BEA MessageQ

Introduction to Message Queuing

BEA MessageQ for OpenVMS Version 5.0
Document Edition 5.0
March 2000

Copyright

Restricted Rights Legend

Trademarks or Service Marks

BEA MessageQ Introduction to Message Queuing

Document Edition	Date	Software Version
5.0	March 2000	BEA MessageQ, Version 5.0

Contents

Preface

Purpose of This Document vii
Who Should Read This Document vii
How This Document Is Organized vii
How to Use This Document viii
Opening the Document in a Web Browser viii
Printing from a Web Browser ix
Documentation Conventions ix
Related Documentation xi
MessageQ Documentation xi
Contact Information xii
Documentation Support xii
Customer Support xii

1. What Is BEA MessageQ?

The Distributed Computing Revolution 1-1
Traditional Versus Distributed Applications 1-2
Major Trends in Distributed Computing 1-3
Distributed Computing Models 1-5
Peer-to-Peer Communication Model 1-5
Client/Server Communication Model 1-6
Technologies for Building Distributed Applications 1-6
DCE/Remote Procedure Call 1-7
Object Transaction Monitoring 1-8
Message Queuing 1-8
Message Queuing Basics 1-9
What Is a Message? 1-9

What Is Message Queuing? 1-10
How Does BEA MessageQ Work? 1-11
Choosing the BEA MessageQ Server or Client 1-12
How the BEA MessageQ Client Works 1-13
When to Choose the BEA MessageQ Client 1-14
Key Features of BEA MessageQ 1-15
BEA MessageQ Benefits 1-16
Standardized Integration Approach 1-17
Guaranteed Delivery 1-17
Application Portability 1-18
Message Bus Simplifies Communication 1-18
Broad Multiplatform Support 1-19
Flexibility to Meet Changing Application Needs 1-20

2. Sending and Receiving BEA MessageQ Messages

Overview of BEA MessageQ API Functions 2-2
Configuring the BEA MessageQ Environment 2-5
Defining Queues and Their Attributes 2-5
Configuring Buses, Groups and Queues 2-8
Designing Your BEA MessageQ Environment 2-8
Configuring Each Message Queuing Group 2-10
Starting Each Message Queuing Group 2-11
Attaching to the Message Queuing Bus 2-11
Attaching by Name 2-13
Attaching by Number 2-13
Attaching to a Temporary Queue 2-13
Sending a Message 2-14
Selecting a Messaging Style 2-16
Using Buffer-Style Messaging 2-17
Using FML-Style Messaging 2-18
Choosing a Delivery Mode 2-18
Sender Notification 2-19
Delivery Interest Point 2-20
Undeliverable Message Action 2-23

Receiving a Message 2-24
Confirming Receipt of a Message 2-24
Using the PAMS Status Buffer 2-25
Using the show_buffer Argument 2-26
Using Message Classes with BEA MessageQ and BEA TUXEDO 2-27
Detaching from the Message Queuing Bus 2-27
Exchanging Messages Between BEA MessageQ and BEA TUXEDO 2-28

3. Designing and Developing BEA MessageQ Applications

Designing a BEA MessageQ Application 3-1
Solving the Business Problem 3-2
Developing the Communications Model 3-3
Defining Major Application Needs 3-5
Choosing the Style of Messaging 3-6
Choosing Recoverable or Nonrecoverable Message Delivery 3-6
Choosing Asynchronous or Synchronous Messaging 3-7
Using Message Broadcasting 3-8
Using Message Selection 3-8
Load Balancing with MRQs 3-8
Choosing Single Reader Queues for Sequential Processing 3-9
Choosing Permanently Active Queues for Data Persistence 3-9
Using BEA MessageQ Naming 3-10
Using FML for Self-Describing Messaging 3-11
Designing Message Flow and System Configuration 3-11
Advanced Message Queuing Features 3-12
FML Self-Describing Messaging 3-13
Recoverable Messaging 3-14
Message Selection 3-16
Broadcasting Messages 3-17
Naming 3-18
Using Message Based Services 3-20
Exchanging Messages Between BEA MessageQ and BEA TUXEDO V6.4 or BEA M3 V2.1 3-21
Enabling the Messaging Bridge 3-24
Additional API Functions 3-24

Defining a Name-to-Queue Translation at Runtime 3-25
Locating the Queue Address for a Queue 3-25
Using Timers 3-26
Obtaining Detailed Status Information 3-27
Obtaining the Number of Pending Messages in a Queue 3-27
Testing and Debugging BEA MessageQ Applications 3-27
BEA MessageQ Script Facility 3-28
BEA MessageQ Test Utility 3-29
Message Tracing 3-29

4. Managing the BEA MessageQ Environment

Understanding the BEA MessageQ Environment 4-1
Anatomy of a Message Queuing Group 4-3
Starting and Stopping Groups, Queues, Links and the CLS 4-4
Monitoring System Performance 4-5
Error Logging and Recovery 4-5

Glossary

Preface

Purpose of This Document

This document provides an introduction to message queuing, a technique for exchanging information between distributed applications using message queues. This document also describes specific features and benefits of BEA MessageQ.

Who Should Read This Document

This document is intended for the following audiences:

- system installers who will install BEA MessageQ on supported platforms
- system administrators who will configure, manage, and troubleshoot BEA MessageQ on supported platforms
- applications designers and developers who are interested in designing, developing, building, and running BEA MessageQ applications

How This Document Is Organized

BEA MessageQ Introduction to Message Queuing is organized as follows:

- Chapter 1, “What Is BEA MessageQ?”discusses distributed computing, describes basic message queuing concepts, and lists the benefits of message queuing in a distributed computing environment.

- Chapter 2, “Sending and Receiving BEA MessageQ Messages” provides an overview of the MessageQ API functions and describes the processes of configuring MessageQ, attaching to a queue, sending and receiving messages, and detaching from the message queuing bus.
- Chapter 3, “Designing and Developing BEA MessageQ Applications” describes the steps involved in designing, testing, and debugging a MessageQ application. This chapter also describes advanced MessageQ features including self-describing messaging, recoverable messaging, message selection, message broadcasting, and naming services.
- Chapter 4, “Managing the BEA MessageQ Environment” describes how to monitor system performance and troubleshoot errors.
- The Glossary defines terms used in describing messaging in general and MessageQ in particular.

How to Use This Document

This document is designed primarily as an online, hypertext document. If you are reading this as a paper publication, note that to get full use from this document you should access it as an online document via the BEA MessageQ Online Documentation CD. The following sections explain how to view this document online, and how to print a copy of this document.

Opening the Document in a Web Browser

To access the online version of this document, open the `index.htm` file in the top-level directory of the BEA MessageQ Online Documentation CD. Click on the link for the Introduction to Message Queuing.

Note: The online documentation requires a Web browser that supports HTML version 3.0. Netscape Navigator version 3.0 or later, or Microsoft Internet Explorer version 3.0 or later are recommended.

Printing from a Web Browser

You can print a copy of this document, one file at a time, from the Web browser. Before you print, make sure that the chapter or appendix you want is displayed and *selected* in your browser.

To select a chapter or appendix, click anywhere inside the chapter or appendix you want to print. If your browser offers a Print Preview feature, you can use the feature to verify which chapter or appendix you are about to print. If your browser offers a Print Frames feature, you can use the feature to select the frame containing the chapter or appendix you want to print.

The BEA MessageQ Online Documentation CD also includes Adobe Acrobat PDF files of all of the online documents. You can use the Adobe Acrobat Reader to print all or a portion of each document. On the CD's main menu, click the Bookshelf button. On the Bookshelf, scroll to the entry for the BEA MessageQ document you want to print and click the PDF option.

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. *Examples*: 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.

Related Documentation

The following sections list the documentation provided with the MessageQ software, related BEA publications, and other publications related to the technology.

MessageQ Documentation

The MessageQ information set consists of the following documents:

BEA MessageQ Installation and Configuration Guide for Windows NT

BEA MessageQ Installation and Configuration Guide for UNIX

BEA MessageQ Installation Guide for OpenVMS

BEA MessageQ Configuration Guide for OpenVMS

BEA MessageQ Programmer's Guide

BEA MessageQ FML Programmer's Guide

BEA MessageQ Reference Manual

BEA MessageQ System Messages

BEA MessageQ Client for Windows User's Guide

BEA MessageQ Client for UNIX User's Guide

BEA MessageQ Client for OpenVMS Guide

Note: The BEA MessageQ Online Documentation CD also includes Adobe Acrobat PDF files of all of the online documents. You can use the Adobe Acrobat Reader to print all or a portion of each document.

Contact Information

The following sections provide information about how to obtain support for the documentation and software.

Documentation Support

If you have questions or comments on the documentation, you can contact the BEA Information Engineering Group by e-mail at **docsupport@beasys.com**. (For information about how to contact Customer Support, refer to the following section.)

Customer Support

If you have any questions about this version of BEA MessageQ, or if you have problems installing and running BEA MessageQ, contact BEA Customer Support through BEA WebSupport at `www.beasys.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

CHAPTER

1 What Is BEA MessageQ?

Message queuing is a technique for information exchange among distributed applications. Message queues can reside in computer memory or on disk. Message queues store messages until they are read by the receiver program. Through message queuing, application programs can execute independently—they do not need to know each other's location or wait for the receiver program to retrieve the message before continuing.

BEA MessageQ is the industry-leading message queuing product providing connectivity to a broad range of multivendor platforms. This chapter provides an overview of:

- The Distributed Computing Revolution
- Message Queuing Basics
- BEA MessageQ Benefits

The Distributed Computing Revolution

During the last two decades, businesses have increasingly moved computing power out of the data center and into the hands of departments and end users. This trend, called **distributed computing**, has accelerated in the last several years due to the proliferation of powerful and easy-to-use PCs and the advent of high-powered workstations and servers that offer high reliability and failover capability. This section describes:

- Traditional Versus Distributed Applications
- Major Trends in Distributed Computing
- Distributed Computing Models
- Technologies for Building Distributed Applications

Traditional Versus Distributed Applications

An application is defined as a program or set of programs designed to perform a particular business task, for example, a payroll application. Applications can be designed and implemented in one large, monolithic structure or they can be broken into separate components. Application components can be assigned to different processes which work cooperatively to perform the desired tasks. Traditional application design places all application components on a single computer system. Therefore, the component programs can share information easily through global memory and synchronize processing through the features of a single operating system.

Distributed computing, on the other hand, spreads out the processing of component tasks onto several computers tied together by a computer network. Designing a distributed application allows application components to run on different computer systems which can maximize efficient use of computing resources while distributing end user access to a corporate information databases.

For example, a traditional payroll application requires all component tasks to be performed on the same computer system. Therefore, all of a company's business locations would have to send employee payroll information to a central site for data entry, processing, and check printing. A distributed payroll application would allow entry of payroll data, check printing, and check distribution to be handled at different locations throughout the company.

Distributing the payroll application can reduce cost and improve efficiency by:

- Putting the data entry of payroll information closer to its source ensuring greater accuracy and faster problem solving
- Eliminating the processing bottlenecks and inefficiencies of centralized data entry and check distribution

Major Trends in Distributed Computing

Today, distributed computing is revolutionizing the way businesses and individuals process information through:

- Mainframe downsizing—replacing expensive corporate mainframes, with smaller, yet highly reliable departmental servers
- PC LAN upsizing—tying end user PC networks together with corporate databases and application systems
- Integrating existing applications—enabling information exchange between legacy applications to improve data integrity and reduce the cost of data entry

By implementing a distributed approach to application development and integration, companies are:

- Eliminating manual and redundant data entry by sharing data automatically between departmental systems
- Exchanging information between applications at remote sites
- Employing diverse computer hardware for different applications
- Consolidating business operations to reduce redundancies and control cost
- Coordinating application processing to promote efficiency
- Consolidating reporting from different information sources within the company
- Reducing software development overhead
- Providing distributed access to data stored on corporate mainframes
- Connecting hundreds of PCs across the company to share data
- Ensuring reliable exchange of information in a diverse environment

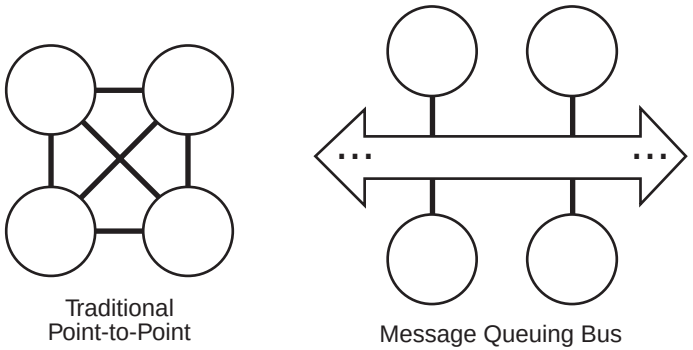
Though distributed processing is very powerful, it is also very complex. Because many different computer systems may be involved in information processing, new issues have arisen in sharing information, synchronizing processing, and sharing results.

The challenge to developers when integrating distributed applications is to provide an efficient means of communication for distributed applications in a heterogeneous networked environment. To manage their information sharing needs, it is most efficient to provide applications with a common mechanism for exchanging information

Middleware is a type of software designed to form a layer between the application and the underlying operating system and network software. It provides applications with a common means of communication and independence from the network and operating system. Middleware provides developers with an application programming interface that is common to all environments. When a function call is embedded in a program, it performs the communication function for the application using the capabilities of the particular operating system and network environment in which it runs.

Figure 1-1 contrasts the middleware approach with the previously used method of linking each individual application in the environment.

Figure 1-1 Contrasting Application Integration Approaches



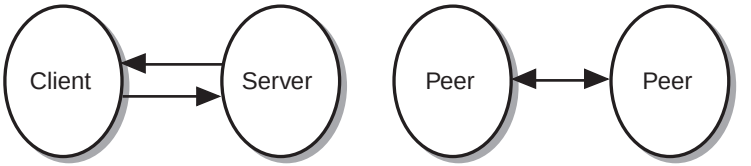
Without middleware to accomplish information exchange, application developers have to write the software for sending and receiving information by learning how to use the features of both network and operating system software to transport the data. And, without a standard approach to information exchange, each application must be programmed to communicate with each and every application in the multiplatform environment.

For example, to exchange information locally on OpenVMS systems, applications can use OpenVMS mailboxes. The same kind of information exchange on a UNIX system would require a knowledge of queues or pipes. Communicating between networked systems requires knowledge of how to exchange information over the network such as TCP/IP socket programming.

Distributed Computing Models

Decomposing an application into its component parts and distributing the parts across disparate computer systems is much more complex than implementing an application on a single system. Software developers use one of two communication models when designing applications to share information in a distributed environment as shown in Figure 1-2.

Figure 1-2 Client/Server versus Peer-to-Peer Information Exchange



- The peer-to-peer model is a conversational style of communication between two applications or application components that exchange information and control as equals.
- The client/server model is a request/response style of communications in which applications are divided into two types of components: those that make requests (clients) and those that fulfill requests (servers).

BEA MessageQ supports both the peer-to-peer or the client/server models of distributed computing.

Peer-to-Peer Communication Model

A message queuing system provides peer-to-peer communication through a standard message-passing mechanism. Communicating programs can operate independently while using the message queuing system to exchange information.

One program initiates communication with a remote program and exchanges messages with it, enabling two-way communication. Data and control information can flow in either direction. And, the communication can be **synchronous** or **asynchronous**. That is, the sender program can wait for a reply from the receiver program or continue immediately.

The peer-to-peer model is used by applications that work cooperatively to process information in a distributed environment. Each program in the distributed application may act as both a requester and fulfiller of service and information requests.

Client/Server Communication Model

The client/server model has emerged during the 1980s as an approach to distributed application design. Using this model, a distributed application is made up of two types of programs: ones that make requests and ones that fulfill requests for services or information.

Client programs require an easy-to-use interface to facilitate user requests for services or information. Because they do not process information, client programs do not need to run on powerful computers. Therefore, they can be designed to run on inexpensive personal computers which offer graphical user interface capabilities. Server programs, on the other hand, must run on faster, more powerful systems such as workstations which can also access large databases of corporate information.

The client/server design models fits well into today's corporate heterogeneous computing environment. With the large amount of PCs distributed throughout the corporation, this model can provide shared and efficient access to corporate information resources with appropriate safeguards.

Technologies for Building Distributed Applications

As with any trend in the computer industry, there is more than one product for building distributed applications. This section describes the three major approaches to distributed application design:

- Remote Procedure Call—one of the standards-based components of the Distributed Computing Environment

- Object Transaction Monitoring—an object-oriented industry standard based on the Common Object Request Broker Architecture (CORBA) combined with transaction processing (TP) monitor technology
- Message Queuing—a loosely-coupled approach to building distributed applications popularized by products from several industry-leading vendors

DCE/Remote Procedure Call

Remote Procedure Call (RPC) is a component of the Distributed Computing Environment (DCE), a software standard for application integration released by the Open Software Foundation. RPCs are modeled after the traditional programming approach where one program invokes another program through a function invocation. The invocation is in the form of a procedure call. Once called, the control of program is given over to the called procedure.

In an RPC implementation, the called procedure resides on and is executed on another system, which can be local or remote. When the called procedure is finished processing the input data, the results are returned to the calling program in the returned arguments of the procedure call. Program control is then returned to the calling program immediately after the RPC is completed.

Since RPCs imitate the call/return structure of a subroutine, they offer only synchronous data exchange between the client (calling program) and the server (called procedure). To overcome this limitation, developers must employ operating system features such as threads or subtasks to force the RPC to process in an asynchronous manner. Using asynchronous RPCs to integrate applications limits portability because the application code has become operating system dependent.

RPCs are best used when an application requires:

- A two-tiered client/server architecture
- Highly interdependent processing of client-to-client or server-to-server interactions
- Uncomplicated, synchronous interaction without the need for high throughput
- Asynchronous processing in a predominantly homogeneous computing environment

Object Transaction Monitoring

The CORBA (Common Object Request Broker Architecture) specification provides a broad and consistent model for building distributed client/server applications by defining:

- An architecture that employs object-oriented technologies and methodologies
- A common client/server application programming interface
- Guidelines for transmitting and translating data among multivendor platforms
- A language for developing distributed application interfaces (Interface Definition Language (IDL))

The CORBA architecture and specification were developed by the Object Management Group (OMG), a consortium of information systems vendors. The goal of CORBA is to promote an object-oriented approach to building and integrating distributed software applications.

