Oracle® Tuxedo Message Queue (OTMQ)

Programming Guide

12c Release 1 (12.1.1)

June 2012

Oracle Tuxedo Message Queue Programming Guide

Oracle Tuxedo Message Queue Programming Guide

This chapter contains the following topics:

Programmer Tasks

Oracle Tuxedo Message Queue (OTMQ) provides the following features to Oracle Tuxedo application programmers:

- A set of application programming interfaces to enqueue messages for subsequent process. The queuing service guarantees to execute the enqueue request successfully. A serial of application programming interfaces are provided to dequeue messages in synchronous or asynchronous way.

- The application program can use the same application programming interface as P2P messaging to do publish/subscribe operations. For more information, see Using Publish/Subscribe.
- Besides the message order pre-defined for one queue, the application program can filter the messages being dequeued from the queue by setting filters. For more information, see Using Filters.
- The application program can choose to ensure message delivery to the target queue. For more information, see Using Recoverable Messaging.
- Also the OTMQ supports flexible way to bind queue name and alias, which allows the programmer to decouple the programming and the configurations of queues. For more information, see Using Naming.

Sending and Receiving Messages

OTMQ provides the basic queuing features.

- Application should first attach to a queue using `tpqattach(3c)` before using queuing features and other advanced features provided by OTMQ.
- For message sending, application calls standard enqueue API `tpenqplus(3c)` with specified block, DIP and UMA, to determine whether messaging is synchronous or asynchronous, recoverable or not, and action to take when delivery failed as shown in Listing 1.

Listing 1 Synchronous OTMQ Queue and Enqueue Message Attachment

```
#define MSG_CLAS_EXAMPLES            2000
#define MSG_TYPE_CLIENT_REQ             1

TPQCTL ctl;
Q_ATTACH_CTL qattachctl;
char q_space[16] = "QSPACE";
char q_name[128] = "myqueue1";
long flags;

/* join the application */
if (tpinit(NULL) == -1)
```

```
{
     (void) fprintf(stderr, "failed to join application: %s\n",
                    tpstrerror(tperrno));
     exit(1);
}

memset(&qattachctl, 0x0, sizeof(qattachctl));
qattachctl.attachmode = TMQ_ATTACH_BY_NAME;
qattachctl.qtype = TMQ_ATTACH_PQ;
qattachctl.namespace_list = NULL;
qattachctl.namespace_list_len = 0;
qattachctl.timeout = 30;

memset(&ctl, 0x0, sizeof(ctl));
ctl.flags |= OTMQ;
flags = TPNOTRAN;

if (tpqattach(q_space, q_name, &ctl, &qattachctl, flags) == -1)
{
     (void) fprintf(stderr, "failed to attach q[%s.%s]: %s\n", q_space,
                    q_name, tpstrerror(tperrno));
     (void) tpterm();
     exit(1);
}

/* get request buffer */
if ((reqstr = tpalloc("STRING", NULL, len)) == NULL)
{
     (void) fprintf(stderr, "unable to allocate STRING buffer: %s",
                    tpstrerror(tperrno));
     exit(1);
}

ctl.msg_class = MSG_CLAS_EXAMPLES;  /* user defined message class */
ctl.msg_type = MSG_TYPE_CLIENT_REQ; /* user defined message type */
ctl.block = OTMQ_DEL_WF;              /* use synchronous way */
ctl.DIP = OTMQ_DIP_MEM;               /* interest point */
ctl.uma = OTMQ_UMA_RTS;             /* undelivered message action - return
to sender */
ctl.timeout = 30;

/* enqueue the message into the destination queue */
if (tpenqplus(target_qspace, target_qname, &ctl, reqstr, 0, 0) == -1)
{
     (void) fprintf(stderr, "Failure to enqueue  %s\n",
     tpstrerror(tperrno)); if (tperrno == TPEDIAGNOSTIC)
     {
            (void) fprintf(stderr, "Diagnostic code=[%d]\t",
            ctl.diagnostic);
}
     tpfree((char *) reqstr);
     (void) tpterm();
     exit(1);
}

/* detach from queue */
/* tpqdetach() */
…
```

- For synchronous message receiving, application calls standard dequeue API tpdeqplus(3c) as shown in Listing 2.

Listing 2 Synchronous Message Dequeue

```
char qspacename[16] = "QSPACE";
char qname[128] = "myqueue2";

/* call tpinit to join the application */
/* tpinit() */
…
/* attach to the queue to dequeue message from, then do the dequeue action */
/* tpqattach() */
…

memset(&ctl, 0x0, sizeof(ctl));
ctl.flags |= OTMQ;
flags = TPNOTRAN;

/* get request buffer, allocate a buffer to store the dequeued message */
len = 100;
if ((reqstr = tpalloc("STRING", NULL, len)) == NULL)
{
     (void) fprintf(stderr, "unable to allocate STRING buffer: %s",
            tpstrerror(tperrno));
     (void) tpterm();
     exit(1);
}

/* dequeue the message from the queue */
ctl.timeout = 30;
if (tpdeqplus(qspacename, qname, &ctl, &reqstr, &len, 0) == -1)
{
     if (tperrno == TPEDIAGNOSTIC)
     {
            (void) fprintf(stderr, "Diagnostic code=[%d]\t",
            ctl.diagnostic);
     } else
     {
           (void) fprintf(stderr, "Failure to dequeue  %s\n",
           tpstrerror(tperrno));
     }
     tpfree((char *) reqstr);
     (void) tpterm();
     exit(1);
}

/* detach from queue */
/* tpqdetach() */
…
```

- For asynchronous message receiving, application calls `tpqgetmsga(3c)` as shown in Listing 3.

