BEA Tuxedo®

Introducing BEA Tuxedo ATMI

Version 10.0
Document Released: September 28, 2007

Contents

1. BEA Tuxedo System Fundamentals

2. BEA Tuxedo ATMI Architecture

4. BEA Tuxedo Management Tools

CHAPTER 1

BEA Tuxedo System Fundamentals

The following sections provide an overview of the BEA Tuxedo programming environment:

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 online transaction processing (OLTP) systems.

The BEA Tuxedo system is a mature product based on over 20 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 online 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.

Architectural Features

The BEA Tuxedo system offers many features to accommodate the architectural aspects of an ATMI application:

- 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.
- 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.
- Scalability—application designers can quickly scale their Tuxedo applications to match varying system load demands because services and servers can be replicated and distributed easily. Designers can set thresholds programmatically to enable the BEA Tuxedo system to spawn new servers or to shut down servers automatically.

Administrative Features

The BEA Tuxedo system offers many features to accommodate the administrative aspects of an ATMI application:

- 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.

- Application-specific and system events notification—the BEA Tuxedo system provides details about application and system events, such as servers unexpectedly terminating and networks failing. When an event is posted by clients or servers, the Tuxedo publish-and-subscribe component looks up all the subscribers to that event and takes appropriate actions, as determined by each subscription.
- Management information base (MIB)—an administrative interface that enables administrators to monitor, configure, and tune their applications through their own programs. It is an implementation-independent management database defined as a set of Field Manipulation Language (FML) attributes, which allows administrators 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.

Programming Features

The BEA Tuxedo system offers many features to accommodate the programming aspects of an ATMI application:

- 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—provide transparent handling of application data across heterogeneous platforms.
- 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, application designers can mix and match databases within an application while maintaining data integrity.
- X/Open TX compliance—the BEA Tuxedo system conforms to the X/Open interface standard for transaction demarcation. BEA Tuxedo also offers its own ATMI interface for transaction demarcation.

Anatomy of the Client/Server Model

In client/server architecture, clients—programs that represent users who need services—and servers—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

The client/server architecture has the following characteristics:

- Asymmetrical protocols—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 inexpensive 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 process or on a different machine across a network. Client/server software usually hides the location of a server from clients by redirecting service requests. Clients should not have to be aware of the location of servers.
- Namespace transparency—clients should be able to use the same naming conventions (and namespace) to locate any server on the network.
- 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 the mixing of 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.

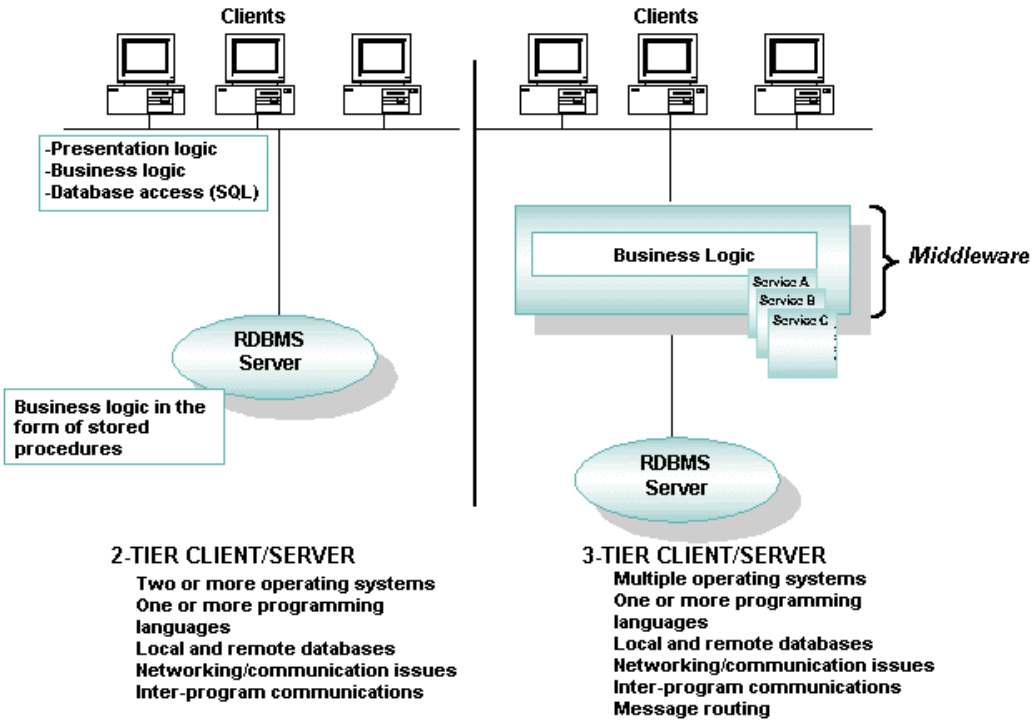
Differences Between 2-Tier and 3-Tier Client/Server Architectures

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 be part of the client program or part of the one or more server programs 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, as illustrated in the following figure.

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



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.

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. BEA recommends 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.

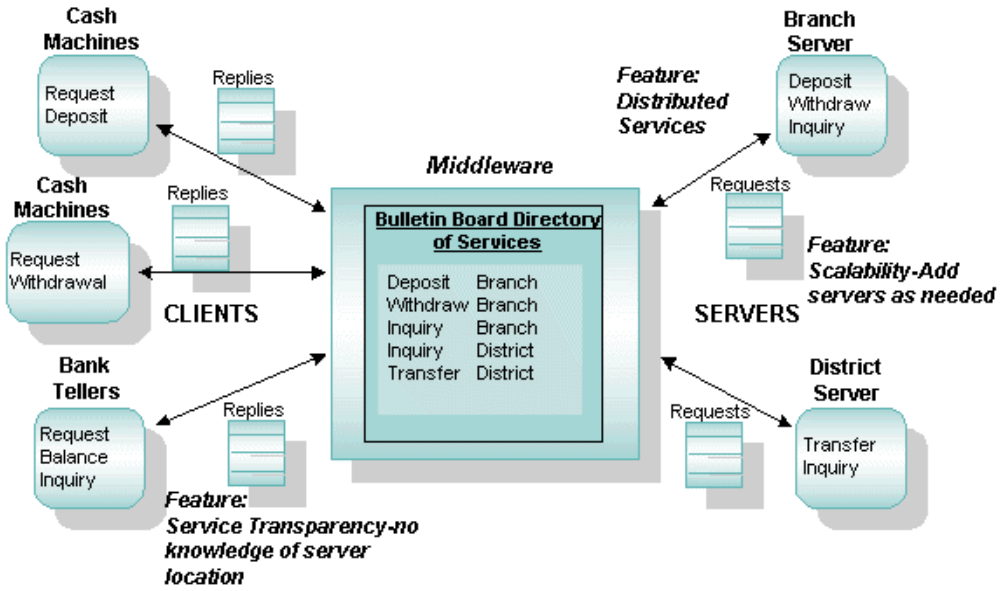
- Small businesses and corporate departments—a LAN-based single-server application is required. Users of this type of application include small businesses, such as a medical practice with several doctors, a multi-department corporation, or a bank with several branch offices. In this type of application, 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 corporate networks, intranets, and the Internet, 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 provides a *global directory advertising service*.

For example, in the following sample banking application, the bulletin board advertises deposit, withdrawal, and inquiry services. The BEA Tuxedo system then finds a server 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 sample banking application shows the primary building blocks of a BEA Tuxedo application:

- Clients—programs that collect input from users, send requests through the BEA Tuxedo system to servers, and deliver server replies to 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 (1) API functions used by the client—to issue requests and receive replies—and the server—to issue replies—and (2) messaging paradigms used to transmit client requests and server responses over a network. Middleware does not include the client user interface, the application logic, or the services provided by the servers.

In the 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.

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 the application. The client leaves the 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 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 multicontexted 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. The server is responsible for dispatching 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 send the client requests to a new service.

Application Processing Services Provided by the BEA Tuxedo System

The BEA Tuxedo system provides services that enable application developers to implement the following functionality in their applications:

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

For descriptions of the Tuxedo application processing services, see “BEA Tuxedo ATMI Architecture” on page 2-1.

Administrative Services Provided by the BEA Tuxedo System

The BEA Tuxedo system provides services that enable application administrators to perform the following administrative tasks:

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

- Message queuing management
- Event management

For descriptions of the Tuxedo system administration processes that provide the administrative services, see "BEA Tuxedo System Administration and Server Processes" on page 3-1 and "BEA Tuxedo Management Tools" on page 4-1. For detailed instructions on using the administrative services, see *Setting Up a BEA Tuxedo Application* and *Administering a BEA Tuxedo Application at Run Time*.

CHAPTER 2

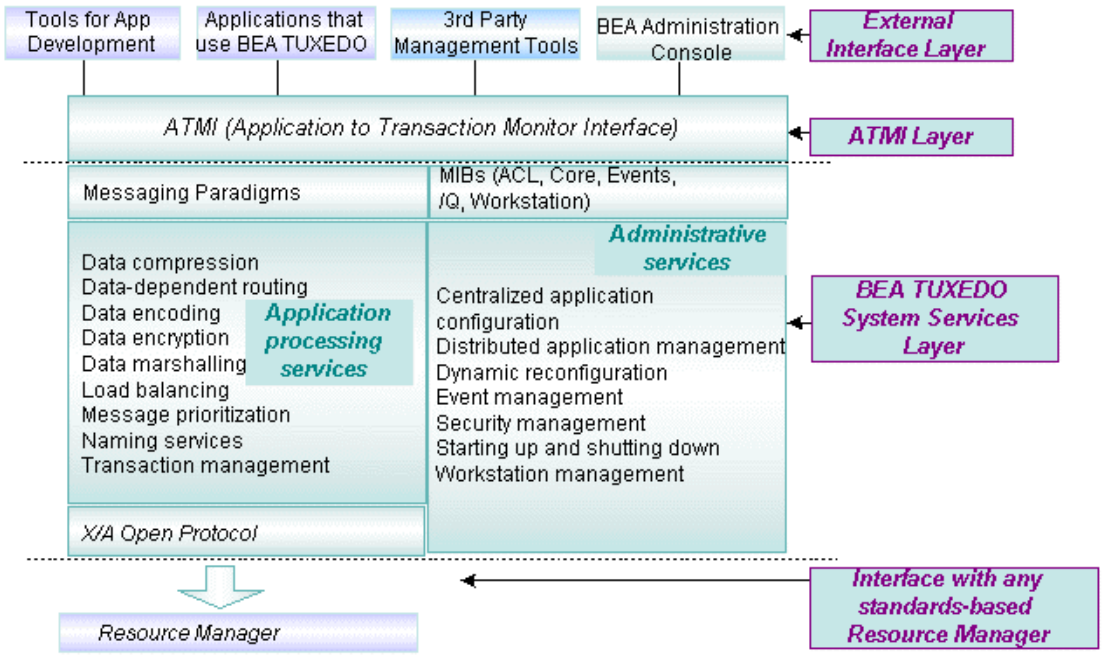
BEA Tuxedo ATMI Architecture

The following sections describe the basic architectural elements of a BEA Tuxedo ATMI environment:

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



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

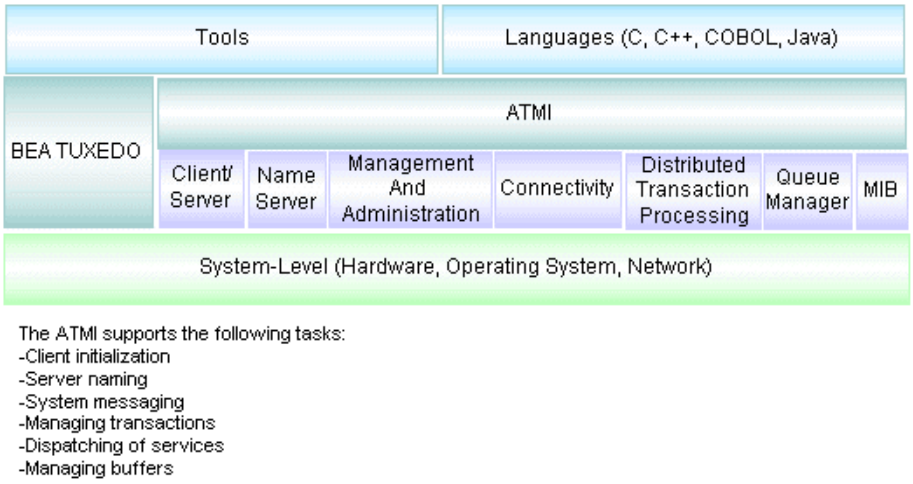
Architectural Part	Description
External interface layer	This layer consists of interfaces between the user and the environment. It includes both tools for application development and administration, such as the BEA Tuxedo Administration Console. The Administration Console can interact with standard management 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.
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. The BEA Tuxedo ATMI messaging paradigms include request/response, conversations, queuing, publish-and-subscribe, and unsolicited notification.
MIB—Management Information Base	The MIB is an interface that enables users to program and administer a BEA Tuxedo ATMI environment easily. MIB operations enable users to perform all management tasks (monitoring, configuring, tuning, and so on). The MIB allows users to perform one task to one object at a time or to build toolkits with which to batch tasks and/or objects. For information about the MIB and the MIB interface, see "BEA Tuxedo Management Tools" on page 4-1.

