THE ENTERPRISE MIDDLEWARE SOLUTION

BEA TUXEDO

/Q Guide

BEA TUXEDO Release 6.5
Document Edition 6.5
February 1999

Copyright

Restricted Rights Legend

Trademarks or Service Marks

BEA TUXEDO /Q Guide

Document Edition	Date	Software Version
6.5	February 1999	BEA TUXEDO Release 6.5

Contents

3. BEA TUXEDO System/Q C Language Programming

4. BEA TUXEDO System/Q COBOL Language Programming

A. A Sample Application

1 Introduction and Overview of BEA TUXEDO System/Q

General Description

BEA TUXEDO System/Q allows messages to be queued to stable storage for later processing. Primitives are added to the BEA TUXEDO application-transaction manager interface, (ATMI), that provide for messages to be added to or read from stable-storage queues. Reply messages and error messages can be queued for later return to client programs. An administrative command interpreter is provided for creating, listing and modifying the queues. Prewritten servers are included to accept requests to enqueue and dequeue messages, to forward messages from the queue for processing and to manage the transactions that involve the queues.

This chapter describes the elements that make up the BEA TUXEDO System/Q feature.

A Picture that Explains Everything . . . Almost

Figure 1-1 is a diagram that shows the components of the queued message facility. We'll use the figure to explain how administrators and programmers work with the feature to define it and use it to queue a message for processing and get back a reply.

A queue space is a resource. Access to the resource is provided by an X/OPEN XA-compliant resource manager interface. This interface is necessary so that enqueuing and dequeuing can be done as part of a 2-phase committed transaction in coordination with other XA-compliant resource managers.

Administrative Tasks

The BEA TUXEDO administrator is responsible for defining servers and creating queue space and queues like those shown between the vertical dashed lines in Figure 1-1.

The administrator must define at least one queue server group with `TMS_QM` as the transaction manager server for the group.

Two additional system-provided servers need to be defined in the configuration file. These servers perform the following functions:

- The message queue server, `TMQUEUE`(5), is used to enqueue and dequeue messages. This provides a surrogate server for doing message operations for clients and servers, whether or not they are local to the queue.
- The message forwarding server, `TMQFORWARD`(5), is used to dequeue and forward messages to application servers. The BEA TUXEDO system provides a `main`() for servers that handles server initialization and termination, allocates buffers to receive and dispatch incoming requests to service routines, and routes replies to the correct destination. All of this processing is transparent to the application. Existing servers do not dequeue their own messages or enqueue replies. One goal of BEA TUXEDO System/Q is to be able to use existing servers to service queued messages, without change. The `TMQFORWARD` server dequeues a message from one or more queues in the queue space, forwards the message to a server with a service that is named the same as the queue, waits for the reply, and queues the success reply or failure reply on the associated reply or failure queues, respectively, as specified by the originator of the message (if the originator specified a reply or failure queue).

An administrator also must create a queue space using the queue administration program, qmadmin(1). The queue space contains a collection of queues. In Figure 1-1, for example, four queues are present within the APP queue space. There is a one-to-one mapping of queue space to queue server group since each queue space is a resource manager instance and only a single RM can exist in a group.

The notion of queue space allows for reducing the administrative overhead associated with a queue by sharing the overhead among a collection of queues in the following ways:

- The queues in a queue space share the stable storage area for messages.
- A single message queue server, TMQUEUE in Figure 1-1, can be used to enqueue and dequeue messages for multiple queues within a single queue space.
- A single message forwarding server, TMQFORWARD in Figure 1-1, can be used to dequeue and forward messages for multiple queues within a single queue space.
- A single transaction manager server, TMS_QM in Figure 1-1, can be used to complete transactions for multiple queues within a single queue space.
- The administrator can define a single server group in the application configuration for the queue space by specifying the group in UBBCONFIG or by using tmconfig(1) to add the group dynamically.
- Finally, when the administrator moves messages between queues within a queue space the overhead is less than if the messages were in different stable storage areas, because a one-phase commit can be done.

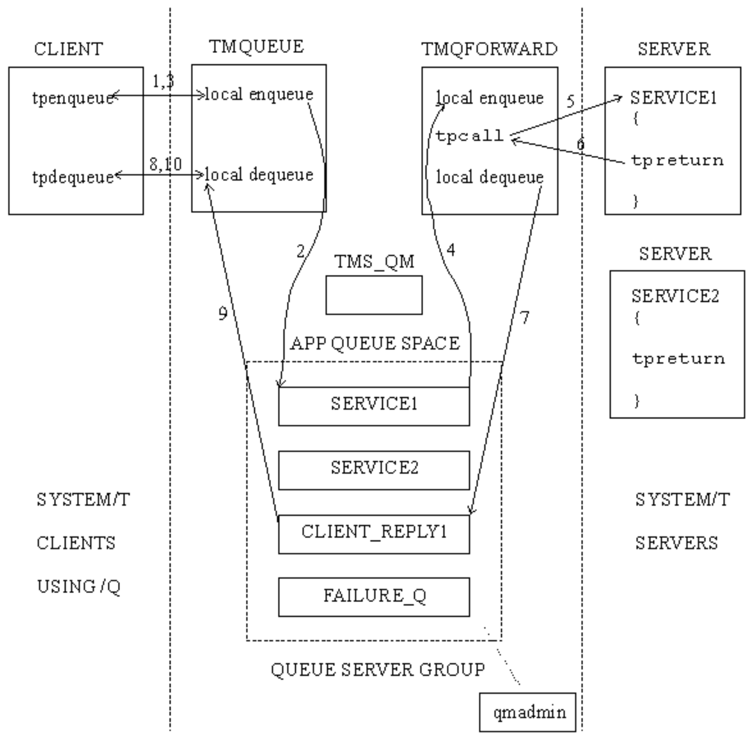
Part of the task of defining a queue is specifying the order for messages on the queue. Queue ordering can be time-based, priority based, FIFO or LIFO, or a combination of those criteria.

The administrator specifies one or more of these sort criteria for the queue; the most significant criteria first. The FIFO and LIFO values can only be the least significant sort criteria. Messages are put on the queue according to the specified sort criteria and dequeued from the top of the queue. The administrator can configure as many message queuing servers as are needed to keep up with the requests generated by clients for the stable queues.

Data-dependent routing can be used to route between multiple server groups with servers offering the same service.

For housekeeping purposes, the administrator can set up a command to be executed when a threshold is reached for a queue that does not routinely get drained. This can be based on the bytes, blocks or percentage of the queue space used by the queue or the number of messages on the queue. The command might boot a `TMQFORWARD` server to drain the queue or send mail to the administrator for manual handling.

Figure 1-1 Queued Message Facility



Programmer's Tasks

In Figure 1-1 (steps 1, 2, 3), a client enqueues a message to the `SERVICE1` queue in the APP queue space using `tpenqueue`(3c). Optionally, the name of a reply queue and a failure queue can be included in the call to `tpenqueue`(). In the example they are the queues `CLIENT_REPLY1` and `FAILURE_Q`. The client can specify a *correlation identifier* value to accompany the message. This value is persistent across queues so that any reply or failure message associated with the queued message can be identified when it is read from the reply or failure queue.

The client can use the default queue ordering (for example, a time after which the message should be dequeued), or can specify an override of the default queue ordering (asking, for example, that this message be put at the top of the queue or ahead of another message on the queue). `tpenqueue`() sends the message to the `TMQUEUE` server, the message is queued to stable storage, and an acknowledgment (step 3) is sent to the client; the acknowledgment is not seen directly by the client but can be assumed when the client gets a successful return. (A failure return includes information about the nature of the failure.)

A message identifier assigned by the queue manager is returned to the application. The identifier can be used to dequeue a specific message. It can also be used in another `tpenqueue`() to identify a message already on the queue that the subsequent message should be enqueued ahead of.

Before an enqueued message is made available for dequeuing, the transaction in which the message is enqueued must be committed successfully.

When the message reaches the top of the queue, the `TMQFORWARD` server dequeues the message and forwards it, via `tpcall`(3c), to a service with the same name as the queue name. In Figure 1-1 the queue and the service are named `SERVICE1` and steps 4, 5, and 6 in the figure show this. The client identifier and the application authentication key are set to the client that caused the message to be enqueued; they accompany the dequeued message as it is sent to the service.

When the service returns a reply, `TMQFORWARD` enqueues the reply (with an optional user-return code) to the reply queue (step 7 in the figure).

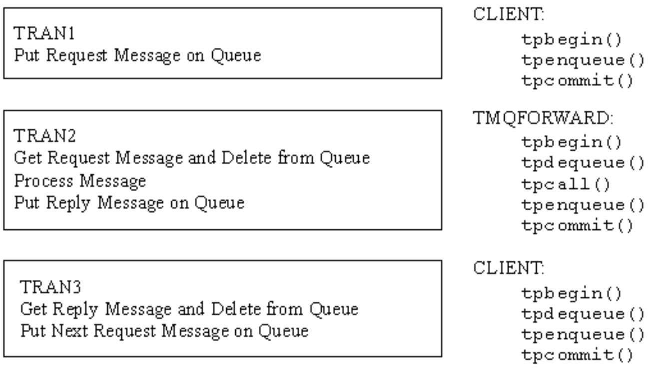
Sometime later, the client uses `tpdequeue`(3c) to read from the reply queue, `CLIENT_REPLY1`, to get the reply message (steps 8, 9 and 10 in Figure 1-1). Messages on the reply queue are not automatically cleaned up; they must be dequeued, either by an application client or server, or by a `TMQFORWARD` server.

Transaction Management

With regard to transaction management, one goal is to ensure reliability by enqueuing and dequeuing messages within global transactions. However, a conflicting goal is to reduce the execution overhead by minimizing the number of transactions that are involved.

An option is provided for the caller to enqueue a message in a transaction separate from any transaction in which the caller is involved (decoupling the queuing from the caller’s transaction). However, a timeout in this situation leaves it unknown as to whether or not the message is enqueued.

Figure 1-2 Transaction Demarcation



A better approach is to enqueue the message within the caller’s transaction, as is shown in Figure 1-2. In this example, the client starts a transaction, queues the message and commits the transaction. The message is dequeued within a second transaction started by `TMQFORWARD`; the service is called with `tpcall`(), is executed and the reply is enqueued within the same transaction. A third transaction, started by the client, is used to dequeue the reply (and possibly enqueue another request message). In ongoing processing the third and first transactions can meld into one since enqueuing the next request can be done in the same transaction as dequeuing the response from the previous request.

Note: The system allows you to dequeue a response from one message and enqueue the next request within the same transaction, but does not allow you to enqueue a request and dequeue the response within the same transaction. The transaction in which the request is enqueued must be successfully committed before the message is available for dequeuing.

Handling Reply Messages

A reply queue can be either specified or not by the application when calling `tpenqueue`(). The effect is as follows:

- If a reply queue is not specified for a queued message, then no further work is required beyond processing the message.
- If a message is dequeued that does specify a reply queue, then the originator of the message expects a reply to be enqueued upon successful completion of the execution of the request.
  - In the case where the application explicitly dequeues the message using `tpdequeue`(), it is the responsibility of the application to call `tpenqueue`() to enqueue the reply. Normally, this would be done in the same transaction in which the request message is dequeued and executed so the entire operation is handled atomically (that is, the reply is enqueued only if the transaction succeeds).
  - In the case where the message is processed (dequeued and passed to the application via a `tpcall`()) by `TMQFORWARD`, then `TMQFORWARD` will enqueue a reply if the application service returns successfully (that is, the service routine called `tpreturn`(3c) with `TPSUCCESS` and `tpcall`() did not return 1). If `tpcall`() receives data, then the typed buffer used is enqueued to the reply queue. If no data is received in `tpcall`(), then a message with no data (that is, a `NULL` message) is enqueued; the fact that a message is enqueued (even if `NULL`) is sufficient to signify that the operation has been completed.

Error Handling

Handling of errors requires both an understanding of the nature of the errors the application may encounter and careful planning and coordination between the BEA TUXEDO administrator and the application program developers. The way BEA TUXEDO System/Q works, if a message is dequeued within a transaction and the transaction is rolled back, then (if the retry parameter is greater than 0) the message ends up back on the queue where it can be dequeued and executed again.

For a transient problem, it may be desirable to delay for a short period before retrying to dequeue and execute the message, allowing the transient problem to clear. For example, if there is a lot of activity against the application database, there may be occasions when all you need is a little time to allow locks in a database to be released by another transaction. Normally, a limit on the number of retries is also useful to ensure that some application flaw doesn't cause significant waste of resources. When a queue is configured by the administrator, both a retry count and a delay period (in seconds) can be specified. A retry count of 0 implies that no retries are done. After the retry count is reached, the message is moved to an error queue that can be configured by the administrator for the queue space.

There are cases where the problem is not transient. For example, the queued message may request operations on an account that does not exist. In this case, it is desirable not to waste any resources by trying again. If the application programmer or administrator determines that failures for a particular operation are never transient, then it is simply a matter of setting the retry count to zero. It is more likely the case that for the same service some problems will be transient and some problems will be permanent; the administrator and application developers need to have more than a single approach to handle errors.

Other variations come about because the application may either dequeue messages directly or use the `TMQFORWARD` server and because an error may cause a transaction to be rolled back and the message requeued while logic dictates that the transaction should be committed. These variations and ways to deal with them are discussed in the chapters on BEA TUXEDO System/Q Programming and BEA TUXEDO System/Q Administration.

Summary

To summarize, BEA TUXEDO System/Q provides the following features to BEA TUXEDO application programmers and administrators:

- An application programming interface that lets you enqueue a request for subsequent processing. The system guarantees to execute the request successfully exactly once (by default, failure causes the message to be put back on the queue). An application programming interface is also provided to dequeue messages either from the top of a queue or by message identifier.
- The application program and/or the administrator can control the ordering of messages on the queue. Control is via the sort criteria, which are a `{LIFO | FIFO}`, time-based criteria, and priority-based criteria. The application can override the ordering to place the message at the queue top or ahead of a specific message that is already queued.
- A BEA TUXEDO server is provided to enqueue and dequeue messages on behalf of, possibly remote, clients and servers. The administrator decides how many copies of the server should be configured.
- A BEA TUXEDO server is provided to dequeue queued messages and forward them for execution. This server allows for existing servers to handle queued requests without modification. Each forwarding server can be configured to handle one or more queues. Transactions are used to guarantee exactly-once processing. The administrator controls how many forwarding servers are configured.
- The administrator can control messages stored on the queues for processing. This includes the number of times requests are retried on failure and how much time elapses between retries, reordering messages on queues, managing queue capacity and so on.

There are many application paradigms in which queued messages can be used. This feature can be used to queue requests when a machine, server, or resource is unavailable or unreliable (for example, in the case of a wide-area network). This feature can also be used for work flow provisioning where each step generates a queued request to do the next step in the process. Yet another use is for batch processing of potentially long running transactions, such that the initiator does not have to wait for completion but is assured that the message will eventually be processed.