The BEA M3 system combines the best of distributed objects and transaction processing (TP) monitor technology into a new platform that is specifically aimed at providing high performance for enterprise distributed object applications using transactions.

The M3 system uses CORBA distributed object technology to provide a common programming model, leveraging from BEA TP monitor technology to provide an enhanced run time by extending the Object Request Broker (ORB) model with online transaction processing (OLTP) functions. The M3 system also leverages from the existing BEA core technology infrastructure for transaction management, security, message transport, administration and manageability, and XA-compliant database support.

Message Queuing

Message queuing offers a loosely-coupled approach to building distributed applications which can be implemented in a synchronous or asynchronous manner. Because messages are application defined, there is no restrictive structure specifying the way in which applications must be written. Instead, messaging API calls are embedded into new or existing application to provide the exchange of information through messages sent to and read from memory or disk-based queues.

Message queuing can be used in applications to perform a variety of functions such as requesting services, exchanging information, or synchronizing processing.

Message queuing is best used when an application requires:

- Asynchronous processing of application components
- Peer-to-peer and/or client/server communication models
- Built-in recoverability for message exchange
- High data interchange rates
- Tight control of the data exchange between a sender and a receiver program

Message Queuing Basics

Message queuing is a technique for sending messages from one program to another by providing an intermediate storage point in computer memory or in a disk file. Messages are stored in message queues until they can be read by the receiver program. By sending and receiving information using message queues, programs can execute independently—they do not need to know each other's location or wait for the receiver program to process the message before continuing. This section describes:

- What Is a Message?
- What Is Message Queuing?
- How Does BEA MessageQ Work?
- Choosing the BEA MessageQ Server or Client
- Key Features of BEA MessageQ

What Is a Message?

A **message** is an application-defined data structure. The application developer defines the content of the message. A message has the following components:

- Data that is defined by the application
- Message attributes that can be used by the application
- Message context defined by the message queuing software that is transparent to the user

Communication through messaging requires both programs to agree upon the type of data in the message and the interpretation of the data. The software that delivers the message ignores its content message; the job of the message queuing system is simply to transport the message data.

What Is Message Queuing?

Message queuing is a technique for sending messages from one program to another by directing messages to a memory- or disk-based queue as an intermediate storage point. The queue stores the messages until they can be processed by the receiver program. By queuing messages, programs can execute independently—they do not need to wait for an application to process a message before continuing.

Message queuing works in the following way:

- Program A makes a call to the message queuing system. The call tells the message queuing system that a message is ready to be sent to Program B.
- The message queuing system sends the message to the system where Program B resides and places it in Program B's queue.
- At the appropriate time, Program B reads the message from its queue and processes the information.

If the system cannot deliver the message because of a communications failure, receiver abort, or system crash, message recovery capabilities enable the message to be re-sent without further application intervention when communication is re-established.

Using message queuing, any program can send messages addressed to any other program that is attached to the message queuing bus. The sender program sends out the messages and can continue processing if an immediate response is not required. For this reason, message queuing adapts well to asynchronous interprocess communication needs.

Message queuing does not require applications to know the structure or state of another application in order to enable communication. As a result, queued communication offers a practical way to integrate applications running in distributed, multivendor environments.

How Does BEA MessageQ Work?

BEA MessageQ is an implementation of a message queuing system. To exchange information using BEA MessageQ, each program must attach to the BEA MessageQ message queuing bus at a particular queue address. The queue address identifies the message queue in which the program receives messages. To send a BEA MessageQ message, a program must know the queue address of the receiver program. In contrast to other message queuing systems, BEA MessageQ applications only attach to a queue in which they will receive messages. They do not attach to the queue to which they send messages.

The BEA MessageQ message queuing bus forms the data highway used to transfer messages between applications by creating a logical interconnection of message queues in a networked environment. Once an application is attached to the message queuing bus at a queue address, it can send messages to any other attached application and is also ready to read messages sent to its own queue or queues.

BEA MessageQ is said to provide a loosely-coupled approach to application integration because applications that share information do *not* have to:

- Know each other's physical location (network address)
- Know how to establish communications between each other
- Be executing at the same time
- Be running on systems with same operating system or network software

A BEA MessageQ message queuing bus is composed of one or more message queuing groups. Message queuing groups offer applications an efficient way to share BEA MessageQ services such as recoverable messaging and message broadcasting on a network of computers. System managers configure cross-group connections to enable applications to exchange information when they are running in different message queuing groups on the same message queuing bus. Each message queuing group is identified by a unique number, the BEA MessageQ group ID. This group ID together with the unique queue number comprise the queue address of each message queue.

BEA MessageQ also allows the configuration of more than one message queuing bus in a networked environment. Applications attached to different message queuing buses cannot communicate with each other. Application developers can use this feature to set up a bus for communication of test programs and another bus for production applications. Mission-critical application processing is then separated from the testing environment. Test programs are easily moved into production by simply changing their bus ID—a configuration step in using BEA MessageQ that is external to the application. This feature may also be used to sparate multiple distributed business applications running on the same network.

Choosing the BEA MessageQ Server or Client

BEA MessageQ provides messaging services to applications running on desktop systems, workstations, mid-range systems, and high-end mainframes. BEA MessageQ offers these messaging services with two types of products:

- BEA MessageQ Servers—for systems with sufficient resources to provide the full range of BEA MessageQ messaging services. A BEA MessageQ Server provides base messaging services, the allocation and management of queues, and the necessary administration tools and utilities to manage a BEA MessageQ message queuing group. BEA MessageQ Servers are offered on workstation, mid-range, and high-end systems including most popular UNIX systems (AIX, HP-UX, Solaris, Tru64 UNIX and others), OpenVMS, and Windows NT.
- BEA MessageQ Clients—a “light-weight”, low cost offering ideal for applications running on systems that do not have the resources to provide full messaging services (such as PCs). The BEA MessageQ Client limits system maintenance overhead and is ideal for deployment of applications that require only base messaging services or are running the desktop client portion of a client/server application. BEA MessageQ Clients are offered across a broad spectrum of systems including Windows 95, Windows NT, UNIX (AIX, HP-UX, Solaris, Tru64 UNIX, and others), OpenVMS, and IBM MVS systems.

All BEA MessageQ environments require the use of at least one message server implementation to offer full message routing to all other BEA MessageQ systems in the network. It is important to note that the terms client and server only refer to the messaging services provided by BEA MessageQ, they do not restrict the types of applications (clients or servers) that can be implemented in a particular environment.

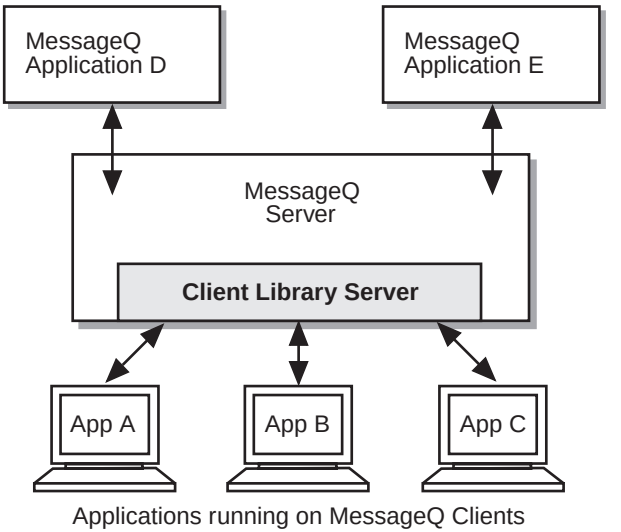
How the BEA MessageQ Client Works

The BEA MessageQ Client is a client implementation of the BEA MessageQ Application Programming Interface (API). It provides message queuing support for distributed network applications along with a BEA MessageQ Server to provide reliable message queuing for distributed multiplatform network applications.

The BEA MessageQ Client is connected to the message queuing bus through a network connection with a Client Library Server (CLS) on a remote BEA MessageQ Server. The CLS acts as a remote agent to perform message queuing operations on behalf of the BEA MessageQ Client. The CLS runs as a background server to handle multiple BEA MessageQ Client connections.

The BEA MessageQ Client establishes a network connection to the CLS when an application attaches to the message queuing bus. The CLS performs all communication with the client application until the application detaches from the message queuing bus. The network connection to the CLS is closed when the application detaches from the message queuing bus. Figure 1-3 shows the BEA MessageQ Server and Client components.

Figure 1-3 How Client Applications Communicate using the CLS



When to Choose the BEA MessageQ Client

The BEA MessageQ Client provides the following benefits:

- Reduces system resource load
- Reduces system management overhead
- Reduces disk space requirements
- Provides network protocol independence

The BEA MessageQ Client provides message queuing capabilities for BEA MessageQ applications using fewer system resources (shared memory and semaphores) and running fewer processes than a BEA MessageQ Server. Therefore, the BEA MessageQ Client enables distributed BEA MessageQ applications to run on smaller, less powerful systems than the systems required to run a BEA MessageQ Server. It also allows for a smaller client footprint for the client part of a client/server application.

Run-time configuration of the BEA MessageQ Client is extremely simple. A minimal configuration requires only the name of the server system, the network endpoint to be used by the CLS, and the desired network transport. Running the BEA MessageQ Client makes it unnecessary to install and configure a BEA MessageQ Server on each system in the network. Instead, a distributed BEA MessageQ environment can consist of a single system running a BEA MessageQ UNIX, Windows NT, or OpenVMS Server and one or more systems running BEA MessageQ Clients.

For example, suppose a small business has 10 networked workstations that need to run a BEA MessageQ application. Without the BEA MessageQ Client, it would be necessary to install, configure, and manage a message queuing group on each workstation. Using the BEA MessageQ Client, however, a BEA MessageQ Server need only be installed and configured on a single workstation. Installing the BEA MessageQ Client on the remaining nine workstations provides message queuing support for all other BEA MessageQ applications in the distributed network.

In this example, only one workstation needs to be sized and configured to optimize performance, reducing the burden of system management to a single machine. System management and configuration for the remaining systems is drastically simplified because managing the BEA MessageQ Client consists mainly of identifying the BEA MessageQ Server that provides full message queuing support. The BEA MessageQ Client can be reconfigured quickly and easily and multiple clients can share the same configuration settings to further reduce system management overhead. This also makes it easy to add additional clients to an application.

The BEA MessageQ Client performs all network operations for client applications making it unnecessary for a client program to be concerned about the underlying network protocol. The BEA MessageQ Client enhances the portability of applications enabling them to be ported to a different operating system and network environment supported by BEA MessageQ with no change to the application code.

Key Features of BEA MessageQ

BEA MessageQ has been recognized by independent industry consultants as the most feature-rich and fastest performing message queuing software available. Its key features are:

- **Recoverable messaging**—guaranteed delivery of a message despite system, process, or network failures
- **Publish and subscribe**—ability to send a message to multiple recipients registered to receive information from a broadcast channel (also called message broadcasting)
- **Naming**—ability to separate application processing from configuration details by allowing applications to refer to queues by name. Name-to-queue address translations are performed by BEA MessageQ at runtime eliminating the need to recode applications when configuration changes are made. This is also called "location independence." BEA MessageQ also allows applications to bind a name to a queue address dynamically at runtime
- **Support for Field Manipulation Language (FML)**—enables applications to encode messages with tags and values that describe the content of the message. The receiver program, therefore, is not programmed to know the exact data structure of the message. Instead, it decodes the message contents using the tag associated with each value. In addition, FML performs data marshaling for applications exchanging information between systems that use different. hardware data formats. FML is also used by BEA TUXEDO.
- **Integration with BEA TUXEDO—**enables BEA MessageQ applications to exchange messages with BEA TUXEDO services and queues. This provides a transparent mechanism for applications to interoperate between BEA MessageQ and BEA TUXEDO.
- **Correlation identifier—**allows a developer to associate a user defined identifier with each message. Applications receiving the message can tag any response to

the message with the same identifier.This feature is useful for asynchronous client/server applications so responses can be matched with associated requests.

- **Message selection**—capability to read messages selectively from queues based on correlation identifier, sequence number, message type, message class, priority, source or a complex set of message attributes
- **Wide array of multiplatform support**—BEA MessageQ runs on every major operating system and hardware platform combination including Windows 95 and Windows NT implementations, all major UNIX versions (AIX, HP-UX, Solaris, and others) and Alpha systems running both Tru64 UNIX and OpenVMS.

BEA MessageQ Benefits

BEA MessageQ facilitates the development of distributed applications by enhancing:

- Productivity—through a standard approach to integration that speeds development, reduces maintenance, and insulates applications from changes in network and operating system software
- Portability—using a single application programming interface for all supported environments so that applications are written once and ported to other systems
- Simplicity—providing a message queuing bus that acts as a single point of communication for all attached applications
- Reliability—with a message recovery system that guarantees delivery in the event of system, process, and network failures
- Interoperability—connecting distributed applications running in all industry-leading environments
- Flexibility—to easily enhance and change applications to meet changing business needs

Standardized Integration Approach

BEA MessageQ is communications middleware that provides software developers with a standard approach to information exchange between distributed applications in a multivendor environment. The BEA MessageQ interface is a set of application programming functions that are common to all BEA MessageQ products. To exchange information with other BEA MessageQ applications, a program simply includes the logic to attach to the BEA MessageQ message queuing bus and send a message and BEA MessageQ figures out how to deliver the message to the system on which the target application's queue resides.

BEA MessageQ API functions can be embedded in new or existing applications. Using BEA MessageQ, application developers no longer need to worry about the underlying transport to send and receive information between applications. In addition, applications no longer require constant maintenance to accommodate changes in operating system and network software. BEA MessageQ also provides productivity tools for developers to test message exchange before all components of the distributed application are complete.

Guaranteed Delivery

BEA MessageQ has built-in message recovery features that enable message delivery in the event of a system, process, or link loss with the network. To guarantee message delivery by BEA MessageQ, an application marks a message as recoverable when it is sent. BEA MessageQ stores each recoverable message in a disk file before sending it to the target queue.

If the message is successfully delivered to the target queue, it is deleted from the disk file. However, if the recoverable message cannot be delivered to the target system due to a system, process, or network failure, BEA MessageQ will automatically resend the messages stored in the disk file at a later time when the failure condition has been resolved.

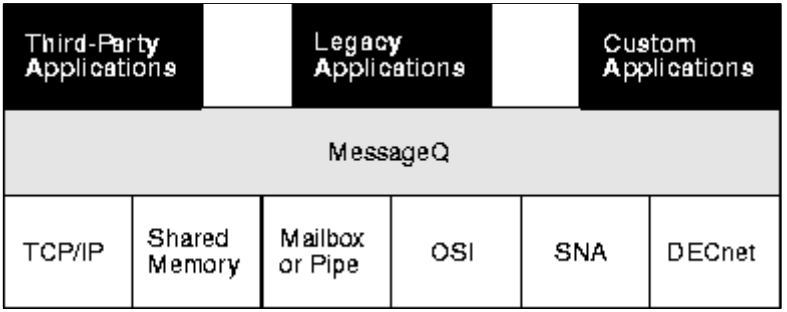
Guaranteed delivery ensures that messages are delivered without further intervention by the sender program. The sender program need only ensure that the message was accepted by the message recovery system in order to be assured that it will be delivered to the target queue.

Application Portability

Because BEA MessageQ uses a common API for all environments, applications move easily using systems from different vendors. For example, if you develop BEA MessageQ applications for Intel PCs running Microsoft Windows NT, the same application programs will run on all major UNIX systems by recompiling and relinking the applications in their target environment.

Figure 1-4 illustrates how the BEA MessageQ API forms a layer between the application and the operating system and network environment—ensuring application portability and shielding applications from changes in underlying software. Note that DECnet is supported only on OpenVMS systems.

Figure 1-4 How the BEA MessageQ API Insulates Applications



Message Bus Simplifies Communication

Message queuing provides a simplified approach to application integration in a distributed multivendor environment. Because BEA MessageQ handles all of the operating system and network-dependent tasks to move a message from one system to another, applications are easier to develop and maintain.

In addition, BEA MessageQ uses a simple application programming interface that consists of four basic callable functions:

- `pams_attach_q`—to attach to the message queuing bus

- `pams_put_msg`—to send a message
- `pams_get_msg`—to retrieve a message
- `pams_detach_q`—to detach from the message queuing bus

Using these four functions, an application program has the ability to exchange information with any other attached application in a distributed, multivendor environment.

Broad Multiplatform Support

BEA MessageQ runs in all industry-leading environments. Refer to Table 1-1 for a listing of BEA MessageQ products illustrating the operating systems supported. TCP/IP networking is supported on all platforms.

Table 1-1 Supported Platform Environments

Product Type	Operating System
Message Server	IBM AIX
	NCR MP-RAS
	Compaq Tru64 UNIX
	Hewlett-Packard HP-UX
	Digital OpenVMS
	Sun Microsystems Solaris
	SCO UnixWare
	SCO OpenServer
	Sequent Dynix/ptx
	Microsoft Windows NT (Intel and Alpha)
Messaging Client	IBM AIX
	NCR MP-RAS
	Compaq Tru64 UNIX

Table 1-1 Supported Platform Environments

Product Type	Operating System
	Hewlett-Packard HP-UX
	SCO UnixWare
	SCO OpenServer
	Sequent Dynix/ptx
	Digital OpenVMS
	Sun Microsystems Solaris
	Microsoft Windows 95
	Microsoft Windows NT (Intel and Alpha)
	IBM MVS
MQSeriesConnection	Hewlett-Packard HP-UX
	IBM AIX
	Sun Microsystems Solaris
	Microsoft Windows NT

Flexibility to Meet Changing Application Needs

BEA MessageQ provides the kind of flexibility applications need to evolve in a rapidly changing application environment through its support of Field Manipulation Language (FML) for self-describing messaging. Using FML, you have a built-in capability to make the following design changes:

- you can add fields to a message that can be read by new applications without disrupting the way existing applications run
- you can change the size of data fields in a message as your needs change without recoding applications because the size of the data field is encoded as part of the message itself

- a single message can be designed to communicate with a variety of applications because the message can be interpreted differently by several applications that use different data fields within the message
- you can design messages to contain information that is not acted upon today, but is part of the future plans of the information system

BEA MessageQ allows you to use double pointers with buffer-style messages (messages using a pre-defined structure agreed upon by the sending and receiving applications). When the receiving application retrieves the message from the queue, the `pams_get_msg` call points to a pointer to dynamically allocated space. This allows for buffer reallocation if the message buffer received is larger than expected. This also means you can change the message structure without having to recode the application.