Architectural Part	Description (Continued)
BEA Tuxedo Services (application processing services and administrative services)	Services and/or capabilities provided by the BEA Tuxedo ATMI environment infrastructure for developing and administering applications. The application processing services available to BEA Tuxedo developers include data compression, data-dependent routing, data encoding, data encryption, data marshalling, load balancing, message prioritization, and service and event naming. These services are described in the discussions that follow. The administrative services available to BEA Tuxedo administrators include startup and shutdown of an application, centralized application configuration, distributed application management, dynamic application reconfiguration, workstation management, security management, transaction management, message queuing management, and event management. The system administration processes that provide the administrative services are described in "BEA Tuxedo System Administration and Server Processes" on page 3-1 and "BEA Tuxedo Management Tools" on page 4-1.
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 (distributed) transaction.

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 (routines, verbs) for defining and controlling global transactions in a BEA Tuxedo ATMI 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.

The ATMI functions knit together distributed programs by enabling them to send and receive data. All ATMI functions send or receive data in typed buffers.

The following table presents 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
Message queuing communications	tpenqueue(3c)	TPENQUEUE(3cbl)	Enqueue a message to a message queue
	tpdequeue(3c)	TPDEQUEUE(3cbl)	Dequeue a message from a message queue
Publish-and-subscribe 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 (see note at end of table)	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 the BEA Tuxedo ATMI transaction management functions is optional. Because BEA Tuxedo also supports the X/Open TX transaction management functions, you may want to use those functions for transaction management.

See Also

- "Using ATMI to Handle System and Application Errors" in *Administering a BEA Tuxedo Application at Run Time*

What Are the BEA Tuxedo ATMI Messaging Paradigms?

Besides managing an application's server processes and managing transactions, BEA Tuxedo ATMI also manages client/server communications, that is, allows clients (and servers) to invoke an application service using any of the messaging paradigms identified in the following table.

BEA Tuxedo ATMI Messaging Paradigm	Description
Request/response communication	A simple type of dialogue involving a single client request and a single response from the called request/response server. Request/response transactions usually involve people and thus require immediate attention; they run in high-priority mode.
Conversational communication	A state-preserving connection—context kept from message to message—between a client and the called conversational server. Conversational transactions also usually involve people and thus require immediate attention; they run in high-priority mode.
Message queuing communication	Time-independent communication among clients and servers. Queued transactions can run as high-priority or low priority messages. The BEA Tuxedo system includes its own bundled version of recoverable queues called */Q*.
Publish-and-subscribe communication	Asynchronous routing of *events* among the clients and servers in a BEA Tuxedo ATMI application. Publish-and-subscribe transactions usually run as high-priority messages. The BEA Tuxedo system has a transactional publish-and-subscribe system called *EventBroker*.
Unsolicited notification messaging	Communication from any client or server to any clients that were not requested or expected by those clients. Unsolicited notifications are handled by EventBroker.

Request/Response Communication

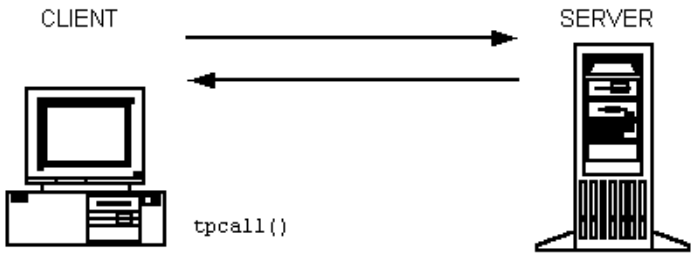
To implement request/response communication between ATMI clients and servers, the BEA Tuxedo system uses interprocess communication (IPC) message queues. Queues are the key to connectionless communication. Each server is assigned an IPC message queue called a request queue, and each client is assigned 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/response model is used for both synchronous and asynchronous service requests.

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.

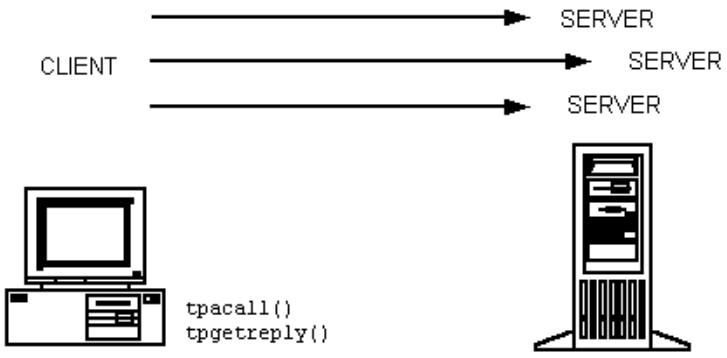
Figure 2-3 Synchronous Request/Response Communication



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 2-4 Asynchronous Request/Response Communication

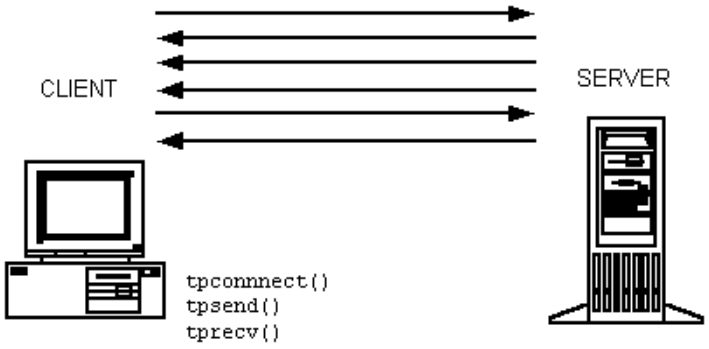


Conversational Communication

Conversational communication is the BEA Tuxedo system implementation of a human-like paradigm for exchanging messages between ATMI 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. By default, conversational servers are available, but more can be spawned automatically if needed.

The BEA Tuxedo system provides an 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.

Figure 2-5 Conversational Communication



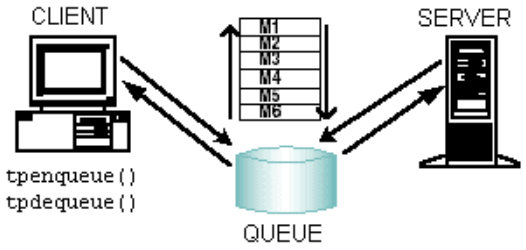
Conversations can be nested but performance may be degraded as a result of doing so. Conversations may contain either transactions or service requests as appropriate. Although a conversational service can make service calls and establish conversations, those service calls and conversations cannot be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

Message Queuing Communication

The BEA Tuxedo system offers a queue-based architecture known as */Q* for ATMI 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 *last in, first out* (LIFO) or *first in, first out* (FIFO), 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



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 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.

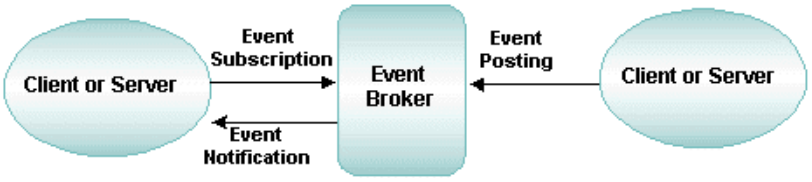
To use an application queue, your program must name the queue to be accessed and the queue space in which it resides. Your application can use more than one queue space and each space can contain more than one message queue.

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.

Publish-and-Subscribe Communication

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

Figure 2-7 Posting and Subscribing to an Event



Event generators (either clients or servers) inform EventBroker of changes and problems as they occur. This process is called posting an event. 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.

Types of Events Reported

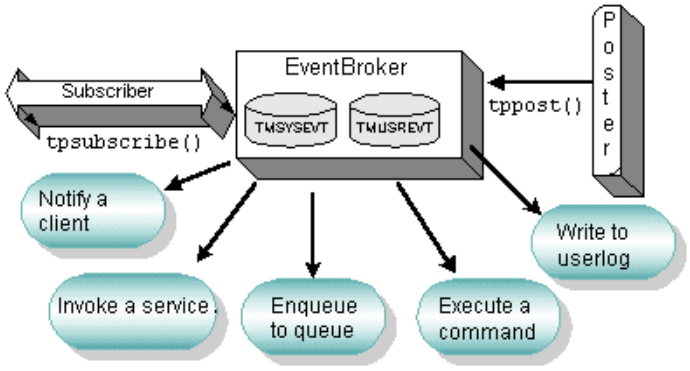
The BEA Tuxedo system supports two different types of event reports:

- Application-defined event reports—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.
- System event reports—provide details about BEA Tuxedo system events, such as server 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.

How Events Are Reported

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

Figure 2-8 Event-based Messaging



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

Unsolicited Communication

The BEA Tuxedo system offers a powerful communication paradigm called *unsolicited notification*. When unsolicited notification occurs, an ATMI client receives a message that it has never requested. This capability, which is managed by EventBroker, 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.

Figure 2-9 Unsolicited Notification Messaging

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.

See Also

- "Using the Request/Response Model (Synchronous Calls)" in *Tutorials for Developing BEA Tuxedo ATMI Applications*
- "Using Conversational Communication" in *Tutorials for Developing BEA Tuxedo ATMI Applications*
- "BEA Tuxedo Message Queuing Servers" on page 3-12
- "Administering Your Application Queues Using Command-Line Utilities" on page 4-10
- "Using Queue-based Communication" in *Tutorials for Developing BEA Tuxedo ATMI Applications*

- "BEA Tuxedo Publish-and-Subscribe Servers" on page 3-13
- "Managing Events Using EventBroker" on page 4-15
- "Using Event-based Communication" in *Tutorials for Developing BEA Tuxedo ATMI Applications*
- "Using Unsolicited Notification" in *Tutorials for Developing BEA Tuxedo ATMI Applications*

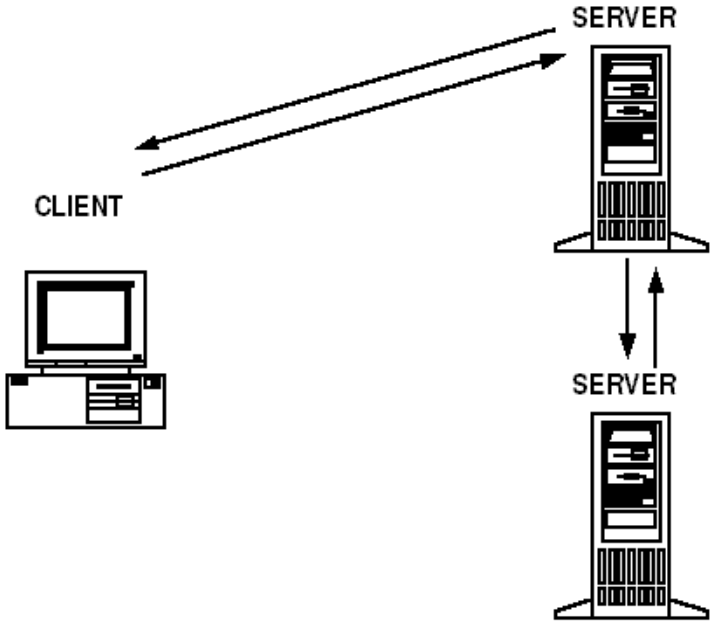
What Are Nested and Forwarded Requests?

