: */
if (status = OCIHandleAlloc((dvoid *)oeh, (dvoid **)&err,
(ub4)OCI_HTYPE_ERROR, (ub4)0, (dvoid **)0))
{
printf(" error in OCIHandleAlloc %d \n", status);
exit(1);
}

/* Enqueue */
/* The bytes to be put into the raw payload:*/
strcpy(message_txt, "Enqueue to a Raw payload queue ");

/* Assign bytes to the OCIRaw pointer :
```

```
Memory needs to be allocated explicitly to OCIRaw*: */
if (status=OCIRawAssignBytes(oeh, err, message_txt, 100,
 &message))
{
printf(" error in  OCIRawAssignBytes  %d \n", status);
exit(1);
}

/*  Embedded PLSQL call to the AQ enqueue procedure : */
EXEC SQL EXECUTE
DECLARE
message_properties   dbms_aq.message_properties_t;
enqueue_options      dbms_aq.enqueue_options_t;
msgid                RAW(16);
BEGIN
/* Bind the host variable message to the raw payload: */
dbms_aq.enqueue(queue_name => 'raw_msg_queue',
message_properties => message_properties,
enqueue_options => enqueue_options,
payload => :message,
msgid => msgid);
END;
END-EXEC;
/* Commit work: */
EXEC SQL COMMIT;

/* Dequeue */
/*  Embedded PLSQL call to the AQ dequeue procedure :*/
EXEC SQL EXECUTE
DECLARE
message_properties  dbms_aq.message_properties_t;
dequeue_options     dbms_aq.dequeue_options_t;
msgid               RAW(16);
BEGIN
/* Return the raw payload into the host variable 'message':*/
dbms_aq.dequeue(queue_name => 'raw_msg_queue',
message_properties => message_properties,
dequeue_options => dequeue_options,
payload => :message,
msgid => msgid);
END;
END-EXEC;
/* Commit work: */
EXEC SQL COMMIT;
}
```

Enqueue and Dequeue of RAW Type Messages Using OCI

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

int main()
{
  OCIEnv       *envhp;
  OCIServer    *srvhp;
  OCIError     *errhp;
  OCISvcCtx    *svchp;
  dvoid        *tmp;
  OCIType      *mesg_tdo = (OCIType *) 0;
  char         msg_text[100];
  OCIRaw       *mesg = (OCIRaw *)0;
  OCIRaw       *deqmesg = (OCIRaw *)0;
  OCIInd       ind = 0;
  dvoid        *indptr = (dvoid *)&ind;
  int          i;

  OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,  (dvoid * (*)()) 0,
               (dvoid * (*)()) 0,  (void (*)()) 0 );

  OCIHandleAlloc((dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
                52, (dvoid **) &tmp);

  OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp  );

  OCIHandleAlloc((dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
                52, (dvoid **) &tmp);
  OCIHandleAlloc((dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                52, (dvoid **) &tmp);

  OCIServerAttach(srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

  OCIHandleAlloc((dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                52, (dvoid **) &tmp);

  OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
           (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);
```

```
  OCILogon(envhp, errhp, &svchp, "AQ", strlen("AQ"), "AQ", strlen("AQ"), 0, 0);

  /* Obtain the TDO of the RAW data type */
  OCITypeByName(envhp, errhp, svchp, (CONST text *)"AQADM", strlen("AQADM"),
                (CONST text *)"RAW", strlen("RAW"),
                (text *)0, 0, OCI_DURATION_SESSION, OCI_TYPEGET_ALL, &mesg_tdo);

  /* Prepare the message payload */
  strcpy(msg_text, "Enqueue to a RAW queue");
  OCIRawAssignBytes(envhp, errhp, msg_text, strlen(msg_text), &mesg);

  /* Enqueue the message into raw_msg_queue */
  OCIAQEnq(svchp, errhp, (CONST text *)"raw_msg_queue", 0, 0,
          mesg_tdo, (dvoid **)&mesg, (dvoid **)&indptr, 0, 0);
  OCITransCommit(svchp, errhp, (ub4) 0);

  /* Dequeue the same message into C variable deqmesg */
  OCIAQDeq(svchp, errhp, (CONST text *)"raw_msg_queue", 0, 0,
           mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&indptr, 0, 0);
  for (i = 0; i < OCIRawSize(envhp, deqmesg); i++)
    printf("%c", *(OCIRawPtr(envhp, deqmesg) + i));
  OCITransCommit(svchp, errhp, (ub4) 0);
}
```

Enqueue and Dequeue of RAW Type Messages Using Java

Setup for AQ Examples

```
/* Create an AQ User: */
CONNECT system/manager

DROP USER aqjava CASCADE;
GRANT CONNECT, RESOURCE, AQ_ADMINISTRATOR_ROLE TO aqjava IDENTIFIED BY aqjava;
GRANT EXECUTE ON DBMS_AQADM TO aqjava;
GRANT EXECUTE ON DBMS_AQ TO aqjava;
CONNECT aqjava/aqjava

/* Set up main class from which we will call subsequent examples and handle
   exceptions: */
import java.sql.*;
import oracle.AQ.*;
```

```
public class test_aqjava
{
   public static void main(String args[])
   {
      AQSession  aq_sess = null;

      try
      {
         aq_sess = createSession(args);

        /* now run the test: */
        runTest(aq_sess);
      }
      catch (Exception ex)
      {
         System.out.println("Exception-1: " + ex);
         ex.printStackTrace();
      }
   }
}

/* Create an AQ Session for the 'aqjava' user as shown in the
   AQDriverManager section above: */
 public static AQSession createSession(String args[])
   {
      Connection db_conn;
      AQSession  aq_sess = null;

      try
      {

         Class.forName("oracle.jdbc.driver.OracleDriver");
         /* your actual hostname, port number, and SID will
         vary from what follows. Here we use 'dlsun736,' '5521,'
         and 'test,' respectively: */

         db_conn =
                  DriverManager.getConnection(
                  "jdbc:oracle:thin:@dlsun736:5521:test",
                  "aqjava", "aqjava");

         System.out.println("JDBC Connection opened ");
         db_conn.setAutoCommit(false);

         /* Load the Oracle8i AQ driver: */
```

```
          Class.forName("oracle.AQ.AQOracleDriver");

          /* Create an AQ Session: */
          aq_sess = AQDriverManager.createAQSession(db_conn);
          System.out.println("Successfully created AQSession ");
       }
       catch (Exception ex)
       {
          System.out.println("Exception: " + ex);
          ex.printStackTrace();
       }
       return aq_sess;
    }

/* Create a queue table and a queue  for the 'aqjava' use: /*
public static void runTest(AQSession aq_sess) throws AQException
{
    AQQueueTableProperty     qtable_prop;
    AQQueueProperty          queue_prop;
    AQQueueTable             q_table;
    AQQueue                  queue;

    /* Create a AQQueueTableProperty object (payload type - RAW): */
    qtable_prop = new AQQueueTableProperty("RAW");

    /* Create a queue table called aq_table1 in aqjava schema: */
    q_table = aq_sess.createQueueTable ("aqjava", "aq_table1", qtable_prop);
    System.out.println("Successfully created aq_table1 in aqjava schema");

    /* Create a new AQQueueProperty object: */
    queue_prop = new AQQueueProperty();

    /* Create a queue called aq_queue1 in aq_table1: */
    queue = aq_sess.createQueue (q_table, "aq_queue1", queue_prop);
    System.out.println("Successfully created aq_queue1 in aq_table1");
}

/* Get a handle to an existing queue table and queue: */
public static void runTest(AQSession aq_sess) throws AQException
{
    AQQueueTable            q_table;
    AQQueue                 queue;

    /* Get a handle to queue table - aq_table1 in aqjava schema: */
    q_table = aq_sess.getQueueTable ("aqjava", "aq_table1");
```

```
        System.out.println("Successful getQueueTable");

        /* Get a handle to a queue - aq_queue1 in aqjava schema: */
        queue = aq_sess.getQueue ("aqjava", "aq_queue1");
        System.out.println("Successful getQueue");
    }


    public static void runTest(AQSession aq_sess) throws AQException
    {
         AQQueueTable             q_table;
         AQQueue                  queue;
         AQMessage                message;
         AQRawPayload             raw_payload;
         AQEnqueueOption          enq_option;
         String                   test_data = "new message";
         byte[]                   b_array;

         /* Get a handle to queue table - aq_table4 in aqjava schema: */
         q_table = aq_sess.getQueueTable ("aqjava", "aq_table4");
         System.out.println("Successful getQueueTable");

         /* Get a handle to a queue - aq_queue4 in aquser schema: */
         queue = aq_sess.getQueue ("aqjava", "aq_queue4");
         System.out.println("Successful getQueue");

         /* Create a message to contain raw payload: */
         message = queue.createMessage();

         /* Get handle to the AQRawPayload object and populate it with raw data: */
         b_array = test_data.getBytes();

         raw_payload = message.getRawPayload();

         raw_payload.setStream(b_array, b_array.length);

         /* Create a AQEnqueueOption object with default options: */
         enq_option = new AQEnqueueOption();
         /* Enqueue the message: */
         queue.enqueue(enq_option, message);

    }
```

Dequeue of Messages Using Java

```
   public static void runTest(AQSession aq_sess) throws AQException
{
     AQQueueTable             q_table;
     AQQueue                  queue;
     AQMessage                message;
     AQRawPayload             raw_payload;
     AQEnqueueOption          enq_option;
     String                   test_data = "new message";
     AQDequeueOption          deq_option;
     byte[]                   b_array;

     /* Get a handle to queue table - aq_table4 in aqjava schema: */
     q_table = aq_sess.getQueueTable ("aqjava", "aq_table4");
     System.out.println("Successful getQueueTable");

     /* Get a handle to a queue - aq_queue4 in aquser schema: */
     queue = aq_sess.getQueue ("aqjava", "aq_queue4");
     System.out.println("Successful getQueue");

     /* Create a message to contain raw payload: */
     message = queue.createMessage();

     /* Get handle to the AQRawPayload object and populate it with raw data: */
     b_array = test_data.getBytes();

     raw_payload = message.getRawPayload();

     raw_payload.setStream(b_array, b_array.length);

     /* Create a AQEnqueueOption object with default options: */
     enq_option = new AQEnqueueOption();

     /* Enqueue the message: */
     queue.enqueue(enq_option, message);
     System.out.println("Successful enqueue");

     /* Create a AQDequeueOption object with default options: */
     deq_option = new AQDequeueOption();

     /* Dequeue a message: */
     message = queue.dequeue(deq_option);
     System.out.println("Successful dequeue");
```

```
        /* Retrieve raw data from the message: */
        raw_payload = message.getRawPayload();

        b_array = raw_payload.getBytes();
    }
```

Dequeue of Messages in Browse Mode Using Java

```
  public static void runTest(AQSession aq_sess) throws AQException
{
      AQQueueTable           q_table;
      AQQueueTable           q_table;
      AQQueue                queue;
      AQMessage              message;
      AQRawPayload           raw_payload;
      AQEnqueueOption        enq_option;
      String                 test_data = "new message";
      AQDequeueOption        deq_option;
      byte[]                  b_array;

      /* Get a handle to queue table - aq_table4 in aqjava schema: */
      q_table = aq_sess.getQueueTable ("aqjava", "aq_table4");
      System.out.println("Successful getQueueTable");

      /* Get a handle to a queue - aq_queue4 in aquser schema: */
      queue = aq_sess.getQueue ("aqjava", "aq_queue4");
      System.out.println("Successful getQueue");

      /* Create a message to contain raw payload: */
      message = queue.createMessage();

      /* Get handle to the AQRawPayload object and populate it with raw data: */
      b_array = test_data.getBytes();

      raw_payload = message.getRawPayload();

      raw_payload.setStream(b_array, b_array.length);

      /* Create a AQEnqueueOption object with default options: */
      enq_option = new AQEnqueueOption();

      /* Enqueue the message: */
      queue.enqueue(enq_option, message);
      System.out.println("Successful enqueue");
```

```
        /* Create a AQDequeueOption object with default options: */
        deq_option = new AQDequeueOption();

        /* Set dequeue mode to BROWSE: */
        deq_option.setDequeueMode(AQDequeueOption.DEQUEUE_BROWSE);

        /* Set wait time to 10 seconds: */
        deq_option.setWaitTime(10);

        /* Dequeue a message: */
        message = queue.dequeue(deq_option);

        /* Retrieve raw data from the message: */
        raw_payload = message.getRawPayload();
        b_array = raw_payload.getBytes();

        String ret_value = new String(b_array);
        System.out.println("Dequeued message: " + ret_value);
  }
```

Enqueue and Dequeue of Messages by Priority Using PL/SQL

When two messages are enqued with the same priority, the message which was enqued earlier will be dequeued first. However, if two messages are of different priorities, the message with the lower value (higher priority) will be dequeued first.

```
/* Enqueue two messages with priority 30 and 5: */
DECLARE
   enqueue_options     dbms_aq.enqueue_options_t;
   message_properties  dbms_aq.message_properties_t;
   message_handle      RAW(16);
   message             aq.message_typ;

BEGIN
   message := message_typ('PRIORITY MESSAGE',
   'enqued at priority 30.');

   message_properties.priority := 30;

   DBMS_AQ.ENQUEUE(queue_name => 'priority_msg_queue',
           enqueue_options    => enqueue_options,
           message_properties => message_properties,
```

```
                payload            => message,
                msgid              => message_handle);

   message := message_typ('PRIORITY MESSAGE',
   'Enqueued at priority 5.');

   message_properties.priority := 5;

   DBMS_AQ.ENQUEUE(queue_name => 'priority_msg_queue',
                enqueue_options    => enqueue_options,
                message_properties => message_properties,
                payload            => message,
                msgid              => message_handle);
END;

/* Dequeue from priority queue: */
DECLARE
   dequeue_options     DBMS_AQ.dequeue_options_t;
   message_properties  DBMS_AQ.message_properties_t;
   message_handle      RAW(16);
   message             aq.message_typ;

BEGIN
   DBMS_AQ.DEQUEUE(queue_name  => 'priority_msg_queue',
              dequeue_options      => dequeue_options,
              message_properties   => message_properties,
              payload              => message,
              msgid                => message_handle);

   DBMS_OUTPUT.PUT_LINE ('Message: ' || message.subject ||
   ' ... ' || message.text );

   COMMIT;

   DBMS_AQ.DEQUEUE(queue_name => 'priority_msg_queue',
              dequeue_options     => dequeue_options,
              message_properties  => message_properties,
              payload             => message,
              msgid               => message_handle);

   DBMS_OUTPUT.PUT_LINE ('Message: ' || message.subject ||
   ' ... ' || message.text );
   COMMIT;
END;
```

```
/* On return, the second message with priority set to 5 will be retrieved before
the message with priority set to 30 since priority takes precedence over enqueue
time. */
```

Enqueue of Messages with Priority Using Java

```
public static void runTest(AQSession aq_sess) throws AQException
{
     AQQueueTable             q_table;
     AQQueue                  queue;
     AQMessage                message;
     AQMessageProperty        m_property;
     AQRawPayload             raw_payload;
     AQEnqueueOption          enq_option;
     String                   test_data;
     byte[]                   b_array;

     /* Get a handle to queue table - aq_table4 in aqjava schema: */
     qtable = aq_sess.getQueueTable ("aqjava", "aq_table4");
     System.out.println("Successful getQueueTable");

     /* Get a handle to a queue - aq_queue4 in aqjava schema: */
     queue = aq_sess.getQueue ("aqjava", "aq_queue4");
     System.out.println("Successful getQueue");

     /* Enqueue 5 messages with priorities with different priorities: */
     for (int i = 0; i < 5; i++ )
       {
          /* Create a message to contain raw payload: */
          message = queue.createMessage();

          test_data = "Small_message_" + (i+1);       /* some test data */

          /* Get a handle to the AQRawPayload object and
             populate it with raw data: */
          b_array = test_data.getBytes();

          raw_payload = message.getRawPayload();

          raw_payload.setStream(b_array, b_array.length);

          /* Set message priority: */
         m_property = message.getMessageProperty();
```

```
         if( i < 2)
            m_property.setPriority(2);
          else
             m_property.setPriority(3);

          /* Create a AQEnqueueOption object with default options: */
          enq_option = new AQEnqueueOption();

          /* Enqueue the message: */
         queue.enqueue(enq_option, message);
         System.out.println("Successful enqueue");
     }
}
```