2 Sending and Receiving BEA MessageQ Messages

The first step in learning how to use BEA MessageQ to exchange information between applications in a distributed environment is to understand how to send and receive BEA MessageQ messages. The following sections describe the basics in sending and receiving BEA MessageQ messages:

- Overview of BEA MessageQ API Functions
- Configuring the BEA MessageQ Environment
- Attaching to the Message Queuing Bus
- Sending a Message
- Receiving a Message
- Using the show_buffer Argument
- Exchanging Messages Between BEA MessageQ and BEA TUXEDO

Overview of BEA MessageQ API Functions

To send and receive messages, application developers embed BEA MessageQ function calls into their applications. After each program is compiled and linked with the BEA MessageQ object libraries, it will be able to send and receive messages.

BEA MessageQ function calls form a portable application programming interface (API). Application programs developed using the C or C++ programming languages need only be recompiled and relinked to enable the messaging functions to work in a different operating system environment.

When applications communicate through message queuing, it is similar to how people communicate using the telephone. Use Table 2-1 to learn how BEA MessageQ API functions are similar to using the telephone to communicate.

Table 2-1 Description of Key PAMS API Functions

Using the API Function...	Is like...	Because...
`pams_attach_q`	Picking up the telephone	Exchange of information requires access to a common means of communication between yourself and the person you want to talk to. When you pick up the telephone receiver and hear a dial tone, you can talk to anyone who is connected to the telephone system. Similarly, your application uses the `pams_attach_q` function to connect to the BEA MessageQ message queuing bus. Attaching to the message queuing bus provides the application with a queue address for receiving messages and a means to share information with all other BEA MessageQ applications.

Table 2-1 Description of Key PAMS API Functions

Using the API Function...	Is like...	Because...
`pams_put_msg`	Dialing a number and talking	After you decide who to call and what to say, you dial the person's telephone number and start talking. To send a message using BEA MessageQ, an application uses the `pams_put_msg` function to send a message to the queue address of the receiver program. BEA MessageQ queue addresses contain two parts, the group number and the queue number. Message queuing groups are like area codes providing a localized grouping of telephone numbers. The queue number is like the telephone number providing the "address" for directing the call to the party you want to speak with.
`pams_get_msg`	Answering a phone call and listening	When your telephone rings, you pick up the receiver and listen to the caller. Similarly, BEA MessageQ applications use the `pams_get_msg` function to retrieve messages from their queue.
`pams_detach_q`	Hanging up	When you are finished talking on the telephone, you hang up. Similarly, if two BEA MessageQ applications are finished exchanging information, they use the `pams_detach_q` function to disconnect from the message queuing bus.
`pams_locate_q`	Using directory assistance	When you remember someone's name but not their telephone number, you call directory assistance. BEA MessageQ applications use the `pams_locate_q` function to obtain a queue address for a queue name at runtime.

Table 2-1 Description of Key PAMS API Functions

Using the API Function...	Is like...	Because...
`pams_get_msg` with selection criteria	Screening calls	Sometimes you may not want to receive all of your calls, so you answer only those that meet particular criteria, such as urgent calls or calls about a particular subject. BEA MessageQ applications can assign characteristics to a message when it is sent so that the receiver program can choose which messages to read. Receiver programs can read messages based on their source, priority, message type, or message class using the `pams_get_msg` function. However, if messages must be selected using a complex set of selection criteria, a selection mask can also be specified using the `pams_set_select` function.
`pams_put_msg` with a recoverable delivery mode	Calling someone with an answering machine	People are not always available when you call them; however, you can be sure they will get your message if they use an answering machine. If they have been away from their telephone, they can replay the messages stored on the tape of the answering machine to obtain their phone messages. Similarly, BEA MessageQ applications can send messages with a recoverable delivery mode using the `pams_put_msg` function. Recoverable messages that cannot be delivered are stored on disk and resent when the receiver program becomes available.
`pams_put_msg` with a broadcast target	Conference calling	Sometimes you need to give several people the same information but you do not want to have to call each person individually. In this case, you hold a conference call to tell everyone the same thing at the same time. BEA MessageQ applications can broadcast a message to many receiver programs at once using a single call to the `pams_put_msg` function using a broadcast target

Configuring the BEA MessageQ Environment

Before you can use the BEA MessageQ message queuing system, you must configure the BEA MessageQ environment. The following topics describe the basics in configuring and starting a message queuing group:

- Define message queues and their attributes
- Setting up the message queuing buses, groups, and queues

Defining Queues and Their Attributes

To use BEA MessageQ, a sender or receiver program must be associated with at least one message queue in which it can receive messages. To become associated with a queue, the sender or receiver program invokes the `pams_attach_q` function to attach to a queue on the message queuing bus.

When designing your application, you need to select attributes of each message queue. Message queues are created and used differently depending upon the combination of attributes selected for each queue. For example, answer the following questions to help you design your message queuing environment:

- Is the need for the queue only temporary, such that it can be created during processing and deleted when it is no longer needed?
- Should the queue be permanently defined so that applications can reference it by name?
- Will messages in the queue be read by a single program or by multiple programs?
- Will the queue receive primary application messages or will it be used to exchange information that is secondary to application processing?
- Does the sender program need to be able to send messages to the queue even if no process is currently attached to it?

- Will the queue be used to store recoverable messages?

BEA MessageQ offers two types of queues: temporary and permanent. Temporary queues are created by BEA MessageQ at runtime when they are requested using the `pams_attach_q` function. Applications use temporary queues when the need for the queue is short lived.

Permanent queues must be defined in the group initialization file. Permanent queues can become active when the group starts or when an application attaches. Applications use permanent queues when there is an ongoing need for the queue to service the application and when applications need to refer to the queue by name or number.

After you have selected the type of queue to use, you must set the following attributes of the queue:

- primary or secondary
- single reader or multireader
- active on attach or permanently active

Each process that attaches to the BEA MessageQ message queuing bus must have a primary queue assigned to it. This queue functions as the "main mailbox" for receiving messages from other processes using BEA MessageQ. In addition, BEA MessageQ applications can use secondary queues as a means of exchanging information among application components without interrupting the flow of messages taking place in the primary queue. In this way, secondary queues are used by application processes as an alternate "mail box" for selected application messages.

Applications can be designed to read messages from one or more queues. Queues defined to be read by a single program are called single reader queues. When a process attaches to a single reader queue, it owns the queue and is the only process that can read from the queue. Queues that are designed to be read by multiple applications are called Multireader queues (MRQs). MRQs are used to store messages that can be read by many simultaneous readers, creating a central "mail box" for several applications or application components to receive messages. Only permanent queues can be defined as MRQs. In addition, MRQs must have the attribute permanently active.

When defining permanent queues in the group initialization file, you have the choice of determining whether the queue becomes active when a process attaches to the queue or if the queue is active when the groups starts up regardless of whether any process is attached. Permanently active queues provide the maximum data persistence for messaging data.

Queue configuration procedures vary based on whether the queue is defined as temporary or permanent as follows:

- Temporary queues do not require any configuration procedure. They are created by BEA MessageQ at runtime when a process requests attachment to a temporary queue using the `pams_attach_q` function. Temporary queues are single-reader queues only. The process that attaches to the queue is the owner of the temporary queue and no other processes can read from the queue. An additional argument of the `pams_attach_q` function allows you to specify whether the temporary queue is a primary or secondary queue. By default, if this argument is not specified, BEA MessageQ uses the first queue to which an application attaches as its primary queue.

- Permanent queues are defined in the initialization file of a BEA MessageQ message queuing group. Each permanent queue is designated with a number and, sometimes a name, which is part of the definition of the group when it starts up.

  A permanent queue is created by BEA MessageQ in one of two ways. First, it can be active on attach which means that is created when a process attaches to the message queuing bus at that queue address. Once a process is attached, permanent queues are available to store messages from sender programs.

  Or, secondly, you also have the option to define permanent queues with the attribute permanently active (always writable). In this case, the queue is not only part of the group definition, but it is actually created when the group starts up. Therefore, permanently active queues can store messages when no process is attached.

The Queue Configuration Table in the group initialization file enables you to specify the following queue characteristics:

- primary or secondary
- single-reader or multireader queue
- active on attach or permanently active (MRQs must be permanently active)
- queue quotas
- MRS attributes such as explicit/implicit confirmation and confirmation order

For a detailed description of how to configure message queues, refer to the administrator's guide for the BEA MessageQ product that you are using.

Configuring Buses, Groups and Queues

Now that you understand the types of message queues you can define, you are ready to begin configuring your BEA MessageQ environment in three simple steps:

- Step 1: Design the message queuing environment
- Step 2: Configure each message queuing group
- Step 3: Start each message queuing group and change configuration data at runtime

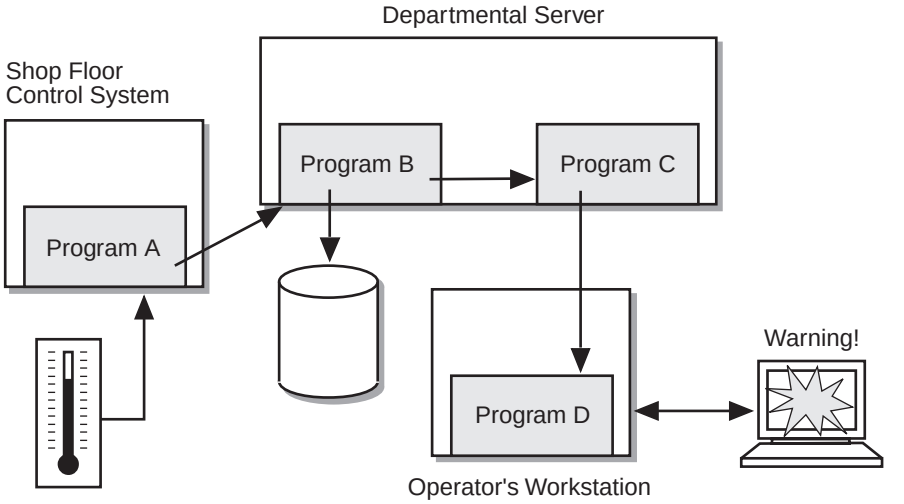
Designing Your BEA MessageQ Environment

The design of your application determines your BEA MessageQ configuration, therefore, you must begin by mapping out:

- the application components that will send and receive information
- which message queues must be created and what attributes they require
- the networked systems on which the applications will run

For example, let's take a look at Figure 2-1 which illustrates the design of a shop-floor monitoring application.

Figure 2-1 Sample BEA MessageQ Application



Program A reads temperatures from a smelting furnace, formats the temperature data as a BEA MessageQ message and sends it to the primary queue of Program B. Program B stores the temperatures in a database from which it can generate graphical charts and reports on demand. Program B also forwards each temperature to Program C for analysis.

Program C reads each message to analyze the temperature reading in the smelting furnace checking to see that it is not outside of the accepted range for the manufacturing process. If the temperature of the furnace becomes too hot or too cold, Program C forwards the temperature message to the primary queue of Program D. Program D displays a temperature warning to alert the shop-floor operator of a potential problem.

As part of the design, you must determine how the application components can be most efficiently deployed into the distributed environment. In this example, the following configuration of networked computer systems are required to support the application:

- Program A runs on a real-time computer system connected to the temperature sensing equipment for the smelting furnace.
- Programs B and C run on a departmental minicomputer which records and analyzes information related to the manufacturing process.
- Program D runs on a shop-floor supervisor’s workstation.

To configure the BEA MessageQ environment to support this example shop-floor monitoring application, you need to define:

- 1 message queuing bus
- 3 message queuing groups (1 for each different system)
- 4 permanent message queues (1 for each application)

Configuring Each Message Queuing Group

To configure a message queuing group, BEA MessageQ uses an ASCII text file called the group initialization file. A sample initialization file is distributed with the BEA MessageQ media kit to illustrate a simple group configuration. To define queues, set their characteristics, and add resources, you make a copy of the template file and then edit the new file using a text editor to create the desired group configuration. Each message queuing group requires its own initialization file.

The major steps in configuring a message queuing group are defining:

- Permanent queues and their attributes
- Cross-group connections between networked computer systems running applications that need to exchange information using BEA MessageQ. You may also need to define message routing for systems with no direct network connection
- Whether recoverable messaging will be used within the group and whether successfully delivered recoverable messages will be stored in a journal
- Whether the group supports message broadcasting
- How parameters are set to regulate message flow and how quickly messages are processed
- Whether a Client Library Server is configured for the group to support communication with one or more BEA MessageQ Clients

For step-by-step instructions on how to configure a BEA MessageQ message queuing group, refer to the administrator's guide for the platforms used in your environment.

Starting Each Message Queuing Group

Once you have created and configured the characteristics of a message queuing group, you invoke the BEA MessageQ startup procedure to start the group. The startup procedure reads the information in the group initialization file in order to configure the group. The startup procedure performs all of the tasks in starting the group including defining the needed queues, names, cross-group connections, and starting the appropriate BEA MessageQ servers to support such features as recoverable messaging and message broadcasting. The procedure for starting a group varies by platform, therefore, you should refer to the installation and configuration guide for your platforms to obtain specific instructions for starting a group.

Once a message queuing group is running, you can change some of the group's characteristics without having to shut down the group and restart it using the Loader utility. After you edit the group initialization file, you can invoke the Loader utility to update the characteristics of the group at runtime.

Attaching to the Message Queuing Bus

To enable message exchange, BEA MessageQ application programs must call the `pams_attach_q` function to attach to a queue on the message queuing bus in which to receive messages. Once attached, the application is free to send messages to any queue on the message queuing bus. The application receives messages in one of the queues to which it is attached. BEA MessageQ does not require an application to attach to a queue to which it will send messages.

The `pams_attach_q` function enables applications to specify an attachment point in the form of a queue name, a queue number, or by requesting the use of a temporary queue. The type of attachment is specified by supplying one of three constants as the **attach_mode** argument:

- `PSYM_ATTACH_BY_NAME`—attaching by name
- `PSYM_ATTACH_BY_NUMBER`—attaching by number
- `PSYM_ATTACH_TEMPORARY`—attaching as a temporary queue

In addition to specifying the attachment type, the `q_type` argument can be used to specify whether the queue should serve as the primary queue or the secondary queue for the application. Additional arguments may be required based on the type of attachment selected.

When the `pams_attach_q` function successfully completes, the `queue_address` argument returns the BEA MessageQ queue address for communicating through the message queuing bus to the application. In addition, this function now includes a `timeout` argument to set a time limit for the attach operation after which control returns to the sender program.

Following are the rules of attachment for BEA MessageQ applications:

- Each program must attach to one queue as its primary queue.
- The primary queue can be a temporary queue that is assigned at runtime or it can be a permanent queue that is defined in the group initialization file as active on attach or permanently active.
- The primary queue can be a single reader queue or a multireader (MRQ) queue on Windows NT and UNIX systems. On OpenVMS systems, the primary queue cannot be an MRQ.
- Single reader queues (primary and secondary) are owned by the process that is attached to the queue. Only that process can read from the queue.
- When an application is attached to a primary queue that has secondary queues defined in the initialization file, the application becomes implicitly attached to the secondary queues after it attaches to the primary queue.
- An application can explicitly attach to a queue as a secondary queue when the queue is not associated with a primary queue.
- An application can have an MRQ as its primary or secondary queue on Windows NT and UNIX systems. On OpenVMS systems, an application cannot attach directly to an MRQ, but reads from the MRQ using a selection filter.
- MRQs are defined in the queue configuration table of the group initialization file by setting the queue type to M. Temporary queues cannot be used as MRQs. MRQs must be defined as permanently active because they must be defined in the group initialization file.

Attaching by Name

When you select the PSYM_ATTACH_BY_NAME option, you must specify:

- the name of the queue to which the application should attach using the q_name argument.
- the number of characters in the queue name using the q_name_len argument.

By default, when attaching by name, the queue name must be configured in the group initialization file of the group in which the application is running. To specify wider search criteria, the application can use the name_space_list argument to specify a list of name tables for BEA MessageQ to use in looking up the queue name. If you use the name_space_list argument, you must use the name_space_list_len to specify the number of entries entered using the name_space_list argument.

Attaching by Number

When you select the PSYM_ATTACH_BY_NUMBER option, you must specify:

- the number of the queue to which the application should attach using the q_name argument. The queue number is specified as an ASCII text string of 4 numeric characters.
- the number of characters in the queue number using the q_name_len argument.

To attach to a queue by number, the queue must be configured in the group initialization file of the group in which the application is running.

Attaching to a Temporary Queue

If you select the PSYM_ATTACH_TEMPORARY option, you can also use the q_type argument to specify whether the temporary queue should serve as the primary or secondary queue for the application. When the queue address is returned, the application uses the temporary queue until its task is complete. Once the application detaches the temporary queue, the messages in the queue are deleted and the queue address is made available for other applications to attach as temporary.

Sending a Message

Applications use the pams_put_msg function to send a message to the target queue of a receiver program. To send a message, the application developer must know:

- Queue address of the target queue

  The target argument is used to specify the queue address of the message queue to which the message is being sent. Each sender program can be developed to send messages directly to a queue using its queue address (group ID and queue number). However, if the configuration of the environment changes, the application will have to be recoded with the new queue address.

  Many applications are developed to reference queues by name or number. In this case the application must call the pams_locate_q function to obtain the queue address for the queue number or name at runtime. Then, the application passes the queue address to the pams_put_msg function.

- Style of messaging to be used

  Application developers can choose buffer-style or FML-style messaging. Buffer-style messaging exchanges information between sender and receiver programs using a predefined message structure. The message is created in an application buffer specified in the msg_area argument of the pams_put_msg function. FML-style messaging uses Field Manipulation Language for self-describing messaging and passes a pointer to an FML32 buffer. (BEA MessageQ supports FML32, the 32-bit version of FML.) FML automatically marshals data among heterogeneous machines. See the *BEA MessageQ FML Programmer's Guide* for more information on FML.

  The msg_size argument is used to specify the style of messaging. This argument contains either the size of the static buffer-style message contained in the msg_area argument, the symbol PSYM_MSG_FML, indicating that the data contained in the msg_area argument is an FML32 buffer, or the symbol PSYM_MSG_LARGE, indicating that the message is a buffer larger than 32K. For large messages, the pointer to the message is contained in the msg_area argument and the size of the large buffer is contained in the large_size argument.

- Priority of the message (0-99)

  The `priority` argument designates the priority of the message. Priorities range from 0 to 99. The larger the value, the higher the priority of the message. Higher priority messages are stored nearer to the top of the queue than lower priority messages. Messages are read in FIFO (first-in, first-out) order within a priority value.