Listing 3 Asynchonous Dequeue Message

```
/* first define the user action to be done when message arrive */
int gotMessage = 0;

int msgAction(long * flag)
{
     printf("Get asynchronous message [%s],flag=0x%X\n",reqstr,flag);
     gotMessage = 1;
}

int main(int argc, char **argv)
{
     char qspacename[16] = "QSPACE";
     char qname[128] = "myqueue1";
...

     /* join the application */
     /* tpinit() */
     …
     /* attach to the queue to dequeue message from */
     /* tpqattach() */
     …
     memset(&qctl,0,sizeof(qctl));
     qctl.flags |= OTMQ;
     qctl.filter_idx = -1;    /* no message filter designated, get the first
     available message in queue */

     size_user_data=100;
```

```
    if( tpqgetmsga(qspacename,
           qname,
           (TPQCTL *)&qctl,
           (char **)&reqstr,
           (long *)&size_user_data,
           (long *)&msgAction,
           (long *)0,
           (long *)0,
           TPNOTIME)  != 0)
     {
       /* print out the error message string or diagnostic code */
       …
       tpfree((char *) reqstr);
       (void) tpterm();
       exit(1);
       }

       /* continue to do other actions, when message arrived in queue,
       * user action "msgAction" will be called */

       …
}
```

- If received message requires confirmation, application calls `tpqconfirmmsg(3c)` to confirm receipt of the message as shown in Listing 4.

Listing 4 Explicit Confirmation for a Dequeued message

```
/* join the application */
/* tpinit() */
…
/* attach to the queue to dequeue message from, then do the dequeue action */

/* tpqattach() */
…
/* dequeue message */
/* tpdeqplus() */
```

```
…
/* check the message delivery status stored in TPQCTL */
if( ctl.status_block.del_psb_status == OTMQ__CONFIRMREQ)
{
     /* This is a message need to be confirmed explicitly,
     * use the dequeued message sequence to confirm */
     if(tpqconfirmmsg(ctl.seq_number, 0) < 0)
       {
           /* print out the error message string or diagnostic code */
           …
           tpfree((char *) reqstr);
           (void) tpterm();
           exit(1);
     }
}
```

Using Filters

OTMQ provides message filter which allows user to retrieve message that matching the selection criteria defined by the message filter. Application can designate message filter when calling standard dequeue API `tpdeqplus(3c)`, or when calling subscription API `tpqsubscribe(3c)`.

Filter Type

OTMQ supports two types of message filter: simple filter and compound filter. Simple filter has lifecycle of only one-time operation (dequeue or subscription). While the compound filter can be pre-defined and re-used in the subsequent dequeue operations.

Simple Filter

- Filter per subscription

  Message filter can be specified when subscribing the user broadcast message. It only impacts the messages retrieved according to this subscription.

- Filter per operation

  Message filter can be specified when executing a tpdeqplus/tpdequeue. It only impacts the result of this operation itself. For simple filter use, you must set `filter_idx=-1` and `flags|=TPQGETBYFILTER`, otherwise it reports an error.

Compound Filter

- Filter per queue

  Message filter can be defined or canceled via a pair of APIs: `tpqsetselect /tpqcancelselect`. Once a filter is defined, the user can use it in a serial of dequeue or subscription operations.

Filter Format

Different types of message filter have different kinds of format. Following sections describe the selection criteria that can be specified for the simple filter or the compound filter.

Simple Filter

For simple filter, it consists of one of the following selection criteria:

- Default Selection

  Enables application to read messages from the queue based on the order in which they arrived. Applications using default selection will read the next pending message from the message queue. Messages are stored by pre-defined queue orders (FIFO, LIFO, priority, etc.).

- Selection by Message Attribute

  Enables the application to select messages based on the message type, message class or message priority, etc.

Table 1 shows how the selection criteria can be defined as select mode and value pairs.

Table 1 Select Mode

Selection Mode	Selection Variable	Mode Description
OTMQ_PQ_TYPE	Message type value	Selects the first pending message from the attached Primary Queue (PQ) that matches the message type value being specified in the selection variable. TPQCTL->flags must set OTMQ|TPQGETBYFILTER|TPQGETBYMSGTYPE
OTMQ_PQ_CLASS	Message class value	Selects the first pending message from the attached Primary Queue (PQ) that matches the message class value being specified in the selection variable. TPQCTL->flags must set OTMQ|TPQGETBYFILTER|TPQGETBYMSGCLASS
OTMQ_PQ_PRI	• Integer value between 0 and 99 • OTMQ_PRI_ANY • OTMQ_PRI_P0 • OTMQ_PRI_P1	Selects the first pending message with a priority equal to an integer between 0 and 99 inclusive (or equal to the selection variable value) from the attached Primary Queue (PQ). Specifying the direct integer value is the preferred method of selecting message by priority Using OTMQ_PRI_ANY enables the reading of any pending messages of all priorities. Using OTMQ_PRI_P0 enables the application to retrieve pending messages of priority 0 only. Using OTMQ_PRI_P1 enables the strict retrieval of pending messages with a priority of 1.

Listing 5 Dequeue Message with Simple Message Filter

```
#define MSG_CLAS_EXAMPLES            2000
#define MSG_TYPE_CLIENT_REQ             1

TPQCTL ctl;
```

```
….
/* join the application */
/* tpinit() */
…

/* attach to the Qspace */
/* tpqattach() */
…

/* select by message attributes */
ctl.flags |= TPQGETBYMSGCLASS;
ctl.msg_class = MSG_CLAS_EXAMPLES;
ctl.flags |= TMQGETBYMSGTYPE;
ctl.msg_type = MSG_TYPE_CLIENT_REQ;
ctl.flags |= TPQGETBYPRIORITY;
ctl.priority = 50;
…
/* call tpdeqplus to dequeue a message with
* message class is "MSG_CLAS_EXAMPLES",
* message type is "MSG_TYPE_CLIENT_REQ" and
* message priority is 50 */
if (tpdeqplus(qspacename, qname, &ctl, &reqstr, &len, 0) == -1)
{
     /* deal with failed scenario */
     ……
}
…
/* detach from Qspace */
/* tpqdetach() */
…
```

Compound Filter

The compound filter allows application to define complex selection criteria for message reception. The selection criteria array can specifies the queues to search, the priority order of message reception, two comparison keys for range checking, and an order key to determine the order in which messages are selected from the queue.

Application calls `tpqsetselect(3c)` function first to define a filter and gets an index handle as return, which can be used later in the standard dequeue API to retrieve messages.

Also the application can call `tpqcancelselect(3c)` to cancel the compound filter defined before as shown in Listing 6.

