BEA Tuxedo

Introducing BEA Tuxedo ATMI

BEA Tuxedo Release 8.0
Document Edition 8.0
June 2001

Introducing BEA Tuxedo ATMI

Document Edition	Date	Software Version
8.0	June 2001	BEA Tuxedo Release 8.0

Contents

2. BEA Tuxedo ATMI Architecture

About This Document

This document provides a general introduction to the core BEA Tuxedo® ATMI (Application-to-Transaction Monitor Interface) programming environment.

This document includes the following topics:

- Chapter 1, "BEA Tuxedo System Fundamentals," provides an overview of the BEA Tuxedo programming environment.
- Chapter 2, "BEA Tuxedo ATMI Architecture," describes the basic architectural elements of a BEA Tuxedo ATMI environment, including external interfaces to the environment, the ATMI layer, the MIB, system services, and the ATMI environment's interface with standards-compliant resource managers.
- Chapter 3, "Three Ways of Viewing the BEA Tuxedo ATMI Infrastructure," describes the BEA Tuxedo ATMI infrastructure from three perspectives: administrative or management, development (using the ATMI), and run time.

What You Need to Know

This document is intended for programmers who want to familiarize themselves with the BEA Tuxedo programming environment and create distributed ATMI applications using the BEA Tuxedo product.

e-docs Web Site

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

How to Print the Document

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

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

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

Related Information

The following documents provide related information about BEA Tuxedo software.

- *Installing the BEA Tuxedo System*—paper copy distributed with the CD
- *BEA Tuxedo Release Notes*—paper copy distributed with the CD
- *Setting Up a BEA Tuxedo Application*—available through the BEA Tuxedo Online Documentation CD, this guide describes how to set up and administer the BEA Tuxedo system.

- *Administering a BEA Tuxedo Application at Run Time*—available through the BEA Tuxedo Online Documentation CD, this guide describes how to administer BEA Tuxedo applications at run time.
- *Getting Started with BEA Tuxedo CORBA Applications*—available through the BEA Tuxedo Online Documentation CD, this guide describes how to develop distributed CORBA applications in the BEA Tuxedo CORBA environment.

For more information about configuring and administering BEA Tuxedo ATMI environment, refer to the *CORBA Bibliography* at http://edocs.bea.com/.

Contact Us!

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

In your e-mail message, please indicate that you are using the documentation for the BEA Tuxedo 8.0 release.

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

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

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

Documentation Conventions

The following documentation conventions are used throughout this document.

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

`monospace text`	Indicates code samples, commands and their options, data structures and their members, data types, directories, and filenames 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

Convention	Item
`{ }`	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.

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

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

CHAPTER

1 BEA Tuxedo System Fundamentals

This topic includes the following sections:

- What Is the BEA Tuxedo System?
- Anatomy of the Client/Server Model
- How the BEA Tuxedo System Fits into the Client/Server Model
- What Are Clients, Servers, and Services in a BEA Tuxedo Environment?
- Services Provided by the BEA Tuxedo System
- BEA Family of Products

For More Information

Many resources are available to help you understand the BEA Tuxedo system. The following books, white papers, and presentations provide information about client/server architecture, building and managing distributed business applications, and using the BEA Tuxedo system to build and manage enterprise applications:

- Andrade, Juan, M. Carges, T. Dwyer, and S. Felts, *The Tuxedo System - Software for Constructing and Managing Distributed Business Applications*. Reading, Massachusetts: Addison-Wesley Publishing, 1996.
- Edwards, Jeri, with D. DeVoe, *3-Tier Client/Server at Work*. New York: John Wiley & Sons, Inc., April 1997.

- Edwards, Jeri, D. Harkey, R. Orfali, *The Essential Client/Server Survival Guide*. New York: John Wiley & Sons, Inc., May 1997.
- Hall, Carl, *Building Client/Server Applications Using Tuxedo - Designing and Building Cost-Effective, High Performance Client/Server Applications Using Tuxedo*. Wiley Computer Publishing.
- Lee, Rich, *BEA Tuxedo Essentials*. Presented at the BEA User's Conference in New Orleans, La., February, 1999.
- MacBlane, Randy, *Managing your BEA Tuxedo Applications Even Over the Internet.* Presented at the BEA User's Conference in San Jose, Ca., May 1997.
- MacBlane, Randy, *Tuxedo's Management Information Base*. Presented at the BEA User's Conference in San Francisco, Ca., February 1996.
- *BEA Tuxedo: The Programming Model* (White Paper)
- *BEA Tuxedo and the Component Software Model* (White Paper)
- *Inter-Application Transaction Processing with BEA Tuxedo Domains* (White Paper)
- *Reliable Queuing Using BEA Tuxedo* (White Paper)

What Is the BEA Tuxedo System?

The BEA Tuxedo system is a *middleware* product that distributes applications across multiple platforms, databases, and operating systems using message-based communications and, if desired, distributed transaction processing.

Middleware is used with client/server applications to distribute processing among multiple servers, manage distributed transactions, and integrate multiple database platforms. Middleware systems are sometimes known as “on-line transaction processing” or “OLTP” systems.

The BEA Tuxedo system is a mature product based on over 15 years of development from a diverse group of technology companies including AT&T, UNIX System Laboratories (USL), Novell, and BEA Systems, Inc. It is both a development platform and an execution platform. The BEA Tuxedo system serves as an extension to the operating system.

The BEA Tuxedo system provides the following:

- An industry standard for the creation and central administration of distributed on-line transaction applications in a heterogeneous client/server environment.
- Ease of use for application developers, who do not need to know all the details about server locations, routing, or platforms used. In a BEA Tuxedo application, these aspects of a program are transparent.
- The fundamental underpinnings for creating, managing, and maintaining reliable, high performance, easily managed distributed systems.

Features of the BEA Tuxedo System

The BEA Tuxedo system offers many features to accommodate the needs of the administrator, architect, and programmer of an application.

Administrative Features

- Password security and access control security—password security allows application designers to control access by requiring passwords at initialization time (authentication). Further control is available through authorization, a means of restricting access to certain application services to clients that have been given explicit permission and that have authenticated identities.
- System events notification—the BEA Tuxedo system provides details about system events, such as servers dying and network failures. When an event is posted by clients or servers, the EventBroker looks up all the subscribers to that event and takes appropriate actions, as determined by each subscription.
- The MIB (Management Information Base)—an administrative interface that enables you to monitor, configure, and tune your application through your own programs. It is an implementation-independent management database defined as a set of FML attributes, which allows you to query or change information.

- Web-based administration—a graphical user interface, available through the World Wide Web, for the configuration and control of BEA Tuxedo applications.

Architectural Features

- Distributed services—allow transparent access to application and/or system services located on different hardware platforms.
- Fast, connectionless communications—clients connect to a bulletin board rather than to servers, thus improving system performance.
- Scalability—you can quickly scale your application to match varying system load demands because services and servers can be replicated and distributed easily. You can set thresholds programmatically to enable the BEA Tuxedo system to spawn new servers or to shut down servers automatically.
- Server transparency—the directory of services on the bulletin board maps service names to servers; clients do not need to be aware of server identity.

Programming Features

- Communication techniques—the application programming interface (API) for the BEA Tuxedo system is a superset of X/Open's XATMI interface called the Application-to-Transaction Monitor Interface or ATMI. The Tuxedo ATMI is a rich set of communication techniques for writing distributed applications.
- Distributed Transaction Processing (DTP)—allows work being done throughout a distributed application to be atomically completed—an essential characteristic of any OLTP system.
- Typed buffers—provides transparent handling of application data across heterogeneous platforms.
- X/Open TX compliance—the BEA Tuxedo system conforms to the X/Open interface standard for transaction demarcation.
- X/Open XA compliance—the BEA Tuxedo system conforms to the X/Open interface standard for transaction database systems (called resource managers). As a result, you can mix and match databases within one application while maintaining data integrity.

See Also

- "Services Provided by the BEA Tuxedo System" on page 1-13
- "BEA Family of Products" on page 1-15

Anatomy of the Client/Server Model

In client/server architecture, clients, or programs that represent users who need services, and servers, or programs that provide services, are separate logical objects that communicate over a network to perform tasks together. A client makes a request for a service and receives a reply to that request; a server receives and processes a request, and sends back the required response.

Characteristics of Client/Server Architecture

- Asymmetrical protocols—there is a many-to-one relationship between clients and a server. Clients always initiate a dialog by requesting a service. Servers wait passively for requests from clients.
- Encapsulation of services—the server is a specialist: when given a message requesting a service, it determines how to get the job done. Servers can be upgraded without affecting clients as long as the published message interface used by both is unchanged.
- Integrity—the code and data for a server are centrally maintained, which results in cheaper maintenance and the protection of shared data integrity. At the same time, clients remain personal and independent.
- Location transparency—the server is a process that can reside on the same machine as a client or on a different machine across a network. Client/server software usually hides the location of a server from clients by redirecting service requests. A program can be a client, a server, or both.
- Message-based exchanges—clients and servers are loosely-coupled processes that can exchange service requests and replies using messages.

- Modular, extensible design—the modular design of a client/server application enables that application to be fault-tolerant. In a fault-tolerant system, failures may occur without causing a shutdown of the entire application. In a fault-tolerant client/server application, one or more servers may fail without stopping the whole system as long as the services offered on the failed servers are available on servers that are still active. Another advantage of modularity is that a client/server application can respond automatically to increasing or decreasing system loads by adding or shutting down one or more services or servers.
- Platform independence—the ideal client/server software is independent of hardware or operating system platforms, allowing you to mix client and server platforms. Clients and servers can be deployed on different hardware using different operating systems, optimizing the type of work each performs.
- Reusable code—service programs can be used on multiple servers.
- Scalability—client/server systems can be scaled horizontally or vertically. Horizontal scaling means adding or removing client workstations with only a slight performance impact. Vertical scaling means migrating to a larger and faster server machine or adding server machines.
- Separation of Client/Server Functionality—client/server is a relationship between processes running on the same or separate machines. A server process is a provider of services. A client is a consumer of services. Client/server provides a clean separation of functions.
- Shared resources—one server can provide services for many clients at the same time, and regulate their access to shared resources.

Differences Between 2-Tier and 3-Tier Client/Server Architectures

Every client/server application contains three functional units:

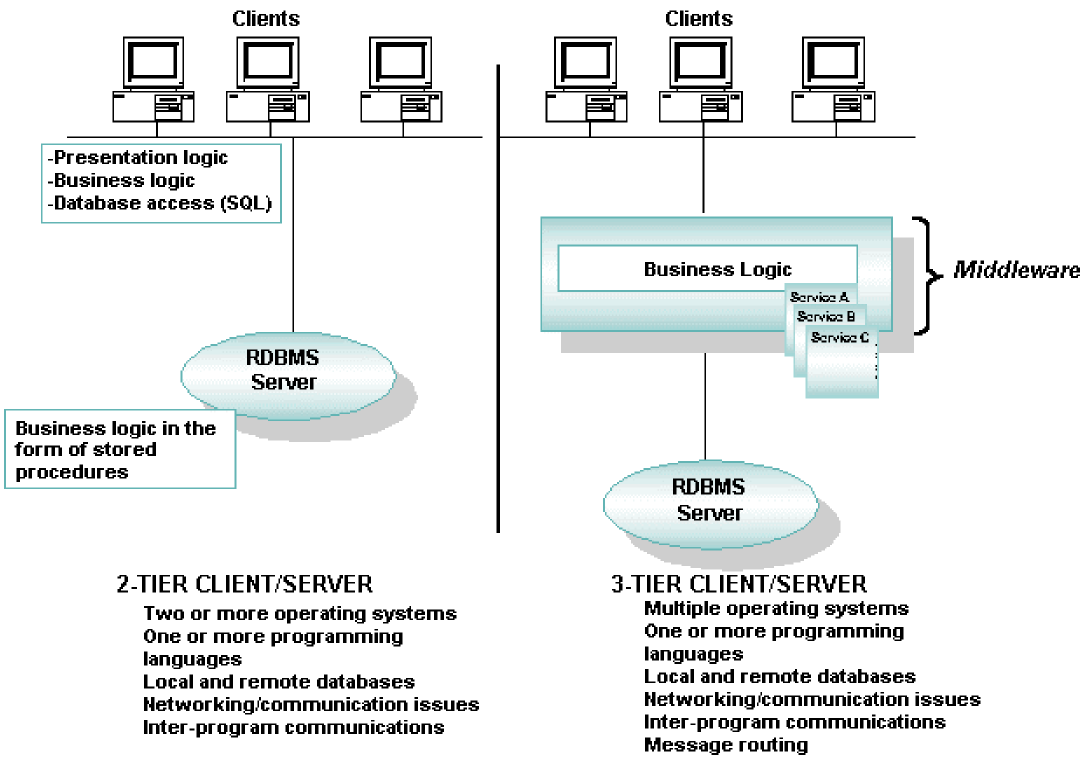
- Presentation logic or user interface (for example, ATM machines)
- Business logic (for example software that enables a customer to request an account balance)
- Data (for example, records of customer accounts)

These functional units can reside on either the client or on one or more servers in your application. Which of the many possible variations you choose depends on how you split the application and which middleware you use to communicate between the tiers.

In 2-tier client/server applications, the business logic is buried inside the user interface on the client or within the database on the server in the form of stored procedures. Alternatively, the business logic can be divided between the client and server. File servers and database servers with stored procedures are examples of 2-tier architecture.

In 3-tier client/server applications, the business logic resides in the middle tier, separate from the data and user interface. In this way, processes can be managed and deployed separately from the user interface and the database. Also, 3-tier systems can integrate data from multiple sources.

Figure 1-1 2-Tier and 3-Tier Client/Server Models


Clients
-Presentation logic
-Business logic
-Database access (SQL)
RDBMS Server
Business logic in the form of stored procedures
2-TIER CLIENT/SERVER
Two or more operating systems
One or more programming languages
Local and remote databases
Networking/communication issues
Inter-program communications
Clients
Business Logic
Service A
Service B
Service C
Middleware
RDBMS Server
3-TIER CLIENT/SERVER
Multiple operating systems
One or more programming languages
Local and remote databases
Networking/communication issues
Inter-program communications
Message routing

Client/Server Variations to Suit Your Needs

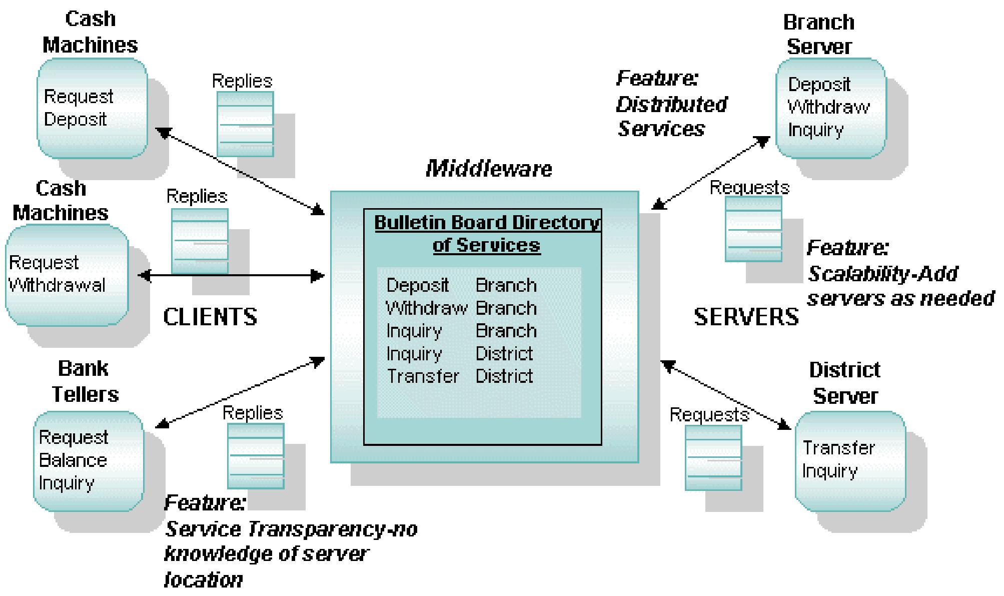
Client/server architecture can accommodate the needs of each of the following situations:

- Small shops and laptops—the client, the middleware software, and most of the business services operate on the same machine. We recommend this approach for one-person businesses such as a dentist's office, a home office, and a business traveler who frequently works on a laptop computer.
- Small businesses and corporate departments—a LAN-based single-server application is required. Users of this type of application include small businesses, such as a medical practice with several doctors, a multi-department corporation, or a bank with several branch offices. In this type of application, multiple clients talk to a local server. Administration is simple: security is implemented at the machine level and failures are detected easily.
- Large enterprises—multiple servers that offer diverse functionality are required. Multiple servers can reside on the Internet, intranets, and corporate networks, all of which are highly scalable. Servers can be partitioned by function, resources, or databases, and can be replicated for increased fault tolerance or enhanced performance. This model provides a great amount of power and flexibility. How well you architect your application is critical to this client/server model. You may need to partition work among servers, or design servers to delegate work to other servers.

How the BEA Tuxedo System Fits into the Client/Server Model

The BEA Tuxedo system fits into the middle of the client/server model. In a BEA Tuxedo application, clients log in and request services offered by an application. The BEA Tuxedo system offers these services through a transparent bulletin board. The bulletin board contains a directory advertising services. In a banking application, for example, the bulletin board might advertise deposit, withdrawal, and inquiry services. The BEA Tuxedo system then finds a server (for example, at the appropriate branch or district office) that can provide the requested services.

Figure 1-2 Clients and Servers in a Sample Banking Application



The preceding figure shows the primary building blocks of a BEA Tuxedo application:

- Clients—programs that collect input from users, sends requests through the BEA Tuxedo system to servers, and then collects the replies from servers and delivers them to the users.

- Servers—programs that encapsulate the business logic into a set of services that define the application.
- Middleware—comprises all the distributed software needed to support interactions between clients and servers. It is the medium that enables a client to obtain a service from a server. Middleware includes: API functions used by the client (to issue requests and receive replies) and the server (to issue replies) and messaging paradigms used to transmit client requests and server responses over a network. Middleware does not include any of the following: the user interface on the client, application logic, and services provided by servers.

In this sample BEA Tuxedo banking application, clients (cash machines and tellers) make requests, and servers (at branch and district offices) provide services and responses. For example, a customer may use a cash machine to find out how much money is available in his personal checking account. The cash machine (a client) calls the server to get the balance. The server receives the request, retrieves the balance, and sends the information to the cash machine.

See Also

- "Anatomy of the Client/Server Model" on page 1-5
- "What Are Clients, Servers, and Services in a BEA Tuxedo Environment?" on page 1-12

What Are Clients, Servers, and Services in a BEA Tuxedo Environment?

This topic describes a client, server, and services in a BEA Tuxedo environment.

What Is a BEA Tuxedo Client?

A client is a program that collects a request from a user and passes that request to a server capable of fulfilling it. It can reside on a PC or workstation as part of the front end of an application. It can also be embedded in software that reads a communication device such as an ATM machine from which data is collected and formatted before being processed by BEA Tuxedo servers.

To be a client, a program must be able to invoke the BEA Tuxedo libraries of functions and procedures known collectively as the Application-to-Transaction-Monitor Interface, or ATMI. The ATMI is supported in several language bindings.

A client joins a Tuxedo application by calling the ATMI client initialization routine. Once it has joined an application, a client can define transaction boundaries and call ATMI functions that enable it to communicate with other programs in your application. The client leaves the BEA Tuxedo ATMI application by issuing an ATMI termination function. By joining an application only when necessary and leaving it once the appropriate task is complete, a client frees BEA Tuxedo system resources for use by other clients and servers.

When building a distributed application, you must determine how information is gathered and presented to your business for processing. You have complete control over where and when to call ATMI or CORBA functions, depending upon your business logic and rules. Your program can join one BEA Tuxedo application, perform some tasks and leave, and then join a different BEA Tuxedo application to perform another task. If you are using a multicontext application, your client can perform tasks in more than one application without leaving any of them.

What Is a BEA Tuxedo Server?

A BEA Tuxedo server is a process that oversees a set of services, dispatching them automatically for clients that request them. A service, in turn, is a function within a server program that performs a particular task needed by a business. A bank, for example, might have one service that accepts deposits and another that reports account balances. A server at this bank might receive requests from clients for both services. It is the server's job to dispatch each request to the appropriate service.

Service functions implement business logic through calls to database interfaces such as SQL and, possibly, calls to the ATMI to access additional services, queues, and other resources. The servers on which these services reside then reply to the clients or forward client requests to a new service.

What Are BEA Tuxedo Services?

A service is a module of application code that performs a task. Services are compiled and link edited to form executable servers.

Services Provided by the BEA Tuxedo System

The BEA Tuxedo system offers many administrative and application processing services to help you streamline and administer your application.

Administrative Services

The BEA Tuxedo system provides services for the following administrative tasks:

- Application queue management
- Centralized application configuration
- Distributed application management

- Dynamic application reconfiguration
- Event management
- Security management
- Startup and shutdown of an application
- Transaction management
- Workstation management

Application Processing Services

The BEA Tuxedo system provides services that enable you to implement the following functionality in your application:

- Data compression
- Data-dependent routing
- Data encoding
- Data encryption
- Data marshalling
- Load balancing
- Message prioritization
- Service and event naming

BEA Family of Products

The BEA product family facilitates end-to-end integration of heterogeneous hardware and software environments allowing businesses to create enterprise-wide transaction processing systems. BEA products enable companies to enjoy the benefits of robust mission-critical applications with the flexibility of distributed client/server computing. Compliant with all leading industry standards, BEA products enable developers to build, deploy, manage, and connect enterprise-wide applications on more than 70 platforms. These products also provide complete integration with market-leading application development tools, systems management solutions, and legacy applications.

This Product...	Provides...
BEA eLink Adapter for Mainframe	A suite of connectivity products that allow seamless integration of BEA Tuxedo distributed applications with enterprise applications.
BEA Jolt	A BEA Tuxedo client API in Java. BEA Jolt takes requests from Java-enabled clients and translates them into BEA Tuxedo application calls.
BEA Tuxedo and BEA Log Central	BEA application management products that provide a complete environment for managing, integrating, and deploying BEA Tuxedo and BEA WebLogic Server applications.
BEA Tuxedo ATMI consists of 4 components: ■ Core BEA Tuxedo ATMI ■ Domains ■ /Q ■ Workstation	■ BEA Tuxedo ATMI core product—enables you to build high-performance, mission-critical, and reliable distributed applications. It provides the framework for building scalable 3-tier client-server applications in heterogeneous, distributed environments. ■ Domains—extends the BEA Tuxedo client/server model to provide transaction interoperability across separately administered BEA Tuxedo applications. ■ /Q—allows reliable queueing of requests. ■ Workstation—offers full client support for a wide variety of operating systems, allowing applications to use remote clients that don’t need a full BEA Tuxedo implementation.

