



BEA Tuxedo®

Scaling, Distributing, and Tuning CORBA Applications

Version 10.0
Document Released: September 28, 2007

Contents

1. Scaling BEA Tuxedo CORBA Applications

About Scaling BEA Tuxedo CORBA Applications	1-2
Application Scalability Requirements	1-2
BEA Tuxedo Scalability Features	1-2
Using Object State Management.	1-3
CORBA Object State Models	1-3
Implementing Stateless and Stateful Objects.	1-4
Parallel Objects	1-6
Replicating Server Processes and Server Groups	1-7
About Replicating Server Processes and Server Groups	1-7
Configuration Options	1-8
Replicating Server Processes	1-8
Replicating Server Groups	1-9
Using Multithreaded Servers.	1-9
About Multithreaded CORBA Servers	1-10
When to Use Multithreaded CORBA Servers	1-10
Coding Recommendations	1-11
Configuring a Multithreaded CORBA Server	1-11
Using Factory-based Routing (CORBA Servers Only)	1-11
About Factory-based Routing	1-12
Characteristics of Factory-based Routing	1-12
How Factory-based Is Implemented	1-13

Configuring Factory-based Routing in the UBBCONFIG File	1-13
Using Parallel Objects	1-14
About Parallel Objects	1-14
Configuring Parallel Objects.	1-16
Multiplexing Incoming Client Connections.	1-17
IIOP Listener and Handler	1-17
Increasing the Number of ISH Processes	1-18

2. Scaling CORBA Server Applications

About Scaling the Production Sample Application	2-1
Design Goals	2-2
How the Application Has Been Scaled	2-2
Changing the OMG IDL.	2-3
Using a Stateless Object Model	2-3
Scaling by Replicating Server Processes and Server Groups.	2-4
Replicating Server Processes in the Production Application	2-5
Replicating Server Groups in the Production Application	2-6
Configuring Replicated Server Processes and Groups in the Production Application	
2-8	
Scaling with Factory-based Routing	2-10
About Factory-based Routing in the Production Application.	2-10
Configuring Factory-based Routing in the UBBCONFIG File	2-11
Implementing Factory-based Routing in a Factory	2-14
What Happens at Run Time	2-15
Additional Design Considerations	2-16
About the Additional Design Considerations	2-16
Instantiating the Registrar and Teller Objects.	2-16
Ensuring That Student Registration Occurs in the Correct Server Group	2-18

Ensuring That the Teller Object Is Instantiated in the Correct Server Group	2-19
Scaling the Application Further	2-21

3. Distributing CORBA Applications

Why Distribute an Application?	3-2
About Distributing an Application	3-2
Benefits of a Distributed Application	3-2
Characteristics of Distributing an Application	3-3
Using Data-dependent Routing (BEA Tuxedo ATMI Servers Only)	3-3
About Data-dependent Routing	3-4
Characteristics of Data-dependent Routing	3-4
Sample Distributed Application	3-4
Example of UBBCONFIG Sections in a Distributed Application	3-5
Configuring the UBBCONFIG File	3-6
About the UBBCONFIG File in Distributed Applications	3-6
Modifying the GROUPS Section	3-7
Modifying the SERVICES Section	3-9
Modifying the INTERFACES Section	3-10
Creating the ROUTING Section	3-12
Configuring the factory_finder.ini (CORBA Applications Only)	3-13
Modifying the Domain Gateway Configuration File to Support Routing	3-13
About the Domain Gateway Configuration File	3-13
Parameters in the DM_ROUTING Section of the DMCONFIG File (BEA Tuxedo ATMI Only)	3-14

4. Tuning CORBA Applications

Maximizing Application Resources	4-3
When to Use MSSQ Sets (BEA Tuxedo ATMI Servers Only)	4-3
Enabling System-controlled Load Balancing	4-5

Configuring Replicated Server Processes and Groups	4-5
Configuring Multithreaded Servers	4-6
Setting the OPENINFO Parameter for Database Interoperation.	4-7
Parameters Used to Configure Multithreaded Servers	4-7
Assigning Priorities to Interfaces	4-8
Bundling Services into Servers (BEA Tuxedo ATMI Servers Only)	4-9
About Bundling Services	4-9
When to Bundle Services	4-9
Performance Options	4-10
Enhancing Efficiency with Application Parameters	4-11
MAXDISPATCHTHREADS	4-11
MINDISPATCHTHREADS	4-12
Setting the MAXACCESSERS, MAXOBJECTS, MAXSERVERS, MAXINTERFACES, and MAXSERVICES Parameters	4-13
Setting the MAXGTT, MAXBUFTYPE, and MAXBUFSTYPE Parameters . . .	4-13
Setting the SANITYSCAN, BLOCKTIME, BBLQUERY, and DBBLWAIT Parameters	4-14
Setting Application Parameters	4-14
Determining IPC Requirements	4-14
Measuring System Traffic	4-16
About System Traffic and Bottlenecks	4-17
Example of Detecting a System Bottleneck	4-17
Detecting Bottlenecks on UNIX	4-18
Detecting Bottlenecks on Windows	4-19

Scaling BEA Tuxedo CORBA Applications

This topic introduces key concepts and tasks for scaling BEA Tuxedo CORBA applications. This topic includes the following sections:

- [About Scaling BEA Tuxedo CORBA Applications](#)
- [Using Object State Management](#)
- [Replicating Server Processes and Server Groups](#)
- [Using Multithreaded Servers](#)
- [Using Factory-based Routing \(CORBA Servers Only\)](#)
- [Multiplexing Incoming Client Connections](#)

For more detailed information and examples for BEA Tuxedo CORBA applications, see [Chapter 2, “Scaling CORBA Server Applications.”](#)

Notes: The BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. BEA Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

About Scaling BEA Tuxedo CORBA Applications

This topic includes the following sections:

- [Application Scalability Requirements](#)
- [BEA Tuxedo Scalability Features](#)
- [Using Object State Management](#)

Application Scalability Requirements

Many applications perform adequately in an environment where between 1 to 10 server processes and 10 to 100 client applications are running. However, in an enterprise environment, applications may need to support hundreds of execution contexts (where the context can be a thread or a process), tens of thousands of client applications, and millions of objects at satisfactory performance levels.

Subjecting an application to exponentially increasing demands quickly reveals any resource shortcomings and performance bottlenecks in the application. Scalability is therefore an essential characteristic of BEA Tuxedo applications.

You can build highly scalable BEA Tuxedo applications by:

- Adding parallel processing capability to enable the BEA Tuxedo domain to process multiple client requests simultaneously.
- Sharing the processing load on the server applications across multiple machines.

BEA Tuxedo Scalability Features

BEA Tuxedo supports large-scale application deployments by:

- Optimizing object state management
- Load balancing objects and requests across replicated server processes and server groups
- Using multithreaded servers, which are appropriate for certain types of applications and processing environments
- For CORBA applications, using factory-based routing
- Using data-dependent routing (BEA Tuxedo ATMI only)
- Multiplexing incoming client connections

Using Object State Management

This topic includes the following sections:

- [CORBA Object State Models](#)
- [Implementing Stateless and Stateful Objects](#)
- [Parallel Objects](#)

Object state management is a fundamental concern of large-scale client/server systems because it is critical that such systems achieve optimized throughput and response time. For more detailed information about using object state management, see “[Using a Stateless Object Model](#)” on [page 2-3](#) and the technical article [Process-Entity Design Pattern](#).

CORBA Object State Models

BEA Tuxedo CORBA supports three object state management models:

- [Method-bound Objects](#)
- [Process-bound Objects](#)
- [Transaction-bound Objects](#)

For more information about these models, see “Server Application Concepts” in [Creating CORBA Server Applications](#)

Method-bound Objects

Method-bound objects are loaded into the machine’s memory only for the duration of the client invocation. When the invocation is complete, the object is deactivated and any state data for that object is flushed from memory. In this document, a method-bound object is considered to be a *stateless object*.

You can use method-bound objects to create a stateless server model in your application. By using a stateless server model, you move requests that are already directed to active objects to any available server, which allows concurrent execution for thousands and even millions of objects. From the client application view, all the objects are available to service requests. However, because the server application maps objects into memory only for the duration of client invocations, few of the objects managed by the server application are in memory at any given moment.

Process-bound Objects

Process-bound objects remain in memory beginning when they are first invoked until the server process in which they are running is shut down. A process-bound object can be activated upon a client invocation or explicitly before any client invocation (a *preactivated* object). Applications can control the deactivation of process-bound objects. In this document, a process-bound object is considered to be a *stateful object*.

When appropriate, process-bound objects with a large amount of state data can remain in memory to service multiple client invocations, thereby avoiding reading and writing the object's state data on each client invocation.

Transaction-bound Objects

Transaction-bound objects can also be considered stateful because, within the scope of a transaction, they can remain in memory between invocations. If the object is activated within the scope of a transaction, the object remains active until the transaction is either committed or rolled back. If the object is activated outside the scope of a transaction, its behavior is the same as that of a method-bound object (it is loaded for the duration of the client invocation).

Implementing Stateless and Stateful Objects

In general, application developers need to balance the costs of implementing stateless objects against the costs of implementing stateful objects.

About Stateless and Stateful Objects

The decision to use stateless or stateful objects depends on various factors. In the case where the cost to initialize an object with its durable state is expensive—because, for example, the object's data takes up a great deal of space, or the durable state is located on a disk very remote from the servant that activates it—it may make sense to keep the object stateful, even if the object is idle during a conversation. In the case where the cost to keep an object active is expensive in terms of machine resource usage, it may make sense to make such an object stateless.

By managing object state in a way that is efficient and appropriate for your application, you can maximize your application's ability to support large numbers of simultaneous client applications that use large numbers of objects. The way that you manage object state depends on the specific characteristics and requirements of your application. For CORBA applications, you manage object state by assigning the `method` activation policy to these objects, which has the effect of deactivating idle object instances so that machine resources can be allocated to other object instances.

When to Use Stateless Objects

Stateless objects generally provide good performance and optimal usage of server resources, because server resources are never used when objects are idle. Using stateless objects is a good approach to implementing server applications and are particularly appropriate when:

- The client application waits for user input between invocations on the object.
- The client request contains all the data needed by the server application, and the server can process the client request using only that data.
- The object has high access rates, but low access rates from any one particular client application.

By making an object stateless, you can generally assure that server application resources are not being reserved unnecessarily while waiting for input from the client application.

An application that employs a stateless object model has the following characteristics:

- Information about and associated with an invocation is not maintained after the server application has finished executing a client request.
- An incoming client request is sent to the first available server process. After the request has been satisfied, the application state disappears and the server application is available for another client application request.
- Durable state information for the object exists outside the server process. With each invocation on this object, the durable state is read into memory.
- Successive requests on an object from a given client application may be processed by a different server process.
- The overall system performance of a machine that is running stateless objects is usually enhanced.

When to Use Stateful Objects

A stateful object, once activated, remains in memory until a specific event occurs, such as the process in which the object exists is shut down, or the transaction in which the object is activated is completed.

Using stateful objects is recommended when:

- An object is used frequently by a large number of client applications, such as long-lived, well-known objects. When the server application keeps these objects active, the client

application typically experiences minimal response time in accessing them. These active objects are shared by many client applications, and therefore relatively few objects of this type exist in memory.

Note: You should carefully consider how objects will potentially be involved in a transaction. An object can be bound to a particular process temporarily (transaction-bound) or permanently (process-bound). An object that is involved in a transaction cannot be invoked by another client application or object (BEA Tuxedo will likely return an error indicating that the object is busy). Stateful objects that are intended to be used by a large number of client applications can create bottlenecks if they are involved in transactions frequently or for long durations.

- A client application must invoke successive operations on an object to complete a transaction, and the client application is not idle while it waits for user input between invocations. If the object were deactivated between invocations, there would be a degradation of response time because state would be written and read between each invocation.

Stateful objects have the following behavior:

- State information is maintained between server invocations, and the object typically remains dedicated to a given client application for a specified duration. Even though data is sent and received between the client and server applications, the server process maintains additional context or application state information in memory.
- When one or more stateful objects use a lot of machine resources, server performance for tasks and processes not associated with the stateful object may be lower than with a stateless server model.

For example, if an object has a lock on a database and is caching large amounts of data in memory, that database and the memory used by that stateful object are unavailable to other objects, potentially for the entire duration of a transaction.

Parallel Objects

Parallel objects are, by definition, stateless objects so they can exist concurrently on more than one server. In release 8.0 of BEA Tuxedo, you can use the Implementation Configuration File (ICF) to force all objects in a specific implementation to be parallel objects. The effect is to improve performance. For more information on parallel objects, see [“Using Parallel Objects” on page 1-14](#).

Replicating Server Processes and Server Groups

This topic includes the following sections:

- [About Replicating Server Processes and Server Groups](#)
- [Configuration Options](#)
- [Replicating Server Processes](#)
- [Replicating Server Groups](#)

For more detailed information about replicating server processes and server groups, see the following topics:

- [“Configuring Replicated Server Processes and Groups” on page 4-5.](#)
- [“Scaling by Replicating Server Processes and Server Groups” on page 2-4.](#)

About Replicating Server Processes and Server Groups

The BEA Tuxedo CORBA environment allows CORBA objects to be deployed across multiple servers to provide additional failover reliability and to split the client’s workload through load balancing. BEA Tuxedo CORBA load balancing is enabled by default. For more information about configuring load balancing, see [“Enabling System-controlled Load Balancing” on page 4-5](#). For more information about distributing the application workload using BEA Tuxedo CORBA features, see [Chapter 3, “Distributing CORBA Applications.”](#)

The BEA Tuxedo architecture provides the following server organization:

- Groups—individual servers can be combined to form a group. A group of servers runs on a single machine. Typically, the servers in a group access common resources (such as a database).
- Domains—machines can be combined to form a domain. A domain is administered centrally. Multiple domains are administered separately. Domains can also be interconnected and requests can be transparently routed from one domain to another. However, each domain is independently administered.

This architecture allows new servers, groups, or machines to be dynamically added or removed, to adapt the application to high- or low-demand periods, or to accommodate internal changes required to the application. The BEA Tuxedo run time provides load balancing and failover by routing requests across available servers.

