ORACLE STREAMS AQ: LESSONS FROM THE TRENCHES

Bill Coulam, DBArtisans

Asynchronous Processing, Queues, MOM and Apple Pie

In typical business computing, distributed and disparate systems often have the need to communicate with one another as the timing, context or passage of events within one part of a business process need to be known by another. There are various ways of accomplishing this.

If you work for a company that implemented all of its systems on a single, unified stack of technology, it is fairly easy to pass information from one subsystem to another. In such an environment, synchronous processing is possible as routine A calls routine B and waits for the reply before returning control to the batch or event-driven client.

But what if you live in the real world and yours is a complex morass of aging, strategic, acquired, and new technologies, none of which can talk directly with each other? What if routine B takes 5 minutes to finish, but routine A can't wait that long? What if A relies on B, but B sometimes hiccups, errors or never returns? What then? What if routines/systems B, C, D and E all need to concurrently know when A completed? How do we decouple A, so that it isn't held up or destroyed by the actions of B, but is still notified when B does complete successfully? Typical synchronous (request-response) processing fails or falls short in these situations, and asynchronous processing options should be evaluated. Should the enterprise use a message bus or broker, a message switch, a workflow system, a TPM[1], a table to persist the messages, a common directory on the file system for formatted messages, publish-subscribe tools, SOA[2]? What?

This is where the architect earns his or her pay. The distributed systems must be able to communicate in a decoupled, asynchronous, generalized manner that they all share. One excellent option is using middleware, which, as its name suggests, acts as the go-between or "middleman" between systems. Message-oriented middleware (MOM) is the most prevalent form of middleware[3], eclipsing older CPI-C and RPC methods of inter-system communication. MOM is often implemented as a message queuing system, allowing the information/event producer to drop a message in the queue and leave. The parties interested in the message will dequeue the message when they wish (pull), or have the message possibly transformed and forwarded to them (push). This flavor of MOM is known as Message Queueing Middleware, or MQM

A message queue serves as the *lingua franca*, or neutral ground between the physically and technologically separate systems, thus removing the impediments imposed by differing platforms, languages, protocols, operating systems, and networking APIs. As mentioned above, the message queue stores, forwards and possibly transforms the message from one datatype or format to another. One of the primary benefits of MQM is robustness. They guarantee the message will be processed only once, never duplicated or lost. They are also built to scale well, easily accommodating growing loads of enqueued messages. Despite message queues acting as neutral zones, there are not many standards when it comes to queuing. One of them is the Java Messaging Service (JMS), a specification for passing and receiving messages with a queue. Most MQM products have implemented JMS. So writing your application to use the JMS APIs is a relatively safe bet.

Oracle Streams AQ Overview

One of the leading MQM systems that supports JMS is Oracle Streams Advanced Queuing, hereafter referred to as Oracle AQ or just AQ. Introduced in Oracle 8.0 as Oracle Advanced Queuing, Oracle's MQM solution has been around for a long time. Underneath the hood Oracle AQ is implemented with the Oracle Database Server, utilizing database tables to store and manage the messages, and inheriting all the security, transactional integrity, scheduling facilities and robustness that come with Oracle. AQ is a built-in, standard feature of all editions of the Oracle Database Server.

The piece of Oracle software that actually manages the queues is the queue monitor process, identified by the token QMN in tools that display operating system processes. Up to 10 queue monitor processes can be configured to help manage Oracle AQ work. This was a manual operation up to Oracle 10.1, which saw the birth of the Queue Monitor Coordinator (QMNC), which handles the quantity of queue monitor processes automatically based on load.

[1] Transaction Processing Monitor/Middleware like Top End and Tuxedo
[2] Service Oriented Architecture
[3] See http://www.middleware.org/mom/basicmom.html for a list of basic MOM/MQM tools and vendors.

