BEA Tuxedo®

Programming a BEA Tuxedo Application Using COBOL

Release 8.1
January 2003

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Contents

4. Writing Clients

5. Writing Servers

6. Writing Request/Response Clients and Servers

7. Writing Conversational Clients and Servers

8. Writing Event-based Clients and Servers

9. Writing Global Transactions

10. Programming a Multithreaded and Multicontexted ATMI

Application

11. Managing Errors

12. COBOL Language Bindings for the Workstation Component

About This Document

This document explains how to program BEA Tuxedo ATMI applications using the COBOL language.

This document covers the following topics:

- Chapter 1, “Introduction to BEA Tuxedo Programming,” provides an overview of the BEA Tuxedo programming, including information on distributed application programming, clients, servers, and the BEA Tuxedo Application-to-Transaction Monitoring (ATMI) interface.
- Chapter 2, “Programming Environment,” describes the BEA Tuxedo programming environment, including information on configuring a BEA Tuxedo system, setting environment variables, and starting and stopping applications.
- Chapter 3, “Managing Typed Records,” provides instructions on managing and using typed records, including VIEW, FML, and XML records.
- Chapter 4, “Writing Clients,” provides instructions on writing and building BEA Tuxedo client applications using the COBOL language. A client process example is provided.
- Chapter 5, “Writing Servers,” provides instructions on writing and building BEA Tuxedo servers using the COBOL language, including defining and advertising services.
- Chapter 6, “Writing Request/Response Clients and Servers,” provides instructions on writing request/response clients and servers, including synchronous and asynchronous messaging, and setting message priorities.
- Chapter 7, “Writing Conversational Clients and Servers,” provides instructions on writing conversational clients and servers, including joining an application, establishing a connection, sending and receiving messages, and ending a conversation.

- Chapter 8, “Writing Event-based Clients and Servers,” provides instructions on writing event-based clients and servers, including handling unsolicited messages and events.
- Chapter 9, “Writing Global Transactions,” provides instructions on writing global transactions, including starting and terminating transactions.
- Chapter 10, “Programming a Multithreaded and Multicontexted ATMI Application,” provides instructions on writing applications where a single process performs multiple tasks simultaneously. The chapter describes programming techniques for multithreading (the inclusion of more than one unit of execution in a single process) and multicontexting (the ability of a single process to have more than one connection within a domain or connections to more than one domain).
- Chapter 11, “Managing Errors,” provides instructions on handling errors, including both system and application errors.
- Chapter 12, “COBOL Language Bindings for the Workstation Component,” provides information on COBOL language binding for UNIX and Microsoft Windows platforms.

What You Need to Know

This document is intended for application developers who are interested in programming applications using the COBOL language in a BEA Tuxedo environment

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

e-docs Web Site

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

How to Print the Document

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

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

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

Related Information

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

- `cobcc(1)` in *BEA Tuxedo Command Reference*
- *BEA Tuxedo ATMI COBOL Function Reference*
- `tuxenv(5)` in *File Formats, Data Descriptions, MIBs, and System Processes Reference*

Contact Us!

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

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

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

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

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

Documentation Conventions

The following documentation conventions are used throughout this document.

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

Convention	Item
italics	Indicates emphasis or book titles.

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

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

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

Convention	Item
|	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.
. . .	Indicates one of the following in a command line: ■ That an argument can be repeated several times in a command line ■ That the statement omits additional optional arguments ■ That you can enter additional parameters, values, or other information The ellipsis itself should never be typed. *Example*: `buildobjclient [-v] [-o name ] [-f file-list]... [-l file-list]...`
. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

CHAPTER

1 Introduction to BEA Tuxedo Programming

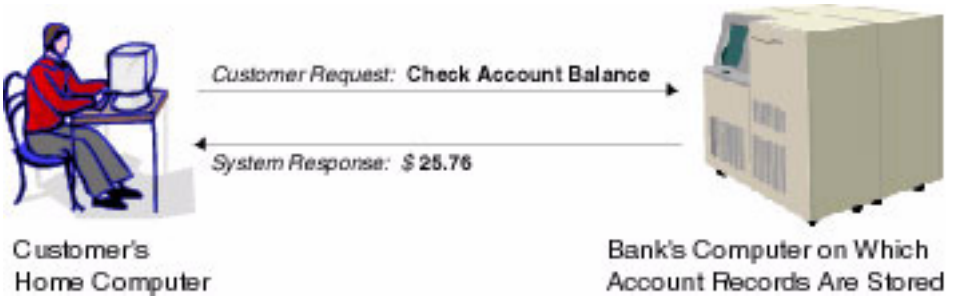
This topic includes the following sections:

- BEA Tuxedo Distributed Application Programming
- Communication Paradigms
- BEA Tuxedo Clients
- BEA Tuxedo Servers
- BEA Tuxedo API: ATMI

BEA Tuxedo Distributed Application Programming

A *distributed application* consists of a set of software modules that reside on multiple hardware systems, and that communicate with one another to accomplish the tasks required of the application. For example, as shown in the following figure, a distributed application for a remote online banking system includes software modules that run on a bank customer's home computer, and a computer system at the bank on which all bank account records are maintained.

Figure 1-1 Distributed Application Example - Online Banking System



The task of checking an account balance, for example, can be performed simply by logging on and selecting an option from a menu. Behind the scenes, the local software module communicates with the remote software module using special application programming interface (API) routines.

The BEA Tuxedo distributed application programming environment provides the API routines necessary to enable secure, reliable communication between the distributed software modules. This API is referred to as the Application-to-Transaction Monitor Interface (ATMI).

The ATMI enables you to:

- Send and receive messages between clients and servers, possibly across a network of heterogeneous machines
- Establish and use client naming and security features
- Define and manage transactions in which data may be stored in several locations
- Generically open and close a resource manager such as a Database Management System (DBMS)
- Manage the flow of service requests and the availability of servers to process them

Communication Paradigms

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

Table 1-1 Communication Paradigms

Paradigm	Description
Request/response communication	Request/response communication enables one software module to send a request to a second software module and wait for a response. Can be synchronous (processing waits until the requester receives the response) or asynchronous (processing continues while the requester waits for the response). This mode is also referred to as client/server interaction. The first software module assumes the role of the client; the second, of the server. Refer to "Writing Request/Response Clients and Servers" on page 6-1 for more information on this paradigm.
Conversational communication	Conversational communication is similar to request/response communication, except that multiple requests and/or responses need to take place before the "conversation" is terminated. With conversational communication, both the client and the server maintain state information until the conversation is disconnected. The application protocol that you are using governs how messages are communicated between the client and server. Conversational communication is commonly used to buffer portions of a lengthy response from a server to a client. Refer to "Writing Conversational Clients and Servers" on page 7-1 for more information on this paradigm.

Paradigm	Description
Application queue-based communication	Application queue-based communication supports deferred or time-independent communication, enabling a client and server to communicate using an application queue. The BEA Tuxedo/Q facility allows messages to be queued to persistent storage (disk) or to non-persistent storage (memory) for later processing or retrieval. For example, application queue-based communication is useful for enqueuing requests when a system goes offline for maintenance, or for buffering communications if the client and server systems are operating at different speeds. Refer to *Using the ATMI /Q Component* for more information on the /Q facility.
Event-based communication	Event-based communication allows a client or server to notify a client when a specific situation (event) occurs. Events are reported in one of two ways: ■ Unsolicited events are unexpected situations that are reported by clients and/or servers directly to clients. ■ Brokered events are unexpected situations or predictable occurrences with unpredictable timeframes that are reported by servers to clients indirectly, through an "anonymous broker" program that receives and distributes messages. Event-based communication is based on the BEA Tuxedo EventBroker facility. Refer to "Writing Event-based Clients and Servers" on page 8-1 for more information on this paradigm.

BEA Tuxedo Clients

A BEA Tuxedo ATMI *client* is a software module that collects a user request and forwards it to a server that offers the requested service. Almost any software module can become a BEA Tuxedo client by calling the ATMI client initialization routine and "joining" the BEA Tuxedo application. The client can then exchange information with the server.

The client calls the ATMI termination routine to "leave" the application and notify the BEA Tuxedo system that it (the client) no longer needs to be tracked. Consequently, BEA Tuxedo application resources are made available for other operations.

The operation of a basic client process can be summarized by the pseudo-code shown in the following listing.

Listing 1-1 Pseudo-code for a Client

```
START PROGRAM
enroll as a client of the BEA TUXEDO application
place initial client identification in data structure
perform until end
get user input
place user input in DATA-REC
send service request
receive reply
pass reply to the user
end perform
leave application
END PROGRAM
```

Most of the actions described in the above listing are implemented with ATMI calls. Others—placing the user input in `DATA-REC` and passing the reply to the user—are implemented with COBOL routines.

An ATMI client may send and receive any number of service requests before leaving the application. The client may send these requests as a series of request/response calls or, if it is important to carry state information from one call to the next, by establishing a connection to a conversational server. In both cases, the logic in the client program is similar, but different ATMI calls are required for these two approaches.

Before you can execute an ATMI client, you must run the `buildclient -C` command to compile it and link it with the BEA Tuxedo ATMI and required libraries. Refer to "Writing Clients" on page 4-1 for information on the `buildclient(1)` command.

BEA Tuxedo Servers

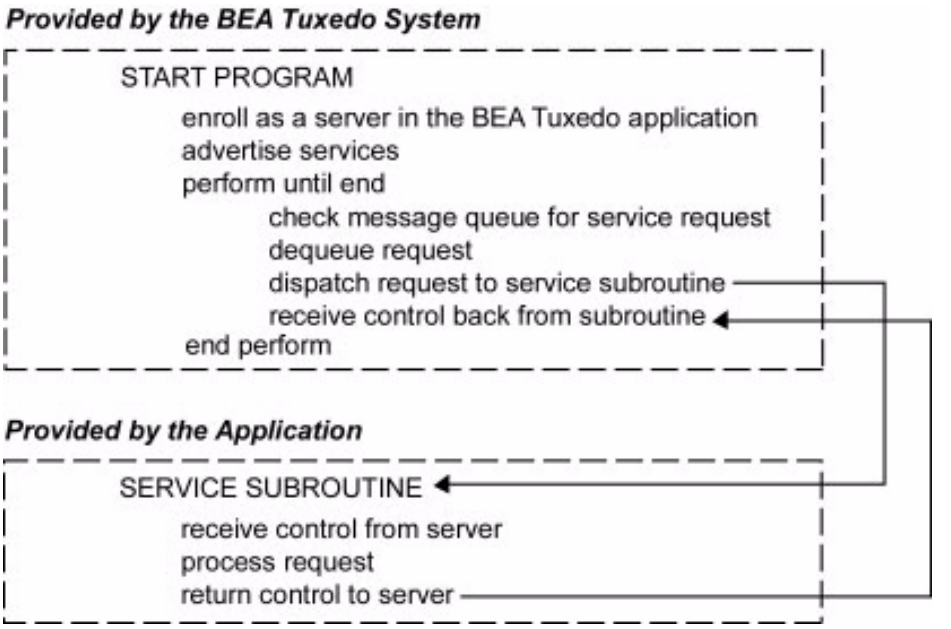
A BEA Tuxedo ATMI *server* is a process that provides one or more *services* to a client. A service is a specific business task that a client may need to perform. Servers receive requests from clients and dispatch them to the appropriate service subroutines.

Basic Server Operation

To build server processes, applications combine their service subroutines with a controlling program provided by the BEA Tuxedo system. This system-supplied controlling program is a set of predefined routines. It performs server initialization and termination and places user input in data structures that can be used to receive and dispatch incoming requests to service routines. All of this processing is transparent to the application.

The following figure summarizes, in pseudo-code, the interaction between a server and a service subroutine.

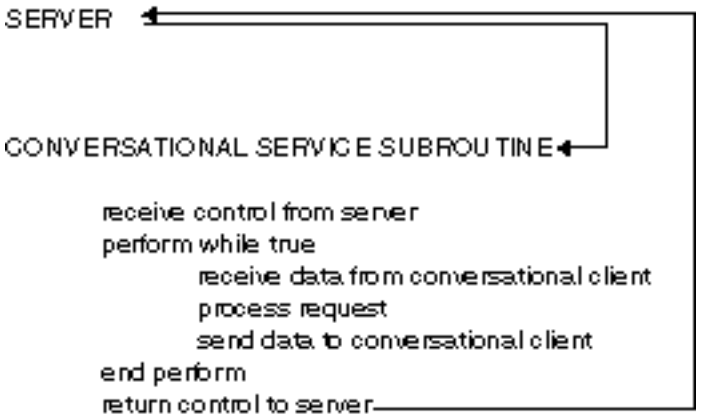
Figure 1-2 Pseudo-code for a Request/Response Server and a Service Subroutine



After initialization, an ATMI waits until a request message is delivered to its message queue, dequeues the request, and dispatches it to a service subroutine for processing. If a reply is required, the reply is considered part of request processing.

The conversational paradigm is somewhat different from request/response, as illustrated by the pseudo-code in the following figure.

Figure 1-3 Pseudo-code for a Conversational Service Subroutine



The BEA Tuxedo system-supplied controlling program contains the code needed to enroll a process as an ATMI server, advertise services, and dequeue requests. ATMI calls are used in service subroutines that process requests. When you are ready to compile and test your service subroutines, you must link edit them with the server and generate an executable server. To do so, run the `buildserver -C` command.

Servers as Requesters

If a client requests several services, or several iterations of the same service, a subset of the services might be transferred to another server for execution. In this case, the server assumes the role of a client, or *requester*. Both clients and servers can be requesters; a client, however, can only be a requester. This coding model is easily accomplished using the BEA Tuxedo ATMI calls.

Note: A request/response server can also forward a request to another server. In this case, the server does not assume the role of client (requester) because the reply is expected by the original client, not by the server forwarding the request.

BEA Tuxedo API: ATMI

In addition to the COBOL code that expresses the logic of your application, you must use the Application-to-Transaction Monitor Interface (ATMI), the interface between your application and the BEA Tuxedo system.

The ATMI is a reasonably compact set of calls used to open and close resources, begin and end transactions, and support communication between clients and servers. The following table summarizes the ATMI calls. Each call is described in the *BEA Tuxedo ATMI COBOL Function Reference*.

Table 1-2 Using the ATMI Calls

For a Task Related to . . .	Use This COBOL Function . . .	To . . .	For More Information, Refer to . . .
Client membership	TPINITIALIZE	Have a client join an application	"Writing Clients" on page 4-1
	TPTERM	Have a client leave an application	
Multiple application context management	TPGETCTXT(3cbl)	Retrieve an identifier for the current threads context	"Programming a Multithreaded and Multicontexted ATMI Application" on page 10-1
	TPSETCTXT(3cbl)	Set the current thread's context in a multicontexted process	
Service entry and return	TPSVCSTART	Get service information	"Writing Servers" on page 5-1
	TPSVRINIT	Initialize a server	
	TPSVRDONE	Terminate a server	
	TPRETURN	End a service routine	
	TPFORWAR	Forward a request	
Dynamic advertisement	TPADVERTISE	Advertise a service name	"Writing Servers" on page 5-1
	TPUNADVERTISE	Unadvertise a service name	

Table 1-2 Using the ATMI Calls

For a Task Related to . . .	Use This COBOL Function . . .	To . . .	For More Information, Refer to . . .
Message priority	TPGPRIO	Get the priority of the last request	"Writing Servers" on page 5-1
	TPSPRIO	Set the priority of the next request	
Request/Response communications	TPCALL	Initiate a synchronous request/response to a service	■ "Writing Servers" on page 5-1 ■ "Writing Request/Response Clients and Servers" on page 6-1
	TPACALL	Initiate an asynchronous request (fanout)	
	TPGETRPLY	Receive an asynchronous response	
	TPCANCEL	Cancel an asynchronous request	
Conversational communications	TPCONNECT	Begin a conversation with a service	"Writing Conversational Clients and Servers" on page 7-1
	TPDISCON	Abnormally terminate a conversation	
	TPSEND	Send a message in a conversation	
	TPRECV	Receive a message in a conversation	
Reliable queuing	TPENQUEUE(3cbl)	Enqueue a message to a message queue	*Using the ATMI /Q Component*
	TPDEQUEUE(3cbl)	Dequeue a message from a message queue	

Table 1-2 Using the ATMI Calls

For a Task Related to . . .	Use This COBOL Function . . .	To . . .	For More Information, Refer to . . .
Event-based communications	TPNOTIFY	Send an unsolicited message to a client	"Writing Event-based Clients and Servers" on page 8-1
	TPBROADCAST	Send messages to several clients	
	TPSETUNSOL	Set unsolicited message call-back	
	TPCHKUNSOL	Check the arrival of unsolicited messages	
	TPGETUNSOL	Get an unsolicited message	
	TPPOST	Post an event message	
	TPSUBSCRIBE	Subscribe to event messages	
	TPUNSUBSCRIBE	Unsubscribe to event messages	
Transaction management	TPBEGIN	Begin a transaction	"Writing Global Transactions" on page 9-1
	TPCOMMIT	Commit the current transaction	
	TPABORT	Roll back the current transaction	
	TPGETLEV	Check whether in transaction mode	
Resource management	TPOPEN(3cbl)	Open a resource manager	■ "Programming a Multithreaded and Multicontexted ATMI Application" on page 10-1 ■ *Getting Started with BEA Tuxedo CORBA Applications*
	TPCLOSE(3cbl)	Close a resource manager	

Table 1-2 Using the ATMI Calls

For a Task Related to . . .	Use This COBOL Function . . .	To . . .	For More Information, Refer to . . .
Security	TPKEYOPEN(3cbl)	Open a key handle for digital signature generation, message encryption, or message decryption	*Using Security in CORBA Applications*
	TPKEYGETINFO(3cbl)	Get information associated with a key handle	
	TPKEYSETINFO(3cbl)	Set optional attributes associated with a key handle	
	TPKEYCLOSE(3cbl)	Close a key handle previously opened using TPKEYOPEN	

CHAPTER

2 Programming Environment

This topic includes the following sections:

- Updating the UBBCONFIG Configuration File
- Setting Environment Variables
- Defining Equivalent Data Types
- Starting and Stopping the Application

Updating the UBBCONFIG Configuration File

The application administrator initially defines the configuration settings for an application in the `UBBCONFIG` configuration file. To customize your programming environment, you may need to create or update a configuration file.

If you need to create or update a configuration file, refer to the following guidelines:

- Copy and edit a file that already exists. For example, the file `ubbshm` that comes with the `bankapp` sample application can provide a good starting point.
- Minimize complexity. For test purposes, set up your application as a shared memory, single-processor system. Use regular operating system files for your data.

- Make sure the IPCKEY parameter in the configuration file does not conflict with any other parameters being used at your installation. Check with your BEA Tuxedo application administrator, and refer to *Setting Up a BEA Tuxedo Application* for more information.
- Set the UID and GID parameters so that you are the owner of the configuration.
- Review the documentation. The configuration file is described in UBBCONFIG(5) in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*.

The following table summarizes the UBBCONFIG configuration file parameters that affect the programming environment. Parameters are listed by functional category.

Table 2-1 Programming-related UBBCONFIG Parameters by Functional Category

Functional Category	Parameter	Section	Description
Global resource limits	MAXSERVERS	RESOURCES	Specifies the maximum number of servers in the configuration. When setting this value, you need to consider the MAX values for all servers.
	MAXSERVICES	RESOURCES	Specifies the maximum total number of services in the configuration.
Data-dependent routing	BUFTYPE	ROUTING	List of types and subtypes of data records for which the specified routing entry is valid.
Link-level encryption	MINENCRYPTBITS	NETWORK	Sets the minimum encryption level that a process accepts.
	MAXENCRYPTBITS	NETWORK	Sets the maximum encryption level that a process accepts.

Table 2-1 Programming-related UBBCONFIG Parameters by Functional Category (Continued)

Functional Category	Parameter	Section	Description
Load balancing	LDBAL	RESOURCES	Flag for specifying whether or not load balancing is enabled. If enabled, the BEA Tuxedo system attempts to balance requests across the network.
	NETLOAD	MACHINES	Numeric value that is added to the load factor of services that are remote from the invoking client, providing a bias for choosing a local server over a remote server. Load balancing must be enabled (that is, LDBAL must be set to Y).
	LOAD	SERVICES	Relative load factor associated with a service instance. The default is 50.
Security	AUTHSVC	RESOURCES	Specifies the name of an application authentication service that is invoked by the system for each client joining the system.
	SECURITY	RESOURCES	Specifies the type of application security to be enforced.

Table 2-1 Programming-related UBBCONFIG Parameters by Functional Category (Continued)

Functional Category	Parameter	Section	Description
Conversational communication	MAXCONV	RESOURCES	Sets the maximum number of simultaneous conversations for a single machine. You can specify a value between 0 and 32,767. The default is 64 if any conversational servers are defined in the SERVERS section; otherwise, the default is 1. The specified value can be overriden for each machine in the MACHINES section.
	CONV	SERVERS	Specifies whether or not conversational communication is supported. If this parameter is set to N or unspecified, a TPCONNECT call to a service fails.
	MIN/MAX	SERVERS	Specifies the minimum and maximum number of occurrences of the server to be started by tmboot(1). If not specified, MIN defaults to 1 and MAX defaults to MIN. The same parameters are available for use with request/response servers. However, conversational servers are automatically spawned as needed. So if you set MIN=1 and MAX=10, for example, tmboot starts one server initially. When a TPCONNECT call is made to a service offered by that server, the system starts a second copy of a server. As each copy is called, a new one is spawned, up to a limit of 10.

Table 2-1 Programming-related UBBCONFIG Parameters by Functional Category (Continued)

Functional Category	Parameter	Section	Description
Transaction management	AUTOTRAN	SERVICES	Controls whether a service routine is placed in transaction mode. If you set this parameter to Y, a transaction in the service subroutine is automatically started whenever a request message is received from another process.
Multithreaded servers	MAXDISPATCHTHREADS	SERVERS	Specifies the maximum number of concurrently dispatched threads that each server process may spawn.
	MINDISPATCHTHREADS	SERVERS	Specifies the number of server dispatch threads started on initial server boot.

The configuration file is an operating system text file. To make it usable by the system, you must execute the `tmloadcf(1)` command to convert the file to a binary file.

See Also

- *Setting Up a BEA Tuxedo Application*
- `UBBCONFIG(5)` in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*

Setting Environment Variables

Initially, the application administrator sets the variables that define the environment in which your application runs. These environment variables are set by assigning values to the `ENVFILE` parameter in the `MACHINES` section of the `UBBCONFIG` file. (Refer to *Setting Up a BEA Tuxedo Application* for more information.)

For the client and server routines in your application, you can update existing environment variables or create new ones. The following table summarizes the most commonly used environment variables. The variables are listed by functional category.

Table 2-2 Programming-related Environment Variables

Function	Environment Variable	Defines the . . .	Used by . . .
Global	TUXDIR	Location of the BEA Tuxedo system binary files.	BEA Tuxedo application programs.
Configuration	TUXCONFIG	Location of the BEA Tuxedo configuration file.	BEA Tuxedo application programs.
Compiling	ALTCC[1]	Command that invokes the COBOL compiler. Default is `cobcc`.	`builclient() -C` and `buildserver() -C` commands.
	ALTCFLAGS[1]	Link edit flags to be passed to the COBOL compiler. Link edit flags are optional.	`builclient() -C` and `buildserver() -C` commands.
	COBOPT	Arguments that you may want to use on the compile command line.	`builclient() -C` and `buildserver() -C` commands.
	COBCPY	Directories that contain a set of the COBOL `COPY` files to be used by the compiler.	`builclient() -C` and `buildserver() -C` commands.
Data compression	TMCMPPRFM	Level of compression between 1 and 9.	BEA Tuxedo application programs that perform data compression.

Function	Environment Variable	Defines the . . .	Used by . . .
Load balancing	TMNETLOAD	Numeric value that is added to the load value for remote queues, making the remote queues appear to have more work than they actually do. As a result, even if load balancing is enabled, local requests are sent to local queues more often than to remote queues.	BEA Tuxedo application programs that perform load balancing.
Record management	FIELDTBLS or FIELDTBLS32	Comma-separated list of field table filenames for FML and FML32 typed records, respectively. Required only for FML VIEW types.	FML and FML32 record types and FML VIEWs.
	FLDTBLDIR or FLDTBLDIR32	Colon-separated list of directories to be searched for the field table files for FML and FML32, respectively. For Windows 2000, a semicolon-separated list is used.	FML and FML32 record types and FML VIEWs.
	VIEWFILES or VIEWFILES32	Comma-separated list of allowable filenames for VIEW and VIEW32 typed records, respectively.	VIEW and VIEW32 record types.
	VIEWDIR or VIEWDIR32	Colon-separated list of directories to be searched for VIEW and VIEW32 files, respectively. For Windows 2000, a semicolon-separated list is used.	VIEW and VIEW32 record types.

1. On a Windows 2000 system, the ALTCC and ALTCFLAGS environment variables are not applicable and setting them will produce unexpected results. You must compile your application first using a COBOL compiler and then pass the resulting object file to the buildclient or buildserver command.

If operating in a UNIX environment, add $TUXDIR/bin to your environment PATH to ensure that your application can locate the executables for the BEA Tuxedo system commands. For more information on setting up the environment, refer to *Setting Up a BEA Tuxedo Application*.

See Also

- *Setting Up a BEA Tuxedo Application*

Defining Equivalent Data Types

The following table lists the C data types for which equivalent COBOL data types are available.

Table 2-3 COBOL Equivalents for C Data Types

C Data Type	Equivalent COBOL Data Type
float	COMP-1
double	COMP-2
long	S9(9) COMP-5[1]
short	S9(4) COMP-5[1]
dec_t	COBOL COMP-3 packed decimal field

1. COMP-5, provided for use with MicroFocus COBOL, allows the COBOL integer fields to match the data format of the corresponding C fields. The data type for VS COBOL II is COMP.

For storage efficiency, COBOL supports packed decimals: two decimal digits packed into each byte with the low-order half byte used to store the sign. The length of a packed decimal may be 1 to 9 bytes with storage available for 1 to 17 digits, including the sign.

The `dec_t` field is defined in a `VIEW`. The size is specified as two values separated by a comma. The first value indicates the total number of bytes occupied by the decimal in COBOL. The second value indicates the number of digits to the right of the decimal point in COBOL. You can use the following formula to convert the `dec_t` field to a COBOL declaration:

`dec_t(`*m*`, `*n*`) => S9(2*`*m*`-(`*n*`+1),`*n*`)COMP-3`

For example, a size specification of 6,4 in the `VIEW` indicates that there are 4 digits to the right of the decimal point and 7 digits to the left, and the last half byte is used to store the sign. A COBOL application programmer represents this as `9(7)V9(4)`, where the `V` represents the decimal point between each value. Note that `FML` does not support the `dec_t` type; if `FML`-dependent `VIEW`s are used, then each field must be mapped to a `C` type in the `VIEW` file. For instance, a packed decimal can be mapped to an `FML` string field, and then the mapping functions can be used to do the conversion between formats.

Starting and Stopping the Application

To start the application, execute the `tmboot(1)` command. The command gets the IPC resources required by the application, and starts administrative processes and application servers.

To stop the application, execute the `tmshutdown(1)` command. The command stops the servers and releases the IPC resources used by the application, except any that might be used by the resource manager, such as a database.

See Also

- `tmboot(1)` and `tmshutdown(1)` in the *BEA Tuxedo Command Reference*

3 Managing Typed Records

This topic includes the following sections:

- Overview of Typed Records
- Defining Typed Records
- Using a VIEW Typed Record
- Using an FML Typed Record
- Using an XML Typed Record

Overview of Typed Records

In order to send data to another application program, the sending program first places the data in a record. BEA Tuxedo ATMI clients use typed records to send messages to ATMI servers. The term “typed record” refers to a pair of COBOL records: a data record and an auxiliary type record. The data record is defined in static storage and contains application data to be passed to another application program. An auxiliary type record accompanies the data record. It specifies the interpretation and translation rules of the data record to be used by the BEA Tuxedo system when passing the information between heterogeneous systems. Typed records make up one of the fundamental features of the distributed programming environment supported by the BEA Tuxedo system.

Why *typed*? In a distributed environment, an application may be installed on heterogeneous systems that communicate across multiple networks using different protocols. Different types of records require different routines to initialize, send and receive messages, and encode and decode data. Each record is designated as a specific type so that the appropriate routines can be called automatically without programmer intervention.

The following table lists the typed records supported by the BEA Tuxedo system and indicates whether or not:

- The record is *self-describing*; in other words, the record data type and length can be determined simply by (a) knowing the type and subtype, and (b) looking at the data.
- The record requires a subtype.
- The system supports data-dependent routing for the typed record.
- The system supports encoding and decoding for the typed record.

If any routing routines are required, the application programmer must provide them as part of the application.**Records**

Table 3-1 Typed Buffers

Typed Record	Description	Self-Describing	Subtype	Data-Dependent Routing	Encoding/Decoding
CARRAY	Undefined array of characters, any of which can be LOW-VALUE. This typed record is used to handle the data opaquely, as the BEA Tuxedo system does not interpret the semantics of the array. Because a CARRAY is not self-describing, the length must always be provided during transmission. Encoding and decoding are not supported for messages sent between machines because the bytes are not interpreted by the system.	No	No	No	No

Table 3-1 Typed Buffers (Continued)

Typed Record	Description	Self-Describing	Subtype	Data-Dependent Routing	Encoding/ Decoding
FML (Field Manipulation Language)	Proprietary BEA Tuxedo system type of self-describing record in which each data field carries its own identifier, an occurrence number, and possibly a length indicator. T record offers data-independence and greater flexibility The FML record uses 16 bits for field identifiers and lengths of fields. Refer to "Using an FML Typed Record" on page 3-14 for more information.	Yes	No	Yes	Yes
FML32	Equivalent to FML but uses 32 bits for field identifiers and lengths of fields, which allows for larger and more fields and, consequently, larger overall records. However, the FML routines that are available for manipulating the FML typed record in the C programming language are not available in COBOL.The primary use of FML32 in COBOL is simply to work with C programs in which VIEW32 or FML32 typed records are used. Refer to "Using an FML Typed Record" on page 3-14 for more information.	Yes	No	Yes	Yes
STRING	Array of characters that terminates with a LOW-VALUE character. The BEA Tuxedo system can convert data automatically when data is exchanged by machines with different character sets.	No	No	No	No

Table 3-1 Typed Buffers (Continued)

Typed Record	Description	Self-Describing	Subtype	Data-Dependent Routing	Encoding/ Decoding
VIEW	COBOL data structure defined by the application. VIEW types must have subtypes that designate individual data structures. A view description file, in which the fields and types that appear in the data structure are defined, must be available to client and server processes that use a data structure described in a VIEW typed record. Encoding and decoding are performed automatically if the record is passed between machines of different types. Refer to "Using a VIEW Typed Record" on page 3-7 for more information.	No	Yes	Yes	Yes
VIEW32	Equivalent to VIEW but uses 32 bits for length and count fields, which allows for larger and more fields and, consequently, larger overall records. The primary use of VIEW32 in COBOL is simply to work with C programs in which VIEW32 or FML32 typed records are used. Refer to "Using a VIEW Typed Record" on page 3-7 for more information.	No	Yes	Yes	Yes
X_COMMON	Equivalent to VIEW, but used for compatibility between COBOL and C programs. Field types should be limited to short, long, and string.	No	Yes	Yes	Yes

Table 3-1 Typed Buffers (Continued)

Typed Record	Description	Self-Describing	Subtype	Data-Dependent Routing	Encoding/ Decoding
XML	An XML document that consists of: ■ Text, in the form of a sequence of encoded characters ■ A description of the logical structure of the document and information about that structure The routing of an XML document can be based on element content, or on element type and an attribute value. The XML parser determines the character encoding being used; if the encoding differs from the native character sets (US-ASCII or EBCDIC) used in the BEA Tuxedo configuration files (UBBCONFIG(5) and DMCONFIG(5)), the element and attribute names are converted to US-ASCII or EBCDIC. Refer to "Using an XML Typed Record" on page 3-21 for more information.	No	No	Yes	No
X_OCTET	Equivalent to CARRAY.	No	No	No	No

All record types are defined in a file called tmtypesw.c in the $TUXDIR/lib directory. Only record types defined in tmtypesw.c are known to your client and server programs. You can edit the tmtypesw.c file to add or remove record types. In addition, you can use the BUFTYPE parameter (in UBBCONFIG) to restrict the types and subtypes that can be processed by a given service.

