BEA Tuxedo®

Using the ATMI /Q Component

Version 9.0
Document Released: June 28, 2005

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Contents

About This Document

BEA Tuxedo /Q Overview

BEA Tuxedo /Q Administration

BEA Tuxedo /Q C Language Programming

BEA Tuxedo /Q COBOL Language Programming

A Sample Application

About This Document

This document explains how to configure, program, and use the /Q component in the BEA Tuxedo environment. The BEA Tuxedo /Q component allows messages to be queued to persistent (disk) or non-persistent storage (memory) for later processing and retrieval.

This document covers the following topics:

- Chapter 1, "BEA Tuxedo /Q Overview," provides an overview of the BEA Tuxedo /Q component architecture and describes administrative and programming tasks required for use.
- Chapter 2, "BEA Tuxedo /Q Administration," provides instructions on how to configure and manage the BEA Tuxedo /Q component.
- Chapter 3, "BEA Tuxedo /Q C Language Programming," provides instructions on using C language functions to enqueue and dequeue messages.
- Chapter 4, "BEA Tuxedo /Q COBOL Language Programming," provides instructions on using COBOL language functions to enqueue and dequeue messages.
- Appendix A, "A Sample Application," provides a sample client-server application using the BEA Tuxedo /Q component.

What You Need to Know

This document is intended for the following audiences:

- administrators who are interested in configuring and managing message queueing applications in a BEA Tuxedo environment

- application developers who are interested in programming message queueing applications in a BEA Tuxedo environment

This document assumes a familiarity with the BEA Tuxedo platform and either C or COBOL programming.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation or go directly to the "e-docs" Product Documentation page at http://e-docs.bea.com.

How to Print the Document

You can print a copy of this document from a Web browser, one file at a time, by using the File—>Print option on your Web browser.

A PDF version of this document is available on the BEA Tuxedo documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the BEA Tuxedo documentation Home page, click the PDF files button and select the document you want to print.

If you do not have the Adobe Acrobat Reader, you can get it for free from the Adobe Web site at http://www.adobe.com/.

Related Information

The following BEA Tuxedo documents contain information that is relevant to using the BEA Tuxedo /Q component and understanding how to implement message queueing applications in the BEA Tuxedo environment:

- `TMQUEUE`(5) and `TMQFORWARD`(5) in *File Formats, Data Descriptions, MIBs, and System Processes Reference*

Contact Us!

Your feedback on the BEA Tuxedo documentation is important to us. Send us e-mail at **docsupport@bea.com** if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the BEA Tuxedo documentation.

In your e-mail message, please indicate that you are using the documentation for the BEA Tuxedo 9.0 release.

If you have any questions about this version of BEA Tuxedo, or if you have problems installing and running BEA Tuxedo, contact BEA Customer Support through BEA WebSupport at **www.bea.com**. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
boldface text	Indicates terms defined in the glossary.
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.
italics	Indicates emphasis or book titles.

Convention	Item

`monospace text`	Indicates code samples, commands and their options, data structures and their members, data types, directories, and file names and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Example*s: LPT1 SIGNON OR
`{ }`	Indicates a set of choices in a syntax line. The braces themselves should never be typed.

`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed.
	Example:
	`buildobjclient [-v] [-o name ] [-f` *`file-list`*`]...`
	`[-l` *`file-list`*`]...`

|	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.

Convention	Item

. . .	Indicates one of the following in a command line:
	• That an argument can be repeated several times in a command line
	• That the statement omits additional optional arguments
	• That you can enter additional parameters, values, or other information
	The ellipsis itself should never be typed.
	Example:
	`buildobjclient [-v] [-o name ] [-f file-list]...`
	`[-l file-list]...`

. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

CHAPTER 1

BEA Tuxedo /Q Overview

This topic includes the following sections:

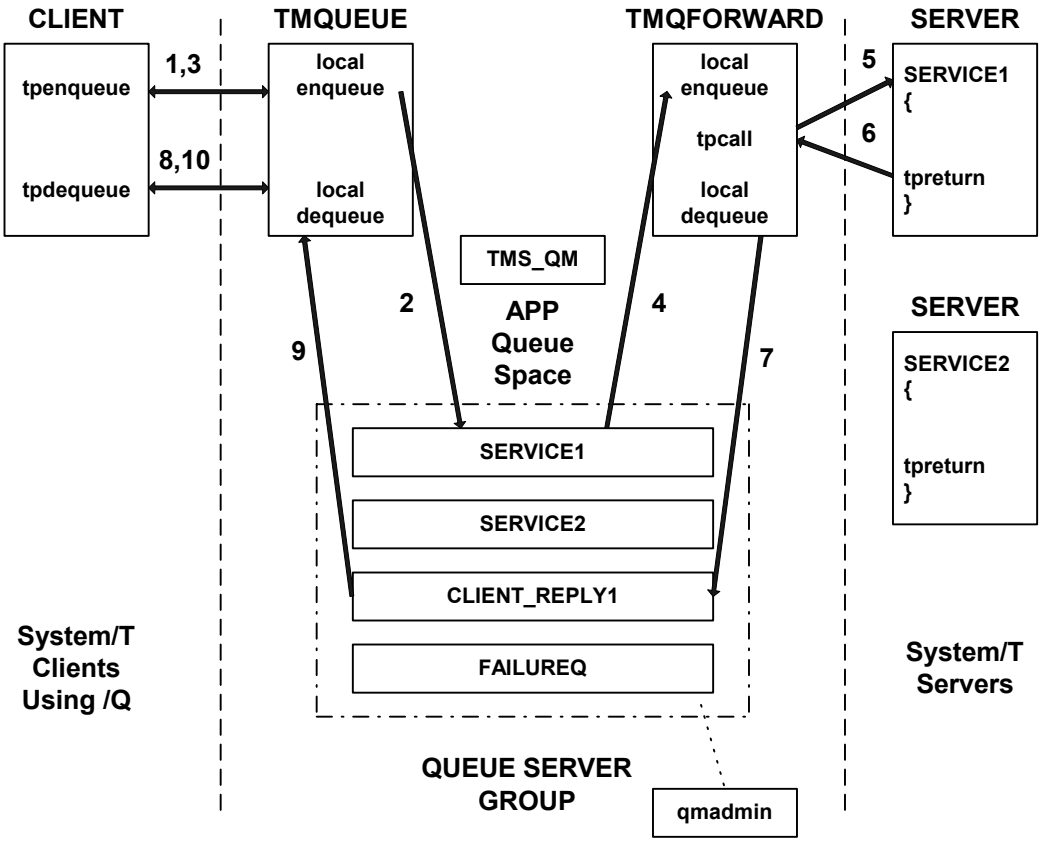
General Description

The BEA Tuxedo /Q component allows messages to be queued to persistent storage (disk) or to non-persistent storage (memory) for later processing or retrieval. The BEA Tuxedo Application-to-Transaction Monitor Interface (ATMI) provides functions that allow messages to be added to or read from 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. Servers are provided 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.

Queuing System Components and Tasks

The following figure shows the components of the queued message facility.

Figure 1-1 Queued Service Invocation



The figure illustrates how each component of the queuing system operates for queued service invocation. In this discussion, we use the figure to explain how administrators and programmers work with the BEA Tuxedo /Q component to define it and use it to queue a message for processing and get back a reply. The queuing service may also be used for simple peer-to-peer communication by using a subset of the components shown in the figure.

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 two-phase committed transaction in coordination with other XA-compliant resource managers.

Administrator Tasks

The BEA Tuxedo administrator is responsible for defining servers and creating queue spaces and queues like those shown between the vertical dashed lines in the figure "Queued Service Invocation" on page 1-2.

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 /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)`, or the `APPQ_MIB(5)` Management Information Base (MIB). The queue space contains a collection of queues. In the figure "Queued Service Invocation" on page 1-2, 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 (RM) 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 persistent and non-persistent storage areas for messages.

- A single message queue server, TMQUEUE in the figure "Queued Service Invocation" on page 1-2, can be used to enqueue and dequeue messages for multiple queues within a single queue space.
- A single message forwarding server, TMQFORWARD in the figure "Queued Service Invocation" on page 1-2, can be used to dequeue and forward messages to services from multiple queues within a single queue space.
- Two instances of the transaction manager server, TMS_QM in the figure "Queued Service Invocation" on page 1-2, can be used to complete transactions for multiple queues within a single queue space. One instance of the transaction manager server is reserved for non-blocking transactions so that they will be processed as quickly as possible and not be held up by blocking transactions. Blocking transactions are handled by the second instance of the transaction manager server.

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) (see tmconfig, wtmconfig(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 determined by message availability time, expiration time, priority, FIFO, LIFO, or a combination of these criteria.

The administrator specifies one or more of these sort criteria for the queue, listing the most significant criteria first. The FIFO and LIFO values must 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.

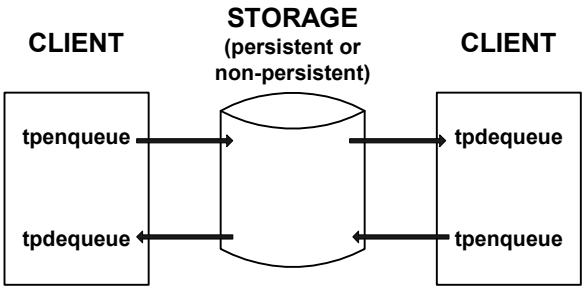
For housekeeping purposes, the administrator can set up a command to be executed when a threshold is reached for a queue that does not routinely get drained. This can be based on the bytes, blocks, or percentage of the queue space used by the queue or the number of messages on the queue. The command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

The BEA Tuxedo system uses the Queueing Services component of the BEA Tuxedo infrastructure for some operations. (The BEA Tuxedo infrastructure provides services such as

security, scalability, message queuing, and transactions.) For example, administrative operations for shared memory are provided by the Queuing Services component. Some functions are not currently applicable to BEA Tuxedo applications; this is noted in descriptions of these functions.

You can also use the queued message facility for peer-to-peer communication between clients, such that a client communicates with other clients without using any forwarding server. The peer-to-peer communication model is shown in the following figure.