Nested and forwarded service requests allow BEA Tuxedo services to act as ATMI clients and call other services.

Nested Requests

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

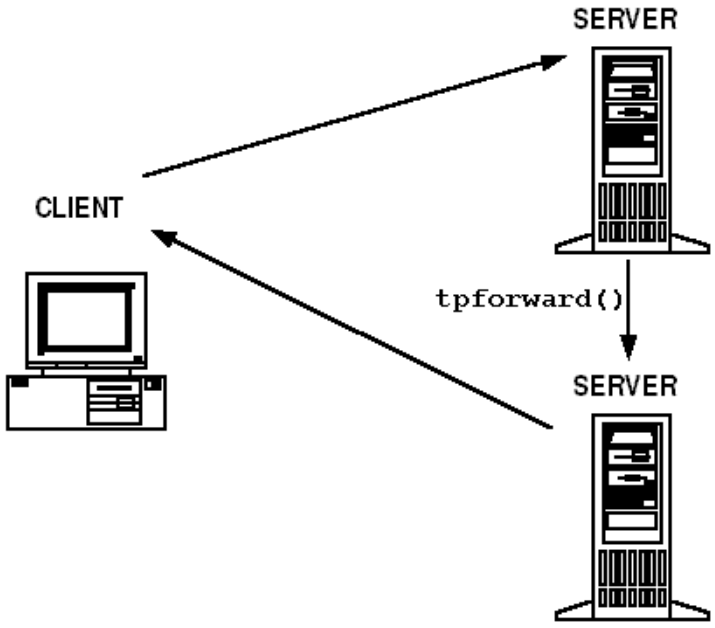
A customer uses a cash machine to transfer money from his or her savings account to her checking account. A BEA Tuxedo application performs the work necessary to transfer the money. First, on behalf of the customer, the client issues a request for a service called `TRANSFER`, and the request is placed 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

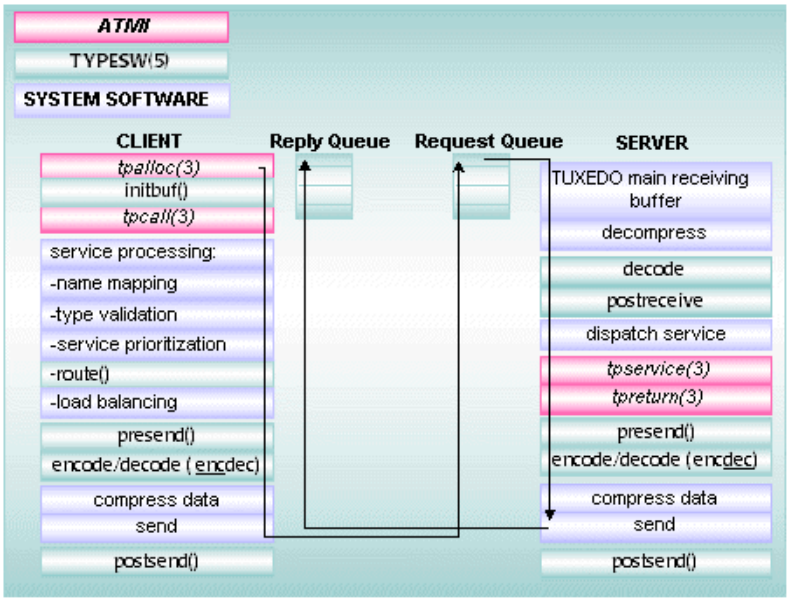
- "Using Nested Calls" in *Tutorials for Developing BEA Tuxedo ATMI Applications*

- "Using Forwarded Calls" in *Tutorials for Developing BEA Tuxedo ATMI Applications*

How Does BEA Tuxedo Process Messages?

All communication within the BEA Tuxedo ATMI environment is accomplished by transferring messages. The BEA Tuxedo system passes service request messages between ATMI clients and servers through operating system (OS) interprocess communications (IPC) message queues. System messages and data are passed between OS-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 *data-dependent routing*, which is an automatic routing option to map messages that meet specific criteria (message value) to a specific server. 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 Field Manipulation Language (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.

Benefits of Service Request Processing

The benefits of service request processing include:

- 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 2-9
- "What Are Typed Buffers?" on page 2-22

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), 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), 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
- MBSTRING

You assign buffer types in the `ENVFILE` parameter defined in the `MACHINES` section of the Tuxedo (`UBBCONFIG`) configuration file. Assigning or overriding them in the `ENVFILE` parameter in the `SERVERS` section of the Tuxedo 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 *BEA Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*.

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. For descriptions of the different types of BEA Tuxedo buffers, see "Managing Typed Buffers" in *Programming a BEA Tuxedo ATMI Application Using C*.

See Also

- `tuxtypes(5)`, `typesw(5)`, and `UBBCONFIG(5)` in *BEA Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*.

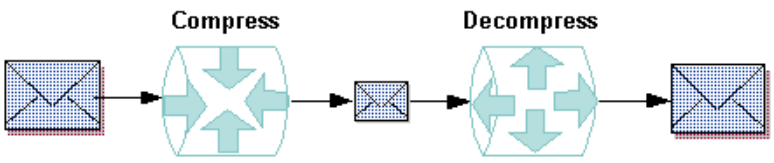
What Is Data Compression?

Data compression is the process of shrinking an application buffer so that 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 (the dynamic part of the MIB). 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 applications

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.

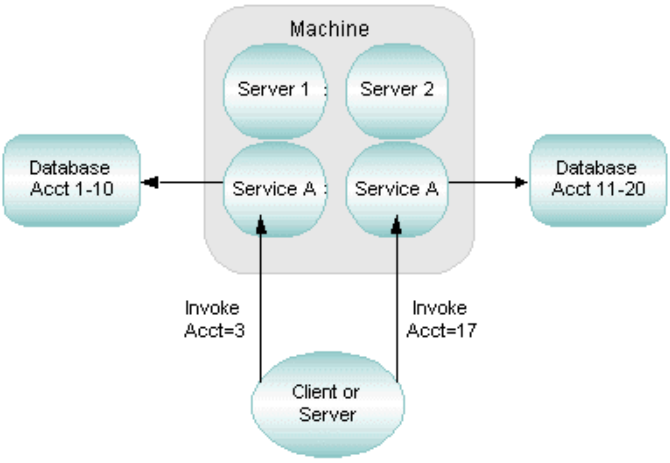
Example of Data-Dependent Routing with a Horizontally Partitioned Database

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:

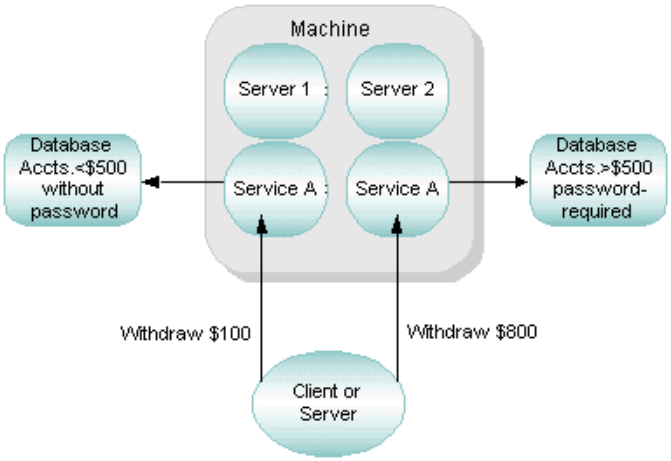
- Customers can withdraw up to $500 without entering a special password.
- Customers must enter a special password to withdraw more than $500.

Two clients issue withdrawal requests: one for $100 and one for $800. If data-dependent routing is enabled to support the withdrawal rules, 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.

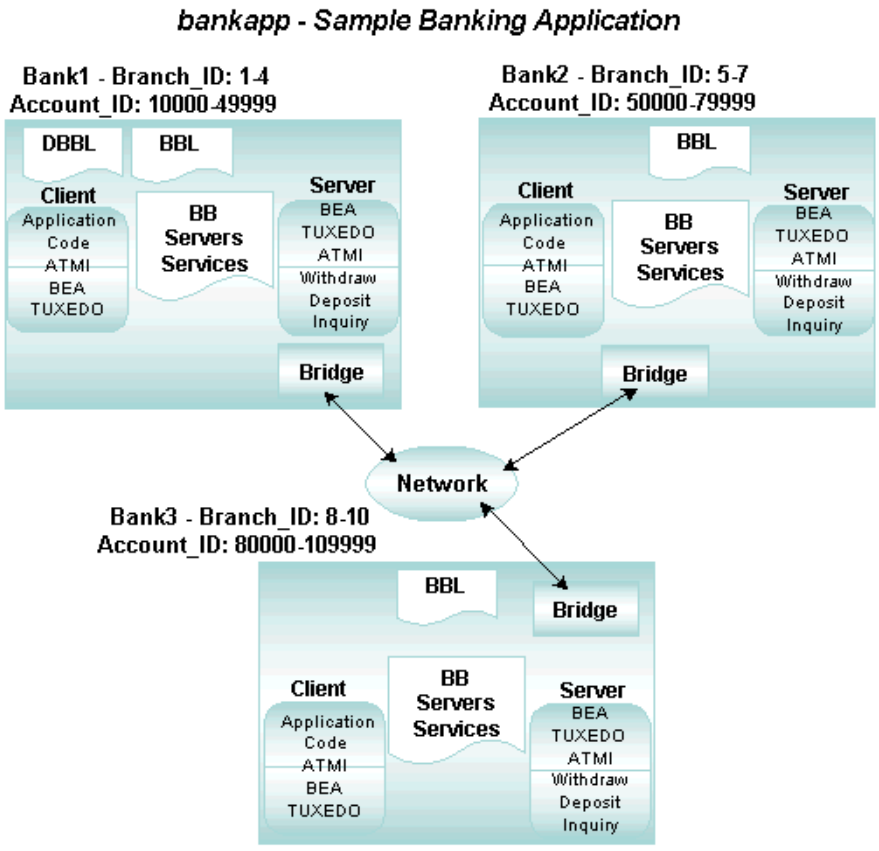
Figure 2-15 Data-Dependent Routing with Rule-Based Servers



Example of Data-Dependent Routing with a Distributed Application

The following diagram shows how client requests are routed to servers in a distributed application. 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 figure, 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 encodes and decodes data to a machine-independent representation for transmission to other machines involved in a BEA Tuxedo application.

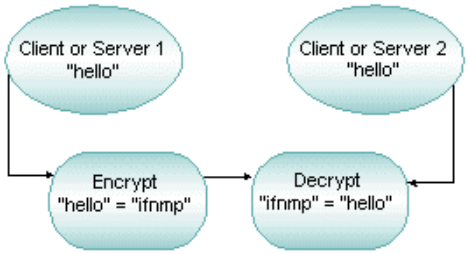
The BEA Tuxedo system employs, by default, the External Data Representation (XDR) algorithm, which can be customized by replacing the BEA Tuxedo system functions with user-written functions. Encoding and decoding are used only between machines and only when a remote machine uses a data representation 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 all users except the user for which the message is intended. 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 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. Load balancing avoids overburdening some servers while leaving others idle or infrequently used. Before sending a request to a service routine, the BEA Tuxedo 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.