The tmtypesw.c file is used to build a shared object or dynamic link library. This object is dynamically loaded by both BEA Tuxedo administrative servers, and application clients and servers.

See Also

- "Using a VIEW Typed Record" on page 3-7

- "Using an FML Typed Record" on page 3-14
- "Using an XML Typed Record" on page 3-21
- `tuxtypes(5)` in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*
- `UBBCONFIG(5)` in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*

Defining Typed Records

The `TPTYPE-REC` COBOL structure is used whenever sending or receiving application data.

The following table lists the `TPTYPE-REC` structure fields.

Field	Description
`REC-TYPE`	Specifies which record type the application wishes to send or receive.
`SUB-TYPE`	Specifies the subtype of the record type, if further classification is required (as it is, for example, in a `VIEW` record).
`LEN`	When data is being sent, specifies the number of bytes to be sent. After a successful transfer, `LEN` contains the number of bytes transferred. When data is being received, `LEN` in `TPTYPE-REC` specifies the number of bytes to be moved into the data record. After a successful call, `LEN` contains the number of bytes moved into the data record. If the size of the incoming message is larger than the size specified in `LEN`, the data is truncated, all data after the `LEN` length is reached is discarded, and `TPTYPE-STATUS` is set to `TPTRUNCATE`.

The following shows the TPTYPE data structure:

```
05 REC-TYPE PIC X(8).
  88 X-OCTET VALUE "X_OCTET".
  88 X-COMMON VALUE "X_COMMON".
05 SUB-TYPE PIC X(16).
05 LEN PIC S9(9) COMP-5.
  88 NO-LENGTH VALUE 0.
05 TPTYPE-STATUS PIC S9(9) COMP-5.
  88 TPTYPEOK  VALUE 0.
  88 TPTRUNCATE VALUE 1.
```

Using a VIEW Typed Record

There are two kinds of VIEW typed records. The first, FML VIEW, is a COBOL record generated from an FML record. The second is simply an independent COBOL record.

The reason for converting FML records into COBOL records and back again (and the purpose of the FML VIEW typed records) is that FML functions are not available in the COBOL programming environment.

For more information on the FML typed record, refer to the *BEA Tuxedo ATMI FML Function Reference*.

To use VIEW typed records, you must perform the following steps:

- Set the appropriate environment variables.
- Describe each structure in view description files.
- Compile the view description files using viewc -C, the BEA Tuxedo view compiler. By running this comand you will produce one or more COBOL COPY files (one per view), each of which contains data description records. These records can be used in the LINKAGE section or the WORKING STORAGE section of the DATA DIVISION, according to the demands of the program.

Setting Environment Variables for a VIEW Typed Record

To use a VIEW typed record in an application, you must set the following environment variables.

Table 3-2 Environment Variables for a VIEW Typed Record

Environment Variable	Description
FIELDTBLS or FIELDTBLS32	Comma-separated list of field table filenames for FML or FML32 typed records. Required only for FML VIEW types.
FLDTBLDIR or FLDTBLDIR32	Colon-separated list of directories to search for the field table files for FML and FML32 typed records. For Microsoft Windows, use a semicolon-separated list. Required only for FML VIEW types.
VIEWFILES or VIEWFILES32	Comma-separated list of allowable filenames for VIEW or VIEW32 description files.
VIEWDIR or VIEWDIR32	Colon-separated list of directories to search for VIEW or VIEW32 files. For Microsoft Windows, use a semicolon-separated list.

Creating a View Description File

To use a VIEW typed record, you must define the COBOL record in a view description file. The view description file includes, a view for each entry, a view that describes the characteristic COBOL procedure mapping and the potential FML conversion pattern. The name of the view corresponds to the name of the copy file that is included in COBOL program.

The following format is used for each record in the view description file:

```
$ /* View structure */
  VIEW viewname
  type    cname     fbname   count   flag    size    null
```

The following table describes the fields that must be specified in the view description file for each COBOL record.

Table 3-3 View Description File Fields

Field	Description
type	Data type of the field. Can be set to `short`, `long`, `float`, `double`, `char`, `string`, or `carray`.
cname	Name of the field as it appears in the COBOL record.
fbname	If you will be using the `FML`-to-`VIEW` or `VIEW`-to-`FML` conversion routines, this field must be included to indicate the corresponding `FML` name. This field name must also appear in the FML field table file. This field is not required for FML-independent VIEWs.
count	Number of times field occurs.
flag	Specifies any of the following optional flag settings: ■ P—change the interpretation of the LOW-VALUE value ■ S—one-way mapping from fielded record to structure ■ F—one-way mapping from structure to fielded record ■ N—zero-way mapping ■ C—generate additional field for associated count member (ACM) ■ L—hold number of bytes transferred for `STRING` and `CARRAY`
size	For `STRING` and `CARRAY` record types, specifies the maximum length of the value. This field is ignored for all other record types.

Table 3-3 View Description File Fields (Continued)

Field	Description
null	User-specified LOW-VALUE value, or minus sign (-) to indicate the default value for a field. LOW-VALUE values are used in VIEW typed records to indicate empty COBOL record members. The default LOW-VALUE value for all numeric types is 0 (0.0 for dec_t). For character types, the default LOW-VALUE value is '\0'. For STRING and CARRAY types, the default LOW-VALUE value is " ". Constants used, by convention, as escape characters can also be used to specify a LOW-VALUE value. The view compiler recognizes the following escape constants: \ddd (where d is an octal digit), \0, \n, \t, \v, \r, \f, \\, \', and \". You may enclose STRING, CARRAY, and LOW-VALUE values in double or single quotes. The view compiler does not accept unescaped quotes within a user-specified LOW-VALUE value. You can also specify the keyword NONE in the LOW-VALUE field of a view member description, which means that there is no LOW-VALUE value for the member. The maximum size of default values for string and character array members is 2660 characters. For more information, refer to the *BEA Tuxedo ATMI FML Function Reference*.

You can include a comment line by prefixing it with the # or $ character. Lines prefixed by a $ sign are included in the .h file.

The following listing is an excerpt from an example view description file based on an FML record. In this case, the *fbname* field must be specified and match that which appears in the corresponding field table file. Note that the CARRAY1 field includes an occurrence count of 2 and sets the C flag to indicate that an additional count element should be created. In addition, the L flag is set to establish a length element that indicates the number of characters with which the application populates the CARRAY1 field.

Listing 3-1 View Description File for FML VIEW

```
$ /* View structure */
  VIEW MYVIEW
  #type     cname      fbname    count    flag     size     null
  float     float1     FLOAT1     1        -        -        0.0
  double    double1    DOUBLE1    1        -        -        0.0
  long      long1      LONG1      1        -        -        0
  short     short1     SHORT1     1        -        -        0
  int       int1       INT1       1        -        -        0
  dec_t     dec1       DEC1       1        -        9,16     0
  char      char1      CHAR1      1        -        -        '\0'
  string    string1    STRING1    1        -        20       '\0'
  carray    carray1    CARRAY1    2        CL       20       '\0'
  END
```

The following listing illustrates the same view description file for an independent `VIEW`.

Listing 3-2 View Description File for an Independent View

```
$ /* View data structure */
  VIEW MYVIEW
  #type    cname    fbname  count    flag     size     null
  float    float1   -       1        -        -        -
  double   double1  -       1        -        -        -
  long     long1    -       1        -        -        -
  short    short1   -       1        -        -        -
  int      int1     -       1        -        -        -
  dec_t    dec1     -       1        -        9,16     -
  char     char1    -       1        -        -        -
  string   string1  -       1        -        20       -
  carray   carray1  -       2        CL       20       -
  END
```

Note that the format is similar to the `FML`-dependent view, except that the *`fbname`* and *`null`* fields are not relevant and are ignored by the `viewc` compiler. You must include a value (for example, a dash) as a placeholder in these fields.

Executing the VIEW Compiler

To compile a VIEW typed record, run the viewc -C command, specifying the name of the view description file as an argument. To specify an independent VIEW, use the -n option. You can optionally specify a directory in which the resulting output file should be written. By default, the output file is written to the current directory.

For example, for an FML-dependent VIEW, the compiler is invoked as follows:

```
viewc -C myview.v
```

Note: To compile a VIEW32 typed record, run the viewc32 -C command.

For an independent VIEW, use the -n option on the command line, as follows:

```
viewc -C -n myview.v
```

The output of the viewc command includes:

- One or more COBOL COPY files; for example, MYVIEW.cbl
- Header file containing a structure definition that may be used by application programs for C routines that share the same view
- Binary version of the source description file; for example, myview.V

  Note: On case-insensitive platforms (for example, Microsoft Windows), the extension used for the names of such files is vv; for example, myview.vv.

The following listing provides an example of the COBOL COPY file created by viewc.

Listing 3-3 COBOL COPY File Example

```
*    VIEWFILE: "myview.v"
*    VIEWNAME: "MYVIEW"
  05 FLOAT1                      USAGE IS COMP-1.
  05 DOUBLE1                     USAGE IS COMP-2.
  05 LONG1                       PIC S9(9) USAGE IS COMP-5.
  05 SHORT1                      PIC S9(4) USAGE IS COMP-5.
  05 FILLER                      PIC X(02).
  05 INT1                        PIC S9(9) USAGE IS COMP-5.
  05 DEC1.
    07 DEC-EXP                   PIC S9(4) USAGE IS COMP-5.
    07 DEC-POS                   PIC S9(4) USAGE IS COMP-5.
```

```
    07 DEC-NDGTS                 PIC S9(4) USAGE IS COMP-5.
*     DEC-DGTS is the actual packed decimal value
    07 DEC-DGTS                  PIC S9(1)V9(16) COMP-3.
    07 FILLER                    PIC X(07).
  05 CHAR1                       PIC X(01).
  05 STRING1                     PIC X(20).
  05 FILLER                      PIC X(01).
  05 L-CARRAY1 OCCURS 2 TIMES    PIC 9(4) USAGE IS COMP-5.
*    LENGTH OF CARRAY1
  05 C-CARRAY1                   PIC S9(4) USAGE IS COMP-5.
*    COUNT OF CARRAY1
  05 CARRAY1 OCCURS 2 TIMES      PIC X(20).
  05 FILLER                      PIC X(02).
```

COBOL `COPY` files for views must be brought into client programs and service subroutines with `COPY` statements.

In the previous example, the compiler includes `FILLER` files so that the alignment of fields in COBOL code matches the alignment in C code.

The format of the packed decimal value, `DEC1`, is composed of five fields. Four fields—`DEC-EXP`, `DEC-POS`, `DEC-NDGTS`, and `FILLER`—are used only in C (they are defined in the `dec_t` type); they are included in the COBOL record for filler. Do not use these fields in COBOL applications.

The fifth field, `DEC-DGTS`, is used by the system to store the actual packed decimal value. You should use this value within the COBOL program. ATMI calls operate on the `DEC-DGTS` field to:

- Populate the field before the record is passed from a C program to a COBOL program.
- Convert the field back to the `dec_t` type when passed from the COBOL program to the C program.

The only restriction is that a COBOL program cannot directly pass a record to a C function outside of the ATMI interface because the decimal formats in the COBOL program and C function do not match.

Finally, note that the sample COBOL `COPY` file includes an `L-CARRAY1` length field that occurs twice, once for each occurrence of `CARRAY1`, and a `C-CARRAY1` count field.

`viewc` creates a C version of the header file that you can use to mix C and COBOL service and/or client programs.

See Also

- "Using an FML Typed Record" on page 3-14
- "Using an XML Typed Record" on page 3-21
- `viewc, viewc32(1)` in the *BEA Tuxedo Command Reference*

Using an FML Typed Record

The `FML` interface was designed for use with the C language. For COBOL, routines are provided that allow you to convert a received `FML` record type to a COBOL record for processing, and then convert the record back to `FML`.

To use `FML` typed records, you must perform the following steps:

- Set the appropriate environment variables.
- Describe the potential fields in an FML field table.
- Initialize the `FML` record using `FINIT`.
- Create an `FML` header file and specify the header file in a `#include` statement C routines that share the same view in the application.

`FML` routines are used to manipulate typed records, including those that convert fielded records to C structures and vice versa. By using these functions, you can access and update data values without having to know how data is structured and stored. For more information on `FML` routines, refer to the *BEA Tuxedo ATMI FML Function Reference*.

Setting Environment Variables for an FML Typed Record

To use an `FML` typed record in an application program, you must set the following environment variables.

Table 3-4 FML Typed Record Environment Variables

Environment Variable	Description
FIELDTBLS or FIELDTBLS32	Comma-separated list of field table filenames for FML or FML32 typed records, respectively.
FLDTBLDIR or FLDTBLDIR32	Colon-separated list of directories to search for the field table files for FML and FML32, respectively. For Microsoft Windows, use a semicolon-separated list.

Creating a Field Table File

Field table files are always required when FML records and/or FML-dependent VIEWs are used. A field table file maps the logical name of a field in an FML record to a string that uniquely identifies the field.

The following format is used for the description of each field in the FML field table:

```
$ /* FML structure */
   *base value
   name    number     type   flags   comments
```

The following table describes the fields that must be specified in the FML field table file for each FML field.

Table 3-5 Field Table File Fields

Field	Description
`*base value`	Specifies a base for offsetting subsequent field numbers, providing an easy way to group and renumber sets of related fields. The `*base` option allows field numbers to be reused. For a 16-bit record, the base plus the relevant number must be greater than or equal to 100 and less than 8191. This field is optional. **Note:** The BEA Tuxedo system reserves field numbers 1-100 and 6000-7000 for internal use. Field numbers 101-8191 are available for application-defined fields with `FML`; field numbers 101-33, 554, and 431, for `FML32`.
name	Identifier for the field. The value must be a string of up to 30 characters, consisting of alphanumeric and underscore characters only.
rel-number	Relative numeric value of the field. This value is added to the current base, if specified, to calculate the field number.
type	Type of the field. This value can be any of the following: `char`, `string`, `short`, `long`, `float`, `double`, or `carray`.
flag	Reserved for future use. A dash (-) should be included as a placeholder.
comment	Optional comment.

All fields are optional, and may be included more than once.

The following example illustrates a field table file that may be used with the FML-dependent VIEW example.

Listing 3-4 Field Table File for FML VIEW

```
# name        number    type      flags    comments
  FLOAT1      110       float     -        -
  DOUBLE1     111       double    -        -
  LONG1       112       long      -        -
  SHORT1      113       short     -        -
  INT1        114       long      -        -
  DEC1        115       string    -        -
  CHAR1       116       char      -        -
  STRING1     117       string    -        -
  CARRAY1     118       carray    -        -
```

Initializing a Typed Record

An FML typed record must be initialized using the FINIT procedure. The TPINIT procedure takes the specified FML record (preferably aligned on a full-word boundary) and uses the value specified in the FML-LENGTH field in the FMLINFO record as the length.

If TPNOCHANGE is set, then any FML record received by a program (rather than created by the program) is initialized automatically. In this case, it is unnecessary to call FINIT.

The following listing shows how to perform an initialization.

Listing 3-5 FML/VIEW Conversion

```
WORKING-STORAGE SECTION.
*RECORD TYPE AND LENGTH
 01 TPTYPE-REC.
        COPY TPTYPE.
*STATUS OF CALL
 01 TPSTATUS-REC.
        COPY TPSTATUS.
* SERVICE CALL FLAGS/RECORD
```

```
 01 TPSVCDEF-REC.
        COPY TPSVCDEF.
* TPINIT FLAGS/RECORD
 01 TPINFDEF-REC.
        COPY TPINFDEF.
* FML CALL FLAGS/RECORD
 01 FML-REC.
        COPY FMLINFO.
*
*
* APPLICATION FML RECORD - ALIGNED
 01 MYFML.
     05 FBFR-DTA OCCURS 100 TIMES   PIC S9(9) USAGE IS COMP-5.
* APPLICATION VIEW RECORD
 01 MYVIEW.
        COPY MYVIEW.

.....

* MOVE DATA INTO MYVIEW

.....

* INITIALIZE FML RECORD
 MOVE LENGTH OF MYFML TO FML-LENGTH.
 CALL "FINIT" USING MYFML FML-REC.
 IF NOT FOK
        MOVE "FINIT Failed" TO LOGMSG-TEXT
        PERFORM DO-USERLOG
        PERFORM EXIT-PROGRAM
 END-IF.

* Convert VIEW to FML Record
 SET FUPDATE TO TRUE.
 MOVE "MYVIEW" TO VIEWNAME.
 CALL "FVSTOF" USING MYFML MYVIEW FML-REC.
 IF NOT FOK
        MOVE "FVSTOF Failed" TO LOGMSG-TEXT
        PERFORM DO-USERLOG
        PERFORM EXIT-PROGRAM
 END-IF.
* CALL THE SERVICE USING THE FML RECORD
 MOVE "FML" TO REC-TYPE IN TPTYPE-REC.
 MOVE SPACES TO SUB-TYPE IN TPTYPE-REC.
 MOVE LENGTH OF MYFML TO LEN.
 CALL "TPCALL" USING TPSVCDEF-REC
        TPTYPE-REC
        MYFML
        TPTYPE-REC
        MYFML
```

```
        TPSTATUS-REC.
 IF NOT TPOK
        MOVE "TPCALL MYFML Failed" TO LOGMSG-TEXT
        PERFORM DO-USERLOG
        PERFORM EXIT-PROGRAM
 END-IF.
* CONVERT THE FML RECORD BACK TO MYVIEW
 CALL "FVFTOS" USING MYFML MYVIEW FML-REC.
 IF NOT FOK
        MOVE "FVFTOS Failed" TO LOGMSG-TEXT
        PERFORM DO-USERLOG
        PERFORM EXIT-PROGRAM
 END-IF.
```

In the preceding listing, the FVSTOF procedure converts an FML record into a VIEW record. The view is defined by including the copy file generated by the view compiler. The FML-REC record provides the VIEWNAME and the FML-MODE transfer mode, which can be set to FUPDATE, FOJOIN, FJOIN, or FCONCAT. The actions associated with these modes are the same as those described in Fupdate, Fupdate32(3fml), Fojoin, Fojoin32(3fml), Fjoin, Fjoin32(3fml), and Fconcat, Fconcat32(3fml).

The FVFTOS procedure converts a VIEW record into an FML record. The parameters are the same as those for an FVSTOF procedure but you do not need to set FML-MODE. The system copies the fields from the fielded record into the structure, based on the element descriptions in the view. If there is no corresponding element in the COBOL record for a field in the fielded record, then the system ignores the field. If there is no corresponding field in the fielded record for an element specified in the COBOL record, the system copies a null value into the element. The null value used can be defined for each element in the view description.

To store multiple occurrences of a field in the COBOL record, a record element should be defined with OCCURS. If the number of occurrences of the field in the record is smaller than the number of occurrences of the element, the extra element slots are assigned null values. Alternatively, if the number of occurrences of the field in the record is higher than the number of occurrences of the element, then the surplus occurrences are ignored.

For FML32 and VIEW32, the FINIT32, FVSTOF32, and FVFTOS32 procedures should be used.

Upon successful completion, the system sets the FML-STATUS to FOK. On error, the system sets the FML-STATUS to a non-zero value.

Creating an FML Header File

In order to use an FML typed record in client programs or service subroutines, you must create an FML header file and specify it in the application #include statements.

To create an FML header file from a field table file, use the mkfldhdr(1) command. For example, to create a file called myview.flds.h, enter the following command:

```
mkfldhdr myview.flds
```

For FML32 typed records, use the mkfldhdr32 command.

The following listing shows the myview.flds.h header file that is created by the mkfldhdr command.

Listing 3-6 myview.flds.h Header File

```
/*       fname     fldid            */
/*       -----     -----            */

#define  FLOAT1     ((FLDID)24686)   /* number: 110 type: float  */
#define  DOUBLE1    ((FLDID)32879)   /* number: 111 type: double */
#define  LONG1      ((FLDID)8304)    /* number: 112 type: long   */
#define  SHORT1     ((FLDID)113)     /* number: 113 type: short  */
#define  INT1       ((FLDID)8306)    /* number: 114 type: long   */
#define  DEC1       ((FLDID)41075)   /* number: 115 type: string */
#define  CHAR1      ((FLDID)16500)   /* number: 116 type: char   */
#define  STRING1    ((FLDID)41077)   /* number: 117 type: string */
#define  CARRAY1    ((FLDID)49270)   /* number: 118 type: carray */
```

Specify the new header file in the #include statement of your application. Once the header file is included, you can refer to fields by their symbolic names.

See Also

- "Using a VIEW Typed Record" on page 3-7
- "Using an XML Typed Record" on page 3-21

- mkfldhdr, mkfldhdr32(1) in the *BEA Tuxedo Command Reference*

Using an XML Typed Record

XXML records enable BEA Tuxedo applications to use XML for exchanging data within and between applications. BEA Tuxedo applications can send and receive simple XML records, and route those records to the appropriate servers. All logic for dealing with XML documents, including parsing, resides in the application.

An XML document consists of:

- A sequence of characters that encode the text of a document
- A description of the logical structure of the document and information about that structure

Formatting and filtering for Events processing (which are supported when a STRING record type is used) are not supported for the XML record type. Therefore, the _tmfilter and _tmformat pointers in the record type switch for XML records are set to LOW-VALUE.

The XML parser in the BEA Tuxedo system performs the following routines:

- Autodetection of character encodings
- Character code conversion
- Detection of element content and attribute values
- Data type conversion

Data-dependent routing is supported for XML records. The routing of an XML document can be based on element content, or on element type and an attribute value. The XML parser determines the character encoding being used; if the encoding differs from the native character sets (US-ASCII or EBCDIC) used in the BEA Tuxedo configuration files (UBBCONFIG and DMCONFIG), the element and attribute names are converted to US-ASCII or EBCDIC.

Attributes configured for routing must be included in an XML document. If an attribute is configured as a routing criteria but it is not included in the XML document, routing processing fails.

The content of an element and the value of an attribute must conform to the syntax and semantics required for a routing field value. The user must also specify the type of the routing field value. `XML` supports only character data. If a range field is numeric, the content or value of that field is converted to a numeric value during routing processing.

See Also

- "Using a VIEW Typed Record" on page 3-7
- "Using an FML Typed Record" on page 3-14

4 Writing Clients

This topic includes the following sections:

- Joining an Application
- Using Features of the TPINFDEF-REC Record
- Leaving the Application
- Building Clients
- Client Process Examples

Joining an Application

Before an ATMI client can perform any service request, it must join the BEA Tuxedo ATMI application, either explicitly or implicitly. Once the client has joined the application, it can initiate requests and receive replies.

A client joins an application explicitly by calling `TPINITIALIZE(3cbl)` with the following signature:

```
01 TPINFDEF-REC.
   COPY TPINFDEF.
01 USER-DATA-REC      PIC X(any-length).
01 TPSTATUS-REC.
   COPY TPSTATUS.
CALL "TPINITIALIZE" USING TPINFDEF-REC USER-DATA-REC TPSTATUS-REC.
```

A client joins an application implicitly by issuing a service request (or any ATMI call) without first calling `TPINITIALIZE`. In this case, `TPINITIALIZE` is called by the BEA Tuxedo system on behalf of the client with the `SPACES` parameter.The `TPINFDEF-REC`

record is a special BEA Tuxedo system typed record used by a client program to pass client identification and authentication information to the system when the client attempts to join the application. It is defined in a COBOL COPY file, as follows:

```
05 USRNAME           PIC X(30).
05 CLTNAME           PIC X(30).
05 PASSWD            PIC X(30).
05 GRPNAME           PIC X(30).
05 NOTIFICATION-FLAG PIC S9(9) COMP-5.
  88 TPU-SIG           VALUE 1.
  88 TPU-DIP           VALUE 2.
  88 TPU-IGN           VALUE 3.
05 ACCESS-FLAG       PIC S9(9) COMP-5.
  88 TPSA-FASTPATH     VALUE 1.
  88 TPSA-PROTECTED    VALUE 2.
05 DATALEN           PIC S9(9) COMP-5.
```

The following table lists the fields that are defined in a COBOL COPY file.

Table 4-1 COBOL COPY File Fields

Field	Description
USRNAME	Name representing the caller. You may want to specify the value returned by the UNIX command getuid(2) within this field. The value of USRNAME may contain up to MAXTIDENT characters (which is defined as 30).
CLTNAME	Name of a client for which the semantics are application-defined. The value of CLTNAME may contain up to MAXTIDENT characters (which is defined as 30).
PASSWD	Application password in unencrypted format that is used by TPINITIALIZE for validation against the application password stored in the TUXCONFIG file. PASSWD is a string of up to MAXTIDENT characters.
GRPNAME	Resource manager group name with which you want to associate the client. The client can access an XA-compliant resource manager as part of a global transaction. The GRPNAME can be a value up to MAXTIDENT characters (which is defined as 30). Currently, however, the GRPNAME must be passed as SPACES specifying that the client is not associated with a resource manager group and is in the default client group.

Field	Description
NOTIFICATION-FLAG	Notification mechanism and system access mode to be used. Refer to "Unsolicited Notification Handling" on page 4-5 for a list of valid values.
ACCESS-FLAG	System access mode used. Refer to "System Access Mode" on page 4-7 for a list of values.
DATALEN	Length of the application-specific data that will be sent to the authentication service. For native clients, it is not encoded by the system, but passed to the authentication service as provided by the client. For workstation clients, client authentication is handled by the system, and passed over the network in encrypted form.

The USRNAME and CLTNAME fields are associated with the client process when TPINITIALIZE is called. Both fields are used for both broadcast notification and the retrieval of administrative statistics.

See Also

- TPINITIALIZE(3cbl) in the *BEA Tuxedo ATMI COBOL Function Reference*

Using Features of the TPINFDEF-REC Record

The ATMI client must explicitly invoke TPINITIALIZE in order to take advantage of the following features of the TPINFDEF-REC record:

- Client Naming
- Unsolicited Notification Handling
- System Access Mode
- Resource Manager Association

- Client Authentication

Client Naming

When an ATMI client joins an application, the BEA Tuxedo system assigns a unique client identifier to it. The identifier is passed to each service called by the client. It can also be used for unsolicited notification.

You can also assign unique client and usernames of up to 30 characters each, by passing them to `TPINITIALIZE` via the `TPINFDEF-REC` record. The BEA Tuxedo system establishes a unique identifier for each process by combining the client and usernames associated with it, with the logical machine identifier (LMID) of the machine on which the process is running. You may choose a method for acquiring the values for these fields.

Note: If a process is executing outside the administrative domain of the application (that is, if it is running on a workstation connected to the administrative domain), the LMID of the machine used by the Workstation client to access the application is assigned.

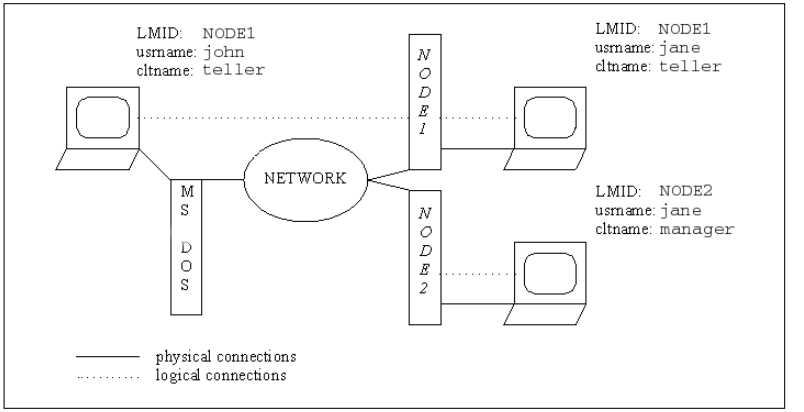
Once a unique identifier for a client process is created:

- Client authentication can be implemented.
- Unsolicited messages can be sent to a specific client or to groups of clients via `TPNOTIFY` and `TPBROADCAST`.
- Detailed statistical information can be gathered via `tmadmin(1)`.

Refer to "Writing Event-based Clients and Servers" for information on sending and receiving unsolicited messages, and the *BEA Tuxedo ATMI C Function Reference* for more information on `tmadmin(1)`.

The following figure shows how names might be associated with clients accessing an application. In the example, the application uses the `cltname` field to indicate a job function.

Figure 4-1 Client Naming



Unsolicited Notification Handling

Unsolicited notification refers to any communication with an ATMI client that is not an expected response to a service request (or an error code). For example, an administrator may broadcast a message to indicate that the system will go down in five minutes.

A client can be notified of an unsolicited message in a number of ways. For example, some operating systems might send a signal to the client and interrupt its current processing. By default, the BEA Tuxedo system checks for unsolicited messages each time an ATMI call is invoked. This approach, referred to as *dip-in,* is advantageous because it:

- Is supported on all platforms
- Does not interrupt the current processing

As some time may elapse between "dip-ins," the application can call the `TPCHKUNSOL` call to check for any waiting unsolicited messages. Refer to "Writing Event-based Clients and Servers" on page 8-1 for more information on the `TPCHKUNSOL` call.

When a client joins an application using TPINITIALIZE, it can control how to handle unsolicited notification messages by defining flags. For client notification, the possible values for NOTIFICATION-FLAG are defined in the following table.

Table 4-2 Client Notification Flags in a TPINFDEF-REC Record

Flag	Description
TPU_SIG	Select unsolicited notification by signals. This flag should be used only with single-threaded, single-context applications. The advantage of using this mode is immediate notification. The disadvantages include: ■ The calling process must have the same *UID* as the sending process when you are running a native client. (Workstation clients do not have this limitation.) ■ TPU_SIG is not available on all platforms (specifically, it is not available on MS-DOS workstations). If you specify this flag but do not meet the system or environmental requirements, the flag is set to TPU_DIP and the event is logged.
TPU_DIP (default)	Select unsolicited notification by dip-in. In this case, the client can specify the name of the message handling routine using the TPSETUNSOL call, and check for waiting unsolicited messages using the TPCHKUNSOL call.
TPU_THREAD	Select THREAD notification in a separate thread. This flag is allowed only on platforms that support multithreading. If TPU_THREAD is specified on a platform that does not support multithreading, it is considered an invalid argument. As a result, an error is returned and TP-STATUS is set to TPEINVAL.
TPU_IGN	Ignore unsolicited notification.

Refer to TPINITIALIZE(3cbl) in the *BEA Tuxedo ATMI COBOL Function Reference* for more information on the TPINFDEF-REC flags.

System Access Mode

An application can access the BEA Tuxedo system through either of two modes: protected or fastpath. The ATMI client can request a mode when it joins an application using TPINITIALIZE. To specify a mode, a client passes one of the following values in the ACCESS-FLAG field of the TPINFDEF-REC record to TPINITIALIZE.

Table 4-3 System Access Flags in a TPINFDEF-REC Record

Mode	Description
TPSA-PROTECTED	Allows ATMI calls within an application to access the BEA Tuxedo system internal tables via shared memory, but protects shared memory against access by application code outside of the BEA Tuxedo system libraries. Overrides the value in UBBCONFIG, except when NO_OVERRIDE is specified. Refer to *Setting Up a BEA Tuxedo Application* for more information on UBBCONFIG.
TPSA-FASTPATH (default)	Allows ATMI calls within application code access to BEA Tuxedo system internals via shared memory. Does not protect shared memory against access by application code outside of the BEA Tuxedo system libraries. Overrides the value of UBBCONFIG except when NO_OVERRIDE is specified. Refer to *Setting Up a BEA Tuxedo Application* for more information on UBBCONFIG.

Resource Manager Association

An application administrator can configure groups for servers associated with a resource manager, including servers that provide administrative processes for coordinating transactions. Refer to *Setting Up a BEA Tuxedo Application* for information on defining groups.

When joining the application, a client can join a particular group by specifying the name of that group in the *grpname* field of TPINFDEF-REC.

Client Authentication

The BEA Tuxedo system provides security at incremental levels, including operating system security, application password, user authentication, optional access control lists, mandatory access control lists, and link-level encryption. Refer to *Setting Up a BEA Tuxedo Application* for information on setting security levels.