Dequeue of Messages after Preview by Criterion Using PL/SQL

An application can preview messages in browse mode or locked mode without deleting the message. The message of interest can then be removed from the queue.

```
/* Enqueue 6 messages to msg_queue
— GREEN, GREEN, YELLOW, VIOLET, BLUE, RED */

DECLARE
   enqueue_options     DBMS_AQ.enqueue_options_t;
   message_properties  DBMS_AQ.message_properties_t;
   message_handle      RAW(16);
   message             aq.message_typ;

BEGIN
   message := message_typ('GREEN',
   'GREEN enqueued to msg_queue first.');

   DBMS_AQ.ENQUEUE(queue_name => 'msg_queue',
         enqueue_options      => enqueue_options,
         message_properties   => message_properties,
         payload              => message,
         msgid                => message_handle);

   message := message_typ('GREEN',
   'GREEN also enqueued to msg_queue second.');

   DBMS_AQ.ENQUEUE(queue_name => 'msg_queue',
         enqueue_options      => enqueue_options,
         message_properties   => message_properties,
```

```
            payload              => message,
            msgid                => message_handle);

      message := message_typ('YELLOW',
      'YELLOW enqueued to msg_queue third.');

      DBMS_AQ.ENQUEUE(queue_name => 'msg_queue',
            enqueue_options      => enqueue_options,
            message_properties   => message_properties,
            payload              => message,
            msgid                => message_handle);

      DBMS_OUTPUT.PUT_LINE ('Message handle: ' || message_handle);

      message := message_typ('VIOLET',
      'VIOLET enqueued to msg_queue fourth.');

      DBMS_AQ.ENQUEUE(queue_name => 'msg_queue',
            enqueue_options      => enqueue_options,
            message_properties   => message_properties,
            payload              => message,
            msgid                => message_handle);

      message := message_typ('BLUE',
      'BLUE enqueued to msg_queue fifth.');

      DBMS_AQ.ENQUEUE(queue_name => 'msg_queue',
            enqueue_options      => enqueue_options,
            message_properties   => message_properties,
            payload              => message,
            msgid                => message_handle);

      message := message_typ('RED',
      'RED enqueued to msg_queue sixth.');

      DBMS_AQ.ENQUEUE(queue_name => 'msg_queue',
            enqueue_options      => enqueue_options,
            message_properties   => message_properties,
            payload              => message,
            msgid                => message_handle);

      COMMIT;
   END;

   /* Dequeue in BROWSE mode until RED is found,
```

```
and remove RED from queue: */
DECLARE
   dequeue_options     DBMS_AQ.dequeue_options_t;
   message_properties  DBMS_AQ.message_properties_t;
   message_handle      RAW(16);
   message             aq.message_typ;

BEGIN
   dequeue_options.dequeue_mode := DBMS_AQ.BROWSE;

   LOOP
      DBMS_AQ.DEQUEUE(queue_name          => ’msg_queue’,
                      dequeue_options    => dequeue_options,
                      message_properties => message_properties,
                      payload            => message,
                      msgid              => message_handle);

      DBMS_OUTPUT.PUT_LINE (’Message: ’ || message.subject ||
                                   ’ ... ’ || message.text );

      EXIT WHEN message.subject = ’RED’;

   END LOOP;

   dequeue_options.dequeue_mode := DBMS_AQ.REMOVE;
   dequeue_options.msgid        := message_handle;

   DBMS_AQ.DEQUEUE(queue_name => ’msg_queue’,
           dequeue_options    => dequeue_options,
           message_properties => message_properties,
           payload            => message,
           msgid              => message_handle);

   DBMS_OUTPUT.PUT_LINE (’Message: ’ || message.subject ||
   ’ ... ’ || message.text );

   COMMIT;
END;

/* Dequeue in LOCKED mode until BLUE is found,
and remove BLUE from queue: */
DECLARE
dequeue_options     dbms_aq.dequeue_options_t;
message_properties  dbms_aq.message_properties_t;
message_handle      RAW(16);
```

```
message             aq.message_typ;

BEGIN
dequeue_options.dequeue_mode := dbms_aq.LOCKED;

      LOOP

dbms_aq.dequeue(queue_name => ’msg_queue’,
                 dequeue_options    => dequeue_options,
                 message_properties => message_properties,
                 payload            => message,
                 msgid              => message_handle);

dbms_output.put_line (’Message: ’ || message.subject ||
          ’ ... ’ || message.text );

EXIT WHEN message.subject = ’BLUE’;
      END LOOP;

dequeue_options.dequeue_mode := dbms_aq.REMOVE;
dequeue_options.msgid        := message_handle;

dbms_aq.dequeue(queue_name => ’msg_queue’,
dequeue_options      => dequeue_options,
message_properties   => message_properties,
payload              => message,
msgid => message_handle);

DBMS_OUTPUT.PUT_LINE (’Message: ’ || message.subject ||
’ ... ’ || message.text );

      COMMIT;
END;
```

Enqueue and Dequeue of Messages with Time Delay and Expiration Using PL/SQL

Note: Expiration is calculated from the earliest dequeue time. So, if an application wants a message to be dequeued no earlier than a week from now, but no later than 3 weeks from now, this requires setting the expiration time for 2 weeks. This scenario is described in the following code segment.

```
/* Enqueue message for delayed availability: */
DECLARE
enqueue_options     dbms_aq.enqueue_options_t;
message_properties  dbms_aq.message_properties_t;
message_handle      RAW(16);
message             aq.Message_typ;

BEGIN
message := Message_typ('DELAYED',
'This message is delayed one week.');
message_properties.delay := 7*24*60*60;
message_properties.expiration := 2*7*24*60*60;

dbms_aq.enqueue(queue_name => 'msg_queue',
enqueue_options      => enqueue_options,
message_properties   => message_properties,
payload            => message,
msgid              => message_handle);

      COMMIT;
END;
```

Enqueue and Dequeue of Messages by Correlation and Message ID Using Pro*C/C++

Note: You may need to set up data structures similar to the following for certain examples to work:

```
$ cat >> message.typ
case=lower
type aq.message_typ
$
$ ott userid=aq/aq intyp=message.typ outtyp=message_o.typ \
code=c hfile=demo.h
$
$ proc intyp=message_o.typ iname=<program name> \
config=<config file> SQLCHECK=SEMANTICS userid=aq/aq
```

```
#include  <stdio.h>
#include  <string.h>
#include  <sqlca.h>
#include  <sql2oci.h>
/* The header file generated by processing
```

```
object type 'aq.Message_typ': */
#include  "pceg.h"

void sql_error(msg)
char *msg;
{
EXEC SQL WHENEVER SQLERROR CONTINUE;
printf("%s\n", msg);
printf("\n% .800s \n", sqlca.sqlerrm.sqlerrmc);
EXEC SQL ROLLBACK WORK RELEASE;
exit(1);
}

main()
{
OCIEnv            *oeh;  /* OCI Env Handle */
OCIError          *err;  /* OCI Error Handle */
Message_typ      *message = (Message_typ*)0; /* queue  payload */
OCIRaw            *msgid = (OCIRaw*)0;  /* message id */
ub1               msgmem[16]="";  /* memory for msgid */
char              user[60]="aq/AQ";  /* user login password */
char              subject[30];  /* components of */
char              txt[80];  /* Message_typ */
char              correlation1[30];  /* message correlation  */
char              correlation2[30];
int               status;  /* code returned by the OCI calls */

/* Dequeue by correlation and msgid */

/* Connect to the database: */
EXEC SQL CONNECT :user;
EXEC SQL WHENEVER SQLERROR DO sql_error("Oracle Error :");

/* Allocate space in the object cache for the host variable: */
EXEC SQL ALLOCATE :message;

/* Get the OCI Env handle: */
if (SQLEnvGet(SQL_SINGLE_RCTX, &oeh) != OCI_SUCCESS)
{
 printf(" error in SQLEnvGet \n");
 exit(1);
}
/* Get the OCI Error handle: */
if (status = OCIHandleAlloc((dvoid *)oeh, (dvoid **)&err,
(ub4)OCI_HTYPE_ERROR, (ub4)0, (dvoid **)0))
```

```
{
printf(" error in OCIHandleAlloc %d \n", status);
exit(1);
}


/* Assign memory for msgid:
Memory needs to be allocated explicitly to OCIRaw*: */
if (status=OCIRawAssignBytes(oeh, err, msgmem, 16, &msgid))
{
printf(" error in  OCIRawAssignBytes  %d \n", status);
exit(1);
}

/* First enqueue */

strcpy(correlation1, "1st message");
strcpy(subject, "NORMAL ENQUEUE1");
strcpy(txt, "The Enqueue was done through PLSQL embedded in PROC");

/* Initialize the components of message: */
EXEC SQL OJECT SET subject, text OF :message TO :subject, :txt;

/* Embedded PLSQL call to the AQ enqueue procedure: */
EXEC SQL EXECUTE
DECLARE
message_properties   dbms_aq.message_properties_t;
enqueue_options      dbms_aq.enqueue_options_t;
BEGIN
/* Bind the host variable 'correlation1': to message correlation*/
message_properties.correlation := :correlation1;

/* Bind the host variable 'message' to payload and
 return message id into host variable 'msgid': */
dbms_aq.enqueue(queue_name => 'msg_queue',
message_properties => message_properties,
enqueue_options => enqueue_options,
payload => :message,
msgid => :msgid);
END;
END-EXEC;
/* Commit work: */
EXEC SQL COMMIT;

printf("Enqueued Message \n");
```

```
printf("Subject  :%s\n",subject);
printf("Text     :%s\n",txt);

/* Second enqueue */

strcpy(correlation2, "2nd message");
strcpy(subject, "NORMAL ENQUEUE2");
strcpy(txt, "The Enqueue was done through PLSQL embedded in PROC");

/* Initialize the components of message: */
EXEC SQL OBJECT SET subject, text OF :messsage TO :subject,:txt;

/* Embedded PLSQL call to the AQ enqueue procedure: */
EXEC SQL EXECUTE
DECLARE
message_properties   dbms_aq.message_properties_t;
enqueue_options      dbms_aq.enqueue_options_t;
msgid                RAW(16);
BEGIN
/* Bind the host variable 'correlation2':  to message correlaiton */
message_properties.correlation := :correlation2;

/* Bind the host variable 'message': to payload */
dbms_aq.enqueue(queue_name => 'msg_queue',
message_properties => message_properties,
enqueue_options => enqueue_options,
payload => :message,
msgid => msgid);
END;
END-EXEC;
/* Commit work: */
EXEC SQL COMMIT;
printf("Enqueued Message \n");
printf("Subject  :%s\n",subject);
printf("Text     :%s\n",txt);

/* First dequeue - by  correlation */

EXEC SQL EXECUTE
DECLARE
message_properties  dbms_aq.message_properties_t;
dequeue_options     dbms_aq.dequeue_options_t;
msgid               RAW(16);
BEGIN
/* Dequeue by  correlation in host variable 'correlation2': */
```

```
dequeue_options.correlation := :correlation2;

/* Return the payload into host variable 'message': */
dbms_aq.dequeue(queue_name => 'msg_queue',
message_properties => message_properties,
dequeue_options => dequeue_options,
payload => :message,
msgid => msgid);
END;
END-EXEC;
/* Commit work : */
EXEC SQL COMMIT;

/* Extract the values of the components of message: */
EXEC SQL OBJECT GET subject, text FROM :message INTO :subject,:txt;

printf("Dequeued Message \n");
printf("Subject  :%s\n",subject);
printf("Text     :%s\n",txt);

/* SECOND DEQUEUE - by MSGID  */

EXEC SQL EXECUTE
DECLARE
message_properties  dbms_aq.message_properties_t;
dequeue_options     dbms_aq.dequeue_options_t;
msgid               RAW(16);
BEGIN
/* Dequeue by msgid in host variable 'msgid': */
dequeue_options.msgid := :msgid;

/* Return the payload into host variable 'message':  */
dbms_aq.dequeue(queue_name => 'msg_queue',
message_properties => message_properties,
dequeue_options => dequeue_options,
payload => :message,
msgid => msgid);
END;
END-EXEC;
/* Commit work: */
EXEC SQL COMMIT;
}
```

Enqueue and Dequeue of Messages by Correlation and Message ID Using OCI

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

struct message
{
  OCIString   *subject;
  OCIString   *data;
};
typedef struct message message;

struct null_message
{
  OCIInd    null_adt;
  OCIInd    null_subject;
  OCIInd    null_data;
};
typedef struct null_message null_message;

int main()
{
  OCIEnv       *envhp;
  OCIServer    *srvhp;
  OCIError     *errhp;
  OCISvcCtx    *svchp;
  dvoid        *tmp;
  OCIType      *mesg_tdo = (OCIType *) 0;
  message      msg;
  null_message nmsg;
  message      *mesg     = &msg;
  null_message *nmesg    = &nmsg;
  message      *deqmesg  = (message *)0;
  null_message *ndeqmesg = (null_message *)0;

  OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,  (dvoid * (*)()) 0,
                (dvoid * (*)()) 0,  (void (*)()) 0 );

  OCIHandleAlloc((dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
                 52, (dvoid **) &tmp);

  OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp  );
```

```
  OCIHandleAlloc((dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
                 52, (dvoid **) &tmp);
  OCIHandleAlloc((dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                 52, (dvoid **) &tmp);

  OCIServerAttach(srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

  OCIHandleAlloc((dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                 52, (dvoid **) &tmp);

  OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
             (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

  OCILogon(envhp, errhp, &svchp, "AQ", strlen("AQ"), "AQ", strlen("AQ"), 0, 0);

  /* Obtain TDO of message_typ */
  OCITypeByName(envhp, errhp, svchp, (CONST text *)"AQ", strlen("AQ"),
                (CONST text *)"MESSAGE_TYP", strlen("MESSAGE_TYP"),
                (text *)0, 0, OCI_DURATION_SESSION, OCI_TYPEGET_ALL, &mesg_tdo);

  /* Prepare the message payload */
  mesg->subject = (OCIString *)0;
  mesg->data = (OCIString *)0;
  OCIStringAssignText(envhp, errhp,
                      (CONST text *)"NORMAL MESSAGE", strlen("NORMAL MESSAGE"),
                      &mesg->subject);
  OCIStringAssignText(envhp, errhp,
                      (CONST text *)"OCI ENQUEUE", strlen("OCI ENQUEUE"),
                      &mesg->data);
  nmesg->null_adt = nmesg->null_subject = nmesg->null_data = OCI_IND_NOTNULL;

  /* Enqueue into the msg_queue */
  OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue", 0, 0,
           mesg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, 0, 0);
  OCITransCommit(svchp, errhp, (ub4) 0);

  /* Dequeue from the msg_queue */
  OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue", 0, 0,
           mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0);
  printf("Subject: %s\n", OCIStringPtr(envhp, deqmesg->subject));
  printf("Text: %s\n", OCIStringPtr(envhp, deqmesg->data));
  OCITransCommit(svchp, errhp, (ub4) 0);
}
```

Enqueue and Dequeue of Messages to/from a Multiconsumer Queue Using PL/SQL

```
/* Create subscriber list: */
DECLARE
   subscriber aq$_agent;

   /* Add subscribers RED and GREEN to the suscriber list: */
BEGIN
   subscriber := aq$_agent('RED', NULL, NULL);
   DBMS_AQADM.ADD_SUBSCRIBER(queue_name => 'msg_queue_multiple',
   subscriber => subscriber);

   subscriber := aq$_agent('GREEN', NULL, NULL);
   DBMS_AQADM.ADD_SUBSCRIBER(queue_name => 'msg_queue_multiple',
   subscriber => subscriber);
END;

DECLARE
   enqueue_options     DBMS_AQ.enqueue_options_t;
   message_properties  DBMS_AQ.message_properties_t;
   recipients          DBMS_AQ.aq$_recipient_list_t;
   message_handle      RAW(16);
   message             aq.message_typ;

   /* Enqueue MESSAGE 1 for subscribers to the queue
   i.e. for RED and GREEN: */
BEGIN
   message := message_typ('MESSAGE 1',
   'This message is queued for queue subscribers.');

   DBMS_AQ.ENQUEUE(queue_name => 'msg_queue_multiple',
   enqueue_options    => enqueue_options,
   message_properties => message_properties,
   payload            => message,
   msgid              => message_handle);

   /* Enqueue MESSAGE 2 for specified recipients i.e. for RED and BLUE.*/
   message := message_typ('MESSAGE 2',
   'This message is queued for two recipients.');
   recipients(1) := aq$_agent('RED', NULL, NULL);
   recipients(2) := aq$_agent('BLUE', NULL, NULL);
   message_properties.recipient_list := recipients;

   DBMS_AQ.ENQUEUE(queue_name => 'msg_queue_multiple',
           enqueue_options    => enqueue_options,
```

```
            message_properties => message_properties,
            payload            => message,
            msgid              => message_handle);

   COMMIT;
END;
```

Note that RED is both a subscriber to the queue, as well as being a specified recipient of MESSAGE 2. By contrast, GREEN is only a subscriber to those messages in the queue (in this case, MESSAGE) for which no recipients have been specified. BLUE, while not a subscriber to the queue, is nevertheless specified to receive MESSAGE 2.