Listing 6 Dequeue Message Using Compund Message Filter

```
char qspacename[16] = "QSPACE";
char qname[128] = "myqueue1";
char src_qname[128] = "from_que";
TPQctl ctl;
selection_array_component_tp  selection_array;
short num_masks = 1;
int index_handle = -1;

/* join the application */
/* tpinit() */

/* attach to the Qspace */
/* tpqattach() */

/* set complex filter to dequeue message with specific message class,
  * and from specific queue*/
memset(&selection_array, 0x0, sizeof(selection_array));
selection_array.key_1_offset = OTMQ_CLASS;
selection_array.key_1_size = 4;
selection_array.key_1_value = MSG_CLAS_EXAMPLES;
selection_array.key_1_oper = OTMQ_OPER_EQ;
selection_array.key_2_offset = OTMQ_SOURCE;
selection_array.key_2_size = 4;
selection_array.key_value_qspace = qspacename;
selection_array.key_value_queue = src_qname;
selection_array.key_2_oper = OTMQ_OPER_EQ;

if(tpqsetselect(&selection_array, &num_masks, &index_handle ) == -1)
{
        /* deal with failed scenario */
        …
}

ctl.filter_idx = index_handle; /* using the filter to dequeue */

if(tpdeqplus(qspacename, qname, &ctl, &reqstr, &len, 0) == -1)
{
        /* deal with failed scenario */
        …
}

/* need to cancel the filter using the index */
if(tpqcancelselect(&index_handle)== -1)
{
        /* deal with failed scenario */
        …
}

/* detach from Qspace */
/* tpqdetach() */
…
```

For more information, see `tpqsetselect(3c)` and `tpqcancelselect(3c)` in the Oracle Tuxedo Message Queue Reference Guide.

Using Publish/Subscribe

The publisher broadcast a message by sending the message to a special "topic". Each topic represents a broadcast stream. A broadcast stream is the set of target queues registered to receive messages directed to a particular topic. The subscriber should register first for a topic to receive the specific broadcasting messages.

The OTMQ Message Queue Manager Server is responsible for maintaining lists of user processes that are interested in the specific topic. In addition, the server is responsible for maintaining the various user definable rules (also known as pub/sub filter) that can be used to selectively extract messages from the broadcast stream that are set by the application using the `tpqsubscribe(3c)`.

A publisher can send a broadcast message using `tpqpublish(3c)`, and a subscriber can retrieve the subscribed message from its attached queue.

Sending Broadcast Messages

To broadcast a message, a publisher program directs the message to the topic that identifies the broadcast stream to use for message distribution. When the application issues the `tpqpublish(3c)` function, OTMQ Message Queue Manager Server deals with the `tpqpublish(3c)` call and transparently redirects the message to all corresponding recipients.

OTMQ Message Queue Manager Server delivers only one copy of each message on the broadcast stream to each target queue, regardless of how many selection matches are made by separate subscription rule entries.

Broadcast messages cannot be stored for automatic message recovery.

Receiving Broadcast Messages

To receive broadcast messages, applications use a standard API `tpqsubscribe(3c)` to register for receipt with the local or remote OTMQ Message Queue Manager Server.

The following topics describe:

- Subscribing to Receive Broadcast Messages
- Subscribing to Receive Selected Broadcast Messages
- Subscription Acknowledgement
- Reading Broadcast Messages
- Unsubscribing Receiving Broadcast Messages

Subscribing to Receive Broadcast Messages

To receive broadcast messages, an application registers a queue with a broadcast stream (topic) that managed by the OTMQ Message Queue Manager Server.

The receiver/subscribing applications register for broadcast messages using the function `tpqsubscribe(3c)`. The registration contains the string formatted topic plus any selection criteria (filter) related to messages that the application wishes to receive.

The application subscribe the broadcast messages using the function `tpqsubscribe(3c)` supplied with the following information:

- The topic: the broadcast stream that wants to subscribe
- The target: the target OTMQ Message Queue Manager Server, and the special flag which means Pub/Sub service.
- The source: the queue name which the requesting application is attaching.

Subscribing to Receive Selected Broadcast Messages

Use the message filter of `tpqsubscribe(3c)` to register for selective reception of broadcast messages. This subscription request registers a target queue to receive a copy of all messages on a broadcast stream that meet a single selection rule.

Filter is a string containing a Boolean filter rule that must be evaluated successfully before the OTMQ Message Queue Server distributing the broadcast messages to the subscriber. Filter rules are specific to the types buffers to which they are applied. For messages of string type, the filter rule is a regular expression. For messages of FML buffers, the filter rule is a string that can be passed to the FML Boolean compiler (see Tuxedo ATMI FML function `Fboolco`).

Table 2 shows how the Filter Regular Expression Rules can be defined.

Table 2 Regular Expression Rules

Rule	Matching Text
character	Itself (character is any ASCII character except the special ones mentioned below).
\ character	Itself except as follows: \\-newline \\t-tab \\b-backspace \\r-carriage return \\f-formfeed
\ special-character	Its un-special self. The special characters are `. * + ? | ( ) [ {` and `\\`. -Any character except the end-of-line character (usually newline or NULL). ^-Beginning of the line. $-End-of-line character.

Table 2 Regular Expression Rules

Rule	Matching Text
[class]	any character in the class denoted by a sequence of characters and/or ranges. A range is given by the construct character-character. For example, the character class, [a-zA-Z0-9_], will match any alphameric character or "_". To be included in the class, a hyphen, "-", must be escaped (preceded by a "\\") or appear first or last in the class. A literal "]" must be escaped or appear first in the class. A literal "^" must be escaped if it appears first in the class.
[^ class]	Any character in the complement of the class with respect to the ASCII character set, excluding the end-of-line character.
RE RE	The sequence. (catenation)
RE | RE	Either the left RE or the right RE. (left to right alternation)
RE *	Zero or more occurrences of RE.
RE +	One or more occurrences of RE.
RE ?	Zero or one occurrences of RE.
RE { n }	n occurrences of RE. n must be between 0 and 255, inclusive.
RE { m, n }	m through n occurrences of RE, inclusive. A missing m is taken to be zero. A missing n denotes m or more occurrences of RE.
(RE)	Explicit precedence/grouping.
(RE) $ n	The text matching RE is copied into the nth user buffer. n may be 0 through 9. User buffers are cleared before matching begins and loaded only if the entire pattern is matched.