Figure 1-2 Peer-to-Peer Communication



Programmer Tasks

In steps 1 through 3 of the figure "Queued Service Invocation" on page 1-2, 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 made available for dequeuing), 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, 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 using BEA Tuxedo /Q for queued service invocation, and 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 the figure "Queued Service Invocation" on page 1-2, 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 "Queued Service Invocation" on page 1-2).

Sometime later (steps 8, 9 and 10 in the figure "Queued Service Invocation" on page 1-2), the client uses tpdequeue(3c) to read from the reply queue CLIENT_REPLY1 in order to get the reply message.

You can dequeue messages without removing them from the queue by using the TPQPEEK flag with tpdequeue(). Messages that have expired or have been deleted by an administrator are immediately removed from the queue.

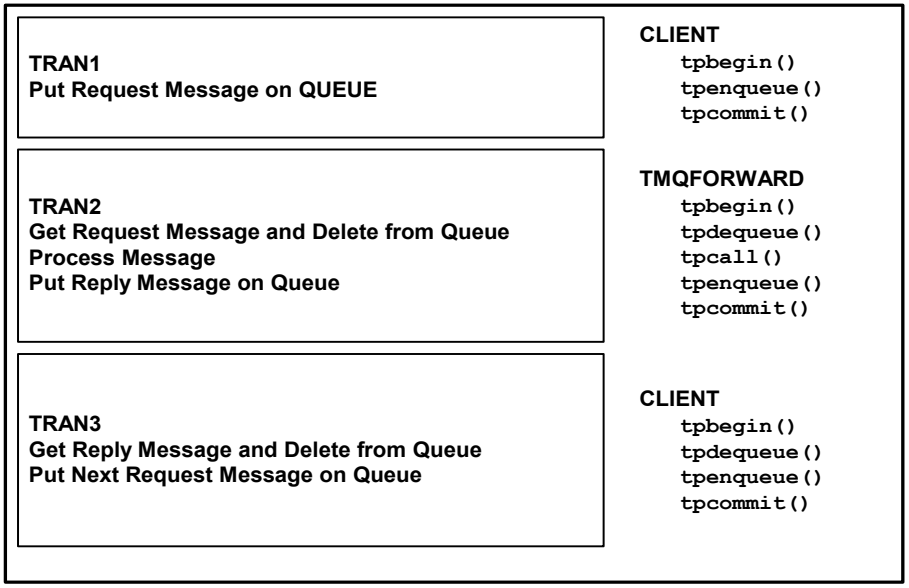
Transaction Management

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 outside 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.

A better approach is to enqueue the message within the caller's transaction, as is shown in the following figure.

Figure 1-3 Transaction Demarcation



In the figure, 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(3c)`, 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 automatically processed by a service (dequeued and passed to the application via a `tpcall`()) by `TMQFORWARD`, `TMQFORWARD` enqueues 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 /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 "BEA Tuxedo /Q Administration" on page 2-1, "BEA Tuxedo /Q C Language Programming" on page 3-1, and "BEA Tuxedo /Q COBOL Language Programming" on page 4-1.

Summary

To summarize, BEA Tuxedo /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 or correlation identifier.
- The application program and/or the administrator can control the ordering of messages on the queue. Control is via the sort criteria, which may be based on message availability time, expiration time, priority, `LIFO`, `FIFO`, or a combination of these 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 to services 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 or wireless networks). 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. This facility may also be used to provide a data pipe between two otherwise unrelated applications in a peer-to-peer relationship.

CHAPTER 2

BEA Tuxedo /Q Administration

This topic includes the following sections:

Introduction

The BEA Tuxedo /Q administrator has three primary areas of responsibility, which are:

- 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.

Available Sample Program Called qsample

A brief example of the use of the queued message facility is distributed with the software and is described in "A Sample Application" on page A-1.

Configuration

Three servers are provided with the BEA Tuxedo /Q component. One is the Transaction Manager Server (TMS), TMS_QM, for the BEA Tuxedo /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.

The application can also choose peer-to-peer communication. In this case, the application communicates with other applications, or a client communicates with other clients, by not using any forwarding server.

Specifying the QM Server Group

In addition to the standard requirements of a group name tag and a value for GRPNO (see UBBCONFIG(5) for details), there must be a server group defined for each queue space the application will use. 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 BEA Tuxedo /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 reference 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) reference page gives a full description of the SERVERS section of the configuration file, but there are some points worth additional emphasis here.

Operation Timeout

TMQUEUE recognizes a -t *timeout* option when specified after the double dash (- -) in the CLOPT parameter. This timeout value affects only operations begun within the server if it finds that a transaction is not 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() or tpdequeue() with the TPNOTRAN flag set to exclude the queue request from the client's transaction. The default for *timeout* 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 *timeout* 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 "TPQCTL Structure" on page 3-5 and "TPQUEDEF-REC Structure" on page 4-6. The COBOL equivalent is the TPQUEDEF-REC record.

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 preceding discussion 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. The following example is taken from the TMQUEUE(5) reference 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 ATMI 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).

Buffer Subtypes

You can assign a subtype to a buffer using the tpalloc(3c) subtype parameter and later extract the subtype using the tptypes(3c) function. This gives you the ability to assign a type to data without having to create an entirely new user-defined ATMI buffer type. This is especially useful for buffers containing character arrays (CARRAY) or strings (STRING).

Specifying the Message Forwarding Server

The third system-supplied server included with the BEA Tuxedo /Q component 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 TMQFORWARD(5) reference page, but the topics 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 incorrect 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 topics 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 periodically 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 of CPU resources 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 (and will create a user log message) 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 about the -e option and the threshold command in "Creating Queue Spaces and Queues" on page 2-8.

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 message is deleted. Also with the -d option, if the original request message is to be deleted at the same time as the retry limit configured for the queue is reached, the original request message is put into the error queue.

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) (see tmconfig, wtmconfig(1)). Of course, the group and the servers will have to be booted once they have been defined.

Creating Queue Spaces and Queues

This topic covers three of the qmadmin(1) commands that are used to establish the resources of the BEA Tuxedo /Q component. The APPQ_MIB Management Information Base provides an alternative method of administering BEA Tuxedo /Q programmatically. See the APPQ_MIB(5) reference page for more information on the MIB.

Working with qmadmin Commands

Most of the key commands of qmadmin have positional parameters. If the positional parameters (those not specified with a dash (-) preceding the option) are not specified on the command line when the command is invoked, qmadmin prompts you for the required information.

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 the queued message facility is installed as part of a new BEA Tuxedo installation, 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. The following 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 must be the same pathname specified in TUXCONFIG. Once you have invoked qmadmin, it is recommend that 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. (You can also use the T_APPQSPACE class of the APPQ_MIB(5) to create a queue space.) 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, n [default=n]):
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 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 permanently lost.

The program does not prompt you to specify the size of the area to reserve in shared memory for storing non-persistent messages for all queues in the queue space. When you require non-persistent (memory-based) messages, you must specify the size of the memory area on the qspacecreate command line with the -n option.

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.

To calculate the number of concurrent transactions, count each of the following as one transaction:

- Each `TMS_QM` server in the group that uses this queue space
- Each `TMQUEUE` or `TMQFORWARD` server in the group that uses this queue space
- `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. The worst case is that all clients access the queue space at the same time.

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. (You can also use the `T_APPQ` class of the `APPQ_MIB(5)` to create a queue.)

The prompt sequence for `qcreate` looks like the following:

```
> qcreate
Queue name: service1
Queue order (priority, time, fifo, lifo): fifo
Out-of-ordering enqueuing (top, msgid, [default=none]): none
Retries [default=0]: 2
```

```
Retry delay in seconds [default=0]: 30
High limit for queue capacity warning (b for bytes used, B for blocks used,
% for percent used, m for messages [default=100%]): 80%
Reset (low) limit for queue capacity warning [default=0%]: 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.

The program does not prompt you for a default delivery policy and memory threshold options. The default delivery policy option allows you to specify whether messages with no specified delivery mode are delivered to persistent (disk-based) or non-persistent (memory-based) storage. The memory threshold option allows you to specify values used to trigger command execution when a non-persistent memory threshold is reached. To use these options, you must specify them on the `qcreate` command line with `-d` and `-n`, respectively.

Specifying Queue Order

Messages are put into the queue based on the order specified by this parameter and dequeued from the top of the queue unless selection criteria are applied to the dequeuing operation. If `priority`, `expiration`, and/or `time` are chosen as queue order criteria, then messages are inserted into the queue according to values in the `TPQCTL` structure. A combination of sort criteria may be specified with the most significant criteria specified first. Separate multiple criteria with commas (`,`). If `fifo` or `lifo` (which are mutually exclusive) are specified, they must be the last value specified. The sequence in which parameters are specified 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 specified 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. During this time, the message is unavailable for any dequeuing operation.

The default for the number of retries is 0, which means that no retries are attempted. When the retry limit is reached, the system moves the message to the error queue for the 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 CPU 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 bytes, blocks, messages or percent of queue capacity) and allow you to specify 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 until the low threshold is reached first prior to the high threshold.)

The following 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 [default=100%]): 80%
Reset (low) limit for queue capacity warning [default=0%]: 10%
Queue capacity command: /usr/app/bin/mailme myqueuespace service1
```

This sequence sets the upper threshold at 80% of disk-based 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 do not get the warning message again unless the queue load drops to 10% of capacity or below, and then rises again to 80%. You can also set thresholds and specify commands for the management of non-persistent (memory-based) queue capacity using the `-n` option of the `qcreate` command.

Note: If you are working on a Windows machine, see "Windows Standard I/O" on page 2-18 for additional information about configuring commands within a `qmadmin`() session.