```
/* Dequeue messages from msg_queue_multiple: */
DECLARE
   dequeue_options     DBMS_AQ.dequeue_options_t;
   message_properties  DBMS_AQ.message_properties_t;
   message_handle      RAW(16);
   message             aq.message_typ;
   no_messages         exception;
   pragma exception_init (no_messages, -25228);

BEGIN

   dequeue_options.wait := DBMS_AQ.NO_WAIT;
   BEGIN
   /* Consumer BLUE will get MESSAGE 2: */
   dequeue_options.consumer_name := 'BLUE';
   dequeue_options.navigation := FIRST_MESSAGE;

   LOOP

   DBMS_AQ.DEQUEUE(queue_name   => 'msg_queue_multiple',
             dequeue_options    => dequeue_options,
             message_properties => message_properties,
             payload            => message,
             msgid              => message_handle);

      DBMS_OUTPUT.PUT_LINE ('Message: ' || message.subject ||
            ' ... ' || message.text );
      dequeue_options.navigation := NEXT_MESSAGE;

   END LOOP;
   EXCEPTION
   WHEN no_messages THEN
```

```
   DBMS_OUTPUT.PUT_LINE ('No more messages for BLUE');
   COMMIT;
END;

BEGIN
/* Consumer RED will get MESSAGE 1 and MESSAGE 2: */
   dequeue_options.consumer_name := 'RED';
   dequeue_options.navigation := FIRST_MESSAGE;
   LOOP
      DBMS_AQ.DEQUEUE(queue_name   => 'msg_queue_multiple',
               dequeue_options    => dequeue_options,
               message_properties => message_properties,
               payload            => message,
               msgid              => message_handle);

      DBMS_OUTPUT.PUT_LINE ('Message: ' || message.subject ||
                                   ' ... ' || message.text );
      dequeue_options.navigation := NEXT_MESSAGE;
   END LOOP;
   EXCEPTION
   WHEN no_messages THEN
      DBMS_OUTPUT.PUT_LINE ('No more messages for RED');
   COMMIT;
END;

BEGIN
   /* Consumer GREEN will get MESSAGE 1: */
   dequeue_options.consumer_name := 'GREEN';
   dequeue_options.navigation := FIRST_MESSAGE;
   LOOP
      DBMS_AQ.DEQUEUE(queue_name   => 'msg_queue_multiple',
               dequeue_options    => dequeue_options,
               message_properties => message_properties,
               payload            => message,
               msgid              => message_handle);

      DBMS_OUTPUT.PUT_LINE ('Message: ' || message.subject ||
          ' ... ' || message.text );
      dequeue_options.navigation := NEXT_MESSAGE;
   END LOOP;
   EXCEPTION
   WHEN no_messages THEN
      DBMS_OUTPUT.PUT_LINE ('No more messages for GREEN');
   COMMIT;
END;
```

Enqueue and Dequeue of Messages to/from a Multiconsumer Queue using OCI

Note: You may need to set up the following data structures for certain examples to work:

```
CONNECT aqadm/aqadm
EXECUTE DBMS_AQADM.CREATE_QUEUE_TABLE(
   queue_table        => 'aq.qtable_multi',
   multiple_consumers => true,
   queue_payload_type => 'aq.message_typ');
EXECUTE DBMS_AQADM.START_QUEUE('aq.msg_queue_multiple');
CONNECT aq/aq
```

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

struct message
{
  OCIString   *subject;
  OCIString   *data;
};
typedef struct message message;

struct null_message
{
  OCIInd    null_adt;
  OCIInd    null_subject;
  OCIInd    null_data;
};
typedef struct null_message null_message;

int main()
{
  OCIEnv                *envhp;
  OCIServer             *srvhp;
  OCIError              *errhp;
  OCISvcCtx             *svchp;
  dvoid                 *tmp;
```

```
OCIType              *mesg_tdo = (OCIType *) 0;
message              msg;
null_message         nmsg;
message              *mesg = &msg;
null_message         *nmesg = &nmsg;
message              *deqmesg = (message *)0;
null_message         *ndeqmesg = (null_message *)0;
OCIAQMsgProperties   *msgprop = (OCIAQMsgProperties *)0;
OCIAQAgent           *agents[2];
OCIAQDeqOptions      *deqopt = (OCIAQDeqOptions *)0;
ub4                  wait = OCI_DEQ_NO_WAIT;
ub4                  navigation = OCI_DEQ_FIRST_MSG;

OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,  (dvoid * (*)()) 0,
              (dvoid * (*)()) 0,  (void (*)()) 0 );



OCIHandleAlloc((dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
               52, (dvoid **) &tmp);

OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp  );

OCIHandleAlloc((dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
               52, (dvoid **) &tmp);
OCIHandleAlloc((dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
               52, (dvoid **) &tmp);

OCIServerAttach(srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

OCIHandleAlloc((dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
               52, (dvoid **) &tmp);

OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
           (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

OCILogon(envhp, errhp, &svchp, "AQ", strlen("AQ"), "AQ", strlen("AQ"), 0, 0);

/* Obtain TDO of message_typ */
OCITypeByName(envhp, errhp, svchp, (CONST text *)"AQ", strlen("AQ"),
             (CONST text *)"MESSAGE_TYP", strlen("MESSAGE_TYP"),
             (text *)0, 0, OCI_DURATION_SESSION, OCI_TYPEGET_ALL, &mesg_tdo);

/* Prepare the message payload */
mesg->subject = (OCIString *)0;
```

```
 mesg->data = (OCIString *)0;
 OCIStringAssignText(envhp, errhp,
                     (CONST text *)"MESSAGE 1", strlen("MESSAGE 1"),
                     &mesg->subject);
 OCIStringAssignText(envhp, errhp,
                    (CONST text *)"mesg for queue subscribers",
                    strlen("mesg for queue subscribers"), &mesg->data);
 nmesg->null_adt = nmesg->null_subject = nmesg->null_data = OCI_IND_NOTNULL;

 /* Enqueue MESSAGE 1 for subscribers to the queue i.e. for RED and GREEN */
 OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue_multiple", 0, 0,
          mesg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, 0, 0);

 /* Enqueue MESSAGE 2 for specified recipients i.e. for RED and BLUE */
 /* prepare message payload */
OCIStringAssignText(envhp, errhp,
                     (CONST text *)"MESSAGE 2", strlen("MESSAGE 2"),
                     &mesg->subject);
 OCIStringAssignText(envhp, errhp,
      (CONST text *)"mesg for two recipients",
      strlen("mesg for two recipients"), &mesg->data);

 /* Allocate AQ message properties and agent descriptors */
 OCIDescriptorAlloc(envhp, (dvoid **)&msgprop,
                    OCI_DTYPE_AQMSG_PROPERTIES, 0, (dvoid **)0);
 OCIDescriptorAlloc(envhp, (dvoid **)&agents[0],
                    OCI_DTYPE_AQAGENT, 0, (dvoid **)0);
 OCIDescriptorAlloc(envhp, (dvoid **)&agents[1],
                    OCI_DTYPE_AQAGENT, 0, (dvoid **)0);

 /* Prepare the recipient list, RED and BLUE */
 OCIAttrSet(agents[0], OCI_DTYPE_AQAGENT, "RED", strlen("RED"),
            OCI_ATTR_AGENT_NAME, errhp);
 OCIAttrSet(agents[1], OCI_DTYPE_AQAGENT, "BLUE", strlen("BLUE"),
            OCI_ATTR_AGENT_NAME, errhp);
 OCIAttrSet(msgprop, OCI_DTYPE_AQMSG_PROPERTIES, (dvoid *)agents, 2,
            OCI_ATTR_RECIPIENT_LIST, errhp);

 OCIAQEnq(svchp, errhp, (CONST text *)"msg_queue_multiple", 0, msgprop,
          mesg_tdo, (dvoid **)&mesg, (dvoid **)&nmesg, 0, 0);

 OCITransCommit(svchp, errhp, (ub4) 0);

 /* Now dequeue the messages using different consumer names */
 /* Allocate dequeue options descriptor to set the dequeue options */
```

```
OCIDescriptorAlloc(envhp, (dvoid **)&deqopt, OCI_DTYPE_AQDEQ_OPTIONS, 0,
                   (dvoid **)0);

/* Set wait parameter to NO_WAIT so that the dequeue returns immediately */
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)&wait, 0,
           OCI_ATTR_WAIT, errhp);

/* Set navigation to FIRST_MESSAGE so that the dequeue resets the position */
/* after a new consumer_name is set in the dequeue options           */
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)&navigation, 0,
           OCI_ATTR_NAVIGATION, errhp);

/* Dequeue from the msg_queue_multiple as consumer BLUE */
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)"BLUE", strlen("BLUE"),
           OCI_ATTR_CONSUMER_NAME, errhp);

while (OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue_multiple", deqopt, 0,
                mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0)
                == OCI_SUCCESS)
{
  printf("Subject: %s\n", OCIStringPtr(envhp, deqmesg->subject));
  printf("Text: %s\n", OCIStringPtr(envhp, deqmesg->data));
}
OCITransCommit(svchp, errhp, (ub4) 0);

/* Dequeue from the msg_queue_multiple as consumer RED */
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS, (dvoid *)"RED", strlen("RED"),
      OCI_ATTR_CONSUMER_NAME, errhp);
while (OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue_multiple", deqopt, 0,
  mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0)
  == OCI_SUCCESS)
{
  printf("Subject: %s\n", OCIStringPtr(envhp, deqmesg->subject));
  printf("Text: %s\n", OCIStringPtr(envhp, deqmesg->data));
}
OCITransCommit(svchp, errhp, (ub4) 0);

/* Dequeue from the msg_queue_multiple as consumer GREEN */
OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS,(dvoid *)"GREEN",strlen("GREEN"),
      OCI_ATTR_CONSUMER_NAME, errhp);
while (OCIAQDeq(svchp, errhp, (CONST text *)"msg_queue_multiple", deqopt, 0,
  mesg_tdo, (dvoid **)&deqmesg, (dvoid **)&ndeqmesg, 0, 0)
  == OCI_SUCCESS)
{
  printf("Subject: %s\n", OCIStringPtr(envhp, deqmesg->subject));
```

```
    printf("Text: %s\n", OCIStringPtr(envhp, deqmesg->data));
  }
  OCITransCommit(svchp, errhp, (ub4) 0);
}
```

Enqueue and Dequeue of Messages Using Message Grouping Using PL/SQL

```
CONNECT aq/aq

EXECUTE DBMS_AQADM.CREATE_QUEUE_TABLE (
   queue_table          => 'aq.msggroup',
   queue_payload_type   => 'aq.message_typ',
   message_grouping     => DBMS_AQADM.TRANSACTIONAL);

EXECUTE DBMS_AQADM.CREATE_QUEUE(
   queue_name      => 'msggroup_queue',
   queue_table     => 'aq.msggroup');

EXECUTE DBMS_AQADM.START_QUEUE(
   queue_name => 'msggroup_queue');

/* Enqueue three messages in each transaction */
DECLARE
   enqueue_options     DBMS_AQ.enqueue_options_t;
   message_properties  DBMS_AQ.message_properties_t;
   message_handle      RAW(16);
   message             aq.message_typ;

BEGIN

  /* Loop through three times, committing after every iteration */
  FOR txnno in 1..3 LOOP

    /* Loop through three times, enqueuing each iteration */
    FOR mesgno in 1..3 LOOP
      message := message_typ('GROUP#' || txnno,
               'Message#' || mesgno ||  ' in group' || txnno);

      DBMS_AQ.ENQUEUE(queue_name         => 'msggroup_queue',
                  enqueue_options        => enqueue_options,
                  message_properties     => message_properties,
                  payload                => message,
                  msgid                  => message_handle);
    END LOOP;
```

```
    /* Commit the transaction */
    COMMIT;
  END LOOP;
END;

/* Now dequeue the messages as groups */
DECLARE
   dequeue_options     DBMS_AQ.dequeue_options_t;
   message_properties  DBMS_AQ.message_properties_t;
   message_handle      RAW(16);
   message             aq.message_typ;

   no_messages   exception;
   end_of_group  exception;

   PRAGMA EXCEPTION_INIT (no_messages, -25228);
   PRAGMA EXCEPTION_INIT (end_of_group, -25235);

BEGIN
   dequeue_options.wait       := DBMS_AQ.NO_WAIT;
   dequeue_options.navigation := DBMS_AQ.FIRST_MESSAGE;

   LOOP
      BEGIN
      DBMS_AQ.DEQUEUE(queue_name   => 'msggroup_queue',
                dequeue_options    => dequeue_options,
                message_properties => message_properties,
                payload            => message,
                msgid              => message_handle);

     DBMS_OUTPUT.PUT_LINE ('Message: ' || message.subject ||
           ' ... ' || message.text );

     dequeue_options.navigation := DBMS_AQ.NEXT_MESSAGE;

     EXCEPTION
       WHEN end_of_group THEN
         DBMS_OUTPUT.PUT_LINE ('Finished processing a group of messages');
         COMMIT;
         dequeue_options.navigation := DBMS_AQ.NEXT_TRANSACTION;
     END;
   END LOOP;
   EXCEPTION
     WHEN no_messages THEN
       DBMS_OUTPUT.PUT_LINE ('No more messages');
```

```
END;
```

Enqueuing and Dequeuing Object Type Messages That Contain LOB Attributes Using PL/SQL

```
/* Create the message payload object type with one or more LOB attributes. On
   enqueue, set the LOB attribute to EMPTY_BLOB. After the enqueue completes,
   before you commit your transaction. Select the LOB attribute from the
   user_data column of the queue table or queue table view. You can now
   use the LOB interfaces (which are available through both OCI and PL/SQL) to
   write the LOB data to the queue.  On dequeue, the message payload
   will contain the LOB locator. You can use this LOB locator after
   the dequeue, but before you commit your transaction, to read the LOB data.
 */
/* Setup the accounts: */

connect system/manager

CREATE USER aqadm IDENTIFIED BY aqadm;
GRANT CONNECT, RESOURCE TO aqadm;
GRANT aq_administrator_role TO aqadm;

CREATE USER aq IDENTIFIED BY aq;
GRANT CONNECT, RESOURCE TO aq;
GRANT EXECUTE ON DBMS_AQ TO aq;
CREATE TYPE aq.message AS OBJECT(id      NUMBER,
                                 subject VARCHAR2(100),
                                 data    BLOB,
                                 trailer NUMBER);
CREATE TABLESPACE aq_tbs DATAFILE 'aq.dbs' SIZE 2M REUSE;


/* create the queue table, queues and start the queue: */

CONNECT aqadm/aqadm
EXECUTE DBMS_AQADM.CREATE_QUEUE_TABLE(
   queue_table        => 'aq.qt1',
   queue_payload_type => 'aq.message');
EXECUTE DBMS_AQADM.CREATE_QUEUE(
   queue_name  => 'aq.queue1',
   queue_table =>  'aq.qt1');
EXECUTE DBMS_AQADM.START_QUEUE(queue_name => 'aq.queue1');
```

```
/* End set up: */

/* Enqueue of Large data types: */

CONNECT aq/aq
CREATE OR REPLACE PROCEDURE blobenqueue(msgno IN NUMBER) AS
enq_userdata aq.message;
enq_msgid    RAW(16);
enqopt       DBMS_AQ.enqueue_options_t;
msgprop      DBMS_AQ.message_properties_t;
lob_loc      BLOB;
buffer       RAW(4096);

BEGIN

   buffer := HEXTORAW(RPAD('FF', 4096, 'FF'));
   enq_userdata := aq.message(msgno, 'Large Lob data', EMPTY_BLOB(), msgno);
   DBMS_AQ.ENQUEUE('aq.queue1', enqopt, msgprop, enq_userdata, enq_msgid);

   --select the lob locator for the queue table
   SELECT t.user_data.data INTO lob_loc
      FROM qt1 t
      WHERE t.msgid = enq_msgid;

   DBMS_LOB.WRITE(lob_loc, 2000, 1, buffer );
   COMMIT;
END;

/* Dequeue lob data: */

CREATE OR REPLACE PROCEDURE blobdequeue AS
   dequeue_options    DBMS_AQ.dequeue_options_t;
   message_properties DBMS_AQ.message_properties_t;
   mid                RAW(16);
   pload              aq.message;
   lob_loc            BLOB;
   amount             BINARY_INTEGER;
   buffer             RAW(4096);

BEGIN
   DBMS_AQ.DEQUEUE('aq.queue1', dequeue_options, message_properties,
                   pload, mid);
   lob_loc := pload.data;

   -- read the lob data info buffer
```

```
   amount := 2000;
   DBMS_LOB.READ(lob_loc, amount, 1, buffer);
   DBMS_OUTPUT.PUT_LINE('Amount of data read: '||amount);
   COMMIT;
END;

/* Do the enqueues and dequeues: */

SET SERVEROUTPUT ON

BEGIN
   FOR i IN 1..5 LOOP
      blobenqueue(i);
   END LOOP;
END;

BEGIN
   FOR i IN 1..5 LOOP
     blobdequeue();
   END LOOP;
END;
```

Propagation

Caution: You may need to create queues or queue tables, or start or enable queues, for certain examples to work:

Enqueue of Messages for remote subscribers/recipients to a Multiconsumer Queue and Propagation Scheduling Using PL/SQL

```
/* Create subscriber list: */
DECLARE
   subscriber aq$_agent;

   /* Add subscribers RED and GREEN with different addresses  to the suscriber
   list: */
BEGIN
   BEGIN
      /* Add subscriber RED that will dequeue messages from another_msg_queue
      queue in the same datatbase */
      subscriber := aq$_agent('RED', 'another_msg_queue', NULL);
      DBMS_AQADM.ADD_SUBSCRIBER(queue_name => 'msg_queue_multiple',
      subscriber => subscriber);

      /* Schedule propagation from msg_queue_multiple to other queues in the
      same
      database: */
      DBMS_AQADM.SCHEDULE_PROPAGATION(queue_name => 'msg_queue_multiple');

      /* Add subscriber GREEN that will dequeue messages from the msg_queue
      queue
      in another database reached by the database link another_db.world */
      subscriber := aq$_agent('GREEN', 'msg_queue@another_db.world', NULL);
      DBMS_AQADM.ADD_SUBSCRIBER(queue_name => 'msg_queue_multiple',
      subscriber => subscriber);

      /* Schedule propagation from msg_queue_multiple to other queues in the
      database "another_database": */
   END;
   BEGIN
      DBMS_AQADM.SCHEDULE_PROPAGATION(queue_name => 'msg_queue_multiple',
      destination  => 'another_db.world');
   END;
END;
```

```
DECLARE
   enqueue_options     DBMS_AQ.enqueue_options_t;
   message_properties  DBMS_AQ.message_properties_t;
   recipients          DBMS_AQ.aq$_recipient_list_t;
   message_handle      RAW(16);
   message             aq.message_typ;

/* Enqueue MESSAGE 1 for subscribers to the queue
i.e. for RED at address another_msg_queue and GREEN at address msg_
queue@another_db.world: */
BEGIN
   message := message_typ('MESSAGE 1',
   'This message is queued for queue subscribers.');

   DBMS_AQ.ENQUEUE(queue_name => 'msg_queue_multiple',
           enqueue_options    => enqueue_options,
           message_properties => message_properties,
           payload            => message,
           msgid              => message_handle);

   /* Enqueue MESSAGE 2 for specified recipients i.e. for RED at address
   another_msg_queue and BLUE.*/
   message := message_typ('MESSAGE 2',
   'This message is queued for two recipients.');
   recipients(1) := aq$_agent('RED', 'another_msg_queue', NULL);
   recipients(2) := aq$_agent('BLUE', NULL, NULL);
   message_properties.recipient_list := recipients;

   DBMS_AQ.ENQUEUE(queue_name => 'msg_queue_multiple',
           enqueue_options    => enqueue_options,
           message_properties => message_properties,
           payload            => message,
           msgid              => message_handle);

   COMMIT;
END;
```

Note: `RED` at address `another_msg_queue` is both a subscriber to the queue, as well as being a specified recipient of `MESSAGE 2`. By contrast, `GREEN` at `address msg_queue@another_db.world` is only a subscriber to those messages in the queue (in this case, `MESSAGE 1`) for which no recipients have been specified. `BLUE`, while not a subscriber to the queue, is nevertheless specified to receive `MESSAGE 2`.