From its inception, some of the major features offered by Oracle AQ included:

- Single-consumer queues for simple, system-to-system communication
- Multi-consumer queues to support publish/subscribe models of message passing, where multiple recipients are interested in a message.
- Message prioritization, ordering, grouping, navigation, selection, inspection, delay, retention and expiration
- SQL-based access to queues, message metadata, payload and performance views
- PL/SQL, Visual Basic and Java interfaces for enqueuing and dequeuing
- Rich payload typing model, supporting both scalar and user-defined datatypes
- And much more

Lots of awaited improvements were added with the release of 9i. My favorite was the ultra-flexible payloads based on the new XMLType datatype.

In 10gR1 the product was "integrated[4]" into Oracle Streams and re-labeled as Oracle *Streams* Advanced Queuing. The JMS interface was upgraded to support JMS 1.1. Heavily changed versions of the core DBMS_AQ and DBMS_AQADM packages were introduced. A C++ interface was added. Nine other items were also enhanced.

More improvements followed in 10gR2, like *buffered* messaging (a new and much faster version of the previous non-persistent messaging), and commit-time ordering, a feature not typically offered by MQM systems. Nothing was enhanced in 11gR2, but there were minor improvements in 11gR1.

The features offered by AQ are comprehensive, too vast to cover in this paper. Instead of repeating information found in the Oracle Streams Advanced Queuing User's Guide, or focusing on an esoteric corner of AQ, the remainder of this paper will concentrate on the most common use of AQ, the single consumer queue, and its maintenance and troubleshooting in the real world, this in hopes of being immediately useful to the widest possible audience. Propagation, notification, transformation, LDAP integration and the Messaging Gateway will not be covered.

Terminology

Before proceeding further, let's ensure we are on the same page with some queuing terms.

In queuing a *message* consists of metadata (information about the message itself, like where it came from, when it arrived, its current state, etc. and *payload*, which is the actual message in a binary or textual format

In a single-consumer queue setup, the application sending the message is referred to as the message *producer* in Oracle documentation. I prefer the term *sender* as it is less ambiguous. However, Oracle docs do refer to the application interested in the message as the *consumer*, a term I like, but which I use interchangeably with the term *receiver*.

Administrative and Operational Privileges

Before anything can be done, the right schemas must be given the right roles and/or privileges. In a typical enterprise database, the players will be

- The DBA schema to configure access controls
- The application schema which will own the queue
- And the connection schema, used by the application server's connection pool, and which has rights to either enqueue to or dequeue from the application schema's queue.

In the code examples below, SYS will be used as the DBA schema, the application schema will be named MYSCHEMA, the connection schema is CLIENTSCHEMA, and a separate subsystem that wants access to enqueue to MYSCHEMA's queue will be named OTHERSCHEMA.

[4] No idea what that means because its PL/SQL interface looks the same to me.

To administer all queues in all schemas, as SYS grant the AQ_ADMINISTRATOR_ROLE or the MANAGE ANY QUEUE system privilege. By default SYS and SYSTEM have the AQ_ADMINISTRATOR_ROLE already, which includes the rather broad MANAGE ANY QUEUE system privilege. However, unless you have specialized circumstances, I recommend that neither be granted to any other users, as that just increases the number of possible points for security failure.

To be able to create, start, stop and drop a queue, as SYS grant the EXECUTE privilege on DBMS_AQ and DBMS_AQADM. No other role or privilege is required for the queue owner. Although there is an AQ_USER_ROLE, it is not necessary or recommended to use it in the latest versions of Oracle. Also, accounts using the JMS interface will need EXECUTE granted on DBMS_AQIN and DBMS_AQJMS.

```
GRANT EXECUTE ON sys.dbms_aqadm TO MYSCHEMA;
GRANT EXECUTE ON sys.dbms_aq TO MYSCHEMA;
GRANT EXECUTE ON sys.dbms_aqin TO MYSCHEMA;
GRANT EXECUTE ON sys.dbms_aqjms TO MYSCHEMA;

GRANT EXECUTE ON sys.dbms_aq TO CLIENTSCHEMA;
GRANT EXECUTE ON sys.dbms_aqin TO CLIENTSCHEMA;
GRANT EXECUTE ON sys.dbms_aqjms TO CLIENTSCHEMA;
```