The application password security level requires every client to provide an application password when it joins the application. The administrator can set or change the application password and must provide it to valid users.

If this level of security is used, BEA Tuxedo system-supplied client programs, such as `ud()`, prompt for the application password. (Refer to *Administering a BEA Tuxedo Application at Run Time* for more information on `ud, wud(1)`.) In turn, application-specific client programs must include code for obtaining the password from a user. The unencrypted password is placed in the `TPINFDEF-REC` record and evaluated when the client calls `TPINITIALIZE` to join the application.

Note: The password should not be displayed on the screen.

You can use `TPCHKAUTH(3cbl)` to determine:

- Whether the application requires any authentication
- If the application requires authentication, which of the following types of authentication is needed:
  - System authentication based on an application password
  - Application authentication based on an application password and user-specific information

Typically, a client should call `TPCHKAUTH` before `TPINITIALIZE` to identify any additional security information that must be provided during initialization.

Refer to *Using Security in CORBA Applications* for more information on security programming techniques.

Leaving the Application

Once all service requests have been issued and replies received, the ATMI client can leave the application using `TPTERM(3cbl)`. The `TPTERM` call signature is as follows:

```
01 TPSTATUS-REC.
   COPY TPSTATUS.
CALL "TPTERM" USING TPSTATUS-REC.
```

Building Clients

To build an executable ATMI client, compile your application with the BEA Tuxedo system libraries and all other referenced files using the `buildclient(1)` command. Include the `-C` option to indicate that you are compiling a COBOL program. Use the following syntax for the `buildclient` command:

```
buildclient -C filename.cbl -o filename -f filenames -l filenames
```

The following table describes the options to the `buildclient` command.

Table 4-4 buildclient Options

This Option or Argument . . .	Allows You to Specify . . .
filename`.cbl`	The COBOL application to be compiled.
`-o` *filename*	The executable output file. The default name for the output file is `a.out`.
`-f` *filenames*	A list of files that are to be link edited before the BEA Tuxedo system libraries are link edited. You can specify `-f` more than once on the command line, and you can include multiple filenames for each occurrence of `-f`. If you specify a COBOL program file (*file*`.cbl`), it is compiled before it is linked. You can specify other object files (*file*`.o`) separately, or in groups in an archive file (*file*`.a`).

This Option or Argument . . .	Allows You to Specify . . .
`-l` *`filenames`*	A list of files that are to be link edited after the BEA Tuxedo system libraries are link edited. You can specify `-l` more than once on the command line, and you can include multiple filenames for each occurrence of `-l`. If you specify a COBOL program file (*`file`*`.cbl`), it is compiled before it is linked. You can specify other object files (*`file`*`.o`) separately, or in groups in an archive file (*`file`*`.a`).
`-r`	The resource manager has access to libraries that should be link edited with the executable server. The application administrator is responsible for predefining all valid resource manager information in the `$TUXDIR/udataobj/RM` file using the `buildtms(1)` command. Only one resource manager can be specified. Refer to *Setting Up a BEA Tuxedo Application* for more information.

Note: The BEA Tuxedo libraries are linked in automatically; you do not need to specify any BEA Tuxedo libraries on the command line.

The order in which you specify the library files to be link edited is significant: it depends on the order in which functions are called in the code, and which libraries contain references to those functions.

By default, the `buildclient` command invokes the UNIX `cc` command. You can set the `ALTCC` and `ALTCFLAGS` environment variables to specify an alternative compile command, and to set flags for the compile and link-edit phases, respectively. By default, `ALTCC` is set to `cobcc`. For more information, refer to "Setting Environment Variables" on page 2-6.

Note: On a Windows 2000 system, the `ALTCC` and `ALTCFLAGS` environment variables are not applicable; setting them will produce unexpected results. You must compile your application by first using a COBOL compiler, and then passing the resulting object file to the `buildclient` command. For example:

```
buildclient -C -o audit -f audit.o
```

The following example command line compiles a COBOL program called `audit.cbl` and generates an executable file named `audit`.

```
buildclient -C -o audit -f audit.cbl
```

See Also

- "Building Servers" on page 5-32
- `buildclient(1)` in the *BEA Tuxedo Command Reference*

Client Process Examples

The following pseudo-code shows how a typical ATMI client process works from the time at which it joins an application to the time at which it leaves the application.

Listing 4-1 Typical Client Process Paradigm

```
. . .
Check level of security
    CALL TPSETUNSOL to name your handler routine for TPU-DIP
    get USRNAME, CLTNAME
    prompt for application PASSWD
    SET TPU-DIP TO TRUE.
    CALL "TPINITIALIZE" USING TPINFDEF-REC
                  USER-DATA-REC
                  TPSTATUS-REC.
    IF NOT TPOK
       error processing
. . .
make service call
receive the reply
check for unsolicited messages
. . .
CALL "TPTERM" USING TPSTATUS-REC.
IF NOT TPOK
        error processing
. . .
EXIT PROGRAM.
```

In this example, `TPINITIALIZE` takes three arguments:

- TPINFDEF-REC, a structure defined in the COBOL COPY file
- User data (USER-DATA-REC)
- TPSTATUS-REC, a status structure defined in the COBOL COPY file

Both TPINITIALIZE and TPTERM return [TPOK] in TP-STATUS IN TPSTATUS-REC upon success. If either command encounters an error, the command fails and sets TP-STATUS to a value that indicates the nature of the error. TPSTATUS-REC is defined in a COBOL COPY file. Refer to "Managing Errors" on page 11-1 for possible TP-STATUS values. Refer to "Introduction to the COBOL Application-Transaction Monitor Interface" in the *BEA Tuxedo ATMI COBOL Function Reference* for a complete list of error codes that can be returned for each of the ATMI calls.

The following example illustrates how to use the TPINITIALIZE and TPTERM routines. This example is borrowed from, bankapp, the sample banking application that is provided with the BEA Tuxedo system.

Listing 4-2 Joining and Leaving an Application

```
IDENTIFICATION DIVISION.
PROGRAM-ID. FIG1-3.
AUTHOR. TUXEDO DEVELOPMENT.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
*
WORKING-STORAGE SECTION.
*****************************************************
* Tuxedo definitions
*****************************************************
01 TPSTATUS-REC.
COPY TPSTATUS.
*
01 TPINFDEF-REC.
COPY TPINFDEF.
*****************************************************
* Log messages definitions
*****************************************************
01 LOGMSG.
     05 FILLER         PIC X(10) VALUE "FIG12-3 =>".
     05 LOGMSG-TEXT    PIC X(50).
01 LOGMSG-LEN          PIC S9(9) COMP-5.
*
01 USER-DATA-REC  PIC X(75).
*****************************************************
```

```
PROCEDURE DIVISION.
START-HERE.
MOVE LENGTH OF LOGMSG TO LOGMSG-LEN.
*****************************************************
* Now register the client with the system.
*****************************************************
MOVE SPACES TO USRNAME.
MOVE SPACES TO CLTNAME.
MOVE SPACES TO PASSWD.
MOVE SPACES TO GRPNAME.
MOVE ZERO TO DATALEN.
SET TPU-DIP TO TRUE.
*
CALL "TPINITIALIZE" USING TPINFDEF-REC
         USER-DATA-REC
         TPSTATUS-REC.
IF NOT TPOK
        MOVE "TPINITIALIZE FAILED" TO LOGMSG-TEXT
        PERFORM DO-USERLOG
        PERFORM EXIT-PROGRAM.
*****************************************************
* Application specific code
*****************************************************
. . .
*****************************************************
*Leave Application
*****************************************************
CALL "TPTERM" USING TPSTATUS-REC.
IF  NOT TPOK
    MOVE "TPTERM FAILED" TO LOGMSG-TEXT
    PERFORM DO-USERLOG.
EXIT-PROGRAM.
STOP RUN.
*****************************************************
* Log messages to the userlog
*****************************************************
DO-USERLOG.
CALL "USERLOG" USING LOGMSG
      LOGMSG-LEN
      TPSTATUS-REC.
```

The previous example shows the client process attempting to join the application with a call to TPINITIALIZE. If an error is encountered, a message is written to the central event log via a call to USERLOG.

CHAPTER

5 Writing Servers

This topic includes the following sections:

- BEA Tuxedo System Controlling Program
- System-supplied Server and Services
- Guidelines for Writing Servers
- Defining a Service
- Terminating a Service Routine
- Advertising and Unadvertising Services
- Building Servers

BEA Tuxedo System Controlling Program

To facilitate the development of ATMI servers, the BEA Tuxedo system provides a predefined controlling program for server load modules. When you execute the `buildserver -C` command, the controlling program is automatically included as part of the server.

Note: The controlling program that the system provides is a closed abstraction; you cannot modify it.

In addition to joining and exiting from an application, the predefined controlling program accomplishes the following tasks on behalf of the server.

- Executes the process ignoring any hangups (that is, it ignores the SIGHUP signal).
- Initiates the cleanup process on receipt of the standard operating system software termination signal (SIGTERM). The server is shut down and must be rebooted if needed again.
- Attaches to shared memory for bulletin board services.
- Creates a message queue for the process.
- Advertises the initial services to be offered by the server. The initial services are either all the services link edited with the predefined controlling program, or a subset specified by the BEA Tuxedo system administrator in the configuration file.
- Processes command-line arguments up to the double dash (--), which indicates the end of system-recognized arguments.
- Calls the routine TPSVRINIT to process any command-line arguments listed after the double dash (--) and optionally to open the resource manager. These command-line arguments are used for application-specific initialization.
- Until ordered to halt, checks its request queue for service request messages.
- When a service request message arrives on the request queue, main() performs the following tasks until ordered to halt:
  - If the -r option is specified, records the starting time of the service request.
  - Updates the bulletin board to indicate that the server is BUSY.
  - Dispatches the service; that is, calls the service subroutine.
- When the service returns from processing its input, main() performs the following tasks until ordered to halt:
  - If the -r option is specified, records the ending time of the service request.
  - Updates statistics.
  - Updates the bulletin board to indicate that the server is IDLE; that is, that the server is ready for work.
  - Checks its queue for the next service request.

- When the server is required to halt, calls TPSVRDONE to perform any required shutdown operations.

As indicated above, the main() routine handles all of the details associated with joining and exiting from an application, managing records and transactions, and handling communication.

Note: Because the system-supplied controlling program accomplishes the work of joining and leaving the application, you should not include calls to the TPINITIALIZE or TPTERM routine in your code. If you do, the routine encounters an error and returns TPEPROTO in TP-STATUS. For more information on the TPINITIALIZE or TPTERM routine, refer to "Writing Clients" on page 4-1.

System-supplied Server and Services

The controlling program provides one system-supplied ATMI server, AUTHSVR, and two subroutines, TPSVRINIT and TPSVRDONE. The default versions of all three, which are described in the following sections, can be modified to suit your application.

Notes: If you want to write your own versions of TPSVRINIT and TPSVRDONE, remember that the default versions of these two routines call tx_open() and tx_close(), respectively. If you write a new version of TPSVRINIT that calls tpopen() rather than tx_open(), you should also write a new version of TPSVRDONE that calls tpclose(). In other words, both routines in an open/close pair must belong to the same set.

System-supplied Server: AUTHSVR( )

You can use the AUTHSVR(5) server to provide individual client authentication for an application. The TPINITIALIZE routine calls this server when the level of security for the application is TPAPPAUTH, USER_AUTH, ACL, or MANDATORY_ACL.

The service in AUTHSVR looks in the USER-DATA-REC record for a user password (not to be confused with the application password specified in the PASSWD field of the TPINFDEF-REC record). By default, the system takes the string in data and searches for a matching string in the /etc/passwd file.

When called by a native-site client, TPINITIALIZE forwards the USER-DATA-REC record as it is received. This means that if the application requires the password to be encrypted, the client program must be coded accordingly.

When called by a Workstation client, TPINITIALIZE encrypts the data before sending it across the network.

System-supplied Services: TPSVRINIT Routine

When a server is booted, the BEA Tuxedo system controlling program calls TPSVRINIT(3cbl) during its initialization phase, before handling any service requests.

If an application does not provide a custom version of this routine within the server, the system uses the default routine provided by the controlling program, which opens the resource manager and logs an entry in the central event log indicating that the server has successfully started. The central user log is an automatically generated file to which processes can write messages by calling the USERLOG(3cbl) routine. Refer to "Managing Errors" on page 11-1 for more information on the central event log.

You can use the TPSVRINIT routine for any initialization processes that might be required by an application, such as the following:

- Receiving command-line options
- Opening a database

The following sections provide code samples showing how these initialization tasks are performed through calls to TPSVRINIT. Although it is not illustrated in the following examples, message exchanges can also be performed within this routine. However, TPSVRINIT fails if it returns with asynchronous replies pending. In this case, the replies are ignored by the BEA Tuxedo system, and the server exits gracefully.

You can also use the TPSVRINIT routine to start and complete transactions, as described in "Managing Errors" on page 11-1.

Use the following signature to call the TPSVRINIT :routine

```
LINKAGE SECTION.
01 CMD-LINE.
     05 ARGC          PIC 9(4) COMP-5.
     05 ARGV.
          10 ARGS PIC X OCCURS 0 TO 9999 DEPENDING ON ARGC.
01 TPSTATUS-REC.
   COPY TPSTATUS.
PROCEDURE DIVISION USING CMD-LINE TPSTATUS-REC.
* User code
EXIT PROGRAM.
```

Receiving Command-line Options

When a server is booted, its first task is to read the server options specified in the configuration file. The options are passed through ARGC, which contains the number of arguments, and ARGV, which contains the arguments separated by a single SPACE character. The predefined controlling program then calls TPSVRINIT.

The following code example shows how the TPSVRINIT routine is used to receive command-line options.

Listing 5-1 Receiving Command-line Options in TPSVRINIT

```
  IDENTIFICATION DIVISION.
  PROGRAM-ID. TPSVRINIT.
  ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  SOURCE-COMPUTER. USL-486.
  OBJECT-COMPUTER. USL-486.
*
  DATA DIVISION.
  WORKING-STORAGE SECTION.
*
  LINKAGE SECTION.
*
  01 CMD-LINE.
      05 ARGC PIC 9(4) COMP-5.
      05 ARGV.
           10 ARGS PIC X OCCURS 0 TO 9999 DEPENDING ON ARGC.
  01 SERVER-INIT-STATUS.
  COPY TPSTATUS.
*
 PROCEDURE DIVISION USING CMD-LINE SERVER-INIT-STATUS.
*********************************************************
* ARGC indicates the number of arguments and ARGV contains the
```

```
* arguments separated by a single SPACE.
*********************************************************
  A-START.
*
   . . . INSPECT the ARGV line and process arguments
   IF arguments are invalid
         SET TPEINVAL IN SERVER-INIT-STATUS TO TRUE.
   ELSE arguments are OK continue
         SET TPOK IN SERVER-INIT-STATUS TO TRUE.
*
   EXIT PROGRAM.
```

Opening a Resource Manager

The following example illustrates another common use of `TPSVRINIT`: opening a resource manager. The BEA Tuxedo system provides routines to open a resource manager, `TPOPEN(3cbl)` and `TXOPEN(3cbl)`. It also provides the complementary routines, `TPCLOSE(3cbl)` and `TXCLOSE(3cbl)`. Applications that use these routines to open and close their resource managers are portable in this respect. They work by accessing the resource manager instance-specific information that is available in the configuration file.

These routine calls are optional and can be used in place of the resource manager specific calls that are sometimes part of the Data Manipulation Language (DML) if the resource manager is a database. Note the use of the `USERLOG(3cbl)` routine to write to the central event log.

Note: To create an initialization function that both receives command-line options and opens a database, combine the following example with the previous example.

Listing 5-2 Opening a Resource Manager in TPSVRINIT

```
  IDENTIFICATION DIVISION.
  PROGRAM-ID. TPSVRINIT.
  ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  SOURCE-COMPUTER. USL-486.
  OBJECT-COMPUTER. USL-486.
*
  DATA DIVISION.
```

```
  WORKING-STORAGE SECTION.
    01 TPSTATUS-REC.
      COPY TPSTATUS.
    01 LOGMSG         PIC X(50).
    01 LOGMSG-LEN    PIC S9(9) COMP-5.
*
  LINKAGE SECTION.
    01 CMD-LINE.
      05 ARGC PIC 9(4) COMP-5.
      05 ARGV.
           10 ARGS PIC X OCCURS 0 TO 9999 DEPENDING ON ARGC.
    01 SERVER-INIT-STATUS.
     COPY TPSTATUS.
*
  PROCEDURE DIVISION USING CMD-LINE SERVER-INIT-STATUS.
  A-START.
    . . . INSPECT the ARGV line and process arguments
    IF arguments are invalid
               MOVE "Invalid Arguments Passed" TO LOGMSG
               PERFORM EXIT-NOW.
    ELSE arguments are OK continue

    CALL "TPOPEN" USING TPSTATUS-REC.
    IF NOT TPOK
          MOVE "TPOPEN Failed" TO LOGMSG
    ELSE IF TPESYSTEM
          MOVE "System /T error has occurred" TO LOGMSG
    ELSE IF TPEOS
          MOVE "An Operating System error has occurred" TO LOGMSG
    ELSE IF TPEPROTO
         MOVE "TPOPEN was called in an improper Context" TO LOGMSG
    ELSE IF TPERMERR
          MOVE "Resource manager Failed to Open" TO LOGMSG
          PERFORM EXIT-NOW.
    SET TPOK IN SERVER-INIT-STATUS TO TRUE.
    EXIT PROGRAM.
 EXIT-NOW.
  SET TPEINVAL IN SERVER-INIT-STATUS TO TRUE
    MOVE 50 LOGMSG-LEN.
    CALL "USERLOG" USING LOGMSG
                  LOGMSG-LEN
                  TPSTATUS-REC.
  EXIT PROGRAM.
```

To guard against errors that may occur during initialization, `TPSVRINIT` can be coded to allow the server to exit gracefully before starting to process service requests.

System-supplied Services: TPSVRDONE Routine

The TPSVRDONE routine calls TPCLOSE to close the resource manager, similarly to the way TPSVRINIT calls TPOPEN to open it.

Use the following signature to call the TPSVRDONE routine:

```
  01 TPSTATUS-REC.
     COPY TPSTATUS.
  PROCEDURE DIVISION.
* User code
  EXIT PROGRAM.
```

The following example illustrates how to use the TPSVRDONE routine to close a resource manager and exit gracefully.

Listing 5-3 Closing a Resource Manager with TPSVRDONE

```
  IDENTIFICATION DIVISION.
  PROGRAM-ID. TPSVRDONE.
  ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  SOURCE-COMPUTER. USL-486.
  OBJECT-COMPUTER. USL-486.
*
  DATA DIVISION.
  WORKING-STORAGE SECTION.
   01 TPSTATUS-REC.
      COPY TPSTATUS.
   01 LOGMSG                PIC X(50).
   01 LOGMSG-LEN            PIC S9(9) COMP-5.
   01 SERVER-DONE-STATUS.
      COPY TPSTATUS.
  PROCEDURE DIVISION.
  A-START.
   CALL "TPCLOSE" USING TPSTATUS-REC.
   IF NOT TPOK
         MOVE "TPCLOSE Failed" TO LOGMSG
   ELSE IF TPESYSTEM
         MOVE "System /T error has occurred" TO LOGMSG
   ELSE IF TPEOS
         MOVE "An Operating System error has occurred" TO LOGMSG
   ELSE IF TPEPROTO
        MOVE "TPCLOSE was called in an improper Context" TO LOGMSG
   ELSE IF TPERMERR
```

```
        MOVE "Resource manager Failed to Open" TO LOGMSG
        PERFORM EXIT-NOW.
  SET TPOK IN SERVER-DONE-STATUS TO TRUE.
  EXIT PROGRAM.
 EXIT-NOW.
  SET TPEINVAL IN SERVER-DONE-STATUS TO TRUE
  MOVE 50 LOGMSG-LEN.
  CALL "USERLOG" USING LOGMSG
                 LOGMSG-LEN
                 TPSTATUS-REC.
  EXIT PROGRAM.
```

Guidelines for Writing Servers

Because the communication details are handled by the BEA Tuxedo system controlling program, you can concentrate on the application service logic rather than communication implementation. For compatibility with the system-supplied controlling program, however, application services must adhere to certain conventions. These conventions are referred to, collectively, as the service template for coding service routines. They are summarized in the following list.

- A request/response service can receive only one request at a time and can send only one reply.
- When processing a request, a request/response service works only on that request. It can accept another only after it has either sent a reply to the requester or forwarded the request to another service for additional processing.
- Service routines must terminate by calling either the TPRETURN or TPFORWAR routine.
- When communicating with another server via TPACALL, the initiating service must either wait for all outstanding replies or invalidate them with TPCANCEL before calling TPRETURN or TPFORWAR.

Defining a Service

When writing a service routine, you must call the `TPSVCSTART(3cbl)` routine before any others. This routine is used to retrieve the service's parameters and data. Use the following signature to call the `TPSVCSTART` routine

```
01 TPSVCDEF-REC.
   COPY TPSVCDEF.
01 TPTYPE-REC.
   COPY TPTYPE.
01 DATA-REC.
   COPY User Data.
01 TPSTATUS-REC.
   COPY TPSTATUS.
CALL "TPSVCSTART" USING TPSVCDEF-REC TPTYPE-REC DATA-REC TPSTATUS-REC.
```

The service information data structure is defined as `TPSVCDEF` in the COBOL `COPY` file. It includes the following members:

```
 05 COMM-HANDLE          PIC S9(9) COMP-5.
 05 TPBLOCK-FLAG         PIC S9(9) COMP-5.
        88 TPBLOCK       VALUE 0.
        88 TPNOBLOCK     VALUE 1.
 05 TPTRAN-FLAG          PIC S9(9) COMP-5.
        88 TPTRAN        VALUE 0.
        88 TPNOTRAN      VALUE 1.
 05 TPREPLY-FLAG         PIC S9(9) COMP-5.
        88 TPREPLY       VALUE 0.
        88 TPNOREPLY     VALUE 1.
 05 TPACK-FLAG         PIC S9(9) COMP-5 REDEFINES TPREPLY-FLAG.
        88 TPNOACK        VALUE 0.
        88 TPACK          VALUE 1.
 05 TPTIME-FLAG          PIC S9(9) COMP-5.
        88 TPTIME        VALUE 0.
        88 TPNOTIME      VALUE 1.
 05 TPSIGRSTRT-FLAG      PIC S9(9) COMP-5.
        88 TPNOSIGRSTRT   VALUE 0.
        88 TPSIGRSTRT     VALUE 1.
 05 TPGETANY-FLAG        PIC S9(9) COMP-5.
        88 TPGETHANDLE   VALUE 0.
        88 TPGETANY      VALUE 1.
 05 TPSENDRECV-FLAG      PIC S9(9) COMP-5.
        88 TPSENDONLY     VALUE 0.
        88 TPRECVONLY     VALUE 1.
 05 TPNOCHANGE-FLAG      PIC S9(9) COMP-5.
```

```
        88 TPCHANGE      VALUE 0.
        88 TPNOCHANGE    VALUE 1.
 05 TPSERVICETYPE-FLAG   PIC S9(9) COMP-5.
        88 TPREQRSP      VALUE 0.
        88 TPCONV        VALUE 1.
*
 05 APPKEY               PIC S9(9) COMP-5.
 05 CLIENTID OCCURS 4 TIMES PIC S9(9) COMP-5.
 05 SERVICE-NAME         PIC X(15).
```

The following table describes the members of a TPSVCDEF data structure.

Table 5-1 TPSVCDEF Data Structure

Field	Description
COMM-HANDLE	Specifies, to the service routine, the communication handle used by the requesting process to invoke the service.
SETTINGS (TPBLOCK-FLAG TPTRAN-FLAG, etc.)	Miscellaneous settings that control server characteristics. For more information on the settings, refer to the *BEA Tuxedo ATMI COBOL Function Reference*.
APPKEY	Reserved for use by the application. If application-specific authentication is part of your design, the application-specific authentication server, which is called at the time a client joins the application, should return a client authentication key, as well as a success or failure indication. The BEA Tuxedo system holds the APPKEY on behalf of the client and passes the information to subsequent service requests in this field. By the time the APPKEY is passed to the service, the client has already been authenticated. However, the APPKEY field can be used within the service to identify the user invoking the service or some other parameters associated with the user.
CLIENTID	Identifier of the client that originates a request.
SERVICE-NAME	Name of the service routine used by the requesting process to invoke the service.

For a description of the TPTYPE-REC data structure, refer to “Defining Typed Records” on page 3-6.

You must code the service in such a way that when it accesses the request data to be placed in DATA-REC, it expects the data to be in a record of the type defined for the service in the configuration file. Upon successful return, DATA-REC contains the data received and LEN contains the actual number of bytes moved.

The following sample listing shows a typical service definition.

Listing 5-4 Typical Service Definition

```
    IDENTIFICATION DIVISION.
    PROGRAM-ID. BUYSR.
    AUTHOR. TUXEDO DEVELOPMENT.
    ENVIRONMENT DIVISION.
    CONFIGURATION SECTION.
    SOURCE-COMPUTER. USL-486.
    OBJECT-COMPUTER. USL-486.
*
    INPUT-OUTPUT SECTION.
    . . .
*****************************************************
* Tuxedo definitions
*****************************************************
    01 TPSVCRET-REC.
    COPY TPSVCRET.
*
    01 TPTYPE-REC.
     COPY TPTYPE.
*
    01 TPSTATUS-REC.
    COPY TPSTATUS.
*
    01 TPSVCDEF-REC.
    COPY TPSVCDEF.
*****************************************************
* Log message definitions
*****************************************************
    01 LOGMSG.
            05 LOGMSG-TEXT        PIC X(50).
*
    01 LOGMSG-LEN                 PIC S9(9) COMP-5.
*****************************************************
* User defined data records
*****************************************************
    01 CUST-REC.
    COPY CUST.
*
```

```
    LINKAGE SECTION.
*
    PROCEDURE DIVISION.
*
 START-BUYSR.
    MOVE LENGTH OF LOGMSG TO LOGMSG-LEN.
    OPEN files or DATABASE
*****************************************************
* Get the data that was sent by the client
*****************************************************
    MOVE "Server Started" TO LOGMSG-TEXT.
    PERFORM DO-USERLOG.
    MOVE LENGTH OF CUST-REC TO LEN IN TPTYPE-REC.
    CALL "TPSVCSTART" USING TPSVCDEF-REC
                      TPTYPE-REC
                      CUST-REC
                      TPSTATUS-REC.
    IF TPTRUNCATE
        MOVE "Input data exceeded CUST-REC length" TO LOGMSG-TEXT
        PERFORM DO-USERLOG
        PERFORM A-999-EXIT.
    IF NOT TPOK
        MOVE "TPSVCSTART Failed" TO LOGMSG-TEXT
        PERFORM DO-USERLOG
        PERFORM A-999-EXIT.
    IF REC-TYPE NOT = "VIEW"
        MOVE "REC-TYPE in not VIEW" TO LOGMSG-TEXT
        PERFORM DO-USERLOG
        PERFORM A-999-EXIT.
    IF SUB-TYPE NOT = "cust"
        MOVE "SUB-TYPE in not cust" TO LOGMSG-TEXT
        PERFORM DO-USERLOG
        PERFORM A-999-EXIT.
     . . .
     set consistency level of the transaction
     . . .
*****************************************************
* Exit
*****************************************************
  A-999-EXIT.
     MOVE "Exiting" TO LOGMSG-TEXT.
     PERFORM DO-USERLOG.
     SET TPFAIL TO TRUE.
     COPY TPRETURN REPLACING TPSVCRET-REC BY TPSVCRET-REC
     TPTYPE-REC BY TPTYPE-REC
     DATA-REC BY CUST-REC
     TPSTATUS-REC BY TPSTATUS-REC.
*****************************************************
* Write to userlog
```

```
*************************************************
  DO-USERLOG.
     CALL "USERLOG" USING LOGMSG
               LOGMSG-LEN
               TPSTATUS-REC.
```

In the preceding example, the request record on the client side was originally sent with REC-TYPE set to VIEW and the SUB-TYPE set to cust. The BUYSR service is defined in the configuration file as a service that knows about the VIEW typed record. BUYSR retrieves the data record by accessing the CUST-REC record. The consistency level of the transaction is specified after this record is retrieved but before the first database access is made. For more details on transaction consistency levels, refer to "Writing Global Transactions" on page 9-1.

Note: The TPGPRIO and TPSPRIO routines, used for getting and setting priorities, respectively, are described in detail in "Setting and Getting Message Priorities" on page 6-16.

The example code in this section shows how a service called PRINTER tests the priority level of the request just received using the TPGPRIO routine. Then, based on the priority level, the application routes the print job to the appropriate destination printer RNAME.

Next, the contents of INPUT-REC are sent to the printer. The application queries TPSVCDEF-REC to determine whether a reply is expected. If so, it returns the name of the destination printer to the client. For more information on the TPRETURN routine, refer to "Terminating a Service Routine" on page 5-18.

Listing 5-5 Checking the Priority of a Received Request

```
    IDENTIFICATION DIVISION.
    PROGRAM-ID. PRINTSR.
    AUTHOR. TUXEDO DEVELOPMENT.
    ENVIRONMENT DIVISION.
    CONFIGURATION SECTION.
    SOURCE-COMPUTER. USL-486.
    OBJECT-COMPUTER. USL-486.
*
    INPUT-OUTPUT SECTION.
    . . .
*************************************************
```

```
* Tuxedo definitions
*******************************************************
    01 TPSVCRET-REC.
    COPY TPSVCRET.
*
    01 TPTYPE-REC.
    COPY TPTYPE.
*
    01 TPSTATUS-REC.
    COPY TPSTATUS.
*
    01 TPSVCDEF-REC.
    COPY TPSVCDEF.
*
    01 TPPRIDEF-REC.
    COPY TPPRIDEF.
*******************************************************
* Log message definitions
*******************************************************
    01 LOGMSG.
            05 FILLER           PIC S9(9) VALUE
                    "TP-STATUS=".
            05 LOG-TP-STATUS    PIC S9(9).
            05 LOGMSG-TEXT      PIC X(50).
*
    01 LOGMSG-LEN  PIC S9(9) COMP-5.
*******************************************************
* User defined data records
*******************************************************
    01 INPUT-REC           PIC X(1000).
    01 PRNAME              PIC X(20).
*
    LINKAGE SECTION.
*
    PROCEDURE DIVISION.
*
 START-PRINTSR.
    MOVE LENGTH OF LOGMSG TO LOGMSG-LEN.
    OPEN files or DATABASE
*******************************************************
* Get the data that was sent by the client
*******************************************************
    MOVE ZERO to TP-STATUS.
    MOVE "Server Started" TO LOGMSG-TEXT.
    PERFORM DO-USERLOG.
    MOVE LENGTH OF INPUT-REC TO LEN.
    CALL "TPSVCSTART" USING TPSVCDEF-REC
                      TPTYPE-REC
                      INPUT-REC
```

```
                        TPSTATUS-REC.
    IF NOT TPOK
             MOVE "TPSVCSTART Failed" TO LOGMSG-TEXT
             PERFORM DO-USERLOG
             SET TPFAIL TO TRUE.
             PERFORM A-999-EXIT.
    . . .
    Check other parameters
    CALL "TPGPRIO" USING TPPRIDEF-REC
                   TPSTATUS-REC.
    IF NOT TPOK
             MOVE "TPGPRIO Failed" TO LOGMSG-TEXT
             PERFORM DO-USERLOG
             SET TPFAIL TO TRUE.
             PERFORM A-999-EXIT.
    IF PRIORITY < 20
             MOVE "BIGJOBS" TO RNAME
    ELSE IF PRIORITY < 60
             MOVE "MEDJOBS" TO RNAME
    ELSE
             MOVE "HIGHSPEED" TO RNAME.
    . . .
    Print INPUT-REC on RNAME printer
    . . .
    IF TPNOREPLY
             MOVE SPACES TO REC-TYPE
             MOVE 0 TO LEN
             SET TPSUCCESS TO TRUE
             PERFORM A-999-EXIT
    IF TPREPLY
             MOVE "STRING" TO REC-TYPE
             MOVE LENGTH OF PRNAME TO LEN
             SET TPSUCCESS TO TRUE
             PERFORM A-999-EXIT.
*****************************************************
* Exit
*****************************************************
 A-999-EXIT.
     MOVE "Exiting" TO LOGMSG-TEXT.
     PERFORM DO-USERLOG.
     SET TPSUCCESS TO TRUE.
     COPY TPRETURN REPLACING TPSVCRET-REC BY TPSVCRET-REC
              TPTYPE-REC buTPTYPE-REC
              DATA-REC BY PRNAME
              TPSTATUS-REC BY TPSTATUS-REC.
*****************************************************
* Write to userlog
*****************************************************
 DO-USERLOG.
```

```
MOVE TP-STATUS TO LOG-TP-STATUS.
CALL "USERLOG" USING LOGMSG
          LOGMSG-LEN
          TPSTATUS-REC.
```

Terminating a Service Routine

The TPRETURN(3cbl), TPCANCEL(3cbl), and TPFORWAR(3cbl) routines specify that a service routine has completed with one of the following actions:

- TPRETURN sends a reply to the calling client.
- TPCANCEL cancels the current request.
- TPFORWAR forwards a request to another service for further processing.