System administrators can scale a BEA Tuxedo application by:

- [Replicating Server Processes](#). Increase the number of server processes to support more active objects within a group and load balancing among servers.
- [Replicating Server Groups](#). Increase the number of server groups so that BEA Tuxedo can balance the load by distributing processing requests across multiple server machines.

Configuration Options

You can configure server applications as:

- A single machine with one or more server processes implementing one or more interfaces. The servers can be single-threaded or multithreaded.
- Multiple machines with multiple server processes and multiple interfaces.

You can add more parallel processing capability to client/server applications by replicating server processes or add more threads. You can add more server groups to split processing across resource managers. For CORBA applications, you can implement factory-based routing, as described in [“Using Factory-based Routing \(CORBA Servers Only\)” on page 1-11](#).

Replicating Server Processes

System administrators can scale an application by replicating the servers to support more concurrent active objects, or process more concurrent requests, on the server node. To configure replicated server processes, see [“Configuring Replicated Server Processes and Groups” on page 4-5](#).

Note: Release 8.0 of BEA Tuxedo supports the user-controlled concurrency model for active objects. For a discussion of the concurrency policy feature, see [“Parallel Objects” on page 1-6](#).

Benefits

The benefits of using replicated server processes include:

- Load balancing incoming requests.
- Processing client requests on any server within a group. As requests arrive in the BEA Tuxedo domain for the server group, BEA Tuxedo routes the request to the least busy server process within that group.
- Improving the server application’s performance by using multiple server processes. Instead of having one server process handling one client request at one time, multiple server processes are available to handle multiple client requests simultaneously.

- Providing failover protection in the event that one of the server processes stops.

Guidelines

To achieve the maximum benefit of using replicated server processes, make sure that the CORBA objects instantiated by your server application have unique object IDs. This allows a client invocation on an object to cause the object to be instantiated on demand, within the bounds of the number of server processes that are available, and not queued up for an already active object.

You should also consider the trade-off between providing better application recovery by using multiple processes versus more efficient performance using threads (for some types of application patterns and processing environments).

Better failover occurs only when you add processes, *not* threads. For information about using single-threaded and multithreaded servers, see [“When to Use Multithreaded CORBA Servers” on page 1-10](#).

Replicating Server Groups

Server groups are unique to BEA Tuxedo and are key to the scalability features of BEA Tuxedo. A group contains one or more servers on a single node. System administrators can scale a BEA Tuxedo application by replicating server groups and configuring load balancing within a domain.

Replicating a server group involves defining another server group with the same type of servers and resource managers to provide parallel access to a shared resource (such as a database). CORBA applications, for example, can use factory-based routing to split processing across the database partitions.

The `UBBCONFIG` file specifies how server groups are configured and where they run. By using multiple server groups, BEA Tuxedo can:

- Spread the processing load for a given application or set of applications across additional machines.
- For CORBA applications, use factory-based routing to send one set of requests on a given interface to one group, and another set of requests on the same interface to another group.

To configure replicated server groups, see [“Configuring Replicated Server Processes and Groups” on page 4-5](#).

Using Multithreaded Servers

This topic includes the following sections:

- [About Multithreaded CORBA Servers](#)
- [When to Use Multithreaded CORBA Servers](#)
- [Coding Recommendations](#)
- [Configuring a Multithreaded CORBA Server](#)

For instructions on how to configure servers for multithreading, see “[Configuring Multithreaded Servers](#)” on page 4-6.

About Multithreaded CORBA Servers

System administrators can scale a BEA Tuxedo application by enabling multithreading in CORBA servers, and by tuning configuration parameters (the maximum number of server threads that can be created) in the application’s `UBBCONFIG` file.

BEA Tuxedo CORBA supports the ability to configure multithreaded CORBA applications. A multithreaded CORBA server can service multiple object requests simultaneously, while a single-threaded CORBA server runs only one request at a time.

Server threads are started and managed by the BEA Tuxedo CORBA software rather than an application program. Internally, BEA Tuxedo CORBA manages a pool of available server threads. If a CORBA server is configured to be multithreaded, then when a client request is received, an available server thread from the thread pool is scheduled to execute the request. While the object is active, the thread is busy. When the request is complete, the thread is returned to the pool of available threads.

When to Use Multithreaded CORBA Servers

Designing an application to use multiple, independent threads provides concurrency within an application and can improve overall throughput. Using multiple threads enables applications to be structured efficiently with threads servicing several independent tasks in parallel.

Multithreading is particularly useful when:

- There is a set of lengthy operations that do not necessarily depend on other processing.
- The amount of data to be shared is small and identifiable.
- You can break the task into various activities that can be executed in parallel.
- There are occasions where objects must be reentrant.

Some computer operations take a substantial amount of time to complete. A multithreaded application design can significantly reduce the wait time between the request and completion of operations. This is true in situations when operations perform a large number of I/O operations such as when accessing a database, invoking operations on remote objects, or are CPU-bound on a multiprocessor machine. Implementing multithreading in a server process can increase the number of requests a server processes in a fixed amount of time.

The primary requirement for multithreaded server applications is the simultaneous handling of multiple client requests. For more information on the requirements and benefits of using multithreaded servers, see *Creating CORBA Server Applications*.

Coding Recommendations

So as to be able to analyze the performance of multithreaded servers, include one of the following identifiers in each message if your client or server application sends messages to the user log (ULOG):

- Object ID
- Transaction ID (if the object is transactional)

Configuring a Multithreaded CORBA Server

To configure a multithreaded CORBA server, you change settings in the application's UBBCONFIG file. For information about defining the UBBCONFIG parameters to implement a multithreaded server, see [“Configuring Multithreaded Servers” on page 4-6](#).

Using Factory-based Routing (CORBA Servers Only)

This topic includes the following sections:

- [About Factory-based Routing](#)
- [How Factory-based Is Implemented](#)
- [Configuring Factory-based Routing in the UBBCONFIG File](#)

This topic introduces factory-based routing in BEA Tuxedo CORBA applications. For more detailed information about using factory-based routing, see [“Configuring Factory-based Routing in the UBBCONFIG File” on page 2-11](#).

About Factory-based Routing

Factory-based routing enables you to specify what server group is associated with an object reference. As a result, you can define the group and machine in which a given object is instantiated and then distribute the processing load for a given application across multiple machines.

With factory-based routing, routing is performed when a factory creates an object reference. The factory specifies field information in its call to the BEA Tuxedo CORBA TP Framework to create an object reference. The TP Framework executes the routing algorithm based on the routing criteria that is defined in the `ROUTING` section of an application's `UBBCONFIG` file. The resulting object reference has, as its target, an appropriate server group for the handling of method invocations on the object reference. Any server that implements the interface in that server group is eligible to activate the servant for the object reference.

Thus, the activation of CORBA objects can be distributed by server group based on the defined criteria and different implementations of CORBA interfaces can be supplied in different groups. So you can replicate the same CORBA interface across multiple server groups, based on defined, group-specific differences.

The primary benefit of factory-based routing is that it provides a simple means to scale an application, and invocations on a given interface in particular, across a growing deployment environment. Distributing the deployment of an application across additional machines is strictly an administrative function that does not require you to recode or rebuild the application.

Characteristics of Factory-based Routing

Factory-based routing has the following characteristics:

- The factory object implementation can indirectly control the location of the created CORBA object by supplying application-specific routing information.
- An implementation of a particular CORBA interface can exist in more than one server process, as shown in [“Configuring Factory-based Routing in the UBBCONFIG File” on page 2-11](#).
- Multiple CORBA interfaces can reside in a single server group.
- All server processes in a particular server group *do not* need to use the same CORBA interfaces.
- All object instances that offer a given interface within a group must support the same version of the implementation.

- Routing uses the bulletin board criteria and occurs in a server call.

How Factory-based Is Implemented

To implement factory-based routing, you must change the way your factories create object references. First, you must coordinate with the system designer to determine the fields and values to be used as the basis for routing. Then, for each interface, you must configure factory-based routing such that the interface definition for the factory specifies the parameter that represents the routing criteria that is used to determine the group ID.

To configure factory-based routing, define the following information in the `UBBCONFIG` file:

- Routing criteria identifier for a CORBA interface in the `INTERFACES` section.
- As many server groups as are required for distributing the system in the `GROUPS` section.
- Routing criteria in the `ROUTING` section.
- Groups, machines, and databases as required.

Notes: When implementing factory-based routing, remember that an object with a given interface and OID can be simultaneously active in two different groups *if* those two groups both contain the same object implementation. This can be avoided if your factories generate unique OIDs. To guarantee that only one object instance of a given interface name and OID is available at any one time in your domain, you must either:

- Use factory-based routing to ensure that objects with a particular OID are always routed to the same group, or
- Configure your domain so that a given object implementation is in only one group.

If multiple clients have an object reference that contains a given interface name and OID, the reference will always be routed to the same object instance.

Thereafter, the object reference will contain additional information that is used to provide an indication of where the target server exists. Factory-based routing is performed once per CORBA object, when the object reference is created.

Configuring Factory-based Routing in the UBBCONFIG File

Routing criteria specify the data values used to route requests to a particular server group. To configure factory-based routing, you define routing criteria in the `ROUTING` section of the `UBBCONFIG` file (for each interface for which requests are routed). For more detailed information

about configuring factory-based routing, see [“Configuring Factory-based Routing in the UBBCONFIG File” on page 2-11](#).

To configure factory-based routing across multiple domains, you must also configure the `factory_finder.ini` file to identify factory objects that are used in the current (local) domain but that are resident in a different (remote) domain. For more information, see “Configuring Multiple Domains for CORBA Applications” in the *Using the BEA Tuxedo Domains Component*.

Using Parallel Objects

This topic includes the following sections:

- [About Parallel Objects](#)
- [Configuring Parallel Objects](#)

About Parallel Objects

Support for parallel objects has been added in release 8.0 of BEA Tuxedo as a performance enhancement. The parallel objects feature enables you to designate all business objects in particular application as stateless objects. The effect is that, unlike stateful business objects, which can only run on one server in a single domain, stateless business objects can run on all servers in a single domain. Thus, the benefits of parallel objects are as follows:

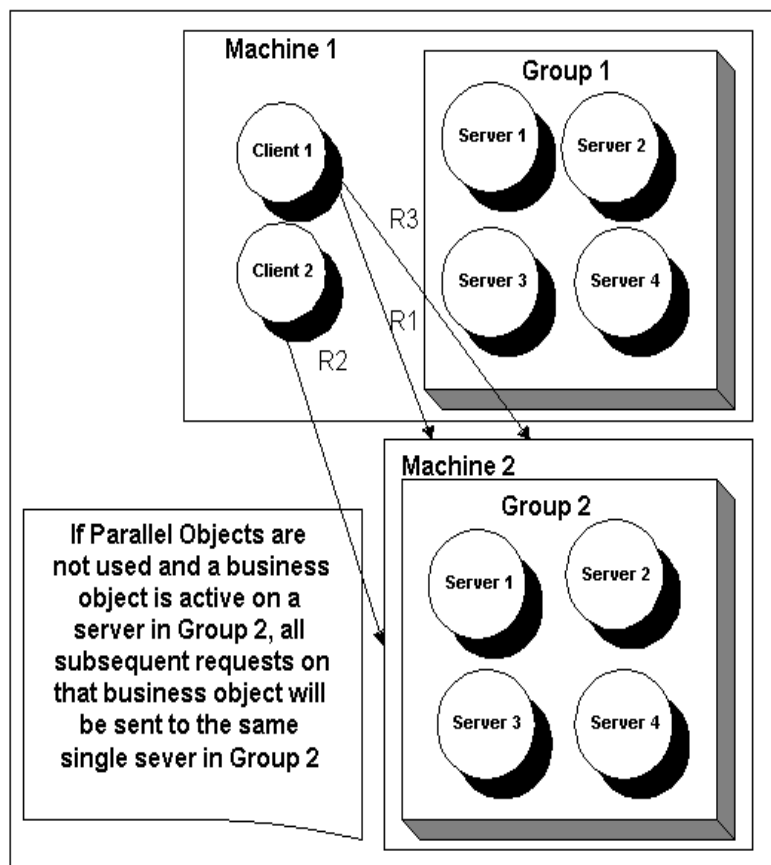
Note: You enable the parallel objects feature by setting the concurrency policy option to `user_controlled` in the ICF file. For more information, see [“Configuring Parallel Objects” on page 1-16](#).

- Parallel objects, which are stateless, can run on multiple servers in the same domain at the same time. The resulting utilization of all servers to service concurrent multiple requests improves performance.
- When BEA Tuxedo services requests to parallel business objects, it always looks for an available server to the local machine first. If all servers on the local machine are busy processing the requested business object, BEA Tuxedo looks for an available server on other machines in the local domain. Thus, if there are multiple servers on the local machine, network traffic is reduced and performance is improved.

As illustrated in [Figure 1-1](#), if a stateful business object is active on a server on Machine 2, all subsequent requests to that business object will be sent to Group 2 on Machine 2. If the active object on Machine 2 is busy processing another request, the request is queued. Even after the business object stops processing requests on Machine 2, all subsequent requests on that stateful

business object will still be sent to Group 2. After the object is deactivated on Machine 2, subsequent requests will be sent to Group 2 on Machine 2 and can be processed by other servers in Group 2.