There are three levels of precedence. In order of decreasing binding strength they are:

- catenation closure (*,+,?,{...})
- catenation
- alternation (|)

As indicated above, parentheses are used to give explicit precedence.

Subscription Acknowledgement

The `tpqsubscribe(3c)` returns a subscription handle back to the subscriber. This handle should be used to unsubscribe the specific subscription.

Reading Broadcast Messages

When a message is sent to a broadcast stream, the OTMQ Message Queue Manager Server will determine which applications have registered to receive that kind of message. Then it automatically sends these messages to the distribution of all matching applications. The receiving application reads the broadcast message from its target queue using the standard dequeue functions. The source queue address of the sender is also provided to the receiving application in the message.

Unsubscribing Receiving Broadcast Messages

An application can withdraw subscribing messages from a broadcast stream by calling the `tpqunsubscribe(3c)`. This action removes the subscription entry from the internal registration tables as shown in Listing 7

Listing 7 Subscribe and Unsubscribe Messages

```
TPEVCTL evctl;
long evt_handle = 0L ;  /* Event Subscription handles */
…
/* join the application */
/* tpinit() */
…

/* attach to the Qspace */
/* tpqattach() */
…
evctl.flags = TPEVQUEUE ;
(void)strcpy (evctl.name1, "QSPACE") ;
(void)strcpy (evctl.name1, "myqueue1") ;
evctl.qctl.flags |= OTMQ;

/* Subscribe */
```

```
evt_handle = tpqsubscribe ("TMQ:QNOT:QSPACE:mytopic",
   NULL,&evctl,TPSIGRSTRT) ;
if (evt_handle == -1L) {
     /* deal with failed scenario */
     …
}
…
/* dequeue subscribed message */
if(tpdeqplus(qspacename, qname, &ctl, &reqstr, &len, 0) == -1)
{
     /* deal with failed scenario */
     …
}

/* Unsubscribe to the subscribed topic */
if (tpqunsubscribe (evt_handle, TPSIGRSTRT) == -1)
{
     /* deal with failed scenario */
     …
}
```

For more information, see tpqsubscribe(3c) and tpqunsubscribe(3c) in the Oracle Tuxedo Message Queue Reference Guide

Using Recoverable Messaging

Applications send messages using the OTMQ standard enqueue function `tpenqplus(3c)` and one of two types of delivery modes: recoverable or non-recoverable. If a message is sent as non-recoverable, the message is lost if it cannot be delivered to the target queue unless the application incorporates an error recovery procedure. If the message is sent as recoverable, OTMQ Message Queue Manager Server and Offline Trade Driver can automatically guarantee delivery to the target queue in spite of system, process, and network failures.

To ensure guaranteed delivery, the OTMQ Message Queue Manager Server writes recoverable messages to nonvolatile storage on the sender or receiver system. Then, if a message cannot be delivered due to an error condition, the OTMQ Offline Trade Driver attempts redelivery of the message by reading it from the recovery journal and redelivering the message to the target queue until delivery is confirmed.

Application developers determine which messages should be sent as recoverable depending upon the needs of the application. Because recoverable messaging requires the extra step of storing the messages on disk, it requires additional processing time and power. To maximize performance, recoverable messaging should only be used when it is critical to application processing.

The OTMQ recoverable messaging feature offers the following benefits:

- Reduces development time by eliminating the need for designing applications to recover messages that cannot be delivered.
- Prevents applications from losing data when applications, systems, or network links fail.
- Simplifies the implementation of an event-driven store and forward capability in networked applications.

OTMQ also offers error recovery features for non-recoverable messages such as the dead letter queue (DLQ) and the ability to return a message to the sender if the message cannot be delivered. This topic describes all of the OTMQ delivery modes to enable you to understand the right choice for your application.

This section contains the following topics

- Choosing a Message Delivery Mode
- How to Send a Recoverable Message
- How to Receive a Recoverable Message
- Using UMAs for Exception Processing

Choosing a Message Delivery Mode

The choice between recoverable and non-recoverable delivery is based upon the needs of the application. To determine the appropriate method for sending a message, the application developer decides:

- Does the application need to know if the message arrived at the target queue?
- If notification is required, how far must the message get before the sender program receives notification that the message has arrived?

- Should the application wait for notification or should it continue processing and receive notification through an asynchronous acknowledgment message?
- If the message is designated as recoverable, does the application need to know if the message has been stored by the recovery system?
- If the message is designated as recoverable, what should happen if it cannot be stored by the recovery system?

The delivery mode specified in `tpenqplus(3c)` function determines:

- Whether the message is sent as recoverable or non-recoverable
- Whether a blocking or non-blocking mode is selected
- Whether the sender program receives notification and how it is received
- The point in the message flow at which the notification is sent

OTMQ uses message recovery journals to store messages that are designated as recoverable. The message recovery journal on the local system is called the store and forward (SAF). The message recovery journal on the remote system is called the destination queue file (DQF). If a recoverable message cannot be delivered, it is stored in either the SAF or DQF queue and is automatically re-sent once communication with the target group is restored.

OTMQ uses auxiliary journals to provide additional message recovery capabilities. The dead letter queue (DLQ) is a memory-based queue for storing undeliverable messages. The dead letter journal (DLJ) provides disk storage for messages that could not be stored for automatic recovery. Undelivered messages stored in the DLQ or DLJ can be retrieved under user or application control by using OTMQ's Journal API `tpqreadjrn(3c)`.

The post confirmation journal (PCJ) stores successfully confirmed recoverable messages.

If the OTMQ message recovery system is unable to store the message, the undeliverable message action (UMA) is taken. Some UMAs enable the message to be recovered at a later time under user or application control.

Choosing Recoverable or Non-recoverable Delivery Mode

The delivery mode consists of two components, the block type (block) and the delivery interest point (DIP). Specify the block and DIP in the TPQCTL structure.

- block - indicates how the sender program wants to receive information about the delivery of the message. You can either wait for the operation to complete (WF), or receive notification in an asynchronous message (AK), or choose not to receive notification (NN) at all.