This Product...	Provides... (Continued)
BEA Tuxedo CORBA consists of 5 components: ■ BEA Tuxedo CORBA core product ■ Bootstrap Object ■ IIOP Listener/Handler ■ ORB Client/Server ■ TP Framework	■ BEA Tuxedo CORBA core product—the Common Object Request Broker Architecture (CORBA) component in BEA Tuxedo uses distributed object technology to provide a rich set of programming models by extending the Object Request Broker (ORB) model with online transaction processing (OLTP) functions. ■ Bootstrap object—establishes communication between an client application and a BEA Tuxedo domain. ■ IIOP Listener/Handler—a process that retrieves a client request and delivers it to the appropriate server application. ■ ORB—serves as an intermediary for requests that client applications send to server applications. ■ TP Framework—provides a programming model that achieves high levels of performance while shielding the application programmer from the complexities of the CORBA interfaces.
BEA WebLogic Server	A Java-application server for developing, integrating, deploying, and managing large-scale, distributed Web, network, and database applications.
BEA WebLogic Collaborate	An XML- and Java-based open market electronic commerce platform for implementing business-to-business e-commerce systems on the Web.
BEA WebLogic Commerce Server & Personalization Server	Enables rapid deployment of adaptable and personalized e-commerce applications to accelerate response time to customer and market demands.
BEA WebLogic Process Integrator	Automates and integrates a business process by managing the sequence of activities and invoking the appropriate resources required by the various activities or steps in the process.

CHAPTER

2 BEA Tuxedo ATMI Architecture

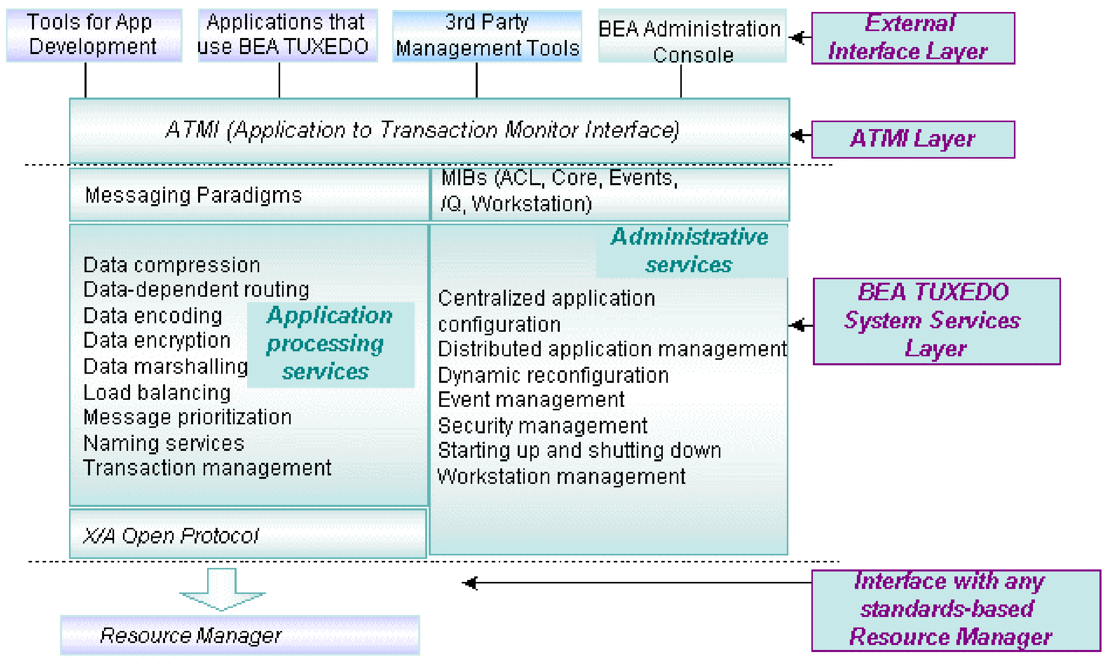
This topic includes the following sections:

- Basic Architecture of the BEA Tuxedo ATMI Environment
- What Are the BEA Tuxedo ATMI Messaging Paradigms?
- How BEA Tuxedo ATMI Processes Messages
- BEA Tuxedo ATMI Application Processing Services
- BEA Tuxedo ATMI Administrative Services

Basic Architecture of the BEA Tuxedo ATMI Environment

The following figure illustrates the basic architectural elements of a BEA Tuxedo ATMI environment: external interfaces to the environment, the ATMI layer, the MIB, BEA Tuxedo system services, and the environment’s interface with standards-compliant resource managers.

Figure 2-1 The BEA Tuxedo ATMI Basic Architecture


Tools for App Development
Applications that use BEA TUXEDO
3rd Party Management Tools
BEA Administration Console
External Interface Layer
ATMI (Application to Transaction Monitor Interface)
ATMI Layer
Messaging Paradigms
MIBs (ACL, Core, Events, /Q, Workstation)
Data compression
Data-dependent routing
Data encoding
Data encryption
Data marshalling
Load balancing
Message prioritization
Naming services
Transaction management
Application processing services
Administrative services
Centralized application configuration
Distributed application management
Dynamic reconfiguration
Event management
Security management
Starting up and shutting down
Workstation management
BEA TUXEDO System Services Layer
X/A Open Protocol
Resource Manager
Interface with any standards-based Resource Manager

As shown in this illustration, the BEA Tuxedo ATMI environment contains the following components:

Architectural Part	Description
External interface layer	This layer consists of interfaces between the user and the environment. It includes both tools for application development and administration, such as the BEA Administration Console. The BEA Administration Console can interact with standard management consoles. Thus a user can manage a BEA Tuxedo ATMI environment and a network configuration from one console. In addition, application architects and developers can build their own administrative tools or application- or market-specific tools on top of the MIB.

Architectural Part	Description (Continued)
ATMI (Application-to-Transaction Monitor Interface)	The interface between an application and the BEA Tuxedo ATMI environment. The ATMI and the BEA Tuxedo environment implement the X/Open DTP model of transaction processing. An abstract environment, the ATMI supports location transparency and hides implementation details. As a result, programmers are free to configure and deploy BEA Tuxedo applications to multiple platforms without modifying the application code.
Messaging paradigms	Different models of transferring messages between a client and a server. Examples include request/response mode, conversational mode, events and unsolicited notification.
Management Information Base (MIB)	The MIB is an interface that enables users to program and administer a BEA Tuxedo ATMI environment easily. MIB operations enable you to perform all management tasks (monitoring, configuring, tuning, and so on). The MIB allows you to perform one task to one object at a time or to build toolkits with which you can batch tasks and/or objects. (For information about available MIBs, see "Available BEA Tuxedo MIBs" on page 3-3.)
BEA Tuxedo Services (administrative services and application processing services)	Services and/or capabilities provided by the BEA Tuxedo ATMI environment infrastructure for developing and administering applications. The application processing services available to developers include: data compression, data-dependent routing, data encoding, load balancing, and transaction management. The administrative services include: centralized application configuration, distributed application management, domains partitioning, dynamic reconfiguration, event and fault management, IPC message queues, and workstation management. (For information on administrative services, see the topic titled: "Three Ways of Viewing the BEA Tuxedo ATMI Infrastructure" on page 3-1.)
Resource Manager	A software product in which data is stored and available for retrieval through application-based queries. The resource manager (RM) interacts with the BEA Tuxedo ATMI environment and implements the XA standard interfaces. The most common example of a resource manager is a database. Resource managers provide transaction capabilities and permanence of actions; they are the entities accessed and controlled within a global transaction.

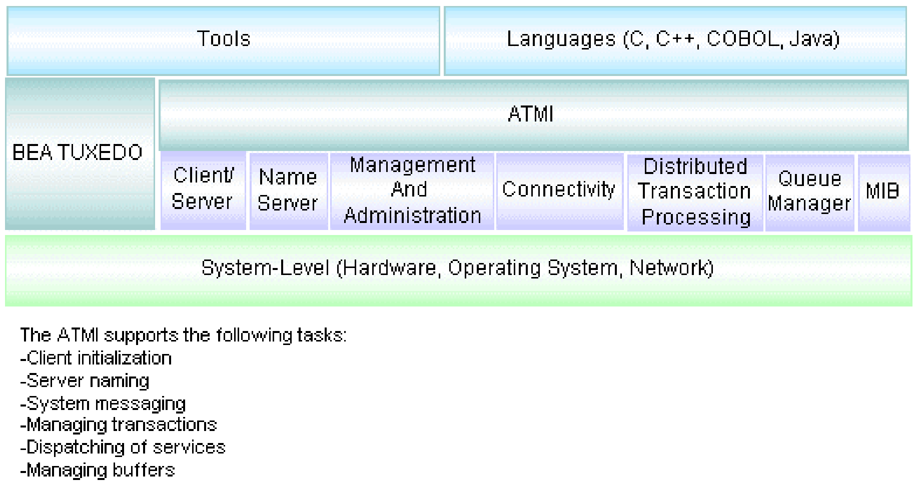
See Also

- "BEA Tuxedo ATMI Administrative Services" on page 2-44
- "BEA Tuxedo ATMI Application Processing Services" on page 2-30

What You Can Do Using the ATMI

The Application-to-Transaction Monitor Interface (ATMI), the BEA Tuxedo API, is an interface for communications, transactions, and management of data buffers that works in all environments supported by the BEA Tuxedo system. It provides the connection between application programs and the BEA Tuxedo system. The ATMI is a simple interface for a comprehensive set of capabilities. It implements the X/Open DTP model of transaction processing.

Figure 2-2 Using the ATMI



The ATMI library offers you a variety of functions for defining and controlling global transactions in a BEA Tuxedo application. Global transactions enable you to manage exclusive units of work spanning multiple programs and resource managers in your distributed application. All work in a single transaction is treated as a logical unit, so that if any one program cannot complete its task successfully, no work is performed by programs in the transaction. Most ATMI functions support different communication styles. These functions knit together distributed programs by enabling them to send and receive data. All ATMI functions send or receive data in typed buffers. Following is a list of ATMI functions (for C and COBOL bindings), and the tasks they perform. The functions are grouped by task.

Table 2-1 Using the ATMI Functions

For a Task Related to...	Use This C Function...	Or This COBOL Function...	To...
Client membership	`tpchkauth(3c)`	`TPCHKAUTH(3cbl)`	Check whether authentication is required
	`tpinit(3c)`	`TPINITIALIZE(3cbl)`	Have a client join an application
	`tpterm(3c)`	`TPTERM(3cbl)`	Have a client leave an application
Buffer management	`tpalloc(3c)`	`N/A`	Create a message buffer
	`tprealloc(3c)`	`N/A`	Resize a message buffer
	`tpfree(3c)`	`N/A`	Free a message buffer
	`tptypes(3c)`	`N/A`	Get a message type and subtype
Message priority	`tpgprio(3c)`	`TPGPRIO(3cbl)`	Get the priority of the last request
	`tpsprio(3c)`	`TPSPRIO(3cbl)`	Set the priority of the next request
Request/response communications	`tpcall(3c)`	`TPCALL(3cbl)`	Initiate a synchronous request/response to a service
	`tpacall(3c)`	`TPACALL(3cbl)`	Initiate an asynchronous request (fanout)
	`tpgetrply(3c)`	`TPGETRPLY(3cbl)`	Receive an asynchronous response
	`tpcancel(3c)`	`TPCANCEL(3cbl)`	Cancel an asynchronous request

Table 2-1 Using the ATMI Functions (Continued)

For a Task Related to...	Use This C Function...	Or This COBOL Function...	To...
Conversational communications	`tpconnect(3c)`	`TPCONNECT(3cbl)`	Begin a conversation with a service
	`tpdiscon(3c)`	`TPDISCON(3cbl)`	Abnormally terminate a conversation
	`tpsend(3c)`	`TPSEND(3cbl)`	Send a message in a conversation
	`tprecv(3c)`	`TPRECV(3cbl)`	Receive a message in a conversation
Reliable queuing	`tpenqueue(3c)`	`TPENQUEUE(3cbl)`	Enqueue a message to a message queue
	`tpdequeue(3c)`	`TPDEQUEUE(3cbl)`	Dequeue a message from a message queue
Event-based communications	`tpnotify(3c)`	`TPNOTIFY(3cbl)`	Send an unsolicited message to a client
	`tpbroadcast(3c)`	`TPBROADCAST(3cbl)`	Send messages to several clients
	`tpsetunsol(3c)`	`TPSETUNSOL(3cbl)`	Set unsolicited message call-back
	`tpchkunsol(3c)`	`TPCHKUNSOL(3cbl)`	Check the arrival of unsolicited messages
	`N/A`	`TPGETUNSOL(3cbl)`	Get an unsolicited message
	`tppost(3c)`	`TPPOST(3cbl)`	Post an event message
	`tpsubscribe(3c)`	`TPSUBSCRIBE(3cbl)`	Subscribe to event messages
	`tpunsubscribe(3c)`	`TPUNSUBSCRIBE(3cbl)`	Unsubscribe to event messages

Table 2-1 Using the ATMI Functions (Continued)

For a Task Related to...	Use This C Function...	Or This COBOL Function...	To...
Transaction management	`tpbegin(3c)`	`TPBEGIN(3cbl)`	Begin a transaction
	`tpcommit(3c)`	`TPCOMMIT(3cbl)`	Commit the current transaction
	`tpabort(3c)`	`TPABORT(3cbl)`	Roll back the current transaction
	`tpgetlev(3c)`	`TPGETLEV(3cbl)`	Check whether in transaction mode
	`tpsuspend(3c)`	`TPSUSPEND(3cbl)`	Suspend the current transaction
	`tpresume(3c)`	`TPRESUME(3cbl)`	Resume a transaction
Service entry and return	`tpsvrinit(3c)`	`TPSVRINIT(3cbl)`	Initialize a server
	`tpsvrdone(3c)`	`TPSVRDONE(3cbl)`	Terminate a server
	`tpservice(3c)`	`N/A`	Prototype for a service entry point
	`N/A`	`TPSVCSTART(3cbl)`	Get service information
	`tpreturn(3c)`	`TPRETURN(3cbl)`	End a service function
	`tpforward(3c)`	`TPFORWAR(3cbl)`	Forward request
Dynamic advertisement	`tpadvertise(3c)`	`TPADVERTISE(3cbl)`	Advertise a service name
	`tpunadvertise(3c)`	`TPUNADVERTISE(3cbl)`	Unadvertise a service name
Resource management	`tpopen(3c)`	`TPOPEN(3cbl)`	Open a resource manager
	`tpclose(3c)`	`TPCLOSE(3cbl)`	Close a resource manager

Note: The use of ATMI transaction management functions is optional.

See Also

- "Using the ATMI to Handle System and Application Errors" on page 2-28 in *Administering a BEA Tuxedo Application at Run Time*

What Are the BEA Tuxedo ATMI Messaging Paradigms?

The following table describes the BEA Tuxedo ATMI messaging paradigms available to application developers.

Table 2-2 BEA Tuxedo ATMI Messaging Paradigms

BEA Tuxedo ATMI Messaging Paradigm	Description
Conversational communication	Service request mode involving multiple 2-way interactions between a client and a dedicated server.
Event-based communication	Publish/subscribe mode.
Queue-based communication	Guaranteed delivery mode.
Request/reply communication	Service request mode that can be synchronous (processing waits until the requester receives the response) or asynchronous (processing continues while the requester waits for the response).
Unsolicited messaging	Communication from any client or server to any clients that were not requested or expected by those clients.

See Also

- "What Is Conversational Communication?" on page 2-9
- "How the EventBroker Works" on page 2-10
- "What Is Queue-based Communication?" on page 2-13
- "What Is Request/Reply Communication?" on page 2-14
- "What Is Unsolicited Communication?" on page 2-17
- "What Are Nested and Forwarded Service Requests?" on page 2-18

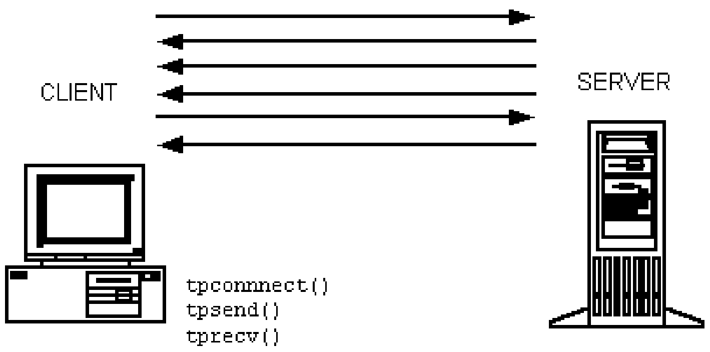
What Is Conversational Communication?

Conversational communication is the BEA Tuxedo system implementation of a human-like paradigm for exchanging messages between clients and servers. In this form of communication, a virtual connection is maintained between the client and server. Just as in a conversation between two people, a number of messages pass back and forth between the two entities until a conclusion is reached. Over the course of the communication, both sides "remember" the point (or state) of the conversation so that relatively long operations, such as ad hoc queries, reports, and file transfers, can be supported. Conversational servers are available by default, but more can be spawned automatically if needed.

The BEA Tuxedo system provides an application programming interface (API) that can be used to create conversations in applications; specifically to connect clients to servers, to send and receive messages, and to end the conversation.

Conversations can be nested but performance may be degraded as a result of doing so. Conversations may contain either transactions or service requests as appropriate. Although a conversational service can make service calls and establish conversations, those service calls and conversations cannot be forwarded. A conversation can be within the scope of, and controlled by a transaction.

Figure 2-3 Conversational Communication



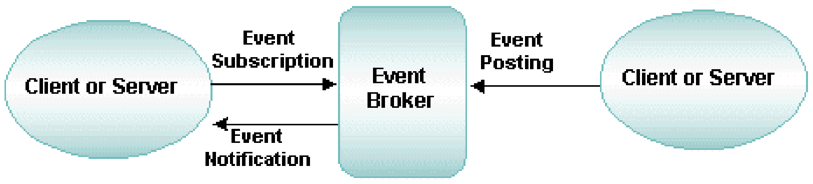
See Also

- "Using Conversational Communication" on page 1-11 in *Tutorials for Developing BEA Tuxedo ATMI Applications*

How the EventBroker Works

The BEA Tuxedo EventBroker provides a communication paradigm in which an arbitrary number of suppliers can post messages for an arbitrary number of subscribers. Because client and server processes that use the EventBroker communicate with one another based on a set of *subscriptions*, this paradigm is known as *publish-and-subscribe* communication. The EventBroker acts like a newspaper delivery person who delivers newspapers only to customers who have paid for a subscription.

Figure 2-4 Posting and Subscribing to an Event



Event generators (either clients or servers) inform the EventBroker of changes and problems as they occur. This process is called posting an event. The EventBroker then matches the name of the event to an event name associated with a list of subscribers, and notifies each subscriber on the list of the event.

See Also

- "What Types of Events Are Reported?" on page 2-11
- "How Are Events Reported?" on page 2-12
- "Using Event-based Communication" on page 1-14 in *Tutorials for Developing BEA Tuxedo ATMI Applications*

What Types of Events Are Reported?

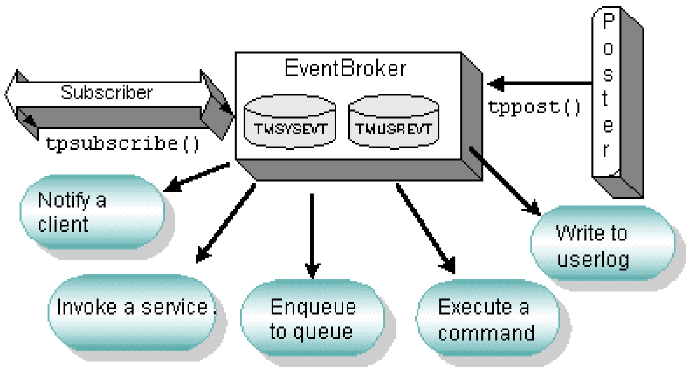
The BEA Tuxedo system supports two different types of event reports:

- System Event reports—provide details about BEA Tuxedo system events, such as servers dying, and network failures. When an event is posted by clients or servers, EventBroker matches the posted event's name to subscriber's of the same events and takes appropriate action determined by each subscription.
- Reports of User Events or Application-Defined Events—allow application programs to post events when certain criteria are met. A banking application, for example, might post an event for withdrawals over a certain limit.

How Are Events Reported?

The EventBroker provides publish-and-subscribe functionality. A process registers a subscription with the EventBroker, indicating interest in a particular event. Subsequently, whenever the EventBroker is notified by another process that the specified event has occurred, the EventBroker reports the occurrence to any process that has subscribed for this event.

Figure 2-5 Event-based Messaging



The EventBroker uses several mechanisms for publishing (that is, issuing notices of) events:

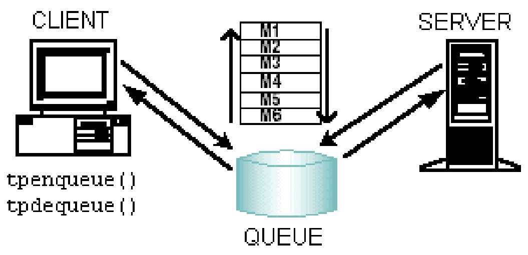
- Disk-based queuing
- Asynchronous service calls
- User log entries
- Unsolicited messages
- System commands

What Is Queue-based Communication?

The BEA Tuxedo system offers a queue-based architecture known as /Q for applications that require persistent storage of data. The /Q component allows any client or server to store messages or service requests in queues and guarantees that any stored request is sent through the transaction protocol to ensure safe storage.

BEA Tuxedo system queues can be ordered as LIFO (last in, first out) or FIFO (first in, first out), or on the basis of time or priority. A collection of queues is administered and referred to as a single entity known as a queue space.