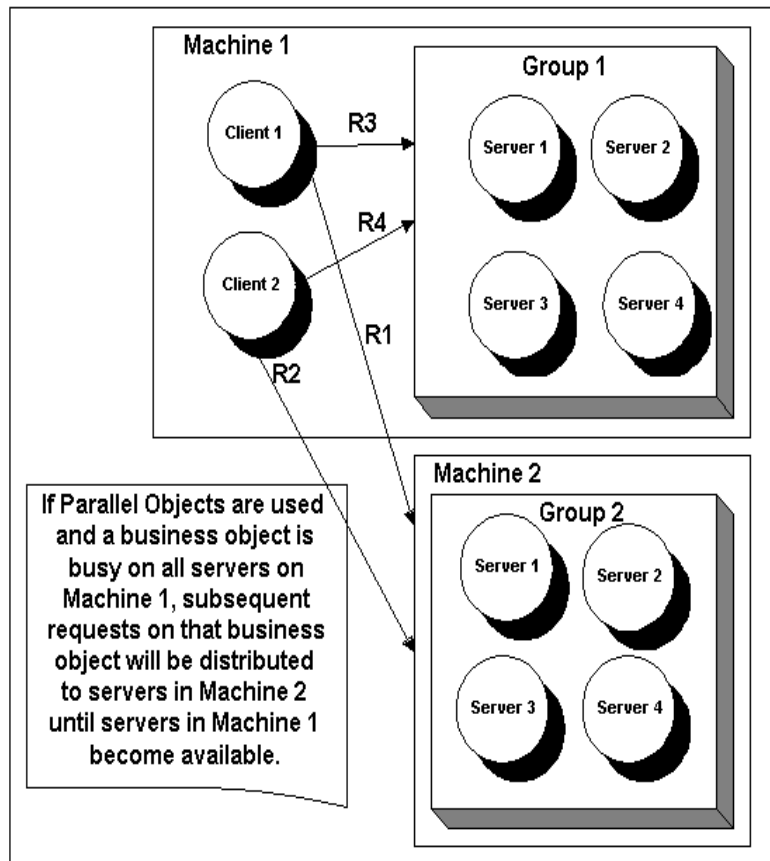
Figure 1-1 Using Stateful Business Objects



As illustrated in [Figure 1-2](#), if a parallel object is running on all the servers in Group 1 on Machine 1 (multiple instances of stateless, user-controlled business objects can run on multiple servers at the same time), subsequent requests to that business object will be sent to Machine 2 and distributed to the servers in Group 2 until a server becomes available in Group 1. As long as there is a server available on the local machine, requests will be distributed to the servers on Machine 1, unless the BEA Tuxedo load-balancing feature determines that, due to loads on the servers, the

request should be serviced by a server in Group 2. To make this determination, the load-balancing feature uses the `LOAD` parameter, which is set in the `INTERFACES` section of the

Figure 1-2 Using Stateless Business Objects



UBBCONFIG file. For information on the `LOAD` parameter, see [“Modifying the INTERFACES Section” on page 3-10](#).

Configuring Parallel Objects

Support for parallel objects was added to BEA Tuxedo in release 8.0. You use the ICF file to implement parallel objects for a particular CORBA application. The ICF includes a

user-controlled concurrency policy option that sets all business objects implemented in the application, to which the ICF file applies, to stateless objects.

The concurrency policy determines whether the Active Object Map (AOM) is used to guarantee that an object is active in only one server at any one time. In previous releases, use of the AOM was mandatory, not optional. Use of the AOM is referred to as system-controlled concurrency. Unlike the system-controlled concurrency model, the user-controlled model, which does not use the AOM, allows the same object to be active in more than one server at a time. Thus, user-controlled concurrency can be used to improve performance and load balancing. For more information about configuring user-controlled concurrency for parallel objects, see “Parallel Objects” in the [CORBA Programming Reference](#).

Multiplexing Incoming Client Connections

This topic includes the following sections:

- [IIOP Listener and Handler](#)
- [Increasing the Number of ISH Processes](#)

System administrators can scale a BEA Tuxedo application by increasing, in the `UBBCONFIG` file, the number of incoming client connections that an application site supports. BEA Tuxedo provides a multicontexted, multistated gateway of listener/handlers to handle the multiplexing of all the requests issued by the client.

IIOP Listener and Handler

The IIOP Listener (ISL) enables access to BEA Tuxedo CORBA objects by remote BEA Tuxedo CORBA clients that use IIOP. The ISL is a process that listens for remote CORBA clients requesting IIOP connections. The IIOP Handler (ISH) is a multiplexor process that acts as a surrogate on behalf of the remote CORBA client. Both the ISL and ISH run on the application site. An application site can have one or more ISL processes and multiple associated ISH processes. Each ISH is associated with a single ISL.

The client connects to the ISL process using a known network address. The ISL balances the load among ISH processes by selecting the best available ISH and passing the connection directly to it. The ISL/ISH manages the context on behalf of the application client. For more information about ISL and ISH, see the description of ISL in the [File Formats, Data Descriptions, MIBs, and System Processes Reference](#).

Increasing the Number of ISH Processes

System administrators can scale a BEA Tuxedo CORBA application by increasing the number of ISH processes on an application site, thereby enabling the ISL to load balance among more ISH processes. By default, an ISH can handle up to 10 client connections. To increase this number, pass the optional `CLOPT -x mpx-factor` parameter to the ISL command, specifying in *mpx-factor* the number of ISH client connections each ISH can handle (up to 4096), and therefore the degree of multiplexing, for the ISH. Increasing the number of ISH processes may affect application performance as the application site services more concurrent processes.

System administrators can tune other ISH options as well to scale BEA Tuxedo applications. For more information, see the description of ISL in the [File Formats, Data Descriptions, MIBs, and System Processes Reference](#).

Scaling CORBA Server Applications

This topic includes the following sections:

- [About Scaling the Production Sample Application](#)
- [Changing the OMG IDL](#)
- [Using a Stateless Object Model](#)
- [Scaling by Replicating Server Processes and Server Groups](#)
- [Scaling with Factory-based Routing](#)
- [Additional Design Considerations](#)
- [Scaling the Application Further](#)

Using the Production sample application as an example, this topic demonstrates scaling an CORBA C++ application to increase its processing capability. Before you begin, be sure to read:

- [Chapter 1, “Scaling BEA Tuxedo CORBA Applications,”](#) for a comprehensive introduction to tuning and scaling BEA Tuxedo CORBA applications.
- Production Sample Application in the BEA Tuxedo online documentation.

About Scaling the Production Sample Application

The Production sample application provides the same end-user functionality as the Wrapper sample application. The Production sample application demonstrates how to use features of the BEA Tuxedo software to scale an existing BEA Tuxedo application.

This section includes the following topics:

- [Design Goals](#)
- [How the Application Has Been Scaled](#)

Design Goals

The primary design goal of the Production sample application is to significantly increase the number of client applications it can accommodate by:

- Processing, in parallel and on one machine, client requests on multiple objects that implement the same interface.
- Directing requests on behalf of certain students to one machine, and other students to other machines.
- Adding more machines to share the processing load.

How the Application Has Been Scaled

To accommodate these design goals, the Production sample application has been scaled by:

- Implementing a stateless object model to scale up the number of client requests the server process can manage simultaneously.
- Replicating the University, Billing, and BEA Tuxedo Teller Application server processes within the groups in which they are configured (the `ORA_GRP` and `APP_GRP` server groups defined in the `UBBCONFIG` file).
- Replicating the `ORA_GRP` and `APP_GRP` server groups on an additional server machine, Production Machine 2, and also partitioning the database.
- Assigning unique object IDs (OIDs) to the following objects so that they can be instantiated multiple times simultaneously in their respective groups.
 - RegistrarFactory
 - Registrar
 - TellerFactory
 - Teller

This makes these objects available on a per-client application (and not per-process) basis, thereby accommodating a parallel processing capability.

- Implementing factory-based routing to direct client requests on behalf of some students to one machine, and other students to another machine.

Note: To make the Production sample application easy to use, this application is configured on the BEA Tuxedo software kit to run on one machine, using one database. The examples shown in this chapter, however, show running this application on two machines using two databases.

The Production sample application is designed so that it can be configured to run on several machines and to use multiple databases. Changing the configuration to multiple machines and databases involves modifying the `UBBCONFIG` file and partitioning the databases, which is described in [“Scaling the Application Further” on page 2-21](#).

The sections that follow describe how the Production sample application uses replicated server processes and server groups, object state management, and factory-based routing to meet its scalability goals.

Changing the OMG IDL

The only OMG IDL changes for the Production sample application are limited to the `find_registrar()` and `find_teller()` operations on, respectively, the `RegistrarFactory` and `TellerFactory` objects. These two operations need to be modified to require, respectively, a student ID and account number, which are needed to implement factory-based routing. See [“Scaling with Factory-based Routing” on page 2-10](#) to read about how the Production sample application implements and uses factory-based routing.

Using a Stateless Object Model

This section describes how object state management is used with the `Registrar` and `Teller` objects in the Production sample applications to increase the application’s scalability. For an introduction to object state management, see [“Using Object State Management” on page 1-3](#).

To increase scalability, the `Registrar` and `Teller` objects are configured in the Production server application with the `method` activation policy. The `method` activation policy assigned to these two objects results in the following behavior changes:

- Whenever these objects are invoked, they are instantiated by the BEA Tuxedo domain in the appropriate server group.
- After the invocation is complete, the BEA Tuxedo domain deactivates these objects.

With the Basic through the Wrapper sample applications, the Registrar object was process-bound (process activation policy). All client requests on the Registrar object invariably went to the same object instance in the memory of the server machine. The Basic sample application design may be adequate for a small-scale deployment. However, as client application demands increase, client requests on the Registrar object eventually become queued, and response time drops.

However, when the Registrar and Teller objects are stateless (method activation policy), and the server processes that manage these objects are replicated, the Registrar and Teller objects can process multiple client requests in parallel. The only constraint on the number of simultaneous client requests that these objects can handle is the number of server processes that are available that can instantiate the Registrar and Teller objects. These stateless objects, thereby, make for more efficient use of machine resources and reduced client response time.

Most importantly, so that BEA Tuxedo CORBA can instantiate copies of the Registrar and Teller objects in each of the replicated server processes, each copy of these objects must be unique. To make each instance of these objects unique, the factories for those objects must assign unique object IDs to them.

For the BEA Tuxedo application to instantiate copies of the Registrar and Teller objects in each of the replicated server application processes, each copy of the Registrar and Teller objects have a unique object ID (OID). The factories that create these objects are responsible for assigning them unique OIDs. For information about generating unique object IDs, see [Creating CORBA Server Applications](#). For more information about other design considerations, see [“Additional Design Considerations” on page 2-16](#).

Scaling by Replicating Server Processes and Server Groups

This topic includes the following sections:

- [Replicating Server Processes in the Production Application](#)
- [Replicating Server Groups in the Production Application](#)
- [Configuring Replicated Server Processes and Groups in the Production Application](#)

This topic describes how the Production sample application was scaled by replicating server processes and server groups. For an introduction to this topic, see [“Replicating Server Processes and Server Groups” on page 1-7](#).

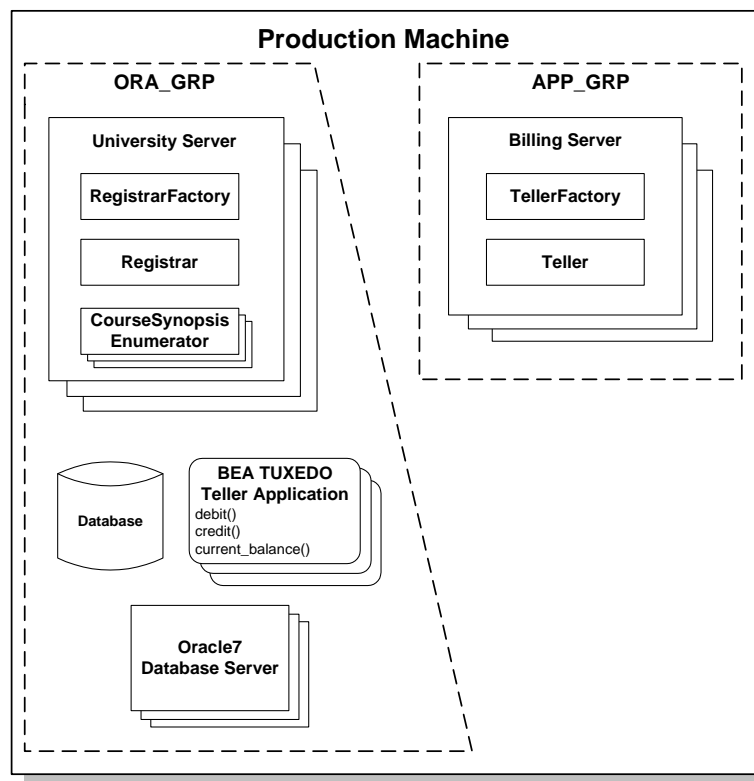
Replicating Server Processes in the Production Application

This section describes how the Production sample application replicates server applications. For an introduction to this feature, see [“Replicating Server Processes” on page 1-8](#).

[Figure 2-1](#) shows the replicated `ORA_GRP` and `APP_GRP` groups running on a single machine.

- The University server application, BEA Tuxedo Teller Application, and Oracle7 TMS server processes are replicated within the `ORA_GRP` group.
- The Billing server process is replicated within the `APP_GRP` group.

Figure 2-1 Replicated Server Groups in the Production Sample



When a request arrives for either of these groups, the BEA Tuxedo domain has several server processes available that can process the request, and the BEA Tuxedo domain can choose the server process that is the least busy.

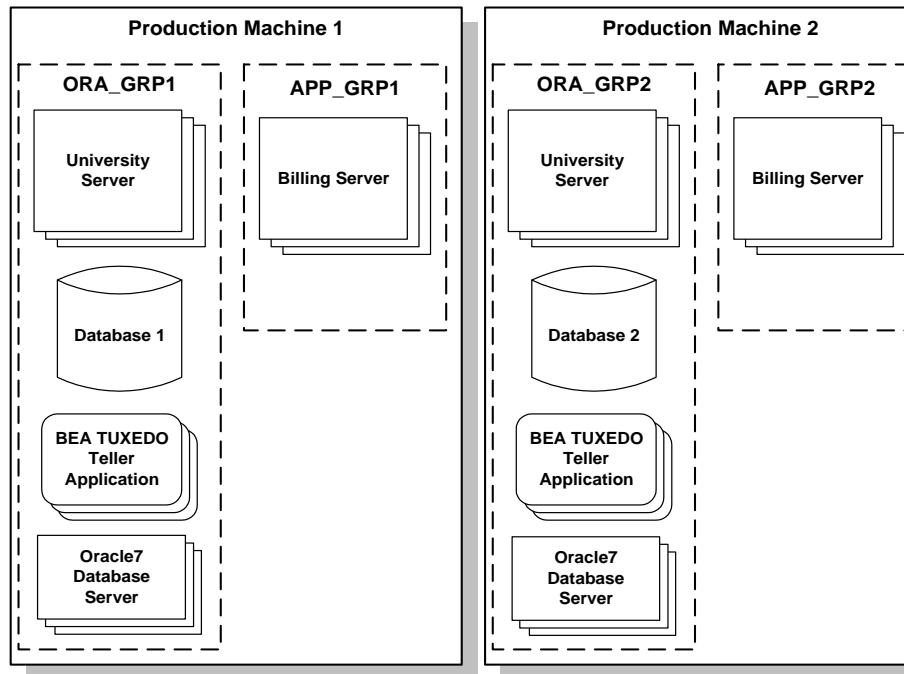
In [Figure 2-1](#), note the following points:

- At any time, there may be no more than one instance of the `RegistrarFactory`, `Registrar`, `TellerFactory`, or `Teller` objects within a given server process.
- There may be any number of `CourseSynopsisEnumerator` objects in any University server process.