Sending Replies

The TPRETURN(3cbl) and TPFORWAR(3cbl) calls are COBOL copy files that contain EXIT statements to mark the end of a service routine and send a message to the requester or forward the request to another service, respectively. Use the following signature to call the TPRETURN routine:

```
01 TPSVCRET-REC.
   COPY TPSVCRET.
01 TPTYPE-REC.
   COPY TPTYPE.
01 DATA-REC.
   COPY User Data.
01 TPSTATUS-REC.
   COPY TPSTATUS.
COPY TPRETURN REPLACING TPSVCRET-REC BY TPSVCRET-REC
              TPTYPE-REC BY TPTYPE-REC
              DATA-REC BY DATA-REC
               TPSTATUS-REC BY TPSTATUS-REC.
```

Note: You must use COPY here instead of CALL to ensure that the EXIT statement is called properly, and the COBOL service routine returns control to the BEA Tuxedo system.

The following listing provides the TPSVCRET-REC record signature:

```
05 TPRETURN-VAL    PIC S9(9) COMP-5.
  88 TPSUCCESS     VALUE 0.
  88 TPFAIL        VALUE 1.
```

```
   88 TPFAIL         VALUE 2.
05 APPL-CODE       PIC S9(9) COMP-5.
```

The following table describes the members of a TPSVCRET-REC data structure.

Table 5-2 TPSVCRET-REC Data Structure Members

Member	Description
TP-RETURN-VAL	Indicates whether or not the service has completed successfully on an application-level. The value is an integer that is represented by a symbolic name. Valid settings include: ■ TPSUCCESS—the calling routine succeeded. The routine stores the reply message in the caller's record. If there is a reply message, it is in the caller's record. ■ TPFAIL (default)—the service terminated unsuccessfully. The routine reports an error message to the client process waiting for the reply. In this case, the client's TPCALL or TPGETRPLY routine call fails and the system sets the TP-STATUS variable to TPESVCFAIL to indicate an application-defined failure. If a reply message was expected, it is available in the caller's record. ■ TPEXIT—the service terminated unsuccessfully. The routine reports an error message to the client process waiting for the reply, and exits. For a description of the effect that the value of this argument has on global transactions, refer to "Writing Global Transactions" on page 9-1.
APPLC-CODE	Returns an application-defined return code to the caller. The client can access the value returned in APPLC-CODE by querying APPL-RETURN-CODE IN TPSTATUS-REC. The routine returns this code regardless of success or failure.

Refer to "Defining a Service" on page 5-10 for a description of the TPTYPE-REC record.

The primary function of a service routine is to process a request and return a reply to a client process. It is not necessary, however, for a single service to do all the work required to perform the requested function. A service can act as a requester and pass a request call to another service the same way a client issues the original request: through calls to TPCALL or TPACALL.

Note: The TPCALL and TPACALL routines are described in detail in "Writing Request/Response Clients and Servers" on page 6-1.

When TPRETURN is called, control always returns to the controlling program. If a service has sent requests with asynchronous replies, it must receive all expected replies or invalidate them with TPCANCEL before returning control to the controlling program. Otherwise, the outstanding replies are automatically dropped when they are received by the BEA Tuxedo system controlling program, and an error is returned to the caller.

If the client invokes the service with TPCALL, after a successful call to TPRETURN, the reply message is available in the O-DATA-REC record. If TPACALL is used to send the request, and TPRETURN returns successfully, the reply message is available in the DATA-REC record of TPGETRPLY.

If a reply is expected and TPRETURN encounters errors while processing its arguments, it sends a failed message to the calling process. The caller detects the error by checking the value placed in TP-STATUS. In the case of failed messages, the system sets the TP-STATUS to TPESVCERR. This situation takes precedence over the value of APPL-RETURN-CODE IN TPSTATUS-REC. If this type of error occurs, no reply data is returned, and both the contents and length of the caller's output record remain unchanged.

If TPRETURN returns a message in a record of an unknown type or a record that is not allowed by the caller (that is, if the call is made with TPNOCHANGE), the system returns TPEOTYPE in TP-STATUS. In this case, application success or failure cannot be determined, and the contents and length of the output record remain unchanged.

The value returned in APPL-RETURN-CODE IN TPSTATUS-REC is not relevant if the TPRETURN routine is invoked and a timeout occurs for the call waiting for the reply. This situation takes precedence over all others in determining the value that is returned in TP-STATUS. In this case, TP-STATUS is set to TPETIME and the reply data is not sent, leaving the contents and length of the caller's reply record unchanged. There are two types of timeouts in the BEA Tuxedo system: blocking and transaction timeouts (discussed in "Writing Global Transactions" on page 9-1).

The example code in this section shows the TRANSFER service that is part of the XFER server. Basically, the TRANSFER service makes synchronous calls to the WITHDRAWAL and DEPOSIT services. It allocates a separate record for the reply message since it must use the request record for the calls to both the WITHDRAWAL and the DEPOSIT services. If the call to WITHDRAWAL fails, the service writes the message cannot withdraw on

the status line of the form and sets TP-RETURN-VAL IN TPSVCRET-REC of the TPRETURN routine to TPFAIL. If the call succeeds, the debit balance is retrieved from the reply record.

Note: In the following example, the application moves the identifier for the "destination account" (which is retrieved from the cr_id variable) to the zeroth occurrence of the ACCOUNT_ID field in the transf fielded record. This move is necessary because this occurrence of the field in an FML record is used for data-dependent routing. Refer to *Setting Up a BEA Tuxedo Application* for more information.

A similar scenario is followed for the call to DEPOSIT. On success, the service sets the TP-RETURN-VAL IN TPSVCRET-REC to TPSUCCESS, returning the pertinent account information to the status line.

Listing 5-6 TPRETURN Routine

```
   IDENTIFICATION DIVISION.
   PROGRAM-ID. TRANSFER.
   AUTHOR. TUXEDO DEVELOPMENT.
   ENVIRONMENT DIVISION.
   CONFIGURATION SECTION.
   SOURCE-COMPUTER. USL-486.
   OBJECT-COMPUTER. USL-486.
*
   INPUT-OUTPUT SECTION.
   . . .
*****************************************************
* Tuxedo definitions
*****************************************************
   01  TPSVCRET-REC.
   COPY TPSVCRET.
*
   01  TPTYPE-REC.
   COPY TPTYPE.
*
   01 TPSTATUS-REC.
   COPY TPSTATUS.
*
   01  TPSVCDEF-REC.
   COPY TPSVCDEF.
*****************************************************
* User defined data records
*****************************************************
```

```
   01 TRANS-REC.
      COPY TRANS-AMOUNT.
*
    LINKAGE SECTION.
*
    PROCEDURE DIVISION.
*
   START-TRANSFER.
*****************************************************
* Get the data that was sent by the client
*****************************************************
    MOVE LENGTH OF TRANS-REC TO LEN.
    CALL "TPSVCSTART" USING TPSVCDEF-REC
                      TPTYPE-REC
                      TRANS-REC
                      TPSTATUS-REC.
    IF NOT TPOK
             MOVE "Transaction Encountered An Error" TO STATUS-LINE
             SET TPFAIL TO TRUE.
             COPY TPRETURN REPLACING TPSVCRET-REC BY TPSVCRET-REC
                      TPTYPE-REC BY TPTYPE-REC
                      DATA-REC BY TRANS-REC
                      TPSTATUS-REC BY TPSTATUS-REC.
    ELSE
             . . . Check other parameters
*****************************************************
* must have a valid debit and credit account number
*****************************************************
    CALL "FIND-ACCOUNT-FUNCTION" USING TRANS-DEBIT-ACCOUNT IN  TRANS-REC.

    IF TRANS-DEBIT-ACCOUNT is not valid
             MOVE "Invalid Debit Account Number"
                      TO STATUS-LINE IN TRANS-REC
             SET TPFAIL TO TRUE
             COPY TPRETURN REPLACING
                      DATA-REC BY TRANS-REC.

    CALL "FIND-ACCOUNT-FUNCTION" USING TRANS-CREDIT-ACCOUNT IN TRANS-REC.

    IF TRANS-CREDIT-ACCOUNT is not valid
             MOVE "Invalid Credit Account Number"
             TO STATUS-LINE IN TRANS-REC
             SET TPFAIL TO TRUE
             COPY TPRETURN REPLACING
                       DATA-REC BY TRANS-REC.
*****************************************************
*  Check amount to transfer
*****************************************************
    IF TRANS-AMOUNT IN TRANS-REC < 0
```

```
            MOVE "Invalid Transfer Amount Requested"
                      TO STATUS-LINE IN TRANS-REC
            SET TPFAIL TO TRUE
            COPY TPRETURN REPLACING
                      DATA-REC BY TRANS-REC.
*****************************************************
*  Make Withdrawal using another service
*****************************************************
    MOVE "WITHDRAWAL" TO SERVICE-NAME.
    . . . set other TPCALL parameters
    CALL "TPCALL" USING . . .
    IF NOT TPOK
            MOVE "Cannot withdraw from debit account"
                      TO STATUS-LINE IN TRANS-REC
            SET TPFAIL TO TRUE
            COPY TPRETURN REPLACING
                      DATA-REC BY TRANS-REC.
*****************************************************
*  Make Deposit using another service
*****************************************************
    MOVE "DEPOSIT" TO SERVICE-NAME.
    . . . set other TPCALL parameters
    CALL "TPCALL" USING . . .
    IF NOT TPOK
            MOVE "Cannot Deposit into credit account"
                      TO STATUS-LINE IN TRANS-REC
            SET TPFAIL TO TRUE
            COPY TPRETURN REPLACING
                      DATA-REC BY TRANS-REC.
    . . .
    MOVE "Transfer completed" TO STATUS-LINE IN TRANS-REC
    . . . MOVE all the data into TRANS-REC needed by the client
    SET TPSUCCESS TO TRUE
    COPY TPRETURN REPLACING
                          DATA-REC BY TRANS-REC.
```

Invalidating Descriptors

If a service calling `TPGETRPLY` (described in detail in "Writing Request/Response Clients and Servers" on page 6-1) fails with `TPETIME` and decides to cancel the request, it can invalidate the descriptor with a call to `TPCANCEL(3cbl)`. If a reply subsequently arrives, it is silently discarded.

TPCANCEL cannot be used for transaction replies (that is, for replies to requests made without the TPNOTRAN flag set). Within a transaction, TPABORT(3cbl) does the same job of invalidating the transaction call descriptor.

The following example shows how to invalidate a reply after timing out.

Listing 5-7 Invalidating a Reply After Timing Out

```
. . .  Set up parameters to TPACALL
SET TPNOTRAN TO TRUE.
CALL "TPACALL" USING TPSVCDEF-REC
                     TPTYPE-REC
                     DEBIT-REC
                     TPSTATUS-REC.
IF  NOT TPOK
       error processing
. . .
CALL "TPGETRPLY" USING TPSVCDEF-REC
                     TPTYPE-REC
                     DEBIT-REC
                     TPSTATUS-REC.
IF  NOT TPOK
       error processing
IF TPETIME
        CALL "TPCANCEL" TPSVCDEF-REC
                     TPSTATUS-REC.
    . . .
    SET TPSUCCESS TO TRUE.
    COPY TPRETURN REPLACING TPSVCRET-REC BY TPSVCRET-REC
                     TPTYPE-REC BY TPTYPE-REC
                     DATA-REC BY DEBIT-REC
                     TPSTATUS-REC BY TPSTATUS-REC.
```

Forwarding Requests

The TPFORWAR(3cbl) routine allows a service to forward a request to another service for further processing.

Use the following signature to call the TPFORWAR routine:

```
01 TPSVCDEF-REC.
   COPY TPSVCDEF.
```

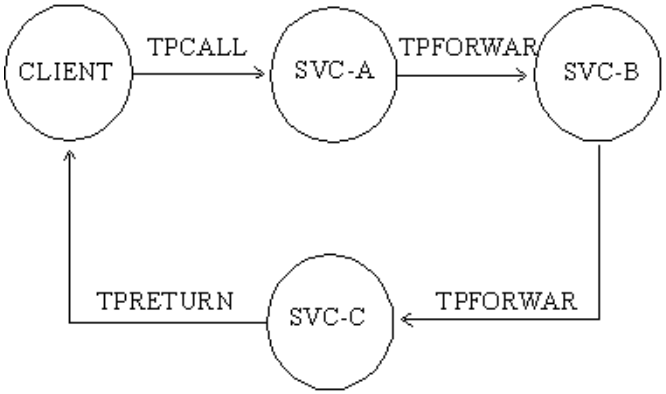
```
01 TPTYPE-REC.
   COPY TPTYPE.
01 DATA-REC.
   COPY User Data.
01 TPSTATUS-REC.
   COPY TPSTATUS.
COPY TPFORWAR REPLACING TPSVCDEF-REC BY TPSVCDEF-REC
              TPTYPE-REC BY TPTYPE-REC
              DATA-REC BY DATA-REC
              TPSTATUS-REC BY TPSTATUS-REC.
```

For descriptions of the TPSVCDEF-REC and TPTYPE-REC records, refer to "Defining a Service" on page 5-10.

The functionality of TPFORWAR differs from a service call: a service that forwards a request does not expect a reply. The responsibility for providing the reply is passed to the service to which the request has been forwarded. The latter service sends the reply to the process that originated the request. It becomes the responsibility of the last server in the forward chain to send the reply to the originating client by invoking TPRETURN.

The following figure shows one possible sequence of events when a request is forwarded from one service to another. Here a client initiates a request using the TPCALL routine and the last service in the chain (SVC_C) provides a reply using the TPRETURN routine.

Figure 5-1 Forwarding a Request



Service routines can forward requests at specified priorities in the same manner that client processes send requests, by using the TPSPRIO routine.

When a process calls TPFORWAR, the system that supplied the controlling program regains control, and the server process is free to do more work.

Note: If a server process is acting as a client and a reply is expected, the server is not allowed to request services from itself. If the only available instance of the desired service is offered by the server process making the request, the call fails, indicating that a recursive call cannot be made. However, if a service routine sends a request (to itself) with the TPNOREPLY communication flag set, or if it forwards the request, the call does not fail because the service is not waiting for itself.

Calling TPFORWAR can be used to indicate success up to that point in processing the request. If no application errors have been detected, you can invoke TPFORWAR, otherwise, you can call TPRETURN with TP-RETURN-VAL IN TPSVCRET-REC set to TPFAIL.

The following example illustrates how the service sends its data record to the DEPOSIT service by calling TPFORWAR. If the new account is added successfully, the branch record is updated to reflect the new account, and the data record is forwarded to the DEPOSIT service. On failure, TPRETURN is called with TP-RETURN-VAL IN TPSVCRET-REC set to TPFAIL and the failure is reported on the status line of the form.

Listing 5-8 How to Use TPFORWAR

```
    . . .
*****************************************************
* Get the data that was sent by the client
*****************************************************
    MOVE LENGTH OF TRANS-REC TO LEN.
    CALL "TPSVCSTART" USING TPSVCDEF-REC
                      TPTYPE-REC
                      TRANS-REC
                      TPSTATUS-REC.
    IF NOT TPOK
             MOVE "Transaction Encountered An Error" TO STATUS-LINE
             SET TPFAIL TO TRUE.
             COPY TPRETURN REPLACING
                      DATA-REC BY TRANS-REC.
    ELSE
             . . . Check other parameters
*****************************************************
* Insert new account record
*****************************************************
```

```
    CALL "ADD-NEW-ACCOUNT-FUNCTION" USING TRANS-ACCOUNT IN TRANS-REC.
    IF Adding New Account Failed
             MOVE "Account not added" TO STATUS-LINE IN TRANS-REC
             SET TPFAIL TO TRUE
             COPY TPRETURN REPLACING
                       DATA-REC BY TRANS-REC.
*****************************************************
* Forward record to the DEPOSIT service to add initial
* balance into account
*****************************************************
    MOVE "DEPOSIT" TO SERVICE-NAME.
    . . . set other TPFORWAR parameters
    COPY TPFORWAR REPLACING
                       DATA-REC BY TRANS-REC.
```

Advertising and Unadvertising Services

When a server is booted, it advertises the services it offers based on the values specified for the CLOPT parameter in the configuration file.

Note: The services that a server may advertise are initially defined when the buildserver command is executed. The -s option allows a comma-separated list of services to be specified. It also allows you to specify a routine with a name that differs from that of the advertised service that is to be called to process the service request. Refer to the buildserver(1) in the *BEA Tuxedo Command Reference* for more information.

The default specification calls for the server to advertise all services with which it was built. Refer to the UBBCONFIG(5) or servopts(5) reference page in the *File Formats, Data Descriptions, MIBs, and System Processes Reference* for more information.

Because an advertised service uses a service table entry in the bulletin board, and can therefore be resource-expensive, an application may boot its servers in such a way that only a subset of the services offered are available. To limit the services available in an application, define the CLOPT parameter, within the appropriate entry in the SERVERS section of the configuration file, to include the desired services in a comma-separated list following the -s option. The -s option also allows you to specify a routine with a

name other than that of the advertised service to be called to process the request. Refer to the servopts(5) reference page in the *File Formats, Data Descriptions, MIBs, and System Processes Reference* for more information.

A BEA Tuxedo application administrator can use the advertise and unadvertise commands of tmadmin(1) to control the services offered by servers. The TPADVERTISE and TPUNADVERTISE routines enable you to dynamically control the advertisement of a service in a request/response or conversational server. The service to be advertised (or unadvertised) must be available within the same server as the service making the request.

Advertising Services

Use the following signature to call the TPADVERTISE(3cbl) routine:

```
01 SERVICE-NAME           PIC X(15).
01 PROGRAM-NAME           PIC X(32).
01 TPSTATUS-REC.
    COPY TPSTATUS.
CALL "TPADVERTISE" USING SERVICE-NAME PROGRAM-NAME TPSTATUS-REC.
```

The following table describes the members of a TPADVERTISE data structure.

Table 5-3 TPADVERTISE Data Structure Members

Member	Description
SERVICE-NAME	Name of the service to be advertised. The service name must be a character string of up to 15 characters. Names longer than 15 characters are truncated. The SPACES string is not a valid value. If it is specified, an error (TPEINVAL) results.
PROGRAM-NAME	BEA Tuxedo system routine that is called to perform a service. Frequently, this name is the same as the name of the service. The SPACES string is not a valid value. If it is specified, an error results.

Unadvertising Services

The TPUNADVERTISE(3cbl) routine removes the name of a service from the service table of the bulletin board so that the service is no longer advertised.

Use the following signature for the TPUNADVERTISE routine:

```
01 SERVICE-NAME           PIC X(15).
01 TPSTATUS-REC.
   COPY TPSTATUS.
CALL "TPUNADVERTISE" USING SERVICE-NAME TPSTATUS-REC.
```

The TPUNADVERTISE data structure contains one member, which is described in the following table.

Table 5-4 TPUNADVERTISE Data Structure Member

Member	Description
SERVICE-NAME	Name of the service to be advertised. The service name must be a character string of up to 15 characters. Names longer than 15 characters are truncated. The SPACES string is not a valid value. If it is specified, an error (TPEINVAL) results.

Example: Dynamic Advertising and Unadvertising of a Service

The following example shows how to use the TPADVERTISE routine. In this example, a server called TLR is programmed to offer only the service called TLRINIT when booted. After some initialization, TLRINIT advertises two services called DEPOSIT and WITHDRAW. Both are performed by the TLRFUNCS routine, and both are built into the TLR server.

After advertising DEPOSIT and WITHDRAW, TLRINIT unadvertises itself.

Listing 5-9 Dynamic Advertising and Unadvertising

```
    . . .
************************************************
* Advertise DEPOSIT service to be processed by
* routine TLRFUNCS
************************************************
    MOVE "DEPOSIT" TO SERVICE-NAME.
    MOVE "TLRFUNCS" TO PROGRAM-NAME.
    CALL "TPADVERTISE" USING SERVICE-NAME
                       PROGRAM-REC
                       TPSTATUS-REC.
    IF NOT TPOK
            error processing
************************************************
* Advertise WITHDRAW service to be processed by
* the same routine TLRFUNCS
************************************************
    MOVE "WITHDRAW" TO SERVICE-NAME.
    MOVE "TLRFUNCS" TO PROGRAM-NAME.
    CALL "TPADVERTISE" USING SERVICE-NAME
                       PROGRAM-REC
                       TPSTATUS-REC.
    IF NOT TPOK
            error processing
************************************************
* Unadvertise TLRINIT service  (yourself)
************************************************
    MOVE "TLRINIT" TO SERVICE-NAME.
    CALL "TPUNADVERTISE" USING SERVICE-NAME
                         TPSTATUS-REC.
    IF NOT TPOK
             error processing
```

Building Servers

To build an executable ATMI server, compile your application service subroutines with the BEA Tuxedo system server adaptor and all other referenced files using the `buildserver(1)` command with the `-C` option.

Note: The BEA Tuxedo server adaptor accepts messages, dispatches work, and manages transactions (if transactions are enabled).

Use the following syntax for the `buildserver` command:

```
buildserver -C -o filename -f filenames -l filenames -s -v
```

The following table describes the `buildserver` command-line options:

Table 5-5 buildserver Command-line Options

This Option . . .	Allows You to Specify the . . .
`-o filename`	Name of the executable output file. The default is `SERVER`.
`-f filenames`	List of files that are link edited before the BEA Tuxedo system libraries. You can specify the `-f` option more than once, and multiple filenames for each occurrence of `-f`. If you specify a COBOL program file (`file.cbl`), it is compiled before it is linked. You can specify other object files (`file.o`) separately, or in groups in an archive file (`file.a`).
`-l filenames`	List of files that are link edited after the BEA Tuxedo system libraries. You can specify the `-l` option more than once, and multiple filenames for each occurrence of `-l`. If you specify a COBOL program file (`file.cbl`), it is compiled before it is linked. You can specify other object files (`file.o`) separately, or in groups in an archive file (`file.a`).
-r *filenames*	List of resource manager access libraries that are link edited with the executable server. The application administrator is responsible for predefining all valid resource manager information in the `$TUXDIR/udataobj/RM` file using the `buildtms(1)` command. You can specify only one resource manager. Refer to *Setting Up a BEA Tuxedo Application* for more information.
-s [*service*:]*routine*	Name of service or services offered by the server and the name of the routine that performs each service. You can specify the `-s` option more than once, and multiple services for each occurrence of `-s`. The server uses the specified service names to advertise its services to clients. Typically, you should assign the same name to both the service and the routine that performs that service. Alternatively, you can specify any names. To assign names, use the following syntax: *service*:*routine*.

Note: The BEA Tuxedo libraries are linked in automatically. You do not need to specify the BEA Tuxedo library names on the command line.

The order in which you specify the library files to be link edited is significant: it depends on the order in which routines are called and which libraries contain references to those functions.

By default, the `buildserver` command invokes the UNIX `cobcc` command. You can specify an alternative compile command and set your own flags for the compile and link-edit phases, however, by setting the `ALTCC` and `ALTCFLAGS` environment variables, respectively. For more information, refer to "Setting Environment Variables" on page 2-6.

Note: On a Windows 2000 system, the `ALTCC` and `ALTCFLAGS` environment variables are not applicable and setting them will produce unexpected results. You must compile your application first using a COBOL compiler and then pass the resulting object file to the `buildserver` command.

The following command processes the `acct.o` application file and creates a server called `ACCT` that contains two services: `NEW_ACCT`, which calls the `OPEN_ACCT` routine, and `CLOSE_ACCT`, which calls a routine of the same name.

```
buildserver -C -o ACCT -f acct.o -s NEW_ACCT:OPEN_ACCT -s CLOSE_ACCT
```

See Also

- "Building Clients" on page 4-10
- `buildclient(1)` in the *BEA Tuxedo Command Reference*

6 Writing Request/Response Clients and Servers

This topic includes the following sections:

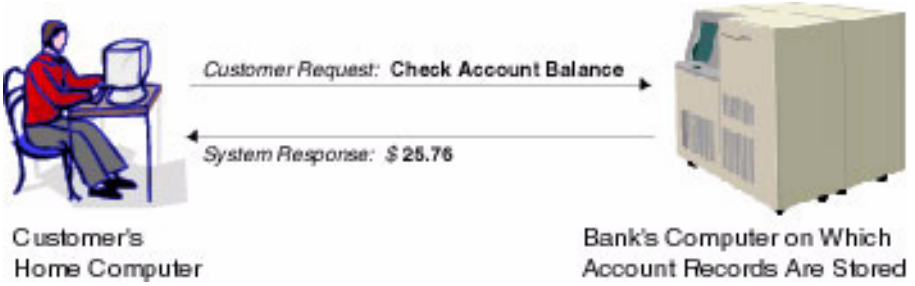
- Overview of Request/Response Communication
- Sending Synchronous Messages
- Sending Asynchronous Messages
- Setting and Getting Message Priorities

Overview of Request/Response Communication

In request/response communication mode, one software module sends a request to a second software module and waits for a response. Because the first software module performs the role of the client, and the second, the role of the server, this mode is also referred to as client/server interaction. Many online banking tasks are programmed in request/response mode. For example, a request for an account balance is executed as follows:

1. A customer (the client) sends a request for an account balance to the Account Record Storage System (the server).
2. The Account Record Storage System (the server) sends a reply to the customer (the client), specifying the dollar amount in the designated account.

Figure 6-1 Example of Request/Response Communication in Online Banking



Once a client process has joined an application, it can then send the request message to a service subroutine for processing and receive a reply message.

Sending Synchronous Messages

The `TPCALL(3cbl)` call sends a request to a service subroutine and synchronously waits for a reply. Use the following signature to call the `TPCALL` routine:

```
01 TPSVCDEF-REC.
   COPY TPSVCDEF.
01 ITPTYPE-REC.
   COPY TPTYPE.
01 IDATA-REC.
   COPY User Data.
01 OTPYTPE-REC.
   COPY TPTYPE.
01 ODATA-REC.
   COPY User Data.
01 TPSTATUS-REC.
   COPY TPSTATUS.
CALL "TPCALL" USING TPSVCDEF-REC
                    ITPTYPE-REC
                    IDATA-REC
```

```
OTPTYPE-REC
ODATA-REC
 TPSTATUS-REC.
```

For more information on the TPSVCDEF data structure, refer to *Programming BEA Tuxedo ATMI Applications Using C*. The IDATA-REC and ITPTYPE-REC structures define the request record. The ODATA-REC and OTPTYPE-REC structures define the reply record. The ITPTYPE-REC and OTPTYPE-REC data structures are similar to the TPTYPE-REC data structure.

TPCALL waits for the expected reply.

Note: Calling the TPCALL routine is logically the same as calling the TPACALL routine, immediately followed by TPGETRPLY, as described in "Sending Asynchronous Messages" on page 6-10.

The request carries the priority set by the system for the specified service (SERVICE-NAME) unless a different priority has been explicitly set by a call to the TPSPRIO routine (described in "Setting and Getting Message Priorities" on page 6-14).

TPCALL returns an integer. On failure, the value of TP-STATUS is set to a value that reflects the type of error that occurred. For information on valid error codes, refer to TPCALL(3cbl) in the *BEA Tuxedo ATMI COBOL Function Reference*.

Note: Communication calls may fail for a variety of reasons, many of which can be corrected at the application level. Possible causes of failure include: application defined errors (TPESVCFAIL), errors in processing return arguments (TPESVCERR), typed record errors (TPEITYPE, TPEOTYPE), timeout errors (TPETIME), and protocol errors (TPEPROTO), among others. For a detailed discussion of errors, refer to "Managing Errors" on page 11-1. For a complete list of possible errors, refer to TPCALL(3cbl) in the *BEA Tuxedo ATMI COBOL Function Reference*.

The BEA Tuxedo system automatically adjusts a record used for receiving a message if the received message is too large for the allocated record. You should test for whether or not the reply records have been resized.

To access the new size of the record, use the address returned in *LEN IN OTPTYPE-REC. To determine whether a reply record has changed in size, compare the size of the reply record before the call to TPCALL with the value of LEN IN OTPTYPE-REC after its return. If LEN IN OTPTYPE-REC is larger than the original size, the record has grown. If not, the record size has not changed.

Example: Using the Same Record for Request and Reply Messages

The following example shows how the client program makes a synchronous call using the same record for both the request and reply messages. In this case, using the same record is appropriate because the `AUDV-REC` message record has been set up to accommodate both request and reply information. The following actions are taken in this code:

1. The service queries the `B_ID` field, but does not overwrite it.
2. The application initializes the `BALANCE` field to zero in preparation for the values to be returned by the service.
3. The `SERVICE-NAME` represents the service name requested. In this example, these variables represent `account` and `teller`, respectively.

Listing 6-1 Using the Same Record for Request and Reply Messages

```
WORKING-STORAGE SECTION.
*****************************************************
* Tuxedo definitions
*****************************************************
  01  TPTYPE-REC.
  COPY TPTYPE.
*
  01 TPSTATUS-REC.
  COPY TPSTATUS.
*
  01  TPSVCDEF-REC.
  COPY TPSVCDEF.
*****************************************************
* Log messages definitions
*****************************************************
  01  LOGMSG.
     05  FILLER           PIC X(6) VALUE  "FIG =>".
     05  LOGMSG-TEXT      PIC X(50).
  01  LOGMSG-LEN          PIC S9(9)  COMP-5.
*
  01  USER-DATA-REC       PIC X(75).
*****************************************************
* This VIEW record (audv) will be sent to the server
*****************************************************
```

```
  01 AUDV-REC.
  COPY AUDV.
*
*****************************************************
  PROCEDURE DIVISION.
  START-FIG.
  MOVE LENGTH OF LOGMSG TO LOGMSG-LEN.
*****************************************************
*  Prepare the audv record
*****************************************************
  MOVE "BRANCH" TO B-ID IN AUDV-REC.
  MOVE 0 TO BALANCE IN AUDV-REC.
  MOVE LENGTH OF AUDV-REC TO LEN.
  MOVE "VIEW" TO REC-TYPE.
  MOVE "audv" TO SUB-TYPE.
  MOVE "SOMESERVICE" TO SERVICE-NAME.
  SET TPBLOCK TO TRUE.
  SET TPNOTRAN TO TRUE.
  SET TPNOTIME TO TRUE.
  SET TPSIGRSTRT TO TRUE.
  SET TPNOCHANGE TO TRUE.
  CALL "TPCALL" USING TPSVCDEF-REC
                TPTYPE-REC
                AUDV-REC
                TPTYPE-REC
                AUDV-REC
                TPSTATUS-REC.
  IF NOT TPOK
         MOVE "Service Failed" TO LOGMSG-TEXT
         PERFORM DO-USERLOG
         PERFORM EXIT-PROGRAM.
  DISPLAY BRANCH and BALANCE
  . . .
```

If the reply is larger than ODATA-REC, then ODATA-REC contains as much of the message as fits in the record. The remainder is discarded and TPCALL sets TP-STATUS IN TPSTATUS-REC to TPTRUNCATE.