Figure 2-6 Queue-based Messaging



Using Application Queues

Application queues are appropriate if you must communicate in a time-independent fashion. Time-independence is a characteristic of programs that operate independently from one another and do not need to synchronize their communications simultaneously. Time-independent programs synchronize by leaving messages for each other in application queues. Messages can be dequeued in any of several ordering schemes, such as first in, first out (FIFO) order, priority order, or time-based order. BEA Tuxedo client and server programs can enqueue messages and dequeue messages from queues. More than one client and server can access the same queue.

To use an application queue, your program must name the queue to be accessed and the queue space in which it resides. Your application can use more than one queue space and each space can contain more than one message queue.

Because application queues reside on a disk, the availability of stored messages is guaranteed even after machine failures. To determine when the use of application queues is appropriate, you need to determine when time-independent synchronization occurs in your business, for example, in filling orders. Orders can be enqueued to disk and depending on specific order criteria, such as items or shipment location, placed in different queue spaces. Within each queue space, you can determine additional criteria, such as cost, state, and so on.

See Also

- "Using Queue-based Communication" on page 1-15 in *Tutorials for Developing BEA Tuxedo ATMI Applications*

What Is Request/Reply Communication?

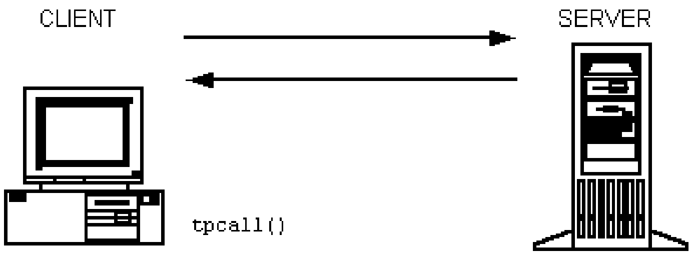
To implement request/reply communication, the BEA Tuxedo system uses IPC message queues. Queues are the key to connectionless communication. Each server is assigned an Inter-Process Communication (IPC) message queue called a request queue and each client is assigned a reply queue. Therefore, rather than establishing and maintaining a connection with a server, a client application can send requests to the server by putting those requests on the server's queue, and then check and retrieve messages from the server by pulling messages from its own reply queue.

The request/reply model is used for both synchronous and asynchronous service requests as described in the following topics.

What Is Synchronous Messaging?

In a synchronous call, a client sends a request to a server, which performs the requested action while the client waits. The server then sends the reply to the client, which receives the reply.

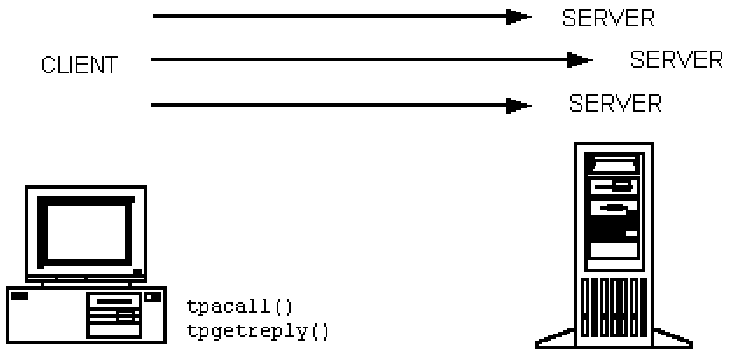
Figure 2-7 Synchronous Request/Reply Communication



What Is Asynchronous Messaging?

In an asynchronous call, the BEA Tuxedo client does not wait for a service request it has submitted to finish before undertaking other tasks. Instead, after issuing a request, the client performs additional tasks (which may include issuing more requests). When a reply to the first request is available, the client retrieves it.

Figure 2-8 Asynchronous Request/Reply Communication



See Also

- "Using the Request/Response Model (Synchronous Calls)" on page 1-7 in *Tutorials for Developing BEA Tuxedo ATMI Applications*

What Is Unsolicited Communication?

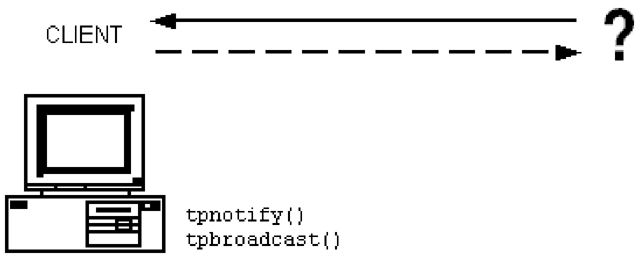
The BEA Tuxedo system offers a powerful communication paradigm called *unsolicited notification*. When unsolicited notification occurs, a BEA Tuxedo client receives a message that it has never requested. This capability makes it possible for application clients to receive notification of application-specific events as they occur, without having to request notification explicitly in real time.

Unsolicited messages can be sent to client processes by name (`tpbroadcast`) or by an identifier received with a previously processed message (`tpnotify`). Messages sent via `tpbroadcast` can originate either in a service or in another client. You can target a narrow or wide audience. You can send a message with or without guaranteed delivery to an individual client through *point-to-point notification* (`tpnotify`), or you can send information to a group of clients (`tpbroadcast`). For example, a server may alert a single client that the account about which the client is inquiring has been closed. Or, a server may send a message to all the clients on a machine to remind the users that the machine will be shut down for maintenance at a specific time.

Any process that wants to be notified about a particular event (such as a machine being shut down for maintenance) can *register* a request, with the system, to be notified automatically. Once registered, a client or server is informed whenever the specified event occurs. This type of automatic communication about an event is called *unsolicited notification*.

Because there is no limit to the number of clients and servers that may generate events and receive unsolicited notification about such events, the task of managing this category of communication can become complex. The BEA Tuxedo system offers a tool for managing unsolicited notification called the *EventBroker*.

Figure 2-9 Unsolicited Notification Messaging



See Also

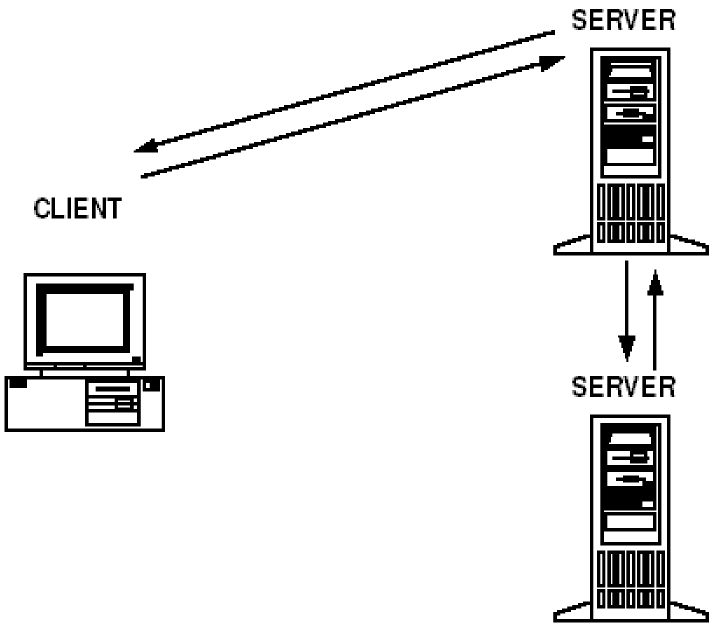
- "Using Unsolicited Notification" on page 1-13 in *Tutorials for Developing BEA Tuxedo ATMI Applications*

What Are Nested and Forwarded Service Requests?

Nested Requests

A powerful feature of the BEA Tuxedo system is that it allows services to act as clients and call other services. Nesting is limited to two levels, which works particularly well in a 3-tier client/server architecture, that is, a system that comprises a presentation logic layer, a business logic layer, and a database layer. In such a system, the presentation layer is used to formulate a request for a particular business function that involves one or more queries to a database. Because nesting is limited to two levels, it does not degrade performance.

Figure 2-10 Nested Service Requests



Benefit of Nested Requests

One benefit of using nested requests is that doing so enables you to keep your code small and reusable, such that each piece performs a limited task. However, if the services in your system are distributed across several servers, nested requests can lead to poor performance. While a nested request is being processed, the original service (that is, the service that issued the nested request) must wait for a response before continuing. Until a response is received, the original service cannot process another request. As a result, messages can get backed up in the request queue for the server on which this service resides.

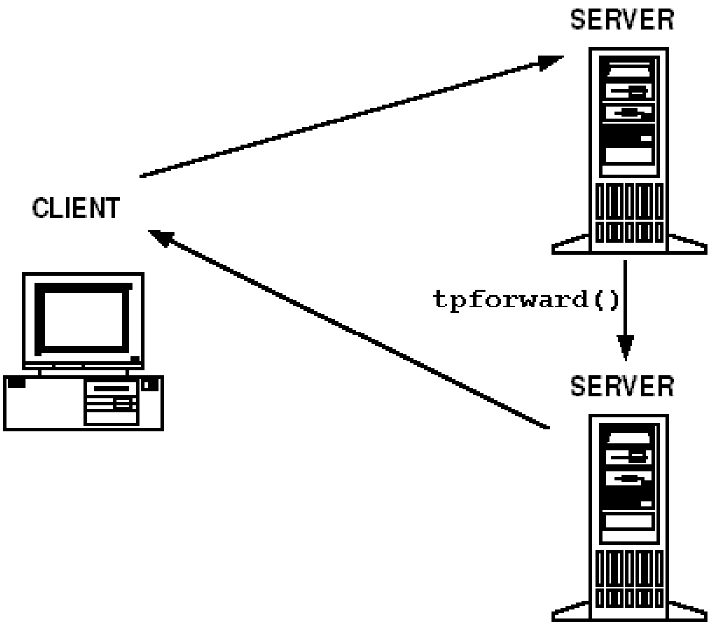
Example of a Nested Service Request

A customer uses a cash machine to transfer money from her savings account to her checking account. A BEA Tuxedo application performs the work necessary to transfer the money. First, on behalf of the customer, the client issues a request for a service called `TRANSFER`, and the request is placed on a queue for a server that provides that service. Next, the TRANSFER service requests two other services, WITHDRAW and DEPOSIT, which are processed by a second server. The WITHDRAW and DEPOSIT services return responses to the TRANSFER service. Finally, TRANSFER sends a response to the client's response queue. When the client retrieves the response from the queue, the system displays a message on the screen of the cash machine, notifying the customer that the transfer is complete.

Forwarded Requests

One alternative to nesting service requests is called request forwarding. Instead of processing a client's request, a service can pass the request to another service. The second service, also, can either process the request or pass it to another service.

Figure 2-11 Forwarded Service Requests



There is no limit to the number of times a request can be forwarded. Because a service that forwards a request does not need to wait for a reply from the service receiving the request, forwarding, unlike nesting requests, does not block servers. Forwarding, however, is not supported by the X/OPEN protocol X/ATMI, which may be a problem in some applications.

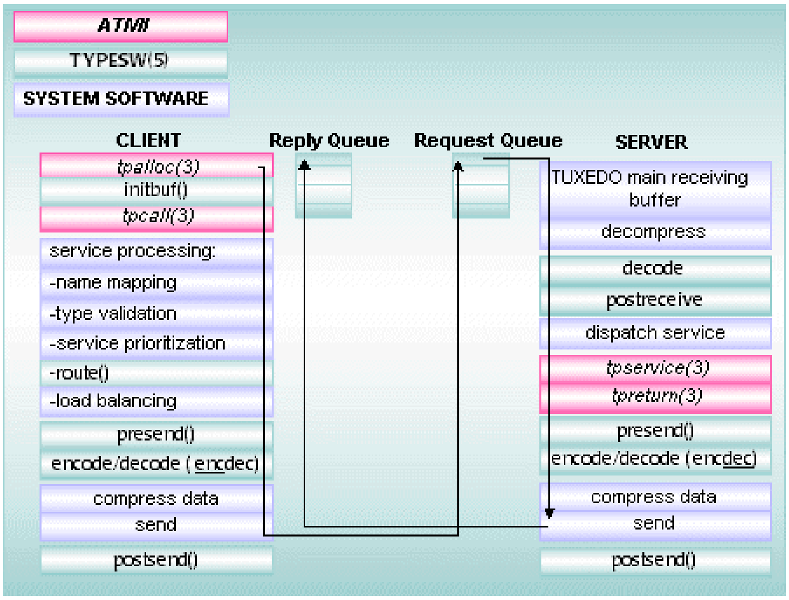
See Also

- "Using Forwarded Calls" on page 1-10 in *Tutorials for Developing BEA Tuxedo ATMI Applications*
- "Using Nested Calls" on page 1-9 in *Tutorials for Developing BEA Tuxedo ATMI Applications*

How BEA Tuxedo ATMI Processes Messages

All communication within the BEA Tuxedo ATMI environment is accomplished by transferring messages. The BEA Tuxedo ATMI environment passes service request messages between clients and servers through operating system Inter-Process Communications (IPC) message queues. System messages and data are passed between operating system-supported, memory-based queues of clients and servers in buffers. In the BEA Tuxedo ATMI environment, messages are packaged in *typed buffers*, buffers that contain both message data and data identifying the types of message data being sent.

Figure 2-12 Processing a Request



A client uses an ATMI function to request a service by name. A *naming* facility is used to check the MIB to determine whether the specified service is currently available. The BEA Tuxedo system uses an automatic routing option to map messages that meet specific criteria (message value) to a specific server. This is called *data-dependent routing*. If messages use data-dependent routing, the system uses the data in the buffer for the routing algorithm. This algorithm provides a method of selecting a group of servers that can process the service request. To avoid burdening a few servers with many requests while leaving other servers that advertise the same services idle, the BEA Tuxedo system maintains a set of metrics in the MIB that help it distribute service requests evenly across all servers. This practice is called *load balancing*.

A local service request may be prepared for a selected server and enqueued on that server's queue with a predefined priority. This practice is called *service prioritization*. Once the service request is on the server, the run-time system retrieves the message in priority order. The message is dispatched to the appropriate service and processed. Then the results are returned to the client queue.

BEA Tuxedo system-provided software offers features that an application can automatically and routinely use during message processing. These features include: data encoding and decoding, data compression and decompression, transactional context setting, and security processing, to name a few. In addition, the BEA Tuxedo system software invokes application business logic by dispatching a service function and passing it to the appropriately preprocessed buffer.

The service routine is executed and returns a reply (also a typed buffer). The run-time system prepares the reply for the client by encoding the message automatically: it packages the data in such a way that it can be transmitted between machines on which different types of byte ordering are used, allowing data to cross network and platform boundaries. The system then sends the message to the client. This process is called *data encoding*. The run-time system on the client retrieves the reply message, decodes it if necessary, and delivers the `FML` buffers (or buffers of another message buffer type) to package the application data. Type validation, encoding, routing, and load balancing are performed as required. Service requests can be performed synchronously or asynchronously.

Remote requests travel through the local bridge to the remote machine, where the remote bridge simply acts as a client and the request is processed as if the client and server were on the same machine. The bridge provides standard data encoding/decoding and uses standard network transports to communicate. Bridges look like ordinary local servers to clients and servers.

What Are the Benefits of Service Request Processing?

- Connectionless processing—this processing, coupled with direct client/server communication, reduces the overhead associated with establishing a connection.
- Reduced network traffic—service requests invoke potentially complex services on remote machines, sending only the minimum data required and receiving minimal results.

See Also

- "What Are the BEA Tuxedo ATMI Messaging Paradigms?" on page 2-8
- "What Are Typed Buffers?" on page 2-24

What Are Typed Buffers?

All ATMI functions send or receive data using typed buffers. The BEA Tuxedo system handles translations and data conversions between dissimilar machines. By using buffers, BEA Tuxedo programs avoid the need to translate data that crosses different platforms with different data representations.

A buffer is a memory area that serves as a logical container for data. When a buffer contains no metadata (that is, no information about itself), then it is an *untyped buffer*. When a buffer includes metadata such as information that can be stored in it (for example, a type and subtype, or string names that characterize a buffer), then it is a *typed buffer*.

Typed buffers can be transmitted over any network, on any operating system, with any protocol supported by the BEA Tuxedo system. They can also be used on platforms with different data representations. As a result, the use of typed buffers facilitates the tasks of translation and data conversion between dissimilar machines.

The BEA Tuxedo system supports five sorts of typed buffers:

- `STRING`
- `VIEW`
- `CARRAY`
- `FML`
- `XML`

You assign buffer types in the `ENVFILE` parameter defined in the `MACHINES` section of the configuration file. Assigning or overriding them in the `ENVFILE` parameter in the `SERVERS` section of the configuration file can make them unavailable to processes that require them.

Definitions of the various types of message buffers are provided in the description of `tm_typesw` in `tuxtypes(5)` in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*. It is to your advantage to change `tm_typesw` so it contains only buffer types specifically needed by a given server.

Characteristics of Buffer Types

When you use ATMI communication functions, your application must first use `tpalloc` to get a buffer from the system, specifying its size, type, and optionally subtype. The BEA Tuxedo system recognizes and processes the buffer type, so that your data is transmitted over any type of network, protocol, and operating system supported by the BEA Tuxedo system. The following table describes the different types of buffers available in a BEA Tuxedo environment.

Table 2-3 Buffer Types Characteristics

Typed Buffer	Description	Purpose
`CARRAY`	Character array type is a collection of characters that is handled opaquely: ■ Characters are not interpreted in any way. ■ No subtypes are specified. ■ Your application must specify the buffer length for `CARRAY` message buffers used as input to ATMI functions.	Data that will not be interpreted by the BEA Tuxedo system and for which data-dependent routing, encoding, or decoding is not required.
`STRING`	A set of non-null characters ending with a null character. The data type is character and the length is determined by counting characters in the buffer until reaching the null character. No subtype is specified.	C programs

Table 2-3 Buffer Types Characteristics (Continued)

Typed Buffer	Description	Purpose
`FML`	Field Manipulation Language (FML) is a data structure that stores tagged values. Values are typed, may be specified more than once, and vary in length. The `FML` buffer is an abstract data type used in operations to create, modify, delete, or access fields. In your program, you access or update a field in the fielded buffer by referencing the identifier, and the `FML` function provides for a run-time translation of the field's location and data type, and performs the operation. One interface to `FML` uses 16 bits (`FML16`) for field identifiers and lengths of fields; the other uses 32 bits (`FML32`). ■ The 16-bit version allows for up to approximately 8000 unique fields, character strings, and arrays of up to 64,000 bytes, and similar lengths for the entire buffer. ■ The 32-bit interface allows for millions of unique fields and buffer lengths of up to two billion bytes. The functionality of the two interfaces is identical. The power of `FML` is in its flexibility. The size of the buffer can vary, depending on the needs of the application for each message. Character fields may also vary in length, so wasted space is avoided. Fielded buffers offer data independence to the application. When writing an application, you do not need to know how or where the data is stored within a fielded buffer. `FML` provides associative field access, so you simply specify a field by name and its value is returned. `FML` also contains conversion functions, so that you can store or retrieve a field in a particular data format, regardless of the underlying storage type. `FML` buffers also support storage of more than one value for a field. The variable length format of fielded buffers allows for multiple field occurrences to be stored and retrieved. Fielded buffers provide a convenient way to transfer a collection of fields, perhaps different with each message, from a client to a server and back, or to store fields in an application queue. We recommend using `FML`, particularly if the interface between clients and servers may change.	■ Communications ■ Creating, modifying, deleting, or accessing fields during operations

Table 2-3 Buffer Types Characteristics (Continued)

Typed Buffer	Description	Purpose
`VIEW`	A `VIEW` is simply a C structure or a COBOL record that has an associated definition of which fields and their types appear in the record in which order. This buffer is used for fixed collections of data elements, or structures or records; its subtype is used to specify the record format name. `VIEW` records are flat data structures. They do not support structures within other structures, nor do they allow arrays of structures or pointers. They support integral data types such as long integer, character, and decimal. `VIEWS` are provided as a way to use C structures and COBOL records with the BEA Tuxedo system. The BEA Tuxedo run-time system understands the record format based on the view description read at run time. When allocating a `VIEW`, your application specifies a buffer type of `VIEW` and a subtype that matches the name of the view. The run-time system can do the following: ■ Determine how much space is needed, based on structure size, so the application need not specify buffer length. ■ Compute how much data to send in a request or response, and handle encoding and decoding when a message is transferred between different machine types.	C structures and COBOL records used with a BEA Tuxedo application
`XML` (Extensible Markup Language)	`XML` buffers enable BEA Tuxedo applications to use XML for exchanging data within and between applications. BEA Tuxedo applications can send and receive simple `XML` buffers, and route those buffers to the appropriate servers. All logic for dealing with the XML documents, including parsing, resides in the application. An XML document consists of: a sequence of characters that encode the text of a document and a logical structure of the document and meta-information related to the structure. The XML parser in the BEA Tuxedo system performs autodetection of character encodings, character code conversion, detection of element content and attribute values, and data type conversion. Data-dependent routing is supported for `XML` buffers.	■ XML documents and datagrams ■ Data interchange between humans and machines, such as from a Web server to a user's browser ■ Data exchange between applications, or from machine to machine

See Also

- "Customizing a Buffer" on page 3-28 in *Programming BEA Tuxedo ATMI Applications Using C*

Using the MIB

The MIB programming interface enables you to manage operations in the BEA Tuxedo system easily. Specifically, it allows you to monitor, configure, and tune your application through your own programs. The MIB can be defined as:

- An implementation-independent management database defined as a set of `FML` attributes.
- A programming interface that enables you to query the BEA Tuxedo system (that is, to obtain information from the system through a `get` operation) or to update the BEA Tuxedo system (that is, to change information in the system through a `set` operation) at any time using a set of ATMI functions. Examples of these functions include `tpalloc`, `tprealloc`, `tpgetrply`, `tpcall`, `tpacall`, `tpenqueue`, and `tpdequeue`.

See Also

- MIB(5) in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*
- "Types of MIB Users" on page 2-29
- "Classes, Attributes, and States in the MIB" on page 2-29

Types of MIB Users

The MIB defines three types of users: system administrators, system operators, and others. The following table describes each type.

Type of User	Characteristics
Application administrator	Person responsible for keeping an application running successfully. The administrator is authorized to use all administrative tools and all MIB administrative capabilities. The administrator configures, manages, and modifies a running production application.
System operator	Monitors and reacts to the daily operation of a production application. The operator monitors statistics about a running application, sometimes reacting to events and alerts by taking actions such as booting servers or shutting down machines. An operator does not reconfigure an application, add servers or machines, or delete machines.
Other	People or processes (such as custom programs) that may need to read the MIB but are not authorized to change the application.