To enqueue or dequeue from another schema, just grant the privilege using the DBMS_AQADM routines GRANT_QUEUE_PRIVILEGE and REVOKE_QUEUE_PRIVILEGE. This will be seen in the enqueuing and dequeuing sections below. Oracle does provide ENQUEUE_ANY and DEQUEUE_ANY system privileges, which bypass the need for schema-level access control, but I feel these should be avoided. Good security begins with only granting the bare minimum of access, not opening the doors wide open.

In 10g, the AQ object types used to specify payload type when creating queue tables are now all publicly accessible. However, if you are still languishing on 9i, as SYS you will need to grant EXECUTE privilege on the AQ object types WITH GRANT OPTION to the application schema.

```
-- 9i only
GRANT EXECUTE ON sys.aq$_jms_message TO MYSCHEMA WITH GRANT OPTION;
```

Queue Design

Once your application schema has the right privileges, you can begin creating your queue, stuffing it full of messages, processing them, monitoring them, and troubleshooting. Before creating the queue though, think about the characteristics of your system.

- How many messages do you expect at peak times?
- How long will the dequeue and processing operation take per message?
- Will multiple clients below allowed to push messages into the queue?
- Will only one client pull messages off the queue?
- What if there are errors; how many dequeue attempts should be allowed before giving up?
- Do you want a wait time injected between dequeue attempts, giving you time to fix an intermittent problem?
- Is the message simple, or rich?
- Is yours an Oracle RAC system; should most AQ operations be tied to a particular node?
- Is there a need to browse the queue, or just dequeue and remove?
- Should the messages be grouped, sorted, prioritized or tagged in a special fashion for the dequeue clients' needs?

And so on. In most cases, the default options will be good enough, as some queue attributes are modifiable later through the DBMS_AQADM.ALTER_QUEUE API. But it is good to invest the extra hour thinking about the possibilities now, rather than waiting until your rather busy queue is humming along in a 24X7 system that can't afford downtime.

One final note, I like to name my queues with "_Q" as the suffix, and my queue tables with "_SQT" for single consumer queue table or "_MQT" for multi-consumer queue table. Also the queue name is limited to 24 characters, not the typical 30 as with most Oracle objects.

Creating a Queue

For this paper, we'll be creating a single-consumer queue, accepting most defaults, but requesting tight block usage with a custom storage clause. Before that can happen a queue table must already exist. Here are a few scripts to create the queue table and queue. All of them would be run as the queue owner, in this case MYSCHEMA.

Since my build environment is automated, I try to ensure my scripts are re-entrant, and clean up after themselves before attempting to create anything new (just in case the object already exists from previous builds or botched build attempts).

```
SET SERVEROUTPUT ON
DECLARE
   l_queue_name VARCHAR2(30) := 'MY_Q';
   l_queue_table_name VARCHAR2(30) := 'MY_SQT';
   lx_queue_is_not EXCEPTION;
   lx_queue_running EXCEPTION;
   lx_queue_tab_is_not EXCEPTION;
   PRAGMA EXCEPTION_INIT(lx_queue_is_not,-24010);
   PRAGMA EXCEPTION_INIT(lx_queue_running,-24011);
   PRAGMA EXCEPTION_INIT(lx_queue_tab_is_not,-24002);
BEGIN
   BEGIN
      dbms_aqadm.drop_queue(queue_name => l_queue_name);
   EXCEPTION
      WHEN lx_queue_is_not THEN
         dbms_output.put_line(l_queue_name||' does not exist. Check spelling.');
      WHEN lx_queue_running THEN
         dbms_output.put_line('Stopping '||l_queue_name);
         dbms_aqadm.stop_queue(queue_name => l_queue_name);
         dbms_output.put_line('Dropping '||l_queue_name);
         dbms_aqadm.drop_queue(queue_name => l_queue_name);
   END;
   BEGIN
      dbms_aqadm.drop_queue_table(queue_table => l_queue_table_name, force=>TRUE);
   EXCEPTION
      WHEN lx_queue_tab_is_not THEN
         dbms_output.put_line(l_queue_table_name||' does not exist. Check spelling.');
   END;
END;
```