Manage Propagation From One Queue To Other Queues In The Same Database Using PL/SQL

```
/* Schedule propagation from queue q1def to other queues in the same database */
EXECUTE DBMS_AQADM.SCHEDULE_PROPAGATION(queue_name => 'q1def');

/* Disable propagation from queue q1def to other queues in the same
database */
EXECUTE DBMS_AQADM.DISABLE_PROPAGATION_SCHEDULE(
   queue_name => 'q1def');

/* Alter schedule from queue q1def to other queues in the same database */
EXECUTE DBMS_AQADM.ALTER_PROPAGATION_SCHEDULE(
   queue_name => 'q1def',
   duration   => '2000',
   next_time  => 'SYSDATE + 3600/86400',
   latency    => '32');

/* Enable propagation from queue q1def to other queues in the same database */
EXECUTE DBMS_AQADM.ENABLE_PROPAGATION_SCHEDULE(
   queue_name => 'q1def');

/* Unschedule propagation from queue q1def to other queues in the same database
*/
EXECUTE DBMS_AQADM.UNSCHEDULE_PROPAGATION(
   queue_name => 'q1def');
```

Manage Propagation From One Queue To Other Queues In Another Database Using PL/SQL

```
/* Schedule propagation from queue q1def to other queues in another  database
reached by the database link another_db.world  */
EXECUTE DBMS_AQADM.SCHEDULE_PROPAGATION(
```

```
   queue_name   => 'q1def',
   destination  => 'another_db.world');

/* Disable propagation from queue q1def to other queues in another database
reached by the database link another_db.world */
EXECUTE DBMS_AQADM.DISABLE_PROPAGATION_SCHEDULE(
   queue_name  => 'q1def',
   destination => 'another_db.world');

/* Alter schedule from queue q1def to other queues in another database  reached
by the database link another_db.world  */
EXECUTE DBMS_AQADM.ALTER_PROPAGATION_SCHEDULE(
   queue_name  => 'q1def',
   destination => 'another_db.world',
   duration    => '2000',
   next_time   => 'SYSDATE +  3600/86400',
   latency     => '32');

/* Enable propagation from queue q1def to other queues in another  database
reached by the database link another_db.world */
EXECUTE DBMS_AQADM.ENABLE_PROPAGATION_SCHEDULE(
   queue_name  => 'q1def',
   destination => 'another_db.world');

/* Unschedule propagation from queue q1def to other queues in another  database
reached by the database link another_db.world */
EXECUTE DBMS_AQADM.UNSCHEDULE_PROPAGATION(
   queue_name  => 'q1def',
   destination => 'another_db.world');
```

Unscheduling Propagation Using PL/SQL

```
/* Unschedule propagation from msg_queue_multiple to the destination another_
db.world */
EXECUTE DBMS_AQADM.UNSCHEDULE_PROPAGATION(
   queue_name => 'msg_queue_multiple',
   destination => 'another_db.world');
```

For additional examples of Alter Propagation, Enable Propagation and Disable Propagation, see:

- "Example: Alter a Propagation Schedule Using PL/SQL (DBMS_AQADM)" on page 4-67
- "Example: Enable a Propagation Using PL/SQL (DBMS_AQADM)" on page 4-69
- "Example: Disable a Propagation Using PL/SQL (DBMS_AQADM)" on page 71

Drop AQ Objects

Caution: You may need to create queues or queue tables, or start, stop, or enable queues, for certain examples to work:

```
/* Cleans up all objects related to the object type: */
CONNECT aq/aq

EXECUTE DBMS_AQADM.STOP_QUEUE (
   queue_name => 'msg_queue');

EXECUTE DBMS_AQADM.DROP_QUEUE (
   queue_name => 'msg_queue');

EXECUTE DBMS_AQADM.DROP_QUEUE_TABLE (
   queue_table => 'aq.objmsgs80_qtab');

/* Cleans up all objects related to the RAW type: */
EXECUTE DBMS_AQADM.STOP_QUEUE (
   queue_name      => 'raw_msg_queue');

EXECUTE DBMS_AQADM.DROP_QUEUE (
   queue_name      => 'raw_msg_queue');

EXECUTE DBMS_AQADM.DROP_QUEUE_TABLE (
   queue_table => 'aq.RawMsgs_qtab');

/* Cleans up all objects related to the priority queue: */
EXECUTE DBMS_AQADM.STOP_QUEUE (
   queue_name     => 'priority_msg_queue');
```

```
EXECUTE DBMS_AQADM.DROP_QUEUE (
   queue_name     => 'priority_msg_queue');

EXECUTE DBMS_AQADM.DROP_QUEUE_TABLE (
   queue_table   => 'aq.priority_msg');

/* Cleans up all objects related to the multiple-consumer queue: */
EXECUTE DBMS_AQADM.STOP_QUEUE (
   queue_name  => 'msg_queue_multiple');

EXECUTE DBMS_AQADM.DROP_QUEUE (
   queue_name  => 'msg_queue_multiple');

EXECUTE DBMS_AQADM.DROP_QUEUE_TABLE (
   queue_table => 'aq.MultiConsumerMsgs_qtab');

DROP TYPE aq.message_typ;
```

Revoke Roles and Privileges

```
CONNECT sys/change_on_install
DROP USER aq;
```

Deploy AQ with XA

Note: You may need to set up the following data structures for certain examples to work:

```
CONNECT system/manager;
DROP USER aqadm CASCADE;
GRANT CONNECT, RESOURCE TO aqadm;
CREATE USER aqadm IDENTIFIED BY aqadm;
GRANT EXECUTE ON DBMS_AQADM TO aqadm;
GRANT Aq_administrator_role TO aqadm;
DROP USER aq CASCADE;
CREATE USER aq IDENTIFIED BY aq;
GRANT CONNECT, RESOURCE TO aq;
GRANT EXECUTE ON dbms_aq TO aq;
EXECUTE DBMS_AQADM.CREATE_QUEUE_TABLE(
   queue_table => 'aq.qtable',
   queue_payload_type => 'RAW');

EXECUTE DBMS_AQADM.CREATE_QUEUE(
   queue_name => 'aq.aqsqueue',
   queue_table => 'aq.qtable');

EXECUTE DBMS_AQADM.START_QUEUE(queue_name =>
'aq.aqsqueue');
```

```
/*
 * The program uses the XA interface to enqueue 100 messages and then
 * dequeue them.
 * Login: aq/aq
 * Requires: AQ_USER_ROLE to be granted to aq
 *        a RAW queue called "aqsqueue" to be created in aqs schema
 *        (above steps can be performed by running aqaq.sql)
 * Message Format: Msgno: [0-1000] HELLO, WORLD!
 * Author: schandra@us.oracle.com
 */

#ifndef OCI_ORACLE
#include <oci.h>
#endif

#include <xa.h>
```

```
/* XA open string */
char xaoinfo[] =  "oracle_xa+ACC=P/AQ/AQ+SESTM=30+Objects=T";

/* template for generating XA XIDs */
XID  xidtempl = { 0x1e0a0a1e, 12, 8, "GTRID001BQual001" };

/* Pointer to Oracle XA function table */
extern struct xa_switch_t xaosw;                         /* Oracle XA switch */
static struct xa_switch_t *xafunc = &xaosw;

/* dummy stubs for ax_reg and ax_unreg */
int ax_reg(rmid, xid, flags)
int  rmid;
XID *xid;
long flags;
{
  xid->formatID = -1;
  return 0;
}

int ax_unreg(rmid, flags)
int     rmid;
long    flags;
{
  return 0;
}

/* generate an XID */
void xidgen(xid, serialno)
XID *xid;
int  serialno;
{
  char seq [11];

  sprintf(seq, "%d", serialno);
  memcpy((void *)xid, (void *)&xidtempl, sizeof(XID));
  strncpy((&xid->data[5]), seq, 3);
}

/* check if XA operation succeeded */
#define checkXAerr(action, funcname)    \
    if ((action) != XA_OK)        \
    {                   \
      printf("%s failed!\n", funcname);   \
      exit(-1);           \
```

```
    } else

/* check if OCI operation succeeded */
static void checkOCIerr(errhp, status)
OCIError *errhp;
sword      status;
{
  text errbuf[512];
  ub4 buflen;
  sb4 errcode;

  if (status == OCI_SUCCESS) return;

  if (status == OCI_ERROR)
  {
    OCIErrorGet((dvoid *) errhp, 1, (text *)0, &errcode, errbuf,
      (ub4)sizeof(errbuf), OCI_HTYPE_ERROR);
    printf("Error - %s\n", errbuf);
  }
  else
    printf("Error - %d\n", status);
  exit (-1);
}


void main(argc, argv)
int    argc;
char **argv;
{
  int         msgno = 0;             /* message being enqueued */
  OCIEnv       *envhp;                /* OCI environment handle */
  OCIError     *errhp;                 /* OCI Error handle */
  OCISvcCtx    *svchp;                    /* OCI Service handle */
  char      message[128];                /* message buffer */
  ub4      mesglen;            /* length of message */
  OCIRaw       *rawmesg = (OCIRaw *)0;       /* message in OCI RAW format */
  OCIInd      ind = 0;                 /* OCI null indicator */
  dvoid          *indptr = (dvoid *)&ind;       /* null indicator pointer */
  OCIType      *mesg_tdo = (OCIType *) 0;         /* TDO for RAW datatype */
  XID      xid;                /* XA's global transaction id */
  ub4      i;                     /* array index */

  checkXAerr(xafunc->xa_open_entry(xaoinfo, 1, TMNOFLAGS), "xaoopen");

  svchp = xaoSvcCtx((text *)0);           /* get service handle from XA */
```

```
envhp = xaoEnv((text *)0);          /* get enviornment handle from XA */

if (!svchp || !envhp)
{
  printf("Unable to obtain OCI Handles from XA!\n");
  exit (-1);
}

OCIHandleAlloc((dvoid *)envhp, (dvoid **)&errhp,
     OCI_HTYPE_ERROR, 0, (dvoid **)0);  /* allocate error handle */

/* enqueue 1000 messages, 1 message per XA transaction */
for (msgno = 0; msgno < 1000; msgno++)
{
  sprintf((const char *)message, "Msgno: %d, Hello, World!", msgno);
  mesglen = (ub4)strlen((const char *)message);
  xidgen(&xid, msgno);                 /* generate an XA xid */

  checkXAerr(xafunc->xa_start_entry(&xid, 1, TMNOFLAGS), "xaostart");

  checkOCIerr(errhp, OCIRawAssignBytes(envhp, errhp, (ub1 *)message, mesglen,
              &rawmesg));

  if (!mesg_tdo)          /* get Type descriptor (TDO) for RAW type */
    checkOCIerr(errhp, OCITypeByName(envhp, errhp, svchp,
                    (CONST text *)"AQADM", strlen("AQADM"),
                     (CONST text *)"RAW", strlen("RAW"),
                 (text *)0, 0, OCI_DURATION_SESSION,
                 OCI_TYPEGET_ALL, &mesg_tdo));

  checkOCIerr(errhp, OCIAQEnq(svchp, errhp, (CONST text *)"aqsqueue",
             0, 0, mesg_tdo, (dvoid **)&rawmesg, &indptr,
          0, 0));

  checkXAerr(xafunc->xa_end_entry(&xid, 1, TMSUCCESS), "xaoend");
  checkXAerr(xafunc->xa_commit_entry(&xid, 1, TMONEPHASE), "xaocommit");
  printf("%s Enqueued\n", message);
}

/* dequeue 1000 messages within one XA transaction */
xidgen(&xid, msgno);                         /* generate an XA xid */
checkXAerr(xafunc->xa_start_entry(&xid, 1, TMNOFLAGS), "xaostart");
for (msgno = 0; msgno < 1000; msgno++)
{
  checkOCIerr(errhp, OCIAQDeq(svchp, errhp, (CONST text *)"aqsqueue",
```

```
              0, 0, mesg_tdo, (dvoid **)&rawmesg, &indptr,
             0, 0));
     if (ind)
       printf("Null Raw Message");
     else
       for (i = 0; i < OCIRawSize(envhp, rawmesg); i++)
    printf("%c", *(OCIRawPtr(envhp, rawmesg) + i));
     printf("\n");

   }
   checkXAerr(xafunc->xa_end_entry(&xid, 1, TMSUCCESS), "xaoend");
   checkXAerr(xafunc->xa_commit_entry(&xid, 1, TMONEPHASE), "xaocommit");
 }
```

AQ and Memory Usage

Create_types.sql : Create Payload Types and Queues in Scott's Schema

---

Note: You may need to set up data structures for certain examples to work, such as:

```
/* Create_types.sql */
CONNECT system/manager
GRANT AQ_ADMINISTRATOR_ROLE, AQ_USER_ROLE TO scott;
CONNECT scott/tiger
CREATE TYPE MESSAGE AS OBJECT (id NUMBER, data VARCHAR2(80));
EXECUTE DBMS_AQADM.CREATE_QUEUE_TABLE(
   queue_table        => 'qt',
   queue_payload_type => 'message');
EXECUTE DBMS_AQADM.CREATE_QUEUE('msgqueue', 'qt');
EXECUTE DBMS_AQADM.START_QUEUE('msgqueue');
```

Enqueue Messages (Free Memory After Every Call) Using OCI

This program, `enqnoreuse.c`, dequeues each line of text from a queue '`msgqueue`' that has been created in scott's schema via *create_types.sql*, above. Messages are enqueued using `enqnoreuse.c` or `enqreuse.c` (see below). If there are no messages, it waits for 60 seconds before timing out. In this program, the dequeue subroutine does not reuse client side objects' memory. It allocates the required memory before dequeue and frees it after the dequeue is complete.

```
#ifndef OCI_ORACLE
#include <oci.h>
#endif

#include <stdio.h>

static void checkerr(OCIError *errhp, sword status);
static void deqmesg(text *buf, ub4 *buflen);

OCIEnv      *envhp;
OCIError    *errhp;
OCISvcCtx   *svchp;

struct message
{
```

```
  OCINumber    id;
  OCIString   *data;
};
typedef struct message message;

struct null_message
{
  OCIInd    null_adt;
  OCIInd    null_id;
  OCIInd    null_data;
};
typedef struct null_message null_message;

static void deqmesg(buf, buflen)
text  *buf;
ub4   *buflen;
{
  OCIType         *mesgtdo = (OCIType *)0;   /* type descr of SCOTT.MESSAGE */
  message         *mesg    = (dvoid *)0;     /* instance of SCOTT.MESSAGE */
  null_message    *mesgind = (dvoid *)0;     /* null indicator */
  OCIAQDeqOptions *deqopt  = (OCIAQDeqOptions *)0;
  ub4              wait    = 60;             /* timeout after 60 seconds */
  ub4              navigation = OCI_DEQ_FIRST_MSG;/* always get head of q */


  /* Get the type descriptor object for the type SCOTT.MESSAGE: */
  checkerr(errhp, OCITypeByName(envhp, errhp, svchp,
           (CONST text *)"SCOTT", strlen("SCOTT"),
           (CONST text *)"MESSAGE", strlen("MESSAGE"),
           (text *)0, 0, OCI_DURATION_SESSION,
           OCI_TYPEGET_ALL, &mesgtdo));

  /* Allocate an instance of SCOTT.MESSAGE, and get its null indicator: */
  checkerr(errhp, OCIObjectNew(envhp, errhp, svchp, OCI_TYPECODE_OBJECT,
           mesgtdo, (dvoid *)0, OCI_DURATION_SESSION,
           TRUE, (dvoid **)&mesg));
  checkerr(errhp, OCIObjectGetInd(envhp, errhp, (dvoid *)mesg,
           (dvoid **)&mesgind));

  /* Allocate a descriptor for dequeue options and set wait time, navigation: */
  checkerr(errhp, OCIDescriptorAlloc(envhp, (dvoid **)&deqopt,
           OCI_DTYPE_AQDEQ_OPTIONS, 0, (dvoid **)0));
  checkerr(errhp, OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS,
           (dvoid *)&wait, 0, OCI_ATTR_WAIT, errhp));
  checkerr(errhp, OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS,
```

```
           (dvoid *)&navigation, 0,
           OCI_ATTR_NAVIGATION, errhp));

  /* Dequeue the message and commit: */
  checkerr(errhp, OCIAQDeq(svchp, errhp, (CONST text *)"msgqueue",
           deqopt, 0, mesgtdo, (dvoid **)&mesg,
           (dvoid **)&mesgind, 0, 0));

  checkerr(errhp, OCITransCommit(svchp, errhp, (ub4) 0));

  /* Copy the message payload text into the user buffer: */
  if (mesgind->null_data)
    *buflen = 0;
  else
    memcpy((dvoid *)buf, (dvoid *)OCIStringPtr(envhp, mesg->data),
           (size_t)(*buflen = OCIStringSize(envhp, mesg->data)));

  /* Free the dequeue options descriptor: */
  checkerr(errhp, OCIDescriptorFree((dvoid *)deqopt, OCI_DTYPE_AQDEQ_OPTIONS));

  /* Free the memory for the objects: */
  Checkerr(errhp, OCIObjectFree(envhp, errhp, (dvoid *)mesg,
           OCI_OBJECTFREE_FORCE));
}                           /* end deqmesg */

void main()
{
  OCIServer     *srvhp;
  OCISession    *usrhp;
  dvoid         *tmp;
  text           buf[80];                 /* payload text */
  ub4            buflen;

  OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,  (dvoid * (*)()) 0,
                (dvoid * (*)()) 0,  (void (*)()) 0 );

  OCIHandleAlloc((dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
                 52, (dvoid **) &tmp);

  OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp  );

  OCIHandleAlloc((dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
                 52, (dvoid **) &tmp);
  OCIHandleAlloc((dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                 52, (dvoid **) &tmp);
```

```
  OCIServerAttach(srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

  OCIHandleAlloc((dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                 52, (dvoid **) &tmp);

  /* Set attribute server context in the service context: */
  OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
             (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

  /* Allocate a user context handle: */
  OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp, (ub4) OCI_HTYPE_SESSION,
                 (size_t) 0, (dvoid **) 0);

  OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION,
             (dvoid *)"scott", (ub4)strlen("scott"), OCI_ATTR_USERNAME, errhp);

  OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION,
             (dvoid *)"tiger", (ub4)strlen("tiger"), OCI_ATTR_PASSWORD, errhp);

  checkerr(errhp, OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS,
           OCI_DEFAULT));

  OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX,
             (dvoid *)usrhp, (ub4)0, OCI_ATTR_SESSION, errhp);

  do {
    deqmesg(buf, &buflen);
    printf("%.*s\n", buflen, buf);
  } while(1);
}                         /* end main */

static void checkerr(errhp, status)
OCIError  *errhp;
sword      status;
{
  text errbuf[512];
  ub4  buflen;
  sb4 errcode;

  if (status == OCI_SUCCESS) return;

  switch (status)
  {
  case OCI_ERROR:
```

```
    OCIErrorGet ((dvoid *) errhp, (ub4) 1, (text *) NULL, &errcode,
                 errbuf, (ub4) sizeof(errbuf), (ub4) OCI_HTYPE_ERROR);
    printf("Error - %s\n", errbuf);
    break;
  case OCI_INVALID_HANDLE:
    printf("Error - OCI_INVALID_HANDLE\n");
    break;
  default:
    printf("Error - %d\n", status);
    break;
  }
  exit(-1);
}                          /* end checkerr */
```