Classes, Attributes, and States in the MIB

Classes are the types of entities, such as servers and machines, that make up a BEA Tuxedo application. *Attributes* are characteristics of the objects in a class: identity, state, configuration parameters, run-time statistics, and so on. There are a number of attributes that are common to MIB operations and replies, and common to individual classes. Every class has a *state* attribute that indicates the state of the object. The state of an object is either *return to the user* or *new, changed state*, if you are invoking an operation on the MIB to change an object’s state.

Independent of classes is a set of common attributes that are defined in the `MIB(5)` reference page. These attributes control the input operations, communicate to the MIB what the user is trying to do, and/or identify to the programmer some of the characteristics of the output buffer that are independent of a particular class.

BEA Tuxedo ATMI Application Processing Services

The BEA Tuxedo ATMI environment offers the following application processing services:

- Data compression
- Data-dependent routing
- Data encoding
- Data encryption
- Data marshalling
- Load balancing
- Message prioritization
- Service and event naming

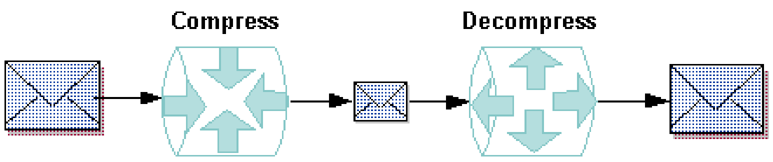
What Is Data Compression?

Data compression is the process of shrinking an application buffer so it can be transmitted more quickly across a network or to a remote domain. By setting a maximum size for an application buffer, you can make sure that compression is triggered automatically for application buffers that match or exceed a specified size. When the buffer arrives at its destination, its data is decompressed, that is, restored to its original size.

Data compression, performed before files are shipped between machines, improves network performance. The process of compression enhances security slightly because it involves scrambling the data.

Note: Data compression also occurs frequently during encryption.

Figure 2-13 Data Compression



What Is Data-dependent Routing?

The BEA Tuxedo system uses an operation called data-dependent routing to enable a client to send requests for the same service to multiple copies of that service. Which copy of the service eventually accepts and processes the request is determined by the data in the request message. Once an administrator has set up data-dependent routing for an application, client requests can be routed automatically to servers based on the data in the requests.

When an application includes multiple copies of the same service, each copy is assigned a unique purpose, just as the first volume of a multivolume encyclopedia contains entries that begin with the letter "A." A list of all copies of the service, along with identifying information about the purpose of each, is kept in a set of routing tables in the BEA Tuxedo bulletin board. When the system receives a client request, it finds an identifying string in the request message and searches the routing tables in the bulletin board for the same string. On the basis of this match, the system identifies the appropriate server to which it can forward the client request.

Note: The bulletin board routing tables can be modified as necessary.

Uses of Data-dependent Routing

Data-dependent routing is useful when clients issue service requests to:

- Horizontally partitioned databases
- Rule-based servers
- Distributed Application

A horizontally partitioned database is an information repository that has been divided into segments, each of which is used to store a different category of information. This arrangement is similar to a library in which each shelf of a bookcase holds books for a different category (for example, biography, fiction, and so on).

A rule-based server is a server that determines whether service requests meet certain, application-specific criteria before forwarding them to service routines. Rule-based servers are useful when you want to handle requests that are almost identical by taking slightly different actions for business reasons.

A distributed application consists of one or more local or remote clients that communicate with one or more servers on several machines linked through a network. A client (or server acting as a client) issues a request for a particular service. The *address* of the request is determined by data (carried in the same buffer that conveys the request), identifying the server that can fulfill the request. More than one server may be able to do so. The BEA Tuxedo system selects a server to receive the request by matching the data to the routing criteria provided in the bulletin board.

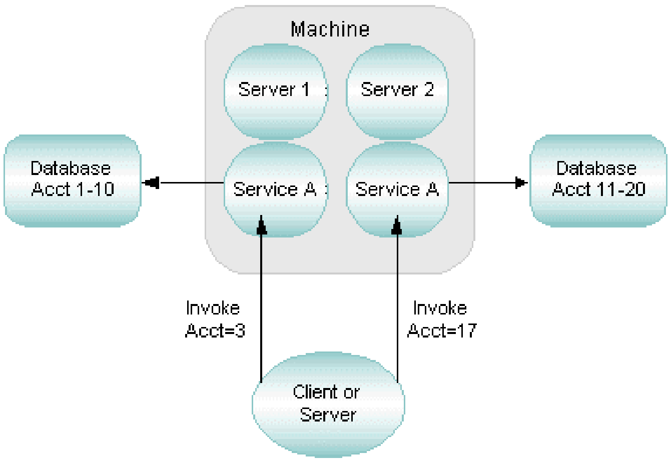
Example of Data-dependent Routing with a Horizontally Partitioned Database

Suppose two clients in a banking application issue requests for the current balance in two accounts: Account 3 and Account 17. If data-dependent routing is being used in the application, then the BEA Tuxedo system performs the following actions:

1. Gets the account numbers for the two service requests (3 and 17).
2. Checks the routing tables on the BEA Tuxexdo bulletin board that show which servers handle which range of data. (In this example, server 1 handles all requests for Accounts 1 through 10; server 2 handles all requests for Accounts 11 through 20.)
3. Sends each request to the appropriate server. Specifically, the system forwards the request about Account 3 to server 1, and the request about Account 17 to server 2.

The following figure illustrates this process.

Figure 2-14 Data-dependent Routing with a Horizontally Partitioned Database



Example of Data-dependent Routing with Rule-based Servers

A banking application includes the following rules:

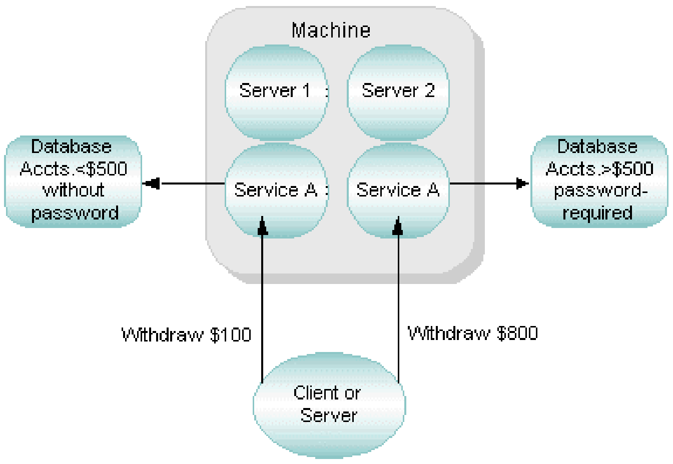
- Customers can withdraw up to $500 without entering a special password.
- Customers must enter a special password to withdraw more than $500.

Two clients issue withdrawal requests: one for $100 and one for $800. If data-dependent routing is enabled to support the withdrawal rules, then the BEA Tuxedo system performs the following actions:

1. Gets the amount specified for withdrawal in the two service requests ($100 and $800).
2. Checks the routing tables on the BEA Tuxedo bulletin board that show which servers handle request for the amount being requested. (In this example, server 1 handles all requests to withdraw amounts up to $500; server 2 handles all requests to withdraw amount over $500.)
3. Sends each request to the appropriate server. Specifically, the system forwards the request for $100 to server 1 and the request for $800 to server 2.

The following figure illustrates this process.

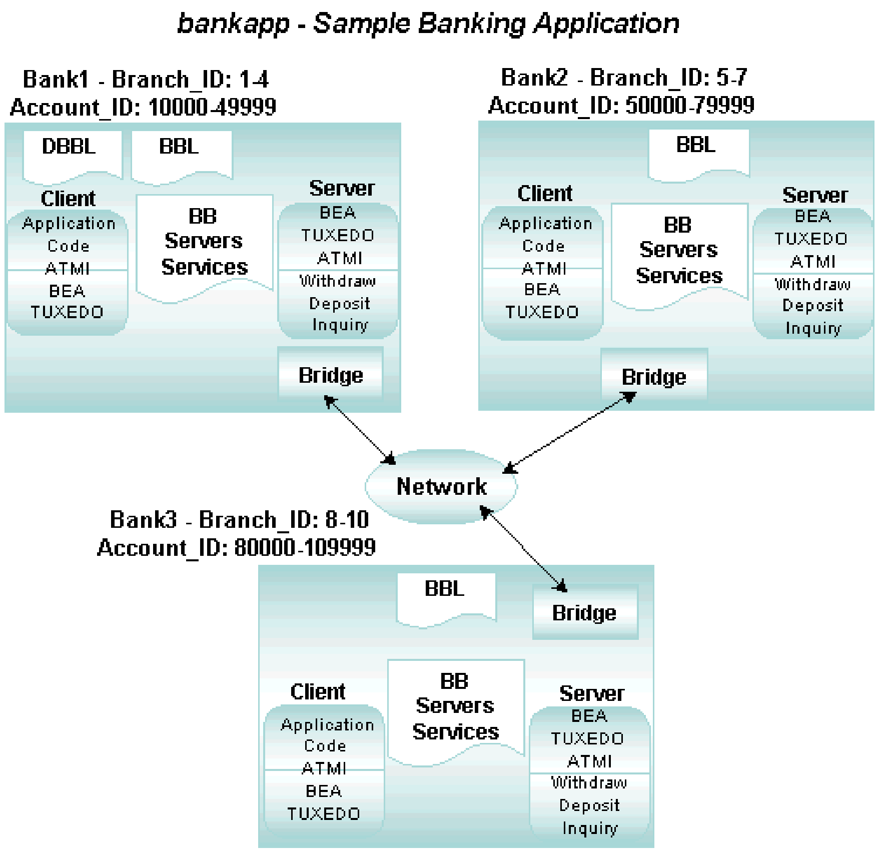
Figure 2-15   Data-dependent Routing with Rule-based Servers



Example of Data-dependent Routing with Distributed Application

The following diagram shows how client requests are routed to servers. In this example, a banking application called `bankapp` uses data-dependent routing. `bankapp` has three server groups (`BANK1`, `BANK2`, and `BANK3`) and two routing criteria (`Account ID` and `Branch ID`). The services `WITHDRAW`, `DEPOSIT`, and `INQUIRY` are routed using the `Account_ID` field; the services `OPEN` and `CLOSE` are routed using the `Branch_ID` field.

Figure 2-16 Sample Banking Application Using Routing Criteria



In the preceding diagram, requests are routed as indicated in the following table.

Withdrawals, Deposits, Inquiries, and Openings or Closings of the Following Accounts . . .	Are Routed to . . .
Numbers 10000–49999 for branches 1–4	Bank1
Numbers 50000–79999 for branches 5–7	Bank2
Numbers 80000–109999 for branches 8–10	Bank3

What Are Encoding and Decoding of Data?

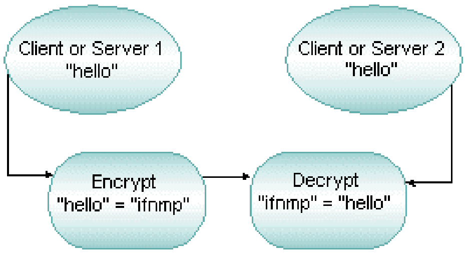
Encoding and *decoding* enable messages with different data representations (for example, byte ordering or character sets) to be transferred between machines. The BEA Tuxedo system accomplishes this by encoding and decoding data to a machine-independent representation for transmission. It employs, by default, the XDR algorithm, which can be customized by replacing the BEA Tuxedo system functions with user-written functions. Encoding and decoding are used only between machines and only when a remote machine uses a data representation other than the one used on the local machine. Encoding and decoding allow machines with different data architectures to operate within a heterogeneous BEA Tuxedo system. Programmers can manage data in representations natural to their own environments.

The BEA Tuxedo system uses buffer types to determine the type of fields contained in a message, and to perform the mapping required for coding tasks. This mapping is not performed by unstructured buffer types such as `X_OCTET` and `CARRAY`. Thus, developers using `X_OCTET` and `CARRAY` buffers are free to deploy in mixed-machine environments.

What Is Data Encryption?

Encryption is the act of converting a message into a coded format that is unintelligible to users. When an encrypted message arrives at its destination, it is decrypted, that is, converted back to its original format.

Figure 2-17 Data Encryption



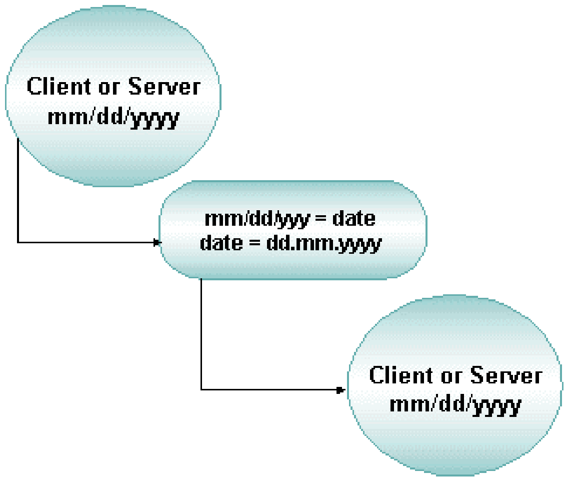
Encryption does not increase the number of bits in the data, but it adds processing time to the task of sending a message. Because data is compressed during encryption, however, lost processing time may be bought back, since less data is being sent across the network. When data is compressed, there is also a moderate boost to security, because the data is somewhat scrambled during compression.

What Is Data Marshalling?

Data marshalling is a method of handling information through the language-based TxRPC (X/Open-TxRPC) offered by the BEA Tuxedo system. TxRPC is a set of protocols for remote procedure calls that supports global transactions. Though a TxRPC call looks like a local procedure call, when a C function is called, the arguments passed to the function are packaged so they can be sent to a server that performs the work of the called function. This argument packaging is called *marshalling*. A function’s arguments are *marshalled* or packaged in a way that allows them to cross network and platform boundaries, and then *unmarshalled* at their destination before being passed to the invoked remote procedure, ready for use.

This process is transparent to the client (the calling program) and the server (the remote procedure). The marshalling and unmarshalling routines are generated automatically by the BEA Tuxedo Interface Definition Language (IDL) compiler. An IDL compiler takes a description of a set of RPCs and generates routines, called stubs, for the client and server programs. These stubs contain marshalling and unmarshalling logic, as well as the communication logic that allows a client and server to exchange marshalled data.

Figure 2-18 Data Marshalling



What Is Load Balancing?

Load balancing is a technique used by the BEA Tuxedo system for distributing service requests evenly among servers that offer the same service. This avoids overburdening some servers while leaving others idle or infrequently used. Before sending a request to a service routine, the system identifies all servers capable of handling the request and selects the one most appropriate for maintaining a balanced load across all the servers in the configuration.

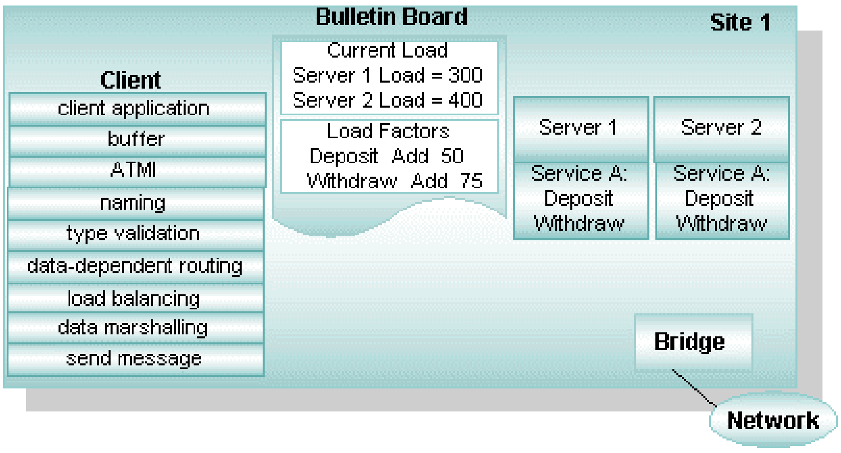
Assigning a Load Factor

Load refers to a number assigned to a service request based on the amount of time required to execute that service. Loads are assigned to services so that the BEA Tuxedo system can understand the relationship between requests. To keep track of the amount of work, or total load, being performed by each server in a configuration, the administrator assigns a *load factor* to every service and service request. A load factor is a number indicating the amount of time needed to execute a service or a request. On the basis of these numbers, statistics are generated for each server and maintained on the bulletin board on each machine. Each bulletin board keeps track of the cumulative load associated with each server, so that when all servers are busy, the BEA Tuxedo system can select the one with the lightest load.

You can control whether a load-balancing algorithm is used on the system as a whole. Such as algorithm should be used only when necessary, that is, only when a service is offered by servers that use more than one queue. Services offered by only one server, or by multiple servers in an MSSQ (Multiple Server, Single Queue) do not need load balancing. The `LDBAL` parameter for these services should be set to `N`. In other cases, you may want to set `LDBAL` to `Y`.

To determine how to assign load factors (in the `SERVICES` section of `UBBCONFIG`), run an application for a long period of time and note the average time it takes to perform each service. Assign a `LOAD` value of `50` (`LOAD=50`) to any service that takes roughly the average amount of time. Any service taking longer than average should have a `LOAD>50`; any service taking less than the average should have a `LOAD<50`.

Figure 2-19 Load Balancing

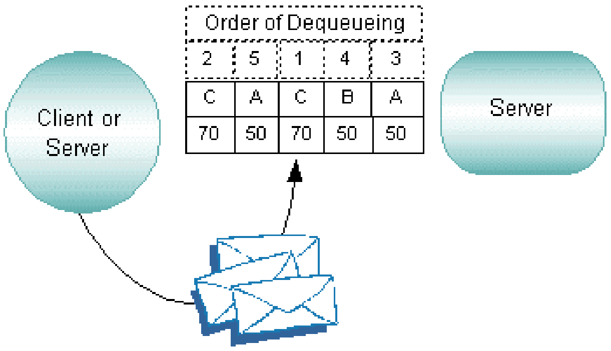


What Is Message Prioritization?

Priorities determine the order in which service requests are dequeued by a server. Priority is assigned by a client to individual services and can range from 1 to100, where 100 represents the highest priority.

All services are assigned a starting priority of 50. A server's starting priority can be changed during application configuration. Once you have defined your set of services, you can assign the appropriate priorities to them. For example, your business may require that some services have a relatively high priority of 70, which means those services are dequeued before those with the lower priority of 50. In the following illustration, a server offers services A (with a priority of 50), B (with a priority of 50), and C (with a priority of 70).

Figure 2-20 Prioritization of Messages



A request for service C is always dequeued before a request for A or B due to the higher priority of C. Requests for A and B have equal priority. This feature is useful in applications in which not all requests are equally urgent or important.

A "starvation prevention" mechanism prevents low-priority messages from waiting endlessly on the queue. Every tenth message is dequeued in FIFO (first in first out) order regardless of priority; the first through the ninth messages are dequeued in order of priority.

What Is Meant by Naming?

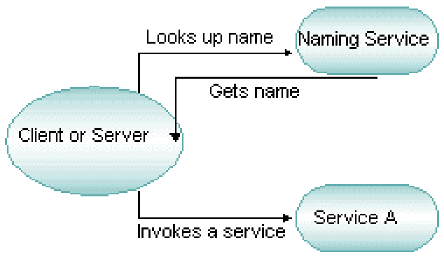
The BEA Tuxedo system uses three naming devices: service names, message queue names, and event names. Names can be any words or alphanumeric strings, as long as they do not begin with a period ("."). Because administrative servers use the BEA Tuxedo system infrastructure, system and application resources must be clearly distinguished.

Naming Services

When services are named, an application component can locate another component through a name. Names can be simple words (such as "deposit") or alphanumeric strings (such as "deposit2"). Names should be selected on the basis of the scope of the application and a map that contains the global picture of the relationships among application components. These maps or services are like the pages in a telephone book for application components.

When a BEA Tuxedo system server is activated, the bulletin board (the dynamic part of the MIB) advertises the names of its services. Service names are associated with a server's physical address so that requests can be routed to that server. Names that programmers use in their applications are completely location transparent. When a client program asks for a service by name, the BEA Tuxedo system consults its name registry in the bulletin board. The name registry provides the information necessary to convert the string name (for example, `TICKET`) to a machine name and the physical address of a server that advertises that service. The BEA Tuxedo system then sends the request to the appropriate server.

Figure 2-21 Locating a Service by Name



Advertising Services

The BEA Tuxedo system uses two administrative servers to coordinate the distribution of information on the bulletin board to all active machines in the application:

- DBBL—the Distinguished Bulletin Board Liaison server propagates global changes to the MIB and maintains the static part of the MIB. The DBBL coordinates the state of different machines involved in an application. Only one DBBL exists for an entire application. It can be migrated to other machines for fault resiliency.
- BBL—the Bulletin Board Liaison server maintains the bulletin board. A BBL resides on every active machine in an application. The BBL coordinates changes to the local MIB and verifies the integrity of application programs active on its machine.

Naming Events

The BEA Tuxedo system offers a publish-and-subscribe mechanism: clients and servers can dynamically register or unregister a standing request to receive alerts (or messages) when a particular event occurs. Other clients and servers post user-defined or system events as they occur in the application. When a client or server no longer needs to be notified about a particular event, the relevant subscription can be cancelled.

See Also

- "How the EventBroker Works" on page 2-10

BEA Tuxedo ATMI Administrative Services

A set of system servers provides the following administrative services needed by the BEA Tuxedo ATMI environment:

- Application queue management
- Centralized application configuration
- Distributed application management
- Dynamic application reconfiguration
- Event management
- Security management
- Startup and shutdown of an application
- Transaction management
- Workstation management

Note: For information on administrative services, see the topic, "Three Ways of Viewing the BEA Tuxedo ATMI Infrastructure" on page 3-1

CHAPTER

3 Three Ways of Viewing the BEA Tuxedo ATMI Infrastructure

This topic includes the following sections:

- Basic BEA Tuxedo ATMI Infrastructure
- Management View: Using Administrative Tools
- BEA Tuxedo ATMI Administrative Services
- Development View: What You Can Do Using the ATMI
- Run-time System View: Using Tools in Different Configurations

Basic BEA Tuxedo ATMI Infrastructure

The BEA Tuxedo ATMI environment provides an infrastructure for the efficient routing, dispatching, and management of application service requests, event postings and notification, and application queues. This infrastructure can be explored from three perspectives:

- Administrative or management perspective—encompasses a variety of tools available to manage your application.