CHAPTER

2 BEA TUXEDO System /Q Administration

Introduction

The BEA TUXEDO System/Q administrator has three primary areas of responsibility that are discussed in the three main sections of this chapter:

- Configuration of resources
- Creation of the queue space and queues
- Monitoring and maintenance of the facility

Close cooperation with the application developers and programmers is a must; the configuration and the queue attributes must reflect the requirements of the application.

Sample Program in Appendix A

A brief example of the use of the queued message facility is distributed with the software and is described in Appendix A, “A Sample Application.”

Configuration

Three servers are provided with the BEA TUXEDO System/Q. One is the TMS server, `TMS_QM`, that is the transaction manager server for the BEA TUXEDO System/Q resource manager. That is, it manages global transactions for the queued message facility. It must be defined in the `GROUPS` section of the configuration file.

The other two, `TMQUEUE`(5) and `TMQFORWARD`(5), provide services to users. They must be defined in the `SERVERS` section of the configuration file.

The application can also create its own queue servers, if the functionality of `TMQFORWARD` does not fully meet the needs of the application. For example, the administrator might want to have a special server to dequeue messages moved to the error queue.

Specifying the QM Server Group

There must be a server group defined for each queue space the application will use. In addition to the standard requirements of a group name tag and a value for `GRPNO` (see `ubbconfig`(5) for details). The `TMSNAME` and `OPENINFO` parameters need to be set. Here are examples:

```
TMSNAME=TMS_QM
```

and

```
OPENINFO="TUXEDO/QM:<device_name:<queue_space_name>"
```

`TMS_QM` is the name for the transaction manager server for TUXEDO System/Q. In the `OPENINFO` parameter, `TUXEDO/QM` is the literal name for the resource manager as it appears in `$TUXDIR/udataobj/RM`. The values for <*device_name*> and <*queue_space_name*> are instance-specific and must be set to the pathname for the universal device list and the name associated with the queue space, respectively. These values are specified by the BEA TUXEDO administrator using `qmadmin`(1).

Note: The chronological order of these specifications is not critical. The configuration file can be created either before or after the queue space is defined. The important thing is that the configuration must be defined and queue space and queues created before the facility can be used.

There can be only one queue space per GROUPS section entry. The CLOSEINFO parameter is not used.

The following example is taken from the manual page for TMQUEUE(5).

```
*GROUPS
TMQUEUEGRP1 GRPNO=1 TMSNAME=TMS_QM
        OPENINFO="TUXEDO/QM:/dev/device1:myqueuespace"
TMQUEUEGRP2 GRPNO=2 TMSNAME=TMS_QM
        OPENINFO="TUXEDO/QM:/dev/device2:myqueuespace"
```

Specifying the Message Queue Server

The TMQUEUE(5) manual page gives a full description of the SERVERS section of the configuration file, but there are some points worth additional emphasis here.

Transaction Timeout

TMQUEUE recognizes a -t *trantime* option when specified after the double dash (- -) in the CLOPT parameter. This timeout value affects only transactions begun within the server, which calls tpbegin(3c) only if it finds that a transaction is not already in effect, in other words, either the client called tpenqueue(3c) or tpdequeue(3c) without first calling tpbegin(3c) or it began a transaction and called tpenqueue(3c) or tpdequeue(3c) with the TPNOTRAN flag set to exclude the queue request from the client's transaction. The default for *trantime* is 30 seconds. If a tpdequeue request is received with the *flags* set to TPQWAIT, a TPETIME error will be returned if the wait exceeds -t *number* seconds.

Note: ctl is a structure of type TPQCTL used by tpenqueue(3c) and tpdequeue(3c) to pass parameters between the calling process and the system. TPQWAIT is a flag setting available in tpdequeue to indicate that the process wishes to wait for a reply message. The structure is explained in detail in the chapters on programming.

Queue Space Names, Queue Names, and Service Names

There is potential confusion among queue space names, queue names, and service names. The first place you are apt to encounter the confusion is in the specification of the message queue server: `TMQUEUE`. When specifying this server in the configuration file you can use the -s flag of the `CLOPT` parameter to name the queue space served by a given instance of the server, which is the same as saying it is a service advertised by the function: `TMQUEUE`. In an application that uses only one queue space, it is not necessary to specify the `CLOPT -s` option; it will default to -s `TMQUEUE:TMQUEUE`. If the application requires more than a single queue space, the names of the queue spaces are included as arguments to the -s option in the `SERVERS` section entry for the queued message server.

An alternative way of making this specification is to rebuild the message queue server, using `buildserver`(1), and name the queue spaces with the similar sounding `-s` option. This has the result of fixing, or *hardcoding*, the service names in the server executable.

```
# The following two specifications are equivalent:

*SERVERS
TMQUEUE SRVGRP="TMQUEUEGRP1" SRVID=1000 RESTART=Y GRACE=0 \
        CLOPT="-s myqueuespace:TMQUEUE"
and

buildserver -o TMQUEUE -s myqueuespace:TMQUEUE -r TUXEDO/QM \
        -f ${TUXDIR}/lib/TMQUEUE.o
followed by
 ..
 ..
 ..
TMQUEUE SRVGRP="TMQUEUEGRP1" SRVID=1000 RESTART=Y GRACE=0 \
        CLOPT="-A"
```

Data-Dependent Routing

The section above described the specification of services (that is, queue space names) in the message queue server. This capability can be used to bring about data-dependent routing of queued messages such that the message is queued for processing by a service within a specific group depending on a value in a field of the message buffer. To do this the same queue space name is specified in two different groups and a routing specification is made part of the configuration file to govern the group where the message is queued. Here is an example taken from the TMQUEUE(5) manual page (the queue space name has been changed):

```
*GROUPS
TMQUEUEGRP1 GRPNO=1 TMSNAME=TMS_QM
        OPENINFO="TUXEDO/QM:/dev/device1:myqueuespace"
TMQUEUEGRP2 GRPNO=2 TMSNAME=TMS_QM
        OPENINFO="TUXEDO/QM:/dev/device2:myqueuespace"
*SERVERS
TMQUEUE SRVGRP="TMQUEUEGRP1" SRVID=1000 RESTART=Y GRACE=0 \
        CLOPT="-s ACCOUNTING:TMQUEUE"
TMQUEUE SRVGRP="TMQUEUEGRP2" SRVID=1000 RESTART=Y GRACE=0 \
        CLOPT="-s ACCOUNTING:TMQUEUE"
*SERVICES
ACCOUNTING  ROUTING="MYROUTING"
*ROUTING
MYROUTING  FIELD=ACCOUNT BUFTYPE="FML" \
        RANGES="MIN-60000:TMQUEUEGRP1,60001-MAX:TMQUEUEGRP2"
```

Customized Buffer Types

TMQUEUE supports all of the standard BEA TUXEDO buffer types. If your application needs to add other types, it can be done by copying $TUXDIR/tuxedo/tuxlib/types/tmsypesw.c, adding an entry for your special buffer types, making sure to leave the final line null, and using the revised file as input to a buildserver(1) command. An example of the buildserver command is shown on the TMQUEUE(5) reference page.

You can also use the -s option of the buildserver command to associate additional service names with TMQUEUE as an alternative to specifying them in the server CLOPT parameter (see above).

Specifying the Message Forwarding Server

The third system-supplied server included with the BEA TUXEDO System/Q is `TMQFORWARD`(5). This is the server that takes messages from specified queues, passes them along to BEA TUXEDO servers via `tpcall`(3c), and handles associated reply messages. The full description of how the server is defined in the configuration file can be found on the manual page, but the sections that follow bring out some points that are worth additional emphasis.

`TMQFORWARD` is referred to as a server and each instance used by an application must be defined in the `SERVERS` section of the configuration file, but it has characteristics that set it apart from ordinary servers. For example:

- It is wrong to specify services for `TMQFORWARD`.
- A client process cannot post a message for `TMQFORWARD` as you would expect in a normal request/response relationship.
- `TMQFORWARD` should not be defined as a member of an `MSSQ` set.
- `TMQFORWARD` should never have a reply queue.

An instance of `TMQFORWARD` is tied to a queue space through the server group with which it is associated, specifically through the third field in the `OPENINFO` statement for the group. In the sections that follow we will examine other key parameters, especially `CLOPT` parameters that come after the double dash.

Queue Names and Service Names: the -q option

A required parameter is `-q` *`queuename`*, *`queuename`*. . . This parameter specifies the queue(s) to be checked by this instance of the server. *`queuename`* is a NULL-terminated string of up to 15 characters; it is the same as the name of the application service that will process the message once it has been taken off the queue by `TMQFORWARD`. It is also the name that a programmer specifies as the second argument of `tpenqueue`(3c) or `tpdequeue`(3c) when preparing to call the message queue server, `TMQUEUE`.

Controlling Transaction Timeout: the -t option

`TMQFORWARD` does its work within a transaction that it begins and ends. The `-t` *`trantime`* option is available to specify the length of time in seconds before the transaction is timed out. The transaction is begun when `TMQFORWARD` finds a message on the queue it is checking; it is committed after a reply has been enqueued either to the reply queue or the failure queue, so the transaction encompasses calling the service that processes the message and receiving a reply. The default is 60 seconds.

Controlling Idle Time: the -i option

Once `TMQFORWARD` is booted it constantly checks the queue to which it is assigned. If it finds the queue empty, it pauses for `-i` *`idletime`* seconds before checking again. If a value is not specified, the default is 30 seconds; a value of 0 says to keep checking the queue constantly, which can be wasteful if the queue is frequently empty.

Controlling Server Exit: the -e option

If the `-e` option is specified, the server will shut itself down gracefully (sending a message to the userlog) when it finds the queue empty. This behavior may be used to your advantage in connection with the threshold command that you can specify for a queue. There is a more complete discussion of this in the section on `qmadmin`(1).

Delete Message after Service Failure: the -d option

When a service request fails after being called by `TMQFORWARD` the transaction is rolled back and the message is put back on the queue for a later retry (up to a limit of retries specified for the queue). The `-d` option adds the following refinement: if the failed service returns a non-NULL reply, the reply (and its associated *`tpurcode`*) are put on a failure queue (if one is associated with the message and the queue exists) and the original request is deleted. The rationale behind this option is that rather than blindly retrying, the originating client can be coded to examine the failure message and determine whether further attempts are reasonable. It provides a way of handling a failure that is due to some inherently reasonable condition (for example, a record is *not found* because the account does not exist).

Customized Buffer Types

Customized application buffer types can be added to the type switch and incorporated into `TMQFORWARD` with the `buildserver`(1) command. It should be noted, however, that when you customize `TMQFORWARD` it is an error to specify service names with a `-s` option.

Dynamic Configuration

We have described configuration parameters in terms of `UBBCONFIG` parameters. However, it should be noted that the specifications in the `GROUPS` and `SERVERS` sections can also be added to the `TUXCONFIG` file of a running application by using `tmconfig`(1). Of course, the group and the servers will have to be booted once they have been defined.

Creating Queue Space and Queues

This section covers three of the `qmadmin`(1) commands that are used to establish the resources of the BEA TUXEDO System/Q facility.

Working with qmadmin Commands

Several of the key commands of `qmadmin` have positional parameters; we refer to `qspacecreate`, `qcreate`, `qspacechange`, and `crdl`. The program prompts for values for parameters, so it probably makes life easier to just enter the command and let the program take over.

Creating an Entry in the Universal Device List: crdl

The universal device list (UDL) is a VTOC file under the control of the BEA TUXEDO system. It maps the physical storage space on a machine where the BEA TUXEDO system is run. An entry in the UDL points to the disk space where the queues and messages of a queue space are stored; the BEA TUXEDO system manages the input and output for that space. If you have an existing BEA TUXEDO application, you are probably already familiar with the UDL and how it is created. If the creation of the queued message facility is part of a new BEA TUXEDO installation, then be informed that the UDL is created by `tmloadcf`(1) when the configuration file is first loaded.

Before you create a queue space, you must create an entry for it in the UDL. Here is an example of the commands:

```
# First invoke the /Q administrative interface, qmadmin
# The QMCONFIG variable points to an existing device where the UDL
# either resides or will reside.
QMCONFIG=/dev/rawfs qmadmin
# Next create the device list entry
crdl /dev/rawfs 50 500
# The above command sets aside 500 physical pages beginning at block 50
# If the UDL has no previous entries, offset (block number) 0 must be used
```

If you are going to add an entry to an existing BEA TUXEDO UDL, the value for the `QMCONFIG` variable will be the same pathname specified in `TUXCONFIG`. Once you have invoked `qmadmin`, we recommend you run a `lidl` command to see where space is available before creating your new entry.

Creating a Queue Space: qspacecreate

A queue space makes use of IPC resources; when you define a queue space you are allocating a shared memory segment and a semaphore. As noted above, the easiest way to use the command is to let it prompt you. The sequence looks like this:

```
> qspacecreate
Queue space name: myqueuespace
IPC Key for queue space: 230458
Size of queue space in disk pages: 200
Number of queues in queue space: 3
Number of concurrent transactions in queue space: 3
Number of concurrent processes in queue space: 3
Number of messages in queue space: 12
Error queue name: errq
Initialize extents (y or n - default no):
Blocking factor (default 16): 16
```

The program insists that you provide values for all prompts except the final three. As you can see, there are defaults for the last two; while you will almost certainly want to name an error queue, you are not required to. If you provide a name here, you still must create the error queue with the `qcreate` command. If you choose not to name an error queue, bear in mind that messages that normally would be moved to the error queue (for example, when a retry limit is reached), are dropped.

The value for the IPC key should be picked so as not to conflict with your other requirements for IPC resources. It should be a value greater than 32,768 and less than 262,143.

The size of the queue space, the number of queues, and the number of messages that can be queued at one time all depend on the needs of your application. Of course, you cannot specify a size greater than the number of pages specified in your UDL entry. In connection with these parameters, you also need to look ahead to the queue capacity parameters for an individual queue within the queue space. Those parameters allow you to (a) set a limit on the number of messages that can be put on a queue, and (b) name a command to be executed when the number of enqueued messages on the queue reaches the threshold. If you specify a low number of concurrent messages for the queue space, you may create a situation where your threshold on a queue will never be reached.

For the number of concurrent transactions count one for each `TMS_QM` server in the group that uses this queue space, one for each `TMQUEUE` or `TMQFORWARD` server in the group that uses this queue space and one for `qmadmin`. If your client programs begin transactions before they call `tpenqueue`, increase the count by the number of clients that might access the queue space concurrently; worst case is all of them.

For the number of concurrent processes count one for each `TMS_QM, TMQUEUE` or `TMQFORWARD` server in the group that uses this queue space and one for a fudge factor.