Enqueue Messages (Reuse Memory) Using OCI

This program, `enqreuse.c`, enqueues each line of text into a queue 'msgqueue' that has been created in scott's schema by executing `create_types.sql`. Each line of text entered by the user is stored in the queue until user enters `EOF`. In this program the enqueue subroutine reuses the memory for the message payload, as well as the AQ message properties descriptor.

```
#ifndef OCI_ORACLE
#include <oci.h>
#endif

#include <stdio.h>

static void checkerr(OCIError *errhp, sword status);
static void enqmesg(ub4 msgno, text *buf);

struct message
{
  OCINumber     id;
  OCIString    *data;
};
typedef struct message message;

struct null_message
{
  OCIInd    null_adt;
  OCIInd    null_id;
  OCIInd    null_data;
};
typedef struct null_message null_message;
```

```
/* Global data reused on calls to enqueue: */
OCIEnv             *envhp;
OCIError           *errhp;
OCISvcCtx          *svchp;
message             msg;
null_message        nmsg;
OCIAQMsgProperties *msgprop;

static void enqmesg(msgno, buf)
ub4     msgno;
text   *buf;
{
  OCIType          *mesgtdo = (OCIType *)0; /* type descr of SCOTT.MESSAGE */
  message          *mesg = &msg;            /* instance of SCOTT.MESSAGE */
  null_message     *mesgind = &nmsg;        /* null indicator */
  text              corrid[128];            /* correlation identifier */

  /* Get the type descriptor object for the type SCOTT.MESSAGE: */
  checkerr(errhp, OCITypeByName(envhp, errhp, svchp,
           (CONST text *)"SCOTT", strlen("SCOTT"),
           (CONST text *)"MESSAGE", strlen("MESSAGE"),
           (text *)0, 0, OCI_DURATION_SESSION,
           OCI_TYPEGET_ALL, &mesgtdo));

  /* Fill in the attributes of SCOTT.MESSAGE: */
  checkerr(errhp, OCINumberFromInt(errhp, &msgno, sizeof(ub4), 0, &mesg->id));
  checkerr(errhp, OCIStringAssignText(envhp, errhp, buf, strlen(buf),
           &mesg->data));
  mesgind->null_adt = mesgind->null_id = mesgind->null_data = 0;

  /* Set the correlation id in the message properties descriptor: */
  sprintf((char *)corrid, "Msg#: %d", msgno);
  checkerr(errhp, OCIAttrSet(msgprop, OCI_DTYPE_AQMSG_PROPERTIES,
           (dvoid *)&corrid, strlen(corrid),
           OCI_ATTR_CORRELATION, errhp));

  /* Enqueue the message and commit: */
  checkerr(errhp, OCIAQEnq(svchp, errhp, (CONST text *)"msgqueue",
           0, msgprop, mesgtdo, (dvoid **)&mesg,
           (dvoid **)&mesgind, 0, 0));

  checkerr(errhp, OCITransCommit(svchp, errhp, (ub4) 0));
}                           /* end enqmesg */
```

```
void main()
{
  OCIServer    *srvhp;
  OCISession   *usrhp;
  dvoid        *tmp;
  text          buf[80];               /* user supplied text */
  int           msgno = 0;

  OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,  (dvoid * (*)()) 0,
                (dvoid * (*)()) 0,  (void (*)()) 0 );

  OCIHandleAlloc((dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
                 52, (dvoid **) &tmp);

  OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp  );

  OCIHandleAlloc((dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
                 52, (dvoid **) &tmp);
  OCIHandleAlloc((dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                 52, (dvoid **) &tmp);

  OCIServerAttach(srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

  OCIHandleAlloc((dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                 52, (dvoid **) &tmp);

  /* Set attribute server context in the service context: */
  OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
           (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

  /* Allocate a user context handle: */
  OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp, (ub4) OCI_HTYPE_SESSION,
                 (size_t) 0, (dvoid **) 0);

  OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION,
           (dvoid *)"scott", (ub4)strlen("scott"), OCI_ATTR_USERNAME, errhp);

  OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION,
           (dvoid *)"tiger", (ub4)strlen("tiger"), OCI_ATTR_PASSWORD, errhp);

  checkerr(errhp, OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS,
         OCI_DEFAULT));

  OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX,
           (dvoid *)usrhp, (ub4)0, OCI_ATTR_SESSION, errhp);
```

```
  /* Allocate a message properties descriptor to fill in correlation id :*/
  checkerr(errhp, OCIDescriptorAlloc(envhp, (dvoid **)&msgprop,
           OCI_DTYPE_AQMSG_PROPERTIES,
           0, (dvoid **)0));
  do {
    printf("Enter a line of text (max 80 chars):");
    if (!gets((char *)buf))
      break;
    enqmesg((ub4)msgno++, buf);
  } while(1);

  /* Free the message properties descriptor: */
  checkerr(errhp, OCIDescriptorFree((dvoid *)msgprop,
           OCI_DTYPE_AQMSG_PROPERTIES));

}                        /* end main */

static void checkerr(errhp, status)
OCIError   *errhp;
sword      status;
{
  text errbuf[512];
  ub4  buflen;
  sb4  errcode;

  if (status == OCI_SUCCESS) return;

  switch (status)
  {
  case OCI_ERROR:
    OCIErrorGet ((dvoid *) errhp, (ub4) 1, (text *) NULL, &errcode,
                 errbuf, (ub4) sizeof(errbuf), (ub4) OCI_HTYPE_ERROR);
    printf("Error - %s\n", errbuf);
    break;
  case OCI_INVALID_HANDLE:
    printf("Error - OCI_INVALID_HANDLE\n");
    break;
  default:
    printf("Error - %d\n", status);
    break;
  }
  exit(-1);
}                         /* end checkerr */
```

Dequeue Messages (Free Memory After Every Call) Using OCI

This program, `deqnoreuse.c`, dequeues each line of text from a queue '`msgqueue`' that has been created in scott's schema by executing *c*`reate_types.sql`. Messages are enqueued using `enqnoreuse` or `enqreuse`. If there are no messages, it waits for 60 seconds before timing out. In this program the dequeue subroutine does not reuse client side objects' memory. It allocates the required memory before dequeue and frees it after the dequeue is complete.

```
#ifndef OCI_ORACLE
#include <oci.h>
#endif

#include <stdio.h>

static void checkerr(OCIError *errhp, sword status);
static void deqmesg(text *buf, ub4 *buflen);

OCIEnv       *envhp;
OCIError     *errhp;
OCISvcCtx    *svchp;

struct message
{
  OCINumber     id;
  OCIString    *data;
};
typedef struct message message;

struct null_message
{
  OCIInd    null_adt;
  OCIInd    null_id;
  OCIInd    null_data;
};
typedef struct null_message null_message;

static void deqmesg(buf, buflen)
text      *buf;
ub4       *buflen;
{
  OCIType          *mesgtdo = (OCIType *)0;  /* type descr of SCOTT.MESSAGE */
  message          *mesg = (dvoid *)0;       /* instance of SCOTT.MESSAGE */
  null_message     *mesgind   = (dvoid *)0;       /* null indicator */
  OCIAQDeqOptions  *deqopt    = (OCIAQDeqOptions *)0;
```

```
ub4              wait      = 60;              /* timeout after 60 seconds */
ub4             navigation = OCI_DEQ_FIRST_MSG;/* always get head of q */

 /* Get the type descriptor object for the type SCOTT.MESSAGE: */
checkerr(errhp, OCITypeByName(envhp, errhp, svchp,
         (CONST text *)"SCOTT", strlen("SCOTT"),
         (CONST text *)"MESSAGE", strlen("MESSAGE"),
         (text *)0, 0, OCI_DURATION_SESSION,
         OCI_TYPEGET_ALL, &mesgtdo));

/* Allocate an instance of SCOTT.MESSAGE, and get its null indicator: */
checkerr(errhp, OCIObjectNew(envhp, errhp, svchp, OCI_TYPECODE_OBJECT,
         mesgtdo, (dvoid *)0, OCI_DURATION_SESSION,
         TRUE, (dvoid **)&mesg));
checkerr(errhp, OCIObjectGetInd(envhp, errhp, (dvoid *)mesg,
         (dvoid **)&mesgind));

/* Allocate a descriptor for dequeue options and set wait time, navigation: */
checkerr(errhp, OCIDescriptorAlloc(envhp, (dvoid **)&deqopt,
         OCI_DTYPE_AQDEQ_OPTIONS, 0, (dvoid **)0));
checkerr(errhp, OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS,
         (dvoid *)&wait, 0, OCI_ATTR_WAIT, errhp));
checkerr(errhp, OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS,
         (dvoid *)&navigation, 0,
         OCI_ATTR_NAVIGATION, errhp));

/* Dequeue the message and commit: */
checkerr(errhp, OCIAQDeq(svchp, errhp, (CONST text *)"msgqueue",
         deqopt, 0, mesgtdo, (dvoid **)&mesg,
         (dvoid **)&mesgind, 0, 0));

checkerr(errhp, OCITransCommit(svchp, errhp, (ub4) 0));

/* Copy the message payload text into the user buffer: */
if (mesgind->null_data)
  *buflen = 0;
else
  memcpy((dvoid *)buf, (dvoid *)OCIStringPtr(envhp, mesg->data),
         (size_t)(*buflen = OCIStringSize(envhp, mesg->data)));

/* Free the dequeue options descriptor: */
checkerr(errhp, OCIDescriptorFree((dvoid *)deqopt, OCI_DTYPE_AQDEQ_OPTIONS));

/* Free the memory for the objects: */
checkerr(errhp, OCIObjectFree(envhp, errhp, (dvoid *)mesg,
```

```
           OCI_OBJECTFREE_FORCE));
}                                     /* end deqmesg */

void main()
{
  OCIServer    *srvhp;
  OCISession   *usrhp;
  dvoid        *tmp;
  text          buf[80];               /* payload text */
  ub4           buflen;

  OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,  (dvoid * (*)()) 0,
                (dvoid * (*)()) 0,  (void (*)()) 0 );

  OCIHandleAlloc((dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
                 52, (dvoid **) &tmp);

  OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp  );

  OCIHandleAlloc((dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
                 52, (dvoid **) &tmp);
  OCIHandleAlloc((dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
                 52, (dvoid **) &tmp);

  OCIServerAttach(srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

  OCIHandleAlloc((dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
                 52, (dvoid **) &tmp);

  /* Set attribute server context in the service context: */
  OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
             (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

  /* Allocate a user context handle: */
  OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp, (ub4) OCI_HTYPE_SESSION,
                 (size_t) 0, (dvoid **) 0);

  OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION,
             (dvoid *)"scott", (ub4)strlen("scott"), OCI_ATTR_USERNAME, errhp);

  OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION,
             (dvoid *)"tiger", (ub4)strlen("tiger"), OCI_ATTR_PASSWORD, errhp);

  checkerr(errhp, OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS,
           OCI_DEFAULT));
```

```
  OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX,
             (dvoid *)usrhp, (ub4)0, OCI_ATTR_SESSION, errhp);

  do {
    deqmesg(buf, &buflen);
    printf("%.*s\n", buflen, buf);
  } while(1);
}                         /* end main */

static void checkerr(errhp, status)
OCIError   *errhp;
sword      status;
{
  text errbuf[512];
  ub4  buflen;
  sb4  errcode;

  if (status == OCI_SUCCESS) return;

  switch (status)
  {
  case OCI_ERROR:
    OCIErrorGet ((dvoid *) errhp, (ub4) 1, (text *) NULL, &errcode,
                 errbuf, (ub4) sizeof(errbuf), (ub4) OCI_HTYPE_ERROR);
    printf("Error - %s\n", errbuf);
    break;
  case OCI_INVALID_HANDLE:
    printf("Error - OCI_INVALID_HANDLE\n");
    break;
  default:
    printf("Error - %d\n", status);
    break;
  }
  exit(-1);
}                          /* end checkerr */
```

Dequeue Messages (Reuse Memory) Using OCI

This program, `deqreuse.c`, dequeues each line of text from a queue '`msgqueue`' that has been created in scott's schema by executing `create_types.sql`. Messages are enqueued using `enqnoreuse.c` or `enqreuse.c`. If there are no messages, it waits for 60 seconds before timing out. In this program, the dequeue subroutine reuses client side objects' memory between invocation of `OCIAQDeq`.

During the first call to `OCIAQDeq`, OCI automatically allocates the memory for the message payload. During subsequent calls to `OCIAQDeq`, the same payload pointers are passed and OCI will automatically resize the payload memory if necessary.

```
#ifndef OCI_ORACLE
#include <oci.h>
#endif

#include <stdio.h>

static void checkerr(OCIError *errhp, sword status);
static void deqmesg(text *buf, ub4 *buflen);

struct message
{
  OCINumber    id;
  OCIString   *data;
};
typedef struct message message;

struct null_message
{
  OCIInd    null_adt;
  OCIInd    null_id;
  OCIInd    null_data;
};
typedef struct null_message null_message;

/* Global data reused on calls to enqueue: */
OCIEnv          *envhp;
OCIError        *errhp;
OCISvcCtx       *svchp;
OCIAQDeqOptions *deqopt;
message         *mesg = (message *)0;
null_message    *mesgind = (null_message *)0;

static void deqmesg(buf, buflen)
text       *buf;
ub4        *buflen;
{



  OCIType        *mesgtdo    = (OCIType *)0; /* type descr of SCOTT.MESSAGE */
```

```
  ub4             wait       = 60;           /* timeout after 60 seconds */
  ub4             navigation = OCI_DEQ_FIRST_MSG;/* always get head of q */

  /* Get the type descriptor object for the type SCOTT.MESSAGE: */
  checkerr(errhp, OCITypeByName(envhp, errhp, svchp,
           (CONST text *)"SCOTT", strlen("SCOTT"),
           (CONST text *)"MESSAGE", strlen("MESSAGE"),
           (text *)0, 0, OCI_DURATION_SESSION,
           OCI_TYPEGET_ALL, &mesgtdo));

  /* Set wait time, navigation in dequeue options: */
  checkerr(errhp, OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS,
           (dvoid *)&wait, 0, OCI_ATTR_WAIT, errhp));
  checkerr(errhp, OCIAttrSet(deqopt, OCI_DTYPE_AQDEQ_OPTIONS,
           (dvoid *)&navigation, 0,
           OCI_ATTR_NAVIGATION, errhp));

  /*
   * Dequeue the message and commit. The memory for the payload will be
   * automatically allocated/resized by OCI:
   */
  checkerr(errhp, OCIAQDeq(svchp, errhp, (CONST text *)"msgqueue",
           deqopt, 0, mesgtdo, (dvoid **)&mesg,
           (dvoid **)&mesgind, 0, 0));

  checkerr(errhp, OCITransCommit(svchp, errhp, (ub4) 0));

  /* Copy the message payload text into the user buffer: */
  if (mesgind->null_data)
    *buflen = 0;
  else
    memcpy((dvoid *)buf, (dvoid *)OCIStringPtr(envhp, mesg->data),
           (size_t)(*buflen = OCIStringSize(envhp, mesg->data)));
}                           /* end deqmesg */

void main()
{
  OCIServer    *srvhp;
  OCISession   *usrhp;
  dvoid        *tmp;
  text          buf[80];                 /* payload text */
  ub4           buflen;

  OCIInitialize((ub4) OCI_OBJECT, (dvoid *)0,  (dvoid * (*)()) 0,
                (dvoid * (*)()) 0,  (void (*)()) 0 );
```

```
OCIHandleAlloc((dvoid *) NULL, (dvoid **) &envhp, (ub4) OCI_HTYPE_ENV,
               52, (dvoid **) &tmp);

OCIEnvInit( &envhp, (ub4) OCI_DEFAULT, 21, (dvoid **) &tmp  );

OCIHandleAlloc((dvoid *) envhp, (dvoid **) &errhp, (ub4) OCI_HTYPE_ERROR,
               52, (dvoid **) &tmp);
OCIHandleAlloc((dvoid *) envhp, (dvoid **) &srvhp, (ub4) OCI_HTYPE_SERVER,
               52, (dvoid **) &tmp);

OCIServerAttach(srvhp, errhp, (text *) 0, (sb4) 0, (ub4) OCI_DEFAULT);

OCIHandleAlloc((dvoid *) envhp, (dvoid **) &svchp, (ub4) OCI_HTYPE_SVCCTX,
               52, (dvoid **) &tmp);

/* set attribute server context in the service context */
OCIAttrSet((dvoid *) svchp, (ub4) OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4) 0,
           (ub4) OCI_ATTR_SERVER, (OCIError *) errhp);

/* allocate a user context handle */
OCIHandleAlloc((dvoid *)envhp, (dvoid **)&usrhp, (ub4) OCI_HTYPE_SESSION,
               (size_t) 0, (dvoid **) 0);

OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION,
           (dvoid *)"scott", (ub4)strlen("scott"), OCI_ATTR_USERNAME, errhp);

OCIAttrSet((dvoid *)usrhp, (ub4)OCI_HTYPE_SESSION,
           (dvoid *)"tiger", (ub4)strlen("tiger"), OCI_ATTR_PASSWORD, errhp);

checkerr(errhp, OCISessionBegin (svchp, errhp, usrhp, OCI_CRED_RDBMS,
         OCI_DEFAULT));

OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX,
           (dvoid *)usrhp, (ub4)0, OCI_ATTR_SESSION, errhp);

/* allocate the dequeue options descriptor */
checkerr(errhp, OCIDescriptorAlloc(envhp, (dvoid **)&deqopt,
         OCI_DTYPE_AQDEQ_OPTIONS, 0, (dvoid **)0));

do {
  deqmesg(buf, &buflen);
  printf("%.*s\n", buflen, buf);
} while(1);
```

```
  /*
   * This program never reaches this point as the dequeue timesout & exits.
   * If it does reach here, it will be a good place to free the dequeue
   * options descriptor using OCIDescriptorFree and free the memory allocated
   * by OCI for the payload using OCIObjectFree
   */
}                         /* end main */

static void checkerr(errhp, status)
OCIError *errhp;
sword      status;
{
  text errbuf[512];
  ub4  buflen;
  sb4  errcode;

  if (status == OCI_SUCCESS) return;

  switch (status)
  {
  case OCI_ERROR:
    OCIErrorGet ((dvoid *) errhp, (ub4) 1, (text *) NULL, &errcode,
                 errbuf, (ub4) sizeof(errbuf), (ub4) OCI_HTYPE_ERROR);
    printf("Error - %s\n", errbuf);
    break;
  case OCI_INVALID_HANDLE:
    printf("Error - OCI_INVALID_HANDLE\n");
    break;
  default:
    printf("Error - %d\n", status);
    break;
  }
  exit(-1);
}                          /* end checkerr */
```