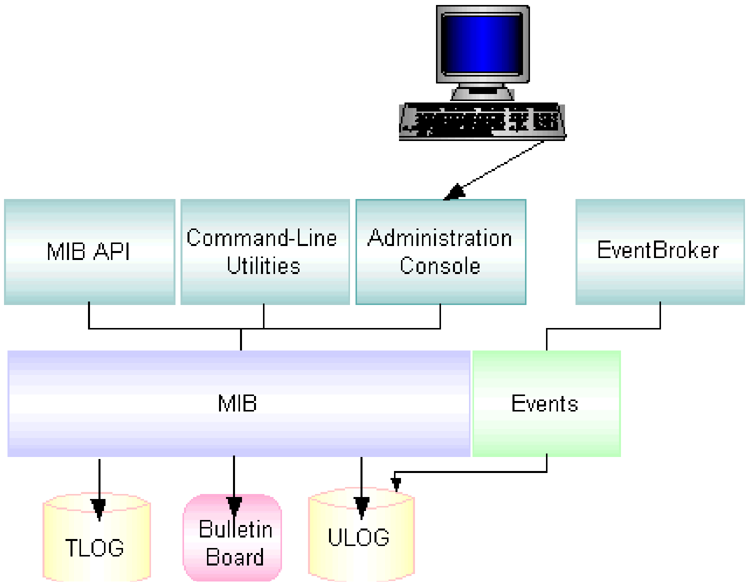
- Development (using the ATMI) perspective—encompasses those tasks you can perform using the ATMI. Clients request services through the ATMI. Server programs group several services, which are invoked according to the rules defined by the ATMI. Application designers construct client and server programs by linking the BEA Tuxedo run-time system with their application code.
- BEA Tuxedo run-time system view—encompasses single, distributed, and multiple domain configurations.

Management View: Using Administrative Tools

The BEA Tuxedo MIB contains all the information necessary for the operation of an application. The MIB is designed to be programmable, so that you can write custom administrative programs. Administrative tools are constructed around the MIB and provide different types of interfaces to it. These tools include the following:

- BEA Administration Console—a Web-based tool used to monitor an application, and to dynamically configure it.
- *BEA Tuxedo administrative servers*—servers that automate most of the management tasks for a distributed application, such as naming services and events, starting up and shutting down an application, dynamically reconfiguring an application, and so on.
- BEA Tuxedo MIB application programming interface—a set of functions for accessing and modifying information in the MIB.
- Command-line utilities—a set of commands used to activate, deactivate, configure, and manage an application (that is, `tmboot`, `tmshutdown`, `tmconfig`, and `tmadmin`, respectively). (See the *BEA Tuxedo Command Reference*.)
- EventBroker—a mechanism that informs administrators of faults or exceptional happenings.

Figure 3-1 Tools to Administer Your Application



Available BEA Tuxedo MIBs

The Management Information Base comprises a *core MIB*, which is common to all applications, and several component MIBs, which are optional. The core MIB, called TM_MIB, defines the parts of an application that are required in every BEA Tuxedo application. It is also used to administer those parts of an application. TM_MIB defines a BEA Tuxedo system application as a set of classes (for example, servers, groups, machines, domains), each of which is made up of objects that are characterized by various attributes (for example, identity and state).

Each of the component MIBs describes a subsystem of the BEA Tuxedo system. The following components are currently available:

- ACL_MIB—used to administer Access Control Lists
- APPQ_MIB—used to administer application stable-storage queues
- DM_MIB—used to administer Tuxedo domains
- EVENT_MIB—used to control event notification and the subscription request database

- WS_MIB—used to manage Workstation groups and processes associated with them

Using the BEA Administration Console

Based on Java and Web technology, the BEA Administration Console lets you operate your BEA Tuxedo domains from virtually anywhere—even from home, given security authorization. The BEA Administration Console is a Java-based applet that you can download into your Internet browser and use to remotely manage BEA Tuxedo system applications.

The BEA Administration Console simplifies many of the system administration tasks required for managing multiple-tier systems. It lets you monitor system events, manage system resources, create and configure administration objects, and view system statistics.

Browser Requirements

Each release of the BEA Tuxedo system supports the currently available browsers. Consult the following BEA Web site for information about browsers currently supported by the BEA Administration Console.

See Also

- “Benefits of Using the BEA Administration Console” on page 3-5
- “Exploring the Main Menu of the BEA Administration Console” on page 3-6
- *BEA Administration Console Online Help*
- “Ways to Monitor Your Application” on page 2-2 in *Administering a BEA Tuxedo Application at Run Time*

Benefits of Using the BEA Administration Console

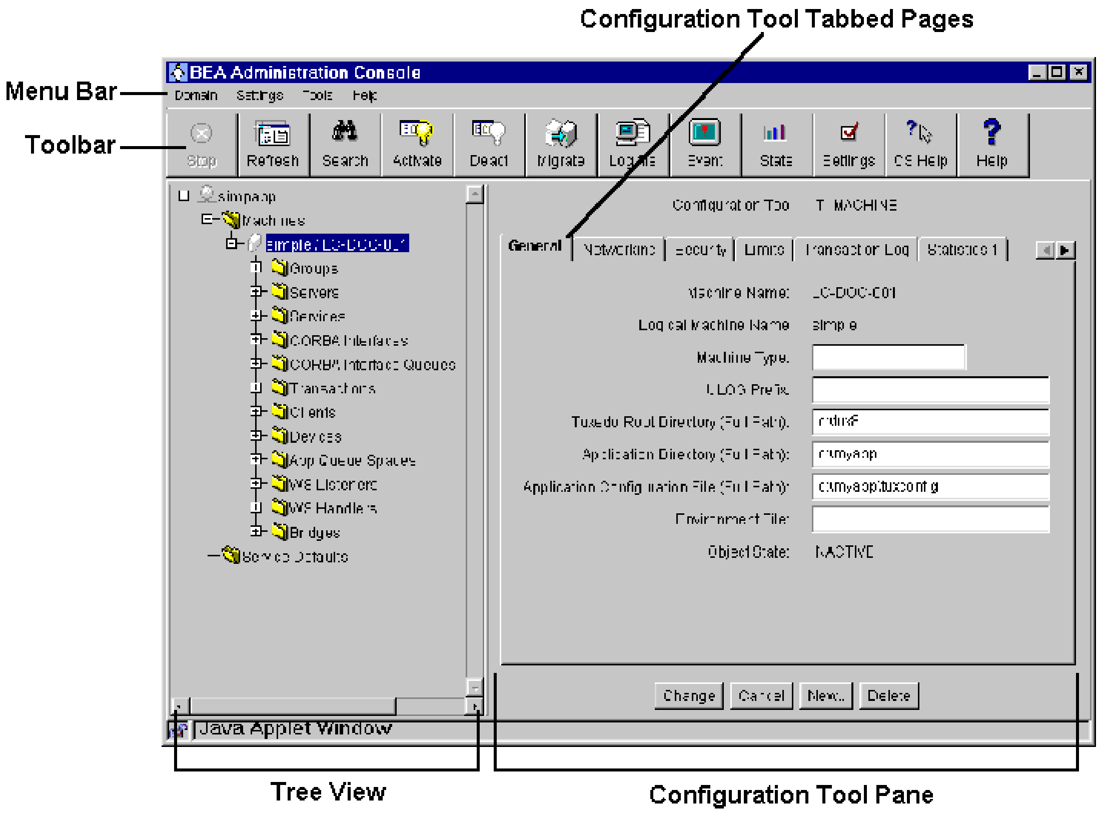
- Authentication—the BEA Administration Console forces users to identify themselves. It prompts the administrator for a username and password. This information is communicated in an encrypted fashion between the browser and the server, where the user’s identity is then verified. (Much of the server setup is done during installation, when server components of the BEA Administration Console are installed and made available to the Web server.)
- Context-sensitive help—context-sensitive help is available for all BEA Administration Console windows and tools. You can request information about any field or area of a window simply by dragging a question mark icon to that field or any area and clicking.
- Encryption—the data transferred between the server side and the browser is compressed (56-bit or 128-bit encryption) so that no one can read it. This makes the system resistant to anyone trying to inject false administrative protocol messages into the stream.
- Firewall readiness—the port on which the BEA Administration Console server listens and interacts with the browser is well defined and configurable; you can configure it to match ports that you want to allow through your firewall. This capability enables you to do Console-based administration through your firewall, if necessary.
- Icons—the icons used in the BEA Administration Console connote state (for example, *not active*) or represent particular objects in the application, for example, machines or servers.
- Java-capable browser—the Java browser supports the Java virtual machine that runs the applets and enables communication.
- No client-side installation—no installation is required on your machine. Point your browser to the URL for a machine in your domain on which the Console server components reside. Then initiate a download of Java applets. The applets implement the BEA Administration Console and establish communication with the server.
- Universal secure access—from any Java-capable browser, you can access the system from anywhere in the world with confidence that security mechanisms are already in place.

Exploring the Main Menu of the BEA Administration Console

When you first bring up the Web and invoke the BEA Administration Console, the main window is displayed. The main window is divided into four major areas:

- Menu bar—menus that provide access to all actions.
- *toolbar*—buttons that provide shortcuts to frequently used action or administrative tools.
- *tree view*—a hierarchical representation of the administrative class objects (such as servers and clients) in a BEA Tuxedo domain.
- Configuration tool—a set of tabbed pages on which you can display, define, and modify the attributes of objects, such as the name of a machine.

Figure 3-2 Main Menu of the Administration Console



Configuration Tool:	T_MACHINE
Machine Name:	LC-DOC-C01
Logical Machine Name:	simple
Machine Type:	
ULOG Prefix:	
Tuxedo Root Directory (Full Path):	c:\tux8
Application Directory (Full Path):	c:\myapp
Application Configuration File (Full Path):	c:\myapp\tuxconfig
Environment File:	
Object State:	INACTIVE

Note: The toolbar buttons and some menu items are not fully displayed unless you are connected to a domain.

What Is the Tree?

The Tree View pane appears in the left column of the main GUI window. The tree is a hierarchical representation of the administrative objects in a single BEA Tuxedo system domain. The GUI graphically depicts the relationship between each object and the others by showing its nesting level and parent objects. You can choose to view a complete tree (comprising all configurable objects of all types in the domain) or a subset of objects.

After you have set up and activated a domain, the Tree is populated with labeled icons, representing the administrative class objects in your domain.

What Is an Administrative Object?

The BEA Administration Console Tree View contains multiple roots, one root for each administrative object. The first root consists of the application domain. The next root displays the object classes defined in the BEA Tuxedo TMIB. Each set of object classes is a part of an application domain. The third level represents an instance of an object belonging to an object class.

For example, suppose your domain includes two machines (both at SITE1) named *`romeo`* and *`juliet`*. Since both machines are *objects*, they are listed in the Tree below the name of the *object class* to which they belong: `Machines`. Therefore, they will be listed as follows:

```
Machines

      SITE1/romeo

      SITE1/juliet
```

The name of each object in the Tree View is preceded by an icon. Each machine, for example, is represented by a computer; each client, by a human figure.

Using the Configuration Tool

The Configuration Tool is a utility that lets you set or change the attributes for a selected class of BEA Tuxedo system objects. When you select an object in the tree, the Configuration Tool Pane for that object is displayed on the right side of the main window.

The tabbed pages in the Configuration Tool area are electronic forms that display and solicit information about the attributes of an administrative object. A set of tabbed pages is provided for each administrative class of objects (such as machines and servers). The number of attributes associated with a class varies greatly, depending on the class. Therefore, anywhere from one to eight folders may be displayed when you invoke the Configuration Tool by selecting an object in the tree.

When the Configuration Tool area is populated, another row of buttons is displayed below the tabbed pages. These four buttons allow you to control the configuration work done in the pages.

Using the Toolbar

The toolbar is a row of 12 buttons that allow you to invoke tools for frequently performed administrative operations. They are labeled with both icons and names. The following table describes each button.

Button	**Description**
Stop	Interrupts the current operation and returns control to the administrator (who can then request a new operation).
Refresh	Updates the tree view and configuration tool pane with the most up-to-date data.
Search	Searches for a particular administrative object class or object in the expanded Tree.
Activate	Activates all or part of a BEA Tuxedo domain.
Deactivate	Deactivates all or part of a BEA Tuxedo domain.
Migrate	Migrates a server group or machine to another location, or swaps the master and backup machines.
Log file	Displays the `ULOG` file from a particular machine in the active domain.
Event	Displays a window for monitoring system-generated events.
Stats	Displays the tabbed pages that allow you to view a graphical presentation of BEA Tuxedo domain activity.

Button	Description
Settings	Provides the option to set the following default settings for the Administration Console session: ■ The location of your BEA Tuxedo online documentation ■ The method for sorting your data (by state or name) ■ Your default work mode (view-only or edit mode)
CS Help	Invokes context-sensitive help. Click a field or a specific area of the console to get information about the selected item.
Help	Opens the BEA Administration Console Online Help in a separate Web browser.

Managing Operations Using the MIB

The AdminAPI is an application programming interface (API) for directly accessing and manipulating system settings in the BEA Tuxedo Management Information Bases (MIBs). You can use the AdminAPI to automate administrative tasks, such as monitoring log files and dynamically reconfiguring an application, thus eliminating the need for human intervention. This advantage can be crucially important in mission-critical, real-time applications. Using the MIB programming interface, you can manage operations in the BEA Tuxedo system easily. Specifically, you can monitor, configure, and tune your application through your own programs. The MIB can be defined as:

- An implementation-independent management database defined as a set of `FML` attributes.
- A programming interface that enables you to query the BEA Tuxedo system (that is, to obtain information from the system through a `get` operation) or to update the BEA Tuxedo system (that is, to change information in the system through a `set` operation) at any time using a set of ATMI functions. Examples of these functions include `tpalloc`, `tprealloc`, `tpgetrply`, `tpcall`, `tpacall`, `tpenqueue`, and `tpdequeue`.

See Also

- `MIB(5)` in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*
- "Types of MIB Users" on page 3-11
- "Classes, Attributes, and States in the MIB" on page 3-12

Types of MIB Users

The MIB defines three types of users: system administrators, system operators, and others. The following table describes each type.

Type of User	Characteristics
Application administrator	Person responsible for keeping an application running successfully. The administrator is authorized to use all administrative tools and all MIB administrative capabilities. The administrator configures, manages, and modifies a running production application.
System operator	Person responsible for monitoring and reacting to the daily operation of a production application. An operator monitors statistics about a running application, sometimes reacting to events and alerts by taking actions such as booting servers or shutting down machines. An operator does not reconfigure an application, add servers or machines, or delete machines.
Other	People or processes (such as custom programs) that may need to read the MIB but are not authorized to change the application.

Classes, Attributes, and States in the MIB

Classes are the types of entities such as servers and machines that make up a BEA Tuxedo application. *Attributes* are characteristics of the objects in a class: identity, state, configuration parameters, run-time statistics, and so on. There are a number of attributes that are common to MIB operations and replies and common to individual classes. Every class has a *state* attribute that indicates the state of the object. The state of an object is either *return to the user* or *new, changed state*, if you are invoking an operation on the MIB to change an object's state.

Independent of classes is a set of common attributes that are defined in the `MIB(5)` reference page. These attributes control the input operations, communicate to the MIB what the user is trying to do, and/or identify to the programmer some of the characteristics of the output buffer that are independent of a particular class.

Using Command-line Utilities

The BEA Tuxedo system provides a set of commands for managing different parts of the system. The commands enable you to access common administrative utilities. These utilities can be used for the following tasks:

- Configuring your application using command-line utilities
- Operating your application using command-line utilities
- Monitoring your application using command-line utilities

Configuring Your Application Using Command-line Utilities

You can configure your application by using command-line utilities such as the `vi` text editor. Specifically, you can use command-line utilities to write the configuration file, `UBBCONFIG`, and translate the file from a text format (`UBBCONFIG`) to a binary format (`TUXCONFIG`), by running the `tmloadcf` command. Then you are ready to boot your application.

You can dynamically administer your configuration by adding servers or machines, deleting machines, and so forth. Updating `TUXCONFIG` (the binary file version), however, does not update the `UBBCONFIG` (the text file version). To synchronize both files, you need to back them up. To do this, you translate the binary file back to text by running the `tmunloadcf` command.

Note: The `UBBCONFIG` is generated and stored by the application administrator in the application directory (`APPDIR`).

Following is a list of common command-line utilities that you can use to configure your application.

- `tmconfig`—a command that enables you to update some configuration file parameters, or `MIB` attributes, and add records to some `TUXCONFIG` sections while the BEA Tuxedo system application is running.
- `tmloadcf`—a command that allows you to load the binary `TUXCONFIG` configuration file.
- `tmunloadcf`—a command that allows you to translate the binary configuration file back to a text version, so that `UBBCONFIG` and `TUXCONFIG` can be synchronized.
- `tpacladd`, `tpaclcvt`, `tpacldel`, and `tpaclmod`—a set of commands that allow you to create or manage access control lists for applications. These commands enable the use of security-related authorization features.
- `tpgrpadd`, `tpgrpdel`, `tpgrpmod`—a set of commands that allow you to create and manage user groups by using access control lists to authorize access to services, queues, and events.

- `tpusradd`, `tpusrdel`, `tpusrmod`—a set of commands that allow you to create and manage a user database for authorization purposes.

See Also

- `UBBCONFIG(5)` in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*
- “Creating the Configuration File” on page 3-22
- “Making Permanent Configuration Changes” on page 3-24

Operating Your Application Using Command-line Utilities

Once you have configured your application successfully, you can use the following command-line utilities to operate your application.

- `tmadmin`—a command that allows you to configure, monitor, and tune a distributed application.
- `tmboot`—a command that allows you to centrally start up your application servers for a distributed application.
- `tmshutdown`—a command that allows you to centrally shut down an application program across a distributed application.

Managing System Events Using EventBroker

The BEA Tuxedo EventBroker performs the following tasks:

- Monitors events and notifies subscribers when events are posted via `tppost`(3c).

- Keeps an administrator informed of changes in an application by tracking events.
- Enhances event monitoring by providing a system-wide summary of events.
- Provides a mechanism through which an event can trigger a variety of notification activities.

The EventBroker recognizes over 100 meaningful state transitions in a MIB object as system events. The postings for system events include the current MIB representation of the object on which the event has occurred, and some event-specific fields that identify the event that occurred. For example, if a machine is partitioned, an event is posted with the following information:

- The name of a *machine class object* (`T_MACHINE`), with all the attributes of that machine
- Some event attributes identifying the event as *machine partitioned*

You can use the EventBroker simply by subscribing to system events. Then, instead of having to query for MIB records, you can be informed automatically when events occur in the MIB by receiving `FML` data buffers representing MIB objects.

See Also

- "What Is an Event?" on page 3-15
- "Subscribing to an Event" on page 3-16
- "Types of Events" on page 3-17
- "Using Event-based Communication" on page 1-14 in *Tutorials for Developing BEA Tuxedo ATMI Applications*

What Is an Event?

An event is a state change or other occurrence in a running application that may warrant special attention from an operator, an administrator, or the software. In the EventBroker, events are assigned one of three severity levels:

- Error—for example, a server has died or a network connection has been dropped
- Informational—for example, a state change has occurred as a result of a process or the detection of a configuration change
- Warning—for example, a client has not been allowed to join the application after failing authentication

Subscribing to an Event

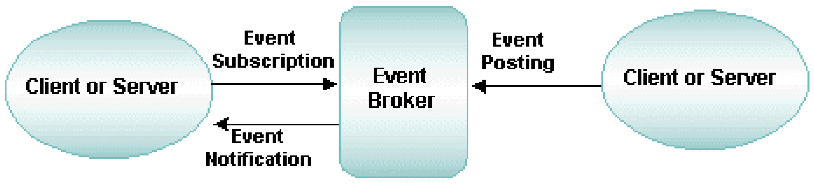
As the administrator for your BEA Tuxedo application, you can enter subscription requests on behalf of a client or server process through calls to the `EVENT_MIB(5)`. You use `tpsubscribe` to subscribe to an event using the EventBroker. You may want to subscribe to Events A, B, and C and request to be informed when they occur.

Each subscription specifies one of the following notification methods:

- Client notification—the EventBroker keeps track of the client's interest in these events and a client is notified in the form of unsolicited notification. Some events are anonymously posted. A client can join an application, independent of whether anyone else has subscribed, and post events to the EventBroker. The EventBroker matches these events against its database of subscriptions and sends an unsolicited notification to the appropriate clients.
- Service calls—if the subscriber wants event notifications to go to service calls, then the `ctl` parameter must point to a valid `TPEVCTL` structure.
- Message enqueuing to stable-storage queues—for subscriptions to stable-storage queues, the queue space, queue name, and correlation identifier are used, in addition to `eventexpr` and `filter`, when determining matches. The correlation identifier can be used to differentiate among several subscriptions for the same event expression and filter rule, destined for the same queue.
- Placing messages on the `ULOG`—using the `T_EVENT_USERLOG` class of `EVENT_MIB`, subscribers can write system `USERLOG` messages. When events are detected and matched, they are written to the `USERLOG`.
- Command-line utilities—using the `T_EVENT_COMMAND` class of `EVENT_MIB`, the EventBroker tracks and matches events. When a match is found, it is passed to the command used when subscribing to the event.

Note: Notification methods are determined by the subscriber process type and the arguments passed to `tpsubscribe`.

Figure 3-3 Subscribing to an Event



See Also

- `EVENT_MIB(5)` in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*
- `tpsubscribe(3c)` in the *BEA Tuxedo ATMI C Function Reference*

Types of Events

The BEA Tuxedo system supports two event types:

- System Events—provide details about BEA Tuxedo system events, such as servers dying, and network failures. When an event is posted by clients or servers, the EventBroker matches the posted event's name to a list of subscribers for that event and takes appropriate action, as determined by each subscription.
- User Events or Application-specific Events—allow application programs to post events when certain criteria are met. An example is a banking application that posts an event for withdrawals over a certain limit.

Differences Between System and Application-specific Events

The following table identifies the differences between system and application-specific events.