Example: Sending a Synchronous Message with TPSIGRSTRT Set

The following example is based on the TRANSFER service, which is part of the XFER server process of bankapp. (bankapp is a sample ATMI application delivered with the BEA Tuxedo system.) The example is based on a service that assumes the role of a client when it calls the WITHDRAWAL and DEPOSIT services. The application sets the communication flag to TPSIGRSTRT in these service calls to give the transaction a better chance of committing. The TPSIGRSTRT flag specifies the action to take if there is a signal interrupt. For more information on communication flags, refer to TPCALL(3cbl) in the *BEA Tuxedo ATMI COBOL Function Reference*.

Listing 6-2 Sending a Synchronous Message with TPSIGRSTRT Set

```
   WORKING-STORAGE SECTION.
*****************************************************
* Tuxedo definitions
*****************************************************
   01 TPTYPE-REC.
   COPY TPTYPE.
*
   01 TPSTATUS-REC.
   COPY TPSTATUS.
 *
   01 TPSVCDEF-REC.
   COPY TPSVCDEF.
*****************************************************
* This VIEW record (audv) will be sent to the server
*****************************************************
   01 AUDV-REC.
   COPY AUDV.
*
*****************************************************
   PROCEDURE DIVISION.
 START-FIG.
*****************************************************
* Prepare the audv record for withdrawal
*****************************************************
  . . .
  MOVE "WITHDRAWAL" TO SERVICE-NAME.
  SET TPSIGRSTRT TO TRUE.
  PERFORM DO-TPCALL.
```

```
  IF NOT TPOK
         MOVE "Cannot withdraw from debit account" TO LOGMSG-TEXT
         PERFORM DO-USERLOG
         PERFORM EXIT-PROGRAM.
  MOVE "DEPOSIT" TO SERVICE-NAME.
  SET TPSIGRSTRT TO TRUE.
  PERFORM DO-TPCALL.
  IF NOT TPOK
         MOVE "Cannot deposit into credit account" TO LOGMSG-TEXT
         PERFORM DO-USERLOG
         PERFORM EXIT-PROGRAM.
  . . .
*****************************************************
*  Perform a TPCALL
*****************************************************
 DO-TPCALL.
   MOVE LENGTH OF AUDV-REC TO LEN.
   MOVE "VIEW" TO REC-TYPE.
   MOVE "audv" TO SUB-TYPE.
   SET TPBLOCK TO TRUE.
   SET TPNOTRAN TO TRUE.
   SET TPNOTIME TO TRUE.
   SET TPNOCHANGE TO TRUE.
   CALL "TPCALL" USING TPSVCDEF-REC
                 TPTYPE-REC
                 AUDV-REC
                 TPTYPE-REC
                 AUDV-REC
                 TPSTATUS-REC.
   . . .
```

Example: Sending a Synchronous Message with TPNOTRAN Set

The following example illustrates a communication call that suppresses transaction mode. The call is made to a service that is not affiliated with a resource manager; it would be an error to allow the service to participate in the transaction. The application prints an accounts receivable report, ACCRV, generated from information obtained from a database named ACCOUNTS.

The service routine REPORT interprets the specified parameters and sends the byte stream for the completed report as a reply. The client uses TPCALL to send the byte stream to a service called PRINTER, which, in turn, sends the byte stream to a printer that is conveniently close to the client. The reply is printed. Finally, the PRINTER service notifies the client that the hard copy is ready to be picked up.

Note: The example "Sending an Asynchronous Message with TPNOTRAN or TPNOREPLY" on page 6-11 shows a similar example using an asynchronous message call.

Listing 6-3 Sending a Synchronous Message with TPNOTRAN Set

```
    WORKING-STORAGE SECTION.
*****************************************************
* Tuxedo definitions
*****************************************************
    01  ITPTYPE-REC.
    COPY TPTYPE.
    01  OTPTYPE-REC.
    COPY TPTYPE.
*
    01 TPSTATUS-REC.
    COPY TPSTATUS.
*
    01  TPSVCDEF-REC.
    COPY TPSVCDEF.
*****************************************************
    01 REPORT-REQUEST         PIC X(100) VALUE SPACES.
    01 REPORT-OUTPUT          PIC X(50000) VALUE SPACES.
*****************************************************
    PROCEDURE DIVISION.
  START-FIG.
    . . .
    join application
    start transaction
    . . .
*****************************************************
* Send report request to REPORT service
* Receive results into REPORT-OUTPUT
*****************************************************
    MOVE "REPORT=accrcv DBNAME=accounts" TO REPORT-REQUEST.
    MOVE "STRING" TO REC-TYPE IN ITYPE-REC.
    MOVE 29 TO LEN IN ITYPE-REC.
    MOVE "STRING" TO REC-TYPE IN OITYPE-REC.
    MOVE 50000 TO LEN IN OTYPE-REC.
```

```
    MOVE "REPORT" TO SERVICE-NAME.
    SET TPTRAN TO TRUE.
    SET TPBLOCK TO TRUE.
    SET TPNOTIME TO TRUE.
    SET TPSIGRSTRT TO TRUE.
    SET TPNOCHANGE TO TRUE.
    CALL "TPCALL" USING TPSVCDEF-REC
                  ITPTYPE-REC
                  REPORT-REQUEST
                  OTPTYPE-REC
                  REPORT-OUTPUT
                  TPSTATUS-REC.
    IF NOT TPOK
       error processing
    IF TPETRUNCATE
       The report was truncated
       error processing
*****************************************************
* Send REPORT-OUTPUT to PRINTER service
*****************************************************
    MOVE "PRINTER" TO SERVICE-NAME.
    SET TPNOTRAN TO TRUE.
    MOVE "STRING" TO REC-TYPE IN ITTYPE-REC.
    MOVE LEN IN OTYPE-REC TO LEN IN ITYPE-REC.
    CALL "TPCALL" USING TPSVCDEF-REC
                  ITPTYPE-REC
                  REPORT-OUTPUT
                  OTPTYPE-REC
                  REPORT-OUTPUT
                  TPSTATUS-REC.
    IF NOT TPOK
         error processing
    . . .
         terminate transaction
          leave application
```

Note: In the preceding example, the term `error routine` indicates that the following tasks are performed: an error message is printed, the transaction is aborted, the client leaves the application, and the program is exited.

This example also shows how the `TPNOCHANGE` communication setting is used to enforce strong record type checking by indicating that the reply message must be returned in the same type of record that was originally allocated. The strong type check flag, `TPNOCHANGE`, forces the reply to be returned in a record of type `STRING`.

A possible reason for this check is to guard against errors that may occur in the REPORT service subroutine, resulting in the use of a reply record of an incorrect type. Another reason is to prevent changes that are not made consistently across all areas of dependency. For example, another programmer may have changed the REPORT service to standardize all replies in another STRING format without modifying the client process to reflect the change.

Sending Asynchronous Messages

This section explains how to:

- Send an asynchronous request using the TPACALL routine
- Get an asynchronous reply using the TPGETRPLY routine

The type of asynchronous processing discussed in this section is sometimes referred to as *fan-out parallelism* because it allows a client's requests to be distributed (or "fanned out") simultaneously to several services for processing.

The other type of asynchronous processing supported by the BEA Tuxedo system is pipeline parallelism in which the TPFORWAR routine is used to pass (or forward) a process from one service to another. For a description of the TPFORWAR routine, refer to "Writing Servers" on page 5-1.

Sending an Asynchronous Request

The TPACALL(3cbl) routine sends a request to a service and immediately returns. Use the following signature to call the TPACALL routine:

```
01 TPSVCDEF-REC.
   COPY TPSVCDEF.
01 TPTYPE-REC.
   COPY TPTYPE.
01 DATA-REC.
   COPY User Data.
01 TPSTATUS-REC.
   COPY TPSTATUS.
CALL "TPACALL" USING TPSVCDEF-REC TPTYPE-REC DATA-REC TPSTATUS-REC.
```

For more information on the TPSVCDEF and TPTYPE-REC data structures, refer to "Defining a Service" on page 5-10.

The TPACALL routine sends a request message to the service named in the SERVICE-NAME and immediately returns from the call. Upon successful completion of the call, the TPACALL routine returns an integer that serves as a communication handle used to access the correct reply for the relevant request. While TPACALL is in transaction mode (as described in "Writing Global Transactions" on page 9-1) there may not be any outstanding replies when the transaction commits; that is, within a given transaction, for each request for which a reply is expected, a corresponding reply must eventually be received.

If the value TPNOREPLY is set, the parameter signals to TPACALL that a reply is not expected. When set, on success TPACALL returns a value of 0 as the reply descriptor. If subsequently passed to the TPGETRPLY routine, this value becomes invalid, this value becomes invalid. Guidelines for using this setting correctly when a process is in transaction mode are discussed in "Writing Global Transactions" on page 9-1.

On error, TPACALL sets TP-STATUS to a value that reflects the nature of the error. TPACALL returns many of the same error codes as TPCALL. The differences between the error codes for these functions are based on the fact that one call is synchronous and the other, asynchronous. These errors are discussed at length in "Managing Errors" on page 11-1.

The following example shows how TPACALL uses the TPNOTRAN and TPNOREPLY settings. This code is similar to the code in "Example: Sending a Synchronous Message with TPNOTRAN Set" on page 6-7. In this case, however, a reply is not expected from the PRINTER service. By setting both TPNOTRAN and TPNOREPLY, the client is indicating that no reply is expected and the PRINTER service will not participate in the current transaction. This situation is discussed more fully in "Managing Errors" on page 11-1.

Listing 6-4 Sending an Asynchronous Message with TPNOTRAN or TPNOREPLY

```
   WORKING-STORAGE SECTION.
*****************************************************
* Tuxedo definitions
*****************************************************
   01 ITPTYPE-REC.
   COPY TPTYPE.
   01 OTPTYPE-REC.
```

```
   COPY TPTYPE.
*
   01 TPSTATUS-REC.
   COPY TPSTATUS.
*
   01 TPSVCDEF-REC.
   COPY TPSVCDEF.
******************************************************
   01 REPORT-REQUEST        PIC X(100) VALUE SPACES.
   01 REPORT-OUTPUT         PIC X(50000) VALUE SPACES.
******************************************************
   PROCEDURE DIVISION.
 START-FIG.
   . . .
    join application
    start transaction
    . . .
 ******************************************************
 * Send report request to REPORT service
 * Receive results into REPORT-OUTPUT
 ******************************************************
    MOVE "REPORT=accrcv DBNAME=accounts" TO REPORT-REQUEST.
    MOVE "STRING" TO REC-TYPE IN ITPTYPE-REC.
    MOVE 29 TO LEN IN ITPTYPE-REC.
    MOVE "STRING" TO REC-TYPE IN OITYPE-REC.
    MOVE 50000 TO LEN IN OTPTYPE-REC.
    MOVE "REPORT" TO SERVICE-NAME.
    SET TPTRAN TO TRUE.
    SET TPBLOCK TO TRUE.
    SET TPNOTIME TO TRUE.
    SET TPSIGRSTRT TO TRUE.
    SET TPREPLY TO TRUE.
    SET TPNOCHANGE TO TRUE.
    CALL "TPCALL" USING TPSVCDEF-REC
                  ITPTYPE-REC
                  REPORT-REQUEST
                  OTPTYPE-REC
                  REPORT-OUTPUT
                  TPSTATUS-REC.
    IF NOT TPOK
       error processing
    IF TPETRUNCATE
       The report was truncated
       error processing
******************************************************
* Send REPORT-OUTPUT to PRINTER service
******************************************************
    MOVE "PRINTER" TO SERVICE-NAME.
    SET TPNOTRAN TO TRUE.
```

```
SET TPNOREPLY TO TRUE.
MOVE "STRING" TO REC-TYPE IN ITPTYPE-REC.
MOVE LEN IN OTPTYPE-REC TO LEN IN ITPTYPE-REC.
CALL "TPACALL" USING TPSVCDEF-REC
                ITPTYPE-REC
                REPORT-OUTPUT
                TPSTATUS-REC.
IF NOT TPOK
   error processing
. . .
commit transaction
leave application
```

Getting an Asynchronous Reply

A reply to a service call can be received asynchronously by calling the TPGETRPLY(3cbl) routine. The TPGETRPLY routine dequeues a reply to a request previously sent by TPACALL.

Use the following signature to call the TPGETRPLY routine:

```
01 TPSVCDEF-REC.
   COPY TPSVCDEF.
01 TPTYPE-REC.
   COPY TPTYPE.
01 DATA-REC.
   COPY User Data.
01 TPSTATUS-REC.
   COPY TPSTATUS.
CALL "TPGETRPLY" USING TPSVCDEF-REC TPTYPE-REC DATA-REC TPSTATUS-REC.
```

For more information on the TPSVCDEF and TPTYPE-REC data structures, refer to "Defining a Service" on page 5-10.

By default, the function waits for the arrival of the reply that corresponds to the value referenced by the communication handle. During this waiting interval, a blocking timeout may occur. A time-out occurs when TPGETRPLY fails and TP-STATUS is set to TPETIME (unless TPNOTIME is set).

Setting and Getting Message Priorities

Two ATMI calls allow you to determine and set the priority of a message request: TPSPRIO(3cbl) and TPGPRIO(3cbl). The priority affects how soon the request is dequeued by the server; servers dequeue requests with the highest priorities first.

This section describes:

- Setting a Message Priority
- Getting a Message Priority

Setting a Message Priority

The TPSPRIO(3cbl) routine enables you to set the priority of a message request.

The TPSPRIO routine affects the priority level of only one request: the next request to be sent by TPCALL or TPACALL, or to be forwarded by a service subroutine.

Use the following signature to call the TPSPRIO routine:

```
01 TPPRIDEF-REC.
   COPY TPPRIDEF.
01 TPSTATUS-REC.
   COPY TPSTATUS.
CALL "TPSPRIO" USING TPPRIDEF-REC TPSTATUS-REC.
```

Use the following signature for the TPPRIDEF-REC data structure.

```
05 PRIORITY     PIC S9(9) COMP-5.
05 PRIO-FLAG    PIC S9(9) COMP-5.
  88 TPABSOLUTE     VALUE 0.
  88 TPRELATIVE     VALUE 1.
```

The following table describes the arguments to the TPSPRIO routine.

Table 6-1 TPSPRIO Routine Fields

Field	Description
PRIORITY	Integer indicating a new priority value. The effect of this argument is controlled by PRIO-FLAG. If PRIO-FLAG is set to 0, PRIORITY specifies a relative value and the sign accompanying the value indicates whether the current priority is incremented or decremented. Otherwise, the value specified indicates an absolute value and PRIORITY must be set to a value between 0 and 100. If you do not specify a value within this range, the system sets the value to 50.
PRIO-FLAG	Indicates whether the value of PRIORITY is treated as a relative value (0, the default) or an absolute value (TPABSOLUTE).

The following sample code is an excerpt from the TRANSFER service. In this example, the TRANSFER service acts as a client by sending a synchronous request, via TPCALL, to the WITHDRAWAL service. TRANSFER also invokes TPSPRIO to increase the priority of its request message to WITHDRAWAL, and to prevent the request from being queued for the WITHDRAWAL service (and later the DEPOSIT service) after waiting on the TRANSFER queue.

Listing 6-5 Setting the Priority of a Request Message

```
    WORKING-STORAGE SECTION.
*****************************************************
* Tuxedo definitions
*****************************************************
    01 TPTYPE-REC.
       COPY TPTYPE.
*
    01 TPSTATUS-REC.
       COPY TPSTATUS.
*
    01 TPSVCDEF-REC.
       COPY TPSVCDEF.
*
    01 TPPRIDEF-REC.
       COPY TPPRIDEF.
*****************************************************
    01 DATA-REC               PIC X(100) VALUE SPACES.
*****************************************************
    PROCEDURE DIVISION.
```

```
START-FIG.
   . . .
   join application
   . . .
   MOVE 30 TO PRIORITY.
   SET TPRELATIVE TO TRUE.
   CALL "TPSPRIO" USING TPPRIDEF-REC TPSTATUS-REC
   IF NOT TPOK
      error processing
   MOVE "CARRAY" TO REC-TYPE.
   MOVE 100 TO LEN.
   MOVE "WITHDRAWAL" TO SERVICE-NAME.
   SET TPTRAN TO TRUE .
   SET TPBLOCK TO TRUE .
   SET TPNOTIME TO TRUE .
   SET TPSIGRSTRT TO TRUE .
   SET TPREPLY TO TRUE .
   CALL "TPACALL" USING TPSVCDEF-REC
                  TPTYPE-REC
                  DATA-REC
                  TPSTATUS-REC.
   IF NOT TPOK
      error processing
   . . .
   leave application
```

Getting a Message Priority

The TPGPRIO(3cbl) routine enables you to get the priority of a message request.

Use the following signature to call the TPGPRIO routine:

```
01 TPPRIDEF-REC.
   COPY TPPRIDEF.
01 TPSTATUS-REC.
   COPY TPSTATUS.
CALL "TPGPRIO" USING TPPRIDEF-REC TPSTATUS-REC.
```

A requester can call the TPGPRIO routine after invoking the TPCALL or TPACALL routine to retrieve the priority of the request message. If a requester calls the function but no request is sent, the routine fails, setting TP-STATUS to TPENOENT. Upon success, TPGPRIO sets TP-STATUS to TPOK and returns an integer value in the range of 1 to 100 (where the highest priority value is 100).

If a priority has not been explicitly set using the TPSPRIO routine, the system sets the message priority to that of the service routine that handles the request. Within an application, the priority of the request-handling service is assigned a default value of 50 unless a system administrator overrides this value.

The following example shows how to determine the priority of a message that was sent in an asynchronous call.

Listing 6-6 Determining the Priority of the Sent Request

```
   WORKING-STORAGE SECTION.
*****************************************************
* Tuxedo definitions
*****************************************************
   01 TPTYPE-REC-1.
   COPY TPTYPE.
   01 TPTYPE-REC-2.
   COPY TPTYPE.
*
   01 TPSTATUS-REC.
   COPY TPSTATUS.
*
   01 TPSVCDEF-REC-1.
   COPY TPSVCDEF.
   01 TPSVCDEF-REC-2.
   COPY TPSVCDEF.
*
   01 TPPRIDEF-REC-1.
   COPY TPPRIDEF.
   01 TPPRIDEF-REC-2.
   COPY TPPRIDEF.
*****************************************************
   01 DATA-REC-1     PIC X(100) VALUE SPACES.
   01 DATA-REC-2     PIC X(100) VALUE SPACES.
*****************************************************
   PROCEDURE DIVISION.
START-FIG.
   . . .
   join application
   populate DATA-REC1 and DATA-REC2 with send request
   . . .
   MOVE "CARRAY" TO REC-TYPE IN TYPE-REC-1.
   MOVE 100 TO LEN IN TYPE-REC-1.
   MOVE "SERVICE1" TO SERVICE-NAME IN TPSVCDEV-REC-1.
   SET TPTRAN TO TRUE IN TPSVCDEV-REC-1.
   SET TPBLOCK TO TRUE IN TPSVCDEV-REC-1.
```

```
   SET TPNOTIME TO TRUE IN TPSVCDEV-REC-1.
   SET TPSIGRSTRT TO TRUE IN TPSVCDEV-REC-1.
   SET TPREPLY TO TRUE IN TPSVCDEV-REC-1.
   CALL "TPACALL" USING TPSVCDEF-REC-1
                  TPTYPE-REC-1
                  DATA-REC-1
                  TPSTATUS-REC.
   IF NOT TPOK
      error processing
   CALL "TPGPRIO" USING TPPRIDEF-REC-1 TPSTATUS-REC
   IF NOT TPOK
      error processing
   MOVE "CARRAY" TO REC-TYPE IN TYPE-REC-2.
   MOVE 100 TO LEN IN TYPE-REC-2.
   MOVE "SERVICE2" TO SERVICE-NAME IN TPSVCDEV-REC-2.
   SET TPTRAN TO TRUE IN TPSVCDEV-REC-2.
   SET TPBLOCK TO TRUE IN TPSVCDEV-REC-2.
   SET TPNOTIME TO TRUE IN TPSVCDEV-REC-2.
   SET TPSIGRSTRT TO TRUE IN TPSVCDEV-REC-2.
   SET TPREPLY TO TRUE IN TPSVCDEV-REC-2.
   CALL "TPACALL" USING TPSVCDEF-REC-2
                  TPTYPE-REC-2
                  DATA-REC-2
                  TPSTATUS-REC.
   IF NOT TPOK
      error processing
   CALL "TPGPRIO" USING TPPRIDEF-REC-2 TPSTATUS-REC
   IF NOT TPOK
      error processing
   IF PRIORITY IN TPSVCDEF-REC-1 >= PRIORITY IN TPSVCDEF-REC-2
               PERFORM DO-GETREPLY1
               PERFORM DO-GETREPLY2
   ELSE
                    PERFORM DO-GETREPLY2
                    PERFORM DO-GETREPLY1
   END-IF.
   . . .
   leave application
DO-GETRPLY1.
   SET TPGETHANDLE TO TRUE IN TPSVCDEV-REC-1.
   SET TPCHANGE TO TRUE IN TPSVCDEV-REC-1.
   SET TPBLOCK TO TRUE IN TPSVCDEV-REC-1.
   SET TPNOTIME TO TRUE IN TPSVCDEV-REC-1.
   SET TPSIGRSTRT TO TRUE IN TPSVCDEV-REC-1.
   CALL "TPGETRPLY" USING TPSVCDEF-REC-1
                    TPTYPE-REC-1
                    DATA-REC-1
                    TPSTATUS-REC.
   IF NOT TPOK
```

```
           error processing
DO-GETRPLY2
  SET TPGETHANDLE TO TRUE IN TPSVCDEV-REC-2.
  SET TPCHANGE TO TRUE IN TPSVCDEV-REC-2.
  SET TPBLOCK TO TRUE IN TPSVCDEV-REC-2.
  SET TPNOTIME TO TRUE IN TPSVCDEV-REC-2.
  SET TPSIGRSTRT TO TRUE IN TPSVCDEV-REC-2.
  CALL "TPGETRPLY" USING TPSVCDEF-REC-2
                   TPTYPE-REC-2
                   DATA-REC-2
                   TPSTATUS-REC.
  IF NOT TPOK
              error processing
```

CHAPTER

7 Writing Conversational Clients and Servers

This topic includes the following sections:

- Overview of Conversational Communication
- Joining an Application
- Establishing a Connection
- Sending and Receiving Messages
- Ending a Conversation
- Building Conversational Clients and Servers
- Understanding Conversational Communication Events

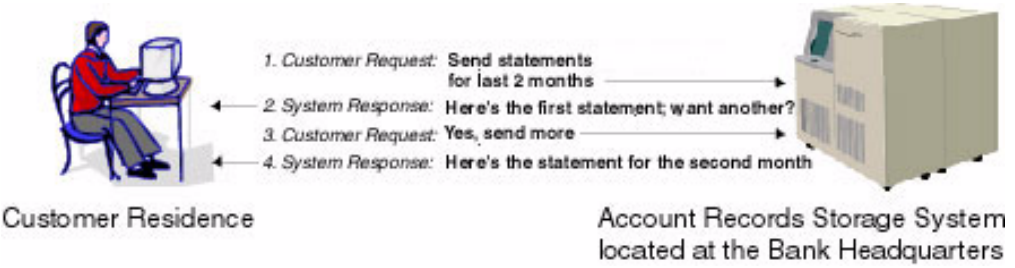
Overview of Conversational Communication

Conversational communication is the BEA Tuxedo system implementation of a human-like paradigm for exchanging messages between ATMI clients and servers. In this form of communication, a virtual connection is maintained between the client (initiator) and server (subordinate) and each side maintains information about the state of the conversation. The connection remains active until an event occurs to terminate it.

During conversational communication, a *half-duplex* connection is established between the client and server. A half-duplex connection allows messages to be sent in only one direction at any given time. Control of the connection can be passed back and forth between the initiator and the subordinate. The process that has control can send messages; the process that does not have control can only receive messages.

To understand how conversational communication works in a BEA Tuxedo ATMI application, consider the following example from an online banking application. In this example, a bank customer requests checking account statements for the past two months.

Figure 7-1 Example of Conversational Communication in an Online Banking Application



1. The customer requests the checking account statements for the past two months.
2. The Account Records Storage System responds by sending the first month's checking account statement followed by a `More` prompt for accessing the remaining month's statement.
3. The customer requests the second month's account statement by selecting the `More` prompt.

Note: The Account Records Storage System must maintain state information so it knows which account statement to return when the customer selects the `More` prompt.

4. The Account Records Storage System sends the remaining month's account statement.

As with request/response communication, the BEA Tuxedo system passes data using typed records. The record types must be recognized by the application. For more information on record types, refer to "Overview of Typed Records" on page 3-1.

Conversational clients and servers have the following characteristics:

- The logical connection between them remains active until terminated.
- Any number of messages can be transmitted across a connection between them.
- Both clients and servers use the TPSEND and TPRECV routines to send and receive data in conversations.

Conversational communication differs from request/response communication in the following ways:

- A conversational client initiates a request for service using TPCONNECT rather than TPCALL or TPACALL.
- A conversational client sends a service request to a conversational server.
- The configuration file reserves part of the conversational server for addressing conversational services.
- Conversational servers are prohibited from making calls using TPFORWAR.

Joining an Application

A conversational client must join an application via a call to TPINITIALIZE before attempting to establish a connection to a service. For more information, refer to "Writing Clients" on page 4-1.

Establishing a Connection

The TPCONNECT(3cbl) routine sets up a conversation:

Use the following signature to call the TPCONNECT routine.

```
01 TPSVCDEF-REC.
     COPY TPSVCDEF.
```

```
01 TPTYPE-REC.
     COPY TPTYPE.

01 DATA-REC.
     COPY User Data.

01 TPSTATUS-REC.
     COPY TPSTATUS.

CALL "TPCONNECT" USING TPSVCDEF-REC TPTYPE-REC DATA-REC TPSTATUS-REC.
```

Refer to "Defining a Service" on page 5-10 for more information on the TPSVCDEF-REC record, and to "Defining Typed Records" on page 3-6 for more information on the TPTYPE-REC record.

At the same time the connection is being established, data can be sent through the DATA-REC with the length of the data specified by LEN IN TPTYPE-REC. The REC-TYPE and SUB-TYPE of the data in DATA-REC must be types recognized by the service being called. If no data is being sent, the value of REC-TYPE is SPACES, and DATA-REC and LEN are ignored.

The BEA Tuxedo system returns a communication handle, COMM-HANDLE IN TPSVCDEF-REC, when a connection is established with TPCONNECT or TPSVCSTART. COMM-HANDLE is used to identify subsequent message transmissions with a particular conversation. A client or conversational service can participate in more than one conversation simultaneously. The maximum number of simultaneous conversations is 64.

In the event of a failure, TPCONNECT sets TP-STATUS to the appropriate error condition. For a list of possible error codes, refer to TPCONNECT(3cbl) in the *BEA Tuxedo ATMI COBOL Function Reference*.

The following example shows how to use the TPCONNECT routine.

Listing 7-1 Establishing a Conversational Connection

```
    . . .
* Prepare the record to send
  MOVE "HELLO" TO DATA-REC.
  MOVE 5 TO LEN.
  MOVE "STRING" TO REC-TYPE.
*
  SET TPBLOCK TO TRUE.
  SET TPNOTRAN TO TRUE.
  SET TPNOTIME TO TRUE.
```

```
  SET TPSIGRSTRT TO TRUE.
  SET TPSENDONLY TO TRUE.
*
  CALL "TPCONNECT" USING TPSVCDEF-REC
                   TPTYPE-REC
                   DATA-REC
                   TPSTATUS-REC.
  IF NOT TPOK
           error processing ...
  ELSE
      COMM-HANDLE is valid.
```

Sending and Receiving Messages

Once the BEA Tuxedo system establishes a conversational connection, communication between the initiator and subordinate is accomplished using send and receive calls. The process with control of the connection can send messages using the `TPSEND(3cbl)` routine; the process without control can receive messages using the `TPRECV(3cbl)` routine.

Note: Initially, the originator (that is, the client) decides which process has control using the `TPSENDONLY` or `TPRECVONLY` flag value of the `TPCONNECT` call. `TPSENDONLY` specifies that control is being retained by the originator; `TPRECVONLY`, that control is being passed to the called service.

Sending Messages

To send a message, use the `TPSEND(3cbl)` routine with the following signature:

```
01 TPSVCDEF-REC.
     COPY TPSVCDEF.

01 TPTYPE-REC.
     COPY TPTYPE.

01 DATA-REC.
     COPY User Data.
```

```
01 TPSTATUS-REC.
     COPY TPSTATUS.

CALL "TPSEND" USING TPSVCDEF-REC TPTYPE-REC USER-DATA-REC TPSTATUS-REC.
```

Refer to "Defining a Service" on page 5-10 for more information on the TPSVCDEF-REC record, and refer to "Defining Typed Records" on page 3-6 for more information on the TPTYPE-REC record.

In the event of a failure, the TPSEND routine sets TP-STATUS to the appropriate error condition. For a list of possible error codes, refer to TPSEND(3cbl) in the *BEA Tuxedo ATMI COBOL Function Reference*.

You are not required to pass control each time you issue the TPSEND routine. In some applications, the process authorized to issue TPSEND calls can execute as many calls as required by the current task before turning over control to the other process. In other applications, however, the logic of the program may require the same process to maintain control of the connection throughout the life of the conversation.

The following example shows how to invoke the TPSEND routine.

Listing 7-2 Sending Data in Conversational Mode

```
   . . .
   SET TPNOBLOCK TO TRUE.
   SET TPNOTIME TO TRUE.
   SET TPSIGRSTRT TO TRUE.
   SET TPRECVONLY TO TRUE.
*
   CALL "TPSEND" USING TPSVCDEF-REC
                       TPTYPE-REC
                       DATA-REC
                       TPSTATUS-REC.
   IF NOT TPOK
       error processing . . .
```

Receiving Messages

To receive data sent over an open connection, use the TPRECV(3cbl) routine with the following signature:

```
01 TPSVCDEF-REC.
     COPY TPSVCDEF.

01 TPTYPE-REC.
     COPY TPTYPE.

01 DATA-REC.
     COPY User Data.

01 TPSTATUS-REC.
     COPY TPSTATUS.

CALL "TPRECV" USING TPSVCDEF-REC TPTYPE-REC DATA-REC TPSTATUS-REC.
```

Refer to "Defining a Service" on page 5-10 for more information on the TPSVCDEF-REC record. Refer to "Defining Typed Records" on page 3-6 for more information on the TPTYPE-REC record.

The following example shows how to use the TPRECV routine.

Listing 7-3 Receiving Data in Conversation

```
  . . .
  SET TPNOCHANGE TO TRUE.
  SET TPBLOCK TO TRUE.
  SET TPNOTIME TO TRUE.
  SET TPSIGRSTRT TO TRUE.
*
  MOVE LENGTH OF DATA-REC TO LEN.
*
  CALL "TPRECV" USING TPSVCDEF-REC
                      TPTYPE-REC
                      DATA-REC
                      TPSTATUS-REC.
  IF NOT TPOK
      error processing . . .
```

Ending a Conversation

A connection can be taken down gracefully and a conversation ended normally through:

- A successful call to TPRETURN in a simple conversation.
- A series of successful calls to TPRETURN in a complex conversation based on a hierarchy of connections.
- Global transactions, as described in "Writing Global Transactions" on page 9-1.

Note: The TPRETURN routine is described in detail in "Writing Request/Response Clients and Servers" on page 6-1.

The following sections describe two scenarios for gracefully terminating conversations that do not include global transactions in which the TPRETURN function is used.