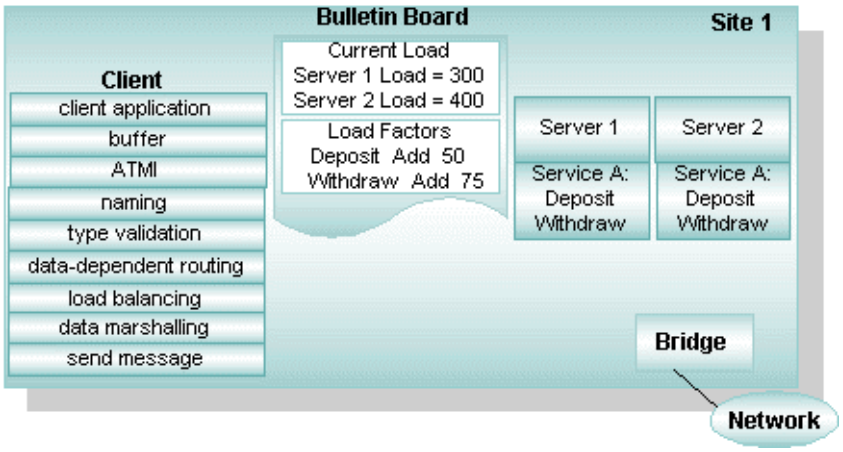
Load refers to a number assigned to a service request based on the amount of time required to execute that service. Loads are assigned to services so that the BEA Tuxedo system can understand the relationship between requests. To keep track of the amount of work, or total load, being performed by each 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 a

Multiple Server, Single Queue (MSSQ) 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 2-18 Load Balancing

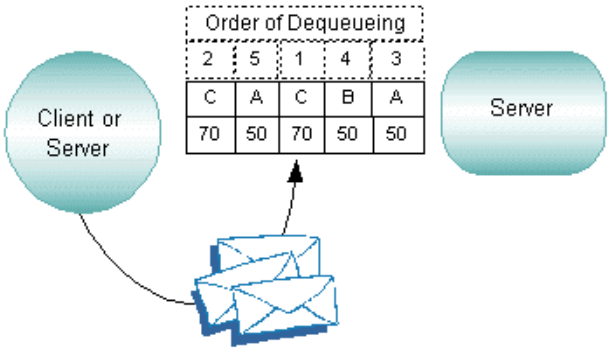


What Is Message Prioritization?

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. After 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 2-19 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 *first in, first out* (FIFO) 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.

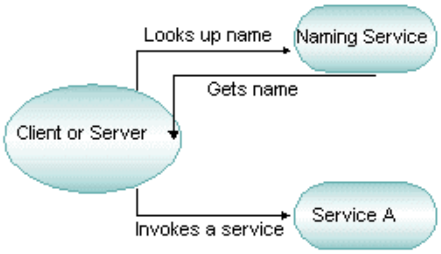
Naming Services

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 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 2-20 Locating a Service by Name



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

- "Publish-and-Subscribe Communication" on page 2-13

CHAPTER 3

BEA Tuxedo System Administration and Server Processes

The following sections describe the core BEA Tuxedo system administration and server processes that together form the infrastructure for ATMI applications built on the BEA Tuxedo system:

- BEA Tuxedo ATMI Infrastructure
- BEA Tuxedo Administration Processes
- BEA Tuxedo Workstation Servers
- BEA Tuxedo Authentication Server
- BEA Tuxedo Transaction Management Server
- BEA Tuxedo Message Queuing Servers
- BEA Tuxedo Publish-and-Subscribe Servers
- BEA Tuxedo Domains (Multiple-Domain) Servers
- System Services Available to Different Types of BEA Tuxedo Configurations

BEA Tuxedo ATMI Infrastructure

The following categories of BEA Tuxedo system processes provide an infrastructure for the efficient routing, dispatching, and management of application service requests, application queues, and event postings and notifications for ATMI applications:

- BEA Tuxedo administration processes
- BEA Tuxedo Workstation server processes
- BEA Tuxedo authentication server process
- BEA Tuxedo transaction management server process
- BEA Tuxedo message queuing server processes
- BEA Tuxedo publish-and-subscribe server processes
- BEA Tuxedo Domains (multiple-domain) server processes

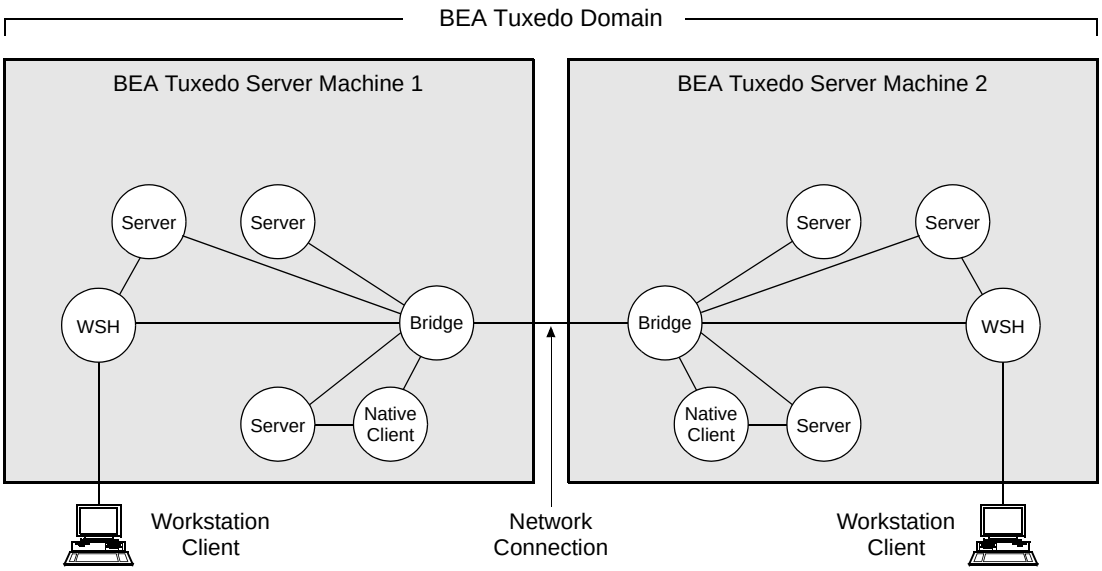
Before exploring these categories of BEA Tuxedo system processes, you should have a good understanding of the following important BEA Tuxedo terms and concepts.

Tuxedo Domain

A BEA Tuxedo domain, also known as a BEA Tuxedo application, is a set of Tuxedo system, client, and server processes administered as a single unit from a single Tuxedo configuration file. A Tuxedo domain consists of many system processes, one or more application client processes, one or more application server processes, and one or more computer machines connected over a network.

The following figure presents a high-level view of a BEA Tuxedo domain.

Figure 3-1 High-Level View of a BEA Tuxedo Domain



In BEA Tuxedo terminology, a *domain* is the same as an *application*—a business application; both terms are used as synonyms throughout the BEA Tuxedo user documentation. Examples of business applications currently running on Tuxedo are airline and hotel reservation systems, credit authorization systems, stock-brokerage systems, banking systems, and automatic teller machines.

The application processes running on the client side of a Tuxedo client/server application are usually referred to as *application clients* or simply *clients*. The application processes running on the server side of a Tuxedo client/server application are usually referred to as *application servers*.

Note: Often, the term *domain*, or *application*, is intended to mean the server-side software of a BEA Tuxedo client/server application.

Tuxedo Configuration File

Each BEA Tuxedo domain is controlled by a configuration file in which installation-dependent parameters are defined. The text version of the configuration file is referred to as `UBBCONFIG`, although the configuration file may have any name, as long as the content of the file conforms to the format described on reference page `UBBCONFIG(5)` in *BEA Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*. Typical configuration filenames begin with the string `ubb`, followed by a mnemonic string, such as `simple` in the filename `ubbsimple`.

The UBBCONFIG file for a Tuxedo domain contains all the information necessary to boot the application, such as lists of its resources, machines, groups, servers, available services, and so on. It consists of nine sections, five of which are required for all configurations: RESOURCES, MACHINES, GROUPS, SERVERS, and SERVICES.

The binary version of the UBBCONFIG file is referred to as TUXCONFIG. As with UBBCONFIG, the TUXCONFIG file may be given any name; the actual name is the device or system filename specified in the TUXCONFIG environment variable.

Tuxedo Master Machine

The master machine, or master node, for a BEA Tuxedo domain is a server machine containing the domain's UBBCONFIG file, and is designated as the master machine in the RESOURCES section of the UBBCONFIG file. Starting, stopping, and administering the one or more server machines in a Tuxedo domain is done through the master machine.

The master machine for a Tuxedo domain also contains the master copy of the TUXCONFIG file. Copies of the TUXCONFIG file are propagated to every other server machine—referred to as *non-master machines*—in a Tuxedo domain whenever the Tuxedo system is booted on the master machine.

In a multiple-machine domain running different releases of the BEA Tuxedo system software, the master machine must run the highest release of the Tuxedo system software in the domain.

Note: Tuxedo 9.1 is a Tuxedo 9.0 *minor release*. Therefore, Tuxedo 9.1 *can* operate as a non-master machine to a Tuxedo 9.0 master machine in a multiple platform environment.

Tuxedo TUXCONFIG Environment Variable

The TUXCONFIG environment variable defines the location on the master machine where the tmloadcf(1) command loads the binary TUXCONFIG file. It must be set to an absolute pathname ending with the device or system filename where TUXCONFIG is to be loaded.

The TUXCONFIG pathname value is designated in the MACHINES section of the UBBCONFIG file. It is specified for the master machine *and* for every other server machine in the Tuxedo domain. When copies of the binary TUXCONFIG file are propagated to non-master machines during system boot, the copies are stored on the non-master machines in accordance to the TUXCONFIG pathname values.

Tuxedo TUXDIR Environment Variable

The `TUXDIR` environment variable defines the installation directory of the BEA Tuxedo system software on the master machine. It must be set to an absolute pathname ending with the name of the installation directory.

The `TUXDIR` pathname value is designated in the `MACHINES` section of the `UBBCONFIG` file. It is specified for the master machine *and* for every other server machine in the Tuxedo domain.

Tuxedo Bulletin Board

The BEA Tuxedo system uses the `TUXCONFIG` file to set up a *bulletin board* (BB) on each server machine in a Tuxedo domain. When a Tuxedo server process becomes active, it advertises the names of its services in the bulletin board. Some information in the bulletin board is global and is replicated on every server machine in the Tuxedo domain (for example, the names and locations of all servers offering a particular service). Other information is local and is visible only on the local bulletin board (for example, the actual number and type of client requests currently waiting on a local server request queue).

The bulletin board provides location and namespace transparency within a Tuxedo domain. Location transparency means that Tuxedo client and server processes do not have to be aware of the location of a resource within the Tuxedo domain. Namespace transparency means that Tuxedo client and server processes can use the same naming conventions (and namespace) to locate any resource in the Tuxedo domain.

See Also

- "How to Create a Configuration File" in *Setting Up a BEA Tuxedo Application*
- "Creating the Configuration File for a Distributed ATMI Application" in *Setting Up a BEA Tuxedo Application*

BEA Tuxedo Administration Processes

The BEA Tuxedo administration processes automate most of the management tasks for a distributed application, including:

- Starting up and shutting down an application
- Dynamically reconfiguring an application

This discussion focuses only on the administration processes that set up and manage the bulletin board in a BEA Tuxedo single-machine or multiple-machine application (domain):

- Single-machine application—one or more local or remote application clients communicating with one or more application servers that reside on the same server machine and belong to a Tuxedo domain.
- Multiple-machine application—one or more local or remote application clients communicating with one or more application servers that reside on multiple server machines and belong to a Tuxedo domain. BEA Tuxedo Bridge processes send and receive service requests between the server machines, and route requests to locally running system or application server processes.

