akadia
Information Technology

# Advanced Queuing AQ, Part II - Oracle's JMS Interface to AQ

**René Steiner, Akadia AG**

Author: René Steiner


**Akadia AG**
Information Technology
Arvenweg 4
CH-3604 Thun
Phone 033 335 86 20
Fax 033 335 86 25
E-Mail info@akadia.com
Web www.akadia.com

Oracle Advanced Queuing AQ is a powerful queuing mechanism for message exchanging between different applications. Part I of these articles introduced AQ and explained how to create queues in the database, use PL/SQL in a Point-to-Point Model, JPublisher and Oracle's Native AQ Interface for Java.

Part II now covers:

- Introduction to **Java Message Service JMS**
- Publish messages and subscribe to topics with **Oracle's JMS Interface to AQ** using multi-consumer queues
- Some words about **Persistence**
- **Oracle Message Broker OMB** outlook

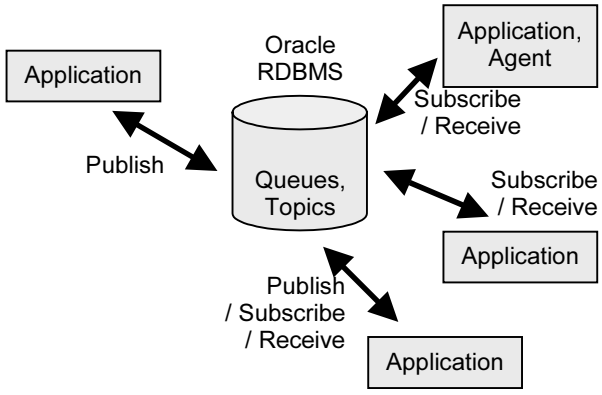
As in the Part I of these articles there are only sample code snippets shown here. They include all necessary statements to understand AQ. Download full running samples and batch files from **www.akadia.com**

Here complete samples are available, showing all the required statements, environment variables, imports, JAR files, error processing's along with Windows batch files to compile and run etc.

### What we want to do - the Two Models

Part I showed a simple system with two applications that use one or more queues together. This approach has been called the **Point-to-Point Model**.

However, in a more sophisticated system we may like to have different applications that publish messages and others that subscribe to certain queues from where they like to consume messages. The **Publish-Subscribe Model** uses multi-consumer queues:




Publisher applications propagate messages to queues which are called topics here. These messages can be either addressed for specific applications or they will be received by all destinations. Applications receiving messages are also called agents. We talk about **broadcast** if the message can be consumed by all applications and about **multicast** if a subscription is required for consummation. To explain broadcast and multicast more clearly the following parallel cases are often used: Broadcast is similar to radio and TV broadcasting received by everybody. Multicast can be seen like a newspaper where you need a subscription. Many people have a subscription but not everybody.

### The Java Message Service JMS

Different enterprise messaging vendors lead by Sun Microsystems, Inc. defined a common API for reliable and flexible message exchange in distributed systems throughout an enterprise. Oracle is one of the companies that implemented JMS and decided to do it by their own Advanced Queuing feature. Other companies implement JMS using another technology. The underlying technology remains exchangeable and developers do not need to learn always proprietary messaging API's.

We will use Oracle's JMS Interface to AQ that implements an interface for Advanced Queuing.

### Other Words for Same Things

If we talk about queues while using Advanced Queuing, we call the same **Topics** if we are in the world of JMS. The same is true for enqueue and dequeue. They're called **Publish** and **Receive** in JMS. Additionally applications are often called **Agents** in JMS.

### Queue Creation

Required database users creation and granting of needed privileges has been described in Part I of these articles. If you missed it please see **www.akadia.com** for samples and explanations.

We created an AQ administration user "aqadm" and a AQ application user called "aquser" for the samples.

A queue table "multi_message_table" with a special object type AQ$_JMS_OBJECT_MESSAGE will be created now. This object type is a reference only and does not yet define the message structure. It gives us the freedom to define later the payload of our messages we like to transfer. The payload is the data type and structure used for every message.

Creating and starting the queue "multi_queue" works in the same way as for the Point-to-Point connection, except that the parameter "multiple_consumers" is set to true.