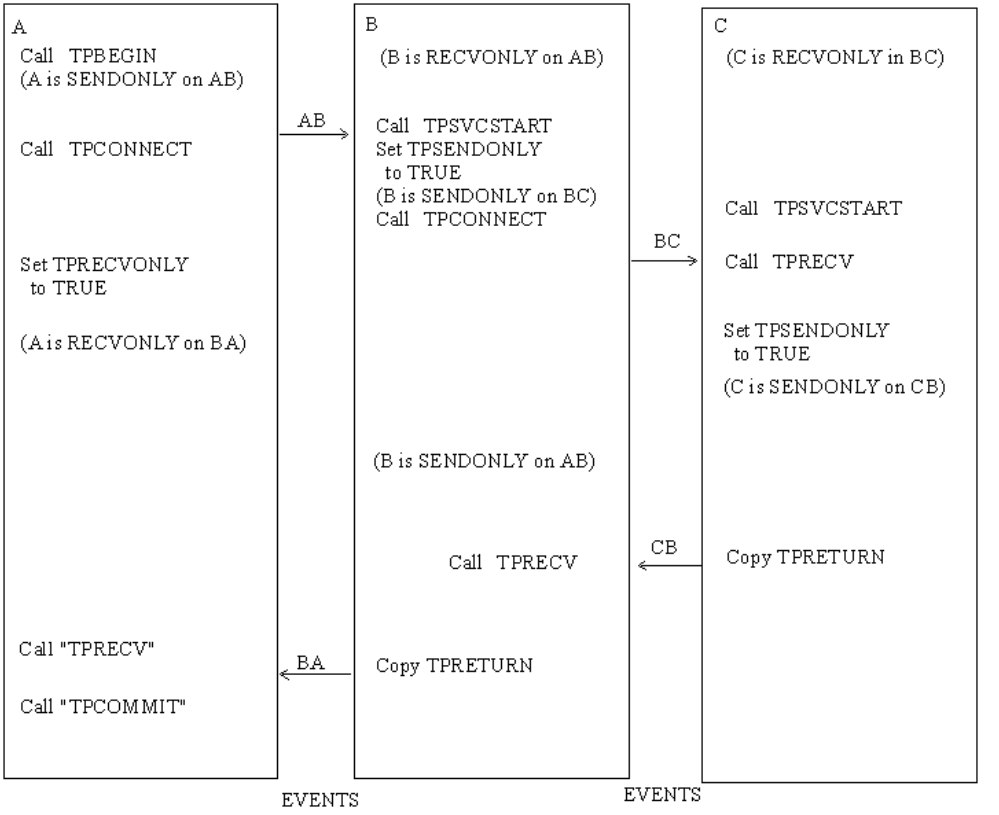
The first example shows how to terminate a simple conversation between two components. The second example illustrates a more complex scenario, with a hierarchical set of conversations.

If you end a conversation with connections still open, the system returns an error. In this case, either TPCOMMIT or TPRETURN fails in a disorderly manner.

Example: Ending a Simple Conversation

The following diagram shows a simple conversation between A and B that terminates gracefully.

Figure 7-2 Simple Conversation Terminating Gracefully



The program flow is as follows:

1. A sets up the connection by calling `TPCONNECT` with `TPSENDONLY` set, indicating that process B is on the receiving end of the conversation.
2. A turns control of the connection over to B by calling `TPSEND` with `TPRECVONLY` set, resulting in the generation of a `TPEV_SENDONLY` event.

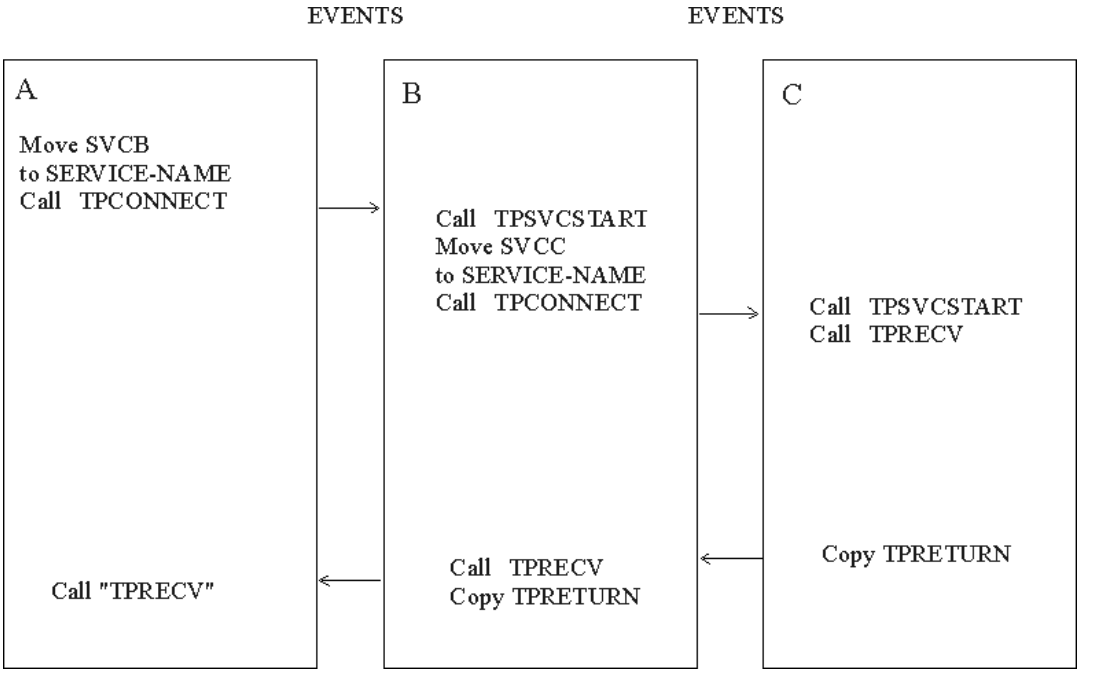
3. The next call by B to TPRECV sets TP-STATUS to TPEEVENT, and returns TPEV_SENDONLY in TPEVENT, indicating that control has passed to B.
4. B calls TPRETURN with TPRETURN-VAL IN TPSVCRET set to TPSUCCESS. This call generates a TPEV_SVCSUCC event for A and gracefully brings down the connection.
5. A calls TPRECV, learns of the event, and recognizes that the conversation has been terminated. Data can be received on this call to TPRECV even if the event is set to TPEV_SVCFAIL.

Note: In this example, A can be either a client or a server, but B must be a server.

Example: Ending a Hierarchical Conversation

The following diagram shows a hierarchical conversation that terminates gracefully.

Figure 7-3 Connection Hierarchy



In the preceding example, service B is a member of a conversation that has initiated a connection to a second service called C. In other words, there are two active connections: A-to-B and B-to-C. If B is in control of both connections, a call to `TPRETURN` has the following effect: the call fails, a `TPEV_SVCERR` event is posted on all open connections, and the connections are closed in a disorderly manner.

In order to terminate both connections normally, an application must execute the following sequence:

1. B calls `TPSEND` with the `TPRECVONLY` flag set on the connection to C, transferring control of the B-to-C connection to C.
2. C calls `TPRETURN` with `TPRETURN-VAL IN TPSVCRET` set to `TPSUCCESS`, `TPFAIL`, or `TPEXIT`, as appropriate.
3. B can then call `TPRETURN`, posting an event (either `TPEV_SVCSUCC` or `TPEV_SVCFAIL`) for A.

Note: It is legal for a conversational service to make request/response calls if it needs to do so to communicate with another service. Therefore, in the preceding example, the calls from B to C may be executed using TPCALL or TPACALL instead of TPCONNECT. Conversational services are not permitted to make calls to TPFORWAR.

Executing a Disorderly Disconnect

The only way in which a disorderly disconnect can be executed is through a call to the TPDISCON(3cbl) routine (which is equivalent to "pulling the plug" on a connection). This routine can be called only by the initiator of a conversation (that is, the client).

Note: This is not the preferred method for bringing down a conversation. To bring down an application gracefully, the subordinate (the server) should call the TPRETURN routine.

Use the following signature to call the TPDISCON routine:

```
01 TPSVCDEF-REC.
     COPY TPSVCDEF.

01 TPSTATUS-REC.
     COPY TPSTATUS.

CALL "TPDISCON" USING TPSVCDEF-REC TPSTATUS-REC.
```

The COMM-HANDLE argument specifies the communication handle returned by the TPCONNECT routine when the connection is established.

The TPDISCON routine generates a TPEV_DISCONIMM event for the service at the other end of the connection, rendering the COMM-HANDLE invalid. If a transaction is in progress, the system aborts it and data may be lost.

If TPDISCON is called from a service that was not the originator of the connection identified by COMM-HANDLE, the routine fails with an error code of TPEBADDESC.

For a list and descriptions of all event and error codes, refer to TPDISCON(3cbl) in the *BEA Tuxedo ATMI COBOL Function Reference*.

Building Conversational Clients and Servers

Use the following commands to build conversational clients and servers:

- `buildclient()` as described in "Building Clients" in *Programming BEA Tuxedo ATMI Applications Using C*
- `buildserver()` as described in "Building Servers" in *Programming BEA Tuxedo ATMI Applications Using C*

For conversational and request/response services, you cannot:

- Build both in the same server
- Assign the same name to both

Understanding Conversational Communication Events

The BEA Tuxedo system recognizes five events in conversational communication. All five events can be posted for `TPRECV`; three can be posted for `TPSEND`.

The following table lists the events, the routines for which they are returned, and a detailed description of each.

Table 7-1 Conversational Communication Events

Event	Received By	Description
`TPEV_SENDONLY`	`TPRECV`	Control of the connection has been passed; this process can now call `TPSEND`.

Table 7-1 Conversational Communication Events

Event	Received By	Description
TPEV_DISCONIMM	TPSEND, TPRECV, TPRETURN	The connection has been torn down and no further communication is possible. The TPDISCON routine posts this event in the originator of the connection, and sends it to all open connections when TPRETURN is called, as long as connections to subordinate services remain open. Connections are closed in a disorderly fashion. If a transaction exists, it is aborted.
TPEV_SVCERR	TPSEND	Received by the originator of the connection, usually indicating that the subordinate program issued a TPRETURN without having control of the connection.
	TPRECV	Received by the originator of the connection, indicating that the subordinate program issued a TPRETURN with TPSUCCESS or TPFAIL and a valid data record, but an error occurred that prevented the call from completing.
TPEV_SVCFAIL	TPSEND	Received by the originator of the connection, indicating that the subordinate program issued a TPRETURN without having control of the connection, and TPRETURN was called with TPFAIL or TPEXIT and no data.
	TPRECV	Received by the originator of the connection, indicating that the subordinate service finished unsuccessfully (TPRETURN was called with TPFAIL or TPEXIT).
TPEV_SVCSUCC	TPRECV	Received by the originator of the connection, indicating that the subordinate service finished successfully; that is, it called TPRETURN with TPSUCCESS.

8 Writing Event-based Clients and Servers

This topic includes the following sections:

- Overview of Events
- Defining the Unsolicited Message Handler
- Sending Unsolicited Messages
- Checking for Unsolicited Messages
- Getting Unsolicited Messages
- Subscribing to Events
- Unsubscribing from Events
- Posting Events

Overview of Events

Event-based communication provides a method for a BEA Tuxedo system process to be notified when a specific situation (event) occurs.

The BEA Tuxedo system supports two types of event-based communication:

- Unsolicited events

- Brokered events

Unsolicited Events

Unsolicited events are messages used to communicate with client programs that are not waiting for and/or expecting a message.

Brokered Events

Brokered events enable a client and a server to communicate transparently with one another via an "anonymous" broker that receives and distributes messages. Such brokering is another client/server communication paradigm that is fundamental to the BEA Tuxedo system.

The EventBroker is a BEA Tuxedo subsystem that receives and filters event posting messages, and distributes them to subscribers. A *poster* is a BEA Tuxedo system process that detects when a specific event has occurred and reports (posts) it to the EventBroker. A *subscriber* is a BEA Tuxedo system process with a standing request to be notified whenever a specific event has been posted.

The BEA Tuxedo system does not impose a fixed ratio of service requesters to service providers; an arbitrary number of posters can post a message for an arbitrary number of subscribers. The posters simply post events, without knowing which processes receive the information or how the information is handled. Subscribers are notified of specified events, without knowing who posted the information. In this way, the EventBroker provides complete location transparency.

Typically, EventBroker applications are designed to handle exception events. An application designer must decide which events in the application constitute exception events and need to be monitored. In a banking application, for example, it might be useful to post an event whenever an unusually large amount of money is withdrawn, but it would not be particularly useful to post an event for every withdrawal transaction. In addition, not all users would need to subscribe to that event; perhaps only the branch manager would need to be notified.

Notification Actions

The EventBroker may be configured such that whenever an event is posted, the EventBroker invokes one or more notification actions for clients and/or servers that have subscribed. The following table lists the types of notification actions that the EventBroker can take.

Table 8-1 EventBroker Notification Actions

Notification Action	Description
Unsolicited notification message	Clients may receive event notification messages in their unsolicited message handling routine, just as if they were sent by the `TPNOTIFY` routine.
Service call	Servers may receive event notification messages as input to service routines, just as if they were sent by `TPACALL`.
Reliable queue	Event notification messages may be stored in a BEA Tuxedo system reliable queue, using `TPDEQUEUE(3cbl)`. Event notification records are stored until requests for contents are issued. A BEA Tuxedo system client or server process may call `TPDEQUEUE(3cbl)` to retrieve these notification records, or alternately `TMQFORWARD(5)` may be configured to automatically dispatch a BEA Tuxedo system service routine that retrieves a notification record. For more information on /Q, see *Using the ATMI /Q Component*.

In addition, the application administrator may create an `EVENT_MIB(5)` entry (by using the BEA Tuxedo administrative API) that performs the following notification actions:

- Invokes a system command
- Writes a message to the system's log file on disk

Note: Only the BEA Tuxedo application administrator is allowed to create an `EVENT_MIB(5)` entry.

For information on the `EVENT_MIB(5)`, refer to the *File Formats, Data Descriptions, MIBs, and System Processes Reference*.

EventBroker Servers

TMUSREVT is the BEA Tuxedo system-supplied server that acts as an EventBroker for *user events*. TMUSREVT processes event report message records, and then filters and distributes them. The BEA Tuxedo application administrator must boot one or more of these servers to activate event brokering.

TMSYSEVT is the BEA Tuxedo system-supplied server that acts as an EventBroker for *system-defined events*. TMSYSEVT and TMUSREVT are similar, but separate servers are provided to allow the application administrator the ability to have different replication strategies for processing notifications of these two types of events. Refer to *Setting Up a BEA Tuxedo Application* for additional information.

System-defined Events

The BEA Tuxedo system itself detects and posts certain predefined events related to system warnings and failures. These tasks are performed by the EventBroker. For example, system-defined events include configuration changes, state changes, connection failures, and machine partitioning. For a complete list of system-defined events detected by the EventBroker, see EVENTS(5) in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*.

System-defined events are defined in advance by the BEA Tuxedo system code and do not require posting. The name of a system-defined event, unlike that of an application-defined event, always begins with a dot (“.”). Names of application-defined events may not begin with a leading dot.

Clients and servers can subscribe to system-defined events. These events, however, should be used mainly by application administrators, not by every client in the application.

When incorporating the EventBroker into your application, remember that it is not intended to provide a mechanism for high-volume distribution to many subscribers. Do not attempt to post an event for every activity that occurs, and do not expect all clients and servers to subscribe. If you overload the EventBroker, system performance may be adversely affected and notifications may be dropped. To minimize the possibility of overload, the application administrator should carefully tune the operating system IPC resources, as explained in *Installing the BEA Tuxedo System*.

Programming Interface for the EventBroker

EventBroker programming interfaces are available for all BEA Tuxedo system server and client processes, including Workstation, in both C and COBOL.

The programmer's job is to code the following sequence:

1. A client or server *posts* a record to an application-defined event name.
2. The posted record is transmitted to any number of processes that have *subscribed* to the event.

Subscribers may be notified in a variety of ways (as discussed in "Notification Actions"), and events may be filtered. Notification and filtering are configured through the programming interface, as well as through the BEA Tuxedo system administrative API.

Defining the Unsolicited Message Handler

To define the unsolicited message handler, use the `TPSETUNSOL(3cbl)` routine with the following signature:

```
01 CURR-ROUTINE   PIC S9(9) COMP-5.
01 PREV-ROUTINE   PIC S9(9) COMP-5.
01 TPSTATUS-REC.
   COPY TPSTATUS.
CALL "TPSETUNSOL" USING CURR-ROUTINE PREV-ROUTINE TPSTATUS-REC.
```

`TPSETUNSOL` allows a client to identify the routine that should be invoked when an unsolicited message is received by the BEA Tuxedo system libraries. Before the first call to `TPSETUNSOL`, any unsolicited messages received by the BEA Tuxedo system libraries on behalf of the client are logged and ignored. The method used by the system for notification and detection is determined by the application default, which can be overridden on a per-client basis. For more information, refer to `TPINITIALIZE(3cbl)` in the *BEA Tuxedo ATMI COBOL Function Reference*.

The `CURR-ROUTINE` parameter identifies one of 16 predefined routines that provide unsolicited message handling: eight C routines, `tm_displatch1` through `_tm_dispatch8`, and eight COBOL routines, `TMDISPATCH9` through `TMDISPATCH16`. (Alternatively, if you set `CURR-ROUTINE` to a value of 0, any unsolicited messages

received by the BEA Tuxedo system libraries on behalf of the client are logged and ignored.) The C routines must conform to the parameter definition provided on TPSETUNSOL(3cbl). When a COBOL routine is used, TPGETUNSOL must be called to receive the data.

The following sample code shows how to set an unsolicited routine in a COBOL program.

Listing 8-1 Setting an Unsolicited Routine

```
*
* Call TPSETUNSOL - Set a COBOL unsolicited message handler
*  Routine TMDISPATCH9 will be called
*
   MOVE 9 to CURR-ROUTINE.
   CALL "TPSETUNSOL" USING
                      CURR-ROUTINE
                      PREV-ROUTINE
                      TPSTATUS-REC.
   IF NOT TPOK
            Routine TMDISPATCH9 will receive unsolicited messages
   ELSE
              Process error condition
```

Sending Unsolicited Messages

The BEA Tuxedo system allows unsolicited messages to be sent to client processes without disturbing the processing of request/response calls or conversational communications.

Unsolicited messages can be sent to client processes by name, using TPBROADCAST(3cbl), or by an identifier received with a previously processed message, using TPNOTIFY(3cbl). Messages sent via TPBROADCAST can originate either in a service or in another client. Messages sent via TPNOTIFY can originate only in a service.

Broadcasting Messages by Name

The TPBROADCAST(3cbl) routine allows a message to be sent to registered clients of the application. It can be called by a service or another client. Registered clients are those that have successfully made a call to TPINITIALIZE and have not yet made a call to TPTERM.

Use the following signature to call the TPBROADCAST routine:

```
01 TPBCTDEF-REC.
   COPY TPBCTDEF.
01 TPTYPE-REC.
   COPY TPTYPE.
01 DATA-REC.
   COPY User Data.
01 TPSTATUS-REC.
   COPY TPSTATUS.
CALL "TPBROADCAST" USING TPBCTDEF-REC TPTYPE-REC DATA-REC TPSTATUS-REC.
```

The following table describes the members of the TPBCTDEF-REC data structure.

Table 8-2 TPBCTDEF-REC Data Structure Members

Member	Description
LMID	Pointer to the logical machine identifier for the client. A value of SPACES acts as a wildcard, so that a message can be directed to groups of clients.
USRNAME	Username of the client process, if one exists. A value of SPACES acts as a wildcard, so that a message can be directed to groups of clients.
CLTNAME	Client name of the client process, if one exists. A value of NULL acts as a wildcard, so that a message can be directed to groups of clients.
Settings (such as TPBLOCK-FLAG)	Settings for the TPBROADCAST command. Refer to TPBROADCAST(3cbl) in the *BEA Tuxedo ATMI COBOL Function Reference* for information on available settings.

Refer to "Defining a Service" on page 5-10 for a description of the TPTYPE-REC record.

The following example illustrates a call to TPBROADCAST for which all clients are targeted. The message to be sent is contained in a STRING record.

Listing 8-2 Using TPBROADCAST

```
    . . .
************************************************
* Prepare the record to broadcasted
************************************************
    MOVE "HELLO, WORLD" TO DATA-REC.
    MOVE 11 TO LEN.
    MOVE "STRING" TO REC-TYPE.
*
    SET TPNOBLOCK TO TRUE.
    SET TPNOTIME TO TRUE.
    SET TPSIGRSTRT TO TRUE.
*
    MOVE SPACES TO LMID.
    MOVE SPACES TO USRNAME.
    MOVE SPACES TO CLTNAME.
    CALL "TPBROADCAST" USING TPBCTDEF-REC
                       TPTYPE-REC
                       DATA-REC
                       TPSTATUS-REC.
    IF NOT TPOK
        error processing
```

Broadcasting Messages by Identifier

The TPNOTIFY(3cbl) routine is used to broadcast a message using an identifier received with a previously processed message. It can be called only from a service.

Use the following signature to call the TPNOTIFY routine:

```
01 TPSVCDEF-REC.
   COPY TPSVCDEF.
01 TPTYPE-REC.
   COPY TPTYPE.
01 DATA-REC.
   COPY User Data.
01 TPSTATUS-REC.
```

```
   COPY TPSTATUS.
CALL "TPNOTIFY" USING TPSVCDEF-REC TPTYPE-REC DATA-REC TPSTATUS-REC.
```

Refer to "Writing Global Transactions" on page 9-1 for information on the TPSVCDEF-REC data structure, and "Defining a Service" on page 5-10 for a description of the TPTYPE-REC record.

Checking for Unsolicited Messages

To check for unsolicited messages while running the client in "dip-in" notification mode, use the TPCHKUNSOL(3cbl) routine with the following signature:

```
01 MSG-NUM          PIC S9(9)  COMP-5.
01 TPSTATUS-REC.
   COPY TPSTATUS.
CALL "TPCHKUNSOL" USING MSG-NUM TPSTATUS-REC.
```

If any messages are pending, the system invokes the unsolicited message handling routine that was specified using TPSETUNSOL. Upon completion, the routine returns either the number of unsolicited messages that were processed and sets TP-STATUS to [TPOK].

If you issue this routine when the client is running in SIGNAL-based, thread-based notification mode, or is ignoring unsolicited messages, the routine has no impact and returns immediately.

The following example shows how to check for the arrival of an unsolicited message.

Listing 8-3 Arrival of an Unsolicited Message

```
*
* Check for unsolicited messages
*
    CALL "TPCHKUNSOL" USING MESS-NUM
                      TPSTATUS-REC.
*
    IF TPOK
         IF MESS-NUM IS = 0
            No messages were processed by the
            unsolicited routine
```

```
    ELSE
         MESS-NUM  number of messages were
            processed by the unsolicited routine
    END-IF
    ELSE
       process error
   END-IF
```

Getting Unsolicited Messages

To get unsolicited messages, you must call the `TPGETUNSOL(3cbl)` routine. This routine can be called, however, only from an unsolicited message handler. Use the following signature to call the `TPGETUNSOL` routine:

```
01 TPTYPE-REC.
   COPY TPTYPE.
01 DATA-REC.
   COPY User data.
01 TPSTATUS-REC.
   COPY TPSTATUS.
CALL "TPGETUNSOL" USING TPTYPE-REC DATA-REC TPSTATUS-REC.
```

Refer to "Defining a Service" on page 5-10 for a description of the `TPTYPE-REC` record.

The following example shows how to get an unsolicited message.

Listing 8-4 Getting an Unsolicited Message

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  TMDISPATCH9.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.  USL-486.
 OBJECT-COMPUTER.  USL-486.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
*
 01 TPTYPE-REC.
```

```
    COPY TPTYPE.
*
 01 TPSTATUS-REC.
    COPY TPSTATUS.
*
 01  DATA-REC                 PIC X(1000).
*
 PROCEDURE DIVISION.
*
 A-000.
*
   MOVE "CARRAY" TO REC-TYPE.
   MOVE 1000  TO LEN.
   CALL "TPGETUNSOL" USING TPTYPE-REC
                     DATA-REC
                     TPSTATUS-REC.
   IF  NOT TPOK
            error processing
*
   Process message
   DISPLAY "TPGETUNSOL IS TPOK".
   DISPLAY "MESSAGE IS" DATA-REC.
   DISPLAY "LENGTH IS" LEN.
   EXIT PROGRAM.
*
```

Subscribing to Events

The TPSUBSCRIBE(3cbl) routine enables a BEA Tuxedo system ATMI client or server to subscribe to an event.

A subscriber can be notified through an unsolicited notification message, a service call, a reliable queue, or other notification methods configured by the application administrator. (For information about configuring alternative notification methods, refer to *Setting Up a BEA Tuxedo Application*.)

Use the following signature to call the TPSUBSCRIBE routine:

```
01  TPEVTDEF-REC.
  COPY TPEVTDEF.
```

```
01  TPQUEDEF-REC.
  COPY TPQUEDEF.

01 TPSTATUS-REC.
  COPY TPSTATUS.

CALL "TPSUBSCRIBE" USING TPEVTDEF-REC TPQUEDEF-REC TPSTATUS-REC
```

The TPEVTDEF-REC data structure signature is as follows:

```
05 TPBLOCK-FLAG     PIC S9(9) COMP-5.
  88 TPBLOCK       VALUE 0.
  88 TPNOBLOCK      VALUE 1.
05 TPTRAN-FLAG     PIC S9(9) COMP-5.
  88 TPTRAN        VALUE 0.
  88 TPNOTRAN       VALUE 1.
05 TPREPLY-FLAG     PIC S9(9) COMP-5.
  88 TPREPLY       VALUE 0.
  88 TPNOREPLY      VALUE 1.
05 TPTIME-FLAG     PIC S9(9) COMP-5.
  88 TPTIME        VALUE 0.
  88 TPNOTIME       VALUE 1.
05 TPSIGRSTRT-FLAG   PIC S9(9) COMP-5.
  88 TPNOSIGRSTRT     VALUE 0.
  88 TPSIGRSTRT      VALUE 1.
05 TPEV-METHOD-FLAG   PIC S9(9) COMP-5.
  88 TPEVNOTIFY      VALUE 0.
  88 TPEVSERVICE     VALUE 1.
  88 TPEVQUEUE      VALUE 2.
05 TPEV-PERSIST-FLAG  PIC S9(9) COMP-5.
  88 TPEVNOPERSIST    VALUE 0.
  88 TPEVPERSIST     VALUE 1.
05 TPEV-TRAN-FLAG    PIC S9(9) COMP-5.
  88 TPEVNOTRAN      VALUE 0.
  88 TPEVTRAN       VALUE 1.
*
05 EVENT-COUNT     PIC S9(9) COMP-5.
05 SUBSCRIPTION-HANDLE PIC S9(9) COMP-5.
05 NAME-1        PIC X(31).
05 NAME-2        PIC X(31).
05 EVENT-NAME      PIC X(31).
05 EVENT-EXPR      PIC X(255).
05 EVENT-FILTER     PIC X(255).
```

The following table describes the members of the TPEVTDEF-REC data structure.

Member	Description
EVENT-COUNT	Event count.
SUBSCRIPTION-HANDLE	Subscription handle.
NAME-1, NAME-2	Name of queued spaces. If the subscriber sets TPEVQUEUE, then event notifications are enqueued to the queue space named by NAME-1 and the queue named by NAME-2.
EVENT-NAME	Event name.
EVENT-EXPR	Set of events to which to subscribe. Consists of a null-terminated string of up to 255 characters containing a regular expression. Regular expressions are of the form specified in tpsubscribe(3c) as described in the *Programming BEA Tuxedo ATMI Applications Using C*. For example, if eventexpr is set to: ■ "\\..*" — the caller is subscribing to all system-defined events. ■ "\\.SysServer.*" — the caller is subscribing to all system-defined events related to servers. ■ "[A-Z].*" — the caller is subscribing to all user events starting with A-Z. ■ ".*(ERR|err).*" — the caller is subscribing to all user events containing either the substring ERR or the substring err (for example, account_error and ERROR_STATE events would both qualify).

Member	Description
EVENT-FILTER	String containing a Boolean filter rule that must be evaluated successfully before the Event Broker posts the event. Upon receiving an event to be posted, the Event Broker applies the filter rule, if one exists, to the posted event's data. If the data passes the filter rule, the Event Broker invokes the notification method specified; otherwise, the Event Broker ignores the notification method. The caller can subscribe to the same event multiple times with different filter rules. By using the event filtering capability, subscribers can be more discriminating about the events for which they are notified. For example, a poster can post an event for withdrawals greater than $10,000.00, but a subscriber may want to specify a higher threshold for being notified, such as $50,000.00. Or, a subscriber may want to be notified of large withdrawals only if made by customers with specified IDs. Filter rules are specific to the typed records to which they are applied. Refer to the TPSUBSCRIBE(3cbl) reference page in the *BEA Tuxedo ATMI COBOL Function Reference* for further information on filter rules.
SETTINGS (TPBLOCK-FLAG, TPTRAN-FLAG, and so on)	Miscellaneous settings that control the server characteristics. For more information on the settings, refer to the *BEA Tuxedo ATMI COBOL Function Reference*.

Refer to *Using the ATMI /Q Component* for more information on the TPQUEDEF-REC data structure.

You can subscribe to both system- and application-defined events using the TPSUBSCRIBE routine.

For purposes of subscriptions (and for MIB updates), service routines executed in a BEA Tuxedo system server process are considered to be trusted code.

Refer to TPSUBSCRIBE(3cbl) in the *BEA Tuxedo ATMI COBOL Function Reference* for more information on the routine.

Unsubscribing from Events

The TPUNSUBSCRIBE(3cbl) routine enables a BEA Tuxedo system ATMI client or server to unsubscribe from an event.

Use the following signature to call the TPUNSUBSCRIBE routine:

```
01  TPEVTDEF-REC.
  COPY TPEVTDEF.

01 TPSTATUS-REC.
  COPY TPSTATUS.

CALL "TPUNSUBSCRIBE" USING TPEVTDEF-REC TPSTATUS-REC
```

Refer to "Subscribing to Events" on page 8-11 for a detailed description of the TPEVTDEF-REC data structure, and to *Using the ATMI /Q Component* for more information on the TPQUEDEF-REC data structure.

Posting Events

The TPPOST(3cbl) routine enables a BEA Tuxedo ATMI client or server to post an event.

Use the following signature to call the TPPOST routine:

```
01  TPEVTDEF-REC.
  COPY TPEVTDEF.