For a description of the administration processes used to start up, shut down, and dynamically reconfigure a Tuxedo application, see "BEA Tuxedo Management Tools" on page 4-1.

What Is the Role of the Bulletin Board?

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

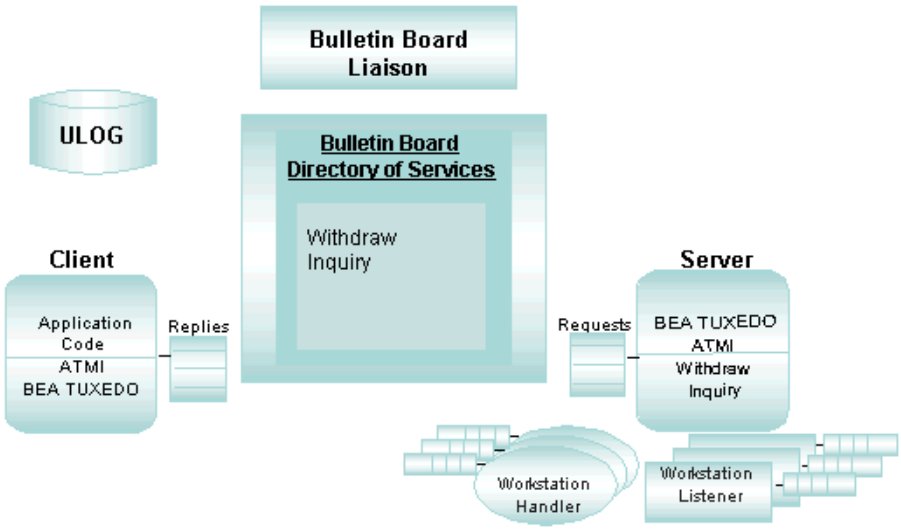
- Assigns service requests to specific servers. When a service is called, the bulletin board looks up servers that offer the requested service. Based on this information, and any data-dependent routing criteria, the bulletin board places the request data on the request queue of a valid server.
- Maintains 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.

Each server machine in a Tuxedo application contains a bulletin board.

What Is the Role of the Bulletin Board Liaison?

The Bulletin Board Liaison (BBL) is a BEA Tuxedo administration process running on each server machine in a Tuxedo application that coordinates changes to the local bulletin board and verifies the sanity of the software programs that are active on the local machine. There is one and only one BBL running on each server machine—including the master machine—in a Tuxedo domain.

Figure 3-2 Bulletin Board and Bulletin Board Liaison



What Is the Distinguished Bulletin Board Liaison (DBBL)?

The Distinguished Bulletin Board Liaison (DBBL) is the BEA Tuxedo administration process that makes it possible to distribute an application across multiple server machines. The DBBL ensures that the Bulletin Board Liaison (BBL) server on each server 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 server machine in the configuration. Servers located on remote machines are accessed through the Bridge process running 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 process.

Note: A Tuxedo single-machine application may or may not have a DBBL process running, depending on the value of the `MODEL` parameter in the `RESOURCES` section of the `UBBCONFIG` file. If `MODEL=SHM`, no DBBL process is running; if `MODEL=MP`, a DBBL process *and* a Bridge process are running. The advantage of having a DBBL is that it periodically checks the health of the BBL and restarts it if it terminates. The disadvantage is that two additional system processes are running: the DBBL and the Bridge.

See Also

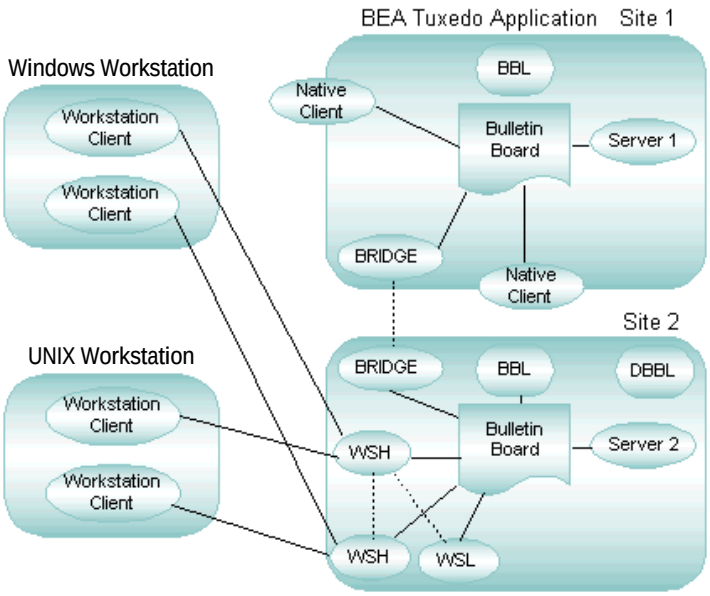
- "Distributing ATMI Applications Across a Network" in *Setting Up a BEA Tuxedo Application*
- "Setting Up the Network for a Distributed Application" in *Setting Up a BEA Tuxedo Application*
- "Managing the Network in a Distributed Application" in *Administering a BEA Tuxedo Application at Run Time*

BEA Tuxedo Workstation Servers

The BEA Tuxedo Workstation server processes allow Workstation clients—remote ATMI clients—to reside on a remote machine that does not have a full BEA Tuxedo server-side installation, that is, a machine that does not support BEA Tuxedo administration servers or a bulletin board. All communication between a Workstation client and the BEA Tuxedo server application takes place over the network.

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 pair of Workstation Listener (WSL) and Workstation Handler (WSH) server processes running in a BEA Tuxedo application that supports all the BEA Tuxedo system software, including ATMI functions and networking software. The following figure shows how the WSL and WSH processes connect Workstation clients to the BEA Tuxedo server application.

Figure 3-3 Handling Workstation Clients



What is the Role of the Workstation Listener?

The Workstation Listener (WSL) is a BEA Tuxedo listening process, running on a BEA Tuxedo server machine, that accepts connection requests from Workstation clients and assigns connections to a Workstation Handler also running on the server machine. It also manages the pool of Workstation Handler processes, starting and stopping them in response to load demands.

An administrator can define several WSLs in a Tuxedo domain to distribute and balance the workstation communication load across multiple server machines.

What is the Role of the Workstation Handler?

The Workstation Handler (WSH) is a BEA Tuxedo gateway process, running on the BEA Tuxedo server machine, that handles communications between Workstation clients and the BEA Tuxedo server application. A WSH process resides within the administrative domain of the application and is registered in the local BEA Tuxedo bulletin board as a client.

Each WSH process can manage multiple Workstation clients. A WSH multiplexes all requests and replies with a particular Workstation client over a single connection.

See Also

- *Using the BEA Tuxedo ATMI Workstation Component*
- "Administering Security" in *Using Security in ATMI Applications*
- `UBBCONFIG(5)`, `WS_MIB(5)`, and `WSL(5)` in *BEA Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*

BEA Tuxedo Authentication Server

The BEA Tuxedo authentication server, named `AUTHSRV`, allows system administrators to configure the additional security needed to authenticate and authorize Workstation clients. `AUTHSVR` provides a single service, which verifies whether the user has the correct authentication level.

Administrators can configure BEA Tuxedo applications with incremental levels of authentication and authorization. Administrators can configure an application so that all servers except `AUTHSVR` have restricted access to shared resources, such as shared memory and message queues.

Application designers can replace `AUTHSVR` with an authentication server that implements logic specific to their application. For example, a company may want to develop a custom authentication server so that it can use the popular Kerberos mechanism for authentication.

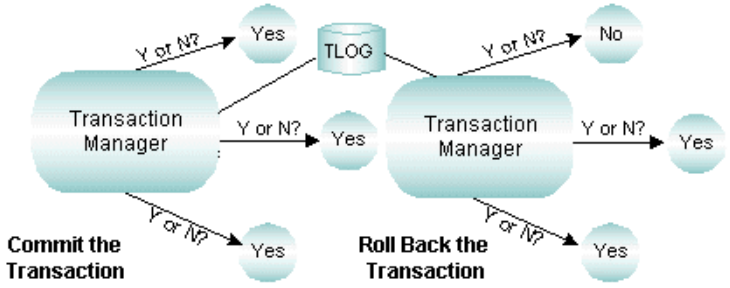
See Also

- "Administering Security" in *Using Security in ATMI Applications*
- `AUTHSVR(5)` in *BEA Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*

BEA Tuxedo Transaction Management Server

The BEA Tuxedo transaction management server, named `TMS`, is responsible for coordinating global transactions, on behalf of BEA Tuxedo ATMI applications, from their point of origin—typically on the client—across one or more server machines, and then back to the originating client. `TMS` tracks transaction participants and supervises a two-phase commit protocol, ensuring that transaction commit and rollback are properly handled at each site.

Figure 3-4 Transaction Manager Servers at Work



Coordinating Operations

To coordinate all the performed operations and all the modules affected by a transaction, TMS directs the actions of one or more resource managers, such as relational databases, hierarchical databases, filesystems, document stores, message queues, and other back-end services. Together, TMS and the resource managers maintain the atomicity of a transaction, but it is TMS that actually manages the two-phase commit protocol and the recovery (if needed) for the transaction.

Tracking Participants with a Transaction Log

In the transaction log (TLOG), TMS logs a global transaction only after receiving all "yes" replies from the global transaction participants at the end of the first phase of a two-phase commit. A TLOG record indicates that a global transaction should be committed; no TLOG record indicates that the transaction should be rolled back. Each server machine in a Tuxedo domain should have its own TLOG.

See Also

- "Configuring Your ATMI Application to Use Transactions" in *Setting Up a BEA Tuxedo Application*
- "Using Transactions" in *Tutorials for Developing BEA Tuxedo ATMI Applications*
- TM_MIB(5) in *BEA Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*

BEA Tuxedo Message Queuing Servers

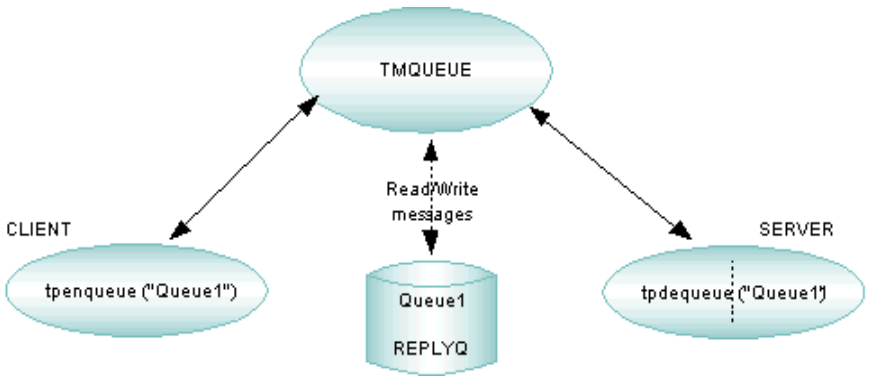
The BEA Tuxedo message queuing servers provide for time-independent communication among clients and servers in a BEA Tuxedo ATMI application. They make it possible for an application, within a global transaction, to store client and server generated messages to stable storage for processing later. A client or server process involved in message queuing communications decides when it wants to retrieve a message off its queue.

The BEA Tuxedo message queuing servers consist of a "message queue manager" server named `TMQUEUE` and a "message forwarding" server named `TMQFORWARD`.

What is the Role of the TMQUEUE Server?

The `TMQUEUE` server stores (enqueues) and retrieves (dequeues) messages on behalf of clients and servers. The following figure shows how `TMQUEUE` works.