Now that any previous build attempts are cleaned out, we can proceed with creating the queue table…

```
BEGIN
   dbms_output.put_line('Creating MY_SQT');
   dbms_aqadm.create_queue_table(
      queue_table        => 'MY_SQT'
     ,queue_payload_type => 'SYS.AQ$_JMS_MESSAGE'
     ,storage_clause => 'PCTFREE 0 PCTUSED 99'
     ,multiple_consumers => FALSE
     ,comment => 'My Queue Table: Supports the blah, blah...');
END;
```

And finally the queue…

```
BEGIN
   dbms_output.put_line('Creating MY_Q');
   dbms_aqadm.create_queue(
      queue_name => 'MY_Q'
     ,queue_table => 'MY_SQT'
     ,comment => 'My Queue: Routes the messages from...');

   dbms_aqadm.start_queue(queue_name=>'MY_Q');
END;
```

That's it! MY_Q is now running and awaiting enqueued messages. A default exception queue was auto-created, as well as two "hidden" views created on top of your queue table, AQ$*queue_table* and AQ$_*queue_table*_F. The former is very useful for monitoring and maintenance. So I like to grant this view to the client schemas which have an interest in the queue.

```
GRANT SELECT ON aq$my_sqt TO OTHERSCHEMA, CLIENTSCHEMA;
```

The “F” view is much like the AQ$*queue_table* view, but is just different enough, it looks like the two views were designed by isolated teams for the same reason. I cannot find the “F” view’s purpose documented anywhere, but it is apparently essential for dequeuing in browse mode, as we found out the hard way. More on that later.

Enqueuing

The queue owner would now grant privileges to those schemas that wish to enqueue and dequeue, like this:

```
BEGIN
   dbms_output.put_line('Granting enqueue privs');
   dbms_aqadm.grant_queue_privilege(
      privilege    => 'ENQUEUE' -- can be ENQUEUE, DEQUEUE, or ALL
     ,queue_name   => 'MY_Q'
     ,grantee      => 'OTHERSCHEMA'
     ,grant_option => FALSE);
END;

BEGIN
   dbms_output.put_line('Granting dequeue privs');
   dbms_aqadm.grant_queue_privilege(
      privilege    => 'DEQUEUE' -- can be ENQUEUE, DEQUEUE, or ALL
     ,queue_name   => 'MY_Q'
     ,grantee      => 'CLIENTSCHEMA'
     ,grant_option => FALSE);
END;
```

Now, logged in as OTHERSCHEMA, we can send a message to the queue in MYSCHEMA. The method of enqueue will depend on which API is used by your application. Many will use the Java JMS API. But since this is a paper for database professionals, we’ll demonstrate using the PL/SQL DBMS_AQ API. Here is a basic enqueue operation.

```
DECLARE
   l_msg           sys.aq$_jms_message;
   l_queue_options dbms_aq.enqueue_options_t;
   l_msg_props     dbms_aq.message_properties_t;
   l_msg_id        RAW(16);
BEGIN
   l_msg := sys.aq$_jms_message.construct(dbms_aqjms.jms_text_message);
   l_msg.set_text('<useful message here>');
   dbms_aq.enqueue(
      queue_name         => 'MYSCHEMA.MY_Q'
     ,enqueue_options    => l_queue_options
     ,message_properties => l_msg_props
     ,payload            => l_msg
     ,msgid              => l_msg_id);
   COMMIT; -- very important; won't enqueue without commit
END;
```

The options and properties variables provide the sender a high degree of control over their message. Exploring all the possibilities is left as an exercise to the reader as this paper is already beyond the maximum page length.