Replicating Server Groups in the Production Application

This section describes how the Production sample application replicates server groups. For an introduction to this feature, see [“Replicating Server Groups” on page 1-9](#).

[Figure 2-2](#) shows the Production sample application groups replicated on another machine, as specified in the application’s `UBBCONFIG` file, as `ORA_GRP2` and `APP_GRP2`.

Figure 2-2 Replicating Server Groups Across Machines

In [Figure 2-2](#), the only difference between the content of the groups on Production Machines 1 and 2 is the database:

- The database on Production Machine 1 contains student and account information for students with IDs between 100001 and 100005.
- The database on Production Machine 2 contains student and account information for students with IDs between 100006 and 100010.

Note: The course information table in both databases is identical.

Note that the student information in a given database may be completely unrelated to the account information in the same database.

For more information about how the Production sample application uses factory-based routing to distribute the application's processing load across multiple machines, see [“Scaling with Factory-based Routing” on page 2-10](#).

Configuring Replicated Server Processes and Groups in the Production Application

[Listing 2-1](#) shows excerpts from the `GROUPS` and `SERVERS` sections of the `UBBCONFIG` file for the Production sample application.

Listing 2-1 GROUPS and SERVERS Sections in a UBBCONFIG File

```
*GROUPS
  APP_GRP1
    LMID      = SITE1
    GRPNO     = 2
    TMSNAME   = TMS
  APP_GRP2
    LMID      = SITE1
    GRPNO     = 3
    TMSNAME   = TMS
  ORA_GRP1
    LMID      = SITE1
    GRPNO     = 4
    OPENINFO  = "ORACLE_XA:Oracle_XA+Acc=P/scott/..."
    CLOSEINFO = " "
    TMSNAME   = "TMS_ORA"
  ORA_GRP2
    LMID      = SITE1
    GRPNO     = 5
    OPENINFO  = "ORACLE_XA:Oracle_XA+Acc=P/scott/..."
    CLOSEINFO = " "
    TMSNAME   = "TMS_ORA"

*SERVERS
  # By default, activate 2 instances of each server
  # and allow the administrator to activate up to 5
  # instances of each server
  DEFAULT:
    MIN      = 2
    MAX      = 5
```


Scaling by Replicating Server Processes and Server Groups

```
tellp_server
    SRVGRP = ORA_GRP1
    SRVID  = 10
    RESTART = N
tellp_server
    SRVGRP = ORA_GRP2
    SRVID  = 10
    RESTART = N
billp_server
    SRVGRP = APP_GRP1
    SRVID  = 10
    RESTART = N
billp_server
    SRVGRP = APP_GRP2
    SRVID  = 10
    RESTART = N
univp_server
    SRVGRP = ORA_GRP1
    SRVID  = 20
    RESTART = N
univp_server
    SRVGRP = ORA_GRP2
    SRVID  = 20
    RESTART = N
```

Scaling with Factory-based Routing

This topic includes the following sections:

- [About Factory-based Routing in the Production Application](#)
- [Configuring Factory-based Routing in the UBBCONFIG File](#)
- [Implementing Factory-based Routing in a Factory](#)
- [What Happens at Run Time](#)

This topic describes how the Production sample application was scaled using factory-based routing. For an introduction to factory-based routing, see [“Using Factory-based Routing \(CORBA Servers Only\)”](#) on page 1-11.

About Factory-based Routing in the Production Application

This section describes how the Production sample application uses a factory-based routing. For an introduction to this feature, see [“Using Factory-based Routing \(CORBA Servers Only\)”](#) on page 1-11.

You can use factory-based routing to expand the load-balancing and scalability features of BEA Tuxedo CORBA. In the Production sample application, you can use factory-based routing to send requests to register one subset of students to one machine, and requests for another subset of students to another machine. As you increase your application’s processing capability, you can easily modify the factory-based routing in your application to add more machines.

The primary design consideration regarding implementing factory-based routing in the Production sample application is in choosing the value on which routing is based. The Production sample application uses factory-based routing in the following ways:

- Requests from client applications to the Registrar object are routed based on the student ID. Requests from student ID 100001 to 100005 go to Production Machine 1. Requests from student ID 100006 to 100010 go to Production Machine 2.
- Requests from the Registrar object to the Teller object are routed based on account number. Billing requests for account 200010 to 200014 go to Production Machine 1. Billing requests for account 200015 to 200019 go to Production Machine 2.

Configuring Factory-based Routing in the UBBCONFIG File

The University Production sample application demonstrates how to implement factory-based routing. The `INTERFACES`, `ROUTING`, and `GROUPS` sections from the `ubb_b.nt` configuration file show how you can implement factory-based routing in a BEA Tuxedo CORBA application. You can find the `ubb_p.nt` or `ubb_p.mk` UBBCONFIG files for this sample in the directory where the BEA Tuxedo software is installed (see the `\samples\corba\university\production` subdirectory).

The UBBCONFIG file must specify the following data in the `INTERFACES` and `ROUTING` sections, as well as how groups and machines are identified.

1. The `INTERFACES` section lists the names of the interfaces for which you want to enable factory-based routing. For each interface, this section specifies the kinds of criteria on which the interface routes. This section specifies the routing criteria via an identifier, `FACTORYROUTING`, as shown in [Listing 2-2](#).

Listing 2-2 INTERFACES Section of a UBBCONFIG File

```
INTERFACES
  "IDL:beasys.com/UniversityP/Registrar:1.0"
    FACTORYROUTING = STU_ID
  "IDL:beasys.com/BillingP/Teller:1.0"
    FACTORYROUTING = ACT_NUM
```

[Listing 2-2](#) shows the fully qualified interface names for the two interfaces in the Production sample in which factory-based routing is used. The `FACTORYROUTING` identifier specifies the names of the routing values, which are `STU_ID` and `ACT_NUM`, respectively.

2. The `ROUTING` section specifies the parameters in [Table 2-1](#) for each routing value.

Table 2-1 Parameters Specified in the ROUTING Section

Parameter	Description
TYPE	Specifies the type of routing. In the Production sample, the type of routing is factory-based routing. Therefore, this parameter is defined as FACTORY.
FIELD	Specifies the variable name that the factory inserts in the routing value. In the Production sample, the field parameters are <code>student_id</code> and <code>account_number</code> , respectively.
FIELDTYPE	Specifies the data type of the routing value. In the Production sample, the field types for <code>student_id</code> and <code>account_number</code> are long.
RANGES	Specifies the values that are routed to each group.

[Listing 2-3](#) shows the ROUTING section of the UBBCONFIG file used in the Production sample application.

Listing 2-3 ROUTING Section of the UBBCONFIG File

```
ROUTING
  STU_ID
    FIELD      = "student_id"
    TYPE       = FACTORY
    FIELDTYPE  = LONG
    RANGES     = "100001-100005:ORA_GRP1,100006-100010:ORA_GRP2"
  ACT_NUM
    FIELD      = "account_number"
    TYPE       = FACTORY
    FIELDTYPE  = LONG
    RANGES     = "200010-200014:APP_GRP1,200015-200019:APP_GRP2"
```

[Listing 2-3](#) shows that `Registrar` object references for students with IDs in one range are routed to one server group, and `Registrar` object references for students with IDs in another range are routed to another group. Likewise, `Teller` object references for accounts

in one range are routed to one server group, and `Teller` object references for accounts in another range are routed to another group.

3. The groups specified by the `RANGES` identifier in the `ROUTING` section of the `UBBCONFIG` file need to be identified and configured. For example, the Production sample specifies four groups: `APP_GRP1`, `APP_GRP2`, `ORA_GRP1`, and `ORA_GRP2`. These groups need to be configured, and the machines on which they run need to be identified.

[Listing 2-4](#) shows the `GROUPS` section of the Production sample `UBBCONFIG` file, in which the `ORA_GRP1` and `ORA_GRP2` groups are configured. Notice how the names in the `GROUPS` section match the group names specified in the `RANGES` parameter in the `ROUTING` section. This is critical for factory-based routing to work correctly. Furthermore, any change in the way groups are configured in an application must be reflected in the `ROUTING` section. (Note that the Production sample packaged with the BEA Tuxedo software is configured to run entirely on one machine. However, you can easily configure this application to run on multiple machines.)

Listing 2-4 GROUPS Section of a UBBCONFIG File

```
*GROUPS
APP_GRP1
    LMID      = SITE1
    GRPNO     = 2
    TMSNAME   = TMS
APP_GRP2
    LMID      = SITE1
    GRPNO     = 3
    TMSNAME   = TMS
ORA_GRP1
    LMID      = SITE1
    GRPNO     = 4
    OPENINFO  =
"ORACLE_XA:Oracle_XA+Acc=P/scott/tiger+SesTm=100+LogDir=."+MaxCur=5"
    CLOSEINFO = " "
    TMSNAME   = "TMS_ORA"
ORA_GRP2
    LMID      = SITE1
    GRPNO     = 5
OPENINFO = "ORACLE_XA:Oracle_XA+Acc=P/scott/tiger+SesTm=100+LogDir=."+MaxCur=5"
    CLOSEINFO = " "
    TMSNAME   = "TMS_ORA"
```

Implementing Factory-based Routing in a Factory

Factories implement factory-based routing in the way the invocation to the `TP::create_object_reference()` operation is implemented. This operation has the C++ binding in [Listing 2-5](#).

Listing 2-5 C++ Binding for `create_object_reference`

```
CORBA::Object_ptr TP::create_object_reference (
    const char* interfaceName,
    const PortableServer::oid &stroid,
    CORBA::NVlist_ptr criteria);
```

The third parameter to this operation, `criteria`, specifies a list of named values to be used for factory-based routing. To implement factory-based routing in a factory, you need to build the `NVlist`. The use of factory-based routing is optional and is dependent on this argument. Instead of using factory-based routing, you can pass a value of 0 (zero) for this argument.

As stated previously, the `RegistrarFactory` object in the Production sample application specifies the value `STU_ID`. This value must exactly match the following information in the `UBBCONFIG` file:

- The routing name, type, and allowable values specified by the `FACTORYROUTING` identifier in the `INTERFACES` section.
- The routing criteria name, field, and field type specified in the `ROUTING` section.

The `RegistrarFactory` object inserts the student ID into the `NVlist` using the code shown in [Listing 2-6](#).

Listing 2-6 `NVlist` in the `RegistrarFactory` Object

```
// put the student id (which is the routing criteria)
// into a CORBA NVList:
CORBA::NVList_var v_criteria;
TP::orb()->create_list(1, v_criteria.out());
CORBA::Any any;
```

```
any <= (CORBA::Long)student;
v_criteria->add_value("student_id", any, 0);
```

The RegistrarFactory object has the invocation to the `TP::create_object_reference()` operation, shown in [Listing 2-7](#), passing the NVlist created in [Listing 2-6](#).

Listing 2-7 Invoking create_object_reference in the RegistrarFactory Object

```
// create the registrar object reference using
// the routing criteria :
CORBA::Object_var v_reg_oref =
    TP::create_object_reference(
        UniversityP::_tc_Registrar->id(),
        object_id,
        v_criteria.in()
    );
```

The Production sample application also uses factory-based routing in the TellerFactory object to determine the group in which Teller objects should be instantiated based on an account number.

What Happens at Run Time

When you implement factory-based routing in a factory, BEA Tuxedo CORBA generates an object reference. The following example shows how the client application gets an object reference to a Registrar object when factory-based routing is implemented.

1. The client application invokes the RegistrarFactory object, requesting a reference to a Registrar object. The request includes a student ID.
2. The RegistrarFactory inserts the student ID into an NVlist, which is used as the routing criteria.
3. The RegistrarFactory invokes the `TP::create_object_reference()` operation, passing the Registrar interface name, a unique OID, and the NVlist.

4. BEA Tuxedo CORBA compares the contents of the routing tables with the value in the `nvlist` to determine a group ID.
5. BEA Tuxedo CORBA inserts information about the group into the object reference.

When the client application subsequently invokes an object using the object reference, BEA Tuxedo CORBA routes the request to the group specified in the object reference.

Note: If you use the process-entity design pattern, you should use caution in how you implement factory-based routing. The object can service only those entities that are contained in the group's database.

Additional Design Considerations

This topic includes the following sections:

- [About the Additional Design Considerations](#)
- [Instantiating the Registrar and Teller Objects](#)
- [Ensuring That Student Registration Occurs in the Correct Server Group](#)
- [Ensuring That the Teller Object Is Instantiated in the Correct Server Group](#)

About the Additional Design Considerations

When designing the `Registrar` and `Teller` objects, you should ensure that:

- The `Registrar` and `Teller` objects work properly for the Production deployment environment; namely, across multiple replicated server processes and multiple groups. Given that the `University` and `Billing` server processes are replicated, the design must consider how these two objects should be instantiated.
- Client requests for registration and billing operations for a given student go to the correct server group, given that the two server groups in the Production BEA Tuxedo domain each deal with different databases.

These objects must have unique object IDs (OIDs) and must be method-bound (that is, they must have the `method` activation policy assigned to them).