- Message type and class to identify the content of the message

  The message `type` and `class` arguments are used to specify unique descriptors identifying the content of the message. Receiver programs can selectively read messages from their queue based on the type and class argument specified for the message.

- Delivery mode and appropriate error handling for the message

  The `delivery` argument of the `pams_put_msg` function determines how the message is delivered and whether the message is designated for guaranteed delivery if a system, process, or network fails. Recoverable messages are stored on disk by the message recovery system until they can be delivered to the target queue of the receiver program. When sending a recoverable message, you must specify the Undeliverable Message Action `uma` argument to determine the action to be taken if the message cannot be delivered to the delivery interest point. You must also supply the PAMS Status Block `psb` argument to receive the success or failure status of the operation. For non-recoverable messages, the default UMA is `DISC` (discard). However, you can use the `RTS` (return-to-sender) and `DLQ` (dead letter queue) UMAs to use BEA MessageQ recovery mechanisms in the event that the message cannot be delivered.

- Timeout requirements

  When using blocking (WF) delivery modes, application developers should use the `timeout` argument to specify the maximum amount of time the `pams_put_msg` function waits for a message to be delivered before returning control to the application. The timeout value is entered in tenths (0.1) of a second. A value of 100 indicates a timeout of 10 seconds. If the timeout occurs before a message is delivered, then `PAMS__TIMEOUT` is returned. Setting this argument to zero indicates the default setting of 30 seconds.

- Designated response queue

  By default, the receiver program will return its response to the primary queue of the sender program. This queue address is supplied by BEA MessageQ as the source argument. Optionally, the sender program can specify the `resp_q` argument identifying an alternate queue for receiving response messages. When the `resp_q` argument is supplied, the receiver program returns its response to the queue address specified by this argument.

- Correlation ID

  The `correlation_id` parameter allows you to associate a user-defined identifier called a correlation ID with each message. Receiving applications can retrieve the correlation ID and tag any response to the original message with the same ID. This allows applications to send multiple requests and then track responses to those requests by matching their correlation ID.

Selecting a Messaging Style

BEA MessageQ enables applications to send messages using two messaging styles:

- Message buffers—applications exchange information by passing data contained in a static message buffer. Buffer-style messaging requires the structure and format of the message buffer contents to be agreed upon in advance by sender and receiver program. If any changes are made to the message data structure, all application programs must be changed accordingly. Buffer-style messaging uses two different approaches based on whether the message is up to 32K in size, or is a large message of up to 4MB in size.

  You can increase the flexibility of message buffers by using double pointers. When the message is read from the queue, the receiving application points to a pointer which in turn points to a message buffer. This allows for automatic buffer reallocation and for the use of different message data structures without recoding the application. Static message buffers are not manuipulated in any way by BEA MessageQ.

- FML32 buffers—applications exchange information by passing a pointer to an FML32 buffer. This messaging style provide a way to separate the BEA MessageQ message from its contents.

  FML uses fielded buffers to provide self-describing messaging, an approach that allows application developers to encode the contents of the message so that it

can be interpreted by the receiver program without prior knowledge of the detailed message structure.

Self-describing messaging adds a dimension of flexibility in message exchange because it allows the components of the message data structure to be changed without affecting existing applications unless the new information is needed. When using FML32 buffers, data is automatically marshalled among heterogeneous machines.

Using Buffer-Style Messaging

Sending and receiving information as message buffers is the easiest way to exchange information using BEA MessageQ. A message buffer is a predefined, static data structure that is identified using a version number. So, for example, when a payroll system sends employee payroll information using version 1 of its payroll data structure, the receiving application can interpret each field of data in the buffer because it knows the definition of the version 1 payroll data structure.

Passing information using a static data structure in the form of a message buffer is the fastest way to exchange information between applications. Because the data structure definition is known to both the sending and receiving applications, no interpretation is required. Therefore, processing of information between both sender and receiver programs is faster.

However, message buffers limit the flexibility of applications to adapt to changing business conditions. To change the data structure, both the sender and receiver programs must be recoded to send and interpret the new message correctly. In addition, all production applications must be shutdown and the newer versions started up for the change to take affect. Such large changes to an integrated application environment often result in synchronization problems where some applications have not yet been restarted using the new message format. This leads to processing errors until all applications are using the same version of the message data structure.

Message buffer flexibility can be enhanced by using the `PSYM_MSG_BUFFER_PTR` symbol. When this symbol is supplied in the `msg_area_len` parameter of the `pams_get_msg(w)` function, the receiving application points to a pointer which in turn points to dynamically allocated space. The message buffer received is placed in the allocated space. This double pointer feature allows the use of different message data structures without recoding the application.

Another limitation in using message buffers is that data is passed "as is" from one system to another in the network. So, if a message must be delivered between two computers that use different byte orders, the application must perform the byte order translation to ensure that the data is interpreted properly by the target application. BEA MessageQ does not perform data marshaling between systems with unlike data formats when messages are sent using the message buffer approach.

Using FML-Style Messaging

A pointer identifies the FML32 message buffer to process. Instead of passing a message buffer containing the message data, the application passes a message pointer to the `pams_put_msg` and `pams_get_msg` functions identifying the message buffer to process.

To use FML-style messages, the sender program begins by specifying `PSYM_MSG_FML` in the `msg_size` parameter of the `pams_put_msg` function. This indicates that the message is formatted as an FML32 buffer.

The `pams_get_msg` function will return an FML32 buffer in the `msg_area` field. When the application receives an FML-style message, the `msg_size` parameter contains `PSYM_MSG_BUFFER_PTR`, and the `msg_area` field contains a pointer to a pointer, which in turn points to an FML32 buffer.

The use of FML-style messages can provide flexibility in application development and design because the FML32 buffer hides the message structure from the sender and receiver programs. FML enables developers to include encoded message contents with information that identifies the content and format of the information for use by the receiver program.

Choosing a Delivery Mode

Sender programs must specify a **delivery mode** for each message sent. The delivery mode determines:

- Whether the sender program uses synchronous or asynchronous message delivery
- Whether the sender program receives notification of message delivery

- The delivery interest point (the point in the message flow to which BEA MessageQ tracks the outcome of delivery)

The delivery mode is specified as an argument to the `pams_put_msg` function using the following BEA MessageQ symbolic constant:

`PDEL_MODE_`*`sn_dip`*

where:

`sn` is the sender notification, and

`dip` is the delivery interest point.

In addition to the delivery mode, BEA MessageQ also allows sender programs to specify an Undeliverable Message Action (UMA) to determine how the message should be handled if it cannot be properly delivered.

The delivery mode argument specifies whether the message is sent using recoverable or nonrecoverable message delivery. Messages sent using recoverable delivery modes are stored on disk by BEA MessageQ for automatic redelivery in the event of process, system, or network failures. Messages sent using nonrecoverable delivery modes are used by applications that do not require automatic recovery in the event of message delivery failure, or which must perform recovery themselves. Nonrecoverable messages are not stored on disk by BEA MessageQ and cannot be resent in the event of delivery failure without application intervention.

Sender Notification

The sender notification portion of the delivery mode argument specifies whether the sender program uses a blocking (synchronous) or nonblocking (asynchronous) style of message delivery and whether it receives notification of message delivery. BEA MessageQ uses the following sender notification codes:

- `AK` (Asynchronous delivery with a notification message)—indicates that the sender program uses a non-blocking style of message delivery and receives notification of message delivery to the delivery interest point in an asynchronous acknowledgment message. The asynchronous acknowledgment message is delivered to the primary or response queue of the sender program as a message of type `MRS_ACK`. Receipt of an `MRS_ACK` message by the sender program reports delivery of the message to the delivery interest point.

  AK sender notification supports higher messaging rates than synchronous delivery while still providing notification of delivery. This delivery style

supports a loosely coupled approach to application integration while supporting the sender program's need to receive acknowledgment of successful or unsuccessful message delivery.

- NN (Asynchronous delivery with NO notification message)—indicates that the sender programs uses a non-blocking style of message delivery and does not receive notification of message delivery to the delivery interest point. NN sender notification supports datagram-style delivery which supports high messaging rates because processing is asynchronous and there is no additional processing required for each message.
- WF (Synchronous delivery)—indicates that the sender program uses a blocking style of message delivery and does not continue processing until the message is received at the delivery interest point.

  WF sender notification is used by applications which require knowledge that message delivery has succeeded to a selected delivery interest point before it can continue processing. This delivery style supports more highly interdependent message processing between sender and receiver programs.

Delivery Interest Point

The message flow is the path between the sender and receiver program that a message will traverse. There are certain points in the message flow that can provide significant indication of the success or failure of the message delivery.

The delivery interest point portion of the delivery mode argument is used to determine the point in the message flow at which the sender program can unblock (if using WF mode) or the point at which the asynchronous acknowledgment message is sent (if using AK notification).

The BEA MessageQ delivery interest point determines the point at which the sender program unblocks or receives asynchronous notification as follows:

- ACK—when the receiver program explicitly acknowledges receipt of a nonrecoverable message using the pams_confirm_msg call
- CONF—when the receiver program explicitly acknowledges receipt of a recoverable message using the pams_confirm_msg call
- DEQ—when a nonrecoverable message is read from the target queue

- DQF—when a recoverable message is stored in the recovery journal on the remote system (DQF)
- MEM—when a nonrecoverable message is stored in memory in the target queue
- SAF—when a recoverable message is stored in the message recovery journal on the local system (SAF)

Table 2-2 and Table 2-3 describe the nonrecoverable and recoverable delivery modes.

Table 2-2 Nonrecoverable Delivery Modes

Delivery Mode	Explanation
PDEL_MODE_AK_ACK	The sender program sends the message, continues processing, and receives an MRS_ACK message when the receiver program explicitly acknowledges receipt of the message using pams_confirm_msg.
PDEL_MODE_AK_DEQ	The sender program sends the message, continues processing, and receives an MRS_ACK message when the receiver program reads the message from the target queue.
PDEL_MODE_AK_MEM	The sender program sends the message, continues processing, and receives an MRS_ACK message when the message is stored in the target queue.
PDEL_MODE_NN_MEM	The sender program sends the message, continues processing, and does not receive notification of message delivery.
PDEL_MODE_WF_ACK	The sender program sends the message and then blocks until the receipt of the message is explicitly acknowledged by the receiver program using the pams_confirm_msg function..
PDEL_MODE_WF_DEQ	The sender program sends the message and then blocks until the message is read from the target queue.
PDEL_MODE_WF_MEM	The sender program sends the message and then blocks until the message is stored in the target queue.

Table 2-3 Recoverable Delivery Modes

Delivery Mode	Explanation
PDEL_MODE_AK_CONF	The sender program sends the message, continues processing, and receives an asynchronous acknowledgment message when the receiver program reads and explicitly confirms receipt of the message using the pams_confirm_msg function.
PDEL_MODE_AK_DQF	The sender program sends the message, continues processing, and receives an asynchronous acknowledgment message when BEA MessageQ successfully stores the message in the remote message recovery (DQF).
PDEL_MODE_AK_SAF	The sender program sends the message, continues processing, and receives an asynchronous acknowledgment message when BEA MessageQ successfully stores the message in the local recovery journal (SAF).
PDEL_MODE_NN_DQF	The sender program sends the message and continues processing. This delivery mode indicates that the message should be stored in the recovery journal of the remote system if it cannot be delivered though the sender program does not require notification that the message was stored in the DQF.
PDEL_MODE_NN_SAF	The sender program sends the message and continues processing. This delivery mode indicates that the message should be stored in the recovery journal of the local system if it cannot be delivered though the sender program does not require notification that the message was stored in the SAF.
PDEL_MODE_WF_CONF	The sender program sends the message and blocks until the message has been received and confirmed by the receiver program.
PDEL_MODE_WF_DQF	The sender program sends the message and blocks until the message is stored in the remote message recovery journal (DQF).
PDEL_MODE_WF_SAF	The sender program blocks until the message is stored in the local message recovery journal (SAF).

Undeliverable Message Action

The pams_put_msg function enables application developers to specify an Undeliverable Message Action (UMA) for both nonrecoverable and recoverable messages. If the UMA for a nonrecoverable message is not specified, BEA MessageQ uses the default UMA DISC. For recoverable messages, the UMA must always be specified.

The UMA, in conjunction with the delivery mode, gives developers the ability to precisely determine how a message should be sent and what to do if the message cannot be delivered. The UMA is taken if the message does not reach the delivery interest point for both recoverable and nonrecoverable messages.

The UMA is specified as an argument to the pams_put_msg function using the following BEA MessageQ symbolic constant:

PDEL_UMA_*xxx*

where *xxx* is one of the following valid UMAs:

- DISC (Discard)—the message is discarded.
- DISCL (Discard and Log)—the message is discarded and the event is recorded in the BEA MessageQ log file. This UMA is available only on OpenVMS systems.
- RTS (Return-to-Sender)—the message is returned to the primary queue or response queue of the sender program.
- DLQ (Dead Letter Queue)—the message is sent to the Dead Letter Queue (DLQ) which is preconfigured in each message queuing group. Message stored in the DLQ can be resent at a later time under program control.
- DLJ (Dead Letter Journal)—the message is stored in the Dead Letter Journal (DLJ) The DLJ file is configured for each message queuing group for which message recovery services is enabled. Messages stored in the DLJ can be resent at a later time under user or program control.
- SAF (Store-and-Forward File)—the message is stored in the local message recovery journal. The message will be sent automatically by BEA MessageQ when the failure condition is resolved.

For a complete description of how to select the UMA appropriate for your application, refer to the *BEA MessageQ Programmer's Guide.*

Receiving a Message

The pams_get_msg function retrieves the next available message from a selected queue and moves it to the location specified in the msg_area argument. When an application reads a message from a queue, the message is moved from the queue into a data buffer defined by the program. Once read, the message no longer exists in the queue.

Messages are read from queues in first-in/first-out (FIFO) order within a priority. Higher priority messages are read before lower priority messages.

BEA MessageQ provides following functions for retrieving messages:

- pams_get_msg—retrieves the next available message from a specified queue and moves it to the location specified in the msg_area argument.
- pams_get_msgw—retrieves the next available message from a specified queue, however, if the queue is empty, this function waits until a message arrives in the queue or a user-specified **timeout** period has elapsed.
- pams_get_msga—provides a mechanism for writing interrupt-driven code on OpenVMS systems only. The pams_get_msga function is a special form of the get message operation that allows multiple asynchronous read operations with full selective reception. BEA MessageQ interrupts the application when a message enters the queue, and executes the action_routine specified in the call. The pams_cancel_get function cancels all pending pams_get_msga requests that match the selection filter.

Confirming Receipt of a Message

When a receiver program reads a message from its queue, it checks the PSB Delivery Status field to see if the message requires explicit confirmation using the pams_confirm_msg function. Nonrecoverable messages sent using the ACK delivery interest point and recoverable messages using the CONF delivery interest point require explicit confirmation. In addition, recoverable messages sent to a queue that is configured for explicit confirmation must be confirmed using the pams_confirm_msg function.

For recoverable messages, the BEA MessageQ message recovery system retains the message until delivery is confirmed. The receiver program must use the `pams_confirm_msg` function to remove successfully delivered recoverable messages from the message recovery journal. The message recovery system attempts redelivery of recoverable messages from the recovery journal each time the target queue detaches and reattaches to the message queuing bus.

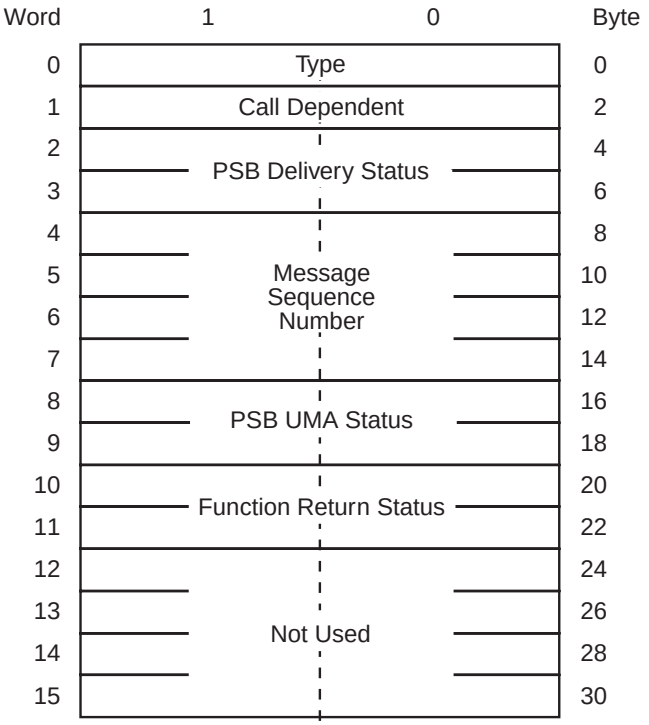
The receiver program reads the PSB delivery status of each message to know which messages to confirm. A PSB delivery status of `PAMS__CONFIRMREQ` indicates that the message requires confirmation. A PSB delivery status of `PAMS__POSSDUPL` also requires confirmation to delete the message from the message recovery system.

Using the PAMS Status Buffer

Applications should check the PSB Delivery Status field of each message to determine if an explicit confirmation is required. A recoverable message that is read from a queue that has the explicit confirmation attribute set requires explicit confirmation. In such situations the receiver program must call the `pams_confirm_msg` function. This function deletes the message from the message recovery journal disk storage. If receipt of a recoverable message is not confirmed, the message continues to be stored by the recovery system and will be redelivered if the application detaches and then reattaches to the queue.

The PSB also contains the Delivery Status Field and UMA Status field which can provide additional information about the successful or unsuccessful completion of an operation. Figure 2-2 illustrates the contents of the PSB.

