akadia
Information Technology

Advanced Queuing AQ, Part I - PL/SQL and Oracle's Native AQ for Java

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 introduced powerful queuing mechanisms where messages can be

exchanged between different programs. They called it Advanced Queuing AQ. Exchanging messages and communicating between different application modules is a key functionally becoming important as soon as we leave the database servers' SQL and PL/SQL programming domain. If we have to do different jobs simultaneously, for instance to communicate with external systems and evaluate complex queries at the same time, it might be a design decision to uncouple "request for service" and "supply for service". In one case an application module deals with all external systems and requests a certain query by posting a message on a queue. In the other case an application gets the message, performs the query and supplies the result back on the queue in between.

While using Oracle Advanced Queuing we do not have to install additional middle-ware dealing with inter-process communication and transaction monitoring. We can directly use an existing and well-known database and can benefit from given functionalities like online backup or transaction processing. Alternatively other simple and non queue based messaging techniques can be used like the Java RMI, which is limited to Java. Or more complex approaches like CORBA, where the complexity lies more in design and conceptual decisions.

This is Part I of two articles that covers the following topics:

- How to **create and administrate queues** in the database
- Using a queue with **PL/SQL** in a Point-to-Point Model
- Using Oracle's **JPublisher** to generate Java classes representing database types
- Dequeue a Point-to-Point message with **Oracle's Native AQ Interface for Java** similar to the PL/SQL dequeue sample

And Part II will cover:

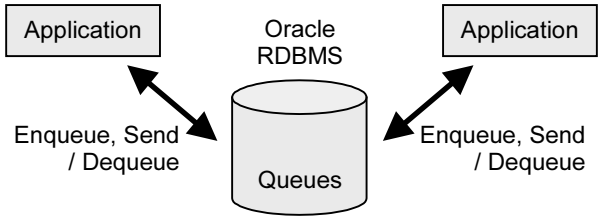
- Introduction to **Java Message Service JMS**
- Publish messages and subscribe to topics with **Oracle's JMS Interface to AQ** using multi-consumer queues
- Some words about **Persistence**
- **Oracle Message Broker OMB** outlook

There are only sample code snippets shown for Java in both articles to avoid going beyond the scope. They include all necessary statements to understand AQ. You may download full running samples 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

In a simple system we could think about two applications that like to use one or more queues together. This approach is called the **Point-to-Point Model**:



The process to put messages on a queue is called enqueue or send whereas the opposite is called dequeue. There may be more than two consumer applications but a single message can only be dequeued once. Consumers may browse the queue without dequeuing messages.

In a more advanced 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. There is no more strict connection between applications. This is called the **Publish-Subscribe Model** and is covered by Part II of these articles.

Queue Creation

For database queue creation we should have an AQ administrator user with the required privileges. It can be used as object owner too. All created queues and message object types will belong to this administrator. Afterwards we can create as many queue users as we like or grant the required privileges to existing users who want to access the queues. To avoid maintaining privileges for every single user, we

will create two roles in this sample. One for the AQ administrator and another for all AQ users.

In these samples the administrator role is called "my_aq_adm_role" and the corresponding user "aqadm". We grant Oracle's AQ role "aq_administrator_role" to our administrator role.

```
CREATE ROLE my_aq_adm_role;
GRANT CONNECT, RESOURCE,
      aq_administrator_role
   TO my_aq_adm_role;
```

The user role is called "my_aq_user_role" and the corresponding sample user "aquser". Here we grant Oracle's AQ role "aq_user_role" and additional system privileges required for basic operations.

```
CREATE ROLE my_aq_user_role;
GRANT CREATE SESSION, aq_user_role
   TO my_aq_user_role;
EXEC DBMS_AQADM.
   GRANT_SYSTEM_PRIVILEGE(
      privilege => 'ENQUEUE_ANY',
      grantee => 'my_aq_user_role',
      admin_option => FALSE);
EXEC DBMS_AQADM.
   GRANT_SYSTEM_PRIVILEGE(
      privilege => 'DEQUEUE_ANY',
      grantee => 'my_aq_user_role',
      admin_option => FALSE);
```

Now we're ready to create the AQ administration user:

```
CREATE USER aqadm
      IDENTIFIED BY aqadm
   DEFAULT TABLESPACE tab
   TEMPORARY TABLESPACE temp;
GRANT my_aq_adm_role TO aqadm;
```

And the queue user for our samples:

```
CREATE USER aquser
      IDENTIFIED BY aquser
   DEFAULT TABLESPACE tab
   TEMPORARY TABLESPACE temp;
GRANT my_aq_user_role TO aquser;
```

For our first queue we will use an object type instead of a base data type like NUMBER or VARCHAR2 as **payload**. The payload is the data type and structure used for every message. To use an object type is more realistic than sending single numbers or strings around but a bit more complicated. In a message we might have an identification number, a title and a message text or content.