Dequeuing

By default a dequeue request retrieves the first available message in the queue. Which message is considered the “first available” is determine by AQ, not by the message receiver. If a higher degree of control is required, the process can be designed so that messages are enqueued with certain properties that allow a receiver to ask for messages that have a certain grouping, correlation identifier, message identifier or that meet a dequeue condition. The order of messages being removed by the queue can also be influenced at queue table creation time, which can be configured to order incoming messages by message priority, enqueue time or commit time. I’ve not found any of this necessary within my various uses of AQ and just go with the default mode.

Oracle AQ also allows one to “dequeue” in a mode that really just browses through the list of messages, and to dequeue in a mode that removes a message, but basically discards it. Browsing will become more important later as we discuss one of the problems we encountered.

If there are blocking processes just awaiting incoming messages, you’ll rarely see any messages in the queue when querying the views or queue table to observe them. But if the messages are failing to dequeue, or if the sender requested a delay, or the queue table was created with a retry_delay, you may actually see messages in the queue. The messages will be in one of three states: READY, PROCESSED and EXPIRED. Ready can be dequeued. Processed or expired cannot.

Let's go ahead and request a message using the DBMS_AQ API from CLIENTSCHEMA:

```
DECLARE
   l_msg           sys.aq$_jms_message;
   l_msg_text      typ.t_maxvc2;
   l_queue_options dbms_aq.dequeue_options_t;
   l_msg_props     dbms_aq.message_properties_t;
   l_msg_id        RAW(2000);
BEGIN

   dbms_aq.dequeue(
      queue_name          => 'MYSCHEMA.MY_Q'
     ,dequeue_options     => l_queue_options
     ,message_properties => l_msg_props
     ,payload             => l_msg
     ,msgid               => l_msg_id);
   COMMIT;

   l_msg.get_text(l_msg_text);
   dbms_output.put_line('Dequeued message text: ' ||
      CHR(10) || l_msg_text);
END;
```

Understand that this is a blocking transaction. If this is issued from a tool like TOAD or SQL Developer, the UI will appear to hang and can't be aborted with the usual OCI signal. The only way to free up the session is to send a message to the queue, or kill the session from another tool, connected as DBA.

Once the message is dequeued, it is processed by the application. When processing is complete, the application can send its own message back on another queue, or make use of AQ notifications, to let the original client know that work is complete. And that is the magic of asynchronous processing: liberating these applications from being crippled by remote calls to other systems and routines they are dependent upon.

AQ can be used to implement publish/subscribe architecture, workflows, progress meters, email handlers, and much, much more. We've only scratched the surface due to limited space. Before concluding, let's examine resources available to troubleshoot your queues, and some real world problems I encountered.

Maintaining a Queue

Once you are happily sending and receiving messages through a queue, it is time to peek under the covers. One of the things you can count on in computing is that that there is always one more bug. Something can, and usually does go wrong. When it does, it's an advantage to know what resources you have to diagnose and treat the queue.

Oracle AQ objects can be managed through OEM, or manually through the DBMS_AQADM interfaces and AQ dictionary views. I find OEM cumbersome, so I prefer to use DBMS_AQADM for the handful of post-release operations I've had to perform.