You can choose to initialize the queue space as you use the `qspacecreate` command, or you can let it be done by the `qopen` command when you first open the queue space.

Creating a Queue: qcreate

Each queue that you intend to use must be created with the `qmadmin qcreate` command. You first have to open the queue space with the `qopen` command. If you do not provide a queue space name, `qopen` will prompt for it.

The prompt sequence for `qcreate` looks like this:

```
> qcreate
Queue name: service1
Queue order (fifo, lifo, priority, time): fifo
Out-of-ordering enqueuing (none, top, msgid): none
Retries: 2
Retry delay in seconds: 30
High limit for queue capacity warning (b for bytes used,
B for blocks used, % for percent used, m for messages): 80%
Reset (low) limit for queue capacity warning: 0%
Queue capacity command:
No default queue capacity command
Queue 'service1' created
```

You can skip all of these prompts (except the prompt for the queue name); if you do not provide a name for the queue, the program displays a warning message and prompts again. For the other parameters the program provides a default and displays a message that specifies the default.

Specifying Queue Order

Messages are enqueued in the order specified by this parameter and dequeued from the top of the queue. The queue order parameter defines how the application wants queue order to be determined. If `priority` and/or `time` are chosen, messages are inserted into the queue according to values in the `TPQCTL` structure or, in the case of `priority`, to the value set by the /Q administrator. If specified, `fifo` or `lifo` (which are mutually exclusive), must be the last parameter selected. The sequence in which parameters are selected determines the sort criteria for the queue. In other words, a specification of `priority, fifo` would say that the queue should be arranged by message priority and that within messages of equal priority they should be dequeued on a first in, first out basis.

Enabling Out-of-order Enqueuing

If the administrator enables out-of-order enqueues; that is, if `top` and/or `msgid` are selected at the prompt, programmers can specify (via values in the `TPQCTL` structure of a `tpenqueue` call) that a message is to be put at the top of the queue or ahead of the message identified by `msgid`. Give this option some thought; once the choice is made you have to destroy and recreate the queue to change it.

Specifying Retry Parameters

Normal behavior for a queued message facility is to put a message back on the queue if the transaction that dequeues it is rolled back. It will be dequeued again when it reaches the top of the queue. You can specify the number of retries that should be attempted and also a time delay between retries. Note that when a dequeued message is put back on the queue for retry, queue order specifications are, in effect, suspended for *`Retry delay`* seconds.

The default for the number of retries is 0, which means that no retries are attempted. When the retry limit is reached (zero or whatever), the system moves the message to the error queue for this queue space, assuming an error queue has been named and created. If the error queue does not exist the message is discarded.

The delay time is expressed in seconds. When message queues are lightly populated so that a message restored to the queue reaches the top almost immediately, you can save cycles by building in a delay factor. Your general policy on retries should be based on the experience of your particular application. If you have a fair amount of contention for the service associated with a given queue, you may get a lot of transient problems. One way to deal with them is to specify a large number of retries. (The number is strictly subjective, as is the time between retries.) If the nature of your application is such that any rolled back transaction signals a failure that is never going to go away, you might want to specify 0 retries and move the message immediately to the error queue. (This is very much like what happens when you specify the `-d` option for `TMQFORWARD`; the only difference is that a non-zero length failure message must be received for `TMQFORWARD` automatically to drop the message from the queue.)

Using Queue Capacity Limits

There are three parameters of the `qcreate` command that can be used to partially automate the management of a queue. The parameters set a high and low threshold figure (it can be expressed as blocks, messages or per cent of queue capacity) and allow you to name a command that is executed when the high threshold is reached. (Actually, the command is executed once when the high threshold is reached, but not again unless the low threshold is reached first.)

Here are two examples of ways the parameters can be used:

```
High limit for queue capacity warning (b for bytes used,
B for blocks used, % for percent used, m for messages): 80%
Reset (low) limit for queue capacity warning: 10%
Queue capacity command: /usr/app/bin/mailme myqueuespace service1
```

This sequence sets the upper threshold at 80% of queue capacity and specifies a command to be executed when the queue is 80% full. The command is a script you have created that sends you a mail message when the threshold is reached (`myqueuespace` and `service1` are hypothetical arguments to your command). Presumably, once you have been informed that the queue is filling up you can take action to ease the situation. You will not get the warning message again unless the queue load drops to 10% of capacity or below, and then rises again to 80%.

The second example is somewhat more automated and takes advantage of the `-e` option of the `TMQFORWARD` server.

```
High limit for queue capacity warning (b for bytes used,
B for blocks used, % for percent used, m for messages): 90%
Reset (low) limit for queue capacity warning: 0%
Queue capacity command: tmboot -i 1002
```

This sequence assumes that you have configured a reserve `TMQFORWARD` server for the queue in question with a `SRVID=1002` number and have included the `-e` option in its `CLOPT` parameter. (It also assumes that the server is not booted or, if booted, has shut itself down as a result of finding the queue empty.) When the queue reaches 90% capacity the tmboot command is executed to boot the reserve server. The `-e` option causes the server to shut itself down when the queue is empty. You have set the low threshold to 0% so as not to kick off unnecessary `tmboot` commands for a server that is already booted.

The default values for the three options are 100%, 0%, and no command.

Reply and Failure Queues

The discussion above about creating a queue and providing parameters for its operation was written from the viewpoint of creating a queue for messages waiting to be processed by a service of the same name, although the parameters for creating a queue are the same regardless of its use. Other queues are possible and indeed highly useful. Included in the `TPQCTL` structure when a message is enqueued to a service queue are fields that can name a reply queue and a failure queue. `TMQFORWARD` detects the success or failure of the `tpcall`(3c) it makes to the requested service and, if these queues have been created by the administrator, queues the reply accordingly. If no reply or failure queue exists, the success or failure response message from the service is dropped leaving the originating client with no information about the outcome of the queued request. Even if there is no reply message from the service, if a reply queue exists, a zero-length message is enqueued there by `TMQFORWARD` to inform the originating client.

When creating a reply or a failure queue, bear in mind that in most cases messages are dequeued from these queues by a client process looking for information about an earlier enqueued request. Since the most common way of dequeuing such messages is by the `msgid` (message identifier) or `corrid` (correlation identifier) associated with the message—as opposed to taking a message off the top of the queue—the queue ordering criteria are less significant; you might just as well settle for `fifo`. However, the `out-of-order` parameter must be configured to permit access by `msgid`. The `retries` and `retry delay` parameters have no significance for reply queues; just take the defaults. The `queue capacity` thresholds and commands are likely to be useful on reply queues, but we recommend using them to alert the administrator so that he or she can intervene.

Error Queues

An error queue is a system queue. If you remember, one of the prompts when you use `qspacecreate` asks for the name of the error queue for this queue space. When you have actually created an error queue of that name, the system uses it as a place to move messages from the service queue that have reached their retry limit. The management of the error queue is up to the administrator who can either deal with the messages manually through commands of `qmadmin` or can set up an automated way of handling them. The `queue capacity` parameters can be used, but all of the other `qcreate` parameters, with the exception of `qname`, do not apply.

Note: We recommend against using the same queue as both an error queue and a service failure queue; doing so would make it more difficult to manage cleanly and could lead to clients trying to access the administrator's area.

Maintenance of the BEA TUXEDO System/Q Feature

This section covers some things the queue administrator may have to do from time to time to keep a queue space operating efficiently.

Adding Extents to a Queue Space

If you find you need more disk storage for a queue space, you can add it with the `qaddext` command of `qmadmin`(1). The command takes the queue space name and a number of pages as arguments. The pages come from extents defined in the UDL for the device in your `QMCONFIG` variable. The queue space must be inactive; you can use the exclamation point to execute a command outside of `qmadmin` to shut down the associated server group. For example:

```
> !tmshutdown -g TMQUEUEGRP1
```

followed by

```
> qclose
> qaddext myqueue 100
```

The queue space must be closed; `qmadmin` will close it for you if you try to add extents to an open queue space.

Backing Up or Moving Queue Space

A convenient command to use to back up a queue space is the UNIX command `dd`. Shut down the associated server group first. The command lines would look like this:

```
tmshutdown -g TMQUEUEGRP1
dd if=<qspace_device_file> of=<output_device_filename>
```

For other options, see `dd`(1) in a UNIX system reference manual.

This same command can be used to migrate the queue space to a machine of the same architecture, although you may need to start the command sequence with a `qmadmin chdl` command to provide a new device name if the present name does not exist on the target machine.

Moving the Queue Space to a Different Type of Machine

If you need to move a queue space to a machine with a different architecture (primarily byte order), the procedure is more complex. Create and run an application program to dequeue all messages from all queues in the queue space and write them out in machine-independent format. Then enqueue the messages in the new queue space.

TMQFORWARD and Non-Global Transactions

Messages dequeued and forwarded using `TMQFORWARD` are executed within a global transaction because the operation crosses group boundaries. If the messages are executed by servers that are not associated with an RM or that do not run within a global transaction, they should have a server group with `TMSNAME=TMS` (for the NULL XA interface).

TMQFORWARD and Commit Control

The global transaction begun by `TMQFORWARD` when it dequeues a message for execution is terminated by a `tpcommit`(). The administrator can set the `CMTRET` parameter in the configuration file to control whether the transaction commits when it is logged or when it is complete. (See the discussion of `CMTRET` in the `RESOURCES` section of the `ubbconfig`(5) reference page.)

Handling Transaction Timeout

Handling transaction timeout requires cooperation between the queue administrator and the programmer developing client programs that dequeue messages. When `tpdequeue`(3c) is called with the *`flags`* argument set to `TPQWAIT`, the `TMQUEUE` server may be blocked waiting for a message to come onto a queue. The number of seconds before it times out is up to:

- The `timeout` flag in the `tpdequeue` call (if the transaction is started in the client)
- The `-t` *`number`* flag of the `TMQUEUE` server (if the client has not started the transaction)

To get around blocking operations using the `TMQUEUE` server it might help to configure two `TMQUEUE` servers (or `MSSQ` sets of multiple `TMQUEUE` servers) that offer different service names for the same queue space. `tpenqueue` and non-waiting `tpdequeue` operations can go to one set of servers; waiting `tpdequeue` operations, to a second set.

TMQFORWARD and Retries for an Unavailable Service

When a `TMQFORWARD` server attempts to forward messages to a service that is not available the situation can develop where the retry limit for the queue may be reached. The message is then moved to the error queue (if one exists). To avoid this situation the administrator should either shut the `TMQFORWARD` server down or set the retry count higher.

When a message is moved to the error queue it is no longer associated with the original queue. If errors are going to be dealt with by the administrator moving the message back to the service queue when the service is known to be available, then the queue name should be stored as part of the `corrid` in the `TPQCTL` structure so the queue name is associated with the message.

3 BEA TUXEDO System/Q C Language Programming

This chapter deals with the use of the ATMI C language functions for enqueuing and dequeuing messages: `tpenqueue`(3c) and `tpdequeue`(3c), plus some ancillary functions.

Prerequisite Knowledge

The BEA TUXEDO programmer coding client or server programs for the queued message facility should be familiar with the C language binding to the BEA TUXEDO ATMI. General guidance on BEA TUXEDO programming is available in the *BEA TUXEDO Programmer's Guide*. Detailed pages on all the ATMI functions are in Section 3c of the *BEA TUXEDO Reference Manual*.

Where Requests Can Originate

The calls used to place a message on a BEA TUXEDO System/Q queue can originate in any client or server process associated with the application. The list includes:

- Clients or servers on the same machine as the queue space or on another machine on the network.
- Conversational programs, although you cannot have a conversational connection with a queue (or with the `TMQUEUE`(5) server).
- Workstation clients via a surrogate process on the server side; the administrative interface is also entirely on the server side.

Emphasis on the Default Case

The coverage of BEA TUXEDO System/Q programming in this chapter reflects the illustration in Chapter 1, or at least the left-hand portion of it. In that figure a client (or a process acting in the role of a client) queues a message by calling `tpenqueue`(3c) and specifying a queue space available through the `TMQUEUE` server. The client later retrieves a reply via a `tpdequeue` call to `TMQUEUE`.

The illustration in Chapter 1 goes on to show the queued message being dequeued by the server `TMQFORWARD` and sent to an application server for processing (via `tpcall`). When a reply to the `tpcall` is received, `TMQFORWARD` enqueues the reply message. Since a major goal of `TMQFORWARD` is to provide an interface between the queue space and existing application services, it does not require further application coding. For that reason, this chapter concentrates on the client-to-`qspace` side.

A brief example of the use of the queued message facility is distributed with the software and is described in Appendix A, “A Sample Application.”

Enqueuing Messages

The syntax for `tpenqueue` is as follows.

```
#include <atmi.h>
int tpenqueue(char *qspace, char *qname, TPQCTL *ctl,
     char *data, long len, long flags)
```

When a `tpenqueue` call is issued it tells the system to store a message on the queue identified in *`qname`* in the space identified in *`qspace`*. The message is in the buffer pointed to by *`data`* and has a length of *`len`*. By the use of bit settings in *`flags`* the system is informed how the call to `tpenqueue` is to be handled. Further information about the handling of the enqueued message and replies is provided in the `TMQCTL` structure pointed to by *`ctl`*.

Command Line Arguments, tpenqueue(3)

There are some important arguments to control the operation of `tpenqueue`(3c). Let's look at some of them.

tpenqueue(): the qspace Argument

`qspace` identifies a queue space previously created by the administrator. When a server is defined in the `SERVERS` section of the configuration file, the service names it offers are aliases for the actual queue space name (which is specified as part of the `OPENINFO` parameter in the `GROUPS` section). For example, when your application uses the server `TMQUEUE`, the value pointed at by the *`qspace`* argument is the name of a service advertised by `TMQUEUE`. If no service aliases are defined, the default service is the same as the server name, `TMQUEUE`. In this case the configuration file can include:

```
TMQUEUE
        SRVGRP = QUE1  SRVID = 1
        GRACE = 0  RESTART = Y CONV = N
        CLOPT = "-A"

or
        CLOPT = "-s TMQUEUE"
```

The entry for server group `QUE1` has an `OPENINFO` parameter that specifies the resource manager, the pathname of the device and the queue space name. The `qspace` argument in a client program can then look like this:

```
if (tpenqueue("TMQUEUE", "STRING", (TPQCTL *)&qctl,
        (char *)reqstr, 0,0) == -1) {
        Error checking
}
```

The example shown on the manual page for `TMQUEUE`(5) shows how an alias for service names can be included when the server is built and specified in the configuration file. The sample program in Appendix A, “A Sample Application,” also specifies an alias service name.

tpenqueue(): the qname Argument

Within a queue space, message queues are named according to application services that process the requests. *`qname`* is a pointer to such a value; an exception in which *`qname`* is not an application service is described later in the chapter.

tpenqueue(): the data and len Arguments

`data` points to a buffer that contains the message to be processed. The buffer must be one that was allocated with a call to `tpalloc`(3c). *`len`* gives the length of the message. Some BEA TUXEDO buffer types (such as FML) do not require *`len`* to be specified; in such cases, the argument is ignored. *`data`* can be `NULL`; when it is, *`len`* is ignored and the message is enqueued with no data portion.

tpenqueue(): the flags Arguments

`flags` values are used to tell the BEA TUXEDO system how the `tpenqueue` call is handled; the following are valid flags:

`TPNOTRAN`
: If the caller is in transaction mode, this flag specifies that the enqueuing of the message is to be done in a separate transaction.