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 [default=100%]): 90%
Reset (low) limit for queue capacity warning [default=0%]: 0%
Queue capacity command: tmboot -i 1002
```

This sequence assumes that you have configured a reserve `TMQFORWARD` server for the queue in question with `SRVID=1002` 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 to be processed by a service of the same name. A queue may also be used for other purposes as well, such as peer-to-peer communication. The parameters for creating a queue are the same regardless of its use. The `TPQCTL` structure used when a message is enqueued to a service queue includes fields to specify a reply queue and a failure queue. `TMQFORWARD` detects the success or failure of the `tpacall(3c)` it makes to the requested service and, if these queues have been created by the administrator, enqueues 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 of the outcome.

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. In this case, fifo is probably sufficient. 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, and the recommended usage is to alert the administrator so that he or she can intervene.

Error Queues

An error queue is a system queue. One of the qspacecreate prompts asks for the name of the error queue for the queue space. When you have actually created an error queue of the name specified, 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 handle the messages manually through commands of qmadmin or can set up an automated way of handling them through the APPQ_MIB MIB. The queue capacity parameters can be used, but all of the other qcreate parameters, with the exception of qname, are not useful for the error queue.

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

Handling Encrypted Message Buffers

In general, TMQUEUE and TMQFORWARD handle *encrypted* message buffers without decrypting them. However, there are situations where the /Q component needs to decrypt enqueued message buffers, as described in "Compatibility/Interaction with /Q" on page 1-58 in *Using Security in ATMI Applications*.

As mentioned in the "Compatibility/Interaction with /Q" discussion, a non-transactional tpdequeue() operation has the side effect of destroying an encrypted queued message if the invoking process does not hold a valid decryption key. Thus, application programmers need to open a decryption key for a process before the process calls tpdequeue() to retrieve an encrypted message; otherwise, the message will be lost.

For information on opening a decryption key, see "Initializing Decryption Keys Through the Plug-ins" on page 2-50 and "Writing Code to Receive Encrypted Messages" on page 3-45 in *Using Security in ATMI Applications*.

Maintenance of the BEA Tuxedo /Q Feature

This topic 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)`. (You can also use the `TA_MAXPAGES` attribute of the `T_APPQSPACE` class of `APPQ_MIB(5)` to add extents.) The `qmadmin` 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` closes it for you if you try to add extents to it while it is open. All non-persistent messages currently in the queue space are lost when the `qaddext` command is issued and completes successfully.

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 should 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.

Similar archival techniques are available on server platforms that do not have the `dd` command. First, shut down the group containing the queue space you want to back up or migrate. Then, use an archival tool to save the queue space device to a file or other medium that may then be used as a backup or used to move the queue space to another server.

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 include TPQWAIT, the TMQUEUE server will wait for a message to arrive on a queue before returning to the caller. The number of seconds before it times out is based on the following:

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

If a message is not immediately available when using TPQWAIT, TMQUEUE requires an action resource so that TMQUEUE may service other requests. You may want to increase the number of actions the queue space may handle concurrently. Use the -A *actions* option to the qspacecreate or qspacechange commands. This option specifies the number of additional actions that can be handled concurrently. When a waiting operation is encountered and additional actions are available, the blocking operation is set aside until it can be satisfied. If no actions are available, the call to tpdequeue fails.

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 handled by the administrator moving the message back to the service queue when the service is known to be available, then the queue name may be stored as part of the corrid in the TPQCTL structure so the queue name is associated with the message.

Windows Standard I/O

In order to carry out a command that you have configured within a `qmadmin`() session, such as the `qchange ... Queue capacity command` described in "Using Queue Capacity Limits" on page 2-12, the Windows `CreateProcess`() function spawns a child process as a `DETACHED PROCESS`. This type of process does *not* have an associated console for standard input/output. Therefore, for instance, if you use standard DOS syntax to set the `qchange ... Queue capacity command` to run a built-in DOS command (such as `dir` or `date`) and then pipe or redirect the standard output to a file, the file will be empty when the command completes.

As an example of resolving this problem, suppose that for the `qchange ... Queue capacity command` you want to capture `date` information in a file using command `date /t > x.out`. To accomplish this task interactively, you would proceed as follows:

```
qmadmin
> qopen yourQspace
> qchange yourQname
> go through all the setups... the threshold queue capacity warning,
   and so on
> "Queue capacity command: " cmd /c date /t > x.out
```

To accomplish this task from a command file, say *`yourFile`*`.cmd`, you would add the command `date /t > x.out` to *`yourFile`*`.cmd` and then proceed as follows:

```
qmadmin
> qopen yourQspace
> qchange yourQname
> go through all the setups... the threshold queue capacity warning,
   and so on
> "Queue capacity command: " yourFile.cmd
```

BEA Tuxedo /Q C Language Programming

This topic includes the following sections:

Introduction

This topic 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 *Programming BEA Tuxedo ATMI Applications Using C*. Detailed pages on all the ATMI functions are in the *BEA Tuxedo ATMI C Function Reference*.

Where Requests Can Originate

The calls used to place a message on a BEA Tuxedo /Q queue can originate from 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 /Q programming in this topic primarily reflects the left-hand portion of the figure "Queued Service Invocation" on page 1-2. In the 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 made available through a `TMQUEUE(5)` server. The client later retrieves a reply via a `tpdequeue(3c)` call to `TMQUEUE`.

The figure "Queued Service Invocation" on page 1-2 shows the queued message being dequeued by the server `TMQFORWARD(5)` and sent to an application server for processing (via `tpcall(3c)`). When a reply to the `tpcall`() is received, `TMQFORWARD` enqueues the reply message. Because 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 topic concentrates on the client-to-queue space side.

A brief example of the use of the queued message facility is distributed with the software and is described in "A Sample Application" on page A-1.

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*.

tpenqueue(3c) Arguments

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 might 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 then looks like this:

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

The example shown on the TMQUEUE(5) reference page shows how alias service names can be included when the server is built and specified in the configuration file. The sample program in "A Sample Application" on page A-1, also specifies an alias service name.

tpenqueue(): The qname Argument

Within a queue space, when queues are being used to invoke services, message queues are named according to the application services available to process requests. *qname* is a pointer to such an application service. Otherwise, qname is simply the name of the location where the message is to be stored until it is dequeued by an application (either the same application that enqueued it or another one).

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 that the length of the message be specified; in such cases, the *len* 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 and this flag is set, the message is not queued within the caller's transaction. A caller in transaction mode that sets this flag is still subject to the transaction timeout (and no other) when queuing the message. If message queuing fails, the caller's transaction is not affected.

TPNOBLOCK
: The message is not enqueued if a blocking condition exists. If this flag is set and a blocking condition exists such as the internal buffers into which the message is transferred are full, the call fails and tperrno(5) is set to TPEBLOCK. If this flag is set and a blocking condition exists because the target queue is opened *exclusively* by another application, the call fails, tperrno() is set to TPEDIAGNOSTIC, and the diagnostic field of the TPQCTL structure is set to QMESHARE. In the latter case, the other application, which is based on a BEA product other than the BEA Tuxedo system, opened the queue for exclusive read and/or write using the Queuing Services API (QSAPI).

  When TPNOBLOCK is not set and a blocking condition exists, the caller blocks until the condition subsides or a timeout occurs (either transaction or blocking timeout). If a timeout occurs, the call fails and tperrno() is set to TPETIME.

TPNOTIME
: Setting this flag signifies that the caller is willing to block indefinitely and wants to be immune to blocking timeouts. Transaction timeouts may still occur.

TPSIGRSTRT
: Setting this flag indicates that any underlying system calls that are interrupted by a signal should be reissued. When this flag is not set and a signal interrupts a system call, the call fails and sets tperrno(5) to TPGOTSIG.

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 fields the application wants the system to fill in. The structure is also used by tpdequeue(); some of the fields do not come into play until the application calls that function. The complete structure is shown in the following listing.

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 */
     long delivery_qos;      /* delivery quality of service */
     long reply_qos;         /* reply message quality of service */
     long exp_time;          /* expiration time */
};
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 control structure. Leaving fields of the structure not set is equivalent to a setting of TPNOFLAGS.

TPQTOP
: Setting this flag 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. TPQTOP and TPQBEFOREMSGID are mutually exclusive flags

TPQBEFOREMSGID
: Setting this flag indicates that the queue ordering be overridden and the message placed in the queue before the message identified by *ctl->msgid*. This request may not be granted depending on whether or not the queue was configured to allow overriding the queue ordering. TPQTOP and TPQBEFOREMSGID are mutually exclusive flags. Note that the entire 32 bytes of the message identifier value are significant, so the value identified by *ctl->msgid* must be completely initialized (for example, padded with NULL characters).

TPQTIME_ABS
: If this flag is set, the message is made available after the time specified by *ctl->deq_time*. The *deq_time* is an absolute time value as generated by time(2) or mktime(3C), if they are available to your application, or gp_mktime(3c), provided with the BEA Tuxedo system. The value set in *ctl->deq_time* is the number of seconds since 00:00:00 Universal Coordinated Time—UTC, January 1,1970. The absolute time is set based on the clock on the machine where the queue manager process resides. TPQTIME_ABS and TPQTIME_REL are mutually exclusive flags.

TPQTIME_REL
: If this flag is set, the message is made available after a time relative to the completion of the enqueuing operation. *ctl->deq_time* specifies the number of seconds to delay after the enqueuing completes before the submitted message should be available. TPQTIME_ABS and TPQTIME_REL are mutually exclusive flags.

TPQPRIORITY
: If this flag is set, the priority at which the request should be enqueued is stored in *ctl->priority*. The priority must be in the range 1 to 100, inclusive. The higher the number, the higher the priority, that is, a message with a higher number is dequeued before a message with a lower number from queues ordered by priority. For queues not ordered by priority, the value is informational.

If this flag is not set, the priority for the message is 50 by default.

TPQCORRID
: If this flag is set, the correlation identifier value specified in *ctl->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. Note that the entire 32 bytes of the correlation identifier value are significant, so the value specified in *ctl->corrid* must be completely initialized (for example, padded with NULL characters).

TPQREPLYQ
: If this flag is set, a reply queue named in *ctl->replyqueue* is associated with the queued message. Any reply to the message is 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 this flag is set, a failure queue named in the *ctl->failurequeue* is associated with the queued message. If (1) the enqueued message is processed by TMQFORWARD(), (2) TMQFORWARD was started with the -d option, and (3) the service fails and returns a non-NULL reply, a failure message consisting of the reply and its associated tpurcode is enqueued 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).

TPQDELIVERYQOS**,** TPQREPLYQOS
: If the TPQDELIVERYQOS flag is set, the flags specified by *ctl->delivery_qos* control the quality of service for delivery of the message. In this case, one of three mutually exclusive flags— TPQQOSDEFAULTPERSIST, TPQQOSPERSISTENT, or TPQQOSNONPERSISTENT—must be set in *ctl->delivery_qos*. If TPQDELIVERYQOS is not set, the default delivery policy of the target queue dictates the delivery quality of service for the message.

  If the TPQREPLYQOS flag is set, the flags specified by *ctl->reply_qos* control the quality of service for any reply to the message. In this case, one of three mutually exclusive flags—TPQQOSDEFAULTPERSIST, TPQQOSPERSISTENT, or TPQQOSNONPERSISTENT—must be set in *ctl->reply_qos*. The TPQREPLYQOS flag is used when a reply is returned from messages processed by TMQFORWARD. Applications not using TMQFORWARD to invoke services may use the TPQREPLYQOS flag as a hint for their own reply mechanism.

  If TPQREPLYQOS is not set, the default delivery policy of the *ctl->replyqueue* queue dictates the delivery quality of service for any reply. Note that the default delivery policy is determined when the reply to a message is enqueued. That is, if the default delivery

policy of the reply queue is modified between the time that the original message is enqueued and the reply to the message is enqueued, the policy used is the one in effect when the reply is finally enqueued.

The following is the list of valid flags for *ctl->delivery_qos* and *ctl->reply_qos*:

TPQQOSDEFAULTPERSIST
: This flag specifies that the message is to be delivered using the default delivery policy specified on the target queue.

TPQQOSPERSISTENT
: This flag specifies that the message is to be delivered in a persistent manner using the disk-based delivery method. When specified, this flag overrides the default delivery policy specified on the target queue.

TPQQOSNONPERSISTENT
: This flag specifies that the message is to be delivered in a non-persistent manner using the memory-based delivery method. Specifically, the message is queued in memory until it is dequeued. When specified, this flag overrides the default delivery policy specified on the target queue. If the caller is transactional, non-persistent messages are enqueued within the caller's transaction, however, non-persistent messages are lost if the system is shut down, crashes, or the IPC shared memory for the queue space is removed.

TPQEXPTIME_ABS
: If this flag is set, the message has an absolute expiration time, which is the absolute time when the message will be removed from the queue.
The absolute expiration time is determined by the clock on the machine where the queue manager process resides.

The absolute expiration time is indicated by the value stored in *ctl->exp_time*. The value of *ctl->exp_time* must be set to an absolute time value generated by `time(2)`, `mktime(3C)`, or `gp_mktime(3c)` (the number of seconds since 00:00:00 Universal Coordinated Time—UTC, January 1, 1970).

If an absolute time is specified that is earlier than the time of the enqueue operation, the operation succeeds, but the message is not counted for the purpose of calculating thresholds. If the expiration time is before the message availability time, the message is not available for dequeuing unless either the availability or expiration time is changed so that the availability time is before the expiration time. In addition, these messages are removed from the queue at expiration time even if they were never available for dequeuing. If a message expires while it is within a transaction, the expiration does not cause the transaction to fail. Messages that expire while being enqueued or dequeued

within a transaction are removed from the queue when the transaction ends. There is no notification that the message has expired.

TPQEXPTIME_ABS, TPQEXPTIME_REL, and TPQEXPTIME_NONE are mutually exclusive flags. If none of these flags is set, the default expiration time associated with the target queue is applied to the message.

TPQEXPTIME_REL

If this flag is set, the message has a relative expiration time, which is the number of seconds *after* the message arrives at the queue that the message is removed from the queue. The relative expiration time is indicated by the value stored in *ctl->exp_time*.

If the expiration time is before the message availability time, the message is not available for dequeuing unless either the availability or expiration time is changed so that the availability time is before the expiration time. In addition, these messages are removed from the queue at expiration time even if they were never available for dequeuing. The expiration of a message during a transaction, does not cause the transaction to fail. Messages that expire while being enqueued or dequeued within a transaction are removed from the queue when the transaction ends. There is no acknowledgment that the message has expired.

TPQEXPTIME_ABS, TPQEXPTIME_REL, and TPQEXPTIME_NONE are mutually exclusive flags. If none of these flags is set, the default expiration time associated with the target queue is applied to the message.

TPQEXPTIME_NONE

Setting this flag indicates that the message should not expire, even if the default policy of the queue includes an expiration time.

TPQEXPTIME_ABS, TPQEXPTIME_REL, and TPQEXPTIME_NONE are mutually exclusive flags. If none of these flags is set, the default expiration time associated with the target queue is applied to the message.

Additionally, the *urcode* field 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 fields 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 */
```