- DIP - determines whether the message is designated as recoverable. When the message reaches the delivery interest point, a notification message is sent (if requested) and the call returns control to the sender program or OTMQ Message Queue Manager Server delivers the asynchronous acknowledgment message.

Non-recoverable delivery interest points enable the sender program to receive notification when the message is stored in the target queue (MEM), when the message is read from the target queue (DEQ), or when the message is read from the target queue and explicitly confirmed by the receiver program using the tpqconfirmmsg(3c) function (ACK).

When a recoverable delivery interest point is selected, the message is stored on disk for automatic recovery. Recoverable delivery interest points enable the sender program to store the message in the local recovery journal (SAF), store the message in the remote recovery journal (DQF), or store the message in the remote recovery journal and receive notification when the message is confirmed by the target application (CONF).

OTMQ does not support all possible combinations of block type and delivery interest points. Table 3 lists the valid delivery modes and their meanings.

Table 3 Delivery Modes

Delivery Mode	Description
(Recoverable Delivery Modes)	

block = OTMQ_DEL_AK
DIP = OTMQ_DIP_CONF
	Send acknowledgment message when the message recovery system confirms message delivery from the remote recovery journal.

block = OTMQ_DEL_AK
DIP = OTMQ_DIP_DQF
	Send acknowledgment message when the message is stored in the remote recovery journal.

block = OTMQ_DEL_AK
DIP = OTMQ_DIP_SAF
	Send acknowledgment message when the message is stored in the local recovery journal.

block = OTMQ_DEL_NN
DIP = OTMQ_DIP_DQF
	Deliver message to the remote recovery journal but do not block and do not send notification.

block = OTMQ_DEL_NN
DIP = OTMQ_DIP_SAF
	Deliver message to the local recovery journal but do not block and do not send notification.

Table 3 Delivery Modes

Delivery Mode	Description

block = OTMQ_DEL_WF
DIP = OTMQ_DIP_CONF
	Block until the message is stored in the remote recovery journal and confirmed by the target application.

block = OTMQ_DEL_WF
DIP = OTMQ_DIP_DQF
	Block until the message is stored in the remote recovery journal.

block = OTMQ_DEL_WF
DIP = OTMQ_DIP_SAF
	Block until the message is stored in the local recovery journal.

(Non-Recoverable Delivery Modes)	

block = OTMQ_DEL_AK
DIP = OTMQ_DIP_ACK
	Send acknowledgment message when the receiver program explicitly confirms delivery using tpqconfirmmsg(3c).

block = OTMQ_DEL_AK
DIP = OTMQ_DIP_DEQ
	Send acknowledgment message when the message is removed from the target queue.

block = OTMQ_DEL_AK
DIP = OTMQ_DIP_MEM
	Send acknowledgment message when the message is stored in the target queue.

block = OTMQ_DEL_NN
DIP = OTMQ_DIP_MEM
	Deliver message to the target queue but do not block and do not send notification.

block = OTMQ_DEL_WF
DIP = OTMQ_DIP_ACK
	Block until the receiver program explicitly confirms delivery using tpqconfirmmsg(3c)

block = OTMQ_DEL_WF
DIP = OTMQ_DIP_DEQ
	Block until the message is removed from the target queue.

block = OTMQ_DEL_WF
DIP = OTMQ_DIP_MEM
	Block until the message is stored in the target queue.

Non-recoverable message delivery is the fastest and most efficient way to send messages. Use non-recoverable delivery modes if:

- High messaging rates are required by the application (hundreds or thousands of messages per second).

- The message content has a finite lifetime; therefore, the value of the information is stale if not received and processed quickly.
- The message is sent locally between two applications in the same message queuing group that tightly cooperate in the processing of an event.
- The message is a control message that causes a component of an application to change state.

Recoverable message delivery is the safest way to send a message; however, it adds significant processing overhead because each message must be stored before it is sent. Use recoverable delivery modes if:

- It is useful to know that the message has arrived; however, the sender does not need to know the state of the receiver.
- The message content should not be lost by the application system.
- The application can tolerate the increased system load and slower messaging rate caused by sending the message

Choosing an Undeliverable Message Action

Using the `tpenqplus(3c)` function in conjunction with the delivery argument, you can use the UMA argument to specify what should happen to the message if it cannot be delivered to the delivery interest point.

With non-recoverable messaging, the UMA indicates the action to be taken if the message cannot be stored in target queue. If a UMA is not specified, OTMQ will take the default action of discarding the message.

With recoverable messaging, the UMA indicates the action to be taken if the message cannot be stored in either the SAF or DQF queues. You must specify a UMA with recoverable delivery modes because your application must perform the exception processing when the message cannot be guaranteed for delivery by OTMQ Message Queue Manager Server.

With recoverable messaging, the UMA may be taken when:

- OTMQ is unable to write to the local SAF queue where the message is designated to be recoverable.
- The cross-group connection to the remote target group is down and the message designated as recoverable on the remote node (DQF) cannot be stored.
- The system resources used by the message recovery system are exhausted.

Table 4 lists the five valid UMAs.

Table 4 UMAs

UMA	Description
OTMQ_UMA_DISC	Discard - the message is deleted.
OTMQ_UMA_RTS	Return to sender - the message is delivered to the sender's response queue.
OTMQ_UMA_SAF	Store and forward - the message is written to the message recovery journal on the sender system.
OTMQ_UMA_DLQ	Dead letter queue - the message is written to the dead letter queue.
OTMQ_UMA_DLJ	Dead letter journal - the message is written to the DLJ.

How to Send a Recoverable Message

To send a recoverable message, use the `tpenqplus(3c)` function supplying the appropriate block type, DIP and UMA in the TPQCTL structure.

In addition, the application should:

- Specify a timeout value when sending recoverable messages with blocking delivery modes.
- Verify the delivery outcome of a send operation from PSB in TPQCTL structure. If the message was failed to be stored by the OTMQ Message Queue Manager Server, the application must check to make sure that the UMA was successfully executed.

The message flow for sending a recoverable message is:

- The application sends a message using tpenqplus(3c) function and with the appropriate block, DIP and UMA arguments.
- The OTMQ Message Queue Manager Server first writes the message to the recovery journal queue on the local (SAF) or remote system (DQF) depending upon the delivery mode specified.
- If the sender is blocked, it continues processing once the message reaches the delivery interest point. If the sender requests notification, it received an acknowledgement message once the message reaches the delivery interest point.