`TPNOBLOCK`
: If this flag is set and a blocking condition exists, the call fails immediately with `tperrno` set to `TPEBLOCK`. When the flag is not set the call blocks until the condition subsides; it fails if a blocking or transaction timeout occurs (`TPETIME`).

TPNOTIME
: This flag asks that the call be immune to blocking timeouts; transaction timeouts may still occur.

TPSIGRSTRT
: This flag says that any underlying system calls that are interrupted by a signal should be reissued. When not specified and a signal is received, the call fails and sets `tperrno` to `TPGOTSIG`.

The TPQCTL Structure

The third argument to `tpenqueue()` is a pointer to a structure of type `TPQCTL`. The `TPQCTL` structure has members that are used by the application and by the BEA TUXEDO system to pass parameters in both directions between application programs and the queued message facility. The client that calls `tpenqueue` sets flags to mark members the application wants the system to fill in. The structure is also used by `tpdequeue`; some of the members do not come into play until the application calls that function. The complete structure is shown in Listing 3-1.

Listing 3-1 The tpqctl_t Structure

```
#define TMQNAMELEN      15
#define TMMSGIDLEN      32
#define TMCORRIDLEN     32
struct tpqctl_t {                /* control parameters to queue primitives */
long flags;                      /* indicates which of the values are set */
long deq_time;                   /* absolute/relative time for dequeuing */
long priority;                   /* enqueue priority */
long diagnostic;                 /* indicates reason for failure */
char msgid[TMMSGIDLEN];          /* id of message before which to queue */
char corrid[TMCORRIDLEN];        /* correlation id used to identify message */
char replyqueue[TMQNAMELEN+1];   /* queue name for reply message */
char failurequeue[TMQNAMELEN+1]; /* queue name for failure message */
CLIENTID cltid;                  /* client identifier for originating client */
long urcode;                     /* application user-return code */
long appkey;                     /* application authentication client key */
};
typedef struct tpqctl_t TPQCTL;
```

The following is a list of valid bits for the *flags* parameter controlling input information for tpenqueue.

TPNOFLAGS
: No flags or values are set. No information is taken from the structure. Leaving fields of the structure not set is equivalent to a setting of TPNOFLAGS.

TPQTOP
: Setting this flag bit indicates that the queue ordering be overridden and the message placed at the top of the queue. This request may not be granted depending on whether or not the queue was configured to allow overriding the queue ordering to put a message at the top of the queue.

TPQBEFOREMSGID
: Setting this flag bit indicates that the queue ordering be overridden and the message placed in the queue before the message identified by the msgid field. This request may not be granted depending on whether or not the queue was configured to allow overriding the queue ordering to put a message ahead of another by msgid. TPQTOP and TPQBEFOREMSGID are mutually exclusive flags. Assumes a prior (successful) call with TPQMSGID set.

TPQTIME_ABS
: If set, the request is to be processed after the time specified by the *deq_time* field. The *deq_time* is an absolute time value as generated by time(2) or mktime(3C), if they are available in your UNIX operating system, or gp_mktime(3c), provided with the BEA TUXEDO system. The value set in the *deq_time* field is the number of seconds since 00:00:00 UTC, January 1,1970. TPQTIME-ABS can be overridden and the message dequeued immediately by MSGID or CORRID.

TPQTIME_REL
: If set, the request is to be processed relative to the completion of the queuing transaction. *deq_time* specifies the number of seconds to delay after the transaction completes before the submitted request should be processed. TPQTIME_REL can be overridden and the message dequeued immediately by MSGID or CORRID. TPQTIME_ABS and TPQTIME_REL are mutually exclusive flags.

TPQPRIORITY
: If set, the priority at which the request should be enqueued is stored in *priority*. The priority must be in the range 1 to 100, inclusive.

`TPQCORRID`

If set, the correlation identifier value specified in *`corrid`* is available when a request is dequeued with `tpdequeue`(3c). This identifier accompanies any reply or failure message that is queued so an application can correlate a reply with a particular request. The entire value should be initialized such that the value can be matched at a later time. This can be done, for example, by padding with null characters to the full 32-character size.

`TPQREPLYQ`

If set, a reply queue named in *`replyqueue`* is associated with the queued message. Any reply to the message will be queued to the named queue within the same queue space as the request message. This string must be NULL-terminated (maximum 15 characters in length). If a reply is generated for the service and a reply queue is not specified or the reply queue does not exist, the reply is dropped.

`TPQFAILUREQ`

If set, a failure queue named in the `failurequeue` field is associated with the queued message. If a failure occurs when executing the enqueued message, a failure message will go to the named queue within the same queue space as the original request message. This string must be NULL-terminated (maximum 15 characters in length).

Additionally, the *`urcode`* element of `TPQCTL` can be set with a user-return code. This value will be returned to the application that calls `tpdequeue`(3c) to dequeue the message.

On output from `tpenqueue`, the following elements may be set in the `TPQCTL` structure:

```
long flags;             /* indicates which of the values are set */
char msgid[32];         /* id of enqueued message */
long diagnostic;        /* indicates reason for failure */
```

An additional setting of the *`flags`* parameter requests output information from `tpenqueue`. If this flag bit is turned on when `tpenqueue` is called, then the associated element in the structure is populated if available and the bit remains set. If the value is not available, `tpenqueue` completes with the flag bit turned off.

`TPQMSGID`

If set and the call to `tpenqueue` was successful, the message identifier will be stored in *`msgid`*.

If the call to `tpenqueue` fails and `tperrno` is set to `TPEDIAGNOSTIC`, a value indicating the reason for failure is returned in *`diagnostic`*. The possible values are:

`[QMEINVAL]`
: An invalid flag value was specified.

`[QMEBADRMID]`
: An invalid resource manager identifier was specified.

`[QMENOTOPEN]`
: The resource manager is not currently open.

`[QMETRAN]`
: The call was made with the `TPNOTRAN` flag and an error occurred trying to start a transaction in which to enqueue the message.

`[QMEBADMSGID]`
: An invalid message identifier was specified.

`[QMESYSTEM]`
: A system error has occurred. The exact nature of the error is written to a log file.

`[QMEOS]`
: An operating system error has occurred.

`[QMENOTA]`
: The transaction in which the message was enqueued was aborted.

`[QMEPROTO]`
: An enqueue was done when the transaction state was not active.

`[QMEBADQUEUE]`
: An invalid or deleted queue name was specified.

`[QMENOSPACE]`
: There is no space on the queue for the message.

The remaining members of the control structure are not used on input to `tpenqueue`.

Overriding the Queue Order

If the administrator in creating a queue allows `tpenqueue` calls to override the order of messages on the queue, you have two mutually exclusive ways to use that capability. You can specify that the message is to be placed at the top of the queue by setting *`flags`* to `TPQTOP` or you can specify that it be placed ahead of a specific message by setting *`flags`* to `TPQBEFOREMSGID` and setting *`msgid`* to the ID of the message you wish to precede. This assumes that you saved the message-ID from a previous call in order to be able to use it here. Your administrator must tell you what the queue supports; it can be created to allow either or both of these overrides, or to allow neither.

Overriding the Queue Priority

If the queue was created with `priority` as a queue ordering parameter, you can set a value in *`priority`* to specify the dequeuing priority for the message. The value must be in the range 1 to 100; the higher the number the higher the priority. If `priority` was not one of the queue ordering parameters, setting a priority here has no effect.

Setting a Dequeuing Time

A queue can be created with `time` as a queue ordering parameter. When this is the case, you can specify in *`deq_time`* either an absolute time for the message to be dequeued or a time relative to the enqueuing transaction. You set *`flags`* to either `TPQTIME_ABS` or `TPQTIME_REL` to say how the value should be treated.

BEA TUXEDO System/Q provides a function, `gp_mktime`(3c), that is used to convert a date and time provided in a `tm` structure to the number of seconds since January 1, 1970. The value is returned in `time_t`, a `typedef'd` long. To set an absolute time for the message to be dequeued (we are using 12:00 noon, December 9, 1992), do the following.

1. Place the values for the date you want to use in the `tm` structure.

```
#include <stdio.h>
#include <time.h>
static char *const wday[] = {
         "Sunday", "Monday", "Tuesday", "Wednesday",
         "Thursday", "Friday", "Saturday", "-unknown-"
};
struct tm time_str;
/*...*/
time_str.tm_year = 1992 - 1900;
time_str.tm_mon = 12 - 1;
time_str.tm_mday = 9;
time_str.tm_hour = 12;
time_str.tm_min = 0;
time_str.tm_sec = 1;
time_str.tm_isdst = -1;
```

2. Call `gp_mktime` to produce a value for `deq_time` and set the *`flags`* to indicate an absolute time is being provided.

```
#include <atmi.h>
TPQCTL qctl;
if ((qctl->deq_time = (long)gp_mktime(&time_str)) == -1) {
        /* check for errors */
}
qctl->flags = TPQTIME_ABS
```

3. Call `tpenqueue`.

```
if (tpenqueue(qspace, qname, qctl, *data,*len,*flags) == -1) {
       /* check for errors */
}
```

If you want to specify a relative time for dequeuing, for example, *nnn* seconds after the completion of the enqueuing transaction, place the number of seconds in `deq_time` and set `flags` to `TPQTIME_REL`.

tpenqueue() and Transactions

Messages are always enqueued within a transaction; the only question is, within whose transaction? There are two choices. If caller of `tpenqueue` is in transaction mode and `TPNOTRAN` is not set, then the enqueuing is done within the caller's transaction. The caller knows for certain from the success or failure of `tpenqueue` whether the message was enqueued or not. If the call succeeds, the message is guaranteed to be on the queue. If the call fails, the transaction is rolled back, including the part where the message was placed on the queue.

If caller of `tpenqueue` is not in transaction mode or if `TPNOTRAN` is set, the message is enqueued in a separate transaction. If the call to `tpenqueue` returns success, the message is guaranteed to be on the queue. If the call to `tpenqueue` fails with a communication error or with a transaction or blocking timeout, the caller is left in doubt about whether the failure occurred before or after the message was enqueued.

Note that specifying `TPNOTRAN` while the caller is not in transaction mode has no meaning.

Dequeuing Replies

The syntax for `tpdequeue` is as follows:

```
#include <atmi.h>
int tpdequeue(char *qspace, char *qname, TPQCTL *ctl, \
        char **data, long *len, long flags)
```

When this call is issued it tells the system to dequeue a message from the *qname* queue in space named *qspace*. The message is placed in a buffer (originally allocated by `tpalloc`(3c)) at the address pointed to by **data*. *len* points to the length of the data. If *len* is 0 on return from `tpdequeue`, the message had no data portion. By the use of bit settings in *flags* the system is informed how the call to `tpdequeue` is to be handled. The `TPQCTL` structure pointed to by *ctl* carries further information about how the call should be handled.

Command Line Arguments, tpdequeue

There are some important arguments to control the operation of `tpdequeue`(3c). Let's look at some of them.

tpdequeue(): the qspace Argument

qspace identifies a queue space previously created by the administrator. When a server is defined in the `SERVERS` section of the configuration file, the service names it offers are aliases for the actual queue space name (which is specified as part of the `OPENINFO` parameter in the `GROUPS` section). For example, when your application uses the server `TMQUEUE`, the value pointed at by the *qspace* argument is the name of a service advertised by `TMQUEUE`. If no service aliases are defined, the default service is the same as the server name, `TMQUEUE`. In this case the configuration file can include:

```
TMQUEUE
        SRVGRP = QUE1  SRVID = 1
        GRACE = 0  RESTART = Y CONV = N
        CLOPT = "-A"
or
        CLOPT = "-s TMQUEUE"
```

The entry for server group `QUE1` has an `OPENINFO` parameter that specifies the resource manager, the pathname of the device and the queue space name. The *qspace* argument in a client program can then look like this:

```
if (tpdequeue("TMQUEUE", "REPLYQ", (TPQCTL *)&qctl,
        (char **)&reqstr, &len,TPNOTIME) == -1) {
        Error checking
}
```

The example shown on the manual page for `TMQUEUE`(5) shows how alias service names can be included when the server is built and specified in the configuration file. The example in Appendix A, "A Sample Application," also specifies an alias service name.

tpdequeue(): the qname Argument

Reply `queue` names in a queue space need to be agreed upon within the application. The administrator creates a reply queue (and often an error queue) in the same manner a message queue is created. *qname* is a pointer to the name.

tpdequeue(): the data and len Arguments

The arguments have a different flavor than they do on `tpenqueue`. **data* points to the address of a buffer where the system is to place the message being dequeued. When `tpdequeue` is called, it is an error for its value to be `NULL`.

When `tpdequeue` returns, *len* points to a long that carries information about the length of the data retrieved. If it is 0, it means that the reply had no data portion. This can be a legitimate and successful reply in some applications; receiving even a 0 length reply can be used to show successful processing of the enqueued request. If you wish to know whether the buffer has changed from before the call to `tpdequeue`, save the prior length and compare it to *len*.

tpdequeue(): the flags Arguments

flags values are used to tell the BEA TUXEDO system how the `tpdequeue` call is handled; the following are valid flags:

`TPNOTRAN`
: If the caller is in transaction mode, this flag specifies that the message is to be dequeued in a separate transaction.

`TPNOBLOCK`
: If this flag is set and a blocking condition exists, the call fails immediately with `tperrno` set to `TPEBLOCK`. When the flag is not set the call blocks until the condition subsides; it fails if a blocking or transaction timeout occurs (`TPETIME`). This blocking condition does not include blocking on the queue itself if the `TPQWAIT` option in *flags* is specified.

`TPNOTIME`
: This flag asks that the call be immune to blocking timeouts; transaction timeouts may still occur.

`TPNOCHANGE`
: When this flag is set, the type of the buffer pointed to by **data* is not allowed to change. By default, if a buffer is received that differs in type from the buffer pointed to by **data*, then **data*'s buffer type changes to the received buffer's type so long as the receiver recognizes the incoming buffer type. That is, the type and subtype of the received buffer must match the type and subtype of the buffer pointed to by **data*.

`TPSIGRSTRT`
: This flag says that any underlying system calls that are interrupted by a signal should be reissued. When not specified and a signal is received, the call fails and sets `tperrno` to `TPGOTSIG`.