01 TPTYPE-REC.
  COPY TPSTATUS.

01 TPDATA-REC.
  COPY TPSTATUS.

01 TPSTATUS-REC.
  COPY TPSTATUS.

CALL "TPPST" USING TPEVTDEF-REC TPTYPE-REC TPDATA-REC TPSTATUS-REC
```

Refer to "Subscribing to Events" on page 8-11 for a detailed description of the `TPEVTDEF-REC` data structure, and to "Defining a Service" on page 5-10 for a description of the `TPTYPE-REC` record.

CHAPTER

9 Writing Global Transactions

This topic includes the following sections:

- What Is a Global Transaction?
- Starting the Transaction
- Terminating the Transaction
- Terminating the Transaction
- Implicitly Defining a Global Transaction
- Defining Global Transactions for an XA-Compliant Server Group
- Testing Whether a Transaction Has Started

What Is a Global Transaction?

A global transaction is a mechanism that allows a set of programming tasks, potentially using more than one resource manager and potentially executing on multiple servers, to be treated as one logical unit.

Once a process is in transaction mode, any service requests made to servers may be processed on behalf of the current transaction. The services that are called and join the transaction are referred to as *transaction participants*. The value returned by a participant may affect the outcome of the transaction.

A global transaction may be composed of several local transactions, each accessing the same resource manager. The resource manager is responsible for performing concurrency control and atomicity of updates. A given local transaction may be either successful or unsuccessful in completing its access; it cannot be partially successful.

A maximum of 16 server groups can participate in a single transaction.

The BEA Tuxedo system manages a global transaction in conjunction with the participating resource managers and treats it as a specific sequence of operations that is characterized by atomicity, consistency, isolation, and durability. In other words, a global transaction is a logical unit of work in which:

- All portions either succeed or have no effect.
- Operations are performed that correctly transform resources from one consistent state to another.
- Intermediate results are not accessible to other transactions, although some processes in a transaction may access the data associated with another process.
- Once a sequence is complete, its results cannot be altered by any kind of failure.

The BEA Tuxedo system tracks the status of each global transaction and determines whether it should be committed or rolled back.

Starting the Transaction

To start a global transaction, use the `TPBEGIN(3cbl)` routine with the following signature:

```
*
  01 TPTRXDEF-REC.
     COPY TPTRXDEF.
*
  01 TPSTATUS-REC.
     COPY TPSTATUS.
*
   CALL "TPBEGIN" USING TPTRXDEF-REC TPSTATUS-REC.
```

The following table describes the TPTRXDEF-REC structure fields

Table 9-1 TPTRXDEF Structure Field

Field	Description
T-OUT	Specifies the amount of time, in seconds, a transaction can execute before timing out. You can set this value to the maximum number of seconds allowed by the system, by specifying a value of 0. In other words, you can set *timeout* to the maximum value for an unsigned long as defined by the system. The use of 0 or an unrealistically large value for the T-OUT parameter delays system detection and reporting of errors. The system uses the T-OUT parameter to ensure that responses to service requests are sent within a reasonable time, and to terminate transactions that encounter problems such as network failures before executing a commit. For a transaction in which a person is waiting for a response, you should set this parameter to a small value: if possible, less than 30 seconds. In a production system, you should set T-OUT to a value large enough to accommodate expected delays due to system load and database contention. A small multiple of the expected average response time is often an appropriate choice. **Note:** The value assigned to the T-OUT parameter should be consistent with that of the SCANUNIT parameter set by the BEA Tuxedo application administrator in the configuration file. The SCANUNIT parameter specifies the frequency with which the system checks, or *scans*, for timed-out transactions and blocked calls in service requests. The value of this parameter represents the interval of time between these periodic scans, referred to as the *scanning unit*. You should set the T-OUT parameter to a value that is greater than the scanning unit. If you set the T-OUT parameter to a value smaller than the scanning unit, there will be a discrepancy between the time at which a transaction times out and the time at which this timeout is discovered by the system. The default value for SCANUNIT is 10 seconds. You may need to discuss the setting of the T-OUT parameter with your application administrator to make sure the value you assign to the T-OUT parameter is compatible with the values assigned to your system parameters.
TRANID	Transaction identifier.

Any process may call TPBEGIN unless the process is already in transaction mode. If TPBEGIN is called in transaction mode, the call fails due to a protocol error and TP-STATUS is set to TPEPROTO. If the process is in transaction mode, the transaction is unaffected by the failure.

The following example provides a high-level view of how a global transaction is defined.

Listing 9-1 Delineating a Transaction

```
. . .
MOVE 0 TO T-OUT.
CALL "TPBEGIN" USING
TPTRXDEF-REC
TPSTATUS-REC.
IF  NOT TPOK
    error processing
. . .
    program statements
. . .
CALL "TPCOMMIT" USING
                TPTRXDEF-REC
                TPSTATUS-REC.
IF  NOT TPOK
    error processing
```

The following example shows how an outstanding reply can cause an error.

Listing 9-2 Error - Starting a Transaction with an Outstanding Reply

```
  . . .
  MOVE "BUY" TO SERVICE-NAME.
  SET TPBLOCK TO TRUE.
  SET TPNOTRAN TO TRUE.
  SET TPREPLY TO TRUE.
  SET TPNOTIME TO TRUE.
  SET TPSIGRSTRT TO TRUE.
  CALL "TPACALL" USING
                 TPSVCDEF-REC
                 TPTYPE-REC
                 BUY-REC
```

```
                TPSTATUS-REC.
  IF  NOT TPOK
      error processing
  . . .
  MOVE 0 TO T-OUT.
  CALL "TPBEGIN" USING
                TPTRXDEF-REC
                TPSTATUS-REC.
  IF  NOT TPOK
      error processing
* ERROR TP-STATUS is set to TPEPROTO
  . . .
      program statements
  . . .
  SET TPBLOCK TO TRUE.
  SET TPNOTRAN TO TRUE.
  SET TPCHANGE TO TRUE.
  SET TPNOTIME TO TRUE.
  SET TPSIGRSTRT TO TRUE.
  SET TPGETANY TO TRUE.
  CALL "TPGETRPLY" USING
                  TPSVCDEF-REC
                  TPTYPE-REC
                  WK-AREA
                  TPSTATUS-REC.
  IF  NOT TPOK
      error processing
```

If a transaction times out, a call to `TPCOMMIT` causes the transaction to be aborted. As a result, `TPCOMMIT` fails and sets `TP-STATUS` to `TPEABORT`.

The following example shows how to test for a transaction timeout. Note that the value of `T-OUT` is set to 30 seconds.

Listing 9-3 Testing for Transaction Timeout

```
. . .
MOVE 30 TO T-OUT.
CALL "TPBEGIN" USING TPTRXDEF-REC TPSTATUS-REC.
IF  NOT TPOK
    MOVE "Failed to BEGIN a transaction" TO LOG-REC-TEXT.
    MOVE 29 to LOG-REC-LEN
    CALL "USERLOG" USING
                  LOG-REC-TEXT
```

```
                    LOG-REC-LEN
                    TPSTATUS-REC
    CALL "TPTERM" USING
                    TPSTATUS-REC
    PERFORM A-999-EXIT.
. . .
       communication CALL statements
. . .
IF  TPETIME
    CALL "TPABORT" USING
                    TPTRXDEF-REC
                    TPSTATUS-REC
IF  NOT TPOK
       error processing
ELSE
   CALL "TPCOMMIT" USING
                    TPTRXDEF-REC
                    TPSTATUS-REC
   IF  NOT TPOK
        error processing
```

Note: When a process is in transaction mode and makes a communication call with TPNOTRAN, it prohibits the called service from becoming a participant in the current transaction. Whether the service request succeeds or fails has no impact on the outcome of the transaction. The transaction can still timeout while waiting for a reply that is due from a service, whether it is part of the transaction or not. Refer to "Managing Errors" on page 11-1 for more information on the effects of the TPNOTRAN flag.

The following example shows how to define a transaction.

Listing 9-4 Defining a Transaction

```
 DATA DIVISION.
 WORKING-STORAGE SECTION.
*
 01  TPTYPE-REC.
 COPY TPTYPE.
*
 01 TPSTATUS-REC.
 COPY TPSTATUS.
*
 01 TPINFDEF-REC.
```

```
 COPY TPINFDEF.
*
 01  TPSVCDEF-REC.
 COPY TPSVCDEF.
*
 01  TPTRXDEF-REC.
 COPY TPTRXDEF.
*
 01 LOG-REC              PIC X(30) VALUE " ".
 01 LOG-REC-LEN          PIC S9(9)  COMP-5.
*
 01 USR-DATA-REC         PIC X(16).
*
 01 AUDV-REC.
     05  AUDV-BRANCH-ID      PIC S9(9) COMP-5.
     05  AUDV-BALANCE        PIC S9(9) COMP-5.
     05  AUDV-ERRMSG         PIC X(60).
*
 PROCEDURE DIVISION.
*
 A-000.
  . . .
* Get Command Line Options  set Variables (Q-BRANCH)
  MOVE SPACES TO USRNAME.
  MOVE SPACES TO CLTNAME.
  MOVE SPACES TO PASSWD.
  MOVE SPACES TO GRPNAME.
  CALL "TPINITIALIZE" USING TPINFDEF-REC
                      USR-DATA-REC
                      TPSTATUS-REC.
  IF  NOT TPOK
      MOVE "Failed to join application" TO LOG-REC
      MOVE 26 TO LOG-REC-LEN
       CALL "USERLOG" USING LOG-REC
                      LOG-REC-LEN
                      TPSTATUS-REC
       PERFORM A-999-EXIT.
* Start global transaction
  MOVE 30 TO T-OUT.
  CALL "TPBEGIN" USING TPTRXDEF-REC TPSTATUS-REC.
  IF  NOT TPOK
      MOVE 29 to LOG-REC-LEN
      MOVE "Failed to begin a transaction" TO LOG-REC
      CALL "USERLOG" USING LOG-REC
                     LOG-REC-LEN
                     TPSTATUS-REC
      PERFORM DO-TPTERM.
* Set up record
  MOVE Q-BRANCH TO AUDV-BRANCH-ID.
```

```
  MOVE ZEROS TO AUDV-BALANCE.
  MOVE SPACES TO AUDV-ERRMSG.
* Set up TPCALL records
  MOVE "GETBALANCE" TO SERVICE-NAME.
  MOVE "VIEW" TO REC-TYPE.
  MOVE LENGTH OF AUDV-REC TO LEN.
  SET TPBLOCK TO TRUE.
  SET TPTRAN IN TPSVCDEF-REC TO TRUE.
  SET TPNOTIME TO TRUE.
  SET TPSIGRSTRT TO TRUE.
  SET TPCHANGE TO TRUE.
*
  CALL "TPCALL" USING TPSVCDEF-REC
                TPTYPE-REC
                AUDV-REC
                TPTYPE-REC
                AUDV-REC
                TPSTATUS-REC.
  IF  NOT TPOK
      MOVE 19 to LOG-REC-LEN
      MOVE "Service call failed" TO LOG-REC
      CALL "USERLOG" USING LOG-REC
                     LOG-REC-LEN
                     TPSTATUS-REC
      PERFORM DO-TPABORT
      PERFORM DO-TPTERM.
* Commit global transaction
  CALL "TPCOMMIT" USING TPTRXDEF-REC
                  TPSTATUS-REC
  IF  NOT TPOK
      MOVE 16 to LOG-REC-LEN
      MOVE "Failed to commit" TO LOG-REC
      CALL "USERLOG" USING LOG-REC
                LOG-REC-LEN
                TPSTATUS-REC
      PERFORM DO-TPTERM.
* Show results only when transaction has completed successfully
  DISPLAY "BRANCH=" Q-BRANCH.
  DISPLAY "BALANCE=" AUDV-BALANCE.
  PERFORM DO-TPTERM.
* Abort the transaction
 DO-TPABORT.
  CALL "TPABORT" USING TPTRXDEF-REC
                 TPSTATUS-REC
  IF  NOT TPOK
      MOVE 26 to LOG-REC-LEN
      MOVE "Failed to abort transaction" TO LOG-REC
      CALL "USERLOG" USING LOG-REC
                     LOG-REC-LEN
```

```
                    TPSTATUS-REC.
* Leave the application
 DO-TPTERM.
  CALL "TPTERM" USING TPSTATUS-REC.
  IF  NOT TPOK
      MOVE 27 to LOG-REC-LEN
      MOVE "Failed to leave application" TO LOG-REC
      CALL "USERLOG" USING LOG-REC
                    LOG-REC-LEN
                    TPSTATUS-REC.
                    EXIT PROGRAM.
*
 A-999-EXIT.
*
 EXIT PROGRAM.
```

Terminating the Transaction

To end a global transaction, call `TPCOMMIT(3cbl)` to commit the current transaction, or `TPABORT(3cbl)` to abort the transaction and roll back all operations.

Note: If `TPCALL`, `TPACALL`, or `TPCONNECT` is called by a process that has explicitly set `TPNOTRAN`, the operations performed by the called service do not become part of the current transaction. In other words, when you call the `TPABORT` routine, the operations performed by these services are not rolled back.

Committing the Current Transaction

The `TPCOMMIT(3cbl)` routine commits the current transaction. When `TPCOMMIT` returns successfully, all changes to resources as a result of the current transaction become permanent.

Use the following signature to call the `TPCOMMIT` routine:

```
*
  01 TPTRXDEF-REC.
     COPY TPTRXDEF.
```

```
*
  01 TPSTATUS-REC.
     COPY TPSTATUS.
*
   CALL "TPCOMMIT" USING TPTRXDEF-REC TPSTATUS-REC.
```

Refer to "Starting the Transaction" on page 9-2 for a description of the TPTRXDEF-REC structure.

Prerequisites for a Transaction Commit

For TPCOMMIT to succeed, the following conditions must be true:

- The calling process must be the same one that initiated the transaction with a call to TPBEGIN.
- The calling process must have no transactional replies (calls made without the TPNOTRAN flag) outstanding.
- The transaction must not be in a rollback-only state and must not be timed out.

If the first condition is false, the call fails and TP-STATUS is set to TPEPROTO, indicating a protocol error. If the second or third condition is false, the call fails and TP-STATUS is set to TPEABORT, indicating that the transaction has been rolled back. If TPCOMMIT is called by the initiator with outstanding transaction replies, the transaction is aborted and those reply descriptors associated with the transaction become invalid. If a participant calls TPCOMMIT or TPABORT, the transaction is unaffected.

A transaction is placed in a rollback-only state if any service call returns TPFAIL or indicates a service error. If TPCOMMIT is called for a rollback-only transaction, the routine cancels the transaction, returns -1, and sets TP-STATUS to TPEABORT. The results are the same if TPCOMMIT is called for a transaction that has already timed out: TPCOMMIT returns -1 and sets TP-STATUS to TPEABORT. Refer to "Managing Errors" on page 11-1 for more information on transaction errors.

Two-phase Commit Protocol

When the TPCOMMIT routine is called, it initiates the *two-phase commit protocol*. This protocol, as the name suggests, consists of two steps:

1. Each participating resource manager indicates a readiness to commit.

2. The initiator of the transaction gives permission to commit to each participating resource manager.

The commit sequence begins when the transaction initiator calls the TPCOMMIT routine. The BEA Tuxedo TMS server process in the designated coordinator group contacts the TMS in each participant group that is to perform the first phase of the commit protocol. The TMS in each group then instructs the resource manager (RM) in that group to commit using the XA protocol that is defined for communications between the Transaction Managers and RMs. The RM writes, to stable storage, the states of the transaction before and after the commit sequence, and indicates success or failure to the TMS. The TMS then passes the response back to the coordinating TMS.

When the coordinating TMS has received a success indication from all groups, it logs a statement to the effect that a transaction is being committed and sends second-phase commit notifications to all participant groups. The RM in each group then finalizes the transaction updates.

If the coordinator TMS is notified of a first-phase commit failure from any group, or if it fails to receive a reply from any group, it sends a rollback notification to each RM and the RMs back out all transaction updates. TPCOMMIT then fails and sets TP-STATUS to TPEABORT.

Selecting Criteria for a Successful Commit

When more than one group is involved in a transaction, you can specify which of two criteria must be met for TPCOMMIT to return successfully:

- When all participants have indicated a readiness to commit (that is, when all participants have reported that phase 1 of the two-phase commit has been logged as complete and the coordinating TMS has written its decision to commit to stable storage)
- When all participants have finished phase 2 of the two-phase commit

To specify one of these prerequisites, set the CMTRET parameter in the RESOURCES section of the configuration file to one of the following values:

- LOGGED—to require completion of phase 1
- COMPLETE—to require completion of phase 2

By default, CMTRET is set to COMPLETE.

Trade-offs Between Possible Commit Criteria

In most cases, when all participants in a global transaction have logged successful completion of phase 1, they do not fail to complete phase 2. By setting CMTRET to LOGGED, you allow a slightly faster return of calls to TCOMMIT, but you run the slight risk that a participant may heuristically complete its part of the transaction in a way that is not consistent with the commit decision.

Whether it is prudent to accept the risk depends to a large extent on the nature of your application. If your application demands complete accuracy (for example, if you are running a financial application), you should probably wait until all participants fully complete the two-phase commit process before returning. If your application is more time-sensitive, you may prefer to have the application execute faster at the expense of accuracy.

Aborting the Current Transaction

Use the TPABORT(3cbl) routine to indicate an abnormal condition and explicitly abort a transaction. This function invalidates the call descriptors of any outstanding transactional replies. None of the changes produced by the transaction are applied to the resource. Use the following signature to call the TPABORT routine:

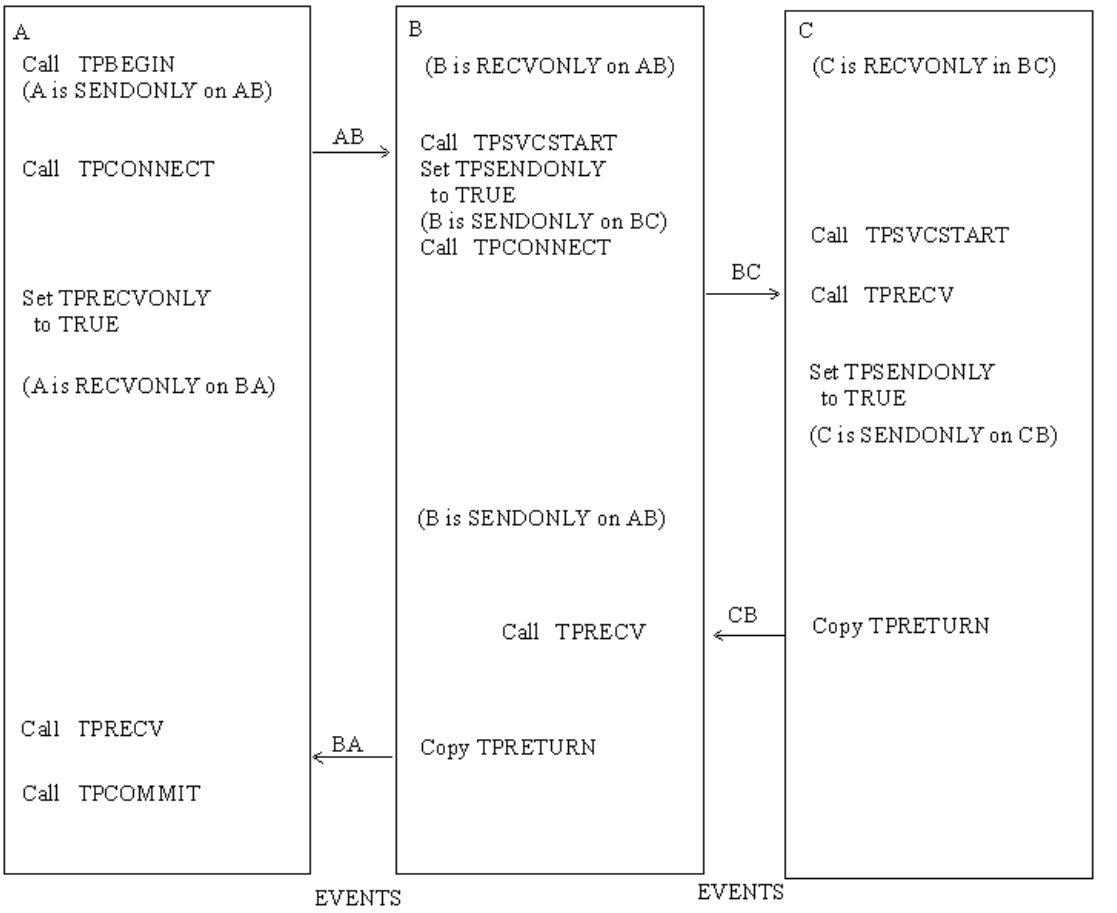
```
*
  01 TPTRXDEF-REC.
     COPY TPTRXDEF.
*
  01 TPSTATUS-REC.
     COPY TPSTATUS.
*
   CALL "TPABORT" USING TPTRXDEF-REC TPSTATUS-REC.
```

Refer to “Starting the Transaction” on page 9-2 for a description of the TPTRXDEF-REC structure.

Example: Committing a Transaction in Conversational Mode

The following figure illustrates a conversational connection hierarchy that includes a global transaction.

Figure 9-1 Connection Hierarchy in Transaction Mode



The connection hierarchy is created through the following process:

1. A client (process A) initiates a connection in transaction mode by calling TPBEGIN and TPCONNECT.
2. The client calls subsidiary services, which are executed.
3. As each subordinate service completes, it sends a reply indicating success or failure (TPEV_SVCSUCC or TPEV_SVCFAIL, respectively) back up through the hierarchy to the process that initiated the transaction. In this example the process that initiated the transaction is the client (process A). When a subordinate service has completed sending replies (that is, when no more replies are outstanding), it must call TPRETURN.
4. The client (process A) determines whether all subordinate services have returned successfully.
   - If so, the client commits the changes made by those services, by calling TPCOMMIT, and completes the transaction.
   - If not, the client calls TPABORT, since it knows that TPCOMMIT could not be successful.

Example: Testing for Participant Errors

In the following sample code, a client makes a synchronous call to the fictitious REPORT service (line 24). Then the code checks for participant failures by testing for errors that can be returned on a communication call (lines 30-42).

Listing 9-5 Testing for Participant Success or Failure

```
01   . . .
02   CALL "TPINITIALIZE" USING TPINFDEF-REC
03   USR-DATA-REC
04   TPSTATUS-REC.
05   IF  NOT TPOK
06       error message,
07       EXIT PROGRAM .
08   MOVE 30 TO T-OUT.
09   CALL "TPBEGIN" USING TPTRXDEF-REC TPSTATUS-REC.
10   IF  NOT TPOK
11       error message,
```

```
12       PERFORM DO-TPTERM.
13  * Set up record
14   MOVE "REPORT=accrcv DBNAME=accounts" TP-RECORD.
15   MOVE 27 TO LEN.
16   MOVE "REPORTS" TO SERVICE-NAME.
17   MOVE "STRING" TO REC-TYPE.
18   SET TPBLOCK TO TRUE.
19   SET TPTRAN IN TPSVCDEF-REC TO TRUE.
20   SET TPNOTIME TO TRUE.
21   SET TPSIGRSTRT TO TRUE.
22 SET TPCHANGE TO TRUE.
23  *
24   CALL "TPCALL" USING TPSVCDEF-REC
25                 TPTYPE-REC
26                 TP-RECORD
27                 TPTYPE-REC
28                 TP-RECORD
29                 TPSTATUS-REC.
30   IF  TPOK
31        PERFORM DO-TPCOMMIT
32        PERFORM DO-TPTERM.
33  * Check return status
34    IF  TPESVCERR
35    DISPLAY "REPORT service's TPRETURN encountered problems"
36    ELSE IF  TPESVCFAIL
37    DISPLAY "REPORT service FAILED with return code=" APPL-RETURN-CODE
38    ELSE IF  TPEOTYPE
39    DISPLAY "REPORT service's reply is not of any known REC-TYPE"
40  *
41    PERFORM DO-TPABORT
42    PERFORM DO-TPTERM.
43  * Commit global transaction
44   DO-TPCOMMIT.
45    CALL "TPCOMMIT" USING TPTRXDEF-REC
46                    TPSTATUS-REC
47    IF  NOT TPOK
48        error message
49  * Abort the transaction
50   DO-TPABORT.
51    CALL "TPABORT" USING TPTRXDEF-REC
52                   TPSTATUS-REC
53    IF  NOT TPOK
54        error message
55  * Leave the application
56   DO-TPTERM.
57    CALL "TPTERM" USING TPSTATUS-REC.
58    IF  NOT TPOK
59        error message
60    EXIT PROGRAM.
```

Implicitly Defining a Global Transaction

An application can start a global transaction in either of two ways:

- Explicitly, by calling ATMI calls, as described in "Starting the Transaction" on page 9-2.
- Implicitly, from within a service routine

This section describes the second method.

You can implicitly place a service routine in transaction mode by setting the system parameter AUTOTRAN in the configuration file. If you set AUTOTRAN to Y, the system automatically starts a transaction in the service subroutine when a request is received from another process.

When implicitly defining a transaction, observe the following rules:

- If a process requests a service from another process when the calling process is *not* in transaction mode and the AUTOTRAN system parameter is set to start a transaction, the system initiates a transaction.
- If a process that is already in transaction mode requests a service from another process, the system's first response is to determine whether or not the caller is set to TPNOTRAN.

  If not set to TPNOTRAN, then the system places the called process in transaction mode through the "rule of propagation." The system does not check the AUTOTRAN parameter.

  If TPTRN-FLAG IN TPSVCDEF-REC is set to TPNOTRAN, the services performed by the called process are not included in the current transaction (that is, the propagation rule is suppressed). The system checks the AUTOTRAN parameter.

  - If AUTOTRAN is set to N (or if it is not set), the system does not place the called process in transaction mode.
  - If AUTOTRAN is set to Y, the system places the called process in transaction mode, but treats it as a new transaction.

Note: Because a service can be placed in transaction mode automatically, it is possible for a service with the TPNOTRAN flag set to call services that have the AUTOTRAN parameter set. If such a service requests another service, the member of the service information structure returns TPTRAN when queried. For example, if the call is made with TPNOTRAN | TPNOREPLY, and the service automatically starts a transaction when called, the information structure is set to TPTRAN | TPNOREPLY.

Defining Global Transactions for an XA-Compliant Server Group

Generally, the application programmer writes a service that is part of an XA-compliant server group to perform some operation via the group's resource manager. In the normal case, the service expects to perform all operations within a transaction. If, on the other hand, the service is called with the communication setting of TPNOTRAN, you may receive unexpected results when executing database operations.

In order to avoid unexpected behavior, design the application so that services in groups associated with XA-compliant resource managers are always called in transaction mode or are always defined in the configuration file with AUTOTRAN set to Y. You should also test the transaction level in the service code early.

Testing Whether a Transaction Has Started

It is important to know whether or not a process is in transaction mode in order to avoid and interpret certain error conditions. For example, it is an error for a process already in transaction mode to call TPBEGIN. When TPBEGIN is called by such a process, it fails and sets TP-STATUS to TPEPROTO to indicate that it was invoked while the caller was already participating in a transaction. The transaction is not affected.

You can design a service subroutine so that it tests whether it is in transaction mode before invoking TPBEGIN. You can test the transaction level by either of the following methods:

- Querying the settings of the service information structure that is passed to the service routine. The service is in transaction mode if the value is set to TPTRAN.
- Calling the TPGETLEV(3cbl) routine.

Use the following signature to call the TPGETLEV routine:

```
01 TPTRXLEV-REC.
   COPY TPTRXLEV.
01 TPSTATUS-REC.
   COPY TPSTATUS.
CALL "TPGETLEV" USING TPTRXLEV-REC TPSTATUS-REC.
```

TPGETLEV returns TP-NOT-IN-TRAN if the caller is not in a transaction and TP-IN-TRAN if the caller is.

The following code sample shows how to test for transaction level using the TPGETLEV routine (line 3). If the process is not already in transaction mode, the application starts a transaction (line 5). If TPBEGIN fails, a message is returned to the status line (line 9) and APPL-CODE IN TPSVCRET-REC of TPRETURN is set to a code that can be retrieved in APL-RETURN-CODE IN TPSTATUS-REC (lines 1 and 11).

Listing 9-6 Testing Transaction Level

```
     . . . Application defined codes
001       77 BEG-FAILED    PIC S9(9) VALUE 3.
     . . .
002  PROCEDURE DIVISION.
     . . .
003  CALL "TPGETLEV" USING TPTRCLEV-REC
                           TPSTATUS-REC.
004  IF NOT TPOK
        error processing  EXIT PROGRAM
005  IF TP-NOT-IN-TRAN
006          MOVE 30 TO T-OUT.
007          CALL "TPBEGIN" USING
                                TPTRXDEF-REC
                                TPSTATUS-REC.
008          IF  NOT TPOK
009               MOVE "Attempt to TPBEGIN within service failed"
                                TO USER-MESSAGE.
```