Figure 2-2 PAMS Status Buffer



Using the show_buffer Argument

The `show_buffer` argument of the `pams_get_msg(w)` function allows you to retrieve additional information, including the message's correlation ID and BEA TUXEDO `urcode` (user return code) when receiving a message. When the optional `show_buffer` argument is specified, the following information is returned:

- the version of the `show_buffer` structure
- the transfer status (success, buffer overflow, or no information to transfer)
- the number of bytes transferred to the application buffer

- a bit field representing which data has been set in the show_buffer (the BEA TUXEDO `urcode` and correlation ID are not associated with a message)
- the BEA TUXEDO `urcode` when exchanging messages between BEA MessageQ and BEA TUXEDO
- the q_address of the latest message target, the original message target, the original message source, and the original message source
- the delivery mode used to queue the message
- the priority used to queue the message
- the byte ordering or encoding schemes for 2- and 4-byte integers or FML buffers
- the correlation ID

Using Message Classes with BEA MessageQ and BEA TUXEDO

New symbolic names for message class values are defined in the `p_typecl.h` include file for use in distinguishing messages received from BEA TUXEDO. Messages originating from BEA TUXEDO have the BEA MessageQ class of MSG_CLAS_TUXEDO. Reply messages from BEA TUXEDO have either the BEA MessageQ class of MSG_CLAS_TUXEDO_TPSUCCESS or MSG_CLAS_TUXEDO_TPFAIL.

Detaching from the Message Queuing Bus

To detach from the message queuing bus, applications can use:

- the `pams_detach_q` function—to detach a selected message queue or all of the application's message queues from the message queuing bus. When an application detaches from its primary queue, this function automatically detaches all secondary queue attachments defined for the primary queue. When the last

message queue has been detached, the application is automatically detached from the BEA MessageQ message queuing bus.

- the `pams_exit` function—terminates all attachments between the application and the BEA MessageQ message queuing bus. All pending messages in temporary queues and those permanent queues which are not defined as permanently active are discarded. Only the messages pending in permanently active queues, including multireader queues, are retained.

Refer to the *BEA MessageQ Programmer's Guide* for more detailed information on how to use these BEA MessageQ functions.

Exchanging Messages Between BEA MessageQ and BEA TUXEDO

BEA MessageQ V5.0 include a messaging bridge that allows the exchange of messages between BEA MessageQ V5.0 and BEA TUXEDO V6.4 or BEA M3 V2.1. BEA MessageQ applications can send a message using `pams_put_msg` that a TUXEDO application can retrieve through a call to `tpdequeue`. TUXEDO applications can send a message using `tpenqueue` that a BEA MessageQ application can retrieve through a call to `pams_get_msg(w)`. In addition, a BEA MessageQ application can invoke a TUXEDO service using `pams_put_msg`. It is also possible for a TUXEDO application to use `tpenqueue` to put a message on a queue and `tpdequeue` to retrive a message from a queue.

3 Designing and Developing BEA MessageQ Applications

Message queuing provides a flexible approach to distributed application development because applications share information through messages stored in queues. Because BEA MessageQ provides a set of portable API functions that support all industry-leading platforms, it frees applications from having to embed operating system or network-specific code in order to accomplish message exchange.

Read the following sections to learn more building BEA MessageQ applications:

- Designing a BEA MessageQ Application
- Advanced Message Queuing Features
- Testing and Debugging BEA MessageQ Applications

Designing a BEA MessageQ Application

To design a distributed application using BEA MessageQ, application developers need to:

- Understand the business problem to be solved
- Develop the communications model for applications to exchange information

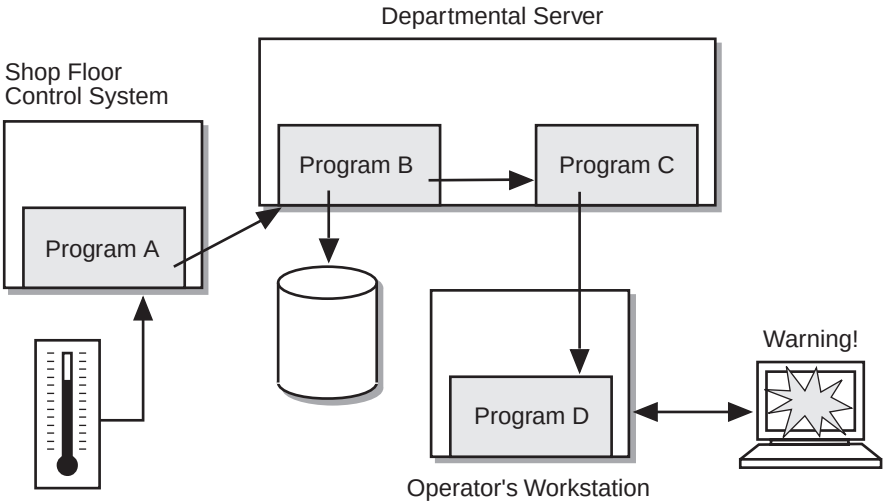
- Decide which BEA MessageQ features best suit the application's needs
- Design the message flow and system configuration to support application deployment

Solving the Business Problem

The first step in developing any application is to identify the business problem. As you research the current user environment and learn about their problems, you will soon determine whether the business need calls for a new application, or if the need is for existing applications to be integrated. Whether the solution requires the integration of new or existing applications, you can employ message queuing as the operating system- and network-independent "glue" that allows the application components to share information.

For example, let's break down the problem solved by our shop-floor monitoring example as shown in Figure 3-1.

Figure 3-1 Sample BEA MessageQ Application



In this case, the user group needed to:

- Monitor temperatures from a real-time shop floor process
- Store temperatures for later analysis and reporting
- Analyze temperature readings to detect out-of-range conditions
- Alert shop-floor supervisors automatically when there was a problem on the floor

The design of the application to solve this problem calls for a number of separate application components as follows:

- A program to read in temperatures from a shop-floor machine controller (Program A)
- A program to store the temperatures in a database for later analysis and forward them to a monitoring application (Program B)
- A program to monitor the temperatures and report on out-of-range conditions (Program C)
- A program to display an alert to a shop-floor supervisor (Program D)

Message queuing was chosen as the integration approach because it provides a loosely-coupled asynchronous means to pass information between application components with high throughput and platform independence. Program A uses datagram-style messaging to quickly pass temperature readings to Program B at 30 second intervals. Program B reads the messages from its memory-based queue and writes them to a database from which they can be analyzed.

Program B also forwards the temperature readings to Program C which checks whether they are above or below an acceptable range. If an unacceptable temperature reading is received, Program C sends a message to Program D running on a shop-floor workstation to trigger a display that alerts a supervisor to the out-of-range condition.

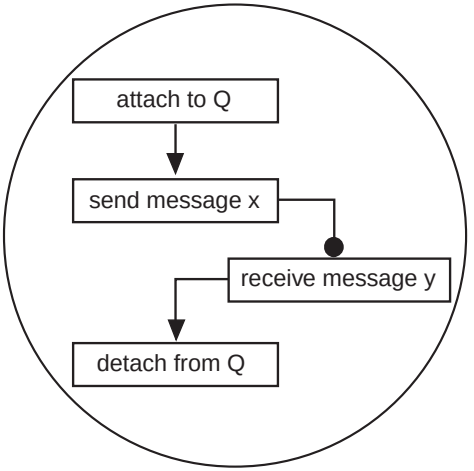
Developing the Communications Model

After you have broken down the application problem into its program components, you are ready to decide the communication model that must be used for each set of interacting programs.

The simplest style of messaging is called datagram messaging. This is a one-way flow of information between two applications that does not follow a request/response paradigm. In our shop-floor monitoring application, for example, temperature readings are sent as datagram messages to the monitoring application. The application reading the temperatures does not require a response to each reading.

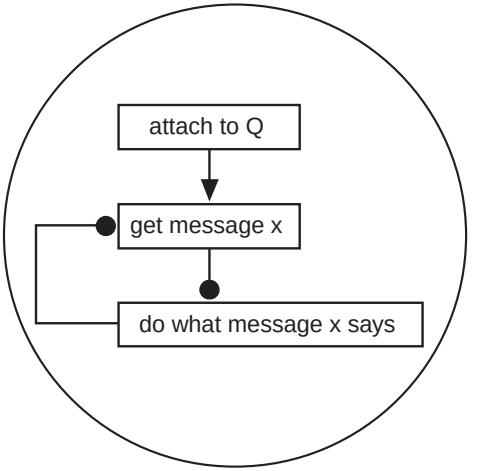
In addition to datagram style messaging, request/response messaging can be used to implement the client/server model of application integration. Using this model a client program sends a request to a server program. The server program reads the request, processes it, and returns the results to the client as shown in Figure 3-2.

Figure 3-2 Request/Response Messaging Paradigm



A more complex communication model that can be used for client/server or peer-to-peer messaging uses queues as service points. Using this model, sender programs direct requests or simply send information in the form of messages to a central queue. Several receiver programs may read the queue, obtain the request or information, process it, and then read the queue for another message as shown in Figure 3-3.

Figure 3-3 Service Point Messaging Paradigm



The communication model determines much of the system design, including the structure of the sender and receiver programs and the message queue configuration. The system designer uses the message queuing communication model that is most efficient for message exchange between each set of application components to be integrated.

Defining Major Application Needs

In addition to designing the most efficient communications model, the application developer must determine how to use BEA MessageQ features to implement the application design. The questions and answers in this topic can help the application developer to identify some of the major aspects of system design and development using BEA MessageQ such as:

- What rate of messaging throughput is required?
- Does the application require reliable delivery?
- Is the processing of message data between applications independent or interdependent?
- Does the application call for simultaneous distribution of information?
- Will the application receive different kinds of messages?

- Could the application benefit by load balancing between servers?
- Is sequential processing of information required?
- What is the requirement for data persistence?
- Should the applications be insulated from configuration changes?
- Does the application environment change frequently?

Choosing the Style of Messaging

Message throughput depends on the style of messaging selected and the delivery mode of the message. For example, buffer-style messaging is faster than FML-based self-describing messaging, because of the extra steps of message encoding and decoding. In addition, limiting message size for buffer-style messages to 32K enables messages to be delivered faster over the distributed network. Though the selection of messaging style is based on the needs of the application, it is important for application developers to consider the performance implications of each messaging style when electing which style to use.

In addition to the selection of messaging style, the delivery mode is the other critical factor in determining messaging throughput. If an application requires high messaging rates, the application developer selects a delivery mode which sends the message to the target queue in memory and requires no notification of whether the message is received (`PDEL_MODE_NN_MEM`). This kind of message is called a datagram because it requires the least processing overhead and provides the fastest messaging throughput. BEA MessageQ is capable of sending datagrams at rates of thousands of messages per second.

Datagram messaging, however, is only useful for applications that do not require a guarantee that every message is received. In our shop-floor monitoring example, the program that sends temperature readings uses datagram messages because, if one message is not received, the next will be sent in 30 seconds. It is not necessary for the receiver program to get every message in order to promptly report on out-of-range conditions.

Choosing Recoverable or Nonrecoverable Message Delivery

In contrast to datagram style messaging, some applications require a guarantee that the message be delivered, though the speed of delivery is not of great concern. For example, a developer could use message queuing to integrate the components of a

manufacturing resource planning (MRP) system. In the just-in-time manufacturing environment, it is critically important that the order processing application notify the inventory application when goods are sold because manufacturing scheduling is based on inventory levels.

In this case, the developer could use recoverable messaging to exchange this kind of important information between applications to ensure that the inventory level is accurately maintained. Recoverable messaging reduces messaging throughput because of the additional system resources required to save messages on disk in case they cannot be delivered. However, the value of automatic recovery in the event of system, process, or network failures might outweigh the disadvantage of additional processing time.

Choosing Asynchronous or Synchronous Messaging

Interdependent applications use a blocking request/response paradigm. For example, when Program A sends a message to Program B, it halts processing until it receives a reply from Program B. BEA MessageQ offers delivery modes to support synchronous communication (`PDEL_MODE_WF_`*xxx*).

For example, let's look at a banking application. A bank teller may enter a request for a withdrawal of money from a customer's account. The banking application requires that the customer's account balance be checked and sufficient funds available before the withdrawal can be made. Therefore, the request message is sent using a blocking delivery mode. The request transaction cannot continue processing until the server application checks the account balance, verifies that the amount of money requested is available for withdrawal, and returns a response to the requesting application to proceed with the transaction.

Other applications share data but operate independently. These applications use asynchronous messaging; the sender program sends the message and continues processing. The receiver program receives and reads the message at a later time—independent of the operation of the sender program.

For example, in a manufacturing resource planning (MRP) system, it is not necessary for the order entry application to halt processing while it waits for the inventory application to receive the message because the data does not affect its own processing. So, for example, the order fulfillment application can send a message to the inventory system identifying how many items were sold using an asynchronous delivery mode (`PDEL_MODE_AK_`*xxx*). The order fulfillment application does not need to halt processing while the inventory application reads the messages and decrease the inventory count.

Using Message Broadcasting

Some applications require simultaneous distribution of information to many recipients at the same time. A stock brokerage program is a good example of this kind of application. As stock prices change, the updated values must be simultaneously displayed on all stock traders' monitors. BEA MessageQ offers a feature called message broadcasting which allows an application to send one message that is simultaneously delivered to all subscribers. This capability is also called publish and subscribe.

Using Message Selection

Applications can receive a variety of message types. For example, an application can receive responses to its requests, notification of successful delivery of asynchronous messages, broadcast messages, timer expiration messages, and so on. The BEA MessageQ API offers a feature called message selection that allows receiver programs to sort out the messages they receive by correlation identifier, sequence number, message source, class, type, priority, or a combination of message attributes. If your application will receive different kinds of messages, you need to include logic for sorting the messages as they are read using the appropriate `pams_get_msg` function.

Load Balancing with MRQs

Many applications would benefit greatly if multiple servers were allowed to process the data instead of a single server. The efficient and dynamic distribution of processing power is commonly referred to as load balancing. BEA MessageQ offers load balancing through its multireader queues (MRQs).

For example, let's look at an order processing system. Company A's sales people enter customer orders using laptop computers by dialing into a main order entry system. After the order information is entered, the transaction is transmitted as a message to the order processing server program. Normally, there are several server programs running to process customer orders.

Each server reads a message from the MRQ that contains the customer order transaction information, processes the order information, and then reads the next available messages. As the number of messages in the MRQ grows, additional servers can be started to handle the load.

MRQs are generally used for load balancing when each message is a self-contained transaction. If an application requires multiple messages to be read and procesessed sequentially, the application uses a single-reader queue as the target queue to ensure proper processing. Optionally, an MRQ can be used to set up a session with a server application which then uses a single-reader queue for the remainder of the transaction. In this case:

- the client application sends an inital request message to an MRQ
- a server application reads the inital request message from the MRQ and then sends the client application a message containing the address of the server's single-reader queue
- the client application reads the message from the server and uses the server's single-reader queue to complete the transaction with additional messages

When implementing this approach, it is important to note that the client application is temporarily stalled while waiting to receive the address of the server's single-reader queue.

Choosing Single Reader Queues for Sequential Processing

If sequential information processing is required, the application must send the messages to a target queue that is defined as a single reader queue. Single-reader queues are owned by a single process which reads messages from the queue in FIFO (first-in/first-out) order.

It is important to note that, by default, BEA MessageQ places the highest priority messages at the top of the queue. Priority ranges from 0 (lowest priority) to 99 (highest priority). For example, priority 1 messages are always placed before priority 0 messages. Messages are placed in first-in/first-out order by message priority.

Choosing Permanently Active Queues for Data Persistence

Some applications require information be available for only a short period of time. Or, applications may require that information be available only when a process is attached to a queue and, therefore, actively retrieving information. In the former case, the application developer would attach to a temporary queue. In the latter case, the application developer would send messages to a permanent queue that is configured to be active only when a process is attached.

For example, a client application that takes account inquiries from customers at an ATM machine only requires the use of a queue long enough to provide a particular customer with one time information about their account. The client application would attach to the message queuing bus using a temporary queue, request the account balance from the message server, and wait for the message server to return the account balance. After the balance inquiry is fulfilled, the client application detaches from the temporary queue and waits for the next account inquiry.

However, if an application requires message data to be captured regardless of whether an application is available to process it, the developer must define the queue as permanently active to enable it to store messages when no application is attached, or use a recoverable delivery mode for sending to this queue.

Using BEA MessageQ Naming

Another important aspect of application development is to decide whether applications should be insulated from changes in the underlying BEA MessageQ environment configuration. For very stable environments which infrequently change equipment configuration, this is not a great concern. However, for dynamic, multiplatform environments, it may be very important to ensure that applications continue to run without recoding despite underlying configuration changes. However, there is a performance loss when naming is used for each queue reference.

Application developers can insulate programs from configuration changes using the BEA MessageQ naming feature. To use naming, applications are designed to refer to queues by name and not by using their queue address. Applications use the `pams_locate_q` function to look up the queue address for a queue name at runtime and pass the value to the `pams_put_msg` function in order to send the message. Naming enhances the flexibility of applications and frees them from requiring maintenance each time the configuration of the BEA MessageQ environment changes.

In addition, BEA MessageQ offers the ability to assign a service point at runtime using the `pams_bind_q` function. So, for example, if applications are designed to read a queue called “parts_orders,” the location of this queue can be determined at runtime by binding the queue name “parts_orders” to the queue address of the server that will be processing the parts orders at that time. The queue name can be unbound and the `pams_bind_q` function issued again to change the location from which parts orders will be obtained providing failover capability. Queue names are available on a group-wide and bus-wide basis providing a wide degree of flexibility to change the runtime environment.

Using FML for Self-Describing Messaging

For application environments subject to frequent change or that run on heterogeneous machine environments, FML-style self-describing messaging provides numerous capabilities to enhance the flexibility of applications. For example, using FML, you can add fields to a message that can be read by new applications without disrupting the way existing applications process the message. In addition, you can change the size of data fields in a message as your needs change without recoding applications because the size of the data field is encoded as part of the message itself.

In addition, messages can be planned with future considerations in mind because a single message can be designed for use by a variety of applications that use different data fields within the message and by future applications will use data fields not in use today. Also, FML marshals the data so that programmers need not be concerned with the different data formats from machine to machine.

Designing Message Flow and System Configuration

After you have broken down the application into its component programs, designed the communications model, and determined the BEA MessageQ features required by each component, you need to map out the messaging flow and determine how the component applications will be deployed in the distributed environment by answering the following questions:

- Which applications must communicate with each other?
- Which computer systems do/will these applications run on?
- Where are the computer systems located?
- What networks and operating systems are these computers running?
- Where are the users located?
- What is the application data flow?

A system designer answers these and many other detailed questions about the application in order to map the flow of information between sender and receiver programs in the distributed heterogeneous network.

Though a system manager may perform the configuration tasks, the following BEA MessageQ entities must be set up in accordance with the general system design in order for applications to exchange messages:

- The message queuing bus which acts as the common mechanism for attached applications to exchange information
- Message queuing groups for all participating nodes in the network
- Message queues for storing information to be read by receiver programs

The BEA MessageQ environment must be configured before applications are able to exchange information. For example, a system designer may designate queue 40 in group number 1 to receive temperature readings from a semiconductor furnace.