The following is a valid bit for the *flags* parameter controlling output information from tpenqueue(). If this flag is turned on when tpenqueue() is called, the /Q server TMQUEUE(5) populates the associated element in the structure with a message identifier. If this flag is turned

off when tpenqueue() is called, TMQUEUE() does *not* populate the associated element in the structure with a message identifier.

TPQMSGID
: If this flag is set and the call to tpenqueue() is successful, the message identifier is stored in *ctl->msgid*. The entire 32 bytes of the message identifier value are significant, so the value stored in *ctl->msgid* is completely initialized (for example, padded with null characters). The actual padding character used for initialization varies between releases of the BEA Tuxedo /Q component.

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

If the call to tpenqueue() fails and tperrno(5) is set to TPEDIAGNOSTIC, a value indicating the reason for failure is returned in *ctl->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 in transaction mode or was made with the TPNOTRAN flag set and an error occurred trying to start a transaction in which to enqueue the message. This diagnostic is not returned by queue managers from BEA Tuxedo release 7.1 or later.

[QMEBADMSGID]
: An invalid message identifier was specified.

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

[QMEOS]
: An operating system error occurred.

[QMEABORTED]
: The operation was aborted. If the aborted operation was being executed within a global transaction, the global transaction is marked rollback-only. Otherwise, the queue manager aborts the operation.

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

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

[QMENOSPACE]
Due to an insufficient resource, such as no space on the queue, the message with its required quality of service (persistent or non-persistent storage) was not enqueued. QMENOSPACE is returned when any of the following configured resources is exceeded: (1) the amount of disk (persistent) space allotted to the queue space, (2) the amount of memory (non-persistent) space allotted to the queue space, (3) the maximum number of simultaneously active transactions allowed for the queue space, (4) the maximum number of messages that the queue space can contain at any one time, (5) the maximum number of concurrent actions that the Queuing Services component can handle, or (6) the maximum number of authenticated users that may concurrently use the Queuing Services component.

[QMERELEASE]
An attempt was made to enqueue a message to a queue manager that is from a version of the BEA Tuxedo system that does not support a newer feature.

[QMESHARE]
When enqueuing a message from a specified queue, the specified queue is opened *exclusively* by another application. The other application is one based on a BEA product other than the BEA Tuxedo system that opened the queue for exclusive read and/or write using the Queuing Services API (QSAPI).

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 include TPQTOP or you can specify that it be placed ahead of a specific message by setting *flags* to include TPQBEFOREMSGID and setting *ctl->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

You can set a value in *ctl->priority* to specify the priority of 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 on the dequeuing order,

however the priority value is retained so that the value can be inspected when the message is dequeued.

Setting a Message Availability Time

You can specify in *deq_time* either an absolute time or a time relative to the completion of the enqueuing operation for the message to be made available. You set *flags* to include either TPQTIME_ABS or TPQTIME_REL to indicate how the value should be treated. A queue may be created with time as a queue ordering criterion, in which case the messages are ordered by the message availability time.

BEA Tuxedo /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 time(2) and mktime(3C) functions may also be used instead of gp_mktime(3c). 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, 2001), 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 = 2001 - 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 that 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 operation, place the number of seconds in `deq_time` and set `flags` to include `TPQTIME_REL`.

tpenqueue() and Transactions

If a 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 a caller of `tpenqueue`() is not in transaction mode or if `TPNOTRAN` is set, the message is enqueued outside of the caller's 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 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 Messages

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 the queue 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.

tpdequeue(3c) Arguments

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 the TMQUEUE 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 may 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 then looks like this:

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

The example shown on the TMQUEUE(5) reference page shows how alias service names can be included when the server is built and specified in the configuration file. The sample program in "A Sample Application" on page A-1, also specifies an alias service/queue space name.

tpdequeue(): The qname Argument

Queue names in a queue space must be agreed upon by the applications that will access the queue space. This is especially important for reply queues. If qname refers to a reply queue, the administrator creates it (and often an error queue) in the same manner that he or she creates any other queue. *qname* is a pointer to the name of the queue from which to retrieve the message or reply.

tpdequeue(): The data and len Arguments

These arguments have slightly different meanings than their counterparts in 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 value of type 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 length prior to the call to tpdequeue() and compare it to *len* after the call completes.

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 and this flag is set, the message is not dequeued within the caller's transaction. A caller in transaction mode that sets this flag is still subject to the transaction timeout (and no other) when dequeuing the message. If message dequeuing fails, the caller's transaction is not affected.

TPNOBLOCK
: The message is not dequeued if a blocking condition exists. If this flag is set and a blocking condition exists such as the internal buffers into which the message is transferred are full, the call fails and tperrno(5) is set to TPEBLOCK. If this flag is set and a blocking condition exists because the target queue is opened *exclusively* by another application, the call fails, tperrno() is set to TPEDIAGNOSTIC, and the diagnostic field of the TPQCTL structure is set to QMESHARE. In the latter case, the other application, which is based on a BEA product other than the BEA Tuxedo system, opened the queue for exclusive read and/or write using the Queuing Services API (QSAPI).

  When TPNOBLOCK is not set and a blocking condition exists, the caller blocks until the condition subsides or a timeout occurs (either transaction or blocking timeout). This blocking condition does not include blocking on the queue itself if the TPQWAIT option in *flags* (of the TPQCTL structure) is specified.