Instantiating the Registrar and Teller Objects

In the `University` server applications that are less sophisticated than the Production sample application, the run-time behavior of the `Registrar` and `Teller` objects was simpler:

- Each object was process-bound, meaning that each was activated the first time it was invoked, and it stayed in memory until the server process in which it ran was shut down.
- Since there was only one server group running in the BEA Tuxedo domain, and only one University and Billing server process in the group, all client requests were directed to the same objects. As multiple client requests arrived in the BEA Tuxedo domain, these objects each processed one client request at one time.
- Because there was only one instance of each object in the server processes in which they ran, neither object needed a unique OID. The OID for each object specified only the Interface Repository ID.

However, because the University and Billing server processes are now replicated, BEA Tuxedo CORBA must be able to differentiate among multiple instances of the `Registrar` and `Teller` objects. For example, if there are two University server processes running in a group, BEA Tuxedo CORBA must have a means to distinguish between the `Registrar` object running in the first University server process and the `Registrar` object running in the second University server process. To distinguish multiple instances of these objects, each object instance must be unique.

To make each `Registrar` and `Teller` object unique, the factories for those objects must change the way in which they make object references to them. For example, when the `RegistrarFactory` object in the Basic sample application created an object reference to the `Registrar` object, the `TP::create_object_reference()` operation specified an OID that consisted only of the string `registrar`. However, in the Production sample application, the same `TP::create_object_reference()` operation uses a generated unique OID instead.

As a result of giving each `Registrar` and `Teller` object a unique OID, multiple instances of these objects may be running simultaneously in the BEA Tuxedo domain. This characteristic is typical of the stateless object model, and is an example of how the BEA Tuxedo domain can be highly scalable while it offers high performance.

Finally, because unique `Registrar` and `Teller` objects need to be brought into memory for each client request on them, it is critical that these objects be deactivated when the invocations on them are completed so that any object state associated with them does not remain idle in memory. The Production server application addresses this issue by assigning the `method` activation policy to these two objects in the Implementation Configuration File (ICF).

Ensuring That Student Registration Occurs in the Correct Server Group

The primary scalability advantage of using replicated server groups is being able to distribute processing across multiple machines. However, if your application interacts with a database, which is the case with the University sample applications, it is critical that you consider the impact of these multiple server groups on the database interactions.

In many cases, you may have one database associated with each machine in your deployment. If your server application is distributed across multiple machines, you must consider how you set up your databases.

The Production sample application, as described in this chapter, uses two databases. However, this application can easily be configured to accommodate more. The system administrator can decide on how many databases to use.

In the Production sample application, the student and account information is partitioned across the two databases, but course information is identical. Having identical course information in both databases is not a problem because the course information is read-only for the purposes of course registration. However, the student and account information is read-write. If multiple databases were also to contain identical data for students and accounts (that is, the database is not partitioned), the application would need to deal with the overhead of synchronizing the updates to student and account information across all the databases each time any student or account information were to change.

The Production sample application uses factory-based routing to send one set of requests to one machine, and another set to the other machine. How factory-based routing is implemented in the `RegistrarFactory` object depends on the way in which references to `Registrar` objects are created.

For example, when the client application sends a request to the `RegistrarFactory` object to get an object reference to a `Registrar` object, the client application includes a student ID in that request. The client application must use the object reference that the `RegistrarFactory` object returns to make all subsequent invocations on a `Registrar` object on a particular student's behalf, because the object reference returned by the factory is group-specific. Therefore, for example, when the client application subsequently invokes the `get_student_details()` operation on the `Registrar` object, the client application can be assured that the `Registrar` object is active in the server group associated with the database containing data for that student.

To show how this works, consider the following execution scenario, which is implemented in the Production sample application:

1. The client application invokes the `find_registrar()` operation on the `RegistrarFactory` object. Included in this invocation is the student ID 1000003.
2. BEA Tuxedo CORBA routes the client request to any `RegistrarFactory` object.
3. The `RegistrarFactory` object uses the student ID to create an object reference to a `Registrar` object in `ORA_GRP1`, based on the routing information in the `UBBCONFIG` file, and returns that object reference to the client application.
4. The client application invokes the `register_for_courses()` operation on the `Registrar` object.
5. BEA Tuxedo CORBA receives the client request and routes it to the server group specified in the object reference. In this case, the client request goes to the University server process in `ORA_GRP1`, which is on Production Machine 1.
6. The University server process instantiates a `Registrar` object and sends the client invocation to it.

The `RegistrarFactory` object from the preceding scenario returns to the client application a unique reference to a `Registrar` object that can be instantiated only in `ORA_GRP1`, which runs on Production Machine 1 and has a database containing student data for students with IDs in the range 100001 to 100005. Therefore, when the client application sends subsequent requests to this `Registrar` object on behalf of a given student, the `Registrar` object interacts with the correct database.

Ensuring That the Teller Object Is Instantiated in the Correct Server Group

When the `Registrar` object needs a `Teller` object, the `Registrar` object invokes the `TellerFactory` object, using the `TellerFactory` object reference cached in the `University Server` object.

However, because factory-based routing is used in the `TellerFactory` object, the `Registrar` object passes the student's account number when the `Registrar` object requests a reference to a `Teller` object. This way, the `TellerFactory` object creates a reference to a `Teller` object in the group that has the correct database.

Note: For the Production sample application to work properly, it is essential that the system administrator configures the server groups and the databases properly. In particular, the system administrator must make sure that a match exists between the routing criteria specified in the routing tables and the databases to which requests using those criteria are routed. Using the Production sample as an example, the database in a given group must

contain the correct student and account information for the requests that are routed to that group.

Scaling the Application Further

In the future, the system administrator of the Production sample application may want to add capacity to the BEA Tuxedo domain. For example, the University may eventually experience a large increase in the student population, or the Production application may be scaled up to accommodate the course registration process for an entire state university system, encompassing several campuses. This can be done without modifying or rebuilding the application.

The system administrator can continually add capacity by:

- Replicating the server groups in the Production sample application across additional machines.

The system administrator must modify the `UBBCONFIG` file to specify the additional server groups, the server processes that run in those groups, and the machines on which the server groups run.

- Changing the factory-based routing tables.

For example, instead of routing to the two existing groups in the Production sample application, the system administrator can modify the routing rules in the `UBBCONFIG` file to partition the application further among additional server groups added to the BEA Tuxedo domain. Any modification to the routing tables must match the information for the configured server groups and machines in the `UBBCONFIG` file.

Note: If you add capacity to an existing BEA Tuxedo CORBA application that uses a database, you must also consider the impact on how the database is set up, particularly when you are using factory-based routing. For example, if the Production sample application is distributed across six machines, the database on each machine must be set up appropriately and in accordance with the routing tables in the `UBBCONFIG` file.

Distributing CORBA Applications

This topic includes the following sections:

- [Why Distribute an Application?](#)
- [Using Data-dependent Routing \(BEA Tuxedo ATMI Servers Only\)](#)
- [Configuring the UBBCONFIG File](#)
- [Configuring the factory_finder.ini \(CORBA Applications Only\)](#)
- [Modifying the Domain Gateway Configuration File to Support Routing](#)

This topic describes how to distribute applications in the BEA Tuxedo CORBA environment, using a CORBA application as an example.

Notes: The BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. BEA Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

Why Distribute an Application?

This topic includes the following sections:

- [About Distributing an Application](#)
- [Benefits of a Distributed Application](#)
- [Characteristics of Distributing an Application](#)

About Distributing an Application

Distributing an application enables you to select which parts of an application should be grouped together logically and where these groups should run. You distribute an application by creating more than one entry in the `GROUPS` section of the `UBBCONFIG` file, and by dividing application resources or tasks among the groups. Creating groups of servers enables you to partition a very large application into its component business applications, and to assure that each of these into logical components is of a manageable size and in an optimal location.

Benefits of a Distributed Application

The benefits of a distributed application include:

- Scalability—to increase the load that an application can sustain:
 - Place extra server processes in a group.
 - Add machines to the application and redistribute the groups across the machines.
 - Replicate a group onto other machines within the application and use load balancing.
 - Segment a database and use data-dependent routing to reach the groups dealing with these separate database segments (the BEA Tuxedo ATMI system).

With the BEA Tuxedo CORBA system, you can use factory-based routing to distribute the processing of a particular CORBA interface across multiple server groups and, if desired, across multiple machines. This feature allows you to distribute the processing load, which can prevent the processing bottlenecks that occur when concurrent, resource-intensive applications compete for the available CPU, memory, disk I/O, and network resources. For an example of using factory-based routing, see [“Scaling with Factory-based Routing” on page 2-10](#).

For more information about BEA Tuxedo CORBA scalability features, see [Chapter 1, “Scaling BEA Tuxedo CORBA Applications.”](#)

- Ease of Development and Maintenance—the separation of the business application logic into services or components that communicate through well-defined messages or interfaces allows both development and maintenance to be similarly separated and thereby simplified.
- Reliability—when multiple machines are in use and one fails, the remainder can continue operation. Similarly, when multiple server processes are within a group and one fails, the others are available to perform work. Finally, if a machine should fail, but there are multiple machines within the application, these other machines can be used to handle the load.
- Coordination of Autonomous Actions—if you have separate applications, you can coordinate autonomous actions, as a single logical *unit of work*, among applications. *Autonomous actions* are actions that involve multiple server groups and multiple resource manager interfaces.

Characteristics of Distributing an Application

A distributed application:

- Enlarges the client and/or server model.
- Establishes multiple server groups.
- Enables transparent access to BEA Tuxedo services or BEA Tuxedo CORBA interfaces.
- In BEA Tuxedo, allows data-dependent partitioning of data.
- In BEA Tuxedo CORBA, allows partitioning of CORBA objects in multiple groups across multiple machines, or distributing application factory interfaces and application interfaces.
- Enables management of multiple resources.
- Supports a networked model.

Using Data-dependent Routing (BEA Tuxedo ATMI Servers Only)

This topic includes the following sections:

- [About Data-dependent Routing](#)
- [Characteristics of Data-dependent Routing](#)
- [Sample Distributed Application](#)

- [Example of UBBCONFIG Sections in a Distributed Application](#)

Note: This topic applies to BEA Tuxedo servers only.

About Data-dependent Routing

Data-dependent routing is a mechanism whereby a service request is routed by a client (or a server acting as a client) to a server within a specific group based on a data value contained within the buffer that is sent. Within the internal code of a service call, BEA Tuxedo chooses a destination server by comparing a data field with the routing criteria it finds in the bulletin board shared memory.

For any given service, a routing criteria identifier can be specified in the `SERVICES` section of the `UBBCONFIG` file. The routing criteria identifier (in particular, the mapping of data ranges to server groups) is specified in the `ROUTING` section.

Characteristics of Data-dependent Routing

Data-dependent routing has the following characteristics:

- The service request assigned to a server in the group is based on a data value.
- Routing uses the bulletin board criteria and occurs in a server call.
- The routing criteria identifier for a service is specified in the `SERVICES` section of the `UBBCONFIG` file.
- The routing criteria identifier is defined in the `ROUTING` section of the `UBBCONFIG` file.

Sample Distributed Application

[Table 3-1](#) illustrates how client requests are routed to servers. In this example, a banking application called `bankapp` uses data-dependent routing. For `bankapp`, there are three groups (`BANKB1`, `BANKB2`, and `BANKB3`), 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.

Table 3-1 Data-dependent Routing Criteria for Sample Distributed Application

Server Group	Routing Criteria	Services
BANKB1	Account_ID: 10000 - 49999	WITHDRAW, DEPOSIT, and INQUIRY
	Branch_ID: 1 - 4	OPEN and CLOSE
BANKB2	Account_ID: 50000 - 79999	WITHDRAW, DEPOSIT, and INQUIRY
	Branch_ID: 5 - 7	OPEN and CLOSE
BANKB3	Account_ID: 80000 - 109999	WITHDRAW, DEPOSIT, and INQUIRY
	Branch_ID: 8 - 10	OPEN and CLOSE

Example of UBBCONFIG Sections in a Distributed Application

[Listing 3-1](#) shows a sample UBBCONFIG file that contains the GROUPS, SERVICES, and ROUTING sections of a configuration file to accomplish data-dependent routing in the BEA Tuxedo system.

Listing 3-1 Sample UBBCONFIG File

```
*GROUPS
BANKB1          GRPNO=1
BANKB2          GRPNO=2
BANKB3          GRPNO=3
#
*SERVICES
WITHDRAW        ROUTING=ACCOUNT_ID
DEPOSIT         ROUTING=ACCOUNT_ID
INQUIRY         ROUTING=ACCOUNT_ID
OPEN_ACCT       ROUTING=BRANCH_ID
CLOSE_ACCT      ROUTING=BRANCH_ID
#
```

```

*ROUTING
ACCOUNT_ID      FIELD=ACCOUNT_ID BUFTYPE="FML"
                 RANGES="MIN - 9999:*,
                        10000-49999:BANKB1,
                        50000-79999:BANKB2,
                        80000-109999:BANKB3,
                        *: *"
BRANCH_ID       FIELD=BRANCH_ID BUFTYPE="FML"
                 RANGES="MIN - 0:*,
                        1-4:BANKB1,
                        5-7:BANKB2,
                        8-10:BANKB3,
                        *: *"

```

Configuring the UBBCONFIG File

This topic includes the following sections:

- [About the UBBCONFIG File in Distributed Applications](#)
- [Modifying the GROUPS Section](#)
- [Modifying the SERVICES Section](#)
- [Modifying the INTERFACES Section](#)
- [Creating the ROUTING Section](#)

For more information about the UBBCONFIG file, see “Creating a Configuration File” in [Setting Up a BEA Tuxedo Application](#).

About the UBBCONFIG File in Distributed Applications

The UBBCONFIG file contains a description of either data-dependent routing (BEA Tuxedo) or factory-based routing (BEA Tuxedo CORBA), as follows:

- The GROUPS section is populated with as many server groups as are required for distributing the system. This allows the system to route a request to a server in a specific group. These groups can all reside on the same site (SHM mode) or, if there is networking, the groups can reside on different sites (MP mode).

- For data-dependent routing in BEA Tuxedo, the `SERVICES` section must list the routing criteria for each service that uses the `ROUTING` parameter.