It's time now to change to the AQ administration user where the previous operations could be performed by any DBA.

```
CONNECT aqadm/aqadm;
```

```
CREATE TYPE queue_message_type
   AS OBJECT (
      no NUMBER,
      title VARCHAR2(30),
      text VARCHAR2(2000) );
/
GRANT EXECUTE ON queue_message_type
   TO my_aq_user_role;
```

Let's create a queue called "message_queue" with a corresponding queue table "queue_message_table". We start the queue so that it can be used from now on.

```
EXEC DBMS_AQADM.
   CREATE_QUEUE_TABLE (
      queue_table => 'queue_message_table',
      queue_payload_type =>
         aqadm.queue_message_type');
EXEC DBMS_AQADM.CREATE_QUEUE (
   queue_name => 'message_queue',
   queue_table => 'queue_message_table');
EXEC DBMS_AQADM.START_QUEUE (
   queue_name => 'message_queue');
```

Now we have a complete queue that is ready to use. All the administrative PL/SQL operations shown are available in Java too. However it's a handy idea to do these steps in a SQL shell.

Using a Queue with PL/SQL in a Point-to-Point Model

To work with queues we connect with our AQ sample user:

```
CONNECT aquser/aquser;
```

Now we like to enqueue a message. We have to name the queue, give some default options and pass our message "my_message" as payload, which is made by our own defined

message. Remember, we live in a transactional environment. We must issue a final COMMIT.

```
CONNECT aquser/aquser;
DECLARE
   queue_options
      DBMS_AQ.ENQUEUE_OPTIONS_T;
   message_properties DBMS_AQ.
         MESSAGE_PROPERTIES_T;
   message_id RAW(16);
   my_message aqadm.queue_message_type;
BEGIN
   my_message := aqadm.
      queue_message_type(
         1,
         'This is a sample message',
         'This message has been posted on ' ||
            TO_CHAR(SYSDATE,
               'DD.MM.YYYY HH24:MI:SS'));
   DBMS_AQ.ENQUEUE(
      queue_name =>
         'aqadm.message_queue',
      enqueue_options => queue_options,
      message_properties =>
         message_properties,
      payload => my_message,
      msgid => message_id);
   COMMIT;
END;
/
```

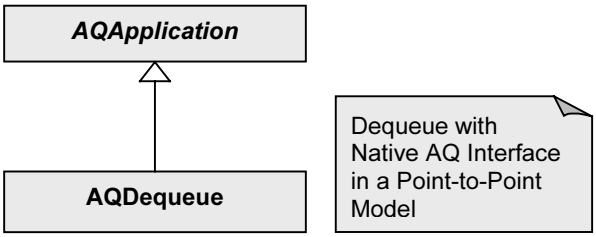
We can dequeue the recently enqueued message. The DBMS_AQ.DEQUEUE statement waits until there is a message to dequeue. The shown code looks very similar to the one above.

```
SET SERVEROUTPUT ON;
DECLARE
   queue_options DBMS_AQ.
      DEQUEUE_OPTIONS_T;
   message_properties DBMS_AQ.
         MESSAGE_PROPERTIES_T;
   message_id RAW(2000);
   my_message aqadm.queue_message_type;
BEGIN
   DBMS_AQ.DEQUEUE(
      queue_name =>
         'aqadm.message_queue',
      dequeue_options => queue_options,
      message_properties =>
         message_properties,
      payload => my_message,
      msgid => message_id );
   COMMIT;
   DBMS_OUTPUT.PUT_LINE(
      'Dequeued no: ' || my_message.no);
   DBMS_OUTPUT.PUT_LINE(
      'Dequeued title: ' || my_message.title);
   DBMS_OUTPUT.PUT_LINE(
      'Dequeued text: ' || my_message.text);
END;
/
```

The PL/SQL samples were easy and straightforward. Not a lot to do. Every kind of application and programming environment could use it like this, assuming they are able to connect to the database and execute PL/SQL stored procedures. However, it is more convenient and is better practise to use the programming languages' own way to deal with messages. That's the point where we should have a look what Java offers...