For more information, see `tpenqplus()` in the Oracle Tuxedo Message Queue Reference Guide.

How to Receive a Recoverable Message

To receive a recoverable message, use the `tpdeqplus(3c)` function. When a recoverable message is delivered to the target queue, the application must perform the following:

- Confirm message receipt, which allows the Offline Trade Driver (`TuxMQFWD(5)`) to delete the message being stored in the recovery journal queue before delivery.
- Check for duplicate messages via return code. Duplicate messages may be sent by the Offline Trade Driver if the application didn't confirm message receipt in time. For more information, see `tpdeqplus(3c)` in the Oracle Tuxedo Message Queue Reference Guide.

The message flow for receipt of a recoverable message is as follows:

- A message is read from the message recovery journal queue by the Offline Trade Driver and sent to the target queue of the receiver.
- The receiver reads the message by calling tpdeqplus(3c) function.
- If the queue is configured for explicit confirmation, the receiver should call the `tpqconfirmmsg(3c)` function to acknowledge receipt of the recoverable message using the message sequence number assigned by the OTMQ Message Manager Server when the message was sent. If the queue is configured for implicit confirmation, the acknowledge message will be sent by `tpdeqplus(3c)` automatically after the recoverable message is dequeued successfully. For more information, see `tmqadmin(1)` in the Oracle Tuxedo Message Queue Reference Guide.
- The `tpqconfirmmsg(3c)` function sends acknowledge notification to the Offline Trade Driver to notify the successful message delivery, which allows the Offline Trade Driver to remove the message from the message recovery journal queues.

Using UMAs for Exception Processing

Using Discard UMA

When the DISC UMA is used, the message is discarded if it cannot be delivered to the delivery interest point specified in the delivery mode argument. The DISC UMA is used when the sender program will handle each exception as it occurs. OTMQ can discard the undeliverable message because the message content is still available in the context of the sender program.

Using the Return-to-Sender UMA

When the RTS UMA is used, the message is redirected to the response queue of the sender program if it cannot be delivered to the delivery interest point specified in the delivery mode argument. The RTS UMA is used when the sender program does not want to process each exception as it occurs. Instead, the sender program redirects undeliverable messages to its main input stream for error handling.

The advantage of using the RTS UMA is that the sender program attaches to one queue and acts upon each message as it is read. The sender program must read the PSB status delivery value of each message to determine if the message is new or an undeliverable message. Messages that could not be stored by the message recovery system and require error handling have a return status of OTMQ__MSGUNDEL.

Using the SAF UMA

When the SAF UMA is used, the message is stored in the local journal queue if the message recovery system is unable to store it in the remote journal queue. The SAF UMA can be used with recoverable delivery interest points of DQF and CONF; however, it does not work with the WF_SAF delivery mode.

Use of the SAF UMA helps to manage the flow control between the sender and receiver systems. If the message cannot be written to the remote journal queue due to insufficient resources or a cross-group link failure, the message will be written to the local journal queue.

Note: SAF here means nearly the same as "SAF" DIP. Once message cannot be delivered, store message into SAF queue.

Using the Dead Letter Queue UMA

When the DLQ UMA is used, the message is redirected to the dead letter queue if it cannot be delivered to the delivery interest point specified in the delivery mode argument. The DLQ UMA is used when the sender program wants to centralize error handling for undeliverable messages in a designated queue while allowing each message to be handled separately.

A dead letter queue is part of the standard group configuration for each OTMQ message queuing group. It provides memory-based storage of all undeliverable messages for the group that could not be stored for automatic recovery. The dead letter queue is defined as queue number 96 It is created automatically by `tmqadmin(1) qspacecreate` command.

To use the dead letter queue, the sender program calls the `tpenqplus(3c)` function specifying the appropriate delivery argument and using OTMQ_UMA_DLQ as the UMA argument. Any messages that cannot be delivered to the receiver program are written to the dead letter queue of the sender's group. An application program can use `tpqreadjrn(3c)` function to retrieve undelivered messages and use the `tpenqplus(3c)` function to attempt redelivery.

An advantage of using the dead letter queue is the ability to recover undeliverable messages on a one-by-one basis. The sender program or another process within the application can attach to the DLQ and handle error recovery for each undeliverable message. A disadvantage of using the dead letter queue is the lack of disk storage for undelivered messages. A system failure on the sending node will cause all undelivered messages in the dead letter queue to be lost.

Using the Dead Letter Journal

When the DLJ UMA is used, the message is written to an auxiliary journal, queue number is 196 (the dead letter journal) if it cannot be delivered to the delivery interest point specified in the delivery mode argument. This UMA can only be used for recoverable messages. The DLJ UMA is used when the sender program needs to centralize error handling procedures and the application can support the resending of many messages from DLJ queue at a delayed interval. Storing undeliverable messages in DLJ queue ensures that they will not be lost if the system goes down, and allows redelivery attempts under user or application control.

The dead letter journal provides disk storage for messages that could not be stored for automatic recovery. It is created automatically by `tmqadmin(1) qspacecreate` command.

To use the dead letter journal, the sender program uses the `tpenqplus(3c)` function specifying the appropriate delivery argument and OTMQ_UMA_DLJ as the UMA argument. Any messages that cannot be stored by the message recovery system are written to the dead letter journal of the sender's group. An application program can use `tpqreadjrn(3c)` function to retrieve undelivered messages and use the `tpenqplus(3c)` function to attempt redelivery as shown in Listing 8.