Table 3-1 Differences Between System and Application-specific Events

Area	Differences
Events	System events are defined in advance by the BEA Tuxedo system code. For an application, designers decide which application events should be monitored. Application programs are written to: (a) detect when an event of interest has occurred, and (b) post the event to the EventBroker through `tppost`.
Event List	A list of the application event subscriptions is made available to interested users just as the BEA Tuxedo system provides a list of system events available to users with `EVENTS(5)`. System event names begin with a dot (.); application-specific event names may not begin with a dot (.).
Subscriptions	Subscribing to an event in an application-specific event broker is similar to subscribing to the BEA Tuxedo System EventBroker. You subscribe by making calls to `tpsubscribe` using the published list of events for the application. `EVENTS(5)` lists the notification message generated by an event as well as the event name (used as an argument when `tppost` is called). Subscribers can use the wildcard capability of regular expressions to make a single call to `tpsubscribe` that covers a whole category of events.

BEA Tuxedo ATMI Administrative Services

A set of system servers provides the following administrative services needed by the BEA Tuxedo ATMI environment:

- Application queue management
- Centralized application configuration
- Distributed application management
- Dynamic application reconfiguration
- Event management
- Security management
- Startup and shutdown of an application
- Transaction management
- Workstation management

Managing Application Queues

Queueing enables programmers to write applications that communicate by accessing one or more queues. Because of the location transparency of queues, administrators can move queues from one machine to another without requiring any programming changes.

The MIB consists of a queue device, queue spaces, and queues (required by an application), and the BEA Tuxedo system servers that enqueue and dequeue messages from a queue space. Administrators can use the BEA Administration Console or command-line utilities to define the queue spaces, queues, and administrative servers in the MIB.

Using qmadmin to Administer Application Queues

The command-line utility `qmadmin` allows you to perform all administration functions for the application queues in a configuration, that is, setting up the universal device list (UDL) and volume table of contents (VTOC) that will contain a queue, defining queue spaces within a queue device, and so on. `qmadmin` enables you to manipulate the file system. Using some run-time monitoring capabilities, you can see how many messages are in queues or how many headers are in messages. You can also change characteristics of queues or messages on queues, delete messages on queues, change the size of devices, and so on. In an application you can have multiple application queue devices, and run application queues on multiple machines. Each machine has its own queue device, so you can run `qmadmin` to monitor and manage a particular application queue device on each machine.

Utility	Description
`qmadmin`	Provides for the creation, inspection and modification of message queues. The name of the device (file) on which the universal device list resides (or will reside) for the queue space may either be specified as a command-line argument or through the environment variable `QMCONFIG`. If both are specified, the command option is used.

Using tmconfig to Modify Your Configuration

The `tmconfig` command enables you to browse and modify the `TUXCONFIG` file and its associated entities, and to add new components (such as machines and servers) while your application is running.

When you modify your configuration file (`TUXCONFIG` on the `MASTER` machine), the `tmconfig` command:

- Updates the `TUXCONFIG` file on all machines in the application that are currently booted.
- Propagates the `TUXCONFIG` file automatically to new machines as they are booted.
- Runs as a BEA Tuxedo system client.

Note: Refer to the `tmconfig, wtmconfig(1)` and `TM_MIB(5)` in the *BEA Tuxedo Command Reference* and the *File Formats, Data Descriptions, MIBs, and System Processes Reference* for information on the semantics, range values, and validation of configuration parameters.

Managing Your Configuration

The configuration of any application is primarily controlled by the creation and maintenance of a configuration file, or UBBCONFIG file. Managing your configuration involves the following tasks:

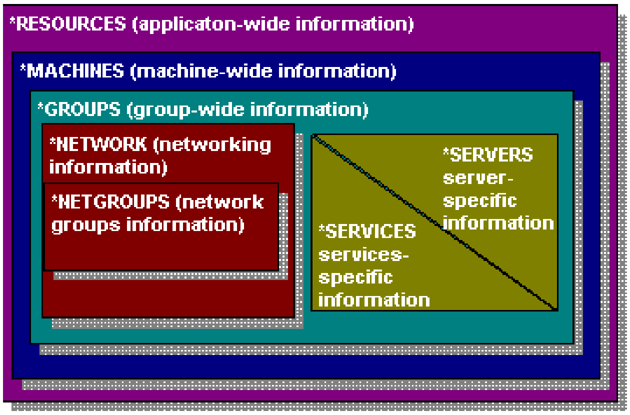
- Creating the configuration file to suit your application needs
- Making permanent configuration changes by updating the UBBCONFIG file
- Changing your configuration while the application is running

Creating the Configuration File

Application configuration data is maintained in the UBBCONFIG, an ordinary text file on the MASTER machine. The configuration file (UBBCONFIG) is a repository that contains all the information necessary to boot an application, such as lists of its resources, machines, groups, servers, available services, and so on. Once written, the UBBCONFIG file is compiled into a binary file, TUXCONFIG. (If you are developing a multidomain application, you must provide a configuration file for each domain in the application.) An application cannot run without a configuration file.

The UBBCONFIG file consists of eight sections, five of which are required for all configurations: RESOURCES, MACHINES, GROUPS, SERVERS, and SERVICES. The RESOURCES and MACHINES sections must be the first and second sections, respectively (as illustrated in the following diagram). GROUPS must be ahead of SERVERS and SERVICES.

Figure 3-4 UBBCONFIG File


*RESOURCES (applicaton-wide information)
*MACHINES (machine-wide information)
*GROUPS (group-wide information)
*NETWORK (networking information)
*NETGROUPS (network groups information)
*SERVERS server-specific information
*SERVICES services-specific information

- `RESOURCES`—(required) contains system-wide parameters that describe the application as a whole
- `MACHINES`—(required) contains logical names and types of physical machines
- `GROUPS`—(required) associates servers with resource managers and machines
- `SERVERS`—identifies each server in the application
- `SERVICES`—identifies each service, and specifies priority, loading, and so on
- `NETWORK`—contains configuration data for LAN environments
- `ROUTING`—contains data-dependent routing tables
- `NETGROUPS`—allows for multiple `BRIDGE`s per machine

Your particular configuration determines which sections of the `UBBCONFIG` file are required. Once you have written your `UBBCONFIG` file, you must compile it into a binary file called `TUXCONFIG`. You can generate your `TUXCONFIG` file by running the the `tmloadcf(1)` command or by using the BEA Administration Console.

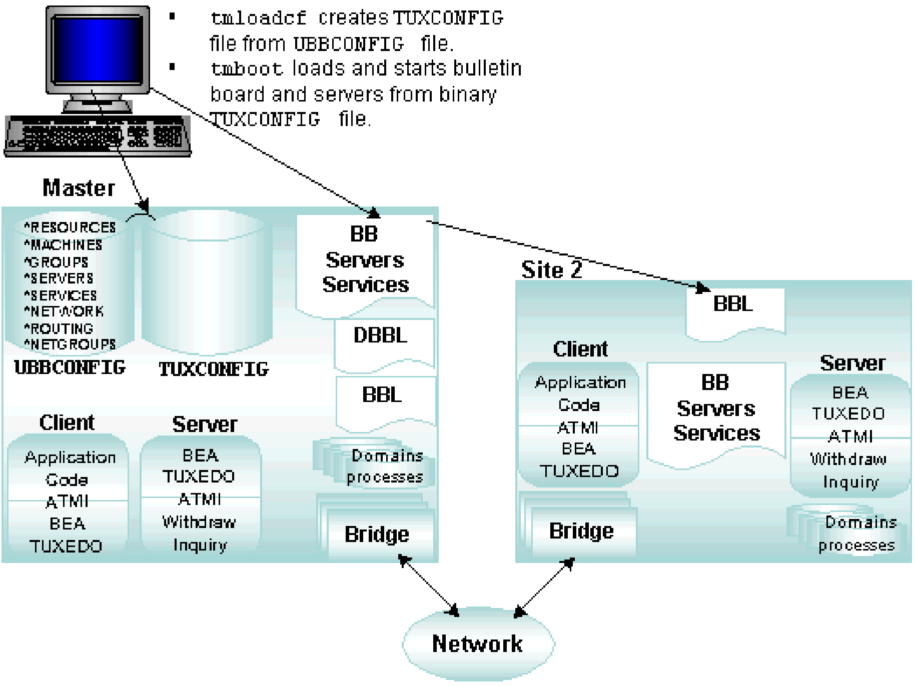
See Also

- "How to Create a Configuration File" on page 3-2 in *Setting Up a BEA Tuxedo Application*

Making Permanent Configuration Changes

To make permanent configuration changes, the administrator can use a text editor to update the configuration parameters in the `UBBCONFIG` file, and use the `tmloadcf` utility to load the text file into the binary `TUXCONFIG` file used by the BEA Tuxedo system. When the application is started, `tmboot` loads `TUXCONFIG` into shared memory to establish the bulletin board, propagating the changes to remote machines if necessary.

Figure 3-5 Configuration Management



Managing Your Configuration Dynamically

Administrators can use the BEA Administration Console or the BEA Tuxedo system command-line utilities to reconfigure applications dynamically, adjusting parameters to respond to varying system loads while the system is running. A revised `TUXCONFIG` file is propagated automatically to all machines in the system as it is updated. However, many `RESOURCES` parameters cannot be changed while the system is running.

Examples of tasks you can do dynamically include: adding servers or machines, deleting machines, and so forth. To ensure that the text and binary versions of your configuration file (`UBBCONFIG` and `TUXCONFIG`, respectively) always match, you need to back them up and synchronize them by using `tmunloadcf`. This command translates the binary file to a text version.

You can change most elements of the system dynamically. You can, for example, spawn new servers, add new machines, or change timeout parameters. There are, however, a few things you cannot change while a system is running:

- Any parameter in the configuration file that affects the size and shape of the bulletin board cannot be changed. Many such parameters are named with the prefix “`MAX`,” such as the `MAXGTT` parameter, which specifies the maximum number of in-flight transactions allowed within the BEA Tuxedo system at any time.
- The processor name of a machine within a particular application cannot be changed. (You can add new machines with different names but you cannot change the name of an existing machine.)
- The values of server executables, assigned to run on `MASTER` and `BACKUP` machines, cannot be changed.

See Also

- “Performing Dynamic Operations Using tmadmin(1)” on page 3-26

Performing Dynamic Operations Using tmadmin(1)

Using the `tmadmin(1)` command, you can perform any of the following operations to a running application:

- Monitor performance by checking statistics on groups, servers, and services (`bbstats`, `bbparms`).
- Modify server and service parameters such as those that change load values (`changeload`), suspend and resume services (`suspend` and `resume`), advertise and unadvertise services (`advertise` and `unadvertise`), and change the `AUTOTRAN` timeout value (`changetrantime`).
- Boot (`boot`), cleanup (`pclean`), and migration (`migratemach`, `migrategroup`).

Commonly Used tmadmin Commands

`tmadmin` provides subcommands that enable you to monitor your run-time system, tune your application, and dynamically configure your application. Following is a list of the most commonly used `tmadmin` commands. (For a comprehensive list of the `tmadmin` commands, refer to the `tmadmin(1)` in the *BEA Tuxedo Command Reference*.)

- `help`—provides you with a list of subcommands, their abbreviation, arguments, and descriptions.
- `printserver` (`psr`)—prints information for application and administrative servers.
- `printservice` (`psc`)—prints information for application and administrative services.
- `printclient` (`pclt`)—prints information for the specified set of client processes. If no arguments or defaults are set, then information on all clients is printed.

Sample Output from the tmadmin Command

Following is sample output from the `tmadmin printserver` (`psr`) command, which provides information about application and administrative servers.

Figure 3-6 Sample Output from the tmadmin printserver Command

```
>psr

Prog Name      Queue Name  Grp Name      ID RqDone Load Done Current Service
---------      ----------  --------      -- ------ --------- ---------------
BBL            83108       SITE1          0      1        50 (  IDLE )
AUDITC         auditc      BANKB1         1      0         0 (  IDLE )
XFER           00001.00101 BANKB1       101      1        30 (  TRANSFER )
TMS_SQL        BANKB1_TMS  BANKB1     30001      0         0 (  IDLE )
ACCT           00001.00102 BANKB1       102      0         0 (  IDLE )
TMS_SQL        BANKB1_TMS  BANKB1     30002      0         0 (  IDLE )
BAL            00001.00103 BANKB1       103      6         7 (  IDLE )
BTADD          00001.00104 BANKB1       104      0         0 (  IDLE )
BALC           00001.00105 BANKB1       105      0         0 (  IDLE )
TLR            tlr1        BANKB1       111      0         0 (  IDLE )
TLR            tlr1        BANKB1       112      3       110 (  WITHDRAWAL )
TLR            tlr1        BANKB1       113      0         0 (  IDLE )
TLR            tlr1        BANKB1       114      0         0 (  IDLE )
TLR            tlr1        BANKB1       115      0         0 (  IDLE )
TLR            tlr1        BANKB1       116      9       100 (  IDLE )
TLR            tlr1        BANKB1       117     20      2048 (  IDLE )
TLR            tlr1        BANKB1       118     30       600 (  IDLE )
TLR            tlr1        BANKB1       119      0         0 (  IDLE )
TLR            tlr1        BANKB1       120      0         0 (  IDLE )
>
```

See Also

- “How a tmadmin Session Works” on page 2-13 in *Administering a BEA Tuxedo Application at Run Time*
- “Using Command-line Utilities to Monitor Your Application” on page 2-10 in *Administering a BEA Tuxedo Application at Run Time*

Managing a Distributed Application Centrally

Even if your BEA Tuxedo application is large and complex, you can perform all run-time administrative functions from one `MASTER` machine. You can do so using the BEA Tuxedo system-supplied command-line utilities, or the BEA Administration Console, or through your third-party administration tools used with the BEA Manager product.

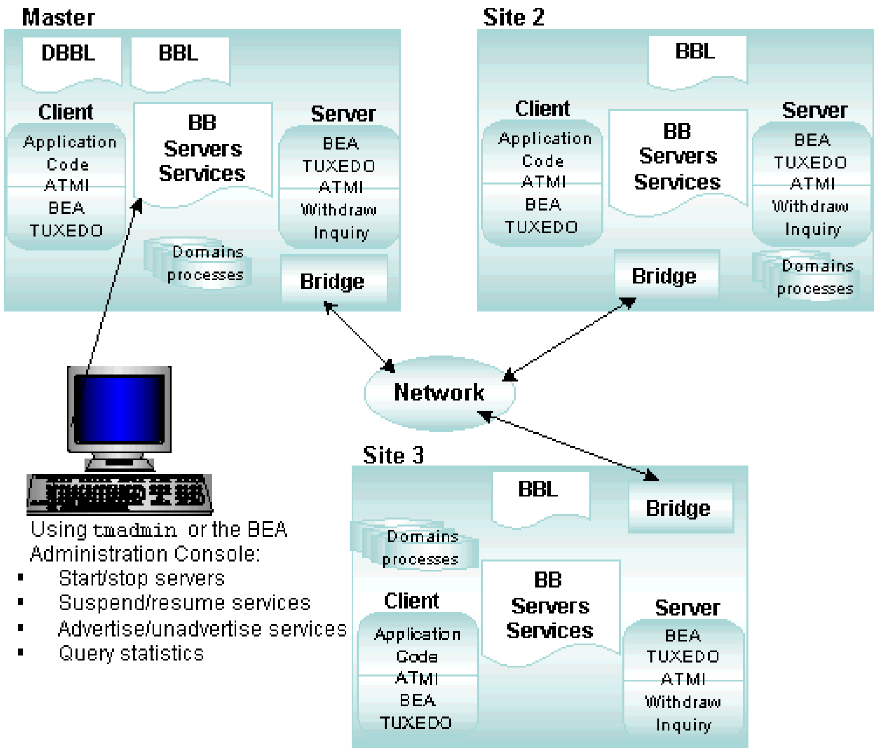
From the `MASTER` machine, you can configure your application, initiate start up and shutdown, and perform administrative tasks during run time. All other machines can query the `MASTER` machine. From the `MASTER` machine, you have control over configuration, fault management, security, monitoring, and performance.

You can use the following two methods to make changes to your system while it is running:

- The BEA Administration Console—a graphical user interface (GUI) to the commands that perform administrative tasks, including dynamic system modification.
- The tmadmin command—a shell-level meta-command that enables you to run 50 subcommands for performing various administrative tasks, including dynamic system modification.

Because it is a graphical user interface, the BEA Administration Console is simpler to use than the `tmadmin` command interpreter. If you prefer using a GUI, bring the BEA Administration Console up on your screen as soon as you are ready to begin an administrative task. Graphics and online help provided with the BEA Administration Console guide you through any task you need to perform. The following illustration shows how you can use the `tmadmin` command or the BEA Administration Console to control a run-time application. All operations can be performed from the `MASTER` machine. The utilities directly affect the bulletin board on the `MASTER` machine, and updates are distributed to other bulletin boards automatically.

Figure 3-7 Centralized Control of a Distributed Application



See Also

- "Using the BEA Administration Console" on page 3-4
- "Performing Dynamic Operations Using tmadmin(1)" on page 3-26

Managing Security

Administrators can configure applications with appropriate levels of security provided by the BEA Tuxedo system. Incremental levels of authentication and authorization can be used to define access to an application. Levels can vary from *no authentication* for highly secure environments, to a password or an access control list (ACL) that filters who can use services, post an event, and enqueue or dequeue a message on a queue.

With an ACL, not only is a user authenticated when joining an application, but permissions are checked automatically when attempts are made to access application entities, such as services. When an ACL is created for a resource, users not included on the list are denied access to the resource. Resources unprotected by an ACL are accessible by any client who successfully joins the application. Resources unprotected by an ACL with the `MANDATORY_ACL` security option specified, are denied for any client who joins the application.

An application can be configured so that all servers (except `AUTHSVR`, the BEA Tuxedo administration server) have restricted access to shared resources, such as shared memory and message queues. When a client joins an application, `AUTHSVR` provides an authentication service that verifies whether the user has the correct authentication level (in the MIB). This service is transparent to the programmer.

See Also

- "Selecting Security Options" on page 3-31
- "Setting Up Security" on page 3-32
- "Administering Security" on page 2-1 in *Using Security in CORBA Applications*
- "Programming Security" on page 3-1 in *Using Security in CORBA Applications*

Selecting Security Options

The following are the security options provided by the BEA Tuxedo system:

- No authentication—clients do not have to be verified before joining an application.
- Application Password—a single password is defined for an entire application and clients must provide the password to join the application.
- User-level Authentication—in addition to an application password, each client must provide a valid username and application-specific data such as a password to join the application.
- Optional Access Control List (ACL)—clients must provide an application password, a username, and a user password. If there is no ACL associated with a user name, permission is granted. This practice enables an administrator to configure access for only those resources that need more security; ACLs need not be configured for services, queues, or events that are open to everyone.
- Mandatory Access Control List (ACL)—clients must provide an application password, a username, and a user password. This level is similar to optional ACL, but an access control list must be configured for every entity (such as a service, queue, or event) that users can access. If mandatory ACLs are being used and there is no ACL for a particular entity, permission for that entity is denied.
- Link-Level Encryption—users of BEA Tuxedo System Security can establish data privacy for messages moving over the network links that connect the machines in a BEA Tuxedo application. The BEA Tuxedo system encrypts data before sending it over a network link and decrypts it as it comes off the link. Three levels of security are offered: 0-bit (no encryption), 56-bit (international), or 128-bit (U.S. and Canada).
- Public key encryption—consists of message-based encryption and message-based digital signature. Message-based encryption reveals user data only to designated recipients. With message-based digital signature, a sending process must prove its identity, and bind that proof to a specific message buffer. Any third party can verify the signature’s authenticity. Undetected tampering is impossible because a digital signature contains a cryptographically secure

checksum computed on the entire contents of a buffer. A digital signature also contains a tamper-proof stamp based on the originating machine's local clock.

- Auditing—collects, stores, and distributes information about operating requests and their outcomes.

Setting Up Security

The type of administrative work and/or programming you must do to set up security for your application depends upon the security options that you choose. Administratively, you need to configure the MIBs using either the BEA Administration Console or the command-line utilities.

You can also build your own security mechanisms. To do so, set the application security level to User-Level Authentication and specify an application service that performs authentication in the BEA Tuxedo MIB.

To enable authentication and authorization, administrators must configure the following in the MIB:

- `AUTHSVR` server
- Identity and passwords of authorized users
- Access control lists used on services, queues, and/or events

Starting Up and Shutting Down Your Application

To start an application, you need to perform the following tasks as stated in *Administering a BEA Tuxedo Application at Run Time*.

1. Set the environment variables as described in “How to Set Your Environment” on page 1-3.
2. Create the `TUXCONFIG` file as described in “How to Create the TUXCONFIG File” on page 1-4.
3. Propagate the BEA Tuxedo software as described in “How to Manually Propagate the Application-Specific Directories and Files” on page 1-5.
4. Create a `TLOG` device, (if required) as described in “How to Create a TLOG Device” on page 1-6.
5. Start `tlisten` at all sites (MP environments) as described in “How to Start tlisten at All Sites” on page 1-7.
6. Boot the application as described in “How to Boot the Application” on page 1-9.

To shut down an application, you need to perform the following task.

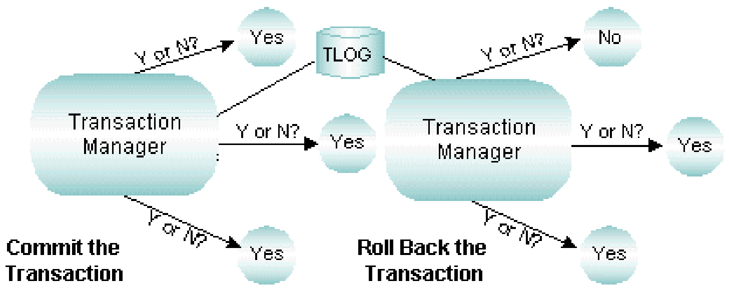
Run `tmshutdown` on the `MASTER` machine as described in “How to Shut Down Your Application” on page 1-11.

Managing Transactions

A powerful feature of the BEA Tuxedo system is the ability to manage transactions for database applications that support the XA interface. Transactions simplify the writing of distributed applications. They allow your application to cope more easily with a large set of problems that can occur in distributed environments, such as machine, program, and network failures.

In a distributed architecture, a local machine involved in a transaction can communicate with a remote machine which may, in turn, communicate with another remote machine. The communication and the work done by the remote machines is part of the transaction, and integrity must be maintained. Keeping track of distributed transaction processing (DTP) can be a complex task because the system must maintain enough information about a transaction to be able to roll it back (that is, to undo it) at any moment.