Once the bus, group, and queue address are defined, the sender program knows where to direct messages containing temperature readings. The receiver program also knows which queue to attach to in order to read and respond to temperature changes in the furnace.

Advanced Message Queuing Features

In addition to its ability to send and receive messages between applications in a distributed multivendor environment, BEA MessageQ has advanced features to provide developers with the following powerful capabilities:

- FML Self-Describing Messaging
- Recoverable Messaging
- Message Selection
- Broadcasting Messages
- Naming
- Using Message Based Services
- Exchanging Messages Between BEA MessageQ and BEA TUXEDO V6.4 or BEA M3 V2.1

- Additional API Functions

FML Self-Describing Messaging

Self-describing messaging using Field Manipulation Lauguage (FML) provides a flexible form of BEA MessageQ messaging. With buffer-style messaging, BEA MessageQ applications pass information using a message buffer whose format and structure were agreed upon by the sender and receiver programs. FML provides a mechanism for passing information as an opaque message buffer.

The `pams_put_msg` function now accepts a pointer to an FML32 buffer as the `msg_area` parameter. The resulting message contains the tags and values needed by the receiver program to decode the message. FML adds significant flexibility in message exchange because, in many cases, the contents of a message can be changed without requiring all related applications to be changed.

One example of the flexibility inherent in the FML messaging style is illustrated through the handling of a change in a message field size of an existing message. Applications that do not use FML must modify the application header files of each applicable sender and receiver program and then they must be recompiled, relinked and restarted. Using FML, however, the application developer need only change the tag associated with the value to indicate the new field size. When the message is received and decoded by the receiver program, the message contains the information on the new field size, therefore, the receiver program can properly interpret the data.

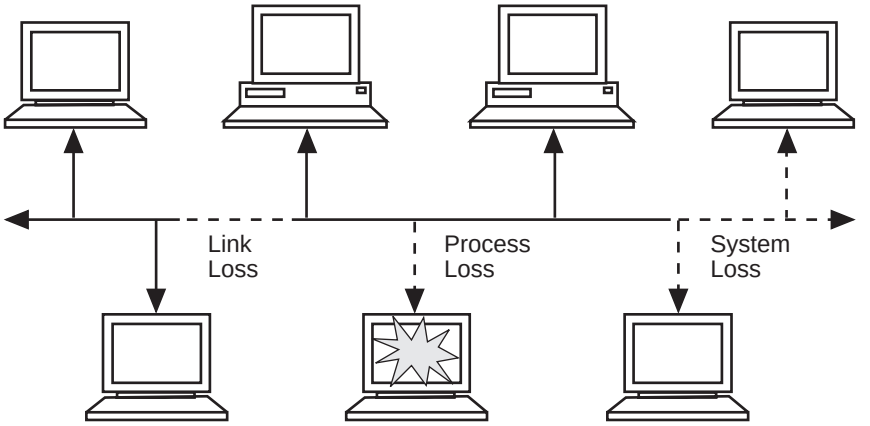
In addition to changing the size of data fields in a message, developers can add fields to FML messages. The new fields are available to be read by applications that have been programmed to read the additional field, however, all existing applications continue to run without a problem.

Another very powerful feature of FML is its ability to provide data transformation for applications exchanging information in heterogeneous multivendor environments. For a complete description of how to use the BEA MessageQ self-describing messaging feature, refer to the *BEA MessageQ Programmer's Guide.*

Recoverable Messaging

When an application sends a message, the final receipt of the message can be interrupted by various failure conditions including system, process, and network failures as shown in Figure 3-4.

Figure 3-4 Recoverable Messaging



However, BEA MessageQ applications can choose to send a message using recoverable delivery modes to enable BEA MessageQ to store messages in a disk file and deliver them as soon as it is possible. Using recoverable messaging BEA MessageQ applications can recover from message delivery failures caused by any of the following:

- Communications failure
- Application task abort
- System crash—sender, receiver, or both

When you send a BEA MessageQ message that is designated as recoverable, it is stored in one of two message recovery journals. The message recovery journal on the local system is called the store and forward (SAF) file. The message recovery journal on the remote system is called the destination queue file (DQF).

The selection of the recovery journal is determined by the delivery mode argument specified in the `pams_put_msg` function. If the delivery of a recoverable message is interrupted by a failure, it is automatically resent from the SAF or the DQF once communication with the target group is restored.

When an application receives and reads a recoverable message from a queue that is configured for explicit confirmation, it must use the `pams_confirm_msg` function to confirm message delivery. Confirming delivery of the recoverable message removes it from the message recovery journal. If the message is not confirmed, it will remain in the recovery journal and be redelivered if the application detaches and reattaches to the queue.

BEA MessageQ offers two types of message confirmation; implicit and explicit. The type of confirmation is set for each message queue as part of group configuration. Applications that receive recoverable messages in queues configured for implicit confirmation do not need to issue the `pams_confirm_msg` call. The message queuing system automatically issues the `pams_confirm_msg` call when the next sequential message is read from the message recovery journal. However, applications receiving recoverable messages in queues configured for explicit confirmation must issue the `pams_confirm_msg` call to delete the message from the message recovery journal.

Another queue characteristic that can be set during group configuration is the message confirmation order. Recoverable messages can be confirmed in order or out-of-order. The default confirmation order is to confirm messages sequentially as they are delivered from the message recovery journal and received by the target application.

In addition to the message recovery journals, BEA MessageQ offers two auxiliary journals to provide additional message recovery capabilities as follows:

- The dead letter journal (DLJ) file provides disk storage for messages that could not be stored for automatic recovery by the message recovery system. Undelivered messages stored in the DLJ file can be re-sent under user or application control.
- The postconfirmation journal (PCJ) file, stores successfully confirmed recoverable messages. It forms an audit trail of message exchange that can be read or printed. The PCJ file can also be used to resend successfully delivered messages if a database has become corrupted and must be restored. Message queuing groups must be configured to store successfully delivered messages in the PCJ.

For a complete description of how to use the BEA MessageQ recoverable messaging feature, refer to the *BEA MessageQ Programmer's Guide.*

Message Selection

When each BEA MessageQ message is sent, the sender program can assign a correlation identifier, message type, message class, and priority to distinguish it from other messages in the target queue. In addition, the message header contains the queue address of the sender program to allow the receiver program to identify the message source.

To selectively read messages, applications use `sel_filter` argument of the `pams_get_msg`, `pams_get_msgw`, or `pams_get_msga` functions. This argument allows developers to select messages by:

- *Default selection*—reads messages in FIFO order by priority. To use the default setting, set both words of the longword to zero.
- *Selection by message queue*—allows the application to retrieve messages based on a queue type or combination of queue types. For example, the application can read messages in the primary queue first, and then read messages in an alternate queue. A series of predefined constants are available to specify message selection by queue type.
- *Selection by correlation identifier*—allows the application to retrieve messages based on correlation identifier. To select by correlation id, use the symbol `PSEL_CORRELATION_ID` defined in `p_symbol.h` file. The application can tag any response to the message with the same identifier.
- *Selection by sequence number*—allows the application to retrieve messages based on the seqence number contained in the PAMS status buffer. This allows precise control over the message selected; a message sequence number applies to one and only one message. To select by sequence number, use the symbol `PSEL_SEQUENCE_NUMBER` defined in `p_symbol.h` file.
- *Message attributes*—allows application developers to retrieve messages based on assigned characteristics that let the receiver program know how to process the message. BEA MessageQ message-based services use reserved messages type and class symbols defined in the `p_typcls.h` file. To create additional type and class codes for your application, create a separate include file containing the type and class code symbols. The receiver program can also use message selection by attribute to read high priority messages before less critical ones.

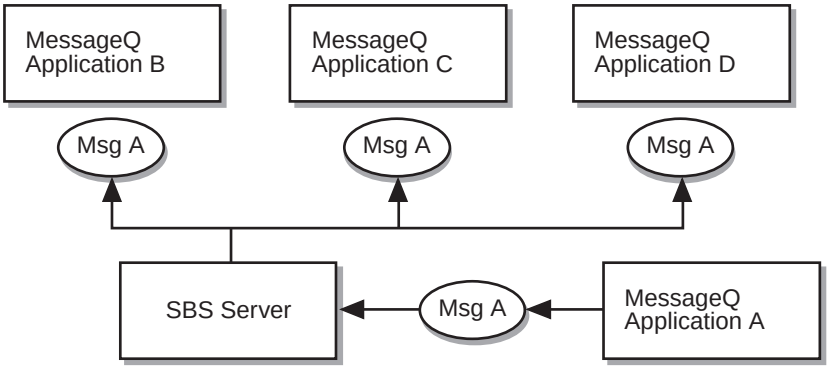
- *Message source*—applications can be programmed to read only those messages from a particular source. To use this option, enter the group ID and the queue number of the source queue from which the application should read messages.
- *Compound selection*—enables developers to create compound selection criteria using the `pams_set_select` function. Compound message selection allows the use of complex rules such as AND/OR operations for reading messages. To use compound selection use the constant `PSEL_BY_MASK` as the first word and the `mask_id` of the selection mask created using `pams_set_select` as the second word. The application can cancel the use of a selection mask using the `pams_cancel_select` function.

For a complete description of how to use the BEA MessageQ message selection feature, refer to the *BEA MessageQ Programmer's Guide.*

Broadcasting Messages

Message **broadcasting** is a style of messaging that enables one sender program to send a message simultaneously to many receiver programs. BEA MessageQ **Selective Broadcast Services (SBS)** manage the broadcasting of data between processes and groups of processes as shown in Figure 3-5.

Figure 3-5 Selective Broadcast Services



BEA MessageQ broadcast services provide applications with:

- One-to-many message queuing
- Lists of application processes that are interested in messages that are broadcast

- User-definable rules, known as selection rules, which can be used to selectively extract messages from a broadcast stream

BEA MessageQ message broadcasting is similar to radio broadcasting. A sender program directs a message to a selected broadcast stream to be received by any interested application. Then, the receiver program "tunes in" just as a listener chooses a particular radio station by registering to receive messages sent to that broadcast stream. The sender program does not know which applications are receiving the messages it sends. Receiver programs register and deregister for message receipt from a particular broadcast stream without affecting the sender program.

Message broadcasting simplifies application development because sender programs do not need to be aware of the number, state, and location of the target queues. Message broadcasting increases efficiency by directing messages to many targets with a single call.

To send a message to multiple recipients simultaneously, the sender program uses the `pams_put_msg` function and specifies a Multipoint Outbound Target (MOT) as the queue address. A broadcast target, numbered between 4000 and 6000, is an identifier for a broadcast stream. A broadcast stream is the set of target queues registered to receive messages directed to a particular broadcast target. The SBS Server in each message queuing group distributes messages to registered receiver programs.

For a complete description of how to use the BEA MessageQ message broadcasting feature, refer to the *BEA MessageQ Programmer's Guide*.

Naming

Naming is a powerful BEA MessageQ capability that enables applications to refer to queues by name instead of by their queue address. Using naming separates applications from the details of the current BEA MessageQ environment configuration and enables system managers to make configuration changes without requiring developers to change their applications.

For example, an order processing application uses a multireader queue called `ORDER_INBOX` to store product order messages from client programs. Order fulfillment server programs read messages from `ORDER_INBOX` to process each order. Initially, `ORDER_INBOX` might be defined as queue 7 in group 1, an HP-UX system. However, after the company purchases a high performance, Compaq system running Tru64 UNIX, this queue may be redefined as queue 8 on group 2 to provide better

performance for the application. In this example, no change is required to either the sender or receiver programs because they refer to the queue by name and not by its queue address.

To obtain the queue address for a queue name at runtime, application developers use the `pams_locate_q` function. Queue names can be defined in BEA MessageQ to have a **local** or **global** scope. A local name can be used as the target queue by applications running in the same message queuing group in which the name was defined. A global name can be used as the target queue by any application on the message queuing bus.

Names can be defined using a **static** or **dynamic** approach. The static approach means that the name-to-queue address translation is defined in the Queue Configuration Table (%QCT) or in the Global Name Table (%GNT) of the BEA MessageQ group initialization file. When the group starts up, the name-to-queue address translations are written to the BEA MessageQ name space. To change a name-to-queue address translation, you must stop the message queuing group, change the queue name definition in the group initialization file and restart the group and its applications. When an application performs a `pams_locate_q` function, it will obtain the new queue address for the queue name.

Dynamic naming means that the name-to-queue address translation is defined at runtime by an application using the `pams_bind_q` function. When the `pams_bind_q` function successfully completes, the name-to queue translation is written to the BEA MessageQ name space. To change the name-to-queue translation, the application must unbind the name from the queue address and use the `pams_bind_q` function to bind a new queue address to the queue name.

The BEA MessageQ process that supports the naming capability is called the Naming Agent. The Naming Agent is responsible for creating entries in the name space and for providing the look up capability for name-to-queue translations at runtime.

To use the BEA MessageQ naming feature, you must configure the message queuing environment as follows:

- the message queuing group that runs the Naming Agent must be identified in the `%NAM` section of the group initialization file
- the name-to-queue translation for each statically defined queue name must be entered to the Queue Configuration Table (`%QCT`) and the Group Name Table (`%GNT`) of the group initialization file. In addition, the GNT section must contain the queue names to be associated with queue addresses at runtime using the `pams_bind_q` function. These names are associated with queue address 0.0 so that the dynamic queue address can be set at runtime.

Refer to the installation and configuration guide for your platform for detailed information on how to configure the BEA MessageQ naming feature. For more detailed information on how to design your application to use naming, refer to the *BEA MessageQ Programmer's Guide*.

Using Message Based Services

BEA MessageQ applications may wish to perform certain standard tasks such as checking the status of a queue or the status of a cross-group connection before sending a message. To make these tasks easier, BEA MessageQ offers message-based services. These are predefined request, notification, and response messages exchanged between application processes and the BEA MessageQ servers that support each message queuing group.

BEA MessageQ offers message-based services for:

- Notifying applications of the availability or unavailability of message queues
- Registering to receive broadcast messages
- Monitoring and controlling link status
- Obtaining the status of all message queues
- Opening, closing, and renaming message recovery journals (OpenVMS only)
- Redirecting the contents of a destination queue file to another queue (OpenVMS only)

For example, an application may want to check whether a queue is available before it sends a message. BEA MessageQ offers built-in availability checking through its message-based services.

To register for availability notification, the application sends an `AVAIL_REG` message to the primary queue of the AVAIL Server running in its message queuing group. The AVAIL server responds by sending an `AVAIL_REG_REPLY` message to the sender program acknowledging that it is registered to receive availability notification.

Thereafter, as queues attach and detach from the message queuing bus, the sender program receives AVAIL and UNAVAIL notification messages identifying which queues have become available and which have become unavailable. When the sender program no longer requires availability notification, it sends a `AVAIL_DEREG` message to the AVAIL Server and notification is terminated.

For a complete description of how to use the BEA MessageQ message-based services feature, refer to the *BEA MessageQ Programmer's Guide.*

Exchanging Messages Between BEA MessageQ and BEA TUXEDO V6.4 or BEA M3 V2.1

BEA MessageQ V5.0 include a messaging bridge that allows the exchange of messages between BEA MessageQ V5.0 and BEA TUXEDO V6.4 or BEA M3 V2.1. BEA MessageQ applications can send a message using `pams_put_msg` that a TUXEDO application can retrieve through a call to `tpdequeue`. TUXEDO applications can send a message using `tpenqueue` that a BEA MessageQ application can retrieve through a call to `pams_get_msg(w)`. In addition, a BEA MessageQ application can invoke a TUXEDO service using `pams_put_msg`. It is also possible for a TUXEDO application to use `tpenqueue` to put a message on a queue and `tpdequeue` to retrive a message from a queue.

This exchange of messages is made possible by two TUXEDO servers that are included in the BEA MessageQ installation and that run on the same machine as BEA MessageQ: `TMQUEUE_BMQ` and `TMQFORWARD_BMQ`.

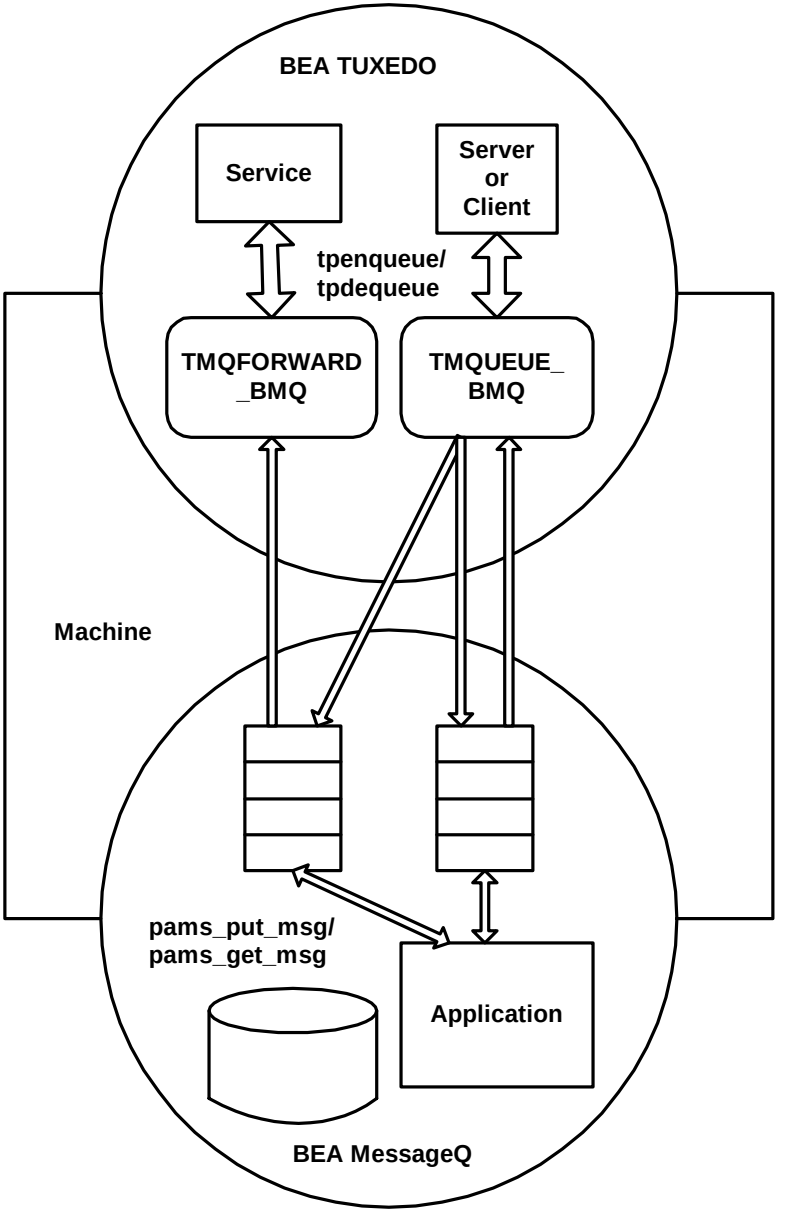
`TMQUEUE_BMQ` redirects TUXEDO `tpenqueue` requests to a BEA MessageQ queue where they can be retrieved with `pams_get_msg(w)`. `TMQUEUE_BMQ` also redirects `pams_put_msg` requests to TUXEDO where they can be retrieved with `tpdequeue`.