The third argument to `tpdequeue`() is a pointer to a structure of type `TPQCTL`. The `TPQCTL` structure has members that are used by the application and by the BEA TUXEDO system to pass parameters in both directions between application programs and the queued message facility. The client that calls `tpdequeue` sets flags to mark members the application wants the system to fill in. As described earlier, the structure is also used by `tpenqueue`; some of the members only apply to that function. The entire structure is shown in Listing 3-1.

On input to `tpdequeue`, the following elements may be set in the `TPQCTL` structure:

```
long flags;       /* indicates which of the values are set */
char msgid[32];   /* id of message to dequeue */
char corrid[32];  /* correlation identifier of message to dequeue */
```

The valid flags on input to `tpdequeue` are:

`TPNOFLAGS`
: No flags are set. No information is taken from the control structure.

`TPQGETBYMSGID`
: If set, it requests that the message identified by *`msgid`* be dequeued. The message identifier would be one that was returned by a prior call to `tpenqueue`. This option cannot be used with the `TPQWAIT` option.

`TPQGETBYCORRID`
: If set, it requests that the message with the correlation identifier specified by *`corrid`* be dequeued. The correlation identifier would be one that the application specified when enqueuing the message with `tpenqueue`. This option cannot be used with the `TPQWAIT` option.

`TPQWAIT`
: If set, it indicates that an error should not be returned if the queue is empty. Instead, the process should block until a message is available.

Following is a list of valid bits for the *flags* parameter controlling output information from `tpdequeue`. If the flag bit is turned on when `tpdequeue` is called, then the associated element (see Listing 3-1) in the structure is populated if available and the bit remains set. If the value is not available, the flag bit is turned off after `tpdequeue` completes.

`TPQPRIORITY`
: If set and the value is available, the priority at which the message was queued is stored in *priority*.

`TPQMSGID`
: If set and the call to `tpdequeue` was successful, the message identifier will be stored in *msgid*.

`TPQCORRID`
: If set and the call to `tpdequeue` was successful and the message was queued with a correlation identifier, the value will be stored in *corrid*. Any reply to a queue must have this correlation identifier.

`TPQREPLYQ`
: If set and the message is associated with a reply queue, the value will be stored in *replyqueue*. Any reply to the message should go to the named reply queue within the same queue space as the request message.

`TPQFAILUREQ`
: If set and the message is associated with a failure queue, the value will be stored in *failurequeue*. Any failure message should go to the named failure queue within the same queue space as the request message.

If the call to `tpdequeue` failed and `tperrno` is set to `TPEDIAGNOSTIC`, a value indicating the reason for failure is returned in *diagnostic*. The valid codes for *diagnostic* include those shown above for `tpenqueue` and the following additional codes:

`[QMENOMSG]`
: No message was available for dequeuing.

`[QMEINUSE]`
: When dequeuing a message by correlation or message identifier, the specified message is in use by another transaction. Otherwise, all messages currently on the queue are in use by other transactions.

Using TPQWAIT

When `tpdequeue` is called with *`flags`* set to `TPQWAIT`, the `TMQUEUE` server may be blocked waiting for a message to come onto the queue. The amount of time it is blocked can be controlled by the transaction timeout value set by the caller in `tpbegin`(3c) or by the `-t` option in the `CLOPT` parameter of the `TMQUEUE` server (if the transaction is started in the server). To avoid blocking `tpenqueue` calls that also use the `TMQUEUE` server, it may be desirable to configure two or more `TMQUEUE` servers (or `MSSQ` sets) offering different service names for the same queue space. It could be set up so that all enqueue and nonwaiting dequeue operations use one set of `TMQUEUE` servers and all waiting dequeue operations use the second set.

Error Handling

In considering how best to handle errors in dequeuing it is helpful to differentiate between errors encountered by `TMQFORWARD` as it attempts to dequeue a message to forward to the requested service and errors that occur in the service that processes the request. This subject was discussed in Chapter 1, but is repeated here in the context of writing application programs.

By default, if a message is dequeued within a transaction and the transaction is rolled back, then (if the `retry` parameter is greater than 0) the message ends up back on the queue and can be dequeued and executed again. It may be desirable to delay for a short period before retrying to dequeue and execute the message, allowing the transient problem to clear (for example, allowing for locks in a database to be released by another transaction). Normally, a limit on the number of retries is also useful to ensure that an application flaw doesn't cause significant waste of resources. When a queue is configured by the administrator, both a retry count and a delay period (in seconds) can be specified. A retry count of 0 implies that no retries are done. After the retry count is reached, the message is moved to an error queue that is configured by the administrator for the queue space. If the error queue is not configured, then messages that have reached the retry count are simply deleted. Messages on the error queue must be handled by the administrator who must work out a way of notifying the originator that meets the requirements of the application. This kind of handling is almost transparent to the originating program that put the message on the queue. There is a virtual guarantee that once a message is successfully enqueued it will be processed according to the parameters of `tpenqueue` and the attributes of the queue. Notification that a message has been moved to the error queue should be a rare occurrence in a system that has properly tuned its queue parameters.

A failure queue (normally, different from the queue space error queue) may be associated with each queued message. This queue is specified on the enqueuing call as the place to put any failure messages. The failure message for a particular request can be identified by an application-generated correlation identifier that is associated with the message when it is enqueued.

The default behavior of retrying until success (or a predefined limit) is quite appropriate when the failure is caused by a transient problem that is later resolved, allowing the message to be handled appropriately.

There are cases where the problem is not transient. For example, the queued message may request operating on an account that does not exist (and the application is such that it won't come into existence within a reasonable time period if at all). In this case, it is desirable not to waste any resources by trying again. If the application programmer or administrator determines that failures for a particular operation are never transient, then it is simply a matter of setting the retry count to zero, although this will require a mechanism to constantly clear the queue space error queue of these messages (for example, a background client that reads the queue periodically). More likely, it is the case that some problems will be transient (for example, database lock contention) and some problems will be permanent (for example, the account doesn't exist) for the same service.

In the case that the message is processed (dequeued and passed to the application via a `tpcall`) by `TMQFORWARD`, there is no mechanism in the information returned by `tpcall` to indicate whether a `TPESVCFAIL` error is caused by a transient or permanent problem.

As in the case where the application is handling the dequeuing, a simple solution is to return success for the service, that is, `tpreturn` with `TPSUCCESS`, even though the operation failed. This allows the transaction to be committed and the message removed from the queue. If reply messages are being used, the information in the buffer returned from the service can indicate that the operation failed and the message will be enqueued on the reply queue. The *rcode* argument of `tpreturn` can also be used to return application specific information.

In the case where the service fails and the transaction must be rolled back, it is not clear whether or not `TMQFORWARD` should execute a second transaction to remove the message from the queue without further processing. By default, `TMQFORWARD` will not delete a message for a service that fails. `TMQFORWARD's` transaction is rolled back and the message is restored to the queue. A command line option may be specified for `TMQFORWARD` that indicates that a message should be deleted from the queue if the service fails and a reply message is sent back with length greater than 0. The message is deleted in a second transaction. The queue must be configured with a delay time and retry count for this to work. If the message is associated with a failure queue, the reply data will be enqueued to the failure queue in the same transaction as the one in which the message is deleted from the queue.

A Procedure for Dequeuing Replies

If your application expects to receive replies to queued messages, here is a procedure you may want to follow.

1. As a preliminary step, the queue space must include a reply queue and a failure queue. The application must also agree on the content of the correlation identifier. The service should be coded to return `TPSUCCESS` on a logical failure and return an explanatory code in the *`rcode`* argument of `tpreturn`.

2. When you call `tpenqueue` to put the message on the queue, set `flags` to turn on the bits for the following flags.

   ```
   TPQCORRID     TPQREPLYQ
   TPQFAILUREQ   TPQMSGID
   ```

   Fill in the values for `corrid`, `replyqueue` and `failurequeue` before issuing the call. On return from the call, save `corrid`.

3. When you call `tpdequeue` to check for a reply, specify the reply queue in the *`qname`* argument and set `flags` to turn on the bits for the following flags:

   ```
   TPQCORRID       TPQREPLYQ
   TPQFAILUREQ     TPQMSGID
   TPQGETCORRID
   ```

   Use the saved correlation identifier to populate `corrid` before issuing the call. If the call to `tpdequeue` fails and sets `tperrno` to `TPEDIAGNOSTIC`, then further information is available in `diagnostic`. If you receive the error code `QMENOMSG`, it means that no message was available for dequeuing.

4. Set up another call to `tpdequeue`. This time have *`qname`* point to the name of the failure queue and set `flags` to turn on the bits for the following flags:

   ```
   TPQCORRID       TPQREPLYQ
   TPQFAILUREQ     TPQMSGID
   TPQGETBYCORRID
   ```

   Populate `corrid` with the correlation identifier. When the call returns, check *`len`* to see if data has been received and check `urcode` to see if the service has returned a user return code.

Sequential Processing of Messages

Sequential processing of messages can be achieved by having one service enqueue a message for the next service in the chain before its transaction is committed. The originating process can track the progress of the sequence with a series of `tpdequeue` calls to the `reply_queue`, if each member uses the same correlation-ID and returns a 0 length reply.

Alternatively, word of the successful completion of the entire sequence can be returned to the originator by using unsolicited notification. To make sure that the last transaction in the sequence ended with a `tpcommit`, a job step can be added that calls `tpnotify` using the client identifier that is carried in the `TPCTL` structure returned from `tpdequeue` or in the `TPSVCINFO` structure passed to the service. The originating client must have called `tpsetunsol` to name the unsolicited message handler being used.

Using Queues to Transfer Anything

In all of the foregoing discussion of enqueuing and dequeuing messages there has been an implicit assumption that the queues were being used as an alternative form of request/response processing. It may have occurred to you that the message itself does not have to be a service request and you would be correct. The queued message facility can be used equally as effectively to transfer data from one process to another.

If it suits your application to use BEA TUXEDO System/Q for this purpose, have the administrator create a separate queue and code your own receiving program for dequeuing *messages* from that queue.

4 BEA TUXEDO System/Q COBOL Language Programming

This chapter deals with the use of the ATMI COBOL language functions for enqueuing and dequeuing messages: `TPENQUEUE` and `TPDEQUEUE`, plus some ancillary functions.

Prerequisite Knowledge

The BEA TUXEDO programmer coding client or server programs for the queued message facility should be familiar with the COBOL language binding to the BEA TUXEDO ATMI. General guidance on BEA TUXEDO programming is available in the *BEA TUXEDO COBOL Guide*. Detailed pages on all the ATMI functions are in Section 3cbl of the *BEA TUXEDO Reference Manual*.

Where Requests Can Originate

The calls used to place a message on a BEA TUXEDO System/Q queue can originate in any client or server process associated with the application. The list includes:

- Clients or servers on the same machine as the queue space or on another machine on the network.
- Conversational programs, although you cannot have a conversational connection with a queue (or with the `TMQUEUE`(5) server).
- Workstation clients via a surrogate process on the native side; the administrative interface is also entirely on the native side.

Emphasis on the Default Case

The coverage of BEA TUXEDO System/Q programming in this chapter reflects the illustration in Chapter 1, or at least the left-hand portion of it. In that figure a client (or a process acting in the role of a client) queues a message by calling `TPENQUEUE` and specifying a queue space available through the `TMQUEUE` server. The client later retrieves a reply via a `TPDEQUEUE` call to `TMQUEUE`.

The illustration in Chapter 1 goes on to show the queued message being dequeued by the server `TMQFORWARD` and sent to an application server for processing (via `TPCALL`). When a reply to the `TPCALL` is received, `TMQFORWARD` enqueues the reply message. Since a major goal of `TMQFORWARD` is to provide an interface between the queue space and existing application services, it does not require further application coding. For that reason, this chapter concentrates on the client-to-qspace side.

Some examples of customization are given after the discussion of the basic model.

Enqueuing Messages

The syntax for TPENQUEUE is as follows.

```
01 TPQUEDEF-REC.
   COPY TPQUEDEF.
01 TPTYPE-REC.
   COPY TPTYPE.
01 DATA-REC.
   COPY User Data.
01 TPSTATUS-REC.
   COPY TPSTATUS.
CALL "TPENQUEUE" USING TPQUEDEF-REC TPTYPE-REC DATA-REC TPSTATUS-REC.
```

When a TPENQUEUE call is issued it tells the system to store a message on the queue identified in QNAME in *TPQUEDEF-REC* in the space identified in QSPACE-NAME in *TPQUEDEF-REC*. The message is in *DATA-REC*, and LEN in *TPTYPE-REC* has the length of the message. By the use of settings in *TPQUEDEF-REC*, the system is informed how the call to TPENQUEUE is to be handled. Further information about the handling of the enqueued message and replies is provided in the *TPQUEDEF-REC* structure.

Command Line Arguments, TPENQUEUE(3)

There are some important arguments to control the operation of TPENQUEUE(3cbl). Let’s look at some of them.

TPENQUEUE: the QSPACE-NAME in TPQUEDEF-REC Argument

QSPACE-NAME identifies a queue space previously created by the administrator. When a server is defined in the SERVERS section of the configuration file, the service names it offers are aliases for the actual queue space name (which is specified as part of the OPENINFO parameter in the GROUPS section). For example, when your application uses the server TMQUEUE, the value pointed at by QSPACE-NAME is the name of a service advertised by TMQUEUE. If no service aliases are defined, the default service is the same as the server name, TMQUEUE. In this case the configuration file can include the following.

```
TMQUEUE
        SRVGRP = QUE1  SRVID = 1
```

```
        GRACE = 0  RESTART = Y CONV = N
        CLOPT = "-A"
or
        CLOPT = "-s TMQUEUE"
```

The entry for server group `QUE1` has an `OPENINFO` parameter that specifies the resource manager, the pathname of the device and the queue space name. The `QSPACE-NAME` argument in a client program can then look like this.

```
 01  TPQUEDEF-REC.
     COPY TPQUEDEF.
 01  TPTYPE-REC.
     COPY TPTYPE.
 01  TPSTATUS-REC.
     COPY TPSTATUS.
 01  USER-DATA-REC  PIC X(100).
*
*
*
 MOVE LOW-VALUES TO TPQUEDEF-REC.
 MOVE "TMQUEUE" TO QSPACE-NAME IN TPQUEDEF-REC.
 MOVE "STRING" TO QNAME IN TPQUEDEF-REC.
 SET TPTRAN IN TPQUEDEF-REC TO TRUE.
 SET TPBLOCK IN TPQUEDEF-REC TO TRUE.
 SET TPTIME IN TPQUEDEF-REC TO TRUE.
 SET TPSIGRSTRT IN TPQUEDEF-REC TO TRUE.
 MOVE LOW-VALUES TO TPTYPE-REC.
 MOVE "STRING" TO REC-TYPE IN TPTYPE-REC.
 MOVE LENGTH OF USER-DATA-REC TO LEN IN TPTYPE-REC.
 CALL "TPENQUEUE" USING
         TPQUEDEF-REC
         TPTYPE-REC
         USER-DATA-REC
         TPSTATUS-REC.
```

The example shown on the reference page for `TMQUEUE`(5) shows how alias service names can be included when the server is built and specified in the configuration file. The example in Appendix A, “A Sample Application,” also specifies an alias service name.