TPNOTIME
: Setting this flag signifies that the caller is willing to block indefinitely and wants to 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

Setting this flag indicates that any underlying system calls that are interrupted by a signal should be reissued. When this flag is not set and a signal interrupts a system call, the call fails and sets tperrno(5) to TPGOTSIG.

TPQCTL Structure

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 fields for which the system should supply values. As described earlier, the structure is also used by tpenqueue(); some of the members apply only to that function. The entire structure is shown in "The tpqctl_t Structure" on page 3-5.

As input to tpdequeue(), the following fields 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 following are valid flags on input to tpdequeue():

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

TPQGETBYMSGID
: Setting this flag requests that the message with the message identifier specified by *ctl->msgid* be dequeued. The message identifier is determined through a prior call to tpenqueue(). Note that the message identifier changes if the message is moved from one queue to another. Note also that the entire 32 bytes of the message identifier value are significant, so the value specified by *ctl->msgid* must be completely initialized (for example, padded with null characters).

TPQGETBYCORRID
: Setting this flag requests that the message with the correlation identifier specified by *ctl->corrid* be dequeued. The correlation identifier is specified by the application when enqueuing the message with tpenqueue(). Note that the entire 32 bytes of the correlation identifier value are significant, so the value specified by *ctl->corrid* must be completely initialized (for example, padded with null characters).

TPQWAIT
: Setting this flag indicates that an error should not be returned if the queue is empty. Instead, the process should wait until a message is available. If TPQWAIT is set in conjunction with TPQGETBYMSGID or TPQGETBYCORRID, it indicates that an error should not be returned if no message with the specified message identifier or correlation identifier is present in the queue. Instead, the process should wait until a message meeting the criteria is available. The process is still subject to the caller's transaction timeout, or, when not in transaction mode, the process is still subject to the timeout specified for the TMQUEUE process by the -t option.

  If a message matching the desired criteria is not immediately available and the configured action resources are exhausted, tpdequeue() returns -1, tperrno() is set to TPEDIAGNOSTIC, and the diagnostic field of the TPQCTL structure is set to QMESYSTEM.

  Note that each tpdequeue() request specifying the TPQWAIT control parameter requires that a queue manager (TMQUEUE) action object be available if a message satisfying the condition is not immediately available. If an action object is not available, the

tpdequeue() request fails. The number of available queue manager actions are specified when a queue space is created or modified. When a waiting dequeue request completes, the associated action object associated is made available for another request.

TPQPEEK
: If this flag is set, the specified message is read but is not removed from the queue. The TPNOTRAN flag must also be set.

  When a thread is non-destructively dequeuing a message using TPQPEEK, the message may not be seen by other non-blocking dequeuers for the brief time the system is processing the non-destructive dequeue request. This includes dequeuers using specific selection criteria (such as message identifier and correlation identifier) that are looking for the message currently being non-destructively dequeued.

The following is a list of valid bits for the *flags* parameter controlling output information from tpdequeue(). For any of these bits, if the flag bit is turned on when tpdequeue() is called, the associated field in the structure (see "The tpqctl_t Structure" on page 3-5) is populated with the value provided when the message was queued, and the bit remains set. If a value is not available (that is, no value was provided when the message was queued) or the bit is not set when tpdequeue() is called, tpdequeue() completes with the flag turned off.

TPQPRIORITY
: If this flag is set, the call to tpdequeue() is successful, and the message was queued with an explicit priority, then the priority is stored in *ctl->priority*. The priority is in the range 1 to 100, inclusive, and the higher the number, the higher the priority (that is, a message with a higher number is dequeued before a message with a lower number). For queues not ordered by priority, the value is informational.

  If no priority was explicitly specified when the message was queued and the call to tpdequeue() is successful, the priority for the message is 50.

TPQMSGID
: If this flag is set and the call to tpdequeue() is successful, the message identifier is stored in *ctl->msgid*. The entire 32 bytes of the message identifier value are significant.

TPQCORRID
: If this flag is set, the call to tpdequeue() is successful, and the message was queued with a correlation identifier, then the correlation identifier is stored in *ctl->corrid*. The entire 32 bytes of the correlation identifier value are significant. Any BEA Tuxedo /Q provided reply to a message has the correlation identifier of the original request message.

TPQDELIVERYQOS
: If this flag is set, the call to tpdequeue() is successful, and the message was queued with a delivery quality of service, then the flag—TPQQOSDEFAULTPERSIST,

TPQQOSPERSISTENT, or TPQQOSNONPERSISTENT—is stored in *ctl->delivery_qos*. If no delivery quality of service was explicitly specified when the message was queued, the default delivery policy of the target queue dictates the delivery quality of service for the message.

TPQREPLYQOS
: If this flag is set, the call to tpdequeue() is successful, and the message was queued with a reply quality of service, then the flag—TPQQOSDEFAULTPERSIST, TPQQOSPERSISTENT, or TPQQOSNONPERSISTENT—is stored in *ctl->reply_qos*. If no reply quality of service was explicitly specified when the message was queued, the default delivery policy of the *ctl->replyqueue* queue dictates the delivery quality of service for any reply.

  Note that the default delivery policy is determined when the reply to a message is enqueued. That is, if the default delivery policy of the reply queue is modified between the time that the original message is enqueued and the reply to the message is enqueued, the policy used is the one in effect when the reply is finally enqueued.

TPQREPLYQ
: If this flag is set, the call to tpdequeue() is successful, and the message was queued with a reply queue, then the name of the reply queue is stored in *ctl->replyqueue*. Any reply to the message should go to the named reply queue within the same queue space as the request message.

TPQFAILUREQ
: If this flag is set, the call to tpdequeue() is successful, and the message was queued with a failure queue, then the name of the failure queue is stored in *ctl->failurequeue*. Any failure message should go to the named failure queue within the same queue space as the request message.

The following remaining bits for the *flags* parameter are cleared (set to zero) when tpdequeue() is called: TPQTOP, TPQBEFOREMSGID, TPQTIME_ABS, TPQTIME_REL, TPQEXPTIME_ABS, TPQEXPTIME_REL, and TPQEXPTIME_NONE. These bits are valid bits for the *flags* parameter controlling input information for tpenqueue().

If the call to tpdequeue() failed and tperrno(5) is set to TPEDIAGNOSTIC, a value indicating the reason for failure is returned in *ctl->diagnostic*. The valid codes for *ctl->diagnostic* include those for tpenqueue() described in "TPQCTL Structure" on page 3-5 (except for QMENOSPACE and QMERELEASE) and the following additional codes.

[QMENOMSG]
: No message was available for dequeuing. Note that it is possible that the message exists on the queue and another application process has read the message from the queue. In this

case, the message may be put back on the queue if that other process rolls back the transaction.

[QMEINUSE]
When dequeuing a message by message identifier or correlation identifier, the specified message is in use by another transaction. Otherwise, all messages currently on the queue are in use by other transactions. This diagnostic is not returned by queue managers from BEA Tuxedo release 7.1 or later.

Using TPQWAIT

When tpdequeue() is called with *flags* (of the TPQCTL structure) set to include TPQWAIT, if a message is not immediately available, the TMQUEUE server waits for the arrival, on the queue, of a message that matches the tpdequeue() request before tpdequeue() returns control to the caller. The TMQUEUE process sets the waiting request aside and processes requests from other processes while waiting to satisfy the first request. If TPQGETBYMSGID and/or TPQGETBYCORRID are also specified, the server waits until a message with the indicated message identifier and/or correlation identifier becomes available on the queue. If neither of these flags is set, the server waits until any message is put onto the queue. The amount of time it waits is controlled by the caller's transaction timeout, if the call is in transaction mode, or by the -t option in the CLOPT parameter of the TMQUEUE server, if the call is not in transaction mode.

The TMQUEUE server can handle a number of waiting tpdequeue() requests at the same time, as long as action resources are available to handle the request. If there are not enough action resources configured for the queue space, tpdequeue() fails. If this happens on your system, increase the number of action resources for the queue space.

Error Handling When Using TMQFORWARD Services

In considering how best to handle errors when dequeuing it is helpful to differentiate between two types of errors:

- Errors encountered by TMQFORWARD(5) as it attempts to dequeue a message to forward to the requested service
- Errors that occur in the service that processes the request

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. The message handling method chosen should be mostly 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.

Procedure for Dequeuing Replies from Services Invoked Through TMQFORWARD

If your application expects to receive replies to queued messages, the following 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(5)` 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 `TPQCTL` 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 for Peer-to-Peer Communication

In all of the foregoing discussion of enqueuing and dequeuing messages there has been an implicit assumption that queues were being used as an alternative form of request/response processing. A message does not have to be a service request. The queued message facility can transfer data from one process to another as effectively as a service request. This style of communication between applications or clients is called peer-to-peer communication.

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

CHAPTER 4

BEA Tuxedo /Q COBOL Language Programming

This topic includes the following sections:

Introduction

This topic provides information about using the ATMI COBOL language functions for enqueuing and dequeuing messages: `TPENQUEUE(3cbl)` and `TPDEQUEUE(3cbl)`, 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 *Programming BEA Tuxedo ATMI*

Applications Using COBOL. Detailed pages on all the ATMI functions are in the *BEA Tuxedo ATMI COBOL Function Reference*.

Where Requests Can Originate

The calls used to place a message on a BEA Tuxedo /Q queue can originate from 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 discussion of BEA Tuxedo /Q programming in this topic primarily reflects the client-side portion of the figure "Queued Service Invocation" on page 1-2. The figure shows how a client (or a process acting in the role of a client) queues a message by calling TPENQUEUE(3cbl) and specifying a queue space made available through a TMQUEUE(5) server. The client later retrieves a reply via a TPDEQUEUE(3cbl) call to TMQUEUE.

The figure shows the queued message being dequeued by the server TMQFORWARD(5) and sent to an application server for processing (via TPCALL(3cbl)). When a reply to TPCALL is received, TMQFORWARD enqueues the reply message. Because TMQFORWARD provides an interface between the queue space and existing application services, further application coding is not required. For that reason, this topic concentrates on the client-to-queue space 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.