Figure 3-8 Transaction Management



To keep track of the participants in a transaction, the BEA Tuxedo system creates a transaction log. To maintain the state of an application as represented by the contents of the computer's memory, the BEA Tuxedo system uses one or more resource managers (or RM; a collection of information and processes for accessing it, such as a database management system). To coordinate all the operations performed and all the modules affected by a transaction, the BEA Tuxedo system uses a Transaction Manager (TM), which directs the actions of the RMs. Together, TMs and RMs maintain the atomicity of a distributed transaction.

Coordinating Operations with a Transaction Manager Server (TMS)

The BEA Tuxedo Transaction Manager (TM) is responsible for coordinating global transactions involving system-wide resources. Local resource managers (RMs) are responsible for individual resources. The transaction manager server (`TMS`) begins, commits, and aborts transactions involving multiple resources. The server uses an embedded SQL interface to the RM to read and update the database accessed by the server group. The `TMS` and RMs use the XA interface to perform all or none of the resource work in a *global transaction*.

Tracking Participants with a Transaction Log (TLOG)

A global transaction is logged in the transaction log (`TLOG`) only when it is in the process of being committed. The `TLOG` records the reply from the global transaction participants at the end of the first phase of a 2-phase-commit protocol. A `TLOG` record indicates that a global transaction should be committed; no `TLOG` record is written for those transactions that are to be rolled back. In the first phase, or pre-commit, each Resource Manager must commit to performing the transaction request. Once all parties commit, transaction management commits and completes the transaction. If either tasks fails because of an application or system failure, both tasks fail and the work performed is undone or "rolled back" to its initial state.

The `TMS` that coordinates global transactions uses the `TLOG` file. Each machine should have its own `TLOG`.

Note: Customers using the Domains feature should note that the Domains gateway performs the functions of the `TMS` in Domains groups. However, Domains uses its own transaction log containing information similar to that in the `TLOG`, in addition to Domains-specific information.

See Also

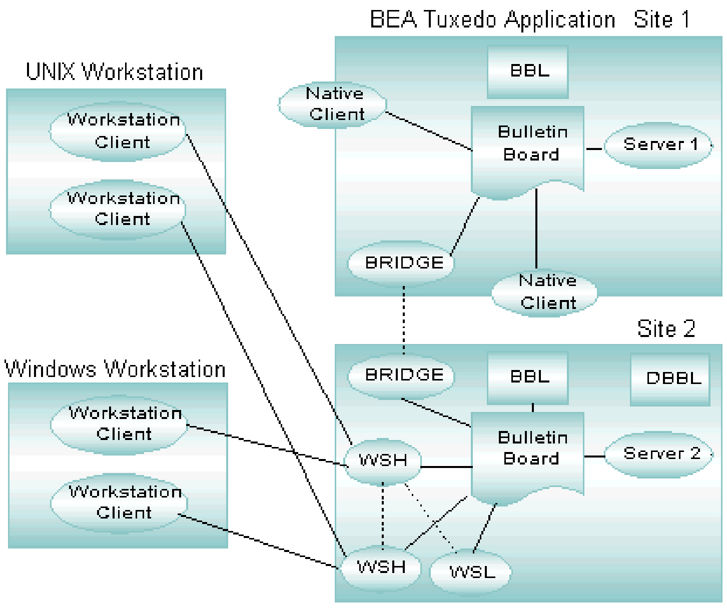
- "Using Transactions" on page 1-18 in *Tutorials for Developing BEA Tuxedo ATMI Applications*
- "Configuring Your ATMI Application to Use Transactions" on page 5-1 in *Setting Up a BEA Tuxedo Application*

Managing Workstations

Workstation clients need enough of the BEA Tuxedo system software to package the information associated with a request. They can then send that information to a system that supports all the BEA Tuxedo system software, including ATMI functions and networking software.

The administrator configures one or more Workstation Listeners (WSLs) to be ready for connection requests from Workstation clients. Each WSL uses one or more associated Workstation Handlers (WSHs) to handle the client's workload. Each WSH manages multiple workstations, multiplexes all communication with a particular workstation over a single connection.

Figure 3-9 Handling Workstation Clients

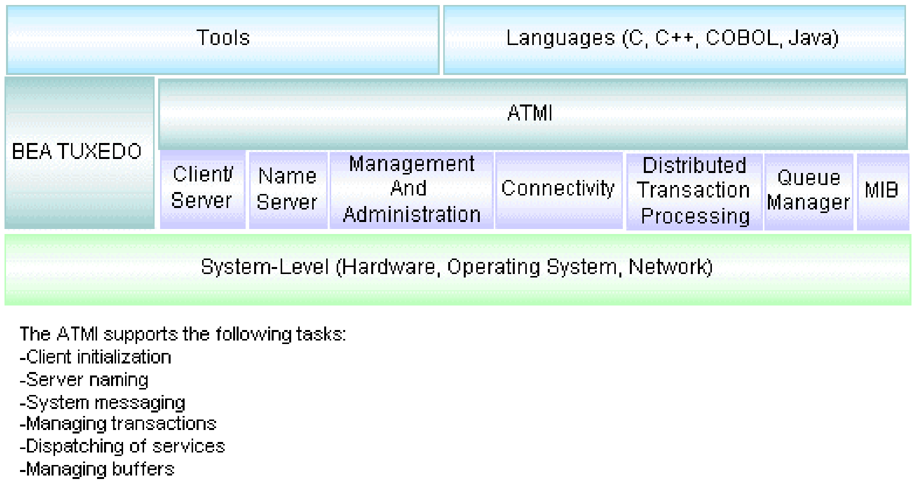


A machine can then handle thousands of Workstation clients. An administrator can define several WSLs in a domain to distribute and balance the workstation communication load across multiple machines. From a programming perspective, all client ATMI programming interfaces are supported for Workstation client development.

Development View: What You Can Do Using the ATMI

The Application-to-Transaction Monitor Interface (ATMI), the BEA Tuxedo API, provides an interface for communications, transactions, and management of data buffers that works in all environments supported by the BEA Tuxedo system. It defines the interface between application programs and the BEA Tuxedo system. The ATMI offers a simple interface for a comprehensive set of capabilities. It implements the X/Open DTP model of transaction processing.

Figure 3-10 Using the ATMI



The ATMI library offers you a variety of functions for defining and controlling global transactions in a BEA Tuxedo application. Global transactions enable you to manage exclusive units of work spanning multiple programs and resource managers in a distributed application. All the work in a single transaction is treated as a logical unit, so that if any one program cannot complete its task successfully, no work is performed by any programs in the transaction. Most ATMI functions support different communication styles. These functions knit together distributed programs by enabling them to exchange data. All ATMI functions send or receive data in typed buffers.

For a list of the ATMI functions (for C and COBOL bindings), grouped by the type of task they perform, refer to Table 2-1, "Using the ATMI Functions," in Chapter 2, "BEA Tuxedo ATMI Architecture."

Note: The use of ATMI transaction management functions is optional.

See Also

- "Using the ATMI to Handle System and Application Errors" on page 2-28 in *Administering a BEA Tuxedo Application at Run Time*
- "Creating a BEA Tuxedo ATMI Client" on page 1-2 in *Tutorials for Developing BEA Tuxedo ATMI Applications*
- "Creating a BEA Tuxedo ATMI Server" on page 1-4 in *Tutorials for Developing BEA Tuxedo ATMI Applications*
- "Using Typed Buffers in Your Application" on page 1-6 in *Tutorials for Developing BEA Tuxedo ATMI Applications*
- "What Are the BEA Tuxedo ATMI Messaging Paradigms?" on page 2-8
- "What Is Meant by Naming?" on page 2-42

Run-time System View: Using Tools in Different Configurations

The BEA Tuxedo system provides tools to create, monitor, and manage both processes and the communication that occurs between processes in a given application. You can use the basic processes and messaging paradigms in many different configurations. Each configuration falls into one of the following run-time categories:

- Single machine application—one or more local or remote clients communicate with one or more servers residing on the same machine.
- Distributed application across multiple machines—one or more local or remote clients communicate with one or more servers residing on several machines in one domain.
- Multiple-domain application—two or more domains communicate with each other.

Run-time System Capabilities

The following table lists the BEA Tuxedo system functionality available in a single-machine application, a distributed application, and a multiple-domain application.

Table 3-2 Functionality Available in Different Types of Configurations

Available Functionality	Single-machine Configuration	Multiple-machine (Distributed) Configuration	Multiple-domain Configuration
ATMI	X	X	X
Messaging paradigms	X	X	X

Table 3-2 Functionality Available in Different Types of Configurations (Continued)

Available Functionality	Single-machine Configuration	Multiple-machine (Distributed) Configuration	Multiple-domain Configuration
Administrative parts:			
Bulletin Board (BB), Bulletin Board Liaison (BBL), `TLOG`, `UBBCONFIG`, `ULOG`, `TUXCONFIG`	X	X	X
Distinguished Bulletin Board Liaison (DBBL)		X	X
Bridges		X	X
Domains processes: `DMADM`, `GWADM`, `GWTDOMAIN` (for TDomains), `dmloadcf`, `dmunloadcf`, and `DMCONFIG`, `DMTLOG` and `BDMCONFIG`			X
Application processes: clients, servers, and services	X	X	X
Queuing	X	X	X
Transaction management	X	X	X
Event management	X	X	
Security management	X	X	X

What Is a Single-machine Configuration?

A single-machine configuration consists of one or more local or remote clients that communicate with one or more servers residing on a single machine running one or more business applications. Even though it may include multiple applications, this type of configuration is considered a single domain because it is administered as a single entity.

All the managed elements (services, servers, and so on) of all the applications in this configuration are defined in and controlled from one BEA Tuxedo configuration file. The basic parts of a single-machine configuration when installed and running on a single machine are illustrated in the following diagram.

Figure 3-11 A Single-machine BEA Tuxedo Configuration

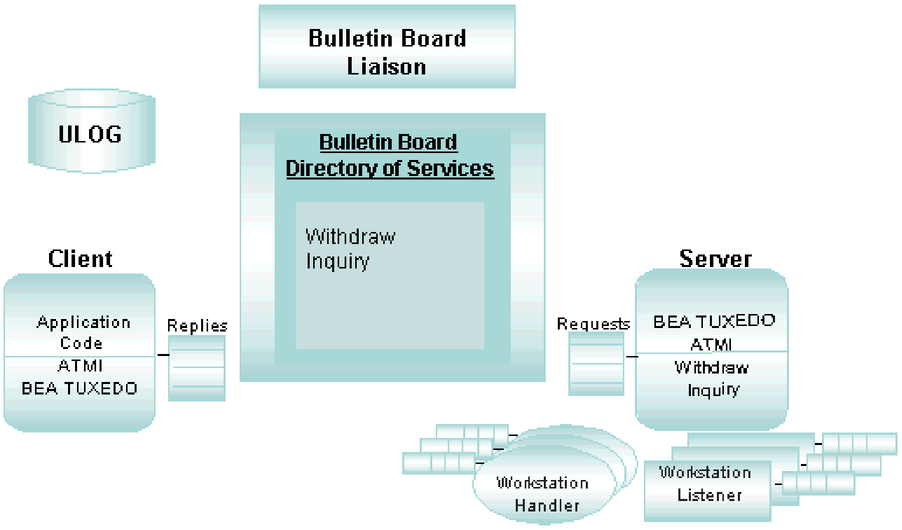


Table 3-3 Parts of a Single-machine Configuration

Single-machine Part	Description
Bulletin Board (BB)	A shared memory segment that holds configuration and dynamic information for the system. It is available to all BEA Tuxedo processes.
Bulletin Board Liaison (BBL)	A BEA Tuxedo administrative process that monitors both the data stored in the bulletin board (including any changes made to it) and all application programs.
Clients	Executable programs that periodically request services through the BEA Tuxedo system. (Client programs are normally written by customers.)

Table 3-3 Parts of a Single-machine Configuration (Continued)

Single-machine Part	Description
Message queues	Communication between clients and servers is performed through operating-system supported, memory-based message queues.
Messaging paradigms	Different models of transferring messages between a client and a server. Examples include request/response mode, conversational mode, events, and unsolicited communication.
Servers	Executable programs that offer named services through the BEA Tuxedo system. (Server programs are normally written by customers.)
Workstation Handler (WSH)	A multi-contexted gateway process on a server that manages service requests from Workstation clients (that is, client processes running on remote sites).
Workstation Listener (WSL)	A server process running on an application site that listens for and distributes connections from Workstation clients (client process running on a remote site).
ULOG (User Log)	A file in which error messages are stored.

See Also

- "How to Create a Configuration File" on page 3-2 in *Setting Up a BEA Tuxedo Application*

What Is a Multiple-machine (Distributed) Configuration?

A distributed-domain (or multiple-machine) configuration consists of one or more business applications running on multiple machines. Although it includes multiple machines, this type of configuration is considered a single domain because it is administered centrally as a single entity. In other words, all the elements (services, servers, machines, and so on) of all the applications on all the machines in this configuration are defined in, and controlled from, one BEA Tuxedo configuration file.

As a business grows, application developers may need to organize different segments of the business by sets of functionality that require administrative autonomy but allow sharing of services and data. Each functionality set defines an application that may span one or more machines, and that is administered independently from other applications. Such a functionally distinct application is referred to as a domain.

The names of domains frequently reflect the functionality provided. When domains have names such as “marketing” and “research and development,” it is easy for customers to find the applications they need.

The basic parts of a configuration distributed across multiple machines are illustrated in the following diagram.

Figure 3-12 Distributed Application

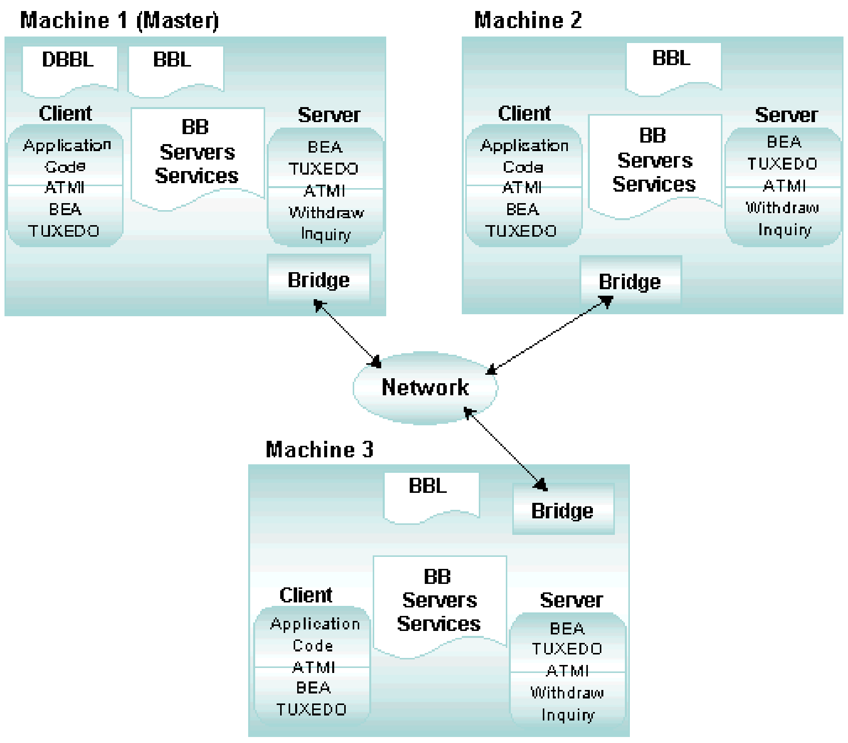


Table 3-4 Parts of a Distributed Configuration

Multiple Machine Part	Description
Bridges	BEA Tuxedo system-supplied servers within a domain that send and receive service requests between machines, and route requests to local servers (literally, to local server queues).

Table 3-4 Parts of a Distributed Configuration (Continued)

Multiple Machine Part	Description
Bulletin Board (BB)	A shared memory segment that holds configuration and dynamic information for the system. It is available to all BEA Tuxedo processes.
Bulletin Board Liaison (BBL)	A BEA Tuxedo administrative process that monitors both the data stored in the bulletin board (including any changes made to it), and all application programs.
Clients	Executable programs that periodically request services through the BEA Tuxedo system. (Client programs are usually by customers.)
Distinguished Bulletin Board Liaison (DBBL)	A process dedicated to making sure that the BBL server on each machine is alive and functioning correctly. This server runs on the *Master* machine of a domain and communicates directly with all administration facilities.
Message queues	Communication between clients and servers is performed through operating-system supported, memory-based message queues.
Messaging paradigms	Different models of transferring messages between a client and a server. Examples include request/response mode, conversational mode, events, and unsolicited communication.
Servers	Executable programs that offer named services through the BEA Tuxedo system. (Server programs are normally written by customers.)
Workstation Handler (WSH)	A multi-contexted gateway process on a server that manages service requests from Workstation clients (that is, client processes running on remote sites).
Workstation Listener (WSL)	A server process running on an application site that listens for and distributes connections from Workstation clients (client processes running on remote sites).
ULOG (User Log)	A file in which error messages are stored.

A configuration that runs on more than one machine requires platform interoperability and server transparency.

- Platform interoperability—means that your application can rely on intermachine communications even when different machines are running different operating systems, without code customization.
- Server transparency—means that a client can access a server without specifying its location. The locations of servers are recorded in the bulletin board and accessed as needed. As a result, servers can be moved, dropped, or added to an application dynamically, without needing to change the application itself.

The DBBL and Bridge servers support these requirements of a distributed-domain configuration.

See Also

- "How to Create the Configuration File for a Multiple-machine (Distributed) Application" on page 3-3 in *Setting Up a BEA Tuxedo Application*
- "Distributing ATMI Applications Across a Network" on page 7-1 in *Setting Up a BEA Tuxedo Application*
- "Creating the Configuration File for a Distributed ATMI Application" on page 8-1 in *Setting Up a BEA Tuxedo Application*
- "Setting Up the Network for a Distributed Application" on page 9-1 in *Setting Up a BEA Tuxedo Application*
- "Managing the Network in a Distributed Application" on page 4-1 in *Administering a BEA Tuxedo Application at Run Time*

What Is a Multiple-domain Configuration?

A multiple-domain configuration consists of two or more domains that communicate with each other. Each domain may be either a single-machine configuration or a multiple-machine configuration. Inter-domain communication is achieved through a highly asynchronous multitasking gateway that processes outgoing and incoming service requests to or from all domains. Multiple BEA Tuxedo domains can be connected, allowing clients in one domain transparent access to services physically located in remote domains. Each domain can share services and data, but is administered separately.

The BEA Tuxedo system provides different types of gateways to accommodate various network transport protocols. Following are the different types of Domains gateways:

- The BEA Tuxedo Domains (TDomains) gateway provides interoperability between two or more BEA Tuxedo applications through a specially designed TP protocol that flows over network transport protocols such as TCP/IP.
- The BEA eLink OSI TP gateways provides interoperability between BEA Tuxedo applications and other transaction processing applications that use the OSI TP standard. OSI TP is a protocol for distributed transaction processing defined by the International Standards Organization (ISO).
- The BEA eLink Adadpter for Mainframe SNA gateway provides interoperability between clients and servers in a BEA Tuxedo domain and clients and servers in an MVS/CICS or MVS/IMS environment in remote SNA domains. It also connects a local BEA Tuxedo domain to multiple SNA networks.
- The BEA eLink Adapter for Mainframe TCP for CICS is a gateway connectivity feature that makes it possible for non-transactional tasks within BEA Tuxedo regions to access services provided by CICS application programs and vice-versa. It enables a BEA Tuxedo domain to communicate via the TCP/IP network transport protocol to a CICS environment.
- The BEA eLink Adapter for Mainframe TCP for IMS is a gateway connectivity feature that provides transparent communications between client and server transactions in an IMS system and a BEA Tuxedo domain, a CICS system, or another IMS system.
- The TOP END Domain Gateway (TEDG) provides interoperability between BEA TOP END systems and BEA Tuxedo domains.

The basic parts of a multiple-domain configuration are illustrated in the following diagram.

Figure 3-13 Multiple-domain Configuration

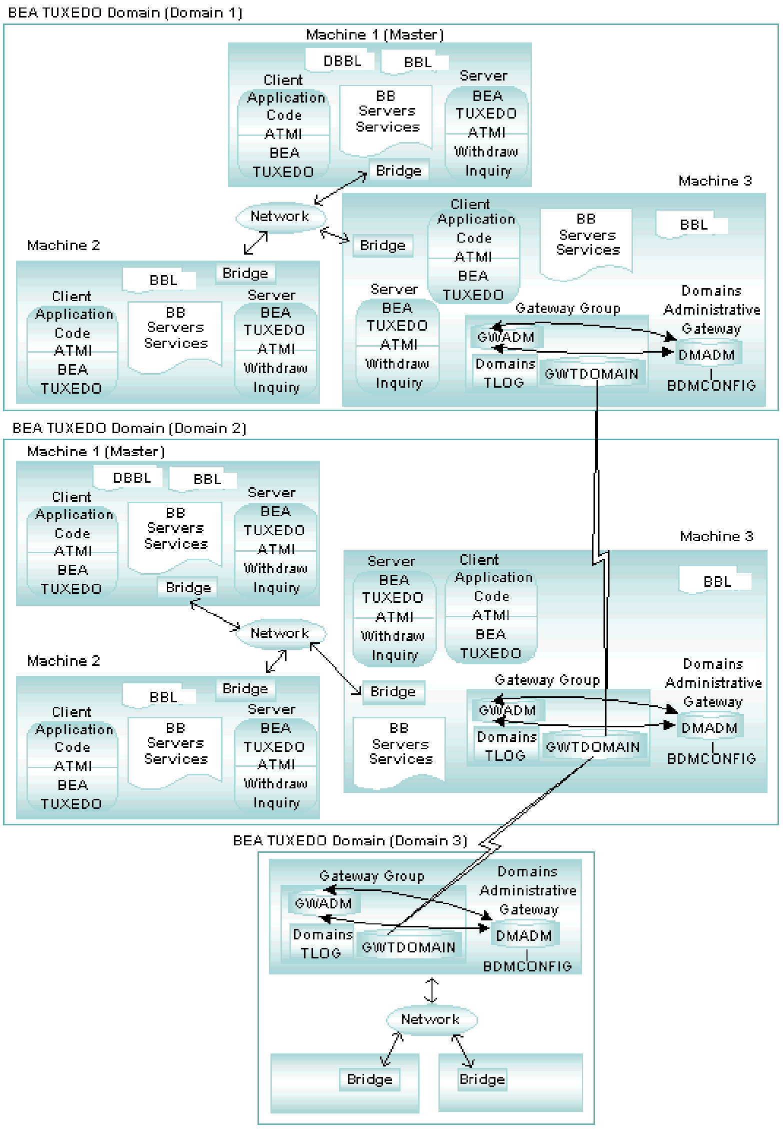


Table 3-5 Parts of a Multiple-domain Configuration

Multiple-domain Part	Description
Bridges	BEA Tuxedo system-supplied servers within a domain that send and receive service requests between machines, and route requests to local servers (literally, to local server queues).
Bulletin Board (BB)	A shared memory segment that holds configuration and dynamic information for the system. It is available to all BEA Tuxedo processes.
Bulletin Board Liaison (BBL)	A BEA Tuxedo administrative process that monitors both the data stored in the bulletin board (including any changes made to it), and all application programs.
Clients	Executable programs that periodically request services through the BEA Tuxedo system. (Client programs are normally written by customers.)
Distinguished Bulletin Board Liaison (DBBL)	Ensures that the BBL servers on each machine are alive and functioning correctly. This server runs on the *Master* machine of an application and communicates directly with any administration facility.
Domains tools: `DMADM`, `GWADM`, `GWTDOMAIN`, `dmloadcf`, `dmunloadcf`, and `DMCONFIG`	■ `DMADM`—the Domains administrative server. ■ `GWADM`—the gateway group administrative server that registers with the `DMADM` server to obtain configuration information used by the gateway group. ■ `GWTDOMAIN`—the gateway process that provides connectivity to remote gateway processes (for TDomains). ■ `dmloadcf`—translates the `DMCONFIG` file to a binary `BDMCONFIG` configuration file. ■ `dmunloadcf`—translates the `BDMCONFIG` configuration. file from the binary representation into ASCII. ■ `DMCONFIG`—the Domains configuration file.
Message queues	Communication between clients and servers is performed through operating-system supported, memory-based message queues.