Note: If a service has multiple entries, each with a different `SRVGRP` parameter, all such entries must set `ROUTING` the same way to ensure consistency for that service. A service can route only on one field, which must be the same for all the same services.

- For factory-based routing in BEA Tuxedo CORBA, the `INTERFACES` section must list the name of the routing criteria for each CORBA interface that uses the `FACTORYROUTING` parameter. This parameter is set to the name of a routing criteria defined in the `ROUTING` section.
- Add a `ROUTING` section to the configuration file to show mappings between data ranges and groups so that the system can send the request to a server in a specific group. Each `ROUTING` section item contains an identifier that is used in the `INTERFACES` section (for BEA Tuxedo ATMI) or in the `SERVICES` section (for BEA Tuxedo).

Modifying the GROUPS Section

The parameters in the `GROUPS` section implement two important aspects of distributed transaction processing:

- They associate a group of servers with a particular `LMID` and a particular instance of a resource manager.
- By allowing a second `LMID` to be associated with the server group, they name an alternate machine to which a group of servers can be migrated if the `MIGRATE` option is specified.

[Table 3-2](#) describes the parameters in the `GROUPS` section.

Table 3-2 Parameters Specified in the GROUPS Section

Parameter	Meaning
LMID	LMID must be assigned in the MACHINES section to indicate that this server group runs on this particular machine. A second LMID value can be specified (separated from the first by a comma) to name an alternate machine to which this server group can be migrated if the MIGRATE option has been specified. Servers in the group must specify RESTART=Y to migrate.
GRPNO	Associates a numeric group number with this server group. The number must be greater than zero (0) and less than 30000. It must be unique among entries in the GROUPS section in this configuration file. (Required)
TMSNAME	Specifies which transaction management server (TMS) should be associated with this server group.
TMSCOUNT	Specifies how many copies of TMSNAME should be started for this server group. The minimum value is 2. If not specified, the default is 3. All TMSNAME servers started for a server group are automatically set up in an MSSQ set. (Optional)

Table 3-2 Parameters Specified in the GROUPS Section (Continued)

Parameter	Meaning
OPENINFO	<p>Specifies information needed to open a particular instance of a particular resource manager, or it indicates that such information is not required for this server group. When a resource manager is named in the OPENINFO parameter, information such as the name of the database and the access mode is included. The entire value string must be enclosed in double quotes and must not be more than 256 characters. The format of the OPENINFO string is dependent on the requirements of the vendor providing the underlying resource manager. The string required by the vendor must be prefixed with <code>rm_name:</code>, which is the published name of the vendor's transaction (XA) interface followed immediately by a colon (:).</p> <p>The OPENINFO parameter is ignored if TMSNAME is not set or is set to TMS. If TMSNAME is set but the OPENINFO string is set to the null string (" ") or if this parameter does not appear on the entry, it means that a resource manager exists for the group but does not require any information for executing an open operation.</p>
CLOSEINFO	<p>Specifies information the resource manager needs when closing a database. The parameter can be omitted or the null string can be specified. The default is the null string.</p>

Modifying the SERVICES Section

The `SERVICES` section contains parameters that control the way application services are handled. An entry line in this section is associated with a service by its identifier name. Because the same service can be link edited with more than one server, the `SRVGRP` parameter is provided to tie the parameters for an instance of a service to a particular group of servers.

Parameters to Modify

Two parameters in the `SERVICES` section are particularly related to distributed transaction processing (DTP) for BEA Tuxedo CORBA applications that use BEA Tuxedo ATMI services: `AUTOTRAN`, and `TRANTIME`.

[Table 3-4](#) describes the parameters in the `SERVICES` section.

Table 3-3 Parameters Specified in the SERVICES Section

Parameter	Meaning
AUTOTRAN	Determines whether a transaction should be started automatically if a message received by this service is not already in transaction mode. The default is N. Use of the parameter should be coordinated with the programmers that code the services for your application.
TRANTIME	Specifies a timeout value, in seconds, for transactions automatically started in this service. The default is 30 seconds. Required only if AUTOTRAN=Y and another timeout value is needed.

Sample SERVICES Section

[Listing 3-2](#) shows a sample SERVICES section.

Listing 3-2 Production Sample SERVICES Section

```
*SERVICES
# Publish Tuxedo Teller application services
#
DEBIT
    AUTOTRAN=Y
CREDIT
    AUTOTRAN=Y
CURRBALANCE
    AUTOTRAN=Y
```

Modifying the INTERFACES Section

The INTERFACES section contains parameters that control the way application interfaces are handled. An entry line in this section is associated with an interface by its identifier name. Because the same interface can be link edited with more than one server, the SRVGRP parameter is provided to tie the parameters for an instance of a interface to a particular group of servers.

Parameters to Modify

Three parameters in the `INTERFACES` section are particularly related to distributed transaction processing (DTP): `FACTORYROUTING`, `AUTOTRAN`, and `TRANTIME`.

[Table 3-4](#) describes the parameters in the `INTERFACES` section.

Table 3-4 Parameters Specified in the `INTERFACES` Section

Parameter	Meaning
<code>FACTORYROUTING = criterion-name</code>	Specifies the name of the routing criteria to be used for factory-based routing for this BEA Tuxedo CORBA interface. You must specify a <code>FACTORYROUTING</code> parameter for interfaces requesting factory-based routing.
<code>AUTOTRAN</code>	Determines whether a transaction should be started automatically if a message received by this interface is not already in transaction mode. The default is <code>N</code> . Use of this parameter should be coordinated with the programmers that code the interface for your application so that it matches the setting of the <code>transaction policy</code> option in the application's ICF file.
<code>TRANTIME</code>	Specifies a timeout value, in seconds, for transactions automatically started in this interface. The default is 30 seconds. Required only if <code>AUTOTRAN=Y</code> and a timeout value other than the default is needed.
<code>LOAD = number</code>	Specifies an arbitrary number between 1 and 100 that represents the relative load that the CORBA interface is expected to impose on the system. The numbering scheme is relative to the <code>LOAD</code> numbers assigned to other CORBA interfaces used by this application. The default is 50. This number is used by the BEA Tuxedo system to select the best server to which to route the request

Sample `INTERFACES` Section

[Listing 3-2](#) shows a sample `INTERFACES` section.

Listing 3-3 Sample `INTERFACES` Section

```
*INTERFACES
```

```
"IDL:beasys.com/UniversityP/Registrar:1.0"
    FACTORYROUTING = STU_ID
    AUTOTRAN=Y
    TRANTIME=50

"IDL:beasys.com/BillingP/Teller:1.0"
    FACTORYROUTING = ACT_NUM
    AUTOTRAN=Y
```

Creating the ROUTING Section

For information about ROUTING parameters that support BEA Tuxedo data-dependent routing or BEA Tuxedo CORBA factory-based routing, see “Creating a Configuration File” in [Setting Up a BEA Tuxedo Application](#).

[Listing 3-4](#) shows the ROUTING section of the UBBCONFIG file used in the Production sample application for factory-based routing.

Listing 3-4 Production Sample ROUTING Section

```
*ROUTING

STU_ID
    FIELD      = "student_id"
    TYPE       = FACTORY
    FIELDTYPE  = LONG
    RANGES     = "100001-100005:ORA_GRP1,100006-100010:ORA_GRP2"

ACT_NUM
    FIELD      = "account_number"
    TYPE       = FACTORY
    FIELDTYPE  = LONG
    RANGES     = "200010-200014:APP_GRP1,200015-200019:APP_GRP2"
```

Configuring the factory_finder.ini (CORBA Applications Only)

For CORBA applications, to configure factory-based routing across multiple domains, you must configure the `factory_finder.ini` file to identify factory objects that are used in the current (local) domain but that are resident in a different (remote) domain. For more information, see “Configuring Multiple Domains Multiple Domains for CORBA Applications in [Using the BEA Tuxedo Domains Component](#).

Modifying the Domain Gateway Configuration File to Support Routing

This topic includes the following sections:

- [About the Domain Gateway Configuration File](#)
- [Parameters in the DM_ROUTING Section of the DMCONFIG File \(BEA Tuxedo ATMI Only\)](#)

This section is specific to BEA Tuxedo and explains how and why you need to modify the domain gateway configuration to support routing. For more information about the domain gateway configuration file, see “Configuring Multiple Domains Multiple Domains for CORBA Applications in [Using the BEA Tuxedo Domains Component](#).

About the Domain Gateway Configuration File

The Domain gateway configuration information is stored in a binary file called `BDMCONFIG`. The `DMCONFIG` file (ASCII) is created and edited with any text editor. The compiled `BDMCONFIG` file can be updated while the system is running by using the `dmadmin(1)` command.

You must have one `BDMCONFIG` file for each BEA Tuxedo application that requires the Domains functionality. System access to the `BDMCONFIG` file is provided through the Domains administrative server, `DMADM(5)`. When a gateway group is booted, the gateway administrative server, `GWADM(5)`, requests from the `DMADM` server a copy of the configuration required by that group. The `GWADM` server and the `DMADM` server also ensure that run-time changes to the configuration are reflected in the corresponding Domain gateway groups.

Note: For more information about modifying the `DMCONFIG` file, see “Configuring Multiple Domains Multiple Domains for CORBA Applications” in [Using the BEA Tuxedo Domains Component](#).

Parameters in the DM_ROUTING Section of the DMCONFIG File (BEA Tuxedo ATMI Only)

The DM_ROUTING section provides information for data-dependent routing of service requests using FML, VIEW, X_C_TYPE, and X_COMMON typed buffers. Lines within the DM_ROUTING section have the form CRITERION_NAME, where CRITERION_NAME is the (identifier) name of the routing entry specified in the SERVICES section. The CRITERION_NAME entry may contain no more than 15 characters.

Parameters to Specify

[Table 3-5](#) describes the parameters in the DM_ROUTING section.

Table 3-5 Parameters Specified in the DM_ROUTING Section

Parameter	Description
<i>FIELD = identifier</i>	Specifies the name of the routing field. It must contain 30 characters or fewer. This field is assumed to be a field name identified in an FML field table (for FML buffers) or an FML VIEW table (for VIEW, X_C_TYPE, or X_COMMON buffers). The FLDTBLDIR and FIELDTBLS environment variables are used to locate FML field tables; the VIEWDIR and VIEWFILES environment variables are used to locate FML VIEW tables. If a field in an FML32 buffer is used for routing, it must have a field number less than or equal to 8191.

Table 3-5 Parameters Specified in the DM_ROUTING Section (Continued)

Parameter	Description
BUFTYPE = "type1[:subtype1[, subtype2 . . .]][:type2[:subtype3[, . . .]]] . . ."	<p>Specifies list of types and subtypes of data buffers for which this routing entry is valid. The types are restricted to FML, VIEW, X_C_TYPE, and X_COMMON.</p> <p>No subtype can be specified for type FML, and subtypes are required for the other types (* is not allowed).</p> <p>Duplicate type/subtype pairs cannot be specified for the same routing criteria name; more than one routing entry can have the same criteria name as long as the type/subtype pairs are unique. This parameter is required.</p> <p>If multiple buffer types are specified for a single routing entry, the data types of the routing field for each buffer type must be the same. (If the field value is not set (for FML buffers), or does not match any specific range, and a wildcard range has not been specified, then an error is returned to the application process that requested the execution of the remote service.) No routing is allowed on CORBA and EJB (TGIOP is not a valid buffer type).</p>

Table 3-5 Parameters Specified in the DM_ROUTING Section (Continued)

Parameter	Description
<code>RANGES</code> <code>= "range1:rdom1[,range2:rdom2 ...]"</code>	<p>Specifies the ranges and associated remote domain names (RDOM) for the routing field. The string must be enclosed in double quotes, with the format of a comma-separated ordered list of range/RDOM pairs.</p> <p>A range is either a single value (signed numeric value or character string in single quotes), or a range of the form <i>lower - upper</i> (where lower and upper are both signed numeric values or character strings in single quotes). The value of <i>lower</i> must be less than or equal to <i>upper</i>. A single quote embedded in a character string value (such as "O'Brien"), must be preceded by two backslashes ("O\\'Brien").</p> <ul style="list-style-type: none"> • Use MIN to indicate the minimum value for the data type of the associated FIELD. For strings and arrays, it is the null string; for character fields, it is 0; for numeric values, it is the minimum numeric value that can be stored in the field. • Use MAX to indicate the maximum value for the data type of the associated FIELD. For strings and arrays, it is effectively an unlimited string of octal-255 characters; for a character field, it is a single octal-255 character; for numeric values, it is the maximum numeric value that can be stored in the field. <p>Thus, MIN - -5 is all numbers less than or equal to -5, and 6 - MAX is all numbers greater than or equal to 6.</p> <p>The metacharacter * (wildcard) in the position of a range indicates any values not covered by the other ranges previously seen in the entry. Only one wildcard range is allowed per entry and it should be last (ranges following it are ignored).</p>

Routing Field Description

The routing field can be of any data type supported in FML or VIEW. A numeric routing field must have numeric range values, and a string routing field must have string range values.

String range values for string, array, and character field types must be placed inside a pair of single quotation marks and cannot be preceded by a sign. Short and long integer values are a string of digits, optionally preceded by a plus (+) or minus (-) sign. Floating point numbers are of the form accepted by the C compiler or atof(): an optional sign, followed by a string of digits optionally containing a decimal point, and an optional e or E followed by an optional sign or space, and an integer.

When a field value matches a range, the associated RDOM value specifies the remote domain to which the request should be routed. An RDOM value of * indicates that the request can go to any remote domain known by the gateway group. Within a range/RDOM pair, the range is separated from the RDOM by a colon (:).

Example of a Five-Site Domain Configuration Using Routing

[Listing 3-5](#) shows a configuration file that defines a five-site domain configuration. It has four bank branch domains communicating with a Central Bank Branch. Three of the bank branches run within other BEA Tuxedo system domains. The fourth branch runs under the control of another TP domain, and OSI-TP is used in the communication with that domain. The example shows the BEA Tuxedo Domain gateway configuration file from the Central Bank point of view. In the DM_TDOMAIN section, this example shows a mirrored gateway for b01.