Looking Under the Hood

When messages are not dequeuing, when things seem to be stuck, the first thing I do is look at the messages inside the queue. One can query the queue table itself, but it is fairly normalized and needs a little translation. Oracle thought this one through and automatically creates a view named after your queue table, prefixed with "AQ$". It is owned by the queue owner and can be granted to others. This is my favorite query for the queue table view:

```
SELECT queue
      ,enq_timestamp
      ,enq_user_id
      ,msg_state
      ,retry_count
      ,original_queue_name
      ,expiration_reason
      ,user_data
  FROM aq$my_sqt t
 ORDER BY 2 DESC;
```

If your favorite SQL tool does not easily, or cannot, display the contents of the user_data object, break the fields out using a table alias and OO dot-notation like this:

```
SELECT t.queue
      ,t.enq_timestamp
      ,t.enq_user_id
      ,t.msg_state
      ,t.retry_count
      ,t.original_queue_name
      ,t.expiration_reason
      -- read straight if text, or convert if message is numeric
      ,TO_NUMBER(t.user_data.text_vc) customer_id
  FROM aq$my_sqt t
 ORDER BY 2 DESC;
```

Although you can query the contents of the queue table, you cannot and should never perform DML (insert, update, delete) operations directly on queue tables. If you need to remove a row or alter a message, use the APIs in DBMS_AQADM to first purge the message, then DBMS_AQ to re-enqueue.

But what if you didn't create the queue? What if you're the Physical DBA, and the Development DBAs have been running amok creating queues? Look into the user/all/dba_queues and user/all/dba_queue_tables data dictionary views to see all the AQ objects in your database.

There is also the [G]V$AQ view used to see all the queue messages in the entire database and, if using GV$AQ, across RAC instances. This query has come in handy to explore that view:

```
SELECT dq.owner
      ,dq.name
      ,dq.queue_type
      ,g.*
  FROM gv$aq g
  JOIN dba_queues dq
    ON dq.qid = g.qid;
```

There are other data dictionary views, but they pertain to features not being addressed in this paper.

Exception Queue

Messages that expire before being dequeued, or transactions that fail retry_count+1 times while processing a message, will see those messages moved out of the intended queue and to the exception queue. If not explicitly specified in the enqueue call, this will be the default exception queue created by Oracle at queue creation time, named AQ$_*queue_table*_E.

One cannot enqueue directly to the exception queue, but it can be dequeued. However, it must be formally started, with the dequeue parameter set to TRUE, in order to be able to dequeue from it or purge from it. This could be useful if you wish to write a scheduled job that periodically checks the exception queue and does something, like emailing error reports to concerned parties, or re-enqueuing the same message to give it another shot.

```
BEGIN
   -- Start the default exception queue as well so we can dequeue from it.
   dbms_output.put_line('Starting AQ$_MY_SQT_E exception Q');
   dbms_aqadm.start_queue(queue_name => 'AQ$_MY_SQT_E', enqueue => FALSE, dequeue => TRUE);
END;
```

It is easy to see which messages are in the exception queue by querying the queue table directly or the aq$*queue_table* view. I find the latter more useful. The first column, queue, will indicate the exception queue the message is now in. Column msg_state will indicate the message is in an EXPIRED state. Column original_queue_name tells us where the expired message was moved from. And column expiration_reason uses plain English why the message is in the exception queue.

Issues and Lessons Learned

The remainder of this paper focuses on some problems encountered while creating, using and managing single consumer queues in three new systems during a transition from 9i to 10gR2. Hopefully some of this information can save the reader from wasting time on similar issues.

Too Many Dequeuers Dequeuing

On one project, each developer had a copy of the app server code running locally on their laptop, but all sharing the same development database. Once the dequeue piece had been written using JMS, everyone got a copy. We called this piece the “listener.” All of a sudden there were eight hungry listeners all concurrently dequeuing from the same database queue. Our first handfuls of messages had errors. There were both data and Java coding bugs. But it was very difficult to find this out because once you enqueued a test message, it was rather random which of the eight developer listeners would dequeue it, and when there were failures in the SQL queries that accompanied the dequeue transaction, the message would be retried 5 times (the default) and fail 5 times and immediately go to exception. We’d enqueue a message, and in the blink of an eye, it was sitting in the exception queue. Quite frustrating. The default delay of 0 seconds between attempts really didn’t work for us.