TPENQUEUE() Arguments

There are some important arguments to control the operation of TPENQUEUE(3cbl). Lets 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 name of the default service is the same as the server name, TMQUEUE. In this case the configuration file might 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 then looks 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 TMQUEUE(5) reference page shows how alias service names can be included when the server is built and specified in the configuration file. The sample program in "A Sample Application" on page A-1, also specifies an alias service name.

TPENQUEUE(): The QNAME in TPQUEDEF-REC Argument

When message queues are being used within a queue space to invoke services, they 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 in "Procedure for Dequeuing Replies from Services Invoked Through TMQFORWARD" on page 4-26.

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 and this setting is used, the message is not enqueued within the caller's transaction. A caller in transaction mode that sets this to true is still subject to the transaction timeout (and no other). If message enqueuing fails that was invoked with this setting, the caller's transaction is not affected. 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
: The message is not enqueued if a blocking condition exists. If TPNOBLOCK is set and a blocking condition exists such as the internal buffers into which the message is transferred are full, the call fails and tperrno(5) is set to TPEBLOCK. If TPNOBLOCK is set and a blocking condition exists because the target queue is opened *exclusively* by another application, the call fails, tperrno() is set to TPEDIAGNOSTIC, and the diagnostic field of the TPQCTL structure is set to QMESHARE. In the latter case, the other application, which is based on a BEA product other than the BEA Tuxedo system, opened the queue for exclusive read and/or write using the Queuing Services API (QSAPI). Either TPNOBLOCK or TPBLOCK must be set.

TPBLOCK
: When TPBLOCK is set and a blocking condition exists, the caller blocks until the condition subsides or a timeout occurs (either transaction or blocking timeout). 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.

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 the following listing.

Listing 4-1 The TPQUEDEF-REC Structure

```
05 TPBLOCK-FLAG         PIC S9(9) COMP-5.
        88 TPBLOCK              VALUE 0.
        88 TPNOBLOCK            VALUE 1.
05 TPTRAN-FLAG  PIC S9(9) COMP-5.
        88 TPTRAN               VALUE 0.
        88 TPNOTRAN             VALUE 1.
05 TPTIME-FLAG  PIC S9(9) COMP-5.
        88 TPTIME               VALUE 0.
        88 TPNOTIME             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 TPCHANGE             VALUE 0.
        88 TPNOCHANGE           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 TPQGETBYMSGIDOLD     VALUE 1.
        88 TPQGETBYCORRIDOLD    VALUE 2.
        88 TPQGETBYMSGID        VALUE 3.
        88 TPQGETBYCORRID       VALUE 4.
05 TPQUE-WAIT-FLAG      PIC S9(9) COMP-5.
        88 TPQNOWAIT            VALUE 0.
        88 TPQWAIT              VALUE 1.
05 TPQUE-DELIVERY-FLAG  PIC S9(9) COMP-5.
        88 TPQNODELIVERYQOS     VALUE 0.
        88 TPQDELIVERYQOS       VALUE 1.
05 TPQUEQOS-DELIVERY-FLAG       PIC S9(9) COMP-5.
        88 TPQQOSDELIVERYDEFAULTPERSIST VALUE 0.
        88 TPQQOSDELIVERYPERSISTENT     VALUE 1.
        88 TPQQOSDELIVERYNONPERSISTENT  VALUE 2.
05 TPQUE-REPLY-FLAG     PIC S9(9) COMP-5.
        88 TPQNOREPLYQOS                VALUE 0.
        88 TPQREPLYQOS                  VALUE 1.
05 TPQUEQOS-REPLY-FLAG  PIC S9(9) COMP-5.
        88 TPQQOSREPLYDEFAULTPERSIST    VALUE 0.
        88 TPQQOSREPLYPERSISTENT        VALUE 1.
```

```
        88 TPQQOSREPLYNONPERSISTENT     VALUE 2.
05 TPQUE-EXPTIME-FLAG   PIC S9(9) COMP-5.
        88 TPQNOEXPTIME          VALUE 0.
        88 TPQEXPTIME-ABS        VALUE 1.
        88 TPQEXPTIME-REL        VALUE 2.
        88 TPQEXPTIME-NONE       VALUE 3.
05 TPQUE-PEEK-FLAG      PIC S9(9) COMP-5.
        88 TPQNOPEEK             VALUE 0.
        88 TPQPEEK               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 QMEABORTED            VALUE -8.
        88 QMEPROTO              VALUE -9.
        88 QMEBADQUEUE           VALUE -10.
        88 QMENOMSG              VALUE -11.
        88 QMEINUSE              VALUE -12.
        88 QMENOSPACE            VALUE -13.
        88 QMERELEASE            VALUE -14.
        88 QMEINVHANDLE          VALUE -15.
        88 QMESHARE              VALUE -16.
05 DEQ-TIME             PIC 9(9) COMP-5.
05 EXP-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 value 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. Set TPQDEFAULT to use default queue ordering. TPQTOP, TPQBEFOREMSGID, or TPQDEFAULT must be set.

TPQBEFOREMSGID
: Setting this value 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. Set TPQDEFAULT to use default queue ordering. TPQTOP, TPQBEFOREMSGID, or TPQDEFAULT must be set.

  Note that the entire 32 bytes of the message identifier value are significant, so the value identified by MSGID must be completely initialized (for example, padded with spaces).

TPQTIME-ABS
: If this value is set, the message is made available after the time specified by DEQ-TIME. DEQ-TIME is an absolute time value as generated by time(2) or mktime(3C) (the number of seconds since 00:00:00 Universal Coordinated Time—UTC, January 1, 1970). Set TPQNOTIME if neither an absolute or relative time value is set. TPQTIME-ABS, TPQTIME-REL, or TPQNOTIME must be set. The absolute time is determined by the clock on the machine where the queue manager process resides.

TPQTIME-REL
: If this value is set, the message is made available after a time relative to the completion of the enqueuing operation. DEQ-TIME specifies the number of seconds to delay after the enqueuing completes before the submitted message should be available. Set TPQNOTIME if neither an absolute or relative time value is set. TPQTIME-ABS, TPQTIME-REL, or TPQNOTIME must be set.

TPQPRIORITY
: If this value is set, the priority at which the message should be enqueued is stored in PRIORITY. The priority must be in the range 1 to 100, inclusive. The higher the number, the higher the priority (that is, a message with a higher number is dequeued before a

message with a lower number). For queues not ordered by priority, this value is informational. If TPQNOPRIORITY is set, the priority for the message is 50 by default.

TPQCORRID

If this value is set, the correlation identifier value specified in CORRID is available when a message is dequeued with TPDEQUEUE(). This identifier accompanies any reply or failure message that is queued so that an application can correlate a reply with a particular request. Set TPQNOCORRID if a correlation identifier is not available.

Note that the entire 32 bytes of the correlation identifier value are significant, so the value specified in CORRID must be completely initialized (for example, padded with spaces).

TPQREPLYQ

If this value is set, a reply queue named in REPLYQUEUE is associated with the queued message. Any reply to the message is queued to the named queue within the same queue space as the request message. Set TPQNOREPLYQ if a reply queue name is not available.

TPQFAILUREQ

If this value is set, a failure queue named in FAILUREQUEUE is associated with the queued message. If (1) the enqueued message is processed by TMQFORWARD(), (2) TMQFORWARD was started with the -d option, and (3) the service fails and returns a non-NULL reply, a failure message consisting of the reply and its associated tpurcode is enqueued to the named queue within the same queue space as the original request message. Set TPQNOFAILUREQ if a failure queue name is not available.

TPQDELIVERYQOS

TPQREPLYQOS

If TPQDELIVERYQOS is set, the flags specified by TPQUEQOS-DELIVERY-FLAG control the quality of service for message delivery. One of the following mutually exclusive flags must be set: TPQQOSDELIVERYDEFAULTPERSIST, TPQQOSDELIVERYPERSISTENT, or TPQQOSDELIVERYNONPERSISTENT. If TPQDELIVERYQOS is not set, TPQNODELIVERYQOS must be set. When TPQNODELIVERYQOS is set, the default delivery policy of the target queue dictates the delivery quality of service for the message.

If TPQREPLYQOS is set, the flags specified by TPQUEQOS-REPLY-FLAG control the quality of service for reply message delivery for any reply. One of the following mutually exclusive flags must be set: TPQQOSREPLYDEFAULTPERSIST, TPQQOSREPLYPERSISTENT, or TPQQOSREPLYNONPERSISTENT. The TPQREPLYQOS flag is used when a reply is returned from messages processed by TMQFORWARD. Applications not using TMQFORWARD to invoke services may use the TPQREPLYQOS flag as a hint for their own reply mechanism.

If TPQREPLYQOS is not set, TPQNOREPLYQOS must be set. When TPQNOREPLYQOS is set, the default delivery policy of the REPLYQUEUE queue dictates the delivery quality of

service for any reply. Note that the default delivery policy is determined when the reply to a message is enqueued. That is, if the default delivery policy of the reply queue is modified between the time that the original message is enqueued and the reply to the message is enqueued, the policy used is the one in effect when the reply is finally enqueued.

The valid TPQUEQOS-DELIVERY-FLAG and TPQUEQOS-REPLY-FLAG flags are:

TPQQOSDELIVERYDEFAULTPERSIST
TPQQOSREPLYDEFAULTPERSIST
: These flags specify that the message is to be delivered using the default delivery policy specified on the target or reply queue.

TPQQOSDELIVERYPERSISTENT
TPQQOSREPLYPERSISTENT
: These flags specify that the message is to be delivered in a persistent manner using the disk-based delivery method. When specified, these flags override the default delivery policy specified on the target or reply queue.

TPQQOSDELIVERYNONPERSISTENT
TPQQOSREPLYNONPERSISTENT
: These flags specify that the message is to be delivered in a non-persistent manner using the memory-based delivery method; the message is queued in memory until it is dequeued. When specified, these flags override the default delivery policy specified on the target or reply queue.

  If the caller is transactional, non-persistent messages are enqueued within the caller's transaction, however, non-persistent messages are lost if the system is shut down or crashes or the IPC shared memory for the queue space is removed.

TPQEXPTIME-ABS
: If this value is set, the message has an absolute expiration time, which is the absolute time when the message will be removed from the queue.
The absolute expiration time is determined by the clock on the machine where the queue manager process resides.

  The absolute expiration time is specified by the value stored in EXP-TIME. EXP-TIME must be set to an absolute time generated by time(2) or mktime(3C) (the number of seconds since 00:00:00 Universal Coordinated Time—UTC, January 1, 1970).

  If an absolute time is specified that is earlier than the time of the enqueue operation, the operation succeeds, but the message is not counted for the purpose of calculating thresholds. If the expiration time is before the message availability time, the message is not available for dequeuing unless either the availability or expiration time is changed so

that the availability time is before the expiration time. In addition, these messages are removed from the queue at expiration time even if they were never available for dequeuing. If a message expires during a transaction, the expiration does not cause the transaction to fail. Messages that expire while being enqueued or dequeued within a transaction are removed from the queue when the transaction ends. There is no acknowledgment that the message has expired.

One of the following must be set: TPQEXPTIME-ABS, TPQEXPTIME-REL, TPQEXPTIME-NONE, or TPQNOEXPTIME.

TPQEXPTIME-REL

If this value is set, the message has a relative expiration time, which is the number of seconds *after* the message arrives at the queue that the message is removed from the queue. The relative expiration time is specified by the value stored in EXP-TIME.

If the expiration time is before the message availability time, the message is not available for dequeuing unless either the availability or expiration time is changed so that the availability time is before the expiration time. In addition, these messages are removed from the queue at expiration time even if they were never available for dequeuing. The expiration of a message during a transaction does cause the transaction to fail. Messages that expire while being enqueued or dequeued within a transaction are removed from the queue when the transaction ends. There is no acknowledgment that the message has expired.

One of the following must be set: TPQEXPTIME-ABS, TPQEXPTIME-REL, TPQEXPTIME-NONE, or TPQNOEXPTIME.

TPQEXPTIME-NONE

Setting this value indicates that the message should not expire. This flag overrides any default expiration policy associated with the target queue. You can remove a message by dequeuing it or by deleting it via an administrative interface. One of the following must be set: TPQEXPTIME-ABS, TPQEXPTIME-REL, TPQEXPTIME-NONE, or TPQNOEXPTIME.

TPQNOEXPTIME

Setting this value specifies that the default expiration time associated with the target queue applies to the message. One of the following must be set: TPQEXPTIME-ABS, TPQEXPTIME-REL, TPQEXPTIME-NONE, or TPQNOEXPTIME.

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

As output from TPENQUEUE(), the following may be set in the *TPQUEDEF-REC* structure:

```
05 DIAGNOSTIC            PIC S9(9) COMP-5.
05 MSGID                 PIC X(32).
```

The following is a valid setting in *TPQUEDEF-REC* controlling output information from TPENQUEUE(). If this setting is true when TPENQUEUE() is called, the BEA Tuxedo /Q server TMQUEUE(5) populates the associated element in the record with a message identifier. If this setting is not true when TPENQUEUE() is called, TMQUEUE() does *not* populate the associated element in the record with a message identifier.

TPQMSGID
: If this value is set and the call to TPENQUEUE() is successful, the message identifier is stored in MSGID. The entire 32 bytes of the message identifier value are significant, so the value stored in MSGID is completely initialized (for example, padded with null characters). The actual padding character used for initialization varies between releases of the BEA Tuxedo /Q component. If TPQNOMSGID is set, the message identifier is not available.

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

If the call to TPENQUEUE() fails and TP-STATUS is set to TPEDIAGNOSTIC, a value indicating the reason for failure is returned in DIAGNOSTIC. The 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 not in transaction mode or was made with the TPNOTRAN setting and an error occurred trying to start a transaction in which to enqueue the message. This diagnostic is not returned by a queue manager from BEA Tuxedo release 7.1 or later.

[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.

[QMEABORTED]
The operation was aborted. If the aborted operation was being executed within a global transaction, the global transaction is marked rollback-only. Otherwise, the queue manager aborts the operation.

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

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

[QMENOSPACE]
Due to an insufficient resource, such as no space on the queue, the message with its required quality of service (persistent or non-persistent storage) was not enqueued. QMENOSPACE is returned when any of the following configured resources is exceeded: (1) the amount of disk (persistent) space allotted to the queue space, (2) the amount of memory (non-persistent) space allotted to the queue space, (3) the maximum number of simultaneously active transactions allowed for the queue space, (4) the maximum number of messages that the queue space can contain at any one time, (5) the maximum number of concurrent actions that the Queuing Services component can handle, or (6) the maximum number of authenticated users that may concurrently use the Queuing Services component.

[QMERELEASE]
An attempt was made to enqueue a message to a queue manager that is from a version of the BEA Tuxedo system that does not support a newer feature.

[QMESHARE]
When enqueuing a message from a specified queue, the specified queue is opened *exclusively* by another application. The other application is one based on a BEA product other than the BEA Tuxedo system that opened the queue for exclusive read and/or write using the Queuing Services API (QSAPI).

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 the override 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

You can set a value in PRIORITY to specify the priority for the message. The value must be in the range 1 to 100; the higher the number, the higher the priority, unlike values specified with the UNIX nice command. If PRIORITY was not one of the queue ordering parameters, setting a priority here has no effect on the dequeuing order. The priority value is retained however, so that it can be inspected when the message is dequeued.

Setting a Message Availability Time

You can specify in DEQ-TIME either an absolute time or a time relative to the completion of the enqueuing operation at which the message is made available. You set either TPQTIME-ABS or TPQTIME-REL to indicate how the value should be treated. A queue may be created with time as a queue-ordering criterion, in which case messages are ordered by the message availability time.

The following example shows how to enqueue a message with a relative time. The sample message will become available 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

If the 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 outside of the caller's 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 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 Messages

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 queue 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.

TPDEQUEUE() Arguments

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 the TMQUEUE 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 name of the default service is the same as that of the server, TMQUEUE. In this case the configuration file may 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 then looks like the following:

```
  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 TMQUEUE(5) reference page shows how alias service names can be included when the server is built and specified in the configuration file. The sample program in "A Sample Application" on page A-1, also specifies an alias service name.

TPDEQUEUE(): The QNAME in TPQUEDEF-REC Argument

Queue names in a queue space must be agreed upon by the applications that will access the queue space. This requirement is especially important for reply queues. If QNAME refers to a reply queue, the administrator creates it (and often an error queue) in the same manner that he or she creates any other queue. *QNAME* contains the name of the queue from which to retrieve a message or reply.

TPDEQUEUE(): The DATA-REC and LEN in TPTYPE-REC Arguments

These 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 length prior to the call to TPDEQUEUE()

and compare it to LEN after the call completes. 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 outside of the caller's 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
: The message is not dequeued if a blocking condition exists. If TPNOBLOCK is set and a blocking condition exists such as the internal buffers into which the message is transferred are full, the call fails and tperrno(5) is set to TPEBLOCK. If TPNOBLOCK is set and a blocking condition exists because the target queue is opened *exclusively* by another application, the call fails, tperrno() is set to TPEDIAGNOSTIC, and the diagnostic field of the TPQCTL structure is set to QMESHARE. In the latter case, the other application, which is based on a BEA product other than the BEA Tuxedo system, opened the queue for exclusive read and/or write using the Queuing Services API (QSAPI). Either TPNOBLOCK or TPBLOCK must be set.

TPBLOCK
: When TPBLOCK is set and a blocking condition exists, the caller blocks until the condition subsides or a timeout occurs (either transaction or blocking timeout). 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
: Setting this value asks that the call be immune to blocking timeouts; transaction timeouts may still occur. Either TPNOTIME or TPTIME must be set.

TPTIME
: Setting this value asks that the call receive blocking timeouts. Either TPNOTIME or TPTIME must be set.

TPNOCHANGE
: If this value is set, the record type of *DATA-REC* is not allowed to change. That is, the type and subtype 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*, *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
: Setting this value says that any underlying system calls that are interrupted by a signal should be reissued. Either TPSIGRSTRT or TPNOSIGRSTRT must be set.

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

TPQUEDEF-REC Structure

The first argument to TPDEQUEUE() is the 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 "The TPQUEDEF-REC Structure" on page 4-6.

As input to TPDEQUEUE(), the following elements may be set in the TPQUEDEF structure:

```
05 MSGID                PIC X(32).
05 CORRID               PIC X(32).
```

The following is a list of valid settings in *TPQUEDEF-REC* that control input for TPDEQUEUE():

TPQGETNEXT
: Setting this value requests that the next message on the queue be dequeued, using the default queue order. One of the following must be set: TPQGETNEXT, TPQGETBYMSGID, or TPQGETBYCORRID.

TPQGETBYMSGID
: Setting this value requests that the message identified by MSGID be dequeued. The message identifier is returned by a prior call to TPENQUEUE(). Note that the message identifier is not valid if the message has moved from one queue to another. Note also that the entire 32 bytes of the message identifier value are significant, so the value identified by MSGID must be completely initialized (for example, padded with spaces).

  One of the following must be set: TPQGETNEXT, TPQGETBYMSGID, or TPQGETBYCORRID.

TPQGETBYCORRID
: Setting this value requests that the message identified by CORRID be dequeued. The correlation identifier is specified by the application when enqueuing the message with TPENQUEUE(). Note that the entire 32 bytes of the correlation identifier value are significant, so the value identified by CORRID must be completely initialized (for example, padded with spaces).

  One of the following must be set: TPQGETNEXT, TPQGETBYMSGID, or TPQGETBYCORRID.

TPQWAIT
: Setting this value indicates that an error should not be returned if the queue is empty. Instead, the process should wait until a message is available. Set TPQNOWAIT to not wait until a message is available. If TPQWAIT is set in conjunction with TPQGETBYMSGID or TPQGETBYCORRID, it indicates that an error should not be returned if no message with the specified message identifier or correlation identifier is present in the queue. Instead, the process should wait until a message meeting the criteria is available. The process is still subject to the caller's transaction timeout, or, when not in transaction mode, the process is still subject to the timeout specified on the TMQUEUE process by the -t option.