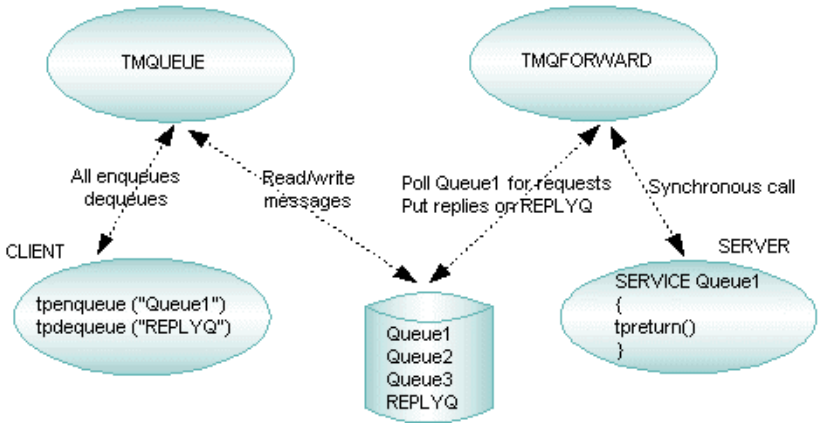
Figure 3-5 Queuing Messages Using TMQUEUE



What is the Role of the TMQFORWARD Server?

The `TMQFORWARD` server dequeues messages and forwards them to the appropriate servers for processing. `TMQFORWARD` is needed only if queued messages require a service call. For example, a queue may be used (on a BEA Tuxedo client or server) for interprocess communication in which one process places the message on the queue and another removes it. The following figure shows how `TMQFORWARD` works.

Figure 3-6 Storing and Forwarding Messages Using TMQFORWARD



See Also

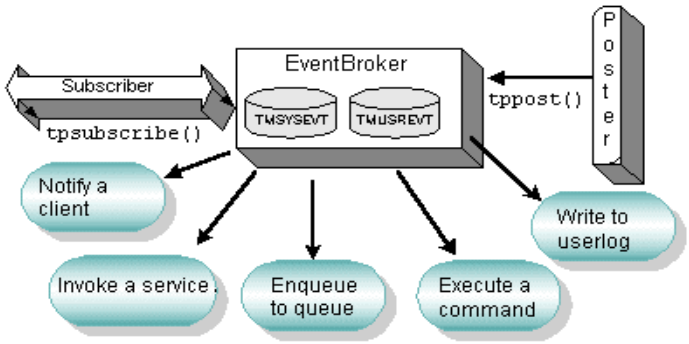
- "Administering Your Application Queues Using Command-Line Utilities" on page 4-10
- *Using the ATMI /Q Component*
- `tpenqueue(3c)` and `tpdequeue(3c)` in *BEA Tuxedo ATMI C Function Reference*
- `APPQ_MIB(5)`, `TMQUEUE(5)`, `TMQFORWARD(5)`, and `UBBCONFIG(5)` in *BEA Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*

BEA Tuxedo Publish-and-Subscribe Servers

The BEA Tuxedo publish-and-subscribe servers provides asynchronous routing of application and system events among the processes running in a BEA Tuxedo ATMI application. An event is a state change or other occurrence in an application program or the BEA Tuxedo system that may be of interest to an administrator, an operator, or the software. Examples of events are "a stock traded at or above a specified price" or "a network failure occurred."

The BEA Tuxedo publish-and-subscribe servers consist of an "application event" server named `TMUSREVT` and a "system event" server named `TMSYSEVT`. The `TMUSREVT` server handles *application events* on behalf of clients and servers, and the `TMSYSEVT` server handles *system events* on behalf of clients and servers. The following figure shows how `TMUSREVT` and `TMSYSEVT` work.

Figure 3-7 Handling Events Using TMUSREVT and TMSYSEVT



See Also

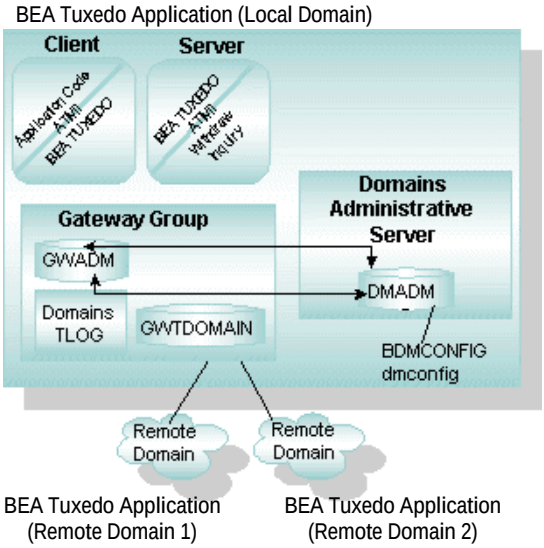
- "Managing Events Using EventBroker" on page 4-15
- "About the EventBroker" in *Administering a BEA Tuxedo Application at Run Time*
- `tppost(3c)`, `tpsubscribe(3c)`, and `tpunsubscribe(3c)`in *BEA Tuxedo ATMI C Function Reference*
- `EVENTS(5)`, `EVENT_MIB(5)`, `TMSYSEVT(5)`, `TMUSREVT(5)`, and `UBBCONFIG(5)` in *BEA Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*

BEA Tuxedo Domains (Multiple-Domain) Servers

The BEA Tuxedo Domains (multiple-domain) server processes extend the BEA Tuxedo system client/server model to provide transaction interoperability across transaction processing (TP) domains. This extension preserves the model and the ATMI interface by making access to services on the remote domain (or accepting service requests from a remote domain) transparent to both the application programmer and the end-user.

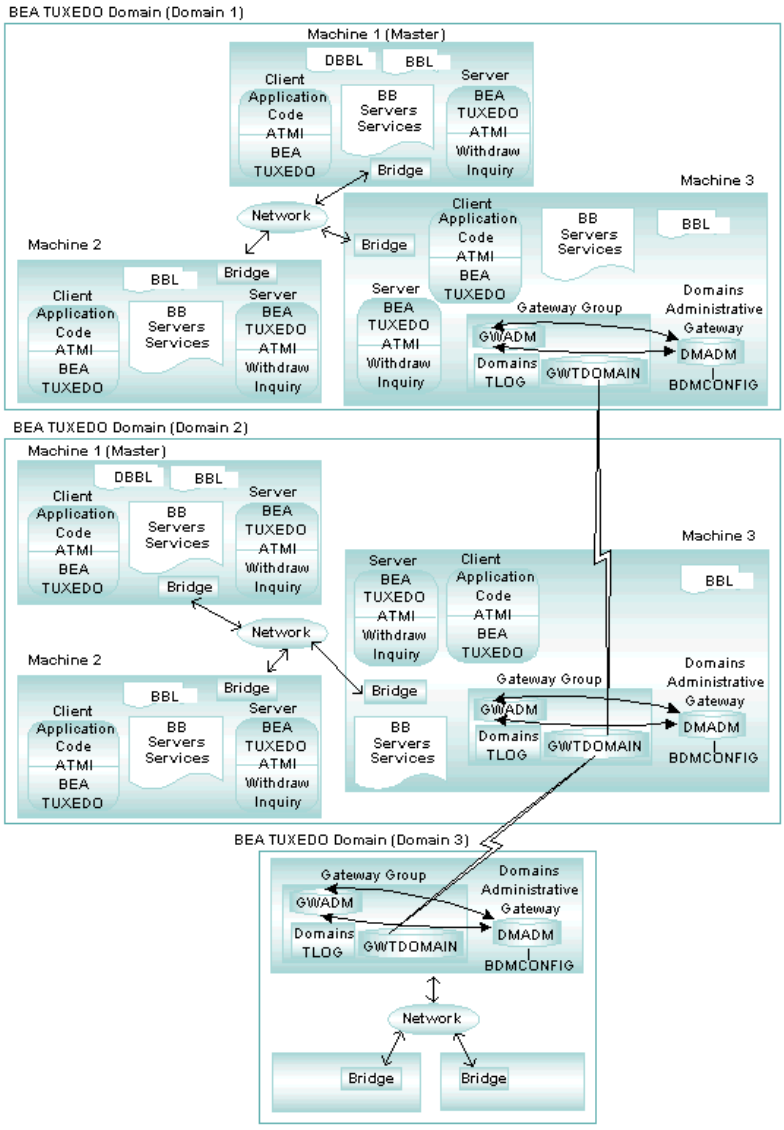
The BEA Tuxedo Domains server processes consist of a "Domains administrative" server named `DMADM`, a "gateway administrative" server named `GWADM`, and one of several types of "domain gateway" servers—for example, the TDomain gateway server, implemented by the `GWTDOMAIN` process. The following figure shows how `DMADM`, `GWADM`, and `GWTDOMAIN` work.

Figure 3-8 Interdomain Communication Using the TDomain Gateway Group



Here is another figure demonstrating the connectivity in a Domains configuration.

Figure 3-9 Domains Configuration



What is the Role of the DMADM Server?

The DMADM server 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 Domains configuration file (BDMCONFIG).

How multiple domains are connected and which services they make accessible to one another are defined in Domains configuration files, the text and binary versions of which are known as DMCONFIG and BDMCONFIG, respectively. Each BEA Tuxedo domain involved in a Domains configuration requires its own Domains configuration file.

What is the Role of the GWADM Server?

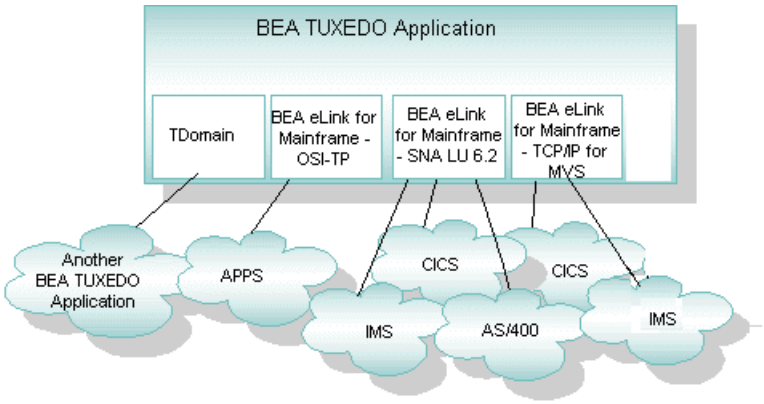
The GWADM server registers with the DMADM server to obtain the configuration information used by the corresponding gateway group. GWADM accepts requests from DMADM for run-time statistics or changes in the run-time options of the specified gateway group.

What is the Role of the Domain Gateway Servers?

Domain gateways are highly asynchronous, multitasking server processes that handle outgoing and incoming service requests to or from remote domains. They make access to services across domains transparent to both the application programmer and the application user.

As shown in the following figure, the BEA Tuxedo system supports several types of domain gateways, to allow a BEA Tuxedo application to communicate with other BEA Tuxedo applications or with applications running on other TP systems.

Figure 3-10 Domain Gateway Types



See Also

- "Administering Your Domains Application Using Command-Line Utilities" on page 4-11
- *Using the BEA Tuxedo Domains Component*
- `DMADM(5)`, `DMCONFIG(5)`, `GWADM(5)`, `GWTDOMAIN(5)`, and `UBBCONFIG(5)` in *BEA Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*
- *BEA eLink Documentation* at `http://e-docs.bea.com/elink/mainfram/mainfram.htm`

System Services Available to Different Types of BEA Tuxedo Configurations

The following table lists the BEA Tuxedo system services available for a BEA Tuxedo single-machine, multiple-machine (distributed), and Domains application. The single-machine and multiple-machine applications are BEA Tuxedo domain configurations. The Domains application is a BEA Tuxedo Domains configuration consisting of two or more BEA Tuxedo domains communicating with one another via TDomain (`GWTDOMAIN`) gateways.