TPENQUEUE: the QNAME in TPQUEDEF-REC Argument

Within a queue space, message queues are named according to application services that process the requests. `QNAME` contains such a value; an exception in which `QNAME` is not an application service is described later in the chapter.

TPENQUEUE: the DATA-REC and LEN in TPTYPE-REC Arguments

DATA-REC contains the message to be processed. LEN in *TPTYPE-REC* gives the length of the message. Some BEA TUXEDO record types (VIEW, for example) do not require LEN to be specified; in such cases, the argument is ignored. If RECTYPE in *TPTYPE-REC* is SPACES, *DATA-REC* and LEN are ignored and the message is enqueued with no data portion.

TPENQUEUE: the Settings in TPQUEDEF-REC

Settings in *TPQUEDEF-REC* are used to tell the BEA TUXEDO system how the TPENQUEUE call is handled; the following are valid settings:

TPNOTRAN
: If the caller is in transaction mode, this setting specifies that the enqueuing of the message is to be done in a separate transaction. Either TPNOTRAN or TPTRAN must be set.

TPTRAN
: If the caller is in transaction mode, this setting specifies that the enqueuing of the message is to be done within the same transaction. Either TPNOTRAN or TPTRAN must be set.

TPNOBLOCK
: If this setting is set and a blocking condition exists, the call fails immediately with TP-STATUS set to TPEBLOCK. Either TPNOBLOCK or TPBLOCK must be set.

TPBLOCK
: If this setting is set and a blocking condition exists, the call blocks until the condition subsides or transaction timeout occurs. Either TPNOBLOCK or TPBLOCK must be set.

TPNOTIME
: This setting asks that the call be immune to blocking timeouts; transaction timeouts may still occur. Either TPNOTIME or TPTIME must be set.

TPTIME
: This setting asks that the call will receive blocking timeouts. Either TPNOTIME or TPTIME must be set.

TPSIGRSTRT

This setting says that any underlying system calls that are interrupted by a signal should be reissued. Either TPSIGRSTRT or TPNOSIGRSTRT must be set.

TPNOSIGRSTRT

This setting says that any underlying system calls that are interrupted by a signal should not be reissued. The call fails and sets TP-STATUS to TPEGOTSIG. Either TPSIGRSTRT or TPNOSIGRSTRT must be set.

The TPQUEDEF-REC Structure

The *TPQUEDEF-REC* structure has members that are used by the application and by the BEA TUXEDO system to pass parameters in both directions between application programs and the queued message facility. It is defined in the COBOL COPY file. The client that calls *TPQUEDEF-REC* uses settings to mark members the application wants the system to fill in. The structure is also used by TPDEQUEUE; some of the members do not come into play until the application calls that function. The complete structure is shown in Listing 4-1.

Listing 4-1 The TPQUEDEF-REC Structure

```
05 TPBLOCK-FLAG        PIC S9(9) COMP-5.
        88 TPNOBLOCK            VALUE 0.
        88 TPBLOCK              VALUE 1.
05 TPTRAN-FLAG          PIC S9(9) COMP-5.
        88 TPNOTRAN             VALUE 0.
        88 TPTRAN               VALUE 1.
05 TPTIME-FLAG          PIC S9(9) COMP-5.
        88 TPNOTIME             VALUE 0.
        88 TPTIME               VALUE 1.
05 TPSIGRSTRT-FLAG      PIC S9(9) COMP-5.
        88 TPNOSIGRSTRT         VALUE 0.
        88 TPSIGRSTRT           VALUE 1.
05 TPNOCHANGE-FLAG      PIC S9(9) COMP-5.
        88 TPNOCHANGE           VALUE 0.
        88 TPCHANGE             VALUE 1.
05 TPQUE-ORDER-FLAG     PIC S9(9) COMP-5.
        88 TPQDEFAULT           VALUE 0.
        88 TPQTOP               VALUE 1.
        88 TPQBEFOREMSGID       VALUE 2.
05 TPQUE-TIME-FLAG      PIC S9(9) COMP-5.
        88 TPQNOTIME            VALUE 0.
```

```
        88 TPQTIME-ABS          VALUE 1.
        88 TPQTIME-REL          VALUE 2.
05 TPQUE-PRIORITY-FLAG  PIC S9(9) COMP-5.
        88 TPQNOPRIORITY        VALUE 0.
        88 TPQPRIORITY          VALUE 1.
05 TPQUE-CORRID-FLAG    PIC S9(9) COMP-5.
        88 TPQNOCORRID          VALUE 0.
        88 TPQCORRID            VALUE 1.
05 TPQUE-REPLYQ-FLAG    PIC S9(9) COMP-5.
        88 TPQNOREPLYQ          VALUE 0.
        88 TPQREPLYQ            VALUE 1.
05 TPQUE-FAILQ-FLAG     PIC S9(9) COMP-5.
        88 TPQNOFAILUREQ        VALUE 0.
        88 TPQFAILUREQ          VALUE 1.
05 TPQUE-MSGID-FLAG     PIC S9(9) COMP-5.
        88 TPQNOMSGID           VALUE 0.
        88 TPQMSGID             VALUE 1.
05 TPQUE-GETBY-FLAG     PIC S9(9) COMP-5.
        88 TPQGETNEXT           VALUE 0.
        88 TPQGETBYMSGID        VALUE 1.
        88 TPQGETBYCORRID       VALUE 2.
05 TPQUE-WAIT-FLAG      PIC S9(9) COMP-5.
        88 TPQNOWAIT            VALUE 0.
        88 TPQWAIT              VALUE 1.
05 DIAGNOSTIC           PIC S9(9) COMP-5.
        88 QMEINVAL             VALUE -1.
        88 QMEBADRMID           VALUE -2.
        88 QMENOTOPEN           VALUE -3.
        88 QMETRAN              VALUE -4.
        88 QMEBADMSGID          VALUE -5.
        88 QMESYSTEM            VALUE -6.
        88 QMEOS                VALUE -7.
        88 QMENOTA              VALUE -8.
        88 QMEPROTO             VALUE -9.
        88 QMEBADQUEUE          VALUE -10.
        88 QMENOMSG             VALUE -11.
        88 QMEINUSE             VALUE -12.
        88 QMENOSPACE           VALUE -13.
05 DEQ-TIME              PIC 9(9) COMP-5.
05 PRIORITY              PIC S9(9) COMP-5.
05 MSGID                 PIC X(32).
05 CORRID                PIC X(32).
05 QNAME                 PIC X(15).
05 QSPACE-NAME           PIC X(15).
05 REPLYQUEUE            PIC X(15).
05 FAILUREQUEUE          PIC X(15).
05 CLIENTID OCCURS 4 TIMES PIC S9(9) COMP-5.
05 APPL-RETURN-CODE      PIC S9(9) COMP-5.
05 APPKEY                PIC S9(9) COMP-5.
```

The following is a list of valid settings for the parameters controlling input information for `TPENQUEUE`.

`TPQTOP`
: Setting this indicates that the queue ordering be overridden and the message placed at the top of the queue. This request may not be granted depending on whether or not the queue was configured to allow overriding the queue ordering to put a message at the top of the queue.

`TPQBEFOREMSGID`
: Setting this indicates that the queue ordering be overridden and the message placed in the queue before the message identified by `MSGID`. This request may not be granted depending on whether or not the queue was configured to allow overriding the queue ordering to put a message ahead of another by `MSGID`. `TPQTOP` and `TPQBEFOREMSGID` are mutually exclusive settings. Assumes a prior (successful) call with `TPQMSGID` set.

`TPQTIME-ABS`
: If set, the request is to be processed after the time specified by `DEQ-TIME`. The `DEQ-TIME` is an absolute time value as generated by `time`(2) or `mktime`(3C), if they are available in your UNIX operating system, or `gp_mktime`(3c), provided with the BEA TUXEDO system. The value set in `DEQ-TIME` is the number of seconds since 00:00:00 UTC, January 1, 1970. `TPQTIME-ABS` can be overridden and the message dequeued immediately by `MSGID` or `CORRID`.

`TPQTIME-REL`
: If set, the request is to be processed relative to the completion of the queuing transaction. `DEQ-TIME` specifies the number of seconds to delay after the transaction completes before the submitted request should be processed. `TPQTIME-REL` can be overridden and the message dequeued immediately by `MSGID` or `CORRID`. `TPQTIME-ABS` and `TPQTIME-REL` are mutually exclusive settings.

`TPQPRIORITY`
: If set, the priority at which the request should be enqueued is stored in `PRIORITY`. `PRIORITY` must be in the range 1 to 100, inclusive.

`TPQCORRID`
: If set, the correlation identifier value specified in `CORRID` is available when a request is dequeued with `TPDEQUEUE`. This identifier accompanies any reply or failure message that is queued so an application can correlate a reply with a particular request. The entire value should be initialized such that the value can be matched at a later time.

`TPQREPLYQ`

If set, a reply queue named in `REPLYQUEUE` is associated with the queued message. Any reply to the message will be queued to the named queue within the same queue space as the request message. If a reply is generated for the service and a reply queue is not specified or the reply queue does not exist, the reply is dropped.

`TPQFAILUREQ`

If set, a failure queue named in `FAILUREQUEUE` is associated with the queued message. If a failure occurs when executing the enqueued message, a failure message will go to the named queue within the same queue space as the original request message.

Additionally, the `APPL-RETURN-CODE` member of *`TPQUEDEF-REC`* can be set with a user-return code. This value will be returned to the application that calls `TPDEQUEUE` to dequeue the message.

On output from `TPENQUEUE`, the following elements may be set in the *`TPQUEDEF-REC`* structure.

```
05 TPQUE-MSGID-FLAG     PIC S9(9) COMP-5.
        88 TPQNOMSGID           VALUE 0.
        88 TPQMSGID             VALUE 1.
05 DIAGNOSTIC           PIC S9(9) COMP-5.
        88 QMEINVAL             VALUE -1.
        88 QMEBADRMID           VALUE -2.
        88 QMENOTOPEN           VALUE -3.
        88 QMETRAN              VALUE -4.
        88 QMEBADMSGID          VALUE -5.
        88 QMESYSTEM            VALUE -6.
        88 QMEOS                VALUE -7.
        88 QMENOTA              VALUE -8.
        88 QMEPROTO             VALUE -9.
        88 QMEBADQUEUE          VALUE -10.
        88 QMENOMSG             VALUE -11.
        88 QMEINUSE             VALUE -12.
        88 QMENOSPACE           VALUE -13.
05 MSGID                PIC X(32).
```

Setting of `TPQUE-MSGID-FLAG` requests output information from `TPENQUEUE`. If this setting bit is turned on when `TPENQUEUE` is called, then the associated element in the structure is populated if available and the bit remains set. If the value is not available, `TPENQUEUE` completes with the setting bit turned off.

TPQMSGID
: If set and the call to TPENQUEUE was successful, the message identifier will be stored in MSGID. If the call to TPENQUEUE fails and TP-STATUS is set to TPEDIAGNOSTIC, a value indicating the reason for failure is returned in DIAGNOSTIC. Following are the possible values.

[QMEINVAL]
: An invalid setting value was specified.

[QMEBADRMID]
: An invalid resource manager identifier was specified.

[QMENOTOPEN]
: The resource manager is not currently open.

[QMETRAN]
: The call was made with TPNOTRAN set and an error occurred trying to start a transaction in which to enqueue the message.

[QMEBADMSGID]
: An invalid message identifier was specified.

[QMESYSTEM]
: A system error has occurred. The exact nature of the error is written to a log file.

[QMEOS]
: An operating system error has occurred.

[QMENOTA]
: The transaction in which the message was enqueued was aborted.

[QMEPROTO]
: An enqueue was done when the transaction state was not active.

[QMEBADQUEUE]
: An invalid or deleted queue name was specified.

[QMENOSPACE]
: There is no space on the queue for the message.

The remaining members of the control structure are not used on input to TPENQUEUE.

Overriding the Queue Order

If the administrator in creating a queue allows `TPENQUEUE` calls to override the order of messages on the queue, you have two mutually exclusive ways to use that capability. You can specify that the message is to be placed at the top of the queue by setting `TPQTOP` or you can specify that it be placed ahead of a specific message by setting `TPQBEFOREMSGID` and setting `MSGID` to the ID of the message you wish to precede. This assumes that you saved the message-ID from a previous call in order to be able to use it here. Your administrator must tell you what the queue supports; it can be created to allow either or both of these overrides, or to allow neither.

Overriding the Queue Priority

If the queue was created with `PRIORITY` as a queue ordering parameter, you can set a value in `PRIORITY` to specify the dequeuing priority for the message. The value must be in the range 1 to 100; the higher the number the higher the priority, unlike the UNIX `nice` command. If `PRIORITY` was not one of the queue ordering parameters, setting a priority here has no effect.

Setting a Dequeuing Time

A queue can be created with `time` as a queue ordering parameter. When this is the case, you can specify in `DEQ-TIME` either an absolute time for the message to be dequeued or a time relative to the enqueuing transaction. You set either `TPQTIME-ABS` or `TPQTIME-REL` to say how the value should be treated.

The following example shows how to enqueue a message with a relative time. It will become eligible for processing sixty seconds in the future.

```
  01  TPQUEDEF-REC.
      COPY TPQUEDEF.
  01  TPTYPE-REC.
      COPY TPTYPE.
  01  TPSTATUS-REC.
      COPY TPSTATUS.
  01  USER-DATA-REC  PIC X(100).
*
*
*
  MOVE LOW-VALUES TO TPQUEDEF-REC.
  MOVE "QSPACE1" TO QSPACE-NAME IN TPQUEDEF-REC.
```

```
MOVE "Q1" TO QNAME IN TPQUEDEF-REC.
SET TPTRAN IN TPQUEDEF-REC TO TRUE.
SET TPBLOCK IN TPQUEDEF-REC TO TRUE.
SET TPTIME IN TPQUEDEF-REC TO TRUE.
SET TPSIGRSTRT IN TPQUEDEF-REC TO TRUE.
SET TPQDEFAULT IN TPQUEDEF-REC TO TRUE.
SET TPQTIME-REL IN TPQUEDEF-REC TO TRUE.
MOVE 60 TO DEQ-TIME IN TPQUEDEF-REC.
SET TPQNOPRIORITY IN TPQUEDEF-REC TO TRUE.
SET TPQNOCORRID IN TPQUEDEF-REC TO TRUE.
SET TPQNOREPLYQ IN TPQUEDEF-REC TO TRUE.
SET TPQNOFAILUREQ IN TPQUEDEF-REC TO TRUE.
SET TPQMSGID IN TPQUEDEF-REC TO TRUE.
MOVE LOW-VALUES TO TPTYPE-REC.
MOVE "STRING" TO REC-TYPE IN TPTYPE-REC.
MOVE LENGTH OF USER-DATA-REC TO LEN IN TPTYPE-REC.
CALL "TPENQUEUE" USING
        TPQUEDEF-REC
        TPTYPE-REC
        USER-DATA-REC
        TPSTATUS-REC.
```

TPENQUEUE and Transactions

Messages are always enqueued within a transaction; the only question is, within whose transaction? There are two choices. If caller of TPENQUEUE is in transaction mode and TPTRAN is set, then the enqueuing is done within the caller's transaction. The caller knows for certain from the success or failure of TPENQUEUE whether the message was enqueued or not. If the call succeeds, the message is guaranteed to be on the queue. If the call fails, the transaction is rolled back, including the part where the message was placed on the queue.