A

Scripts for Implementing ’BooksOnLine’

This Appendix contains the following scripts:

- tkaqdoca.sql: Script to Create Users, Objects, Queue Tables, Queues & Subscribers
- tkaqdocd.sql: Examples of Administrative and Operational Interfaces
- tkaqdoce.sql: Operational Examples
- tkaqdocp.sql: Examples of Operational Interfaces
- tkaqdocc.sql: Clean-Up Script

tkaqdoca.sql: Script to Create Users, Objects, Queue Tables, Queues & Subscribers

```
Rem $Header: tkaqdoca.sql 26-jan-99.17:50:37 aquser1 Exp $
Rem
Rem tkaqdoca.sql
Rem
Rem  Copyright (c) Oracle Corporation 1998, 1999. All Rights Reserved.
Rem
Rem    NAME
Rem      tkaqdoca.sql - TKAQ DOCumentation Admin examples file

Rem Set up a queue admin account and individual accounts for each application
Rem
connect system/manager
set serveroutput on;
set echo on;

Rem Create a common admin account for all BooksOnLine applications
Rem
create user BOLADM identified by BOLADM;
grant connect, resource, aq_administrator_role to BOLADM;
grant execute on dbms_aq to BOLADM;
grant execute on dbms_aqadm to BOLADM;
execute dbms_aqadm.grant_system_privilege('ENQUEUE_ANY','BOLADM',FALSE);
execute dbms_aqadm.grant_system_privilege('DEQUEUE_ANY','BOLADM',FALSE);

Rem Create the application schemas and grant appropriate permission
Rem to all schemas

Rem Create an account for Order Entry
create user OE identified by OE;
grant connect, resource to OE;
grant execute on dbms_aq to OE;
grant execute on dbms_aqadm to OE;

Rem Create an account for WR Shipping
create user WS identified by WS;
grant connect, resource to WS;
grant execute on dbms_aq to WS;
grant execute on dbms_aqadm to WS;

Rem Create an account for ER Shipping
create user ES identified by ES;
grant connect, resource to ES;
```

```
grant execute on dbms_aq to ES;
grant execute on dbms_aqadm to ES;

Rem Create an account for Overseas Shipping
create user OS identified by OS;
grant connect, resource to OS;
grant execute on dbms_aq to OS;
grant execute on dbms_aqadm to OS;

Rem Create an account for Customer Billing
Rem Customer Billing, for security reason, has an admin schema that
Rem hosts all the queue tables and an application schema from where
Rem the application runs.
create user CBADM identified by CBADM;
grant connect, resource to CBADM;
grant execute on dbms_aq to CBADM;
grant execute on dbms_aqadm to CBADM;

create user CB identified by CB;
grant connect, resource to CB;
grant execute on dbms_aq to CB;
grant execute on dbms_aqadm to CB;

Rem Create an account for Customer Service
create user CS identified by CS;
grant connect, resource to CS;
grant execute on dbms_aq to CS;
grant execute on dbms_aqadm to CS;


Rem All object types are created in the administrator schema.
Rem All application schemas that host any propagation source
Rem queues are given the ENQUEUE_ANY system level privilege
Rem allowing the application schemas to enqueue to the destination
Rem queue.
Rem
connect BOLADM/BOLADM;

Rem Create objects

create or replace type customer_typ as object (
        custno          number,
        name            varchar2(100),
        street          varchar2(100),
        city            varchar2(30),
```

```
        state           varchar2(2),
        zip             number,
        country         varchar2(100));
/

create or replace type book_typ as object (
        title           varchar2(100),
        authors         varchar2(100),
        ISBN            number,
        price           number);
/

create or replace type orderitem_typ as object (
        quantity        number,
        item            book_typ,
        subtotal        number);
/

create or replace type orderitemlist_vartyp as varray (20) of orderitem_typ;
/

create or replace type order_typ as object (
        orderno         number,
        status          varchar2(30),
        ordertype       varchar2(30),
        orderregion     varchar2(30),
        customer        customer_typ,
        paymentmethod   varchar2(30),
        items           orderitemlist_vartyp,
        total           number);
/

grant execute on order_typ to OE;
grant execute on orderitemlist_vartyp to OE;
grant execute on orderitem_typ to OE;
grant execute on book_typ to OE;
grant execute on customer_typ to OE;
execute dbms_aqadm.grant_system_privilege('ENQUEUE_ANY','OE',FALSE);

grant execute on order_typ to WS;
grant execute on orderitemlist_vartyp to WS;
grant execute on orderitem_typ to WS;
grant execute on book_typ to WS;
grant execute on customer_typ to WS;
execute dbms_aqadm.grant_system_privilege('ENQUEUE_ANY','WS',FALSE);
```

```
grant execute on order_typ to ES;
grant execute on orderitemlist_vartyp to ES;
grant execute on orderitem_typ to ES;
grant execute on book_typ to ES;
grant execute on customer_typ to ES;
execute dbms_aqadm.grant_system_privilege('ENQUEUE_ANY','ES',FALSE);

grant execute on order_typ to OS;
grant execute on orderitemlist_vartyp to OS;
grant execute on orderitem_typ to OS;
grant execute on book_typ to OS;
grant execute on customer_typ to OS;
execute dbms_aqadm.grant_system_privilege('ENQUEUE_ANY','OS',FALSE);

grant execute on order_typ to CBADM;
grant execute on orderitemlist_vartyp to CBADM;
grant execute on orderitem_typ to CBADM;
grant execute on book_typ to CBADM;
grant execute on customer_typ to CBADM;

grant execute on order_typ to CB;
grant execute on orderitemlist_vartyp to CB;
grant execute on orderitem_typ to CB;
grant execute on book_typ to CB;
grant execute on customer_typ to CB;

grant execute on order_typ to CS;
grant execute on orderitemlist_vartyp to CS;
grant execute on orderitem_typ to CS;
grant execute on book_typ to CS;
grant execute on customer_typ to CS;

Rem Create queue tables, queues for OE
Rem
connect OE/OE;
begin
dbms_aqadm.create_queue_table(
        queue_table => 'OE_orders_sqtab',
        comment => 'Order Entry Single Consumer Orders queue table',
        queue_payload_type => 'BOLADM.order_typ',
        message_grouping => DBMS_AQADM.TRANSACTIONAL,
        compatible => '8.1',
        primary_instance => 1,
        secondary_instance => 2);
```

```
end;
/

Rem Create a priority queue table for OE
begin
dbms_aqadm.create_queue_table(
        queue_table => 'OE_orders_pr_mqtab',
        sort_list =>'priority,enq_time',
        comment => 'Order Entry Priority MultiConsumer Orders queue table',
        multiple_consumers => TRUE,
        queue_payload_type => 'BOLADM.order_typ',
        compatible => '8.1',
        primary_instance => 2,
        secondary_instance => 1);
end;
/

begin
dbms_aqadm.create_queue (
        queue_name              => 'OE_neworders_que',
        queue_table             => 'OE_orders_sqtab');
end;
/

begin
dbms_aqadm.create_queue (
        queue_name              => 'OE_bookedorders_que',
        queue_table             => 'OE_orders_pr_mqtab');
end;
/

Rem Orders in OE_bookedorders_que are being propagated to WS_bookedorders_que,
Rem ES_bookedorders_que and OS_bookedorders_que according to the region
Rem the books are shipped to.  At the time an order is placed, the customer
Rem can request Fed-ex shipping (priority 1), priority air shipping (priority
Rem 2) and ground shipping (priority 3).  An priority queue is created in
Rem each region, the shipping applications will dequeue from these priority
Rem queues according to the orders' shipping priorities, processes the orders
Rem and enqueue the processed orders into
Rem the shipped_orders queues or the back_orders queues.  Both the shipped_
Rem orders queues and the back_orders queues are FIFO queues. However,
Rem orders put into the back_orders_queues are enqueued with delay time
Rem set to 1 day, so that each order in the back_order_queues is processed
Rem only once a day until the shipment is filled.
```

```
Rem Create queue tables, queues for WS Shipping
connect WS/WS;

Rem Create a priority queue table for WS shipping
begin
dbms_aqadm.create_queue_table(
        queue_table => 'WS_orders_pr_mqtab',
        sort_list =>'priority,enq_time',
        comment => 'West Shipping Priority MultiConsumer Orders queue table',
        multiple_consumers => TRUE,
        queue_payload_type => 'BOLADM.order_typ',
        compatible => '8.1');
end;
/

Rem Create a FIFO queue tables for WS shipping
begin
dbms_aqadm.create_queue_table(
        queue_table => 'WS_orders_mqtab',
        comment => 'West Shipping Multi Consumer Orders queue table',
        multiple_consumers => TRUE,
        queue_payload_type => 'BOLADM.order_typ',
        compatible => '8.1');
end;
/

Rem Booked orders are stored in the priority queue table
begin
dbms_aqadm.create_queue (
        queue_name             => 'WS_bookedorders_que',
        queue_table            => 'WS_orders_pr_mqtab');
end;
/

Rem Shipped orders and back orders are stored in the FIFO queue table
begin
dbms_aqadm.create_queue (
        queue_name             => 'WS_shippedorders_que',
        queue_table            => 'WS_orders_mqtab');
end;
/


begin
dbms_aqadm.create_queue (
```

```
        queue_name              => 'WS_backorders_que',
        queue_table             => 'WS_orders_mqtab');
end;
/

Rem
Rem  In order to test history, set retention to 1 DAY for the queues
Rem  in WS

begin
dbms_aqadm.alter_queue(
         queue_name => 'WS_bookedorders_que',
         retention_time => 86400);
end;
/

begin
dbms_aqadm.alter_queue(
         queue_name => 'WS_shippedorders_que',
         retention_time => 86400);
end;
/

begin
dbms_aqadm.alter_queue(
         queue_name => 'WS_backorders_que',
         retention_time => 86400);
end;
/


Rem Create queue tables, queues for ES Shipping
connect ES/ES;

Rem Create a priority queue table for ES shipping
begin
dbms_aqadm.create_queue_table(
        queue_table => 'ES_orders_mqtab',
        comment => 'East Shipping Multi Consumer Orders queue table',
        multiple_consumers => TRUE,
        queue_payload_type => 'BOLADM.order_typ',
        compatible => '8.1');
end;
/
```

```
Rem Create a FIFO queue tables for ES shipping
begin
dbms_aqadm.create_queue_table(
        queue_table => 'ES_orders_pr_mqtab',
        sort_list =>'priority,enq_time',
        comment => 'East Shipping Priority Multi Consumer Orders queue table',
        multiple_consumers => TRUE,
        queue_payload_type => 'BOLADM.order_typ',
        compatible => '8.1');
end;
/

Rem Booked orders are stored in the priority queue table
begin
dbms_aqadm.create_queue (
        queue_name             => 'ES_bookedorders_que',
        queue_table            => 'ES_orders_pr_mqtab');
end;
/

Rem Shipped orders and back orders are stored in the FIFO queue table
begin
dbms_aqadm.create_queue (
        queue_name             => 'ES_shippedorders_que',
        queue_table            => 'ES_orders_mqtab');
end;
/

begin
dbms_aqadm.create_queue (
        queue_name             => 'ES_backorders_que',
        queue_table            => 'ES_orders_mqtab');
end;
/


Rem Create queue tables, queues for Overseas Shipping
connect OS/OS;

Rem Create a priority queue table for OS shipping
begin
dbms_aqadm.create_queue_table(
        queue_table => 'OS_orders_pr_mqtab',
        sort_list =>'priority,enq_time',
        comment => 'Overseas Shipping Priority MultiConsumer Orders queue
```

```
table',
        multiple_consumers => TRUE,
        queue_payload_type => 'BOLADM.order_typ',
        compatible => '8.1');
end;
/

Rem Create a FIFO queue tables for OS shipping
begin
dbms_aqadm.create_queue_table(
        queue_table => 'OS_orders_mqtab',
        comment => 'Overseas Shipping Multi Consumer Orders queue table',
        multiple_consumers => TRUE,
        queue_payload_type => 'BOLADM.order_typ',
        compatible => '8.1');
end;
/

Rem Booked orders are stored in the priority queue table
begin
dbms_aqadm.create_queue (
        queue_name              => 'OS_bookedorders_que',
        queue_table             => 'OS_orders_pr_mqtab');
end;
/

Rem Shipped orders and back orders are stored in the FIFO queue table
begin
dbms_aqadm.create_queue (
        queue_name              => 'OS_shippedorders_que',
        queue_table             => 'OS_orders_mqtab');
end;
/

begin
dbms_aqadm.create_queue (
        queue_name              => 'OS_backorders_que',
        queue_table             => 'OS_orders_mqtab');
end;
/


Rem Create queue tables, queues for Customer Billing
connect CBADM/CBADM;
begin
```

```
dbms_aqadm.create_queue_table(
        queue_table => 'CBADM_orders_sqtab',
        comment => 'Customer Billing Single Consumer Orders queue table',
        queue_payload_type => 'BOLADM.order_typ',
        compatible => '8.1');

dbms_aqadm.create_queue_table(
        queue_table => 'CBADM_orders_mqtab',
        comment => 'Customer Billing Multi Consumer Service queue table',
        multiple_consumers => TRUE,
        queue_payload_type => 'BOLADM.order_typ',
        compatible => '8.1');

dbms_aqadm.create_queue (
        queue_name              => 'CBADM_shippedorders_que',
        queue_table             => 'CBADM_orders_sqtab');

end;
/

Rem Grant dequeue privilege on the shopped orders queue to the Customer Billing
Rem application.  The CB application retrieves shipped orders (not billed yet)
Rem from the shopped orders queue.
execute dbms_aqadm.grant_queue_privilege('DEQUEUE', 'CBADM_shippedorders_que',
'CB', FALSE);

begin
dbms_aqadm.create_queue (
        queue_name              => 'CBADM_billedorders_que',
        queue_table             => 'CBADM_orders_mqtab');
end;
/

Rem Grant enqueue privilege on the billed orders queue to Customer Billing
Rem application.  The CB application is allowed to put billed orders into
Rem this queue.
execute dbms_aqadm.grant_queue_privilege('ENQUEUE', 'CBADM_billedorders_que',
'CB', FALSE);


Rem Customer support tracks the state of the customer request in the system
Rem
Rem At any point, customer request can be in one of the following states
Rem  A. BOOKED  B. SHIPPED  C. BACKED   D. BILLED
Rem  Given the order number the customer support will return the state
```

```
Rem  the order is in. This state is maintained in the order_status_table

connect CS/CS;

CREATE TABLE Order_Status_Table(customer_order       boladm.order_typ,
                                status               varchar2(30));

Rem Create queue tables, queues for Customer Service

begin
dbms_aqadm.create_queue_table(
        queue_table => 'CS_order_status_qt',
        comment => 'Customer Status multi consumer queue table',
        multiple_consumers => TRUE,
        queue_payload_type => 'BOLADM.order_typ',
        compatible => '8.1');

dbms_aqadm.create_queue (
        queue_name              => 'CS_bookedorders_que',
        queue_table             => 'CS_order_status_qt');

dbms_aqadm.create_queue (
        queue_name              => 'CS_backorders_que',
        queue_table             => 'CS_order_status_qt');

dbms_aqadm.create_queue (
        queue_name              => 'CS_shippedorders_que',
        queue_table             => 'CS_order_status_qt');

dbms_aqadm.create_queue (
        queue_name              => 'CS_billedorders_que',
        queue_table             => 'CS_order_status_qt');

end;
/

Rem Create the Subscribers for OE queues
Rem Add the Subscribers for the OE booked_orders queue

connect OE/OE;

Rem Add a rule-based subscriber for West Shipping
Rem West Shipping handles Western region US orders
Rem Rush Western region orders are handled by East Shipping
declare
```

```
  subscriber     aq$_agent;
begin
  subscriber := aq$_agent('West_Shipping', 'WS.WS_bookedorders_que', null);
  dbms_aqadm.add_subscriber(queue_name => 'OE.OE_bookedorders_que',
                            subscriber => subscriber,
                            rule       => 'tab.user_data.orderregion =
''WESTERN'' AND tab.user_data.ordertype != ''RUSH''');
end;
/

Rem Add a rule-based subscriber for East Shipping
Rem East shipping handles all Eastern region orders
Rem East shipping also handles all US rush orders
declare
  subscriber     aq$_agent;
begin
  subscriber := aq$_agent('East_Shipping', 'ES.ES_bookedorders_que', null);
  dbms_aqadm.add_subscriber(queue_name => 'OE.OE_bookedorders_que',
                            subscriber => subscriber,
                            rule       => 'tab.user_data.orderregion =
''EASTERN'' OR (tab.user_data.ordertype = ''RUSH'' AND tab.user_
data.customer.country = ''USA'') ');
end;
/

Rem Add a rule-based subscriber for Overseas Shipping
Rem Intl Shipping handles all non-US orders
declare
  subscriber     aq$_agent;
begin
  subscriber := aq$_agent('Overseas_Shipping', 'OS.OS_bookedorders_que', null);
  dbms_aqadm.add_subscriber(queue_name => 'OE.OE_bookedorders_que',
                            subscriber => subscriber,
                            rule       => 'tab.user_data.orderregion =
''INTERNATIONAL''');
end;
/

Rem Add the Customer Service order queues as a subscribers to the
Rem corresponding queues  in OrderEntry, Shipping and Billing

declare
  subscriber     aq$_agent;
begin
  /* Subscribe to the booked orders queue */
```

```
  subscriber := aq$_agent('BOOKED_ORDER', 'CS.CS_bookedorders_que', null);
  dbms_aqadm.add_subscriber(queue_name => 'OE.OE_bookedorders_que',
                            subscriber => subscriber);
end;
/


connect WS/WS;

declare
  subscriber     aq$_agent;
begin
  /* Subscribe to the WS back orders queue */
  subscriber := aq$_agent('BACK_ORDER', 'CS.CS_backorders_que', null);
  dbms_aqadm.add_subscriber(queue_name => 'WS.WS_backorders_que',
                            subscriber => subscriber);
end;
/

declare
  subscriber     aq$_agent;
begin
  /* Subscribe to the WS shipped orders queue */
  subscriber := aq$_agent('SHIPPED_ORDER', 'CS.CS_shippedorders_que', null);
  dbms_aqadm.add_subscriber(queue_name => 'WS.WS_shippedorders_que',
                            subscriber => subscriber);
end;
/

connect CBADM/CBADM;
declare
  subscriber     aq$_agent;
begin
  /* Subscribe to the BILLING billed orders queue */
  subscriber := aq$_agent('BILLED_ORDER', 'CS.CS_billedorders_que', null);
  dbms_aqadm.add_subscriber(queue_name => 'CBADM.CBADM_billedorders_que',
                            subscriber => subscriber);

end;
/


Rem
Rem BOLADM will Start all the queues
Rem
```

```
connect BOLADM/BOLADM
execute dbms_aqadm.start_queue(queue_name => 'OE.OE_neworders_que');
execute dbms_aqadm.start_queue(queue_name => 'OE.OE_bookedorders_que');
execute dbms_aqadm.start_queue(queue_name => 'WS.WS_bookedorders_que');
execute dbms_aqadm.start_queue(queue_name => 'WS.WS_shippedorders_que');
execute dbms_aqadm.start_queue(queue_name => 'WS.WS_backorders_que');
execute dbms_aqadm.start_queue(queue_name => 'ES.ES_bookedorders_que');
execute dbms_aqadm.start_queue(queue_name => 'ES.ES_shippedorders_que');
execute dbms_aqadm.start_queue(queue_name => 'ES.ES_backorders_que');
execute dbms_aqadm.start_queue(queue_name => 'OS.OS_bookedorders_que');
execute dbms_aqadm.start_queue(queue_name => 'OS.OS_shippedorders_que');
execute dbms_aqadm.start_queue(queue_name => 'OS.OS_backorders_que');
execute dbms_aqadm.start_queue(queue_name => 'CBADM.CBADM_shippedorders_que');
execute dbms_aqadm.start_queue(queue_name => 'CBADM.CBADM_billedorders_que');
execute dbms_aqadm.start_queue(queue_name => 'CS.CS_bookedorders_que');
execute dbms_aqadm.start_queue(queue_name => 'CS.CS_backorders_que');
execute dbms_aqadm.start_queue(queue_name => 'CS.CS_shippedorders_que');
execute dbms_aqadm.start_queue(queue_name => 'CS.CS_billedorders_que');

connect system/manager

Rem
Rem Start job_queue_processes to handle AQ propagation
Rem

alter system set job_queue_processes=4;
```