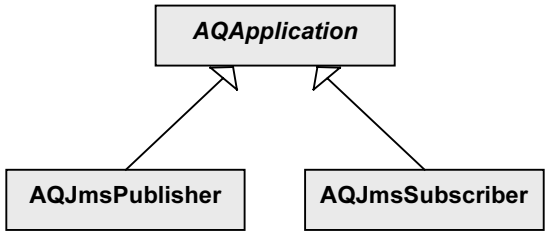
We use the AQ administration user "aqadm":

```
CONNECT aqadm/aqadm;
```

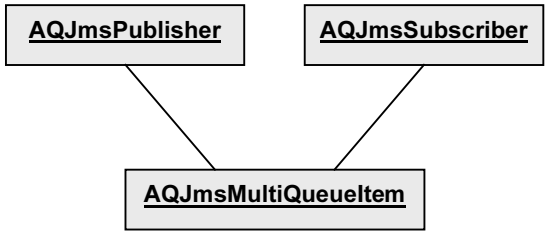
```
EXEC DBMS_AQADM.
    CREATE_QUEUE_TABLE (
    queue_table => 'multi_message_table',
    queue_payload_type =>
        'SYS.AQ$_JMS_OBJECT_MESSAGE',
    multiple_consumers => TRUE);
EXEC DBMS_AQADM.CREATE_QUEUE (
    queue_name => 'multi_queue',
    queue_table => 'multi_message_table');
EXEC DBMS_AQADM.START_QUEUE (
    queue_name => 'multi_queue');
```

### Introducing the Java Samples

When using Java we leave the procedural area and step into the object oriented world. The abstract base class AQApplication offers all required steps we must perform before we are able to start working with queues.




The UML diagram shows our sample classes AQJmsPublisher and AQJmsSubscriber derived from AQApplication. They will be used to act as publishers and subscribers.

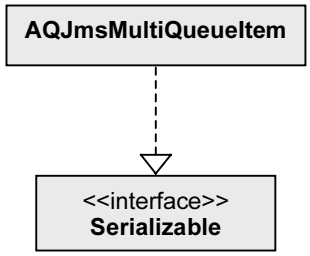



Both classes instantiate the payload class AQJmsMultiQueueItem as message data type.

We will focus down to those statements we must know about, in order to work with queues. It's not necessarily required to understand the whole object oriented concept.

### Create Class for Message Content

In our sample application the following class is used as payload for message content. We are free in choosing the member variables and methods. However, the class must implement the Serializable interface.




The Serializable interface in the java.io library is used for object serialisation. Serialisation means, exchanging objects between programs on the same machine and between remote computers. The objects are transferred via streams and networks conserving their current states and data. They are restored by receivers, become alive again and continue to work.

```
public class AQJmsMultiQueueItem
                implements Serializable {

    private int _no;
    private String _title;
    private String _text;

    public AQJmsMultiQueueItem(
                int no, String title, String text) {
        _no  = no;
        _title = title;
        _text = text;
    }
```

```
    public int getNo() { return _no; }
    public String getTitle() { return _title; }
    public String getText() { return _text; }
}
```

**Publish a Message with Oracle's JMS Interface to AQ**

To establish a connection to a topic we need to create a connection factory using JDBC. As connection string we use a host name for *HOST* and an Oracle database SID for *SID*. Between these two values the listener port address must be specified, e.g. 1521. We use a Properties object to pass the AQ user name and password.

```
Properties info = new Properties();
info.put("aquser", "aquser");

TopicConnectionFactory
    topicConnectionFactory = AQjmsFactory.
        getTopicConnectionFactory(
            "jdbc:oracle:thin:@HOST:1521:SID",
            info);
```

With the factory we can get now two things: An AQ topic connection and an AQ topic session. We pass true for a transactional session and request client acknowledges. This simply means that we like a transactional behaviour and that clients perform ROLLBACKs and COMMITs.

```
aq.connection = topicConnectionFactory.
    createTopicConnection("aquser", "aquser");
aq.session = aq.connection.
    createTopicSession(true,
        Session.CLIENT_ACKNOWLEDGE);
```

We can start the connection and create a publisher afterwards. We could pass a topic to the publisher at the place where we pass null now. But without specifying a topic here we can work with more than one topic using the same publisher. Afterwards we get a reference to the topic we like to use now. The topic is owned by "aqadm" and is called "MULTI_QUEUE".

```
aq.connection.start();

TopicPublisher publisher = aq.session.
    createPublisher(null);
Topic topic = ((AQjmsSession) aq.session).
    getTopic("aqadm", "MULTI_QUEUE");
```