`TMQFORWARD_BMQ` listens on specified BEA MessageQ queues and forwards `pams_put_msg` or `tpenqueue` requests to a TUXEDO service. It also puts the reply or failure message on the sender's response queue.

The target queue and service are defined when `TMQUEUE_BMQ` and `TMQFORWARD_BMQ` are configured. This ensures that message exchange between BEA MessageQ and TUXEDO is transparent to the application.

Figure 3-6 illustrates message exchange between BEA MessageQ and TUXEDO.

Figure 3-6 Message Exchange Between BEA MessageQ and TUXEDO


BEA TUXEDO
Service
Server or Client
tpenqueue/ tpdequeue
TMQFORWARD _BMQ
TMQUEUE_ BMQ
Machine
pams_put_msg/ pams_get_msg
Application
BEA MessageQ

Enabling the Messaging Bridge

The TMQUEUE_BMQ and TMQFORWARD_BMQ servers are part of the BEA MessageQ installation and are installed when BEA MessageQ is installed. During the installation procedure, you are prompted to choose one of the following installation options for BEA MessageQ and TUXEDO integration:

install on top of BEA TUXEDO V6.4
install on top of BEA M3 V2.1
install without BEA TUXEDO

Note that if you are installing BEA MessageQ on OpenVMS, you do not have the option of installing over BEA M3 V2.1. Also, you must install BEA MessageQ for OpenVMS on an OpenVMS AXP 7.1 system to use the messaging bridge.

If you choose to install on top of BEA TUXEDO V6.4 or BEA M3 V2.1, the applicable files for the TMQUEUE_BMQ and TMQFORWARD_BMQ servers are installed on your system. If you install without BEA TUXEDO, the TMQUEUE_BMQ and TMQFORWARD_BMQ servers are not installed on your system. See the installation and configuration documentation for your system for detailed installation and configuration instructions.

Once the TMQUEUE_BMQ and TMQFORWARD_BMQ servers are installed, the system administrator enables message enqueuing and dequeuing for the application by specifying the servers as application servers in the *SERVERS section of the TUXEDO ubbconfig file. See the TMQUEUE_BMQ and TMQFORWARD_BMQ reference pages in the *BEA MessageQ Reference Manual* for detailed information on the server configuration syntax.

Additional API Functions

In addition to its API functions for sending and receiving messages, BEA MessageQ offers the following additional API functions to facilitate the development of distributed applications:

- pams_bind_q—used to set local and global names for queues at runtime
- pams_locate_q—obtains the queue address for a queue name at run-time

- pams_set_timer and pams_cancel_timer—sends a notification message to an application at a particular time of day or when a specified time period has elapsed
- pams_status_text—returns detailed status information for the API call
- putil_show_pending—provides the total number of pending messages for a queue

Defining a Name-to-Queue Translation at Runtime

The pams_bind_q function is designed to dynamically associate a queue name with a queue address at runtime. This function enables a server application to dynamically sign up to service a queue alias at runtime.

For example, an application may have client programs that submit orders for widgets by sending BEA MessageQ messages containing the appropriate information to a queue called "widget_orders." In addition, the application has a server program that processes widget orders. To maximize flexibility, the server program starts up and then binds the address of its primary queue to the queue name "widget_orders" which is defined in the group initialization file. The client programs are designed to perform the name-to-queue address translation at runtime using the pams_locate_q function and orders are sent to the primary queue of the server program.

If the server should fail, or if the server program is moved to a faster system, the pams_bind_q function can be used to unbind the queue name from the primary queue of one server program and bind it to the primary queue of another. The redefinition of the queue address of "widget_orders" is handled by BEA MessageQ and is invisible to the client programs which require no reprogramming to direct messages to a different queue. When the queue address for the name is redefined, the message from the client applications are automatically redirected to the new queue address.

Locating the Queue Address for a Queue

To send a message to a target queue, the application developer must supply a queue address as the target argument to the pams_put_msg function. Depending on the needs of the application, the queue address may be set when the program is compiled or may be supplied when the application is running.

To specify the queue address at compile time, the application developer supplies the queue number and group ID of the target queue to the `pams_put_msg` function. This information must match the group configuration information for the BEA MessageQ environment. If the group and queue number of the target queue do not exist in the group configuration information, the message cannot be delivered.

BEA MessageQ also allows the queue address of the target queue to be resolved at runtime. Using this approach, the application refers to queues only by name. To obtain their queue addresses, the application invokes the `pams_locate_q` function to obtain the queue address for a queue name. When the queue address is returned by the `pams_locate_q` function, the developer uses it to supply the queue address to the `pams_put_msg` function. Designing applications to refer to queues by name, adds some processing overhead at runtime, however, it increases flexibility over compile time resolution by insulating applications from changes in environment configuration.

Using Timers

BEA MessageQ offers a timer API function that eliminates the need to write application-specific timer code. The PAMS timer function sends a timer expiration message to an application when:

- a specified amount of time has elapsed—just as a cooking timer, for example, it signals the application that 30 minutes has passed and an event should be triggered
- a time of day has arrived—just as an alarm clock, for example, it signals the application that it is now 10 o'clock and an event should be triggered

The application sets a timer using the `pams_set_timer` function by supplying a `timer_id`, the type of timer and the value to be set. An application can set multiple timers by supplying each with a unique `timer_id`.

When the specified time has elapsed or the time of day has arrived, BEA MessageQ sends a priority 1 message with a message type of `MSG_TYPE_TIMER_EXPIRED` to the application's source queue. The data structure of the `TIMER_EXPIRED` message contains the `timer_id` to enable the application to discern which timer-related event to trigger.

The application cancels timers using the `pams_cancel_timer` function by supplying the `timer_id` of the timer to cancel. All pending timer expiration messages with the `timer_id` of the timer being canceled are purged from the queue.

Obtaining Detailed Status Information

Application developers can use the `pams_status_text` function to obtain a descriptive text string and a severity level for each API return value. This API function receives the status value and returns a text description in the following format:

```
PAMS__SUCCESS, normal successful completion
```

The text description contains the text name of the return code (as it appears in the documentation and development include files) followed by a comma, a space, and then a status description. If the user buffer is large enough, the string is zero terminated.

In addition to the text description, this function returns a code indicating the severity level for both success and error messages. Severity levels are designed to provide more information about the message being returned.

Obtaining the Number of Pending Messages in a Queue

The `putil_show_pending` function provides the number of pending messages for a single queue or a list of queues. The value returned by this function contains the total number of messages in each memory queue as well as the number of messages in the local and remote recovery journals targeted for delivery to the selected queue. This function can be used to monitor for bottlenecks in application processing and message flow design.

Testing and Debugging BEA MessageQ Applications

BEA MessageQ provides the following powerful tools that assist application developers testing and debugging distributed applications:

- The BEA MessageQ Script Facility—provides a means of simulating message exchange between applications under development.
- The BEA MessageQ Test Utility—provides a simple way to test message exchange with an existing application.

- Message Tracing—provides a means to diagnose problems with message exchange between applications by creating a log file of all BEA MessageQ events between the two processes.

BEA MessageQ Script Facility

The BEA MessageQ Script facility provides a productivity tool for application developers to use in simulating message exchange between programs. Instead of writing a test program, you create a script file containing instructions for capturing messages sent or received by an application, replay captured messages, or simulate messages sent from an application that is still under development.

Message simulation offers a shortcut for sending messages to an application. Instead of writing a program to send a message, you can use a text editor to create a script file. The script file contains the message information and other instructions such as whether to log the message exchange.

The message information and other instructions are entered to the script file using the BEA MessageQ scripting language. When script processing is enabled, BEA MessageQ processes the contents of the script file and delivers the message to the target queue where it can be read by the receiver program.

Message capture provides a mechanism for viewing the messages sent or received by an application. To capture messages, you use the scripting language to create a script file that identifies the messages to be captured. Captured messages can be displayed on the screen, written to a log file, or both. When script processing is enabled, BEA MessageQ captures the messages and displays or logs them as specified in the script file.

Message replay uses the messages captured in a log file as input to an application in exactly the same way as messages entered to a script file. The script replay feature lets developers capture messages sent or received by an application and supply them as input to another program. By tracking the program's response to the captured messages, the developer can debug message exchange between programs that share information using BEA MessageQ.

Note: The BEA MessageQ script facility is available on UNIX and OpenVMS systems only.

For a complete description of how to use the BEA MessageQ script utility, refer to the *BEA MessageQ Programmer's Guide*.

BEA MessageQ Test Utility

The BEA MessageQ Test utility is a productivity tool that allows software developers to test message exchange with an existing application. A developer interacts with the graphical user interface or character-cell interface of the Test utility to:

- Attach to a permanent or temporary queue
- Read messages sent by an application or script file
- Send messages to a defined target queue

To run the Test utility, the developer must begin by setting environment variables to specify the bus and group in which the test application is running. Then the developers uses the pulldown options to build the attach, send, or receive function entering the same information required as arguments to these API function calls.

The Test utility provides a quick and easy means for application developers to:

- Build interactive tests of application modules.
- Send and receive messages to any target from any source.
- Test the message flow and messaging rates for a set of queues

To view a sample run of the Test Utility, refer to the installation and configuration guide for your environment.

Message Tracing

The BEA MessageQ message tracing feature logs internal messaging events to a file as they happen. You can use this file to diagnose application failures as you debug your application.

It is important to note that message tracing generates a high volume of output; therefore, you should only enable tracing for diagnostic purposes in the event of a problem. For more information on how to set up message tracing, refer to the *BEA MessageQ Programmer's Guide*.

4 Managing the BEA MessageQ Environment

After you develop your BEA MessageQ programs and deploy your distributed application into the production environment, you need to monitor and tune your system's performance and occasionally troubleshoot BEA MessageQ problems.

To successfully manage your distributed BEA MessageQ applications, you need to:

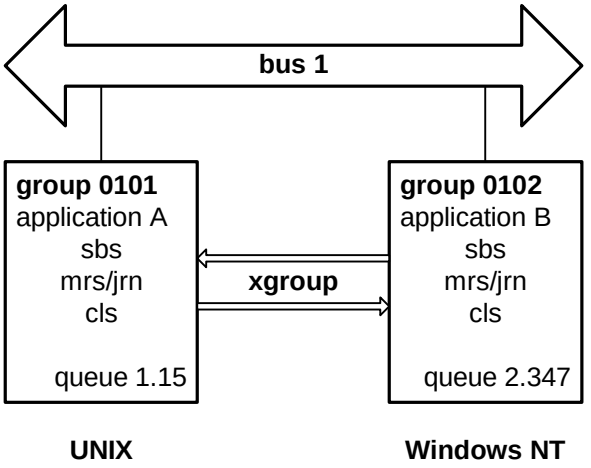
- Understand the BEA MessageQ Environment
- Monitor System Performance
- Troubleshoot Errors

Understanding the BEA MessageQ Environment

To efficiently manage and troubleshoot a distributed BEA MessageQ application, it is important to be able to visualize the components of the BEA MessageQ environment. Figure 4-1 shows how Application A running in Group 1 can be configured to exchange messages with Application B running in Group 2 on the same message

queuing bus though the systems do not run the same operating system. Communication between the two groups is enabled using the network communications link between both systems and a BEA MessageQ cross-group link.

Figure 4-1 The BEA MessageQ Environment



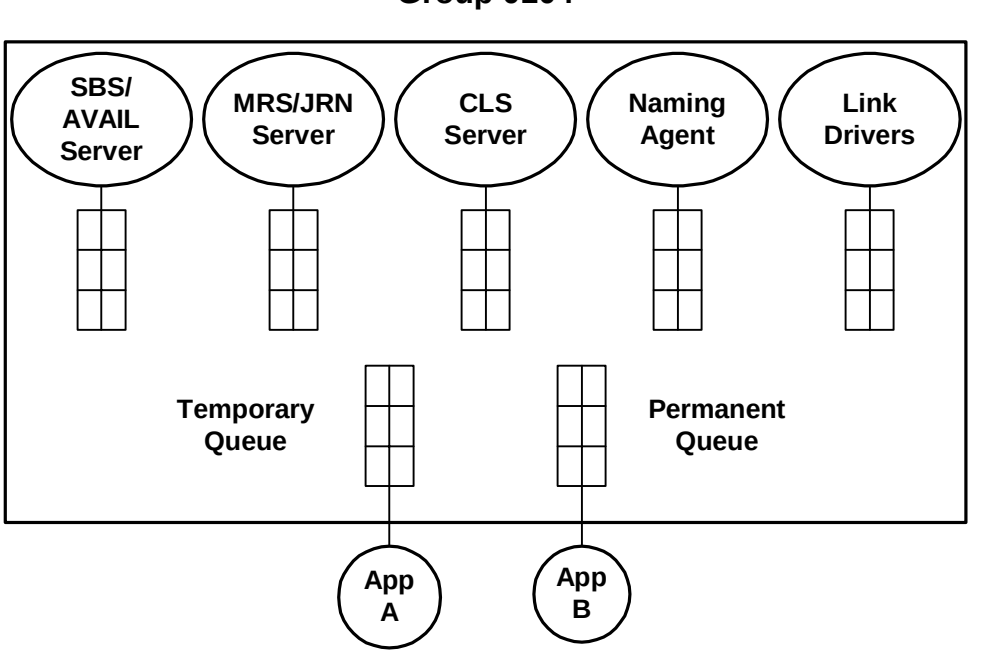
This typical configuration of the BEA MessageQ environment consists of:

- a single **message queuing bus** to provide the communication backbone for applications to exchange information using message queuing
- one or more **message queuing groups** per system. A message queuing group enables multiple queues to efficiently share BEA MessageQ services such as message recovery and broadcast services
- one or more **message queues** to receive BEA MessageQ messages. Message queues can be temporarily assigned for use by BEA MessageQ or can be permanently defined in the group initialization file
- one or more **cross-group connections** to enable message exchange between message queuing groups on the message queuing bus. (If the message queuing groups reside on different computer systems, a network connection must be present to enable the cross-group connection.)

Anatomy of a Message Queuing Group

Message queuing groups are designed to provide centralized resources for a group of queues running on a host system. As shown in Figure 4-2, each message queuing group may run a number of BEA MessageQ servers to service the needs of the temporary and permanent queues to which applications are attached.

Figure 4-2 BEA MessageQ Servers



Depending upon the services enabled in the group initialization file, a message queuing group may run the following processes:

- SBS Server—distributes broadcast messages based on the selection criteria set by registered applications
- Client Library Server—provides full message queuing services for applications running on BEA MessageQ Clients

- JRN Server—writes successfully delivered recoverable messages to the post confirmation journal; also writes recoverable messages to disk-based message recovery journals and resends the messages in the event of delivery failure
- Link Drivers—enables cross-group communication for applications running on different computer systems in different BEA MessageQ message queuing groups
- NA—the naming agent accesses and manages the BEA MessageQ bus-wide name space

Starting and Stopping Groups, Queues, Links and the CLS

As options of both its character-cell and GUI-based Monitor utility, BEA MessageQ enables users to interactively:

- Stop a message queuing group slowly (allowing processes to exit and clean up)
- Stop a message queuing group fast (immediate shutdown without clean up)
- Start a message queue
- Stop a message queue slowly (allowing messages to be read until the queue is empty)
- Stop a message queue fast (queue stops immediately and existing messages are lost)
- Start a cross-group connection
- Stop a cross-group connection
- Start the Client Library Server
- Stop the Client Library Server

Monitoring System Performance

The BEA MessageQ Monitor utility helps developers observe the BEA MessageQ environment on local and remote nodes. Developers and system managers can use the summary and detailed display of information by the Monitor utility to tune the BEA MessageQ system configuration.

To monitor or control your BEA MessageQ groups or buses, you can invoke either the Motif-based Monitor Utility or the character-cell Monitor utility. Either interface can be used to perform the following sets of functions:

- Collecting and displaying statistics for each queue
- Collecting and displaying statistics for each cross-group link

Error Logging and Recovery

BEA MessageQ has an error logging mechanism to display and capture informational, warning, and error messages that can occur during processing. The messages display a description of the condition to help developers gather more information about failure conditions within a message queuing group.

On UNIX and Windows NT systems the BEA MessageQ an error log file is created when the group is started using the appropriate switch on the dmqstartup command line. Error logs can be created for each message queuing group. On OpenVMS systems, an error log can be created at group startup or the System Manager utility can be used to redirect output to several error log files.