Listing 8 Using UMA with Undelivered Message Example

```
TPQCTL ctl;
int type;
…

/* join the application */
/* tpinit() */

/* attach to the QSpace */
/* tpqattach() */

/* get request buffer */
if ((reqstr = tpalloc("STRING", NULL, len)) == NULL)
{
     (void) fprintf(stderr, "unable to allocate STRING buffer: %s",
               tpstrerror(tperrno));
     exit(1);
}

ctl.block = OTMQ_DEL_WF;              /* use synchronous way */
ctl.DIP = OTMQ_DIP_SAF;               /* interest point */
ctl.uma = OTMQ_UMA_DLJ;              /* undelivered message action - Dead
Letter Journal*/

/* enqueue the message into the destination queue */
if (tpenqplus(target_qspace, target_qname, &ctl, reqstr, 0, 0) == -1)
{
     /* deal with failed scenario */
     …
}
…
…
/* done other works, handle failed message in DLJ before exit */
ctl.flags |=OTMQ;
ctl.flags |= TPQREADJRN;
type = DLJ_HANDLE;
if (tpqreadjrn(myqspace, myqueue, &ctl, &rcv_buf, &len, 0) == -1)
{
     /* deal with failed scenario */
     …
}

/* enqueue the failed message again */
if (tpenqplus(target_qspace, target_qname, &ctl, rcv_buf, 0, 0) == -1)
{
     /* deal with failed scenario */
     …
}
…
/* detach from the Qspace */
/* tpqdetach() */
…
```

The DIP and UMA Support List

Table 5 lists the valid delivery modes and UMA combinations.

Table 5 DIP and UMA Support List

	UMA				
Delivery Mode	SAF	DLJ	DLQ	RTS	DISC
block = OTMQ_DEL_NN DIP = OTMQ_DIP_MEM	N	N	Y	Y	Y
block = OTMQ_DEL_WF DIP = OTMQ_DIP_MEM	N	N	Y	Y	Y
block = OTMQ_DEL_AK DIP = OTMQ_DIP_MEM	N	N	Y	Y	Y
block = OTMQ_DEL_AK DIP = OTMQ_DIP_DEQ	N	N	Y	Y	Y
block = OTMQ_DEL_WF DIP = OTMQ_DIP_ACK	N	N	Y	Y	Y
block = OTMQ_DEL _AK DIP = OTMQ_DIP_ACK	N	N	Y	Y	Y
block = OTMQ_DEL _WF DIP = OTMQ_DIP_DEQ	N	N	Y	Y	Y
block = OTMQ_DEL _AK DIP = OTMQ_DIP_SAF	N	Y	Y	Y	Y

Table 5 DIP and UMA Support List

	UMA				
block = OTMQ_DEL _WF DIP = OTMQ_DIP_SAF	N	Y	Y	Y	Y
block = OTMQ_DEL _NN DIP = OTMQ_DIP_SAF	N	Y	Y	Y	Y
block = OTMQ_DEL _AK DIP = OTMQ_DIP_CONF	Y	Y	Y	Y	Y
block = OTMQ_DEL _WF DIP = OTMQ_DIP_CONF	Y	Y	Y	Y	Y
block = OTMQ_DEL _NN DIP = OTMQ_DIP_DQF	Y	Y	Y	Y	Y
block = OTMQ_DEL _WF DIP = OTMQ_DIP_DQF	Y	Y	Y	Y	Y
block = OTMQ_DEL _AK DIP = OTMQ_DIP_DQF	Y	Y	Y	Y	Y

Using Naming

In OTMQ configuration, each message queue gets a name when created, and also can get an alias at runtime. Naming is a powerful feature that enables OTMQ applications to identify message queues by name/alias whether they reside on the local system or on another system.

Application developers use the OTMQ naming feature to separate their applications from the underlying OTMQ environment configuration. By referring to message queues by name/alias in the applications, developers do not have to modify their applications code when the OTMQ environment configuration changes.

The user must configure `TMQ_NA(5)` server in UBB to take advantage of the naming service.

Naming Service

Naming service is provided by the OTMQ Naming Server. It can access and manage both the local namespace and global namespace for the runtime queue lookup when an application refers to a queue by alias, or dynamic binding a queue alias to a specified queue name. OTMQ Naming Server provides two services: one for the local scope alias lookup (Local Naming Service), another for the global's (Global Naming Service).

Name Scope

When a name or alias is defined for message queue, it is assigned a name scope. Queue name or alias can have a local (intra-qspace) or global (inter-qspace) scope. A local alias can be used by applications running in the same queue space in which the alias was defined. A global alias can be used by any applications.

Name Space

A name space is the repository where message queue alias and their associated message queue address (queue space and queue name) are stored. OTMQ Naming Server must look up the alias in the name space to find its associated actual queue address in order to send a message to the named queue.

OTMQ Naming Server uses three levels of name spaces: process, local (qspace-wide) and global (cross qspace wide). When OTMQ Naming Server start up, the local scope alias will be stored in local name space. The global scope alias will be stored in global name space. The process name space is an application cache used to improve performance. The alias can be cached at different level name space, user can bypass caching when using `tpqlocate(3c)` if they prefer accuracy over performance.

Process Level Name Space

When application attaches to the OTMQ, application automatically creates the empty process name space. When this process locates/binds an alias for the first time, it is cached in the process name space.

Local Name Space

When OTMQ Naming Server starts up, it automatically creates the local name space. Also local scope queue alias defined by applications will be cached in the local name space.

Global Name Space

To create the global name space, use a flat file system by creating the directory in which the OTMQ naming service will maintain the name space.

To use global naming, you must create a global namespace on the nodes on which the Naming Server runs. OTMQ allows user to configure at most two global naming services (primary and backup). To enable the backup naming service to take responsibility when the primary one is down, all the global naming services must be configured to use the same global name space. Therefore, when configured naming services run on different systems, they must use a shared file system.

After creating the name space, you must set the `DMQNS_DEVICE` environment variable to specify a device name for the name space because access to the name space for global naming is system dependent. Therefore, when a global naming service is configured, it must be told what device name to be used when it accesses this name space. This is done by setting the environment variable `DMQNS_DEVICE` as follows:

- For Windows NT, it should be set to a drive letter followed by a colon (for example, c:> o a full qualified share name (e.g. `\\machine\share)`
- For UNIX, it should be set to a file system specification (for example, / or `/usr` or `/mnt/dmqns`)

Note: this environment variable need only be set for the OTMQ Naming Server which provides the naming services. To use the global naming service, at least some portion of the global namespace file path must be specified. For example on UNIX, you can define it as "`/u/mydir`".