tkaqdocd.sql: Examples of Administrative and Operational Interfaces

```
Rem
Rem $Header: tkaqdocd.sql 26-jan-99.17:51:23 aquser1 Exp $
Rem
Rem tkaqdocd.sql
Rem
Rem  Copyright (c) Oracle Corporation 1998, 1999. All Rights Reserved.
Rem
Rem    NAME
Rem      tkaqdocd.sql - <one-line expansion of the name>
Rem
Rem    DESCRIPTION
Rem      <short description of component this file declares/defines>
Rem
Rem    NOTES
Rem      <other useful comments, qualifications, etc.>
Rem
Rem    MODIFIED   (MM/DD/YY)
Rem    aquser1     01/26/99 - fix comments
Rem    aquser1     12/07/98 - ryaseen: convert to SQLPLUS format
Rem    aquser1     10/29/98 - adjust agent list and update_status
Rem    aquser1     10/27/98 - listen call, history and non-persistent queues
Rem    aquser1     10/27/98 - Created
Rem


Rem
Rem  Schedule propagation for the shipping, billing, order entry queues
Rem

connect OE/OE;

execute dbms_aqadm.schedule_propagation(queue_name => 'OE.OE_bookedorders_que');

connect WS/WS;
execute dbms_aqadm.schedule_propagation(queue_name => 'WS.WS_backorders_que');
execute dbms_aqadm.schedule_propagation(queue_name => 'WS.WS_shippedorders_
que');

connect CBADM/CBADM;
execute dbms_aqadm.schedule_propagation(queue_name => 'CBADM.CBADM_billedorders_
que');
```

```
Rem
Rem   Customer service application
Rem
Rem   This application monitors the status queue for messages and updates
Rem   the Order_Status table.


connect CS/CS

Rem
Rem  Dequeus messages from the 'queue' for 'consumer'

CREATE OR REPLACE PROCEDURE DEQUEUE_MESSAGE(
                         queue      IN   VARCHAR2,
                         consumer   IN   VARCHAR2,
                         message    OUT  BOLADM.order_typ)
IS

dopt                     dbms_aq.dequeue_options_t;
mprop                    dbms_aq.message_properties_t;
deq_msgid                raw(16);
BEGIN
  dopt.dequeue_mode := dbms_aq.REMOVE;
  dopt.navigation := dbms_aq.FIRST_MESSAGE;
  dopt.consumer_name := consumer;

  dbms_aq.dequeue(
                queue_name => queue,
                dequeue_options => dopt,
                message_properties => mprop,
                payload => message,
                msgid => deq_msgid);
  commit;
END;
/



Rem
Rem  Updates the status of the order in the status table
Rem

CREATE OR REPLACE PROCEDURE update_status(
                                new_status    IN VARCHAR2,
                                order_msg    IN BOLADM.ORDER_TYP)
```

```
IS
 old_status    VARCHAR2(30);
 dummy         NUMBER;
BEGIN

  BEGIN
    /* query old status from the table */
    SELECT st.status INTO old_status from order_status_table st
       where st.customer_order.orderno = order_msg.orderno;

  /* Status can be 'BOOKED_ORDER', 'SHIPPED_ORDER', 'BACK_ORDER'
   *          and   'BILLED_ORDER'
   */

   IF new_status = 'SHIPPED_ORDER' THEN
      IF old_status = 'BILLED_ORDER' THEN
        return;             /* message about a previous state */
      END IF;
   ELSIF new_status = 'BACK_ORDER' THEN
      IF old_status = 'SHIPPED_ORDER' OR old_status = 'BILLED_ORDER' THEN
        return;             /* message about a previous state */
      END IF;
   END IF;

   /* update the order status  */
     UPDATE order_status_table st
        SET st.customer_order = order_msg, st.status = new_status
        where st.customer_order.orderno = order_msg.orderno;

   COMMIT;

  EXCEPTION
  WHEN OTHERS  THEN     /* change to no data found */
    /* first update for the order */
    INSERT INTO order_status_table(customer_order, status)
    VALUES (order_msg, new_status);
    COMMIT;

  END;
END;
/


Rem
Rem  Monitors the customer service queues for 'time' seconds
```

```
Rem

 CREATE OR REPLACE PROCEDURE MONITOR_STATUS_QUEUE(time  IN  NUMBER)
IS
  agent_w_message   aq$_agent;
  agent_list        dbms_aq.agent_list_t;
  wait_time         INTEGER := 120;
  no_message        EXCEPTION;
  pragma EXCEPTION_INIT(no_message, -25254);
  order_msg         boladm.order_typ;
  new_status        VARCHAR2(30);
  monitor           BOOLEAN := TRUE;
  begin_time        number;
  end_time          number;
BEGIN

 begin_time :=  dbms_utility.get_time;
 WHILE (monitor)
 LOOP
 BEGIN
  agent_list(1) := aq$_agent('BILLED_ORDER', 'CS_billedorders_que', NULL);
  agent_list(2) := aq$_agent('SHIPPED_ORDER', 'CS_shippedorders_que', NULL);
  agent_list(3) := aq$_agent('BACK_ORDER', 'CS_backorders_que', NULL);
  agent_list(4) := aq$_agent('Booked_ORDER', 'CS_bookedorders_que', NULL);

   /* wait for order status messages */
   dbms_aq.listen(agent_list, wait_time, agent_w_message);

   dbms_output.put_line('Agent' || agent_w_message.name || ' Address '|| agent_
w_message.address);
   /* dequeue the message from the queue */
   dequeue_message(agent_w_message.address, agent_w_message.name, order_msg);

   /* update the status of the order depending on the type of the message
    * the name of the agent contains the new state
    */
   update_status(agent_w_message.name, order_msg);

  /* exit if we have been working long enough */
   end_time := dbms_utility.get_time;
   IF  (end_time - begin_time > time)   THEN
     EXIT;
   END IF;

  EXCEPTION
```

```
  WHEN  no_message  THEN
    dbms_output.put_line('No messages in the past 2 minutes');
       end_time := dbms_utility.get_time;
    /* exit if we have done enough work */
    IF  (end_time - begin_time > time)   THEN
      EXIT;
    END IF;
  END;

  END LOOP;
END;
/




Rem
Rem   History queries
Rem

Rem
Rem   Average processing time for messages in western shipping:
Rem   Difference between the ship- time and book-time for the order
Rem
Rem   NOTE: we assume that order id is the correlation identifier
Rem         Only processed messages are considered.

Connect WS/WS


SELECT  SUM(SO.enq_time - BO.enq_time) / count (*) AVG_PRCS_TIME
 FROM WS.AQ$WS_orders_pr_mqtab BO , WS.AQ$WS_orders_mqtab SO
 WHERE SO.msg_state = 'PROCESSED' and BO.msg_state = 'PROCESSED'
 AND SO.corr_id = BO.corr_id and SO.queue = 'WS_shippedorders_que';


Rem
Rem  Average backed up time (again only processed messages are considered
Rem

SELECT SUM(BACK.deq_time - BACK.enq_time)/count (*) AVG_BACK_TIME
  FROM WS.AQ$WS_orders_mqtab BACK
  WHERE BACK.msg_state = 'PROCESSED' and BACK.queue = 'WS_backorders_que';
```

tkaqdoce.sql: Operational Examples

```
Rem
Rem $Header: tkaqdoce.sql 26-jan-99.17:51:28 aquser1 Exp $
Rem
Rem tkaqdocl.sql
Rem
Rem  Copyright (c) Oracle Corporation 1998, 1999. All Rights Reserved.
Rem


set echo on

Rem ==================================================================
Rem       Demonstrate enqueuing a backorder with delay time set
Rem       to 1 day.  This will guarantee that each backorder will
Rem       be processed only once a day until the order is filled.
Rem ==================================================================

Rem Create a package that enqueue with delay set to one day
connect BOLADM/BOLADM
create or replace procedure requeue_unfilled_order(sale_region varchar2,
                                                   backorder order_typ)
as
  back_order_queue_name    varchar2(62);
  enqopt                   dbms_aq.enqueue_options_t;
  msgprop                  dbms_aq.message_properties_t;
  enq_msgid                raw(16);
begin
  -- Choose a back order queue based the the region
  IF sale_region = 'WEST' THEN
    back_order_queue_name := 'WS.WS_backorders_que';
  ELSIF sale_region = 'EAST' THEN
    back_order_queue_name := 'ES.ES_backorders_que';
  ELSE
    back_order_queue_name := 'OS.OS_backorders_que';
  END IF;

  -- Enqueue the order with delay time set to 1 day
  msgprop.delay := 60*60*24;
  dbms_aq.enqueue(back_order_queue_name, enqopt, msgprop,
                  backorder, enq_msgid);
end;
```