To solve these problems, we reduced the number of developers running an active listener agent, and eventually placed a dynamic (table-driven) parameter in place that allowed us to specify which app server hosts were allowed to do the dequeuing. We also altered the queue to allow only 4 retries, but spaced one hour apart.

```
BEGIN
   dbms_aqadm.alter_queue(
      queue_name => 'MY_Q'
     ,max_retries => 4
     ,retry_delay => 3600);
END;
```

This gave us 4 hours to examine error logs and data, allowing the underlying issue to be fixed, and the message to process successfully on its next try. This reduced the time we spent trying to get messages out of the exception queue.

Once and Only Once, Bah! Try “Twice and Often Twice”

Our Staging environment looks just like Production, except for one key difference, two app server instances in Prod. So although we tested thoroughly, once we went to Production, to our great horror we found that most enqueued messages were seemingly dequeuing twice, being processed twice within the same second, and creating duplicate records in the downstream system.

This was a single-consumer queue, and according to Oracle docs and queuing theory, any message in the queue should only be dequeued once. Well, we observed something else in Development when there were so many dequeuing agents connected concurrently. So we put safeguards (SQL queries) in place at the top and bottom of the dequeue transaction to ensure duplication wouldn’t happen. That was 8 months prior to release. Now here we were in Production, seeing the same symptoms we hadn’t seen since the start of the year.

Our top developers looked into the two Tomcat app server logs and observed that the same message was dequeued by the two listener processes at the exact same time. The listener processes then went on to bypass both sets of safeguards, not seeing each other’s DB work, because they apparently committed at the same time as well. This created duplicate entries. Very embarrassing.

The fact that a message could be dequeued more than once from a single-consumer, persistent queue violates most every principle and benefit upon which an Oracle-based messaging solution is touted. It was acting like a database that had never heard of ACID transactions.

After much research, we found document **413024.1** on Oracle Support which explains it all. **Bug 5590163** details the issue about messages being dequeued more than once. Unfortunately, Oracle’s “fix” for that bug created Bug 7393292, still present in 10.2.0.4. The whole mess is finally fixed in 10.2.0.5

Purging

On the same project mentioned earlier, the listener agent did too much. Within the same transaction, it would dequeue, then issue about six SQL statements, gathering information, and do a few database updates. There would often be data problems in development due to all the unit testing, and looser access to tables there. When there were data problems, an Oracle error would be raised, the transaction would fail, the message would not be removed from the queue, and the retry_count would be incremented, eventually moving to the exception queue.

This happened often enough, that our exception queue got quite dirty. We didn't want to see this happen in production. We wanted a way to clear things out, to keep things clean. This is possible using the purge_queue_table() routine of DBMS_AQADM. It's a little tricky though because the purge condition parameter is a piece of a WHERE clause, and you have to know what the underlying table, columns and alias is in order for a purge to succeed. Keep in mind you are purging the queue table, and the purge condition is referring to columns in the queue table. You can purge an entire queue table like this:

```
DECLARE
   l_purge_opt dbms_aqadm.aq$_purge_options_t;
BEGIN
   l_purge_opt.block := TRUE;
   dbms_aqadm.purge_queue_table(
      queue_table     => 'MY_SQT'
     ,purge_condition => 'queue IN (''AQ$_MY_SQT_E'',''MY_Q'')'
     ,purge_options   => l_purge_opt);
END;
```

But we often just wanted to purge messages pertaining to a certain test account. To do this we had to reach into the user_data object. The Oracle docs for purge_queue_table say that every column in the purge condition must be prefaced with the alias "qtview." This is not strictly true. We found by accident that the "qtview" prefix is only necessary for attributes of the user_data object. Here is how we can do a targeted purge of only one message:

```
DECLARE
   l_purge_opt dbms_aqadm.aq$_purge_options_t;
BEGIN
   l_purge_opt.block := FALSE; -- don't block enqueue or dequeue attempts (this is the default)
   dbms_aqadm.purge_queue_table(
      queue_table     => 'MY_SQT'
     ,purge_condition => 'queue = ''MY_Q'' AND qtview.user_data.text_vc = ''hello world'''
     ,purge_options   => l_purge_opt
   );
END;
```