Lets make an instance of the recently created payload class AQJmsMultiQueueItem. This object is converted into a JMS object message.

```
AQJmsMultiQueueItem messageData =
    new AQJmsMultiQueueItem(
        0,
        "Published message title",
        "This is the message text");
ObjectMessage objectMessage =
    aq.session.createObjectMessage(
        messageData);
```

We don't want to send and broadcast this message to everybody. Instead, we prepare a list of agents for multicasting. These recipients are identified by subscription names, e.g. "SUBSCRIPTION1" and "SUBSCRIPTION2". The null parameter could be used to pass an address identifying agents on remote machines.

```
AQjmsAgent[] recipientList =
    new AQjmsAgent[2];
recipientList[0] =
    new AQjmsAgent("SUBSCRIPTION1", null);
recipientList[1] =
    new AQjmsAgent("SUBSCRIPTION2", null);
```

Finally we publish the message to the topic along with the recipient list, commit the whole thing and close the session and connection.

```
((AQjmsTopicPublisher) publisher).
    publish(topic, objectMessage, recipientList);

aq.session.commit();
aq.session.close();
aq.connection.close();
```

**Subscribe to a Topic and Receive a Message**

We can create a subscriber agent on a topic and call it for example "SUBSCRIPTION1". The null parameter could be replaced by a message selector that filters some of the received messages. Here we like to get all.

```
TopicReceiver subscriber =
    ((AQjmsSession) aq.session).
            createTopicReceiver(
                topic,
                "SUBSCRIPTION1",
```

```
                null);
```

When calling the receive method the program waits until a message appears on the topic. After 60 seconds it runs into a time-out, the program continues and the "objectMessage" stays null, if no message appears. The time-out is specified in milliseconds.

```
System.out.println(
    "Waiting 60 seconds for message");
ObjectMessage objectMessage =
    (ObjectMessage) subscriber.receive(60000);
```

We read back our payload stored in the object message.

```
if (objectMessage != null) {

    AQJmsMultiQueueItem messageData =
        (AQJmsMultiQueueItem)
                objectMessage.getObject();

    System.out.println("Received no: " +
        messageData.getNo());
    System.out.println("Received title: " +
        messageData.getTitle());
    System.out.println("Received text: " +
        messageData.getText());
}
```

Again we need a final commit and we close the session and connection.

```
aq.session.commit();
aq.session.close();
aq.connection.close();
```

**Some Words About Persistence**

The queues and all messages inside are persistent. Persistency means that sending and receiving messages is a transaction controlled operation. The familiar statements ROLLBACK and COMMIT can do this. Messages survive even system crashes. We may use our samples to send a message, shutdown the database, restart it and receive the message.

**Oracle Message Broker OMB Outlook**

While working with Advanced Queuing and JMS we often come across the Oracle Message Broker OMB in literature and in browsing the web. The question arises if we need OMB and for what purpose it is.
The Oracle Message Broker provides a platform-independent messaging mechanism. The complexity of different underlying messaging technologies should be hidden. The Java Message Service JMS is the foundation of the message broker. Many different drivers from several vendors are supported. Advanced Queuing is one of them. In these articles we use directly Native AQ and JMS without any message broker. If we need to connect many different platforms the complexity could be reduced by using a common interface and broker mechanism. To describe OMB is not part of this article. See for instance "Oracle Message Broker Administration Guide" for details.

**Conclusion and Prospects**

Oracle Advanced Queuing is a rather simple but powerful way to work with messages and PL/SQL, Native AQ or the Java Message Service JMS. Both articles Parts I and II speak only about the main functionality and the essential operations that are required to produce functioning programs. There are a lot more features available such as message prioritisation, message grouping, rule-based subscription, message scheduling, message histories and many more...
In distributed systems the Java Message Service JMS can be a good decision for a platform independent messaging implementation. On different connected platforms, several JMS implementations can be used and can even be replaced if necessary. Using Oracle's JMS Interface to AQ on nodes where an Oracle database server is already installed can be very efficient and cost effective. No third-party messaging solution must be evaluated and because we are familiar with Oracle less learning effort is required.

**Links and Documents**

Samples shown in this article
www.akadia.com

Java Message Service JMS
java.sun.com/products/jms

Oracle Documentation

Application Developer's Guide
- Advanced Queuing
Supplied PL/SQL Packages Reference
Supplied Java Packages Reference

**Contact**

René Steiner
E-Mail rene.steiner@akadia.com

Akadia AG
Information Technology
Arvenweg 4
CH-3604 Thun
Phone 033 335 86 20
Fax 033 335 86 25
E-Mail info@akadia.com
Web www.akadia.com