  If a message matching the desired criteria is not immediately available and the configured action resources are exhausted, TPDEQUEUE fails, TP-STATUS is set to TPEDIAGNOSTIC, and DIAGNOSTIC is set to QMESYSTEM.

  Note that each TPDEQUEUE() request specifying the TPQWAIT control parameter requires that a queue manager (TMQUEUE) action object be available if a message satisfying the condition is not immediately available. If one is not available, the TPDEQUEUE() request fails. The number of available queue manager actions are specified when a queue space is created or modified. When a waiting dequeue request completes, the associated action object associated is made available for another request.

TPQPEEK
: If TPQPEEK is set, the specified message is read but not removed from the queue. The TPNOTRAN flag must be set. It is not possible to read messages enqueued or dequeued within a transaction before the transaction completes.

When a thread is non-destructively dequeuing a message using TPQPEEK, the message may not be seen by other non-blocking dequeuers for the brief time the system is processing the non-destructive dequeue request. This includes dequeuers using specific selection criteria (such as message identifier and correlation identifier) that are looking for the message currently being non-destructively dequeued.

On output from TPDEQUEUE(), the following elements may be set in *TPQUEDEF-REC*:

```
05 PRIORITY     PIC S9(9) COMP-5.
05 MSGID        PIC X(32).
05 CORRID       PIC X(32).
05 TPQUEQOS-DELIVERY-FLAG    PIC S9(9) COMP-5.
05 TPQUEQOS-REPLY-FLAG       PIC S9(9) COMP-5.
05 REPLYQUEUE                PIC X(15).
05 FAILUREQUEUE              PIC X(15).
05 DIAGNOSTIC                PIC S9(9) COMP-5.
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 in *TPQUEDEF-REC* controlling output information from TPDEQUEUE(). For any of these settings, if the setting is true when TPDEQUEUE() is called, the associated element in the record is populated with the value provided when the message was queued, and the setting remains true. If the value is not available (that is, no value was provided when the message was queued) or the setting is not true when TPDEQUEUE() is called, TPDEQUEUE() completes with the setting not true.

TPQPRIORITY

If this value is set, the call to TPDEQUEUE() is successful, and the message was queued with an explicit priority, then the priority is stored in PRIORITY. The priority is in the range 1 to 100, inclusive, and the higher the number, the higher the priority (that is, a message with a higher number is dequeued before a message with a lower number). If TPQNOPRIORITY is set, the priority is not available.

Note that if no priority was explicitly specified when the message was queued, the priority for the message is 50.

TPQMSGID

If this value is set and the call to TPDEQUEUE() is successful, the message identifier is stored in MSGID. The entire 32 bytes of the message identifier value are significant. If TPQNOMSGID is set, the message identifier is not available.

TPQCORRID

If this value is set, the call to TPDEQUEUE() is successful, and the message was queued with a correlation identifier, then the correlation identifier is stored in CORRID. The entire 32 bytes of the correlation identifier value are significant. Any BEA Tuxedo /Q provided reply to a message has the correlation identifier of the original message. If TPQNOCORRID is set, the correlation identifier is not available.

TPQDELIVERYQOS

If this value is set, the call to TPDEQUEUE() is successful, and the message was queued with a delivery quality of service, then the flag—TPQQOSDELIVERYDEFAULTPERSIST, TPQQOSDELIVERYPERSISTENT, or TPQQOSDELIVERYNONPERSISTENT—specified by TPQUEQOS-DELIVERY-FLAG indicates the delivery quality of service. If TPQNODELIVERYQOS is set, the delivery quality of service is not available.

Note that if no delivery quality of service was explicitly specified when the message was queued, the default delivery policy of the target queue dictates the delivery quality of service for the message.

TPQREPLYQOS

If this value is set, the call to TPDEQUEUE() is successful, and the message was queued with a reply quality of service, then the flag—TPQQOSREPLYDEFAULTPERSIST, TPQQOSREPLYPERSISTENT, or TPQQOSREPLYNONPERSISTENT—specified by TPQUEQOS-REPLY-FLAG indicates the reply quality of service. If TPQNOREPLYQOS is set, the reply quality of service is not available.

Note that if no reply quality of service was explicitly specified when the message was queued, the default delivery policy of the REPLYQUEUE queue dictates the delivery quality of service for any reply. The default delivery policy is determined when the reply to a message is enqueued. That is, if the default delivery policy of the reply queue is modified between the time that the original message is enqueued and the reply to the message is enqueued, the policy used is the one in effect when the reply is finally enqueued.

TPQREPLYQ

If this value is set, the call to TPDEQUEUE() is successful, and the message was queued with a reply queue, then the name of the reply queue is stored in REPLYQUEUE. Any reply to the message should go to the named reply queue within the same queue space as the request message. If TPQNOREPLYQ is set, the reply queue is not available.

TPQFAILUREQ

If this value is set, the call to TPDEQUEUE() is successful, and the message was queued with a failure queue, then the name of the failure queue is stored in FAILUREQUEUE. Any failure message should go to the named failure queue within the same queue space as the request message. If TPQNOFAILUREQ is set, the failure queue is not available.

The remaining settings in *TPQUEDEF-REC* are set to the following values when TPDEQUEUE() is called: TPQNOTOP, TPQNOBEFOREMSGID, TPQNOTIME_ABS, TPQNOTIME_REL, TPQNOEXPTIME_ABS, TPQNOEXPTIME_REL, and TPQNOEXPTIME_NONE.

If the call to TPDEQUEUE() fails 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 for TPENQUEUE() described in "TPQUEDEF-REC Structure" on page 4-6 (except for QMENOSPACE and QMERELEASE) and the following additional codes.

[QMENOMSG**]**
: No message was available for dequeuing. Note that it is possible that the message exists on the queue and another application process has read the message from the queue. In this case, the message may be put back on the queue if that other process rolls back the transaction.

[QMEINUSE**]**
: When dequeuing a message by message identifier or correlation identifier, the specified message is in use by another transaction. Otherwise all messages currently on the queue are in use by other transactions. This diagnostic is not returned by a queue manager from BEA Tuxedo release 7.1 or later.

Using TPQWAIT

When TPDEQUEUE() is called with flags set to include TPQWAIT, if a message is not immediately available, the TMQUEUE server waits for the arrival, on the queue, of a message that matches the TPDEQUEUE() request before TPDEQUEUE() returns control to the caller. The TMQUEUE process sets the waiting request aside and processes requests from other processes while waiting to satisfy the first request. If TPQGETBYMSGID and/or TPQGETBYCORRID are also specified, the server waits until a message with the indicated message identifier and/or correlation identifier becomes available on the queue. If neither of these flags is set, the server waits until any message is put onto the queue. The amount of time it waits is controlled by the caller's transaction timeout, if the call is in transaction mode, or by the -t option in the CLOPT parameter of the TMQUEUE server, if the call is not in transaction mode.

The TMQUEUE server can handle a number of waiting TPDEQUEUE() requests at the same time, as long as action resources are available to handle the request. If there are not enough action resources configured for the queue space, TPDEQUEUE() fails. If this happens on your system, increase the number of action resources for the queue space.

Error Handling When Using TMQFORWARD Services

In considering how best to handle errors when dequeuing it is helpful to differentiate between two types of errors:

- Errors encountered by `TMQFORWARD(5)` as it attempts to dequeue a message to forward to the requested service
- Errors that occur in the service that processes the request

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. The message handling method chosen should be mostly 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 is enqueued to the failure queue in the same transaction as the one in which the message is deleted from the queue.

Procedure for Dequeuing Replies from Services Invoked Through TMQFORWARD

If your application expects to receive replies to queued messages, the following 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 for Peer-to-Peer Communication

In all of the foregoing discussion of enqueuing and dequeuing messages there has been an implicit assumption that queues were being used as an alternative form of request/response processing. A message does not have to be a service request. The queued message facility can transfer data from one process to another as effectively as a service request. This style of communication between applications or clients is called peer-to-peer communication.

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

APPENDIX A

A Sample Application

This topic includes the following sections:

Overview

The sample application in this topic contains a description of a one-client, one-server application using BEA Tuxedo /Q called `qsample`. An interactive form of this software is distributed with the BEA Tuxedo software.

Prerequisites

Before you can run the sample application, the BEA Tuxedo software must be installed and built so that the files and commands referred to in this topic are available. If you are personally responsible for installing the BEA Tuxedo software, consult the *Installing the BEA Tuxedo System* 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.

What Is qsample?

qsample is a very basic BEA Tuxedo application that uses BEA Tuxedo /Q. It has one application client and server, and uses two system servers. TMQUEUE(5) and TMQFORWARD(5). 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.

Building qsample

The following procedure provides instructions on building and running the qsample application.

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 and how to configure and run it.

`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 `setenv` file.

   Open the `setenv` file and modify the `TUXDIR` value to the absolute path of the root directory of the BEA Tuxedo system installation. Remove the angle bracket characters (< and >) when editing this value.

5. Run `runsample`.

   The `runsample` script contains several commands; each command is preceded by a comment line that describes the purpose of the command.

```
#set the environment
. ./setenv
chmod +w ubb.sample
```

```
uname="`uname -n`"
ed ubb.sample<<!
g;<uname -n>;s;;${uname};
g;<full path of Tuxedo software>;s;;${TUXDIR};
g;<full path of APPDIR>;s;;${APPDIR};
w
q
!
#build the client and server
make client server
#create the tuxconfig file
tmloadcf -y ubb.sample
#create the TLOG
./crlog
#create the QUE
./crque
#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
./rmipc
#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 generated 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 topic, we suggest some things that we recommend you look at and try; you will undoubtedly be able to think of others as you explore the application more closely.

setenv: Set 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, SHLIB_PATH, or LIBPATH are important if you are building the system with shared libraries. The correct variable to use depends on your operating system. 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 /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 /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.