Table 3-1 Capabilities Available in Different Types of BEA Tuxedo Configurations

Available Capability	Single-Machine Application	Multiple-Machine Application	Domains Application
ATMI	X	X	X
Messaging paradigms	X	X	X
Administration parts:			
`UBBCONFIG`, `TUXCONFIG`,	X	X	X
Bulletin Board (BB),	X	X	X
Bulletin Board Liaison (BBL),	X	X	X
Distinguished Bulletin Board Liaison (DBBL),	See Note at end of table	X	X
ULOG, TLOG,	X	X	X
Bridges		X	X
Administrative processes:			
`tmloadcf`, `tmunloadcf`,	X	X	X
`tmboot`, `tmadmin`, ...	X	X	X
For an overview of BEA Tuxedo administrative processes, see "Managing Operations Using Command-Line Utilities" on page 4-9.			
Domains parts:			
`DMCONFIG`, `BDMCONFIG`,			X
`DMADM`, `GWADM`, `GWTDOMAIN`,			X
`DMTLOG`			X
Domains administrative processes:			
`dmloadcf`, `dmunloadcf`,			X
`dmadmin`			X
For an overview of the BEA Tuxedo Domains administrative processes, see "Administering Your Domains Application Using Command-Line Utilities" on page 4-11.			
Application processes: clients, servers, and services	X	X	X

Table 3-1 Capabilities Available in Different Types of BEA Tuxedo Configurations (Continued)

Available Capability	Single-Machine Application	Multiple-Machine Application	Domains Application
Workstation client management	X	X	X
Security management	X	X	X
Transaction management	X	X	X
Message queuing management	X	X	X
Event management	X	X	

Note: A Tuxedo single-machine application may or may not have a DBBL process running, depending on the value of the `MODEL` parameter in the `RESOURCES` section of the `UBBCONFIG` file. If `MODEL=SHM`, no DBBL process is running; if `MODEL=MP`, a DBBL process *and* a Bridge process are running. The advantage of having a DBBL is that it periodically checks the health of the BBL and restarts it if it terminates. The disadvantage is that two additional system processes are running: the DBBL and the Bridge.

CHAPTER 4

BEA Tuxedo Management Tools

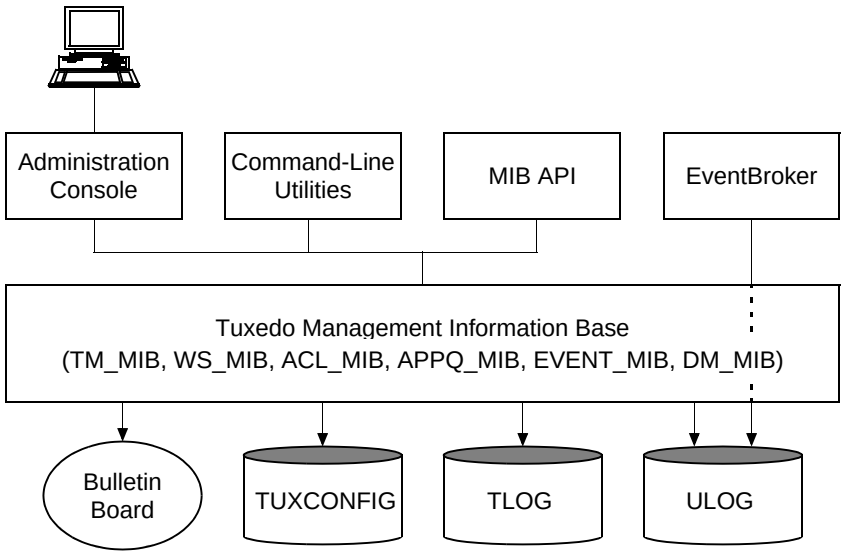
The following sections describe the BEA Tuxedo administration processes available to users for managing Tuxedo applications:

- BEA Tuxedo Tool Architecture
- Management Operations Using the BEA Tuxedo Administration Console
- Exploring the Main Menu of the BEA Tuxedo Administration Console
- Managing Operations Using the MIB
- Managing Operations Using Command-Line Utilities
- Managing Events Using EventBroker

BEA Tuxedo Tool Architecture

As shown in the following figure, the BEA Tuxedo administration processes used to manage a Tuxedo application encompass a variety of tools constructed around the BEA Tuxedo management information base (MIB).

Figure 4-1 Tools to Administer Your BEA Tuxedo Application



The BEA Tuxedo MIB contains all the information necessary for the operation of a Tuxedo application. It contains the TM_MIB, which is common to all applications, and the following component MIBs, each of which describes a subsystem of the BEA Tuxedo system:

- WS_MIB—used to manage Workstation groups and processes associated with them
- ACL_MIB—used to administer access control lists (ACLs)
- APPQ_MIB—used to administer application stable-storage queues
- EVENT_MIB—used to control event notification and the subscription request database
- DM_MIB—used to administer a Tuxedo Domains (multiple-domain) configuration

The MIB reference pages (`TM_MIB(5)`, generic reference page `MIB(5)`, ...) are defined in *BEA Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*.

Tool Interfaces with the MIB

The BEA Tuxedo administration tools, briefly described in the following list, provide different types of interfaces to the MIB:

- BEA Tuxedo Administration Console—a Web-based GUI application used to monitor a Tuxedo application and to dynamically configure it.
- Command-line utilities—a set of commands used to activate, deactivate, configure, and manage a Tuxedo application.
- BEA Tuxedo MIB application programming interface—a set of functions for accessing and modifying information in the MIB.
- EventBroker—a BEA Tuxedo component that provides asynchronous routing of application events among the client and server processes running in a Tuxedo application, and distributes system events—typically faults or exceptional happenings—to whichever application processes want to receive them.

MIB Interfaces with Other System Components

The MIB accesses the following BEA Tuxedo system components:

- `TUXCONFIG` file—binary version of a Tuxedo application's configuration (`UBBCONFIG`) file. Every server machine in a Tuxedo application stores a copy of the `TUXCONFIG` file. The MIB updates the `TUXCONFIG` file and reads information from the `TUXCONFIG` file.
- Bulletin board—a memory segment in which all the configuration and dynamic processing information for a Tuxedo application is held at run time. Every server machine in a Tuxedo application has a bulletin board. The MIB updates the bulletin board and reads information from the bulletin board.
- ULOG—a user log file in which Tuxedo system and application messages—error messages, warning messages, information messages, and debugging messages—are stored. Every server machine in a Tuxedo application should have a ULOG. The MIB gathers information from the ULOG.
- TLOG—a transaction log file in which records of committed global transactions are stored. Every server machine in a Tuxedo application should have a TLOG. The MIB gathers information from the TLOG.

Management Operations Using the BEA Tuxedo Administration Console

Based on Java and Web technology, the BEA Tuxedo Administration Console lets you operate your BEA Tuxedo applications from virtually anywhere—even from home, given security

authorization. The Administration Console is a Java-based applet that you can download into your Web browser and use to remotely manage Tuxedo applications.

The 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.

Benefits of Using the BEA Tuxedo Administration Console

- Authentication—the 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 Tuxedo Administration Console are installed and made available to the Web server.
- Context-sensitive help—context-sensitive help is available for all 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. Encryption makes the system resistant to anyone trying to inject false administrative protocol messages into the stream.
- Firewall readiness—the port on which the BEA Tuxedo 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 Administration Console show state (for example, *not active*) or represent particular objects in the Tuxedo 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 Tuxedo application on which the Console server components reside, then initiate a download of Java applets. The applets implement the BEA Tuxedo 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.

Browser Requirements

Each release of the BEA Tuxedo system supports the currently available browsers. For information about browsers currently supported by the BEA Tuxedo Administration Console, see "Starting the BEA Tuxedo Administration Console" in *Installing the BEA Tuxedo System*.

Limitations

The BEA Tuxedo Administration Console has not been updated to support any new features introduced after BEA Tuxedo release 7.1.

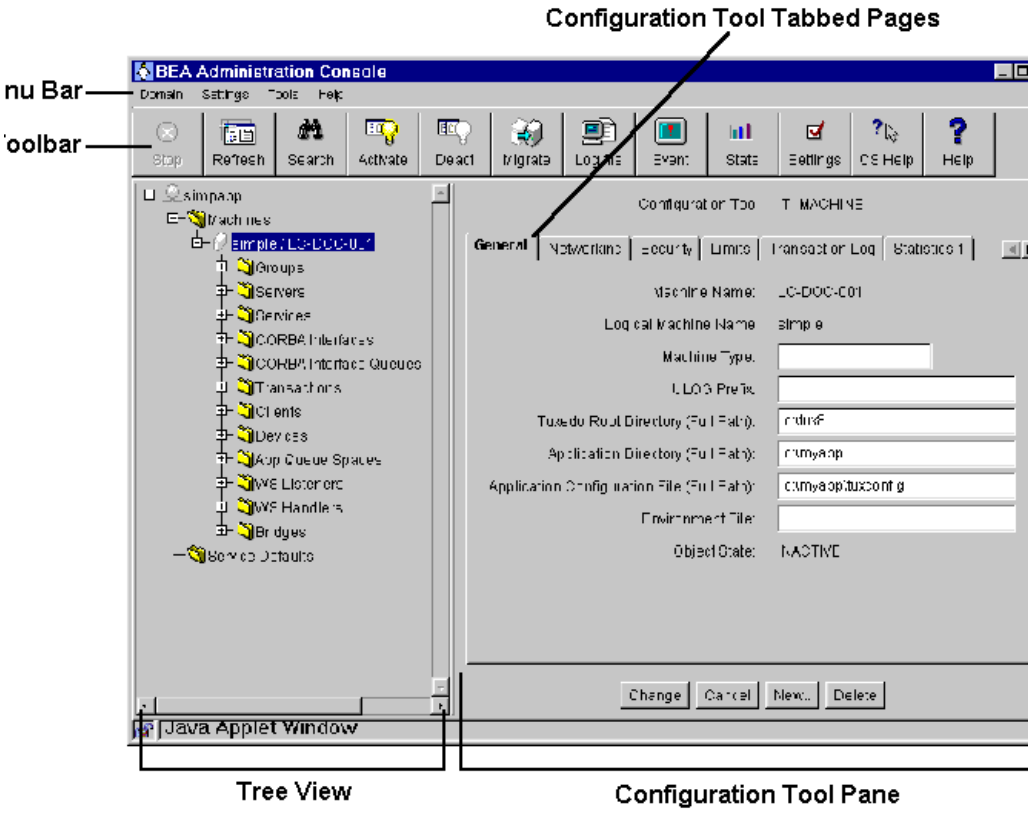
See Also

- *BEA Tuxedo Administration Console Online Help*
- "Exploring the Main Menu of the BEA Tuxedo Administration Console" on page 4-5
- "Ways to Monitor Your Application" in *Administering a BEA Tuxedo Application at Run Time*
- "Starting the BEA Tuxedo Administration Console" in *Installing the BEA Tuxedo System*

Exploring the Main Menu of the BEA Tuxedo Administration Console

When you first bring up the Web and invoke the BEA Tuxedo Administration Console, the following main window appears.

Figure 4-2 Main Menu of the BEA Tuxedo Administration Console



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. The toolbar buttons and some menu items are not fully displayed unless you are connected to a Tuxedo application.
- Tree View—a hierarchical representation of the administrative class objects (such as servers and clients) in a BEA Tuxedo application.
- 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.

Understanding the Tree View

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 application. 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 Tuxedo application) or a subset of objects.

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

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

For example, suppose your application 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 Tuxedo application.
Deactivate	Deactivates all or part of a Tuxedo application.
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 Tuxedo application.
Event	Displays a window for monitoring system-generated events.
Stats	Displays the tabbed pages that allow you to view a graphical presentation of Tuxedo application activity.
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 Administration Console Online Help in a separate Web browser.