tkaqdocp.sql: Examples of Operational Interfaces

```
Rem
Rem $Header: tkaqdocp.sql 26-jan-99.17:50:54 aquser1 Exp $
Rem
Rem tkaqdocp.sql
Rem
Rem  Copyright (c) Oracle Corporation 1998, 1999. All Rights Reserved.
Rem
Rem    NAME
Rem      tkaqdocp.sql - <one-line expansion of the name>
Rem

set echo on;

Rem =================================================================
Rem                       Illustrating Support for OPS
Rem =================================================================

Rem Login into OE account
connect OE/OE;
set serveroutput on;

Rem check instance affinity of OE queue tables from AQ administrative view

select queue_table, primary_instance, secondary_instance, owner_instance
from user_queue_tables;

Rem alter instance affinity of OE queue tables

begin
dbms_aqadm.alter_queue_table(
        queue_table => 'OE.OE_orders_sqtab',
        primary_instance => 2,
        secondary_instance => 1);
end;
/

begin
dbms_aqadm.alter_queue_table(
        queue_table => 'OE.OE_orders_pr_mqtab',
        primary_instance => 1,
        secondary_instance => 2);
end;
/
```

```
Rem check instance affinity of OE queue tables from AQ administrative view

select queue_table, primary_instance, secondary_instance, owner_instance
from user_queue_tables;

Rem ==================================================================
Rem                     Illustrating Propagation Scheduling
Rem ==================================================================

Rem Login into OE account

set echo on;
connect OE/OE;
set serveroutput on;

Rem
Rem Schedule Propagation from bookedorders_que to shipping
Rem

execute dbms_aqadm.schedule_propagation(queue_name => 'OE.OE_bookedorders_que');

Rem Login into boladm account
set echo on;
connect boladm/boladm;
set serveroutput on;

Rem create a procedure to enqueue an order
create or replace procedure order_enq(book_title   in varchar2,
                                      book_qty     in number,
                                      order_num    in number,
                                      shipping_priority in number,
                                      cust_state   in varchar2,
                                      cust_country in varchar2,
                                      cust_region  in varchar2,
                                      cust_ord_typ in varchar2) as

OE_enq_order_data        BOLADM.order_typ;
OE_enq_cust_data         BOLADM.customer_typ;
OE_enq_book_data         BOLADM.book_typ;
OE_enq_item_data         BOLADM.orderitem_typ;
OE_enq_item_list         BOLADM.orderitemlist_vartyp;
enqopt                   dbms_aq.enqueue_options_t;
msgprop                  dbms_aq.message_properties_t;
enq_msgid                raw(16);
```

```
begin

        msgprop.correlation := cust_ord_typ;
        OE_enq_cust_data := BOLADM.customer_typ(NULL, NULL, NULL, NULL,
                                cust_state, NULL, cust_country);
        OE_enq_book_data := BOLADM.book_typ(book_title, NULL, NULL, NULL);
        OE_enq_item_data := BOLADM.orderitem_typ(book_qty,
                                OE_enq_book_data, NULL);
        OE_enq_item_list := BOLADM.orderitemlist_vartyp(
                                BOLADM.orderitem_typ(book_qty,
                                OE_enq_book_data, NULL));
        OE_enq_order_data := BOLADM.order_typ(order_num, NULL,
                                cust_ord_typ, cust_region,
                                OE_enq_cust_data, NULL,
                                OE_enq_item_list, NULL);

        -- Put the shipping priority into message property before
        -- enqueueing the message
        msgprop.priority := shipping_priority;
        dbms_aq.enqueue('OE.OE_bookedorders_que', enqopt, msgprop,
                        OE_enq_order_data, enq_msgid);
end;
/

show errors;

grant execute on order_enq to OE;

Rem now create a procedure to dequeue booked orders for shipment processing
create or replace procedure shipping_bookedorder_deq(
                                        consumer in varchar2,
                                        deqmode in binary_integer) as

deq_cust_data           BOLADM.customer_typ;
deq_book_data           BOLADM.book_typ;
deq_item_data           BOLADM.orderitem_typ;
deq_msgid               RAW(16);
dopt                    dbms_aq.dequeue_options_t;
mprop                   dbms_aq.message_properties_t;
deq_order_data          BOLADM.order_typ;
qname                   varchar2(30);
no_messages             exception;
pragma exception_init   (no_messages, -25228);
new_orders              BOOLEAN := TRUE;
```

```
begin

        dopt.consumer_name := consumer;
        dopt.wait := DBMS_AQ.NO_WAIT;
        dopt.dequeue_mode := deqmode;
        dopt.navigation := dbms_aq.FIRST_MESSAGE;

        IF (consumer = 'West_Shipping') THEN
                qname := 'WS.WS_bookedorders_que';
        ELSIF (consumer = 'East_Shipping') THEN
                qname := 'ES.ES_bookedorders_que';
        ELSE
                qname := 'OS.OS_bookedorders_que';
        END IF;

        WHILE (new_orders) LOOP
          BEGIN
            dbms_aq.dequeue(
                queue_name => qname,
                dequeue_options => dopt,
                message_properties => mprop,
                payload => deq_order_data,
                msgid => deq_msgid);

            deq_item_data := deq_order_data.items(1);
            deq_book_data := deq_item_data.item;
            deq_cust_data := deq_order_data.customer;

            dbms_output.put_line(' **** next booked order **** ');
            dbms_output.put_line('order_num: ' || deq_order_data.orderno ||
                        ' book_title: ' || deq_book_data.title ||
                        ' quantity: ' || deq_item_data.quantity);
            dbms_output.put_line('ship_state: ' || deq_cust_data.state ||
                        ' ship_country: ' || deq_cust_data.country ||
                        ' ship_order_type: ' || deq_order_data.ordertype);
            dopt.navigation := dbms_aq.NEXT_MESSAGE;
          EXCEPTION
            WHEN no_messages THEN
                 dbms_output.put_line (' ---- NO MORE BOOKED ORDERS ---- ');
                 new_orders := FALSE;
          END;
        END LOOP;

end;
```

```
/
show errors;

Rem now create a procedure to dequeue rush orders for shipment
create or replace procedure get_rushtitles(consumer in varchar2) as

deq_cust_data           BOLADM.customer_typ;
deq_book_data           BOLADM.book_typ;
deq_item_data           BOLADM.orderitem_typ;
deq_msgid               RAW(16);
dopt                    dbms_aq.dequeue_options_t;
mprop                   dbms_aq.message_properties_t;
deq_order_data          BOLADM.order_typ;
qname                   varchar2(30);
no_messages             exception;
pragma exception_init    (no_messages, -25228);
new_orders              BOOLEAN := TRUE;

begin

        dopt.consumer_name := consumer;
        dopt.wait := 1;
        dopt.correlation := 'RUSH';

        IF (consumer = 'West_Shipping') THEN
                qname := 'WS.WS_bookedorders_que';
        ELSIF (consumer = 'East_Shipping') THEN
                qname := 'ES.ES_bookedorders_que';
        ELSE
                qname := 'OS.OS_bookedorders_que';
        END IF;

        WHILE (new_orders) LOOP
          BEGIN
            dbms_aq.dequeue(
                queue_name => qname,
                dequeue_options => dopt,
                message_properties => mprop,
                payload => deq_order_data,
                msgid => deq_msgid);

            deq_item_data := deq_order_data.items(1);
            deq_book_data := deq_item_data.item;

            dbms_output.put_line(' rushorder book_title: ' ||
```

```
                                deq_book_data.title ||
                        ' quantity: ' || deq_item_data.quantity);
          EXCEPTION
            WHEN no_messages THEN
                 dbms_output.put_line (' ---- NO MORE RUSH TITLES ---- ');
                 new_orders := FALSE;
          END;
        END LOOP;

end;
/
show errors;

Rem now create a procedure to dequeue orders for handling North American
Rem orders
create or replace procedure get_northamerican_orders as

deq_cust_data            BOLADM.customer_typ;
deq_book_data            BOLADM.book_typ;
deq_item_data            BOLADM.orderitem_typ;
deq_msgid                RAW(16);
dopt                     dbms_aq.dequeue_options_t;
mprop                    dbms_aq.message_properties_t;
deq_order_data           BOLADM.order_typ;
deq_order_nodata         BOLADM.order_typ;
qname                    varchar2(30);
no_messages              exception;
pragma exception_init    (no_messages, -25228);
new_orders               BOOLEAN := TRUE;

begin

        dopt.consumer_name := 'Overseas_Shipping';
        dopt.wait := DBMS_AQ.NO_WAIT;
        dopt.navigation := dbms_aq.FIRST_MESSAGE;
        dopt.dequeue_mode := DBMS_AQ.LOCKED;

        qname := 'OS.OS_bookedorders_que';

        WHILE (new_orders) LOOP
          BEGIN
            dbms_aq.dequeue(
                queue_name => qname,
                dequeue_options => dopt,
                message_properties => mprop,
```

```
                payload => deq_order_data,
                msgid => deq_msgid);

            deq_item_data := deq_order_data.items(1);
            deq_book_data := deq_item_data.item;
            deq_cust_data := deq_order_data.customer;

            IF (deq_cust_data.country = 'Canada' OR
                deq_cust_data.country = 'Mexico' ) THEN

                dopt.dequeue_mode := dbms_aq.REMOVE_NODATA;
                dopt.msgid := deq_msgid;
                dbms_aq.dequeue(
                        queue_name => qname,
                        dequeue_options => dopt,
                        message_properties => mprop,
                        payload => deq_order_nodata,
                        msgid => deq_msgid);

                dbms_output.put_line(' **** next booked order **** ');
                dbms_output.put_line('order_no: ' || deq_order_data.orderno ||
                        ' book_title: ' || deq_book_data.title ||
                        ' quantity: ' || deq_item_data.quantity);
                dbms_output.put_line('ship_state: ' || deq_cust_data.state ||
                        ' ship_country: ' || deq_cust_data.country ||
                        ' ship_order_type: ' || deq_order_data.ordertype);

            END IF;

            commit;
            dopt.dequeue_mode := DBMS_AQ.LOCKED;
            dopt.msgid := NULL;
            dopt.navigation := dbms_aq.NEXT_MESSAGE;
          EXCEPTION
            WHEN no_messages THEN
                 dbms_output.put_line (' ---- NO MORE BOOKED ORDERS ---- ');
                 new_orders := FALSE;
          END;
        END LOOP;

end;
/
show errors;

grant execute on shipping_bookedorder_deq to WS;
```

```
grant execute on shipping_bookedorder_deq to ES;
grant execute on shipping_bookedorder_deq to OS;
grant execute on shipping_bookedorder_deq to CS;

grant execute on get_rushtitles to ES;

grant execute on get_northamerican_orders to OS;

Rem Login into OE account
connect OE/OE;
set serveroutput on;

Rem
Rem Enqueue some orders into OE_bookedorders_que
Rem

execute BOLADM.order_enq('My First   Book', 1, 1001, 3,'CA', 'USA', 'WESTERN',
'NORMAL');
execute BOLADM.order_enq('My Second  Book', 2, 1002, 3,'NY', 'USA', 'EASTERN',
'NORMAL');
execute BOLADM.order_enq('My Third   Book', 3, 1003, 3, '',   'Canada',
'INTERNATIONAL', 'NORMAL');
execute BOLADM.order_enq('My Fourth  Book', 4, 1004, 2, 'NV', 'USA', 'WESTERN',
'RUSH');
execute BOLADM.order_enq('My Fifth   Book', 5, 1005, 2, 'MA', 'USA', 'EASTERN',
'RUSH');
execute BOLADM.order_enq('My Sixth   Book', 6, 1006, 3,''  , 'UK',
'INTERNATIONAL', 'NORMAL');
execute BOLADM.order_enq('My Seventh Book', 7, 1007, 1,'',   'Canada',
'INTERNATIONAL', 'RUSH');
execute BOLADM.order_enq('My Eighth  Book', 8, 1008, 3,'',   'Mexico',
'INTERNATIONAL', 'NORMAL');
execute BOLADM.order_enq('My Ninth   Book', 9, 1009, 1, 'CA', 'USA', 'WESTERN',
'RUSH');
execute BOLADM.order_enq('My Tenth   Book', 8, 1010, 3, ''  , 'UK',
'INTERNATIONAL', 'NORMAL');
execute BOLADM.order_enq('My Last    Book', 7, 1011, 3, ''  , 'Mexico',
'INTERNATIONAL', 'NORMAL');
commit;
/

Rem
Rem Wait for Propagation to Complete
Rem
```

```
execute dbms_lock.sleep(100);

Rem ====================================================================
Rem                  Illustrating Dequeue Modes/Methods
Rem ====================================================================


connect WS/WS;
set serveroutput on;

Rem Dequeue all booked orders for West_Shipping
execute BOLADM.shipping_bookedorder_deq('West_Shipping', DBMS_AQ.REMOVE);
commit;
/

connect ES/ES;
set serveroutput on;

Rem Browse all booked orders for East_Shipping
execute BOLADM.shipping_bookedorder_deq('East_Shipping', DBMS_AQ.BROWSE);

Rem Dequeue all rush order titles for East_Shipping
execute BOLADM.get_rushtitles('East_Shipping');
commit;
/

Rem Dequeue all remaining booked orders (normal order) for East_Shipping
execute BOLADM.shipping_bookedorder_deq('East_Shipping', DBMS_AQ.REMOVE);
commit;
/

connect OS/OS;
set serveroutput on;

Rem Dequeue all international North American orders for Overseas_Shipping
execute BOLADM.get_northamerican_orders;
commit;
/

Rem Dequeue rest of the booked orders for Overseas_Shipping
execute BOLADM.shipping_bookedorder_deq('Overseas_Shipping', DBMS_AQ.REMOVE);
commit;
/
```

```
Rem ======================================================================
Rem           Illustrating Enhanced Propagation Capabilities
Rem ======================================================================

connect OE/OE;
set serveroutput on;

Rem
Rem Get propagation schedule information & statistics
Rem

Rem get averages
select avg_time, avg_number, avg_size from user_queue_schedules;

Rem get totals
select total_time, total_number, total_bytes from user_queue_schedules;

Rem get status information of schedule (present only when active)
select process_name, session_id, instance, schedule_disabled
        from user_queue_schedules;

Rem get information about last and next execution
select last_run_date, last_run_time, next_run_date, next_run_time
        from user_queue_schedules;

Rem get last error information if any
select failures, last_error_msg, last_error_date, last_error_time
        from user_queue_schedules;

Rem disable propagation schedule for booked orders

execute dbms_aqadm.disable_propagation_schedule(queue_name => 'OE_bookedorders_
que');
execute dbms_lock.sleep(30);
select schedule_disabled from user_queue_schedules;

Rem alter propagation schedule for booked orders to execute every
Rem 15 mins (900 seconds) for a window duration of 300 seconds

begin
dbms_aqadm.alter_propagation_schedule(
        queue_name => 'OE_bookedorders_que',
        duration => 300,
        next_time => 'SYSDATE + 900/86400',
        latency => 25);
```

```
end;
/

execute dbms_lock.sleep(30);
select next_time, latency, propagation_window from user_queue_schedules;

Rem enable propagation schedule for booked orders

execute dbms_aqadm.enable_propagation_schedule(queue_name => 'OE_bookedorders_
que');
execute dbms_lock.sleep(30);
select schedule_disabled from user_queue_schedules;

Rem unschedule propagation for booked orders

execute dbms_aqadm.unschedule_propagation(queue_name => 'OE.OE_bookedorders_
que');

set echo on;

Rem ====================================================================
Rem                       Illustrating Message Grouping
Rem ====================================================================

Rem Login into boladm account
set echo on;
connect boladm/boladm;
set serveroutput on;

Rem now create a procedure to handle order entry
create or replace procedure new_order_enq(book_title   in varchar2,
                                          book_qty     in number,
                                          order_num    in number,
                                          cust_state   in varchar2) as

OE_enq_order_data        BOLADM.order_typ;
OE_enq_cust_data         BOLADM.customer_typ;
OE_enq_book_data         BOLADM.book_typ;
OE_enq_item_data         BOLADM.orderitem_typ;
OE_enq_item_list         BOLADM.orderitemlist_vartyp;
enqopt                   dbms_aq.enqueue_options_t;
msgprop                  dbms_aq.message_properties_t;
enq_msgid                raw(16);

begin
```

```
        OE_enq_cust_data := BOLADM.customer_typ(NULL, NULL, NULL, NULL,
                                cust_state, NULL, NULL);
        OE_enq_book_data := BOLADM.book_typ(book_title, NULL, NULL, NULL);
        OE_enq_item_data := BOLADM.orderitem_typ(book_qty,
                                OE_enq_book_data, NULL);
        OE_enq_item_list := BOLADM.orderitemlist_vartyp(
                                BOLADM.orderitem_typ(book_qty,
                                OE_enq_book_data, NULL));
        OE_enq_order_data := BOLADM.order_typ(order_num, NULL,
                                NULL, NULL,
                                OE_enq_cust_data, NULL,
                                OE_enq_item_list, NULL);
        dbms_aq.enqueue('OE.OE_neworders_que', enqopt, msgprop,
                        OE_enq_order_data, enq_msgid);
end;
/
show errors;

Rem now create a procedure to handle order enqueue
create or replace procedure same_order_enq(book_title   in varchar2,
                                           book_qty     in number) as

OE_enq_order_data       BOLADM.order_typ;
OE_enq_book_data        BOLADM.book_typ;
OE_enq_item_data        BOLADM.orderitem_typ;
OE_enq_item_list        BOLADM.orderitemlist_vartyp;
enqopt                  dbms_aq.enqueue_options_t;
msgprop                 dbms_aq.message_properties_t;
enq_msgid               raw(16);

begin

        OE_enq_book_data := BOLADM.book_typ(book_title, NULL, NULL, NULL);
        OE_enq_item_data := BOLADM.orderitem_typ(book_qty,
                                OE_enq_book_data, NULL);
        OE_enq_item_list := BOLADM.orderitemlist_vartyp(
                                BOLADM.orderitem_typ(book_qty,
                                OE_enq_book_data, NULL));
        OE_enq_order_data := BOLADM.order_typ(NULL, NULL,
                                NULL, NULL,
                                NULL, NULL,
                                OE_enq_item_list, NULL);
        dbms_aq.enqueue('OE.OE_neworders_que', enqopt, msgprop,
                        OE_enq_order_data, enq_msgid);
```

```
end;
/
show errors;

grant execute on new_order_enq to OE;
grant execute on same_order_enq to OE;

Rem now create a procedure to get new orders by dequeuing
create or replace procedure get_new_orders as

deq_cust_data           BOLADM.customer_typ;
deq_book_data           BOLADM.book_typ;
deq_item_data           BOLADM.orderitem_typ;
deq_msgid               RAW(16);
dopt                    dbms_aq.dequeue_options_t;
mprop                   dbms_aq.message_properties_t;
deq_order_data          BOLADM.order_typ;
qname                   varchar2(30);
no_messages             exception;
end_of_group            exception;
pragma exception_init   (no_messages, -25228);
pragma exception_init   (end_of_group, -25235);
new_orders              BOOLEAN := TRUE;

begin

        dopt.wait := 1;
        dopt.navigation := DBMS_AQ.FIRST_MESSAGE;
        qname := 'OE.OE_neworders_que';
        WHILE (new_orders) LOOP
          BEGIN
            LOOP
                BEGIN
                    dbms_aq.dequeue(
                        queue_name => qname,
                        dequeue_options => dopt,
                        message_properties => mprop,
                        payload => deq_order_data,
                        msgid => deq_msgid);

                    deq_item_data := deq_order_data.items(1);
                    deq_book_data := deq_item_data.item;
                    deq_cust_data := deq_order_data.customer;

                    IF (deq_cust_data IS NOT NULL) THEN
```

```
                      dbms_output.put_line(' **** NEXT ORDER **** ');
                      dbms_output.put_line('order_num: ' ||
                                deq_order_data.orderno);
                      dbms_output.put_line('ship_state: ' ||
                                deq_cust_data.state);
                    END IF;
                    dbms_output.put_line(' ---- next book ---- ');
                    dbms_output.put_line(' book_title: ' ||
                                deq_book_data.title ||
                                ' quantity: ' || deq_item_data.quantity);
                EXCEPTION
                    WHEN end_of_group THEN
                      dbms_output.put_line ('*** END OF ORDER ***');
                      commit;
                      dopt.navigation := DBMS_AQ.NEXT_TRANSACTION;
                END;
            END LOOP;
          EXCEPTION
            WHEN no_messages THEN
                 dbms_output.put_line (' ---- NO MORE NEW ORDERS ---- ');
                 new_orders := FALSE;
          END;
        END LOOP;

end;
/

show errors;

grant execute on get_new_orders to OE;

Rem Login into OE account
connect OE/OE;
set serveroutput on;

Rem
Rem Enqueue some orders using message grouping into OE_neworders_que
Rem

Rem First Order
execute BOLADM.new_order_enq('My First   Book', 1, 1001, 'CA');
execute BOLADM.same_order_enq('My Second  Book', 2);
commit;
/
```

```
Rem Second Order
execute BOLADM.new_order_enq('My Third   Book', 1, 1002, 'WA');
commit;
/

Rem Third Order
execute BOLADM.new_order_enq('My Fourth  Book', 1, 1003, 'NV');
execute BOLADM.same_order_enq('My Fifth   Book', 3);
execute BOLADM.same_order_enq('My Sixth   Book', 2);
commit;
/

Rem Fourth Order
execute BOLADM.new_order_enq('My Seventh Book', 1, 1004, 'MA');
execute BOLADM.same_order_enq('My Eighth  Book', 3);
execute BOLADM.same_order_enq('My Ninth   Book', 2);
commit;
/

Rem
Rem Dequeue the neworders
Rem

execute BOLADM.get_new_orders;
```

tkaqdocc.sql: Clean-Up Script

```
Rem
Rem $Header: tkaqdocc.sql 26-jan-99.17:51:05 aquser1 Exp $
Rem
Rem tkaqdocc.sql
Rem
Rem  Copyright (c) Oracle Corporation 1998, 1999. All Rights Reserved.
Rem
Rem    NAME
Rem      tkaqdocc.sql - <one-line expansion of the name>
Rem

set echo on;
connect system/manager
set serveroutput on;

drop user WS cascade;
drop user ES cascade;
drop user OS cascade;
drop user CB cascade;
drop user CBADM cascade;
drop user CS cascade;
drop user OE cascade;
drop user boladm cascade;
```

Index