Browsing Queues

One of our other projects was seeing symptoms very similar to our twice and often twice problem. After days of research, Oracle SRs, and head-scratching, an internal DBA performed SQL tracing and discovered the error deep within the wrapped package DBMS_AQIN. A few more tests and deduction finally led to the conclusion that when queue browsing from a schema other than the queue owner is desired, the queue owner must grant SELECT access on the undocumented AQ$_*queue_table*_F views, of which there is one automatically created for each new queue.

AQ$_JMS_MESSAGE type

The payload used to create the queue, the user-defined type AQ$_JMS_MESSAGE owned by SYS, changed between 9i and 10g. As we were upgrading, this caused errors until I finally figured out that 9i has only one constructor, and it requires an integer, one of the constants defined in the spec of the AQ$_JMS_MESSAGE type. But 10g has several different kinds of constructors, one of which can be a local text message variable. And this type of constructor didn't exist in 9i, so we had to alter our creation scripts until we had migrated to 10g.

```
-- 10g
DECLARE
   l_msg SYS.AQ$_JMS_MESSAGE;
   l_text_msg SYS.AQ$_JMS_TEXT_MESSAGE;
   l_queue_options DBMS_AQ.ENQUEUE_OPTIONS_T;
   l_msg_props DBMS_AQ.MESSAGE_PROPERTIES_T;
   l_msg_id RAW(16);
BEGIN
   l_text_msg := SYS.AQ$_JMS_TEXT_MESSAGE.construct();
   l_text_msg.set_text('<useful message here>');
   l_msg := SYS.AQ$_JMS_MESSAGE.CONSTRUCT(l_text_msg); -- Does not work on 9i

   DBMS_AQ.ENQUEUE(queue_name => 'MYSCHEMA.MY_Q'
                  ,enqueue_options => l_queue_options
                  ,message_properties => l_msg_props
                  ,payload => l_msg
                  ,msgid => l_msg_id);
END;
```

I learned that in 9i and 10g it is more straightforward to use the older constructor, passing in the constant that identifies the type of message you intend to place in the payload.

```
-- 9i and 10g
DECLARE
   l_msg SYS.AQ$_JMS_MESSAGE;
   l_queue_options DBMS_AQ.ENQUEUE_OPTIONS_T;
   l_msg_props DBMS_AQ.MESSAGE_PROPERTIES_T;
   l_msg_id RAW(16);
BEGIN
   l_msg := SYS.AQ$_JMS_MESSAGE.CONSTRUCT(DBMS_AQJMS.JMS_TEXT_MESSAGE);

   l_msg.set_text('<useful message here>');
   DBMS_AQ.ENQUEUE(queue_name => 'MYSCHEMA.MY_Q'
                  ,enqueue_options => l_queue_options
                  ,message_properties => l_msg_props
                  ,payload => l_msg
                  ,msgid => l_msg_id);
END;
```

Genuine Errors

Once all the issues mentioned above were out of the way, things were generally quite clean and worked great. However, having suffered Post Traumatic Stress Disorder from the previous bevy of bugs, we wanted to put some defensive code in place that would let us know if we ever did see a real error in Production.

The solution was fairly simple. We attached an after update trigger to the queue table which looked for changes in the retry count. If it was anything other than its original value, an error had occurred. This would cause a message to be passed to a packaged routine which wrote a helpful email and sent it to the maintenance team.

```
CREATE OR REPLACE TRIGGER my_sqt_au_trg
   AFTER UPDATE ON my_sqt
   FOR EACH ROW
DECLARE
BEGIN

   IF (:old.retry_count <> :new.retry_count) THEN
      my_q_mgr.handle_retry(:old.user_data.text_vc);
   END IF;

END my_sqt_au_trg;
```