If the caller of TPENQUEUE is not in transaction mode or if TPNOTRAN is set, the message is enqueued in a separate transaction. If the call to TPENQUEUE returns success, the message is guaranteed to be on the queue. If the call to TPENQUEUE fails with a communication error or with a transaction or blocking timeout, the caller is left in doubt about whether the failure occurred before or after the message was enqueued.

Note that specifying TPNOTRAN while the caller is not in transaction mode has no meaning.

Dequeuing Replies

The syntax for `TPDEQUEUE` is as follows.

```
01 TPQUEDEF-REC.
   COPY TPQUEDEF.
01 TPTYPE-REC.
   COPY TPTYPE.
01 DATA-REC.
   COPY User Data.
01 TPSTATUS-REC.
   COPY TPSTATUS.
CALL "TPDEQUEUE" USING TPQUEDEF-REC TPTYPE-REC DATA-REC TPSTATUS-REC.
```

When this call is issued it tells the system to dequeue a message from the `QNAME` in *`TPQUEDEF-REC`* queue, in the space named `QSPACE-NAME` in *`TPQUEDEF-REC`*. The message is placed in *`DATA-REC`*. `LEN` in *`TPTYPE-REC`* is set to the length of the data. If `LEN` is 0 on return from `TPDEQUEUE`, the message had no data portion. By the use of settings in *`TPQUEDEF-REC`* the system is informed how the call to `TPDEQUEUE` is to be handled.

Command Line Arguments, TPDEQUEUE(3)

There are some important arguments to control the operation of `TPDEQUEUE`(3cbl). Let's look at some of them.

TPDEQUEUE: the QSPACE-NAME in TPQUEDEF-REC Argument

`QSPACE-NAME` identifies a queue space previously created by the administrator. When a server is defined in the `SERVERS` section of the configuration file, the service names it offers are aliases for the actual queue space name (which is specified as part of the `OPENINFO` parameter in the `GROUPS` section). For example, when your application uses the server `TMQUEUE`, the value pointed at by `QSPACE-NAME` is the name of a service advertised by TMQUEUE. If no service aliases are defined, the default service is the same as the server name, TMQUEUE. In this case the configuration file can include the following.

```
TMQUEUE
        SRVGRP = QUE1  SRVID = 1
        GRACE = 0  RESTART = Y CONV = N
        CLOPT = "-A"
or
        CLOPT = "-s TMQUEUE"
```

The entry for server group QUE1 has an OPENINFO parameter that specifies the resource manager, the pathname of the device and the queue space name. The QSPACE-NAME argument in a client program can then look like this:

```
  01 TPQUEDEF-REC.
     COPY TPQUEDEF.
  01 TPTYPE-REC.
     COPY TPTYPE.
  01 TPSTATUS-REC.
     COPY TPSTATUS.
  01 USER-DATA-REC  PIC X(100).
*
*
*
  MOVE LOW-VALUES TO TPQUEDEF-REC.
  MOVE "TMQUEUE" TO QSPACE-NAME IN TPQUEDEF-REC.
  MOVE "REPLYQ" TO QNAME IN TPQUEDEF-REC.
  SET TPTRAN IN TPQUEDEF-REC TO TRUE.
  SET TPBLOCK IN TPQUEDEF-REC TO TRUE.
  SET TPTIME IN TPQUEDEF-REC TO TRUE.
  SET TPSIGRSTRT IN TPQUEDEF-REC TO TRUE.
  MOVE LOW-VALUES TO TPTYPE-REC.
  MOVE "STRING" TO REC-TYPE IN TPTYPE-REC.
  MOVE LENGTH OF USER-DATA-REC TO LEN IN TPTYPE-REC.
  CALL "TPDEQUEUE" USING
          TPQUEDEF-REC
          TPTYPE-REC
          USER-DATA-REC
          TPSTATUS-REC.
```

The example shown on the reference page for TMQUEUE(5) shows how alias service names can be included when the server is built and specified in the configuration file. The example in Appendix A, “A Sample Application,” also specifies an alias service name.

TPDEQUEUE: the QNAME in TPQUEDEF-REC Argument

Reply queue names in a queue space need to be agreed upon within the application. The administrator creates a reply queue (and often an error queue) in the same manner a message queue is created. `QNAME` contains the name.

TPDEQUEUE: the DATA-REC and LEN in TPTYPE-REC Arguments

The arguments have a different flavor than they do on `TPENQUEUE`. *`DATA-REC`* is where the system is to place the message being dequeued.

It is an error for `LEN` to be 0 on input. When `TPDEQUEUE` returns, `LEN` contains the length of the data retrieved. If it is 0, it means that the reply had no data portion. This can be a legitimate and successful reply in some applications; receiving even a 0 length reply can be used to show successful processing of the enqueued request. If you wish to know whether the record has changed from before the call to `TPDEQUEUE`, save the prior length and compare it to `LEN`. If the reply is larger than `LEN`, then *`DATA-REC`* will contain only as many bytes as will fit. The remainder are discarded and `TPDEQUEUE` fails with `TPTRUNCATE`.

TPDEQUEUE: the Settings in TPQUEDEF-REC

Settings in *`TPQUEDEF-REC`* are used to tell the BEA TUXEDO system how the `TPDEQUEUE` call is handled; the following are valid settings:

`TPNOTRAN`
: If the caller is in transaction mode, this setting specifies that the message is to be dequeued in a separate transaction. Either `TPNOTRAN` or `TPTRAN` must be set.

`TPTRAN`
: If the caller is in transaction mode, this setting specifies that the message is to be dequeued within the same transaction. Either `TPNOTRAN` or `TPTRAN` must be set.

`TPNOBLOCK`
: If this setting is set and a blocking condition exists, the call fails immediately with `TP-STATUS` set to `TPEBLOCK`. This blocking condition does not include blocking on the queue itself if the `TPQWAIT` setting is specified. Either `TPNOBLOCK` or `TPBLOCK` must be set.

TPBLOCK
: If this setting is set and a blocking condition exists, the call blocks until the condition subsides or timeout occurs (TPETIME). This blocking condition does not include blocking on the queue itself if the TPQWAIT setting is specified. Either TPNOBLOCK or TPBLOCK must be set.

TPNOTIME
: This setting asks that the call be immune to blocking timeouts; transaction timeouts may still occur. Either TPNOTIME or TPTIME must be set.

TPTIME
: This setting asks that the call receive blocking timeouts. Either TPNOTIME or TPTIME must be set.

TPNOCHANGE
: When this setting is set, the record type of *DATA-REC* is not allowed to change. That is, the type and sub-type of the received record must match the type and subtype of the record *DATA-REC*. Either TPNOCHANGE or TPCHANGE must be set.

TPCHANGE
: By default, if a record is received that differs in type from the record *DATA-REC*, then *DATA-REC*'s record type changes to the received record's type so long as the receiver recognizes the incoming record type. That is, the type and sub-type of the received record must match the type and sub-type of the record *DATA-REC*. Either TPNOCHANGE or TPCHANGE must be set.

TPSIGRSTRT
: This setting says that any underlying system calls that are interrupted by a signal should be reissued. Either TPSIGRSTRT or TPNOSIGRSTRT must be set.

TPNOSIGRSTRT
: When this setting is specified and a signal is received, the call fails and sets TP-STATUS to TPEGOTSIG. Either TPSIGRSTRT or TPNOSIGRSTRT must be set.

The first argument to TPDEQUEUE is a structure *TPQUEDEF-REC*. The *TPQUEDEF-REC* structure has members that are used by the application and by the BEA TUXEDO system to pass parameters in both directions between application programs and the queued message facility. The client that calls TPDEQUEUE uses settings to mark members the application wants the system to fill in. As described earlier, the structure is also used by TPENQUEUE; some of the members only apply to that function. The entire structure is shown in Listing 4-1.

On input to `TPDEQUEUE`, the following elements may be set in the `TPQUEDEF` structure.

```
05 TPBLOCK-FLAG         PIC S9(9) COMP-5.
       88 TPNOBLOCK             VALUE 0.
       88 TPBLOCK               VALUE 1.
05 TPTRAN-FLAG  PIC S9(9) COMP-5.
       88 TPNOTRAN              VALUE 0.
       88 TPTRAN                VALUE 1.
05 TPTIME-FLAG  PIC S9(9) COMP-5.
       88 TPNOTIME              VALUE 0.
       88 TPTIME                VALUE 1.
05 TPSIGRSTRT-FLAG      PIC S9(9) COMP-5.
       88 TPNOSIGRSTRT VALUE 0.
       88 TPSIGRSTRT            VALUE 1.
05 TPNOCHANGE-FLAG      PIC S9(9) COMP-5.
       88 TPNOCHANGE   VALUE 0.
       88 TPCHANGE              VALUE 1.
05 TPQUE-ORDER-FLAG     PIC S9(9) COMP-5.
       88 TPQDEFAULT            VALUE 0.
       88 TPQTOP                VALUE 1.
       88 TPQBEFOREMSGID        VALUE 2.
05 TPQUE-TIME-FLAG      PIC S9(9) COMP-5.
       88 TPQNOTIME             VALUE 0.
       88 TPQTIME-ABS  VALUE 1.
       88 TPQTIME-REL  VALUE 2.
05 TPQUE-PRIORITY-FLAG  PIC S9(9) COMP-5.
       88 TPQNOPRIORITY         VALUE 0.
       88 TPQPRIORITY  VALUE 1.
05 TPQUE-CORRID-FLAG    PIC S9(9) COMP-5.
       88 TPQNOCORRID  VALUE 0.
       88 TPQCORRID             VALUE 1.
05 TPQUE-REPLYQ-FLAG    PIC S9(9) COMP-5.
       88 TPQNOREPLYQ  VALUE 0.
       88 TPQREPLYQ             VALUE 1.
05 TPQUE-FAILQ-FLAG     PIC S9(9) COMP-5.
       88 TPQNOFAILUREQ         VALUE 0.
       88 TPQFAILUREQ  VALUE 1.
05 TPQUE-MSGID-FLAG     PIC S9(9) COMP-5.
       88 TPQNOMSGID            VALUE 0.
       88 TPQMSGID              VALUE 1.
05 TPQUE-GETBY-FLAG     PIC S9(9) COMP-5.
       88 TPQGETNEXT            VALUE 0.
       88 TPQGETBYMSGID         VALUE 1.
       88 TPQGETBYCORRID        VALUE 2.
05 TPQUE-WAIT-FLAG      PIC S9(9) COMP-5.
       88 TPQNOWAIT             VALUE 0.
       88 TPQWAIT               VALUE 1.
05 MSGID                PIC X(32).
05 CORRID               PIC X(32).
05 QNAME                PIC X(15).
05 QSPACE-NAME  PIC X(15).
```

Following are valid settings on input to `TPDEQUEUE`.

`TPNO`*`string`*
: No settings are set. No information is taken from the control structure.

`TPQGETBYMSGID`
: If set, it requests that the message identified by `MSGID` be dequeued. The message identifier would be one that was returned by a prior call to `TPENQUEUE`. This option cannot be used with the `TPQWAIT` setting.

`TPQGETBYCORRID`
: If set, it requests that the message with the correlation identifier specified by `CORRID` be dequeued. The correlation identifier would be one that the application specified when enqueuing the message with `TPENQUEUE`. This option cannot be used with the `TPQWAIT` setting.

`TPQWAIT`
: If set, it indicates that an error should not be returned if the queue is empty. Instead, the process should block until a message is available.

Following is a list of valid settings for the parameters controlling output information from `TPDEQUEUE`. If the setting is true when `TPDEQUEUE` is called, then the associated element (see Listing 4-1) in the structure is populated if available and the setting remains true. If the value is not available, the setting will not be true after `TPDEQUEUE` completes.

`TPQPRIORITY`
: If set and the value is available, the priority at which the message was queued is stored in `PRIORITY`.

`TPQMSGID`
: If set and the call to `TPDEQUEUE` was successful, the message identifier will be stored in `MSGID`.

`TPQCORRID`
: If set and the call to `TPDEQUEUE` was successful and the message was queued with a correlation identifier, the value will be stored in `CORRID`. Any reply to a queue must have this correlation identifier.

TPQREPLYQ

If set and the message is associated with a reply queue, the value will be stored in REPLYQUEUE. Any reply to the message should go to the named reply queue within the same queue space as the request message.

TPQFAILUREQ

If set and the message is associated with a failure queue, the value will be stored in FAILUREQUEUE. Any failure message should go to the named failure queue within the same queue space as the request message.

If the call to TPDEQUEUE failed and TP-STATUS is set to TPEDIAGNOSTIC, a value indicating the reason for failure is returned in DIAGNOSTIC. The valid settings for DIAGNOSTIC include those shown above for TPENQUEUE and the following additional codes.

[QMENOMSG]

No message was available for dequeuing.

[QMEINUSE]

When dequeuing a message by correlation or message identifier, the specified message is in use by another transaction. Otherwise, all messages currently on the queue are in use by other transactions.

Using TPQWAIT

When TPDEQUEUE is called with TPQWAIT set, the TMQUEUE server may be blocked waiting for a message to come onto the queue. The amount of time it is blocked can be controlled by the transaction timeout value set by the caller in TPBEGIN or by the -t option in the CLOPT parameter of the TMQUEUE server (if the transaction is started in the server). To avoid blocking TPENQUEUE calls that also use the TMQUEUE server, it may be desirable to configure two or more TMQUEUE servers (or MSSQ sets) offering different service names for the same queue space. It could be set up so that all enqueue and non-waiting dequeue operations use one set of TMQUEUE servers and all waiting dequeue operations use the second set.

Error Handling

In considering how best to handle errors in dequeuing it is helpful to differentiate between errors encountered by TMQFORWARD as it attempts to dequeue a message to forward to the requested service and errors that occur in the service that processes the request. This subject was discussed in Chapter 1, “Introduction and Overview of BEA TUXEDO System/Q,” but is repeated here in the context of writing application programs.