Listing 4-1 shows the kind of information logged for each BEA MessageQ event on a UNIX or a Windows NT system:

- the name of the process that logged the error
- the date and time on which the message was logged
- a description of the successful event or error condition

Listing 4-1 Sample BEA MessageQ Event Log File

```
************ dmqgcp (4150) 10-DEC-1999 15:25:23 ************
gcp, group control process for group 19 is running
************ dmqqe (4536) 10-DEC-1999 15:25:24 ************
qe, queuing engine is running
************ dmqloader (3366) 10-DEC-1999 15:25:24 ************
ldr, MessageQ System Loader starting
ldr, Parsing PROFILE Section
ldr, Parsing MRS Section
ldr, Parsing GROUP Section
ldr, Parsing ROUTE Section
ldr, Parsing QCT Section
ldr, Parsing GNT Section
ldr, Bad parameter sent to the GCP at line 318
ldr, Bad parameter sent to the GCP at line 330
ldr, Bad parameter sent to the GCP at line 331
ldr, Parsing CLS Section
ldr, Parsing NAM Section
ldr, Loader exiting normally
************ dmqjourn (2590) 10-DEC-1999 15:25:27 ************
jrn, journal process for group 19 is running
************ dmqld (2579.0) 10-DEC-1999 15:25:27 ************
ld, link listener for group 19 is running
************ dmqld (2591.0) 10-DEC-1999 15:25:27 ************
ld, link sender for group 19 to group 18 is running
************ dmqld (2579.0) 10-DEC-1999 15:25:32 ************
ld, link receiver for group 19 from group 18 is running
************ dmqgcp (4150) 10-DEC-1999 15:26:21 ************
ipi, dequeue message failed
************ dmqgcp (4150) 10-DEC-1999 15:26:21 ************
gcp, group control process for group 19 has exited
************ dmqqe (4536) 10-DEC-1999 15:26:21 ************
qe, queuing engine has exited
```

Glossary

access control list (ACL)

A list that defines the kinds of access to be granted or denied to users of an object. Access control lists can be created or objects such as files and devices.

acknowledgment (ACK)

A status message that indicates the completion of an operation.

address

See queue address.

application

A program or collection of programs designed to perform a function or business task.

application programming interface (API)

An interface used by application programs to call services external to the program. The API supports the exchange of information in a multivendor environment.

application protocols

An agreed set of rules that govern the management of connections between partner programs. See also *duplex connection* and *simplex connection*.

asynchronous

Pertaining to a style of message queuing whereby messages can be sent or received at any time without waiting for the receiver program to receive, process, or respond to a specific event. Contrast with *synchronous*.

asynchronous system trap (AST)

An software-simulated interrupt to a user-defined service routine. ASTs enable a user_process to be notified asynchronously of the occurrence of a specific event. If a user has defined an AST routine for an event, the system interrupts the process and executes the AST routine when that event occurs. When the AST routine ex-

its, the system resumes execution of the process at the point where it was interrupted.

attach

To make a process known to the BEA MessageQ message queuing bus and allow it to receive messages at a particular queue address.

attachment point

A particular queue location on the BEA MessageQ message queuing bus that allows communication between processes without requiring a formal connection sequence.

blocking

Pertaining to a synchronous style of message delivery where the program is forced to wait for an action to complete. Contrast with *nonblocking*.

broadcast distribution

The action of delivering a message to all processes interested in a particular broadcast stream.

broadcasting

A style of communicating that uses one message sender program and multiple message receiver programs. This capability is also called "publish and subscribe."

broadcast stream

A data message pipeline that has a single entry point and multiple exit points. Messages sent to the broadcast stream are simultaneously distributed to all registered queues. See also *private broadcast stream* and *universal broadcast stream*.

buffer

An internal memory area used for temporary storage of data records during input or output operations.

buffer pool

A common memory area that stores message buffers for a message queuing group. A buffer pool consists of fixed-size memory structures that can hold one message each.

bus ID

A reference value that distinguishes one BEA MessageQ message queuing bus

from another.

class

A 16-bit piece of data that describes a grouping or category of message types. Also called *message class*. See also *type*.

client

A computing system entity that uses the services of other system entities called servers. See also *server*.

client/server model

A hardware or software system design used in developing distributed applications. In the client/server model, a server system provides common database access, performs computations, and assumes system management tasks for its clients.

COM Server

A BEA MessageQ for OpenVMS server process that passes cross-group messages to other BEA MessageQ message queuing groups. A COM Server creates the BEA MessageQ message queuing bus environment and must be activated before message queuing can occur.

configuration editor

A Windows editor used for defining and managing BEA MessageQ buses, groups, and related information.

configuration file

A text file comprised of information line items used to configure BEA MessageQ software. This file is also called the group initialization file. The configuration data for message queuing groups is standard for all platforms.

confirmation

See message confirmation.

connectionless

Pertaining to not having a logical link. A connection does not have to be established with a partner process in order to pass information between them.

connection-oriented

Pertaining to a communication method where two partners must establish a connection before they can exchange messages.

correlation ID

A user-defined value associated with and identifying a specific message. Receiving applications can retrieve the correlation ID and tag any responses with the same value. This aids in matching responses with requests.

cross-group

Pertaining to messages that pass between BEA MessageQ message queuing groups. A cross-group message is targeted to a message queuing group outside of the local group. Cross-group connections enable applications to share information across different systems connected to the same message queuing bus.

datagram

A "best effort" style of message delivery in which a nonrecoverable attempt is made to deliver a message. If the message cannot be delivered to a target, then an error is logged.

dead letter journal (DLJ)

A file that provides nonvolatile disk storage for messages that cannot be stored for automatic recovery. Applications use the DLJ file to resend undelivered messages. Also called *DLJ file.*

dead letter queue (DLQ)

The permanent message queue that provides memory-based storage of all recoverable messages that could not be stored for automatic delivery. Also called *DLQ file.*

delivery interest point

A component of the delivery mode that indicates the step in the message recovery data flow at which the sender program is notified.

delivery mode

A selection of options that specify how the sender program receives notification of recoverable message delivery and the point in the message flow at which the notification is sent. See also *message delivery*.

destination queue file (DQF)

A message recovery journal that provides nonvolatile storage on a remote system for automatic recovery and delivery of messages. Also called *DQF file*.

distributed application

An application that divides the user interface, processing, or data among one or more units that execute on a single central processing unit (CPU) or multiple nodes in a network.

distributed computing

An application design methodology that places data entry and application processing close to departmental and functional end users. These users are most familiar with the input requirements and need the processed output to support their business objectives.

Distributed Name Services (DNS)

A heavyweight namespace that BEA MessageQ can use to store global names. Using DNS on OpenVMS systems enables BEA MessageQ to locate the queue address for a queue defined by any group on the message queuing bus. See also *naming*.

distribution

A stage in broadcast services where the SBS Server delivers a message to receiver programs.

distribution queue

A queue address that is specified in a broadcast or availability services registration message. The distribution queue is the final destination of a broadcast or availability notification message.

DLJ file

See dead letter journal.

DLQ file

See dead letter queue.

DQF file

See destination queue file.

duplex connection

An application protocol where the initiating partner is the sender program and the accepting partner is the receiver program, until the sender program requests a direction change and becomes the new receiver program. The accepting partner then becomes the sender program and remains the sender program until requesting a

direction change. Contrast with *simplex connection*.

event

A network- or system-specific occurrence, such as timer expiration, for which the logging component maintains a record.

explicit confirmation

A type of message confirmation that requires the receiver program to delete the message from the recovery journal using a message sequence number. The message is not deleted until the receiver program has finished processing the information in it.

facility

A collection of one or more computer programs that implement a set of related functions or services. The implementation of a facility can consist of either a process or a procedure.

failover

1) The process of a reconfiguration after a hard fault or for planned maintenance.

2) The ability of a system or component to reconfigure itself.

Field Manipulation Language (FML)

Field Manipulation Language (FML) is a set of C language functions for defining and manipulating storage structures called fielded buffers, that contain attribute-value pairs in fields. The attribute is the field's identifier, and the associated value represents the field's data content.

FML

See *Field Manipulation Language*.

full duplex

Pertaining to a communications method in which data can be transmitted and received at the same time.

global data structure

A data structure that can be shared by multiple processes.

global sections

An OpenVMS shared memory segment potentially available to all processes in the system. Access is protected by standard access control mechanisms.

group

See *message queuing group*.

group ID

The internal number of the BEA MessageQ message queuing group. The group ID is part of the queue address. Each group ID must be unique within the message queuing bus.

group name

The symbolic name associated with the BEA MessageQ group ID.

half-duplex

Pertaining to a communication method where one partner is sending data when the other partner is receiving data. See also *duplex connection*.

heterogeneous computing environment

An environment in which applications run on computer systems from different vendors employing various operating system and networking software.

heterogeneous messaging

The use of different communications methods to transfer messages.

heterogeneous operating systems

A configuration of a variety of computers and operating systems connected by networking hardware and software.

implicit confirmation

A type of message confirmation on BEA MessageQ for UNIX and Windows NT systems that automatically deletes a recoverable message from a journal file. The receiver program does not need to respond to the receipt of the message.

inbound conversation allocation

The allocation of conversations that are initiated by an OpenVMS transaction program.

initiator-only deallocation

A method of duplex connection termination where the initiating partner is the only one who can terminate the connection normally. Contrast with *open deallocation*.

interprocess communication

Two-way communication between active independent processes.

journaled guaranteed delivery

A method used by applications to guarantee BEA MessageQ message delivery in which the sending process sends a message that is delivered to the target disk queue.

journal file

A disk file that records all received and confirmed messages.

journaling

Writing to a auxiliary message recovery journal file.

journal replay

A method for resending messages stored in the DLJ or PCJ files.

link driver

A process that establishes a communications link between message queuing groups. Using the queuing engine, each link driver sends outbound messages and delivers inbound messages.

Linked List Sections

A set of global sections that is used to store the BEA MessageQ message buffers for a message queuing group. See also *buffer pool*.

message

A data item that is transmitted over a communications medium. A message contains a message header and data portion. The message header is comprised of attributes, which are defined by the application program, and context, which is added by the messaging tool.

message-based services

Predefined request, notification, and response messages exchanged between the application and BEA MessageQ server process.

message capture

A part of the Script Facility that provides a mechanism for viewing messages that are sent or received by an application.

message confirmation

An action taken by the receiver program, which indicates to the message queuing system that the processing of a recoverable message has completed. A message confirmation terminates the message system's responsibility for the recoverable message.

Message Control Section (MCS)

A global section that stores information about message queues and other global information, such as send and receive counters.

message delivery

The processing steps performed by the message queuing system when moving the message from a sender program to the receiver program's message queue.

message queue

An attachment point on the BEA MessageQ message queuing bus where pending messages are stored. A message queue is identified by a queue number and can be primary, secondary, or multireader.

message queuing

Interprocess communication and information exchange between two or more co-operating processes accomplished by directing messages to a memory- or disk-based queue as an intermediate storage point.

message queuing bus

A transparent communication mechanism that uses a simple logical bus topology. A message queuing bus provides a standard set of program-callable subroutines that allow message transfer between programs and message queues. See also *MessageQ message queuing bus*.

message queuing group

A set of logical addresses on the BEA MessageQ message queuing bus, all sharing a set of common BEA MessageQ resources. Each message queuing group resides on a single system. However, multiple groups can reside on the same system. The interconnections between groups define the extent of a message queuing bus.

Message Recovery Services (MRS)

A set of BEA MessageQ services that manage the automatic redelivery of critical messages.

Multipoint Outbound Target (MOT)

An entry point to a broadcast stream. A range of queue addresses is reserved to define a set of unique broadcast streams.

multireader queue (MRQ)

An optional queue type on the BEA MessageQ message queuing bus that stores messages that can be read by several simultaneous readers. Each reader, in turn, receives the next message in first-in/first-out (FIFO) order from the queue. A multireader queue can be permanent or permanently active. See also *queue type*.

naming

Pertaining to the use of a symbolic entity in place of an actual value. BEA MessageQ uses character strings for names, which, when translated, reveal queue addresses.

network

A collection of interconnected individual computer systems.

node

An individual computer system in a network that can communicate with other computer systems in the network.

nonblocking

Pertaining to an asynchronous style of message delivery where the program does not have to wait for an action to complete. The nonblocking style generally involves receiving an acknowledgment message when the action is complete. Contrast with *blocking*.

notification

A type of message-based service that supplies up-to-date information on events as they occur.

open deallocation

A method of duplex connection termination where the current sender can terminate the connection normally, regardless of which partner initiated the connection. Contrast with *initiator-only deallocation*.

operand

Data in the message header or message data structure that will be compared.

outbound conversation allocation

The allocation of conversations that are initiated by a CICS transaction program.

PAMS

Process Activation and Message Support. PAMS is the original name for the BEA MessageQ message queuing system. The BEA MessageQ API preserves the original product acronym in the name of each callable service to protect customer investment in application development.

PCJ file

See postconfirmation journal file.

pending

Pertaining to a message that is currently in a queue.

permanent outbound target

A type of outbound target that supports a method of message delivery where BEA MessageQ clients can request that outbound traffic be delivered to a predetermined BEA MessageQ queue. The queue must be a permanent queue in the designated group.

permanent queue

A message queue that is always at the same address on the BEA MessageQ message queuing bus. It exists regardless of whether a process is attached to it. A permanent queue retains its name and address after the process detaches, but loses any pending messages. See also *permanently active queue.* Contrast with *temporary queue*.

permanently active queue

A message queue that can receive messages without an application attachment. It retains its name and messages after the process detaches from BEA MessageQ. See also *permanent queue*.

platform

The combination of hardware, operating systems, and windowing systems that supports an application.

port server

A class of BEA MessageQ application that provides a connection to the BEA MessageQ message queuing bus for client applications executing on platforms

that do not have a BEA MessageQ implementation.

postconfirmation journal file (PCJ)

A disk file that holds confirmed recoverable messages that can be retrieved for audit trailing. Also called *PCJ file.*

primary queue

The one required queue used when a process attaches to the BEA MessageQ message queuing bus. There can be only one primary queue for each process. It is used as the default return address on all messages sent by that process. A primary queue can be permanent, permanently active, or temporary. See also *queue type*.

private broadcast stream

A MOT address range indicating that messages are restricted to distribution by one SBS Server, which restricts distribution to queues that have registered with that SBS Server. See also *broadcast stream.*

process

The basic entity scheduled by the system software, a process provides the context in which an image executes.

queue

See *message queue*.

queue address

A longword value that uniquely identifies the attachment point on the BEA MessageQ message queuing bus. An address includes a group ID and a queue number.

queue attribute

A specific characteristic of a queue that determines the features of the queue. Some examples of queue attributes are: permanent or temporary, recoverable or volatile, FIFO or non-FIFO capability, and so on.

queue number

A number that represents a unique location of a permanent or temporary queue address within a BEA MessageQ message queuing group. There must be at least one queue number for every process using the BEA MessageQ message queuing bus.

queue type

A description of a message queue as being primary, secondary, or multireader.

queuing engine

A process that handles all message traffic between message queuing groups. One queuing engine is created for each group. The queuing engine creates the global sections of memory for message queues within the group.

quota

The total amount of a system resource, such as disk space, that a job is allowed to use in an accounting period.

receive message quota

The application-defined limitation (in bytes) on pending messages in a queue.

receiver program

The application program in a connection that is accepting messages from the sender_program.

recoverable message

A message that is temporarily stored on a disk file and is guaranteed delivery if an application, system, or network fails.

registration

A stage in broadcast services where an application program subscribes to a broadcast stream by sending a registration message to the SBS Server.

Registry

A database in the Windows NT operating system that stores system and optional software configuration information.

reliable transmission

Pertaining to messages that are guaranteed to be delivered to a target queue. Contrast with *recoverable message*.

request

A type of message-based service that obtains information or registers to receive ongoing notifications.

response

A type of message-based service that provides information to fulfill requests or acknowledge registration and deregistration requests.

return-to-sender

A method of BEA MessageQ message delivery in which a nonrecoverable attempt is made to deliver a message. If the message cannot be delivered, it is returned to the sending process marked with a special return status.

SAF file

See store and forward.

Script facility

A productivity tool that speeds application testing by providing message simulation, capture, and replay abilities.

script file

A file with special syntax defining message information.

secondary queue

An optional private queue type used in conjunction with a primary queue. It provides a secondary address for messages. A secondary queue can be permanent, permanently active, or temporary. See also *queue type*.

Selective Broadcast Services (SBS)

BEA MessageQ services that enable an application to send a message to many receiving applications with a single send operation.

semaphore

In BEA MessageQ software, a common data structure used to serialize access to shared data structures.

sender notification

A component of the delivery mode that indicates how the sender program wants to receive information about the delivery of the message.

sender program

The application program in a connection that is sending messages to the receiver program.

sequence number

The message sequence number is generated by the BEA MessageQ message recovery system for each recoverable message. This value is passed to the receiver program in the PAMS status buffer (PSB) of the `pams_get_msg` function when it

reads each recoverable message.

server

A software module designed to perform a specific function for many users. See also *client* and *client/server model.*

sessionless

Pertaining to the absence of protocols required to manage communications between processes.

shared memory segment

A portion of memory that can be accessed by two or more processes.

simplex connection

An application protocol where the initiating partner is always the sender program and the accepting partner is always the receiver program. The receiver program can signal an error, but cannot send. Contrast with *duplex connection*.

source queue

A queue address of the program that sent the message.

stateless

Pertaining to the absence of protocols to identify the stages within a message transmission.

store and forward (SAF)

A message recovery journal that provides nonvolatile storage on the sender's system for automatic recovery and delivery of messages. Also called *SAF file.*

submission

A stage in broadcast services where an application program inserts a message on a broadcast stream.

synchronous

Pertaining to a message queuing system where the sender program must wait for a specific event or reply. Contrast with *asynchronous*.

target

A generic term for the client application with which another client application wants to establish a connection.

target queue

The queue address of the receiver program of the message.

TCP/IP

Transport Control Protocol/Internet Protocol. TCP/IP is a set of protocols that governs the transport of information between computers and networks of dissimilar types. Both Internet and UNIX based systems use TCP/IP protocols.

temporary queue

A queue that exists only for the duration of the process attachment to the BEA MessageQ message queuing bus. The assignment of the queue is not permanently defined. A temporary queue loses all messages in the queue when the process detaches from the queue. Contrast with *permanent queue*.

transparent

Relating to IBM systems, which use EBCDIC data encoding format. BEA MessageQ clients expect data in ASCII format. For a target defined as transparent, the LU6.2 Port Server does not provide data encoding format translation. Contrast with *nontransparent*.

type

A 16-bit piece of data BEA MessageQ uses to identify a kind of message from all other messages in the application. See also *class*.

undeliverable message action (UMA)

The action that occurs when the BEA MessageQ message queuing bus is unable to store a message. The UMA specifies the action to be taken with the recoverable message if it cannot be stored for guaranteed delivery by the message recovery system.

universal broadcast stream

A MOT address range indicating that messages can be distributed by all SBS Servers. Distribution is across the entire message queuing bus wherever SBS software is running. See also *broadcast stream*.

user process

A user's program image.

utility

A program that provides a set of related general-purpose functions, such as a pro-

gram development utility (an editor, a linker).

wait for dequeue

A method of BEA MessageQ message delivery in which the sending process is blocked until the message is read from the target queue by the receiver program.

wait for enqueue

A method of BEA MessageQ message delivery in which the sender program process is blocked until the message is written to the target queue. A return status indicates if the message is successfully written to the queue. This delivery method guarantees message delivery when message recovery services are not available on the target platform.