Listing 3-5 DMCONFIG File for a Five-Site Domains Configuration

```
# BEA TUXEDO DOMAIN CONFIGURATION FILE FOR THE CENTRAL BANK
#
#
*DM_LOCAL_DOMAINS
# <local domain name> <Gateway Group name> <domain type> <domain id> <log device>
#                               [<audit log>] [<blocktime>]
#                               [<log name>] [<log offset>] [<log size>]
#                               [<maxrdom>] [<maxrdtran>] [<maxtran>]
#                               [<maxdatalen>] [<security>]
#                               [<tuxconfig>] [<tuxoffset>]
#
#
#
DEFAULT: SECURITY = NONE
c01      GWGRP = bankg1
         TYPE = TDOMAIN
         DOMAINID = "BA.CENTRAL01"
         DMTLOGDEV = "/usr/apps/bank/DMTLOG"
         DMTLOGNAME = "DMTLG_C01"
c02      GWGRP = bankg2
         TYPE = OSITP
         DOMAINID = "BA.CENTRAL01"
         DMTLOGDEV = "/usr/apps/bank/DMTLOG"
         DMTLOGNAME = "DMTLG_C02"
         NWDEVICE = "OSITP"
         URCH = "ABCD"
#
*DM_REMOTE_DOMAINS
```

```

#<remote domain name> <domain type> <domain id>
#
b01      TYPE = TDOMAIN
          DOMAINID = "BA.BANK01"
b02      TYPE = TDOMAIN
          DOMAINID = "BA.BANK02"
b03      TYPE = TDOMAIN
          DOMAINID = "BA.BANK03"
b04      TYPE = OSITP
          DOMAINID = "BA.BANK04"
          URCH = "ABCD"

#
*DM_TDOMAIN
#
#      <local or remote domainname> <network address> [nwdevice]
#
# Local network addresses
c01      NWADDR = "//newyork.acme.com:65432"      NWDEVICE = "/dev/tcp"
c02      NWADDR = "//192.76.7.47:65433"          NWDEVICE = "/dev/tcp"
# Remote network addresses: second b01 specifies a mirrored gateway
b01      NWADDR = "//192.11.109.5:1025" NWDEVICE = "/dev/tcp"
b01      NWADDR = "//194.12.110.5:1025" NWDEVICE = "/dev/tcp"
b02      NWADDR = "//dallas.acme.com:65432" NWDEVICE = "/dev/tcp"
b03      NWADDR = "//192.11.109.156:4244" NWDEVICE = "/dev/tcp"
#
*DM_OSITP
#
#<local or remote domain name> <apt> <aeq>
#                                     [<aet>] [<acn>] [<apid>] [<aeid>]
#                                     [<profile>]
#
c02      APT = "BA.CENTRAL01"
          AEQ = "TUXEDO.R.4.2.1"
          AET = "{1.3.15.0.3},{1}"
          ACN = "XATMI"
b04      APT = "BA.BANK04"
          AEQ = "TUXEDO.R.4.2.1"
          AET = "{1.3.15.0.4},{1}"
          ACN = "XATMI"
*DM_LOCAL_SERVICES
#<service_name> [<Local Domain name>] [<access control>] [<exported svcname>]
#               [<inbuftype>] [<outbuftype>]
#
open_act      ACL = branch
close_act     ACL = branch
credit
debit
balance
loan          LDOM = c02      ACL = loans

```


Modifying the Domain Gateway Configuration File to Support Routing

```
*DM_REMOTE_SERVICES
#<service_name>    [<Remote domain name>] [<local domain name>]
#                  [<remote svcname>] [<routing>] [<conv>]
#                  [<trantime>] [<inbuftype>] [<outbuftype>]
#
tlr_add    LDOM = c01  ROUTING = ACCOUNT
tlr_bal    LDOM = c01  ROUTING = ACCOUNT
tlr_add    RDOM = b04  LDOM = c02 RNAME = "TPSU002"
tlr_bal    RDOM = b04  LDOM = c02 RNAME = "TPSU003"
*DM_ROUTING
# <routing criteria>    <field> <typed buffer> <ranges>
#
ACCOUNT FIELD = branchid BUFTYPE = "VIEW:account"
          RANGES = "MIN - 1000:b01, 1001-3000:b02, *:b03"
*DM_ACCESS_CONTROL
#<acl name>    <Remote domain list>
#
branch  ACLIST = b01, b02, b03
loans   ACLIST = b04
```

Tuning CORBA Applications

This topic includes the following sections:

- [Maximizing Application Resources](#)
- [When to Use MSSQ Sets \(BEA Tuxedo ATMI Servers Only\)](#)
- [Enabling System-controlled Load Balancing](#)
- [Configuring Replicated Server Processes and Groups](#)
- [Configuring Multithreaded Servers](#)
- [Bundling Services into Servers \(BEA Tuxedo ATMI Servers Only\)](#)
- [Performance Options](#)
- [Enhancing Efficiency with Application Parameters](#)
- [Setting Application Parameters](#)
- [Determining IPC Requirements](#)
- [Measuring System Traffic](#)

For more information about monitoring BEA Tuxedo applications, see “Monitoring a Running System” in the [Administering a BEA Tuxedo Application at Run Time](#).

Notes: The BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB text references, associated code

samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. BEA Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

Maximizing Application Resources

Making correct decisions in the following areas can improve the functioning of your BEA Tuxedo applications:

- [When to Use MSSQ Sets \(BEA Tuxedo ATMI Servers Only\)](#).
- How to assign load factors.
- How to package interfaces and/or services into servers.
- How to set application parameters.
- How to tune operating system IPC parameters.
- How to detect and eliminate bottlenecks.

When to Use MSSQ Sets (BEA Tuxedo ATMI Servers Only)

Note: Multiple Servers, Single Queue (MSSQ) sets are not supported in BEA Tuxedo CORBA servers.

[Table 4-1](#) describes when to use MSSQ sets with BEA Tuxedo servers.

Table 4-1 When and When Not to Use MSSQ Sets

Use MSSQ Sets When	Do Not Use MSSQ Sets When
There are several, but not too many servers.	There is a large number of servers. (A compromise is to use many MSSQ sets.)
Buffer sizes are not too large.	Buffer sizes are large enough to exhaust one queue.
The servers offer identical sets of services.	Services are different for each server.
The messages involved are reasonably sized.	Long messages are being passed to the services causing the queue to be exhausted. This causes nonblocking sends to fail, or blocking sends to block.
Optimization and consistency of service turnaround time is paramount.	Optimization and consistency of service turnaround time is not critical.

The following two analogies help to show why using MSSQ sets is sometimes, but not always, beneficial:

- An application in which MSSQ sets are used appropriately is similar to a bank, where all the tellers offer the same services and customers wait in line for the first available teller. This efficient arrangement ensures the best use of available services.
- An application in which it is better to avoid using MSSQ sets is similar to a supermarket, where each cashier offers a different set of services: some accept cash only, some accept credit cards, and still others serve only customers buying fewer than ten items.

Enabling System-controlled Load Balancing

You can control whether a load-balancing algorithm is used on the BEA Tuxedo system as a whole. When load balancing is used, a load factor is applied to each service within the system, allowing you to track the total load on every server. Every service request is sent to the qualified server that is least loaded.

Note: On BEA Tuxedo CORBA systems, system-controlled load balancing is enabled automatically. You *cannot* disable load balancing by specifying `LDBAL=N`.

To determine how to assign load factors (located in the `SERVICES` section), run an application continually and calculate the average time it takes for each service to be performed. Assign a `LOAD` value of 50 (`LOAD=50`) to any service that requires the average amount of time that you calculated. Any service taking longer to execute than the calculated average should have a `LOAD>50`. Any service taking less to execute than the calculated average should have a `LOAD<50`.

A `LOAD` factor is assigned to each service performed, which keeps track of the total load of services that each server has performed. Each service request is routed to the server with the smallest total load. The routing of that request causes the server's total to be increased by the `LOAD` factor of the service requested.

You can also apply `LOAD` factors to interfaces. For more information about `LOAD` factors, see “Creating a Configuration File” in the [Administering a BEA Tuxedo Application at Run Time](#).

Configuring Replicated Server Processes and Groups

To configure replicated server processes and groups in the BEA Tuxedo domain, complete the following steps:

1. Edit the application's `UBBCONFIG` file using a text editor.
2. In the `GROUPS` section, specify the names of the groups you want to configure.
3. In the `SERVERS` section, specify the parameters in [Table 4-2](#) for the server process you want to replicate.

Table 4-2 Parameters Specified in the SERVERS Section

Parameter	Description
Server application name	Specifies the name of the executable file that contains the application server.
GROUP	Specifies the name of the group to which the server process belongs. If you are replicating a server process across multiple groups, specify the server process once for each group.
SRVID	Specifies a numeric identifier, giving the server process a unique identity.
MIN	Specifies the number of instances of the server process to start when you start the application.
MAX	Specifies the maximum number of server processes that can be running at any one time.

The `MIN` and `MAX` parameters determine the degree to which a given server application can process requests on a given interface in parallel. During run time, the system administrator can examine resource bottlenecks and start additional server processes, if necessary, thereby scaling the application. For more information, see “Monitoring a Running Application” in the [Administering a BEA Tuxedo Application at Run Time](#).

Note: The `MAX` parameter controls the maximum number of instances. However, BEA Tuxedo does not spawn instances automatically. The system will automatically start up to the specified `MIN` number of instances. Between `MIN` and `MAX`, the system administrator will need to spawn new instances manually. Once `MAX` is reached, an error will be returned by `tmboot`, `tmadmin`, or the `TMIB` API.

Configuring Multithreaded Servers

This topic includes the following sections:

- [Setting the OPENINFO Parameter for Database Interoperation](#)
- [Parameters Used to Configure Multithreaded Servers](#)
- [Assigning Priorities to Interfaces](#)

For more information about multithreaded servers, see [“Using Multithreaded Servers” on page 1-9](#).

Setting the OPENINFO Parameter for Database Interoperation

To enable the use of threads by a multithreaded server when interoperating with the Oracle XA database software, you must add `Threads=true` to the `OPENINFO` parameter in the `GROUPS` section of the `UBBCONFIG` file, as shown in [Listing 4-1](#). For more information, see the Oracle XA online documentation.

Listing 4-1 Adding `Threads=true` to the `OPENINFO` Parameter

```
OPENINFO="ORACLE_XA:Oracle_XA+Acc=P/scott/tiger+SesTm=100+LogDir=."+MaxCur=
5+Threads=true"
```

Parameters Used to Configure Multithreaded Servers

The following parameters are used to configure multithreaded CORBA servers. These parameters are set in the `UBBCONFIG` file:

- `MAXOBJECTS`

Note: While the `MAXOBJECTS` parameter does not specifically apply to threads, you may want to increase this parameter because multithreaded applications have the potential to activate more objects at any point in time than single-threaded applications.

- `MAXACCESSERS`

- `MAXDISPATCHTHREADS`

- `MINDISPATCHTHREADS`

- `THREADSTACKSIZE`

- `CONCURR_STRATEGY`

For a description how to set these parameters, see the following topics:

- “Creating the Configuration File” and “How to Configure the BEA Tuxedo System to Take Advantage of Threads” in [Setting Up a BEA Tuxedo Application](#).
- “Setting the MAXACCESSERS, MAXSERVERS, MAXINTERFACES, and MAXSERVICES Parameters” in [Administering a BEA Tuxedo Application at Run Time](#).

Assigning Priorities to Interfaces

This topic includes the following sections:

- [About Priorities to Interfaces](#)
- [Characteristics of the PRIO Parameter](#)

About Priorities to Interfaces

You can exert significant control over the flow of data in an application by assigning priorities to BEA Tuxedo Interfaces using the `PRIO` parameter. For a CORBA application running on a BEA Tuxedo system, you can specify the `PRIO` parameter for each interface named in the `INTERFACES` section of the application’s `UBBCONFIG` file.

For example, Server 1 offers Interfaces A, B, and C. Interfaces A and B have a priority of 50 and Interface C has a priority of 70. An interface requested for C is always dequeued before a request for A or B. Requests for A and B are dequeued equally with respect to one another. The system dequeues every tenth request in first in first out (FIFO) order to prevent a message from waiting indefinitely on the queue.

You can also dynamically change a priority with the `tpsprio()` call. Only preferred clients should be able to increase the interface priority. In a system on which servers perform interface request, the server can call `tpsprio()` to increase the priority of its interface so the user does not wait in line for every interface request that is required.

Characteristics of the PRIO Parameter

The `PRIO` parameter should be used carefully. Depending on the order of messages on the queue (for example, A, B, and C), some (such as A and B) will be dequeued only one in ten times. This means reduced performance and potential slow turnaround time on the service.

The characteristics of the `PRIO` parameter are as follows:

- It determines the priority of an interface on the server’s queue.
- The highest assigned priority gets first preference. This interface should occur less frequently.

- A lower priority message does not remain forever enqueued, because every tenth message is retrieved on a FIFO basis. Response time should not be a concern of the lower priority interface.

Assigning priorities enables you to provide more efficient service to the most important requests and slower service to the less important requests. You can also give priority to specific users or in specific circumstances.

Bundling Services into Servers (BEA Tuxedo ATMI Servers Only)

This topic includes the following sections:

- [About Bundling Services](#)
- [When to Bundle Services](#)

About Bundling Services

The easiest way to package services into server executables is to not package them at all. Unfortunately, if you do not package services, the number of server executables, and also message queues and semaphores, rises beyond an acceptable level. There is a trade-off between not bundling services and bundling services too much.

When to Bundle Services

You should bundle services for the following reasons:

- Functional similarity—if some services are similar in their role in the application, you can bundle them in the same server. The application can offer all or none of them at a given time. An example is the `bankapp` application, in which the `WITHDRAW`, `DEPOSIT`, and `INQUIRY` services are all teller operations. Administration of services becomes simpler.
- Similar libraries—for example, if you have three services that use the same 100K library and three services that use different 100K libraries, bundling the first three services saves 200K. Often, functionally equivalent services have similar libraries.
- Filling the queue—bundle only as many services into a server as the queue can handle. Each service added to an unfilled MSSQ set may add relatively little to the size of an executable, and nothing to the number of queues in the system. Once the queue is filled, however, the system performance degrades and you must create more executables to compensate.