When an application refers to a queue by alias using the `tpqlocate(3c)` or the `tpqbind(3c)` functions, it can specify the alias as one of the following:

- unqualified name: The application uses only the queue alias such as "widgets" and does not specify the path. The naming service automatically prefixes the name with the value of the environment variable `DMQNS_DEVICE`. Furthermore, it prefix the value of the environment variable DMQNS_DEFAULTPATH begins with a "/". For example, if the `DMQNS_DEVICE` environment variable is set to "dev" and the DMQNS_DEFAULTPATH is set to "`/inventory`", the naming service would search for the name "widgets" in: `/dev/inventory/widgets`
- partially qualified name: The application specifies the queue alias and a portion of the path name. The naming service automatically prefixes the pathname and queue alias with the device specified as the `DMQNS_DEVICE` environment variable and the setting of the DMQNS_DEFAULTPATH environment variable. For example, if the `DMQNS_DEVICE`

environment variable is set to "/dev" and the DMQNS_DEFAULTPATH is set to "`/inventory`", the naming service would search for the name "`test/widgets`" in: `/dev/inventory/test/widgets`.

- fully qualified name: The application specifies that the alias is a fully qualified name using "/" as the first character of the name. When the first character of a name begins with "/", the naming service does not prefix any information to the name other than the device name specified by the `DMQNS_DEVICE` environment variable. This means that a fully qualified name includes the full path name and queue name. For example, if the `DMQNS_DEVICE` environment variable is set to "dev" and the DMQNS_DEFAULTPATH is set to "/inventory", the naming service will search for the name "/production/test/widgets" in: /dev/production/test/widgets. Listing 9 shows a global namespace file example.

Listing 9 Global Namespace File Example

```
PrimaryQ_1        0.0        L
myqueue1         0.0        G
MRQ13_1         1.13        L
SQ14_2           0.0        G
```

Attaching and Locating Queues

An application must attach to a queue using the `tpqattach(3c)` function before reading message from or sending message to a queue. It can identify the queue by its alias or its name. When sending a message, the target queue is always identified by its name. An application can directly use the queue name or it can use the `tpqlocate(3c)` function to derive the queue name from its alias.Listing 10 shows locating queue example.

Listing 10 Locating Queue Example

```
static char q_space[16] = "QSPACE";
static char q_name[128] = "myqueue1";
…
Q_NAME_CTL  name_ctl;
long scope = NAME_SCOPE_P;
```

```
/* join the application */
/* tpinit() */

/* attach to the QSpace */
/* tpqattach() */

/* locate queue named "Primary_queue" */
name_ctl.pName = "Primary_queue";
wait_mode = OTMQ_WF_RESP;  /* use synchronous way to get response */

if(tpqlocate(q_space, &name_ctl, 0, NULL, &scope, wait_mode, 0) == -1)
{
     /* deal with failed scenario */
     …
}
```

Static and Dynamic Binding of Queue Aliases

OTMQ offers two approaches to associating a queue alias with a queue address: static and dynamic.

Static binding refers to associating a queue name with a queue alias using the name space file. To enable such association, you can specify the name space file when creating the queue space; or you can stop the Naming Server, then update the queue space to specify a name space file, then restart the Naming Server again. For more information, see `tmqadmin(1)` in the Oracle Tuxedo Message Queue Reference Guide.

Dynamic binding refers to the use of `tpqbind(3c)` to associate a queue alias to a queue name at application runtime. The queue alias will not be bound to a specific queue name until the `tpqbind(3c)` successfully return. To modify such association, the application must first unbind the queue alias from the specific queue name using `tpqbind(3c)`, and use the same API to associate another queue alias to the queue name again. When the application detach from the queue or exit the queue space, the Naming Server will unbind the association for this application automatically. shows a dynamic binding queue example.

Listing 11 Dynamic Binding Queue Example

```
static char q_space[16] = "QSPACE";
static char q_name[128] = "myqueue1";

…
Q_NAME_CTL  name_ctl;
long scope = NAME_SCOPE_G;

name_ctl.pName = "Primary_queue";
name_ctl.pGroup = q_space;
name_ctl.pQueue = q_name;

bind_time_out = 30;

if(tpqbind(q_space, &name_ctl, &scope, bind_time_out) == -1)
{
     /* deal with failed scenario */
     …
}
```

For more information, see tpqlocate(3c) and tpqbind(3c) in the Oracle Tuxedo Message Queue Reference Guide.

Using WS SAF

In WS mode, OTMQ messages that are sent using a recoverable delivery mode are written to the local store-and-forward (SAF) journal (`tmqsaf.jrn`) when the connection to the server system is not available.

When the feature is enabled, messages sent using the following reliable delivery modes are saved to the journal:

```
OTMQ_DIP_MEM & OTMQ_DEL_WF (using OTMQ_UMA_SAF)
OTMQ_DIP_DQF & OTMQ_DEL_WF
OTMQ_DIP_DQF & OTMQ_DEL_AK
OTMQ_DIP_SAF & OTMQ_DEL_WF
```

`OTMQ_DIP_SAF & OTMQ_DEL_AK`

OTMQ WS configuration options allow the SAF journal to be configured as follows:

- A fixed-size file that does not reuse disk blocks
- A fixed-size file that reuses (cycles) disk blocks
- A dynamic file that grows indefinitely until no more disk blocks are available

These options allow you to determine how disk resources are used for message journals. Journal files that grow indefinitely periodically allocate an extent of disk blocks as needed to store messages. When all messages are sent from the SAF and the journal is empty, the disk blocks used by the journal are freed and the journal file is removed. Journal file size is in units of disk block size (4096 bytes).

Building Applications

As counterparts of Tuxedo buildclient(1) and buildserver(1) commands, OTMQ provides `buildqclient(1)` and `buildqserver(1)`.

`buildqclient(1)` is used to construct an OTMQ client module. The command combines the supplied files with the standard Oracle Tuxedo ATMI libraries and OTMQ libraries to form a load module.

`buildqserver(1)` is used to construct an OTMQ server load module. The command combines the supplied files with the standard server main routine, the standard Oracle Tuxedo ATMI libraries and OTMQ libraries to form a load module.

For more information, see buildqclient(1) and buildqserver(1) in the Oracle Tuxedo Message Queue Reference Guide.