By default, if a message is dequeued within a transaction and the transaction is rolled back, then the message ends up back on the queue and can be dequeued and executed again. It may be desirable to delay for a short period before retrying to dequeue and execute the message, allowing the transient problem to clear (for example, allowing for locks in a database to be released by another transaction). Normally, a limit on the number of retries is also useful to ensure that an application flaw doesn't cause significant waste of resources. When a queue is configured by the administrator, both a retry count and a delay period (in seconds) can be specified. A retry count of 0 implies that no retries are done. After the retry count is reached, the message is moved to an error queue that is configured by the administrator for the queue space. If the error queue is not configured, then messages that have reached the retry count are simply deleted. Messages on the error queue must be handled by the administrator who must work out a way of notifying the originator that meets the requirements of the application. This kind of handling is almost transparent to the originating program that put the message on the queue. There is a virtual guarantee that once a message is successfully enqueued it will be processed according to the parameters of TPENQUEUE and the attributes of the queue. Notification that a message has been moved to the error queue should be a rare occurrence in a system that has properly tuned its queue parameters.

A failure queue (normally, different from the queue space error queue) may be associated with each queued message. This queue is specified on the enqueuing call as the place to put any failure messages. The failure message for a particular request can be identified by an application-generated correlation identifier that is associated with the message when it is enqueued.

The default behavior of retrying until success (or a predefined limit) is quite appropriate when the failure is caused by a transient problem that is later resolved, allowing the message to be handled appropriately.

There are cases where the problem is not transient. For example, the queued message may request operating on an account that does not exist (and the application is such that it won't come into existence within a reasonable time period if at all). In this case, it is desirable not to waste any resources by trying again. If the application programmer or administrator determines that failures for a particular operation are never transient, then it is simply a matter of setting the retry count to zero, although this will require a mechanism to constantly clear the queue space error queue of these messages (for example, a background client that reads the queue periodically). More likely, it is the case that some problems will be transient (for example, database lock contention) and some problems will be permanent (for example, the account doesn't exist) for the same service.

In the case that the message is processed (dequeued and passed to the application via a `TPCALL`) by `TMQFORWARD`, there is no mechanism in the information returned by `TPCALL` to indicate whether a `TPESVCFAIL` error is caused by a transient or permanent problem.

As in the case where the application is handling the dequeuing, a simple solution is to return success for the service, that is, `TPRETURN` with `TPSUCCESS`, even though the operation failed. This allows the transaction to be committed and the message removed from the queue. If reply messages are being used, the information in the buffer returned from the service can indicate that the operation failed and the message will be enqueued on the reply queue. The `APPL-CODE` in the *`TPSVCRET-REC`* argument of `TPRETURN` can also be used to return application specific information.

In the case where the service fails and the transaction must be rolled back, it is not clear whether or not `TMQFORWARD` should execute a second transaction to remove the message from the queue without further processing. By default, `TMQFORWARD` will not delete a message for a service that fails. `TMQFORWARD`'s transaction is rolled back and the message is restored to the queue. A command line option may be specified for `TMQFORWARD` that indicates that a message should be deleted from the queue if the service fails and a reply message is sent back with length greater than 0. The message is deleted in a second transaction. The queue must be configured with a delay time and retry count for this to work. If the message is associated with a failure queue, the reply data will be enqueued to the failure queue in the same transaction as the one in which the message is deleted from the queue.

A Procedure for Dequeuing Replies

If your application expects to receive replies to queued messages, here is a procedure you may want to follow:

1. As a preliminary step, the queue space must include a reply queue and a failure queue. The application must also agree on the content of the correlation identifier. The service should be coded to return `TPSUCCESS` on a logical failure and return an explanatory code in the `APPL-CODE` in the *`TPSVCRET-REC`* argument of `TPRETURN`.

2. When you call `TPENQUEUE` to put the message on the queue, set the following:

   ```
   TPQCORRID      TPQREPLYQ
   TPQFAILUREQ    TPQMSGID
   ```

   (Fill in the values for `CORRID`, `REPLYQUEUE` and `FAILUREQUEUE` before issuing the call. On return from the call, save `CORRID`.)

3. When you call `TPDEQUEUE` to check for a reply, specify the reply queue in `QNAME` and set the following:

   ```
   TPQCORRID       TPQREPLYQ
   TPQFAILUREQ     TPQMSGID
   TPQGETBYCORRID
   ```

   (Use the saved correlation identifier to populate `CORRID` before issuing the call. If the call to `TPDEQUEUE` fails and sets `TP-STATUS` to `TPEDIAGNOSTIC`, then further information is available in the `DIAGNOSTIC` settings. If you receive the error code `QMENOMSG`, it means that no message was available for dequeuing.)

4. Set up another call to `TPDEQUEUE`. This time have `QNAME` point to the name of the failure queue and set the following:

   ```
   TPQCORRID       TPQREPLYQ
   TPQFAILUREQ     TPQMSGID
   TPQGETBYCORRID
   ```

   Populate `TPQCORRID` with the correlation identifier. When the call returns, check `LEN` to see if data has been received and check `APPL-RETURN-CODE` to see if the service has returned a user return code.

Sequential Processing of Messages

Sequential processing of messages can be achieved by having one service enqueue a message for the next service in the chain before its transaction is committed. The originating process can track the progress of the sequence with a series of `TPDEQUEUE` calls to the `reply_queue`, if each member uses the same correlation-ID and returns a 0 length reply.

Alternatively, word of the successful completion of the entire sequence can be returned to the originator by using unsolicited notification. To make sure that the last transaction in the sequence ended with a `TPCOMMIT`, a job step can be added that calls `TPNOTIFY` using the client identifier that is carried in the *`TPQUEDEF-REC`* structure. The originating client must have called `TPSETUNSOL` to name the unsolicited message handler being used.

Using Queues to Transfer Anything

In all of the foregoing discussion of enqueuing and dequeuing messages there has been an implicit assumption that the queues were being used as an alternative form of request/response processing. It may have occurred to you that the message itself does not have to be a service request and you would be correct. The queued message facility can be used equally as effectively to transfer data from one process to another.

If it suits your application to use BEA TUXEDO System/Q for this purpose, have the administrator create a separate queue and code your own receiving program for dequeuing *messages* from that queue.

A A Sample Application

What This Appendix Is About

This appendix contains a description of a one-client, one-server application using BEA TUXEDO System/Q called `qsample`. An interactive form of this software is distributed with the BEA TUXEDO software.

Some Preliminaries

Before you can run this example the BEA TUXEDO software must be installed and built so that the files and commands referred to in this chapter are available. If you are personally responsible for installing the BEA TUXEDO software, consult the *BEA TUXEDO Installation Guide* for information about how to install the BEA TUXEDO system.

If the installation has already been done by someone else, you need to know the pathname of the root directory of the installed software. You also need to have read and execute permissions on the directories and files in the BEA TUXEDO directory structure so you can copy `qsample` files and execute BEA TUXEDO commands.

The qsample Application

`qsample` is a very basic BEA TUXEDO application that uses BEA TUXEDO System/Q. It has one application client and server, and uses two system servers. `TMQUEUE` and `TMQFORWARD`. The client calls `TMQUEUE` to enqueue a message in a queue space created for `qsample`. The message is dequeued by `TMQFORWARD` and passed to the application server. The server converts a string from lower case to upper case and returns to `TMQFORWARD`. `TMQFORWARD` enqueues the reply message. The client meanwhile has called `TMQUEUE` to dequeue the reply. When the reply is received, the client displays it on the user's screen.

What follows is a procedure to build and run the example.

1. Make a directory for `qsample` and `cd` to it:

   ```
   mkdir qsampdir
   cd qsampdir
   ```

   This is suggested so you will be able to see clearly the `qsample` files you have at the start and the additional files you create along the way. Use the standard shell (`/bin/sh`) or the Korn shell; not the C shell (`/bin/csh`).

2. Copy the `qsample` files.

   ```
   cp $TUXDIR/apps/qsample/* .
   ```

   You will be editing some of the files and making them executable, so it is best to begin with a copy of the files rather than the originals delivered with the software.

3. List the files.

   ```
   $ ls
   README
   client.c
   crlog
   crque
   makefile
   rmipc
   runsample
   server.c
   setenv
   ubb.sample
   $
   ```

The files that make up the application are:

`README`
: A file that describes the application

`setenv`
: A script that sets environment variables

`crlog`
: A script that creates a `TLOG` file

`crque`
: A script that defines the queue space and queues for the application

`makefile`
: A makefile that creates the executables for the application

`client.c`
: The source code for the client program

`server.c`
: The source code for the server program

`ubb.sample`
: The ASCII form of the configuration file for the application

`runsample`
: A script that calls all the necessary commands to build and run the sample application

`rmipc`
: A script that removes the IPC resources for the queue space

4. Edit the files.

   Five of the files have location-specific entries that you must edit to provide your own directory pathnames and machine name.

   The text to be replaced is enclosed in angle brackets. You need to substitute the absolute path for `TUXDIR` and `APPDIR`, and the machine name of the machine you are running on.

   Here is a summary of the required values:

   `TUXDIR`
   : The absolute path of the root directory of the BEA TUXEDO software

   `APPDIR`
   : The absolute path of the directory in which your application will run

`machine`

The machine name of the machine on which your application will run. This name is the output of the `uname -n` command (where `uname` is supported on the target platform).

The six files that must be edited (and the value that you must put in) are:

```
crlog         APPDIR pathname
crque         APPDIR pathname
makefile      TUXDIR pathname
rmipc         APPDIR pathname
ubb.sample    TUXDIR pathname, APPDIR pathname, machine name
setenv        TUXDIR pathname
```

5. Run `runsample`.

The `runsample` script contains 11 commands; each command is preceded by a comment line that says what the command does.

```
#set the environment
 . ./setenv
#build the client and server
make client server
#create the tuxconfig file
tmloadcf -y ubb.sample
#create the TLOG
#create the QUE
#boot the application
tmboot -y
#run the client
client
#shutdown the application
tmshutdown -y
#remove the client and server
make clean
#remove the QUE ipc resources
#remove all files created
rm tuxconfig QUE stdout stderr TLOG ULOG*
```

When you run this script you will see a series of messages on your screen that are output by the various commands. Included among them are the following lines.

```
before: this is a q example
after: THIS IS A Q EXAMPLE
```

The `before:` line is a copy of the string that `client` enqueues for processing by `server`. The `after:` line is what `server` sends back. These two lines prove that the program worked successfully.

Suggestions for Further Exploration

While it might prove interesting to build and run the sample application using `runserver`, you will probably find it more instructive to examine the individual pieces of the application. In this section we suggest some things we recommend you look at and try; you will undoubtedly be able to think of others as you explore the application more closely.

setenv: Setting the Environment

The script `setenv` is an example of a file often used in BEA TUXEDO development. Three of the variables that are set (`TUXDIR`, `APPDIR`, and `PATH`) are needed whenever you are working with the BEA TUXEDO system. Notice that if you are running on a SUN machine, there is another `bin` you must have at the beginning of your `PATH` variable. `LD_LIBRARY_PATH` is important if you are building the system with shared libraries. `TUXCONFIG` must be set before you can boot the system. `QMADMIN` can be set in a variable or provided on the `qmadmin`(1) command line.

Points to consider: should you plan to have such a file where you will be doing your BEA TUXEDO System/Q work? Should you have a command in your `.profile` so that you set your environment as you log in?

makefile: Make Your Application

Notice that the `makefile` uses `buildserver`(1) and `buildclient`(1) to build the server and client, respectively. You can, of course, execute these commands individually or use the capability of `make` to keep the application current.

While we are on the subject of the `makefile`, this might be a good time to look through the `.c` files for the client and server programs. Of particular interest in connection with BEA TUXEDO System/Q are the `tpenqueue` and `tpdequeue` calls. Notice particularly the values for the *`qspace`* and the *`qname`* arguments. When we look at the configuration file, we will see where those values come from.

ubb.sample: The ASCII Configuration File

The three most pertinent entries in the configuration file are the CLOPT parameters for the TMQUEUE and TMQFORWARD servers and the OPENINFO parameter in the *GROUPS entry. We will extract those items to call them to your attention here:

```
# First the CLOPT parameter from TMQUEUE:
        CLOPT = "-s QSPACENAME:TMQUEUE --  "
# Then the CLOPT parameter from TMQFORWARD:
        CLOPT="-- -i 2 -q STRING"
# Finally, the OPENINFO parameter from the QUE1 group:
        OPENINFO = "TUXEDO/QM:<APPDIR pathname>/QUE:QSPACE"
```

The CLOPT parameter from TMQUEUE specifies a service alias of QSPACENAME. Look back again at client.c and check the *qspace* argument of tpenqueue and tpdequeue. The CLOPT parameter for TMQFORWARD specifies a service STRING by means of the -q option. This is also the name given to the queue where messages are enqueued for that service and is specified as the *qname* argument of tpenqueue in client.c.

The tmloadcf(1) command is used to compile the ASCII configuration file into a TUXCONFIG file.

crlog: Create the Transaction Log

The script in crlog invokes tmadmin(1) to create a device list entry for the TLOG and then create the log for the site specified in our configuration. Because all messages for the queued message facility are enqueued and dequeued within transactions, you must have a log in which to keep track of transactions managed by the TMS_QM server.

crque: Create the Queue Space and Queues

The script in crque invokes qmadmin(1) to create the queue space and queues for the sample application. Notice that the queue space is named QSPACE (that is also the name specified as the last argument of the OPENINFO parameter in the configuration file). Queues named STRING and RPLYQ are created. In the qspacecreate portion of the script an error queue is named, but the script does not include any qcreate command to create that queue. That is a modification you might want to make later.

Boot, Run, and Shut Down the Application

After making the application programs, loading `TUXCONFIG`, and creating the queue space and queues, the next step is to boot the application and run it. The command to boot is

```
tmboot -y
```

The `-y` option keeps `tmboot` from prompting for an okay before booting.

The sample application is run simply by entering the command:

```
client
```

The `tmshutdown` command is used to bring the application down.

Clean Up

The `runsample` script includes three commands that restore the environment to the state it was in before the script was run. The `make clean` command uses `make` to remove the object and executable files for the client and server.

The `rmipc` command is included because the IPC resources for the queue space are not automatically removed by `tmshutdown` (which does remove the BEA TUXEDO IPC resources used by the application). If you look at `rmipc` you will find that it invokes `qmadmin` and uses its version of the `ipcrm` command, naming `QSPACE` to identify resources to be removed.

The final command in the script is the `rm` command, which removes a number of files that are generated by the application. There is no harm in leaving these files; in fact, as you work more with the sample application you will probably want to keep `tuxconfig`, `QUE`, and `TLOG` to save having to recreate them.