- Placement of call-dependent services—avoid placing, in the same server, two (or more) services that call each other. If you do so, the server will issue a call to itself, causing a deadlock.

Performance Options

Performance options were added to BEA Tuxedo in release 8.0. These options enable you to turn off specific features in the BEA Tuxedo infrastructure. You should turn off these features only if they are not required by your CORBA or ATMI applications. [Table 4-3](#) describes these options.

Table 4-3 Performance Options

Option	Description	How to set . . .
Service and Interface Caching options (SICACHEENTRIESMAX and TMSICACHEENTRIESMAX)	This option enables you to cache service and interface entries, and to use the cached copies of the service or interface without locking the bulletin board.	For more information about these options, see Administering a BEA Tuxedo Application at Run Time and <code>UBBCONFIG(5)</code> and <code>TM_MIB(5)</code> , and <code>tuxenv(5)</code> in the File Formats, Data Descriptions, MIBs, and System Processes Reference .
Turning off threads (TMNOTTHREADS)	Set this option to yes to turn off multithreaded processing. For applications that do not use threads, turning them off should significantly improve performance.	You use the <code>tuxenv(5)</code> to set this option. For more information, see Administering a BEA Tuxedo Application at Run Time and <code>tuxenv(5)</code> in the File Formats, Data Descriptions, MIBs, and System Processes Reference .

Table 4-3 Performance Options (Continued)

Option	Description	How to set . . .
Turning off auditing and authorization (Options { [NO_AA] })	Setting this option disables the auditing and authorization functions on a per application basis.	You set this option in the RESOURCES section of the UBBCONFIG file. For more information, see Administering a BEA Tuxedo Application at Run Time and OPTION in the RESOURCES section of UBBCONFIG(5) in the File Formats, Data Descriptions, MIBs, and System Processes Reference .
Turning off XA Transactions (NO_XA)	Setting this option turns Off XA Transactions.	For more information about the NO_XA option, see Administering a BEA Tuxedo Application at Run Time and UBBCONFIG(5) and TM_MIB(5) in the File Formats, Data Descriptions, MIBs, and System Processes Reference .

Enhancing Efficiency with Application Parameters

This topic includes the following sections:

- [Determining IPC Requirements](#)
- [MINDISPATCHTHREADS](#)
- [MAXDISPATCHTHREADS](#)
- [Setting the MAXGTT, MAXBUFTYPE, and MAXBUFSTYPE Parameters](#)
- [Setting the SANITYSCAN, BLOCKTIME, BBLQUERY, and DBBLWAIT Parameters](#)

You can set these application parameters to enhance the efficiency of your system.

MAXDISPATCHTHREADS

The MAXDISPATCHTHREADS parameter determines the maximum number of concurrently dispatched threads that each server process can spawn. When specifying this parameter, consider the following:

- The value for MAXDISPATCHTHREADS determines the maximum size that the thread pool can grow to be, as it increases in size to accommodate incoming requests.

- The default value for `MAXDISPATCHTHREADS` is 1. If you specify a value greater than 1, the system creates and uses a special dispatcher thread. This dispatcher thread is not included in the number of threads determining the maximum size of the thread pool.
- Specifying a value of 1 for the `MAXDISPATCHTHREADS` parameter indicates that the server application should be configured as a single-threaded server. A value greater than 1 indicates that the server application should be configured as a multithreaded server.
- The value you specify for the `MAXDISPATCHTHREADS` parameter must not be less than the value you specify for the `MINDISPATCHTHREADS` parameter.
- The operating system resources limit the maximum number of threads that can be created in a process. `MAXDISPATCHTHREADS` should be less than that limit, minus the number of application managed threads that your application requires.

The value of the `MAXDISPATCHTHREADS` parameter affects other parameters. For example, the `MAXACCESSORS` parameter controls the number of simultaneous accesses to the BEA Tuxedo system, and each thread counts as one accessor. For a multithreaded server application, you must account for the number of system-managed threads that each server is configured to run. A system-managed thread is a thread that is started and managed by the BEA Tuxedo software, as opposed to threads started and managed by an application. Internally, BEA Tuxedo manages a pool of available system-managed threads. When a client request is received, an available system-managed thread from the thread pool is scheduled to execute the request. When the request is completed, the system-managed thread is returned to the pool of available threads.

For example, if that you have 4 multithreaded servers in your system and each server is configured to run 50 system-managed threads, the accessor requirement for these servers is the sum total of the accessors, calculated as follows:

$$50 + 50 + 50 + 50 = 200 \text{ accessors}$$

MINDISPATCHTHREADS

Use the `MINDISPATCHTHREADS` parameter to specify the number of server dispatch threads that are started when the server is initially booted. When you specify this parameter, consider the following:

- The value for `MINDISPATCHTHREADS` determines the initial allocation of threads in the thread pool.
- The separate dispatcher thread that is created when `MAXDISPATCHTHREADS` is greater than 1 is not counted as part of the `MINDISPATCHTHREADS` limit.

- The value you specify for `MINDISPATCHTHREADS` must not be greater than the value you specify for `MAXDISPATCHTHREADS`.
- The default value for `MINDISPATCHTHREADS` is 0.

Setting the `MAXACCESSERS`, `MAXOBJECTS`, `MAXSERVERS`, `MAXINTERFACES`, and `MAXSERVICES` Parameters

The `MAXACCESSERS`, `MAXOBJECTS`, `MAXSERVERS`, `MAXINTERFACES`, and `MAXSERVICES` parameters increase semaphore and shared memory costs, so you should choose the minimum value that satisfies the needs of the system. You should also allow for the variation in the number of clients accessing the system at the same time. Defaults may be appropriate for a generous allocation of IPC resources. However, it is prudent to set these parameters to the lowest appropriate values for the application.

For multithreaded servers, you must account for the number of threads that each server is configured to run. The `MAXACCESSERS` parameter sets the maximum number of concurrent accessors of a BEA Tuxedo system. Accessors include native and remote clients, servers, and administration processes. For more information on setting the `MAXACCESSERS` parameter, see [“MAXDISPATCHTHREADS” on page 4-11](#).

Setting the `MAXGTT`, `MAXBUFTYPE`, and `MAXBUFSTYPE` Parameters

You should increase the value of the `MAXGTT` parameter if the product of multiplying the number of clients in the system times the percentage of time they are committing a transaction is close to 100. This may require a great number of clients, depending on the speed of commit. If you increase `MAXGTT`, you should also increase `TLOGSIZE` accordingly for every machine. You should set `MAXGTT` to 0 for applications that do not use distributed transactions.

You can limit the number of buffer types and subtypes allowed in the application with the `MAXBUFTYPE` and `MAXBUFSTYPE` parameters, respectively. The current default for `MAXBUFTYPE` is 16. Unless you are creating many user-defined buffer types, you can omit `MAXBUFTYPE`. However, if you intend to use many different `VIEW` subtypes, you may want to set `MAXBUFSTYPE` to exceed its current default of 32.

Setting the SANITYSCAN, BLOCKTIME, BBLQUERY, and DBBLWAIT Parameters

If a system is running on slower processors (for example, due to heavy usage), you can increase the timing parameters: `SANITYSCAN`, `BLOCKTIME`, and individual transaction timeouts. If networking is slow, you can increase the value of the `BLOCKTIME`, `BBLQUERY`, and `DBBLWAIT` parameters.

Setting Application Parameters

[Table 4-4](#) describes the system parameters available for tuning an application.

Table 4-4 System Parameters for Application Tuning

Parameters	Action
<code>MAXACCESSERS</code> , <code>MAXOBJECTS</code> , <code>MAXSERVERS</code> , <code>MAXINTERFACES</code> , and <code>MAXSERVICES</code>	Set the smallest satisfactory value because of IPC cost. Allow for extra clients.
<code>MAXGTT</code> , <code>MAXBUFTYPE</code> , and <code>MAXBUFSTYPE</code>	Increase <code>MAXGTT</code> for many clients; set <code>MAXGTT</code> to 0 for nontransactional applications. Use <code>MAXBUFTYPE</code> only if you create eight or more user-defined buffer types. If you use many different <code>VIEW</code> subtypes, increase the value of <code>MAXBUFSTYPE</code> .
<code>BLOCKTIME</code> , <code>TRANTIME</code> , and <code>SANITYSCAN</code>	Increase the value for a slow system.
<code>BLOCKTIME</code> , <code>TRANTIME</code> , <code>BBLQUERY</code> , and <code>DBBLWAIT</code>	Increase values for slow networking.

Determining IPC Requirements

The values of different system parameters determine IPC requirements. You can use the `tmboot -c` command to test a configuration's IPC needs. The values of the following parameters affect the IPC needs of an application:

- MAXACCESSERS
- REPLYQ
- RQADDR (that allows MSSQ sets to be formed)
- MAXSERVERS
- MAXSERVICES
- MAXGTT

Table 4-5 describes the system parameters that affect the IPC needs of an application.

Table 4-5 Tuning IPC Parameters

Parameter(s)	Action
MAXACCESSERS	Equals the number of semaphores. Number of message queues is almost equal to MAXACCESSERS + the number of servers with reply queues (the number of servers in MSSQ set + the number of MSSQ sets).

Table 4-5 Tuning IPC Parameters (Continued)

Parameter(s)	Action
MAXSERVERS, MAXSERVICES, and MAXGTT	While MAXSERVERS, MAXSERVICES, MAXGTT, and the overall size of the ROUTING, GROUP, and NETWORK sections affect the size of shared memory, an attempt to devise formulas that correlate these parameters can become complex. Instead, simply run <code>tmboot -c</code> or <code>tmloadcf -c</code> to calculate the minimum IPC resource requirements for your application.
Queue-related kernel parameters	<p>Need to be tuned to manage the flow of buffer traffic between clients and servers. The maximum total size of a queue in bytes must be large enough to handle the largest message in the application, and to typically be 75 to 85 percent full. A smaller percentage is wasteful. A larger percentage causes message sends to block too frequently.</p> <p>Set the maximum size for a message to handle the largest buffer that the application sends.</p> <p>Maximum queue length (the largest number of messages that are allowed to sit on a queue at once) must be adequate for the application's operations.</p> <p>Simulate or run the application to measure the average fullness of a queue or its average length. This may be a trial and error process in which tunables are estimated before the application is run and are adjusted after running under performance analysis.</p> <p>For a large system, analyze the effect of parameter settings on the size of the operating system kernel. If unacceptable, reduce the number of application processes or distribute the application to more machines to reduce MAXACCESSERS.</p>

Measuring System Traffic

This topic includes the following sections:

- [About System Traffic and Bottlenecks](#)
- [Example of Detecting a System Bottleneck](#)
- [Detecting Bottlenecks on UNIX](#)
- [Detecting Bottlenecks on Windows](#)

For more information about monitoring BEA Tuxedo applications and measuring traffic, see “Monitoring a Running System” in the [Administering a BEA Tuxedo Application at Run Time](#).

About System Traffic and Bottlenecks

Bottlenecks can occur in your system when traffic volume nears resource capacity. You can measure service traffic using a global counter in your implementation code.

For example, in Tuxedo applications, when `tpsvrinit()` is invoked at boot time, you can initialize a global counter and record a starting time. Subsequently, each time a particular service is called, the counter is incremented. When the server is shut down by invoking the `tpsvrdone()` function, the final count and the ending time are recorded. This mechanism allows you to determine how busy a particular service is over a specified period of time.

Note: For CORBA C++ applications, use the `Server::initialize()` and `Server::release()` operations.

In BEA Tuxedo, bottlenecks can originate from data flow patterns. The quickest way to detect bottlenecks is to begin with the client and measure the amount of time required by relevant services.

Example of Detecting a System Bottleneck

Suppose Client 1 requires 4 seconds to print to the screen. Calls to `time(2)` determine that the `tpcall` to service A is the culprit with a 3.7 second delay. Service A is monitored at the top and bottom and takes 0.5 seconds. This implies that a queue may be clogged, which was determined by using the `pq` command.

On the other hand, suppose service A takes 3.2 seconds. The individual parts of Service A can be bracketed and measured. Perhaps Service A issues a `tpcall` to Service B, which requires 2.8 seconds. It should then be possible to isolate queue time or message send blocking time. Once the relevant amount of time has been identified, the application can be retuned to handle the traffic.

Using `time(2)`, you can measure the duration of the following:

- The entire client program.
- A client service request only.
- The entire service function.
- The service function making a service request (if any).

Detecting Bottlenecks on UNIX

On UNIX systems, the `sar(1)` command provides valuable performance information that can be used to find system bottlenecks. You can use the `sar(1)` command to:

- Sample cumulative activity counters in the operating system at predetermined intervals.
- Extract data from a system file.

Table 4-6 describes the `sar(1)` command options.

Table 4-6 `sar(1)` Command Options

Option	Description
-u	Gathers CPU utilization numbers, including the portion of the time running in user mode, running in system mode, idle with some process waiting for block I/O, and otherwise idle.
-b	Reports buffer activity, including transfers per second of data between system buffers and disk, or other block devices.
-c	Reports system call activity. This includes system calls of all types, as well as specific system calls such as <code>fork(2)</code> and <code>exec(2)</code> .
-w	Monitors system swapping activity. This includes the number of transfers for swap-ins and swap-outs.
-q	Reports average queue lengths while occupied and the percent of time occupied.
-m	Reports message and system semaphore activities, including the number of primitives per second.
-p	Reports paging activity, including the address translation page faults, page faults and protection errors, and the valid pages reclaimed for free lists.
-r	Reports unused memory pages and disk blocks, including the average number of pages available to user processes and the disk blocks available for process swapping.

Note: Some UNIX platforms do not provide the `sar(1)` command, but offer equivalent commands instead. BSD, for example, offers the `iostat(1)` command. Sun Microsystems, Inc. offers `perfmer(1)`.

Detecting Bottlenecks on Windows

On Windows, use the Performance Monitor to collect system information and detect bottlenecks. Click the Start button and select Programs, then Administration Tools, and then click Performance Monitor.