Table 3-5 Parts of a Multiple-domain Configuration (Continued)

Multiple-domain Part	Description
Messaging paradigms	Different models of transferring messages between a client and a server. Examples include request/response mode, conversational mode, events, and unsolicited communication.
Servers	Executable programs that offer named services through the BEA Tuxedo system. (Server programs are normally written by customers.)
Workstation Handler (WSH)	A multi-contexted gateway process on a server that manages service requests from Workstation clients (that is, client processes running on remote sites).
Workstation Listener (WSL)	A server process running on an application site that listens for and distributes connections from Workstation clients (client processes running on remote sites).
ULOG (User Log)	A file in which error messages are stored.

See Also

- "What Is a Single-machine Configuration?" on page 3-40
- "What Are the Domains Administrative Tools?" on page 3-55
- "How to Create the Configuration File for a Multiple-domain Application" on page 3-4 in *Setting Up a BEA Tuxedo Application*

Features of a Multiple-domain Configuration

A configuration that includes more than one domain requires platform interoperability and server transparency:

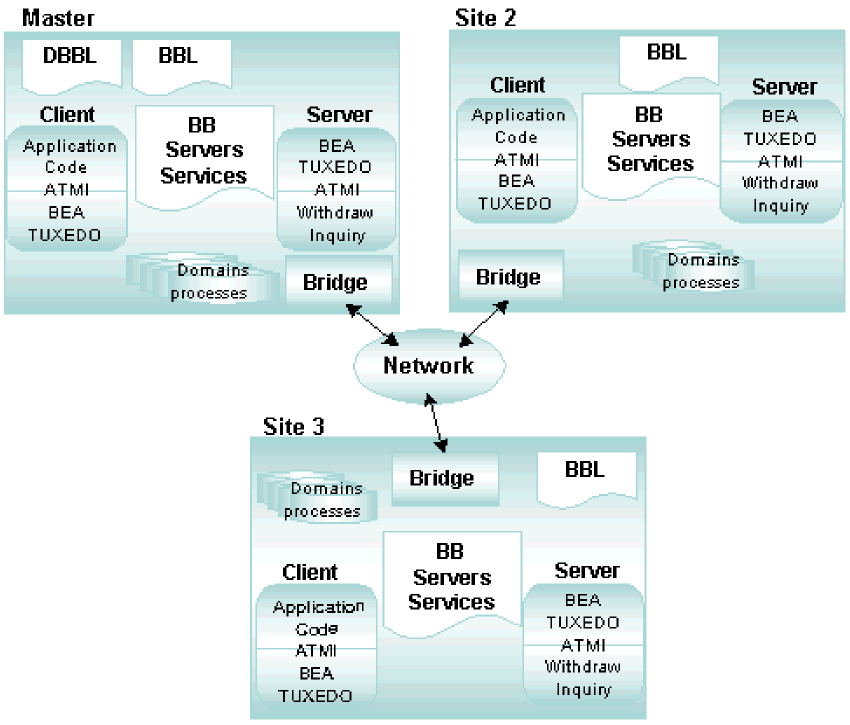
- Platform interoperability—means that your application can rely on intermachine communications even when different machines are running different operating systems, without code customization.
- Server transparency—means that a client can access a server without specifying its location. The locations of servers are recorded in the bulletin board and accessed as needed. As a result, servers can be moved, dropped, or added to an application dynamically, without needing to change the application itself.

What Is a BEA Tuxedo Bridge?

A BEA Tuxedo Bridge is a server, provided by the BEA Tuxedo system, for sending and receiving service requests between machines, and routing requests to local server queues.

Each bridge enables a network connection to be created with every other bridge in the system. Network connections are established as needed and then maintained indefinitely. Bridges are hidden servers, that is, they are started and stopped automatically, as needed, without an explicit configuration entry. Messages are asynchronously sent across these persistent network connections. No network connection overhead is incurred for individual messages.

Figure 3-14 Using Bridges in a Multiple-machine (Distributed) Application



See Also

- "Setting Up the Network for a Distributed Application" on page 9-1 in *Setting Up a BEA Tuxedo Application*
- "Creating the Configuration File for a Distributed ATMI Application" on page 8-1 in *Setting Up a BEA Tuxedo Application*

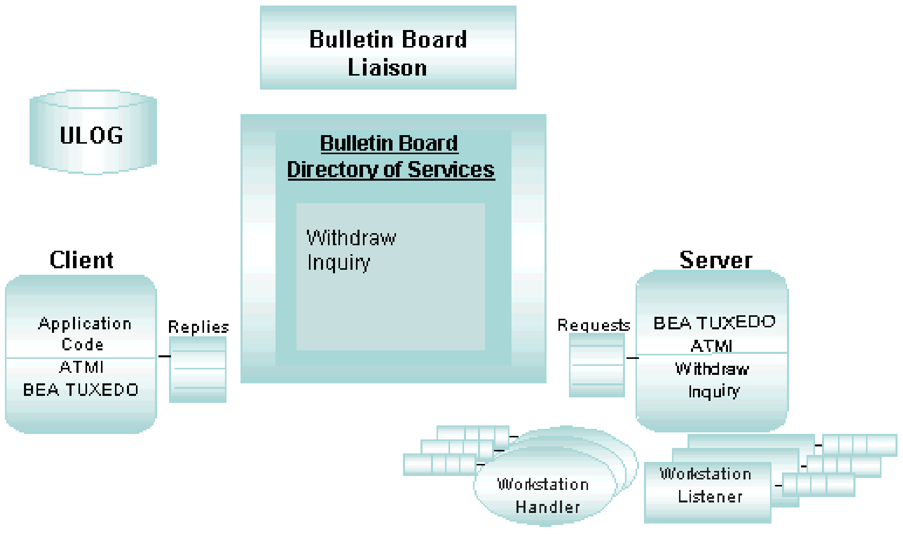
What Is the Role of the Bulletin Board and Bulletin Board Liaison?

The bulletin board (BB) is a memory segment in which all the application configuration and dynamic processing information is held at run time. It provides the following functionality:

- Assigns service requests to specific servers. When a service is called, the bulletin board looks up servers that offer the requested service. Based on this information, and any data-dependent routing criteria, the bulletin board places the request data on the request queue of a valid server.
- Maintains dynamic information about the *state* of an application, such as how many requests are waiting on a given server's queue and how many requests have been processed.
- Provides server location transparency, allowing an application to be developed independently of deployment. Therefore, development and deployment costs are minimized.
- Supports service name aliases, allowing multiple names to be assigned to the same service. This capability is useful for constructing interpreters, such as gateways.

The Bulletin Board Liaison (BBL) is a BEA Tuxedo server that performs periodic health checks of the bulletin board and coordinates functions of all parts of the system.

Figure 3-15 Bulletin Board and Bulletin Board Liaison



What Are Clients and Servers?

- Client—a program that collects a request from a user and passes that request to a server capable of fulfilling it. It can reside on a PC or workstation as part of the front-end of an application gathering input from users. It can also be embedded in software that reads a communication device such as an ATM machine from which data is collected and formatted before being processed by BEA Tuxedo servers.
- Server—a process that oversees a set of services and dispatches services automatically for clients that request them. A service, in turn, is a function within the server program that performs a particular task needed by a business. A bank, for example, might have one service that accepts deposits and another that reports account balances. A server at this bank might receive requests from clients for both services. It is the server’s job to dispatch each request to the appropriate service.

What Is the Distinguished Bulletin Board Liaison (DBBL)?

The Distinguished Bulletin Board Liaison (DBBL) is the server that makes it possible to distribute an application across multiple machines. The DBBL ensures that the Bulletin Board Liaison (BBL) server on each machine is alive and functioning correctly. The DBBL runs on the master machine of an application and communicates directly with all administration facilities.

The DBBL ensures that configuration and service addressing information is replicated to the bulletin board on each machine in the configuration. Servers located on remote machines are accessed through the bridge on the local machine. Servers on the local machine are accessed directly. All local communications are performed through high performance operating system message queues. Remote communications are performed in two phases. First, service requests are forwarded to a remote machine through the (local) bridge. Second, when a request reaches the remote machine, operating system messages are used to send the request to the appropriate server.

What Are the Domains Administrative Tools?

To build a multiple-domain configuration, you need to integrate your existing BEA Tuxedo application with other domains. You need to ensure interoperability across domains, preserve access to services on all domains, and accept service requests from all domains. You can perform these functions through a highly asynchronous multitasking gateway that processes outgoing and incoming service requests to or from all domains. To use the gateway, you must add entries for domain gateway groups and gateway servers to the `TUXCONFIG` file. The following illustration shows the tools provided by the BEA Tuxedo system for setting up and maintaining a multiple-domain configuration.

Figure 3-16 Domains Administrative Tools

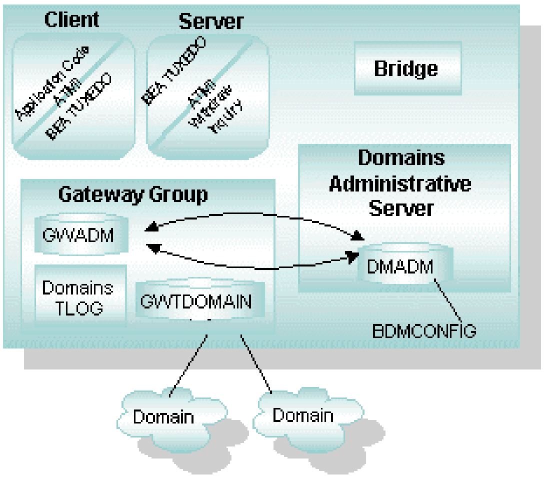


Table 3-6 Domains Administrative Tools

Domains Tool	Description
`dmadmin`(1)	A command that allows you to configure, monitor, and tune domain gateway groups dynamically. Use this command to update the `BDMCONFIG` file while an application is running. The command acts as a front-end process that translates administrative commands to service requests to the `DMADMIN` service, a generic administrative service advertised by the `DMADM` server. The `DMADMIN` service invokes the validation, retrieval, or update functions provided by the `DMADM` server to maintain the `BDMCONFIG` file.
`DMCONFIG`(5), `BDMCONFIG`	All Domains configuration information is stored in a binary file called the `BDMCONFIG` file. You can create and edit the text version of the Domains gateway configuration file, `DMCONFIG`, with any text editor. You can update the compiled `BDMCONFIG` file while the system is running.
`dmloadcf` and `dmunloadcf`	`dmloadcf`—reads the `DMCONFIG` file, checks the syntax, and optionally loads a binary `BDMCONFIG` configuration file. `dmunloadcf`—translates the `BDMCONFIG` configuration file from binary to text format.

Table 3-6 Domains Administrative Tools (Continued)

Domains Tool	Description
`DMADM`(5)	A Domains administrative server that enables you to manage a Domains configuration at run time. `DMADM` provides a registration service for gateway groups. This service is requested by `GWADM` servers as part of their initialization procedure. The registration service downloads the configuration information required by the requesting gateway group. The `DMADM` server maintains a list of registered gateway groups, and propagates to these groups any changes made to the configuration.
`GWADM`(5)	A gateway administrative server that supports run-time administration of a specific gateway group. This server registers with the `DMADM` server to obtain the configuration information used by the corresponding gateway group. `GWADM` accepts requests from `DMADMIN` to obtain run-time statistics or to change the run-time options of the specified gateway group. Periodically, `GWADM` sends an “I-am-alive” message to the `DMADM` server. If no reply is received from `DMADM`, `GWADM` registers again. This process ensures the `GWADM` server always has the current copy of the Domains configuration for its group.
`GWTDOMAIN`(5)	A gateway process that receives and forwards messages from clients and servers in all connected domains (for TDomains).
`BDMCONFIG`	The binary version of the configuration file for a multiple-domain configuration.

What Are IPC Message Queues?

The BEA Tuxedo system uses IPC message queues to support communication between processes that are executed on a particular machine. IPC message queues are transient memory areas, typically provided by the underlying operating system, used for communication between clients and servers. By default, each server has its own IPC message queue on which to receive requests and replies, referred to as a Single Server, Single Queue (SSSQ). If you prefer, however, you can override the default and assign multiple servers to read from the same queue. This arrangement is referred to as Multiple Servers, Single Queue (MSSQ). You can use both SSSQ and MSSQ sets in the same application. Servers can be assigned to either type of queue.

When to Use Single Server, Single Queues (SSSQ)

To understand how SSSQ sets work, consider an analogy that can be found in your supermarket, where there may be several checkout lines. Each line is like a separate queue in which customers wait for a clerk at one register, who determines how fast that line is serviced. If a delay is introduced by one person, each subsequent person is also delayed on that line, but the delay has no effect on other lines. This scheme can be used to load balance and throttle work across several servers offering different kinds of services. Customers with relatively small requests can be processed by a server with a separate queue, thus speeding throughput by guaranteeing available cycles or registers for small requests.

When to Use Multiple Server, Single Queue (MSSQ) Sets

The MSSQ scheme offers additional load balancing through IPC messaging, which is offered by the operating system. One queue is accommodated by several servers offering identical services at all times. If the server queue to which a request is sent is part of an MSSQ set, the message is dequeued to the first available server. Thus load balancing is provided at the individual queue level.

When a server is part of an MSSQ set, it must be configured with its own reply queue. When the server makes requests to other servers, the replies must be returned to the original requesting server; they must not be dequeued by other servers in the MSSQ set.

In many applications, Multiple Server, Single Queue (MSSQ) sets can play an important role. They are ideal when you need to minimize the total waiting time for services. If it is unacceptable for a service request to wait while a server capable of fulfilling that request remains idle, MSSQ sets should be used.

We recommend using an MSSQ set in the following situations:

- Service turnaround time is paramount.
- You have a reasonable number of servers (between 2 and 12).
- Servers offer identical sets of services.
- The messages involved are reasonably sized (less than 75% of the queue size).
- You can configure MSSQ sets to be dynamic so they automatically spawn and reduce servers based upon a queue load.

Note: For fault tolerance, you should always use MSSQ sets with two or more servers.

An MSSQ set is inappropriate when long messages are being passed to services. Long messages can cause a queue to be exhausted. When a queue is exhausted, either non-blocking sends fail or blocking sends block.

We recommend against using an MSSQ set in the following situations:

- Buffer sizes are large enough to exhaust one queue.
- You have a large number of servers. (You can compromise by using a few MSSQ sets.)
- Each server offers different services.

Example

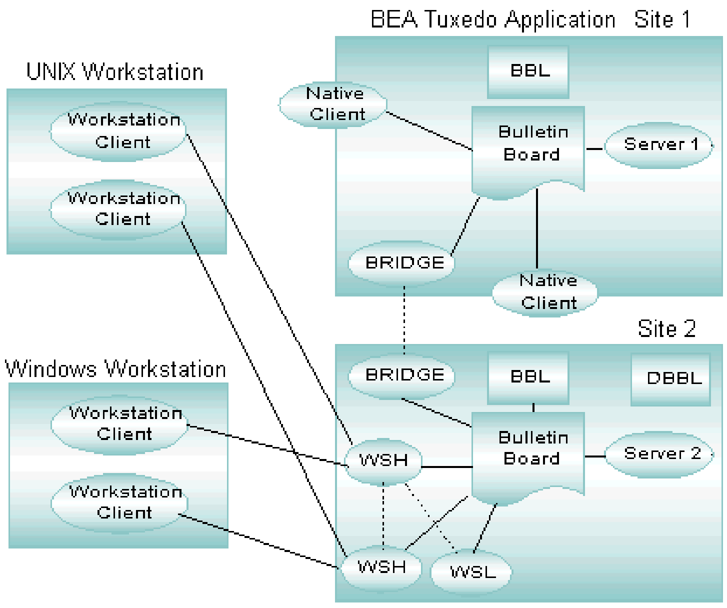
To consider how MSSQ sets work, consider an analogy that can be found in your bank, where several tellers performing identical services handle a single line of customers. The next available teller always takes the next person in line. In this scenario, each teller must be able to perform all customer services. In a BEA Tuxedo environment, all servers set up to share a single queue must offer the identical set of services at all times. The advantage of MSSQ sets is that they offer a second form of load balancing at the individual queue level.

What Are the Workstation Handler and Workstation Listener?

The Workstation component extends the availability of a native BEA Tuxedo application to clients that reside on workstations. With this component, workstations need not be within the administrative domain of the application.

The following figure shows an application with two Workstation clients (WSC). One client is running on a UNIX system workstation, while the other client is running on a Windows 2000 workstation. Both WSCs are communicating with the application through the Workstation Handler (WSH) process. Initially, both joined by communicating with the Workstation Listener (WSL). The Workstation defines an environment in which clients can access the services of an application through a surrogate handler process.

Figure 3-17 BEA Tuxedo Application with the Workstation Component

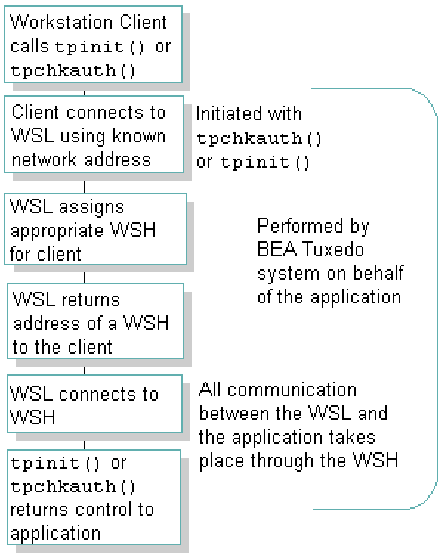


The programming environment on a Workstation is determined by the operating system of the machine. A Local Area Network (LAN) provides a connection to the administrative domain of the application, affording greater flexibility in the choice of hardware and software platforms on which you can deliver application services.

How a Workstation Client Connects to an Application

A Workstation client connects to an application in the following way.

Figure 3-18 WSC Connecting to an Application


Workstation Client calls tpinit() or tpchkauth()
Client connects to WSL using known network address
WSL assigns appropriate WSH for client
WSL returns address of a WSH to the client
WSL connects to WSH
tpinit() or tpchkauth() returns control to application
Initiated with tpchkauth() or tpinit()
Performed by BEA Tuxedo system on behalf of the application
All communication between the WSL and the application takes place through the WSH

What Is the User Log (ULOG)?

The user log (`ULOG`) is a file to which all messages generated by the BEA Tuxedo system—error messages, warning messages, information messages, and debugging messages—are written. Application clients and servers can also write to the user log. A new log is created every day and there can be a different log on each machine. However, a `ULOG` can be shared by multiple machines when a remote file system is being used.

The `ULOG` provides an administrator with a record of system events from which the causes of most BEA Tuxedo system and application failures can be determined. You can view the `ULOG`, a text file, with any text editor. The `ULOG` also contains messages generated by the `tlisten` process. The `tlisten` process provides remote service connections for other machines. Each machine, including the master machine, should have a `tlisten` process running on it.

How Is the ULOG Created?

A `ULOG` is created by the BEA Tuxedo system whenever one of the following activities occurs:

- A new configuration file is loaded.
- An application is booted.

Example of a ULOG Message

The following is an example of a `ULOG` message:

```
121449.gumby!simpserv.27190.1.0: LIBTUX_CAT:262: std main starting
```

A `ULOG` message consists of two parts: a tag and text.

The tag consists of the following:

- *A* 6-digit string (*hhmmss*) representing the time of day (in terms of hour, minute, and second)
- The name of the machine (as returned, on UNIX systems, by the `uname -n` command)
- The name and process identifier of the process that is logging the message. (This process ID can optionally include a transaction ID.) Also included is a thread ID (`1`) and a context ID (`0`).

Note: Placeholders are printed in the `thread_ID` and `context_ID` field of entries for single-threaded applications. (Whether an application is multithreaded is not apparent until more than one thread is used.)

The text consists of the following:

- The name of the message catalog
- The message number
- The BEA Tuxedo system message

The Tag Indicates...	The Text Indicates...
■ The message was written into the log at approximately 12:15 P.M. ■ The machine on which the error occurred was gumby. ■ The message was logged by the simpserv process, which has a process ID of 27190. ■ The thread ID is `1`. ■ The context ID is `0`.	■ The message came from the `LIBTUX` catalog. ■ The number of the message is 262. ■ The message itself reads as follows: `std main starting`.

Note: For more information about a message, note its catalog name and number. With this information, you can look up the message in the appropriate catalog.

Where the ULOG Resides

By default, the user log is called `ULOG.`*`mmddyy`* (where *`mmddyy`* represents the date in terms of month, day, and year) and it is created in the `$APPDIR` directory. You can place this file in any location, however, by setting the `ULOGPFX` parameter in the `MACHINES` section of the `UBBCONFIG` file.