Introducing the Java Samples

While using Java, it's not only the different programming syntax we use but also the way we design programs. We leave the procedural area and enter into the object oriented world. In these samples we use an abstract base class AQApplication to hide all the steps we must perform before we are able to start working with queues.



This UML diagram shows our sample class AQDequeue derived from AQApplication. 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.

Using JPublisher to Prepare an Oracle Object Type for Java

When we created our queue we created also an Oracle object type to be used for messages. Because we cannot use Oracle data types in Java we must have a Java class to fill the dequeued message in. JPublisher can do this job for us. It connects to the database and

creates a Java class matching the specified Oracle object type.

```
set CLASSPATH=
D:\Oracle\Product\8.1.7\jdbc\lib\classes12.zip;
D:\Oracle\Product\8.1.7\sqlj\lib\translator.zip;
D:\Oracle\Product\8.1.7\sqlj\lib\runtime.zip

jpub -user=aqadm/aqadm
        -sql=QUEUE_MESSAGE_TYPE
        -usertypes=oracle
        -methods=false
```

-user=aqadm/aqadm
Object owner and password to which the to be translated objects belong.

-sql=QUEUE_MESSAGE_TYPE
One or more object types and packages that you want JPublisher to translate. Use commas for separation.

-usertypes=oracle
The oracle mapping maps Oracle datatypes to their corresponding Java classes.

-methods=false
If true, JPublisher generates SQLJ classes for PL/SQL packages and wrapper methods for methods in packages and object types. SQLJ wraps static SQL operations in Java code. We do not use SQLJ here, thus we pass false.

JPublisher connects to the database and creates a Java class QUEUE_MESSAGE_TYPE for us. We can use this class now as a Java data type to receive messages posted by another Java or PL/SQL client.

Dequeue a Point-to-Point Message with Oracle's Native AQ Interface for Java

We use the previously shown PL/SQL sample and replace the dequeue functionality by Java to show the similarities. Additionally it can be used to show message exchanging between PL/SQL and Java. Enqueuing in Java is very similar again to dequeuing and is therefore not shown here.

Before we can start using Oracle's Native AQ Interface for Java we must connect to the database via JDBC. As a 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.

```
Class.forName(
    "oracle.jdbc.driver.OracleDriver");

aq.connection = DriverManager.getConnection(
    "jdbc:oracle:thin:@HOST:1521:SID,
    "aquser", "aquser");
aq.connection.setAutoCommit(false);
```

Afterwards we create a so-called AQ session passing the AQ connection:

```
Class.forName("oracle.AQ.AQOracleDriver");
aq.session = AQDriverManager.
    createAQSession(aq.connection);
```

Now we're ready to get a reference to the queue we like to use. To do so, we pass the queue owner and the queue name:

```
AQQueue queue = aq.session.getQueue(
    "aqadm", "MESSAGE_QUEUE");
```

For dequeuing we create default options and pass them along with an instance of JPublisher's created message data type QUEUE_MESSAGE_TYPE.

```
AQDequeueOption dequeueOption
    = new AQDequeueOption();

System.out.println(
    "Waiting for message to dequeue...");
AQMessage message =
    ((AQOracleQueue)queue).dequeue(
        dequeueOption,
        QUEUE_MESSAGE_TYPE.getFactory());
```

To get the message content we convert the raw payload into our message type.

```
AQObjectPayload payload =
    message.getObjectPayload();
QUEUE_MESSAGE_TYPE messageData =
    (QUEUE_MESSAGE_TYPE) payload.
        getPayloadData();

aq.connection.commit();

System.out.println("Dequeued no: " +
    messageData.getNo());
System.out.println("Dequeued title: " +
```

```
    messageData.getTitle());
System.out.println("Dequeued text: " +
    messageData.getText());
```

Like in PL/SQL, we need a final COMMIT.

Conclusion

Oracle Advanced Queuing is a powerful and rather simple way of working with queues. The libraries available for Java offer a smart way to enqueue and dequeue messages without too much programming overhead. In more sophisticated projects Advanced Queuing's whole functionality can be taken into account. Oracle's Advanced Queuing developer's guide exceeds thousand pages. This might be seen as an indication that in advanced projects a reasonable amount of time is required to understand the different concepts and possibilities. This should not be seen as a disadvantage for Oracle's Advanced Queuing, the same is true for other similar communication or queuing technologies.

Only essential issues and aspects have been covered by this article. Part II of these articles will cover somewhat more sophisticated concepts such as publishing and subscribing a message with Oracle's Java Message Service JMS Interface to AQ.

Links and Documents

Samples shown in this article
www.akadia.com

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