See Also

- *BEA Tuxedo Administration Console Online Help*
- "Ways to Monitor Your Application" in *Administering a BEA Tuxedo Application at Run Time*
- "Starting the BEA Tuxedo Administration Console" in *Installing the BEA Tuxedo System*

Managing Operations Using Command-Line Utilities

BEA Tuxedo provides a set of commands for managing different parts of an application built on the BEA Tuxedo 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
- Administering your application queues using command-line utilities
- Administering your Domains application using command-line utilities

Configuring Your Application Using Command-Line Utilities

You can configure your application by using command-line utilities. Specifically, you can use a text editor to create and edit the configuration file (`UBBCONFIG`) for your application, and then use the command-line utility named `tmloadcf` to translate the text file (`UBBCONFIG`) to a binary file (`TUXCONFIG`). You are then ready to boot your application.

The following list identifies common command-line utilities that you can use to configure your application:

- `tmloadcf(1)`—a command, run on the master machine, that allows you to compile your application's `UBBCONFIG` file into the binary `TUXCONFIG` file. The `tmloadcf` command loads the binary file to the location defined by the `TUXCONFIG` environment variable.
- `tmunloadcf(1)`—a command, run on the master machine, that allows you to translate the binary `TUXCONFIG` file back to a text version, so that the `UBBCONFIG` and `TUXCONFIG` files can be synchronized. The `tmunloadcf` command prints the text version to standard output.

  Note: Dynamically updating the binary `TUXCONFIG` file does *not* update the text `UBBCONFIG` file.

- tpusradd(1), tpusrdel(1), tpusrmod(1)—a set of commands that allow you to create and manage a user database for authorization purposes.
- tpgrpadd(1), tpgrpdel(1), tpgrpmod(1)—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.
- tpacladd(1), tpaclcvt(1), tpacldel(1), and tpaclmod(1)—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.

Operating Your Application Using Command-Line Utilities

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

- tmboot(1)—a command, run on the master machine, that allows you to centrally start up your application servers. The tmboot command reads the TUXCONFIG environment variable to locate your application's TUXCONFIG file. The tmboot command loads TUXCONFIG into shared memory to establish the bulletin board, propagating the changes to the remote server machines in a multiple-machine domain.
- tmadmin(1)—an interactive meta-command, typically run on the master machine, that enables you to run subcommands to configure, monitor, and tune your application. You can use the tmadmin command before your application is booted (in configuration mode) or when your application is running.
- tmconfig(1)—another interactive meta-command, typically run on the master machine, that enables you to run subcommands to configure, monitor, and tune your application. You can use the tmconfig command only when your application is running. The tmconfig command is more powerful but less user friendly than the tmadmin command.
- tmshutdown(1)—a command, run on the master machine, that allows you to centrally shut down your application servers. The tmshutdown command reads the TUXCONFIG environment variable to locate your application's TUXCONFIG file.

Administering Your Application Queues Using Command-Line Utilities

You use the command-line utility qmadmin(1) to perform all administration functions for the application queues in your application. Like the tmadmin and tmconfig commands, qmadmin is an interactive meta-command that enables you to run many subcommands.

In a BEA Tuxedo application, you can have multiple application queue devices, and you can run application queues on multiple server machines. Each machine has its own queue device, so you can run qmadmin to monitor and manage a particular application queue device on each server machine.

Administering Your Domains Application Using Command-Line Utilities

To build a BEA Tuxedo Domains (multiple-domain) application, you integrate your existing BEA Tuxedo application with other domains. To do so, you must add a domain gateway group of system servers (DMADM, GWADM, and GWTDOMAIN) to your UBBCONFIG file. These servers are described in "BEA Tuxedo Domains (Multiple-Domain) Servers" on page 3-14.

All Domains configuration information for a BEA Tuxedo application involved in a Domains configuration is stored in a file known as DMCONFIG. Similar to the UBBCONFIG file, the DMCONFIG file may have any name as long as the content of the file conforms to the format described on reference page DMCONFIG(5) in *BEA Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*. You use a text editor to create and edit the DMCONFIG file, and then use the command-line utility named dmloadcf to translate the text file (DMCONFIG) to a binary file (BDMCONFIG). The BDMCONFIG file must reside on the machine that will run the DMADM server.

Note: The DMADM server may run on any machine (master machine, non-master machine) in a Tuxedo domain.

The following list identifies the command-line utilities that you can use to configure and operate the domain gateway group of system servers for a BEA Tuxedo application involved in a Domains configuration:

- dmloadcf(1)—a command, run on the same machine as the DMADM server, that allows you to compile an application's DMCONFIG file into the binary BDMCONFIG file. The dmloadcf command loads the binary file to the location defined by the BDMCONFIG environment variable.
- dmunloadcf(1)—a command, run on the same machine as the DMADM server, that allows you to translate the binary BDMCONFIG file back to a text version, so that the DMCONFIG and BDMCONFIG files can be synchronized. The dmunloadcf command prints the text version to standard output.

  Note: Dynamically updating the binary BDMCONFIG file does *not* update the text DMCONFIG file.

- `dmadmin(1)`—an interactive meta-command, typically run on the same machine as the `DMADM` server, that enables you to run subcommands to configure, monitor, and tune domain gateway groups. You can use the `dmadmin` command before your application is booted (in configuration mode) or when your application is running.

See Also

- *BEA Tuxedo Command Reference*.
- `DMADM(5)`, `DMCONFIG(5)`, `GWADM(5)`, `GWTDOMAIN(5)`, and `UBBCONFIG(5)`in *BEA Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*
- "Using Command-line Utilities to Monitor Your Application" in *Administering a BEA Tuxedo Application at Run Time*
- "BEA Tuxedo Administration Processes" on page 3-5
- "BEA Tuxedo Message Queuing Servers" on page 3-12
- "BEA Tuxedo Domains (Multiple-Domain) Servers" on page 3-14

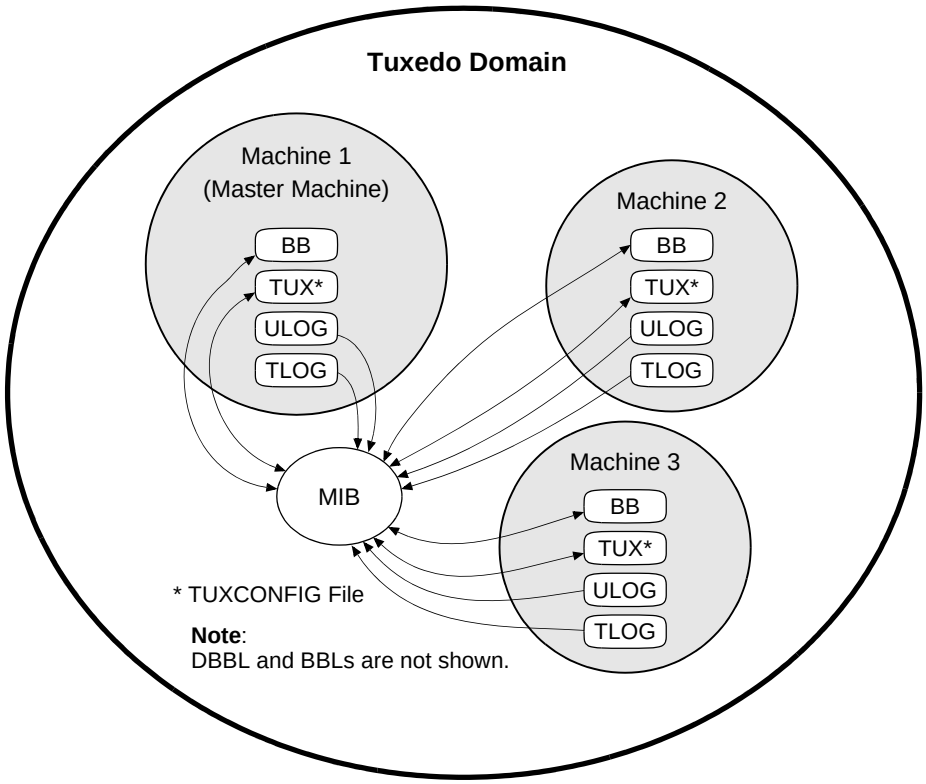
Managing Operations Using the MIB

The BEA Tuxedo MIB is used to administer a Tuxedo application. It defines the parts of an application that are required in every Tuxedo domain. MIB defines a Tuxedo 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).

When a Tuxedo server machine becomes active, it advertises the names of its services in the bulletin board (BB), which is the run-time (dynamic) representation of the MIB. (The bulletin board is where global and local state changes to the MIB are posted.) The Tuxedo system uses the binary `TUXCONFIG` file on the master machine to construct the bulletin board, and propagates a copy of the `TUXCONFIG` to the non-master machines in the application to set up the bulletin board on those machines. A bulletin board runs on each server machine in a Tuxedo application.

The following figure presents a high-level view of BEA Tuxedo MIB operation.

Figure 4-3 High-Level View of BEA Tuxedo MIB Operation



AdminAPI

The AdminAPI is an application programming interface for directly accessing and manipulating system settings in the BEA Tuxedo MIB. 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 Field Manipulation Language (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`.

Types of MIB Users

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

Type of User	Characteristics
System (or 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.

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.

See Also

- ACL_MIB(5), APPQ_MIB(5), DM_MIB(5), EVENT_MIB(5), MIB(5), TM_MIB(5), and WS_MIB(5) in *BEA Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*
- *Programming a BEA Tuxedo Application Using FML*

Managing Events Using EventBroker

An event is a state change or other occurrence in an application program or the BEA Tuxedo system that may be of interest to an administrator, an operator, or the software. Examples of events are “a stock traded at or above a specified price” or “a network failure occurred.”

BEA Tuxedo EventBroker provides asynchronous routing of application and system events among the processes running in a BEA Tuxedo ATMI application. Application events are occurrences of application-defined events. System events are occurrences of system-defined events.

Differences Between Application-Defined and System-Defined Events

Application-defined events are defined by application designers and are therefore application specific. Any of the events defined for an application may be tracked by the client and server processes running in the application.

System-defined events are defined by the BEA Tuxedo system code and are generally associated with objects defined in TM_MIB(5). A complete list of system-defined events is published on the EVENTS(5) reference page in *BEA Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*. Any of these events may be tracked by users of the BEA Tuxedo system.

Preparing an Application for Event Monitoring

The following table presents the basic tasks for preparing a BEA Tuxedo application for event monitoring.

Task	Description
1. Decide which events to monitor	Application programs are written to (a) detect when an event of interest has occurred and (b) post the event to the EventBroker through `tppost(3c)`. Application designers decide which events should be monitored. For system events, application designers select system-defined events from the `EVENTS(5)` reference page.
2. Create an events 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-defined event names begin with a dot (.); application-defined event names may not begin with a dot (.) To prepare an application-defined events list, application designers should consult the `EVENTS(5)`, `TMUSREVT(5)`, `TMSYSEVT(5)`, and `field_tables(5)` reference pages.

Subscribing to Events

As the administrator for your BEA Tuxedo application, you can enter subscription requests on behalf of a client or server process by making calls to `tpsubscribe(3c)` using the published list of application-defined or system-defined events. `EVENTS(5)`lists the notification message generated by a system event as well as the event name (used as an argument when `tppost(3c)` 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.

Each subscription for a system-defined event specifies one of several notification methods. One such method is placing messages in 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 ULOG.

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.

See Also

- "BEA Tuxedo Publish-and-Subscribe Servers" on page 3-13

- "About the EventBroker" in *Administering a BEA Tuxedo Application at Run Time*
- "Subscribing to Events" in *Administering a BEA Tuxedo Application at Run Time*
- `tppost(3c)`, `tpsubscribe(3c)`, and `tpunsubscribe(3c)`in *BEA Tuxedo ATMI C Function Reference*
- `EVENTS(5)`, `EVENT_MIB(5)`, `TMSYSEVT(5)`, `TMUSREVT(5)`, and `UBBCONFIG(5)` in *BEA Tuxedo File Formats, Data Descriptions, MIBs, and System Processes Reference*
- "Using Event-based Communication" in *Tutorials for Developing BEA Tuxedo ATMI Applications*