```
010              SET TPFAIL TO TRUE.
011              MOVE BEG-FAILED TO APPL-CODE.
012              COPY TPRETURN REPLACING
013                        DATA-REC BY USER-MESSAGE.
     . . .
```

If the AUTOTRAN parameter is set to Y, you do not need to call the TPBEGIN, and TPCOMMIT or TPABORT transaction routines explicitly. As a result, you can avoid the overhead of testing for transaction level. In addition, you can set the TRANTIME parameter to specify the time-out interval: the amount of time that may elapse after a transaction for a service begins, and before it is rolled back if not completed.

For example, suppose you are revising the OPEN_ACCT service shown in the preceding code listing. Currently, OPEN_ACCT defines the transaction explicitly and then tests for its existence. To reduce the overhead introduced by these tasks, you can eliminate them from the code. Therefore, you need to require that whenever OPEN_ACCT is called, it is called in transaction mode. To specify this requirement, enable the AUTOTRAN and TRANTIME system parameters in the configuration file.

See Also

- Description of the AUTOTRAN configuration parameter in the section "Implicitly Defining a Global Transaction" on page 9-16 of *Setting Up a BEA Tuxedo Application*.
- TRANTIME configuration parameter in *Setting Up a BEA Tuxedo Application*.

CHAPTER

10 Programming a Multithreaded and Multicontexted ATMI Application

This topic includes the following sections:

- Support for Programming a Multithreaded/Multicontexted ATMI Application
- Planning and Designing a Multithreaded/Multicontexted ATMI Application
- Implementing a Multithreaded/ Multicontexted ATMI Application
- Testing a Multithreaded/Multicontexted ATMI Application

Support for Programming a Multithreaded/Multicontexted ATMI Application

The BEA Tuxedo system supports only:

- Kernel-level threads packages (user-level threads packages are not supported)
- Multithreaded applications written in C (multithreaded COBOL applications are not supported)
- Multicontexted applications written in either C or COBOL

If your operating system supports POSIX threads functions as well as other types of threads functions, we recommend using the POSIX threads functions, which make your code easier to port to other platforms later.

To find out whether your platform supports a kernel-level threads package, C functions, or POSIX functions, see the data sheet for your operating system in *Installing the BEA Tuxedo System*.

Platform-specific Considerations for Multithreaded/Multicontexted Applications

Many platforms have idiosyncratic requirements for multithreaded and multicontexted applications. *Installing the BEA Tuxedo System* lists these platform-specific requirements. To find out what is needed on your platform, check the appropriate data sheet.

See Also

- "What Are Multithreading and Multicontexting?" on page 10-4
- "Advantages and Disadvantages of a Multithreaded/Multicontexted ATMI Application" on page 10-8
- "How Multithreading and Multicontexting Work in a Client" on page 10-11
- "How Multithreading and Multicontexting Work in an ATMI Server" on page 10-17

Planning and Designing a Multithreaded/Multicontexted ATMI Application

This topic includes the following sections:

- What Are Multithreading and Multicontexting?
- Advantages and Disadvantages of a Multithreaded/Multicontexted ATMI Application
- How Multithreading and Multicontexting Work in a Client
- How Multithreading and Multicontexting Work in an ATMI Server
- Design Considerations for a Multithreaded and Multicontexted ATMI Application

What Are Multithreading and Multicontexting?

The BEA Tuxedo system allows you to use a single process to perform multiple tasks simultaneously. The programming techniques for implementing this sort of process usage are *multithreading* and *multicontexting*. This topic provides basic information about these techniques:

- What Is Multithreading?
- What Is Multicontexting?

What Is Multithreading?

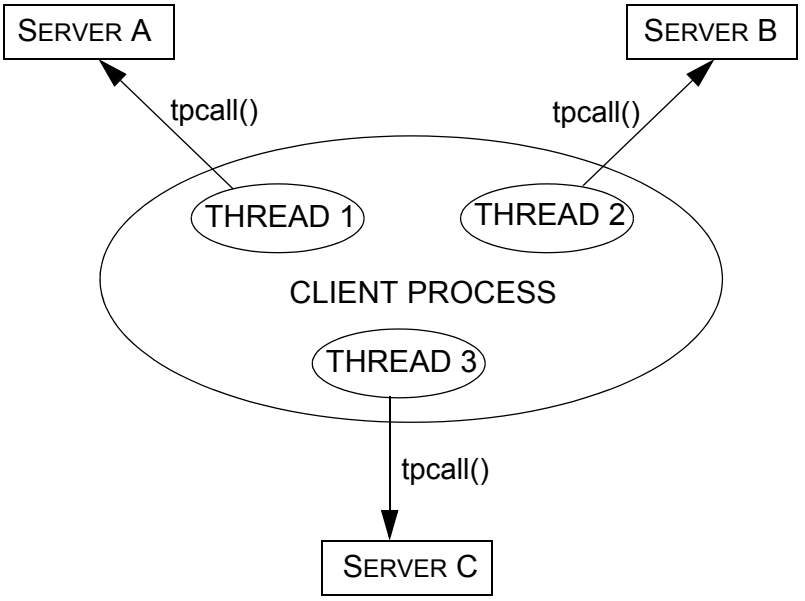
Multithreading is the inclusion of more than one unit of execution in a single process. In a multithreaded application, multiple simultaneous calls can be made from the same process. For example, an individual process is not limited to one outstanding `tpcall(3c)`.

In a server, multithreading requires multicontexting except when application-created threads are used in a singled-context server. The only way to create a multithreaded, single-context application is to use application-created threads.

The BEA Tuxedo system supports multithreaded applications written in C. It does not support multithreaded COBOL applications.

The following diagram shows how a multithreaded client can issue calls to three servers simultaneously.

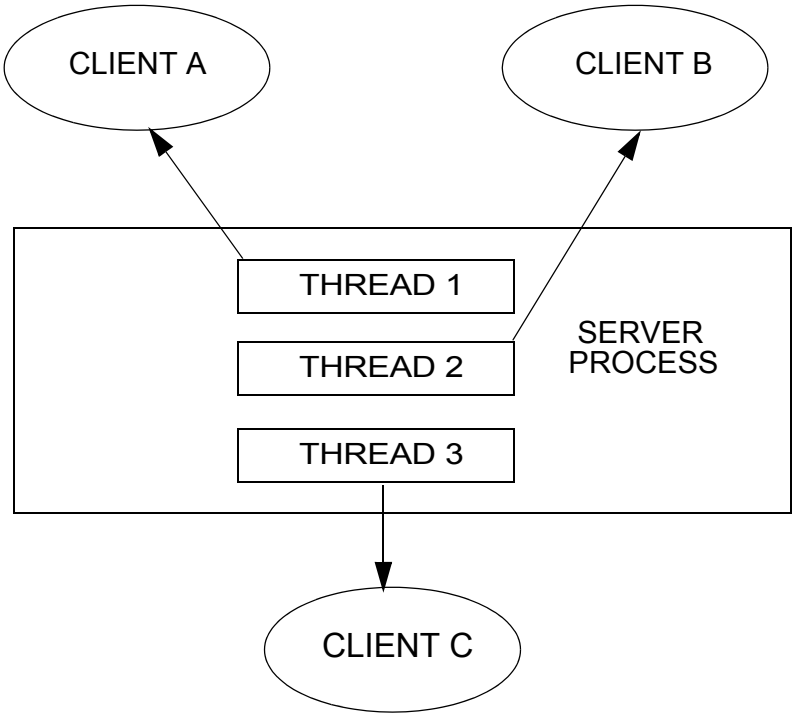
Figure 10-1 Sample Multithreaded Process



In a multithreaded application, multiple service-dispatched threads are available in the same server, which means that fewer servers need to be started for that application.

The following diagram shows how a server process can dispatch multiple threads to different clients simultaneously.

Figure 10-2 Multiple Service Threads Dispatched in One Server Process



What Is Multicontexting?

A context is an association to a domain. Multicontexting is the ability of a single process to have one of the following:

- More than one connection within a domain
- Connections to more than one domain

Multicontexting can be used in both clients and servers. When used in servers, multicontexting implies the use of multithreading, as well.

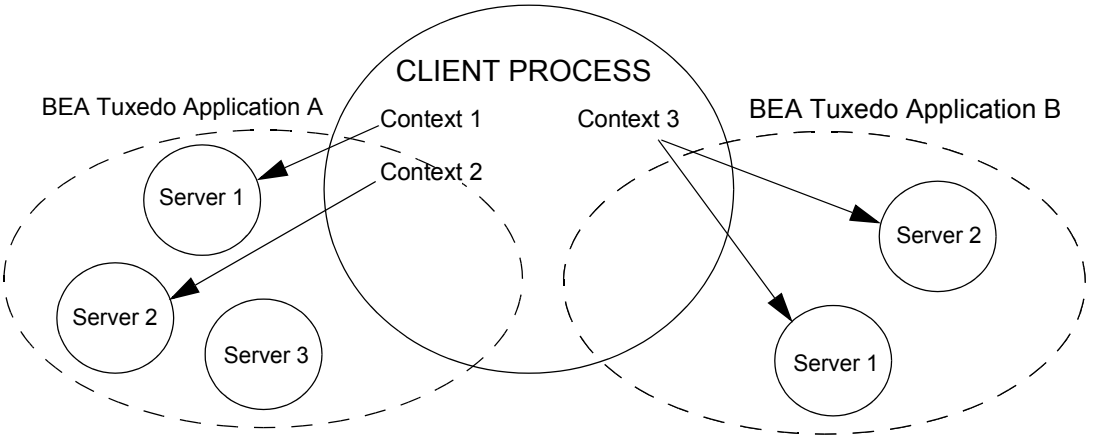
For a more complete list of the characteristics of a context, see "Context Attributes" in one of the following sections:

- "Writing Code to Enable Multicontexting in an ATMI Client" on page 10-31
- "Writing Code to Enable Multicontexting and Multithreading in an ATMI Server" on page 10-40

The BEA Tuxedo system supports multicontexted applications written in either C or COBOL. Multithreaded applications, however, are supported only in C.

The following diagram shows how a multicontexted client process works within a domain. Each arrow represents an outstanding call to a server.

Figure 10-3 Multicontexted Process in Two Domains



Licensing a Multithreaded or Multicontexted Application

For licensing purposes, each context is counted as one user. Additional licenses are not required to accommodate multiple threads within one context. For example:

- If a process has two contexts associated with Application A and one with Application B, the BEA Tuxedo system counts a total of three users (two in Application A and one in Application B).
- If a process has multiple threads accessing one application within the same context, the system counts only one user.

See Also

- "Advantages and Disadvantages of a Multithreaded/Multicontexted ATMI Application" on page 10-8
- "How Multithreading and Multicontexting Work in a Client" on page 10-11
- "How Multithreading and Multicontexting Work in an ATMI Server" on page 10-17

Advantages and Disadvantages of a Multithreaded/Multicontexted ATMI Application

Multithreading and multicontexting are powerful tools for enhancing the performance of BEA Tuxedo applications—given the appropriate circumstances. Before embarking on a plan to use these techniques, however, it is important to understand potential benefits and pitfalls.

Advantages of a Multithreaded/Multicontexted ATMI Application

Multithreaded and multicontexted ATMI applications offer the following advantages:

- Improved performance and concurrency

  For certain applications, performance and concurrency can be improved by using multithreading and multicontexting together. In other applications, performance can be unaffected or even degraded by using multithreading and multicontexting together. How performance is affected depends on your application.

- Simplified coding of remote procedure calls and conversations

  In some applications it is easier to code different remote procedure calls and conversations in separate threads than to manage them from the same thread.

- Simultaneous access to multiple applications

  Your BEA Tuxedo clients can be connected to more than one application at a time.

- Reduced number of required servers

  Because one server can dispatch multiple service threads, the number of servers to start for your application is reduced. This capability for multiple dispatched threads is especially useful for conversational servers, which otherwise must be dedicated to one client for the entire duration of a conversation.

For applications in which client threads are created by the Microsoft Internet Information Server API or the Netscape Enterprise Server interface (that is, the NSAPI), the use of multiple threads is essential if you want to obtain the full benefits afforded by these tools. This may be true of other tools, as well.

Disadvantages of a Multithreaded/Multicontexted ATMI Application

Multithreaded and multicontexted ATMI applications present the following disadvantages:

- Difficulty of writing code

  Multithreaded and multicontexted applications are not easy to write. Only experienced programmers should undertake coding for these types of applications.

- Difficulty of debugging

  It is much harder to replicate an error in a multithreaded or multicontexted application than it is to do so in a single-threaded, single-contexted application. As a result, it is more difficult, in the former case, to identify and verify root causes when errors occur.

- Difficulty of managing concurrency

  The task of managing concurrency among threads is difficult and has the potential to introduce new problems into an application.

- Difficulty of testing

  Testing a multithreaded application is more difficult than testing a single-threaded application because defects are often timing-related and more difficult to reproduce.

- Difficulty of porting existing code

  Existing code often requires significant re-architecting to take advantage of multithreading and multicontexting. Programmers need to:

  - Remove static variables
  - Replace any function calls that are not thread-safe
  - Replace any other code that is not thread-safe

  Because the completed port must be tested and retested, the work required to port a multithreaded and/or multicontexted application is substantial.

See Also

- "What Are Multithreading and Multicontexting?" on page 10-4
- "How Multithreading and Multicontexting Work in a Client" on page 10-11
- "How Multithreading and Multicontexting Work in an ATMI Server" on page 10-17
- "Design Considerations for a Multithreaded and Multicontexted ATMI Application" on page 10-22

How Multithreading and Multicontexting Work in a Client

When a multithreaded and multicontexted application is active, the life cycle of a client can be described in three phases:

- Start-up Phase
- Work Phase
- Completion Phase

Start-up Phase

In the start-up phase the following events occur:

- Some client threads join one or more BEA Tuxedo applications by calling `tpinit(3c)`.
- Other client threads share the contexts created by the first set of threads by calling `tpsetctxt(3c)`.
- Some client threads join multiple contexts.

- Some client threads switch to an existing context.

Note: There may also be threads that work independently of the BEA Tuxedo system. We do not consider such threads in this documentation.

Client Threads Join Multiple Contexts

A client in a BEA Tuxedo multicontexted application can have more than one application association as long as the following rules are observed:

- All associations must be made to the same installation of the BEA Tuxedo system.
- All application associations must be made from the same type of client. In other words, one of the following must be true:
  - All application associations must be made from native clients only.
  - All application associations must be made from Workstation clients only.

To join multiple contexts, clients call the `tpinit(3c)` function with the `TPMULTICONTEXTS` flag set in the *`flags`* element of the `TPINFO` data type.

When `tpinit()` is called with the `TPMULTICONTEXTS` flag set, a new application association is created and is designated the current association for the thread. The BEA Tuxedo domain to which the new association is made is determined by the value of the `TUXCONFIG` or `WSENVFILE/WSNADDR` environment variable.

Client Threads Switch to an Existing Context

Many ATMI functions operate on a per-context basis. (For a complete list, see “Using Per-context Functions and Data Structures in a Multithreaded ATMI Client” on page 10-52.) In such cases, the target context must be the current context. Although clients can join more than one context, at any time, in any thread, only one context can be the current context.

As task priorities shift within an application, requiring interactions with one BEA Tuxedo domain rather than another, it is sometimes advantageous to reassign a thread from one context to another.

In such situations, one client threads calls `tpgetctxt(3c)` and passes the handle that is returned (the value of which is the current context) to a second client thread. The second thread then associates itself with the current context by calling `tpsetctxt(3c)` and specifying the handle it received from `tpgetctxt(3c)` via the first thread.

Once the second thread is associated with the desired context, it is available to perform tasks executed by ATMI functions that operate on a per-context basis. For details, see "Using Per-context Functions and Data Structures in a Multithreaded ATMI Client" on page 10-52.

Work Phase

In this phase each thread performs a task. The following is a list of sample tasks:

- A thread issues a request for a service.
- A thread gets the reply to a service request.
- A thread initiates and/or participates in a conversation.
- A thread begins, commits, or rolls back a transaction.

Service Requests

A thread sends a request to a server by calling either `tpcall(3c)` for a synchronous request or `tpacall(3c)` for an asynchronous request. If the request is sent with `tpcall()`, then the reply is received without further action by any thread.

Replies to Service Requests

If an asynchronous request for a service has been sent with `tpcall(3c)`, a thread in the same context (which may or may not be the same thread that sent the request) gets the reply by calling `tpgetrply(3c)`.

Transactions

If one thread starts a transaction, then all threads that share the context of that thread also share the transaction.

Many threads in a context may work on a transaction, but only one thread may commit or abort it. The thread that commits or aborts the transaction can be any thread working on the transaction; it is not necessarily the same thread that started the transaction. Threaded applications are responsible for providing appropriate synchronization so that the normal rules of transactions are followed. (For example, there can be no outstanding RPC calls or conversations when a transaction is committed, and no stray calls are allowed after a transaction has been committed or aborted.) A process may be part of at most one transaction for each of its application associations.

If one thread of an application calls `tpcommit(3c)` concurrently with an RPC or conversational call in another thread of the application, the system acts as if the calls were issued in some serial order. An application context may temporarily suspend work on a transaction by calling `tpsuspend(3c)` and then start another transaction subject to the same restrictions that exist for single-threaded and single-context programs.

Unsolicited Messages

For each context in a multithreaded or multicontexted application, you may choose one of three methods for handling unsolicited messages.

A context may . . .	**By setting . . .**
Ignore unsolicited messages	`TPU_IGN`
Use dip-in notification	`TPU_DIP`
Use dedicated thread notification. (available only for C applications)	`TPU_THREAD`

The following caveats apply:

- SIGNAL-based notification is not allowed in multithreaded or multicontexted processes.

- If your application runs on a platform that supports multicontexting but not multithreading, then you cannot use the TPU_THREAD unsolicited notification method. As a result, you cannot receive immediate notification of events.

  If receiving immediate notification of events is important to your application, then you should carefully consider whether to use a multicontexted approach on this platform.

- Dedicated thread notification is available only:
  - For applications written in C
  - On multithreaded platforms supported by the BEA Tuxedo system

When dedicated thread notification is chosen, the system dedicates a separate thread to receive unsolicited messages and dispatch the unsolicited message handler. Only one copy of the unsolicited message handler can run at any one time in a given context.

If tpinit(3c) is called on a platform for which the BEA Tuxedo system does not support threads, with parameters indicating that TPU_THREAD notification is being requested on a platform that does not support threads, tpinit() returns -1 and sets tperrno to TPEINVAL. If the UBBCONFIG(5) default NOTIFY option is set to THREAD but threads are not available on a particular machine, the default behavior for that machine is downgraded to DIPIN. The difference between these two behaviors allows an administrator to specify a default for all machines in a mixed configuration—a configuration that includes some machines that support threads and some that do not—but it does not allow a client to explicitly request a behavior that is not available on its machine.

If tpsetunsol(3c) is called from a thread that is not associated with a context, a per-process default unsolicited message handler for all new tpinit(3c) contexts created is established. A specific context may change the unsolicited message handler for that context by calling tpsetunsol() again when the context is active. The per-process default unsolicited message handler may be changed by again calling tpsetunsol() in a thread not currently associated with a context.

If a process has multiple associations with the same application, then each association is assigned a different CLIENTID so that it is possible to send an unsolicited message to a specific application association. If a process has multiple associations with the same application, then any tpbroadcast(3c) is sent separately to each of the application associations that meet the broadcast criteria. When performing a dip-in check for receiving unsolicited messages, an application checks for only those messages sent to the current application association.

In addition to the ATMI functions permitted in unsolicited message handlers, it is permissible to call `tpgetctxt(3c)` within an unsolicited message handler. This functionality allows an unsolicited message handler to create another thread to perform any more substantial ATMI work required within the same context.

Userlog Maintains Thread-specific Information

For each thread in each application, `userlog(3c)` records the following identifying information:

`process_ID.thread_ID.context_ID`

Placeholders are printed in the *`thread_ID`* and *`context_ID`* fields of entries for non-threaded platforms and single-contexted applications.

The `TM_MIB(5)` supports this functionality in the `TA_THREADID` and `TA_CONTEXTID` fields in the `T_ULOG` class.

Completion Phase

In this phase, when the client process is about to exit, on behalf of the current context and all associated threads, a thread ends its application association by calling `tpterm(3c)`. Like other ATMI functions, `tpterm()` operates on the current context. It affects all threads for which the context is set to the terminated context, and terminates any commonality of context among these threads.

A well-designed application normally waits for all work in a particular context to complete before it calls `tpterm()`. Be sure that all threads are synchronized before your application calls `tpterm()`.

See Also

- "What Are Multithreading and Multicontexting?" on page 10-4
- "Design Considerations for a Multithreaded and Multicontexted ATMI Application" on page 10-22
- "Writing Code to Enable Multicontexting in an ATMI Client" on page 10-31
- "Writing a Multithreaded ATMI Client" on page 10-45
- "Synchronizing Threads Before an ATMI Client Termination" on page 10-34

How Multithreading and Multicontexting Work in an ATMI Server

The events that occur in an ATMI server when a multithreaded and multicontexted application is active can be described in three phases:

- Start-up Phase
- Work Phase
- Completion Phase

Start-up Phase

What happens during the start-up phase depends on the value of the `MINDISPATCHTHREADS` and `MAXDISPATCHTHREADS` parameters in the configuration file.

If the value of `MINDISPATCHTHREADS` **is . . .**	**And the value of** `MAXDISPATCHTHREADS` **is . . .**	**Then . . .**
0	> 1	1. The BEA Tuxedo system creates a thread dispatcher. 2. The dispatcher calls `tpsvrinit(3c)` to join the application.
> 0	> 1	1. The BEA Tuxedo system creates a thread dispatcher. 2. The dispatcher calls `tpsvrinit(3c)` to join the application. 3. The BEA Tuxedo system creates additional threads for handling service requests, and a context for each new thread. 4. Each new system-created thread calls `tpsvrthrinit(3c)` to join the application.

Work Phase

In this phase, the following activities occur:

- Multiple client requests to one server are handled concurrently in multiple contexts. The system allocates a separate thread for each request.
- If necessary, additional threads (up to the number indicated by `MAXDISPATCHTHREADS`) are created.
- The system keeps statistics on server threads.

Server-dispatched Threads Are Used

In response to clients' requests for a service, the server dispatcher creates multiple threads (up to a configurable maximum) in one server that can be assigned to various client requests concurrently. A server cannot become a client by calling `tpinit(3c)`.

Each dispatched thread is associated with a separate context. This feature is useful in both conversational and RPC servers. It is especially useful for conversational servers which otherwise sit idle, waiting for the client side of a conversation while other conversational connections are waiting for service.

This functionality is controlled by the following parameters in the `SERVERS` section of the `UBBCONFIG(5)` file and the `TM_MIB(5)`.

UBBCONFIG Parameter	MIB Parameter	Default
`MINDISPATCHTHREADS`	`TA_MINDISPATCHTHREADS`	0
`MAXDISPATCHTHREADS`	`TA_MAXDISPATCHTHREADS`	1
`THREADSTACKSIZE`	`TA_THREADSTACKSIZE`	0 (representing the OS default)

- Each dispatched thread is created with the stack size specified by `THREADSTACKSIZE` (or `TA_THREADSTACKSIZE`). If this parameter is not specified or has a value of 0, the operating system default is used. On a few operating systems on which the default is too small to be used by the BEA Tuxedo system, a larger default is used.
- If the value of this parameter is not specified or is 0, or if the operating system does not support setting a `THREADSTACKSIZE`, then the operating system default is used.
- `MINDISPATCHTHREADS` (or `TA_MINDISPATCHTHREADS`) must be less than or equal to `MAXDISPATCHTHREADS` (or `TA_MAXDISPATCHTHREADS`).
- If `MAXDISPATCHTHREADS` (or `TA_MAXDISPATCHTHREADS`) is 1, then the dispatcher thread and the service function thread are the same thread.
- If `MAXDISPATCHTHREADS` (or `TA_MAXDISPATCHTHREADS`) is greater than 1, any separate thread used for dispatching other threads does not count toward the limit of dispatched threads.

- Initially, the system boots `MINDISPATCHTHREADS` (or `TA_MINDISPATCHTHREADS`) server threads.
- The system never boots more than `MAXDISPATCHTHREADS` (or `TA_MAXDISPATCHTHREADS`) server threads.

Application-created Threads Are Used

Using your operating system functions, you may create additional threads within an application server. Application-created threads may:

- Operate independently of the BEA Tuxedo system
- Operate in the same context as an existing server dispatch thread
- Perform work on behalf of server dispatch contexts

Some restrictions govern what you can do if you create threads in your application.

- Servers may not become clients by calling `tpinit(3c)`.
- Initially, application-created server threads are not associated with any server dispatch context. An application-created server thread may call `tpsetctxt(3c)` (and pass it a value returned by a previous call to `tpgetctxt(3c)` within a server-dispatched thread) to associate itself with that server-dispatched context.
- An application-created server thread cannot call `tpreturn(3c)` or `tpforward(3c)`. When an application-created server thread has finished its work, it must call `tpsetctxt(3c)` with the context set to `TPNULLCONTEXT` before the originally dispatched thread calls `tpreturn()`.

Bulletin Board Liaison Verifies Sanity of System Processes

The Bulletin Board Liaison (BBL) periodically checks servers. If a server is taking too long to execute a particular service request, the BBL kills that server. (If specified, the BBL then restarts the server.) If the BBL kills a multicontexted server, the other service calls that are currently being executed are also terminated as a result of the process being killed.

The BBL also sends a message to any process or thread that has been waiting longer than its timeout value to receive a message. The blocking message receive call then returns an error indicating a timeout.

System Keeps Statistics on Server Threads

For each server, the BEA Tuxedo system maintains statistics for the following information:

- Maximum number of server-dispatched threads allowed
- Number of server-dispatched threads currently in use (TA_CURDISPATCHTHREADS)
- High-water mark of concurrent server-dispatched threads since the server was booted (TA_HWDISPATCHTHREADS)
- Number of server-dispatched threads historically started (TA_NUMDISPATCHTHREADS)

Userlog Maintains Thread-specific Information

For each thread in each application, userlog(3c) records the following identifying information:

process_ID.thread_ID.context_ID

Placeholders are printed in the *thread_ID* and *context_ID* fields of entries for non-threaded platforms and single-contexted applications.

The TM_MIB(5) supports this functionality in the TA_THREADID and TA_CONTEXTID fields in the T_ULOG class.

Completion Phase

When the application is shut down, tpsvrthrdone(3c) and tpsvrdone(3c) are called to perform any termination processing that is necessary, such as closing a resource manager.

See Also

- "What Are Multithreading and Multicontexting?" on page 10-4
- "Design Considerations for a Multithreaded and Multicontexted ATMI Application" on page 10-22
- "Writing Code to Enable Multicontexting and Multithreading in an ATMI Server" on page 10-40
- "Writing a Multithreaded ATMI Server" on page 10-59

Design Considerations for a Multithreaded and Multicontexted ATMI Application

Multithreaded and multicontexted ATMI applications are appropriate for some BEA Tuxedo domains, but not all. To decide whether to create such applications, you should answer several basic questions about the following:

- Your development and run-time environments
- Design requirements for your application
- Type of threads model to use
- Interoperability restrictions for Workstation clients

Environment Requirements

When considering the development of multithreaded and/or multicontexted applications, examine the following aspects of your development and run-time environments:

- Do you have an experienced team of programmers capable of writing and debugging multithreaded and multicontexted programs that successfully manage concurrency and synchronization?
- Are the multithreading features of the BEA Tuxedo system supported on the platform on which you are developing your application? These features are supported only on platforms with an OS-provided threads package, providing an appropriate level of functionality.
- Do the resource managers (RMs) used by your servers support multithreading? If so, consider the following issues, as well:
  - Do you need to set any parameters required by your RM to enable multithreaded access by your servers? For example, if you use an Oracle database with a multithreaded application, you must set the `THREADS=true` parameter as part of the `OPENINFO` string passed to Oracle. By doing so, you make it possible for individual threads to operate as separate Oracle associations.
  - Does your RM support a mixed mode of operation? A mixed-mode operation is a form of access such that multiple threads in a process can map to one RM association while other threads in the same process simultaneously map to different RM associations. Within one process, for example, Threads A and B map to RM Association X, while Thread C maps to RM Association Y.

    Not all RMs support mixed-mode operation. Some require all threads in a given process to map to the same RM association. If you are designing an application that will make use of transactional RM access within application-created threads, make sure your RM supports mixed-mode operation.

Design Requirements

When designing a multithreaded and/or multicontexted application, you should consider the following design questions:

- Is the task performed by your application suitable for multithreading and/or multicontexting?
- Do you want to connect to more than one BEA Tuxedo application? How many connections to each target application do you want?
- What synchronization issues need to be addressed in your application?
- Will you need to port your application to another platform after you have put your initial application into production?

Is the Task of Your Application Suitable for Multithreading and/or Multicontexting?

The following table provides a list of questions to help you decide whether your application would be improved if it were multithreaded and/or multicontexted. This list is not comprehensive; your individual requirements will determine other factors that should be considered.

For additional suggestions, we recommend that you consult a multithreaded and/or multicontexted programming publication.

If the answer to this question . . .	Is YES, then you might consider using . . .
Does your client need to connect to more than one application without using the Domains feature?	Multicontexting.
Does your client perform the role of a multiplexer within your application? For example, have you designated one machine in your application the “surrogate” for 100 other machines?	Multicontexting.
Does your client use multicontexting?	Multithreading. By allocating one thread per context, you can simplify your code.

If the answer to this question . . .	Is YES, then you might consider using . . .
Does your client perform two or more tasks that can be executed independently for a long time such that the performance gains from concurrent execution outweigh the costs and complexities of threads synchronization?	Multithreading.
Do you want one server to process multiple concurrent requests?	Multithreading. Assign a value greater than 1 to `MAXDISPATCHTHREADS`. This value enables multiple clients, each in its own thread, for the server.
If your client or server had multiple threads, would it be necessary to synchronize them after each thread had performed only a little work?	Not using multithreading.

How Many Applications and Connections Do You Want?

Decide how many applications you want to access and the number of connections you want to make.

- If you want connections to more than one application, then we recommend one of the following:
  - A single-threaded, multicontexted application
  - A multithreaded, multicontexted application
- If you want more than one connection to an application, then we recommend a multithreaded, multicontexted application.
- If you want only one connection to one application, then we recommend one of the following:
  - Multithreaded, single-contexted clients
  - Single-threaded, single-contexted clients

  In both cases, multithreaded, multicontexted servers may be used.

What Synchronization Issues Need to Be Addressed?

This issue is an important one during the design phase. It is, however, beyond the scope of this documentation. Please refer to a publication about multithreaded and/or multicontexted programming.

Will You Need to Port Your Application?

If you may need to port your application in the future, you should keep in mind that different operating systems have different sets of functions. If you think you may want to port your application after completing the initial version of it on one platform, remember to consider the amount of staff time that will be needed to revise the code with a different set of functions.

Which Threads Model Is Best for You?

Various models for multithreaded programs are now being used, including the following:

- Boss/worker model
- Siblings model
- Workflow model

We do not discuss threads models in this documentation. We recommend that you research all available models and consider your design requirements carefully when choosing a programming model for your application.

Interoperability Restrictions for Workstation Clients

Interoperability between release 7.1 Workstation clients and applications based on pre-7.1 releases of the BEA Tuxedo system is supported in any of the following situations:

- The client is neither multithreaded nor multicontexted.
- The client is multicontexted.
- The client is multithreaded and each thread is in a different context.

A BEA Tuxedo Release 7.1 Workstation client with multiple threads in a single context cannot interoperate with a pre-7.1 release of the BEA Tuxedo system.

See Also

- "Advantages and Disadvantages of a Multithreaded/Multicontexted ATMI Application" on page 10-8
- "Preliminary Guidelines for Programming a Multithreaded/Multicontexted ATMI Application" on page 10-28

Implementing a Multithreaded/ Multicontexted ATMI Application

Preliminary Guidelines for Programming a Multithreaded/Multicontexted ATMI Application

Before you start coding, make sure you have fulfilled or thought about the following:

Prerequisites for a Multithreaded ATMI Application

Make sure your environment meets the following prerequisites before starting your development project.

- Your operating system must provide a suitable threads package supported by the BEA Tuxedo system.

  The BEA Tuxedo system does not supply tools for creating threads, but it supports various threads packages provided by different operating systems. To create and synchronize threads, you must use the functions native to your operating system. To find out which, if any, threads packages are supported by your operating system, see *Installing the BEA Tuxedo System*.

- If you are using multithreaded servers, the resource managers used by those servers must support threads.

General Multithreaded Programming Considerations

Only experienced programmers should write multithreaded programs. In particular, programmers should already be familiar with basic design issues specific to this task, such as:

- The need for concurrency control among multiple threads
- The need to avoid the use of static variables in most instances
- Potential problems that may arise from the use of signals in multithreaded programs

These are just a few of the issues, too numerous to list here, with which we assume any programmer undertaking the writing of a multithreaded program is already familiar. These issues are discussed in many commercially available books on the subject of multithreaded programming.

Concurrency Considerations

Multithreading enables different threads of an application to perform concurrent operations on the same conversation. We do not recommend this approach, but the BEA Tuxedo system does not forbid it. If different threads perform concurrent operations on the same conversation, the system acts as if the concurrent calls were issued in some arbitrary order.

When programming with multiple threads, you must manage the concurrency among them by using mutexes or other concurrency-control functions. Here are three examples of the need for concurrency control:

- When multithreaded threads are operating on the same context, the programmer must ensure that functions are being executed in the required serial order. For example, all RPC calls and conversations must be compiled before `tpcommit(3c)` can be called. If `tpcommit()` is called from a thread other than the thread from which all these RPC or conversational calls are made, some concurrency control is probably required in the application.
- Similarly, it is permissible to call `tpacall(3c)` in one thread and `tpgetrply(3c)` in another, but the application must either:
  - Ensure that `tpacall()` is called before `tpgetrply()`, or
  - Manage the consequences if `tpacall()` is not called before `tpgetrply()`
- Multiple threads may operate on the same conversation but application programmers must realize that if different threads issue `tpsend(3c)` at approximately the same time, the system acts as though these `tpsend()` calls have been issued in an arbitrary order.

  For most applications, the best strategy is to code all the operations for one conversation in one thread. The second best strategy is to serialize these operations using concurrency control.

See Also

- "Design Considerations for a Multithreaded and Multicontexted ATMI Application" on page 10-22
- "Writing Code to Enable Multicontexting in an ATMI Client" on page 10-31
- "Writing Code to Enable Multicontexting and Multithreading in an ATMI Server" on page 10-40
- "Writing a Multithreaded ATMI Client" on page 10-45
- "Writing a Multithreaded ATMI Server" on page 10-59

Writing Code to Enable Multicontexting in an ATMI Client

To enable multicontexting in a client, you must write code that:

- Sets up multicontexting at initialization time
- Implements security
- If multithreading is also being used, synchronizes threads
- Switches contexts
- Handles unsolicited messages for each context

If your application uses transactions, you should also keep in mind the consequences of multicontexting for transactions. For more information, see "Coding Rules for Transactions in a Multithreaded/Multicontexted ATMI Application" on page 10-39.

Note: The instructions and sample code provided in this section refer to the C library functions provided by the BEA Tuxedo system. Equivalent COBOL library functions are also available; for details, see the *BEA Tuxedo COBOL Function Reference*.

Context Attributes

When writing your code, keep in mind the following considerations about contexts:

- If an application-created server thread exits without changing context before the original dispatched thread exits, then `tpreturn(3c)` or `tpforward(3c)` fails. The execution of a thread exit does not automatically trigger a call to `tpsetctxt(3c)` to change the context to `TPNULLCONTEXT`.
- For all contexts in a process, the same buffer type switch must be used.
- As with any other type of data structure, a multithreaded application must properly make use of BEA Tuxedo buffers, that is, buffers should not be used concurrently in two calls when one of the following may be true:
  - Both calls may use the buffer
  - Both calls may free the buffer
  - One call may use the buffer and one call may free the buffer
- If you call `tpinit(3c)` more than once, either to join multiple applications or to make multiple connections to a single application, keep in mind that on each `tpinit()` you must accommodate whatever security mechanisms have been established.

Setting Up Multicontexting at Initialization

When a client is ready to join an application, specify `tpinit(3c)` with the `TPMULTICONTEXTS` flag set, as shown in the following sample code.

Listing 10-1 Sample Code for a Client Joining a Multicontexted Application

```
#include <stdio.h>
#include <atmi.h>

TPINIT   *   tpinitbuf;

main()
{
       tpinitbuf = tpalloc(TPINIT, NULL, TPINITNEED(0));

       tpinitbuf->flags = TPMULTICONTEXTS;
              .
              .
              .
       if (tpinit (tpinitbuf) == -1) {

              ERROR_PROCESSING_CODE
       }
                     .
                     .
                     .
}
```

A new application association is created and assigned to the BEA Tuxedo domain specified in the `TUXCONFIG` or `WSENVFILE/WSNADDR` environment variable.

Note: In any one process, either all calls to `tpinit(3c)` must include the `TPMULTICONTEXTS` flag or else no call to `tpinit()` may include this flag. The only exception to this rule is that if all of a client's application associations are terminated by successful calls to `tpterm(3c)`, then the process is restored to a state in which the inclusion of the `TPMULTICONTEXTS` flag in the next call to `tpinit()` is optional.

Implementing Security for a Multicontexted ATMI Client

Each application association in the same process requires a separate security validation. The nature of that validation depends on the type of security mechanisms used in your application. In a BEA Tuxedo application you might, for example, use a system-level password or an application password.

As the programmer of a multicontexted application, you are responsible for identifying the type of security used in your application and implementing it for each application association in a process.

Synchronizing Threads Before an ATMI Client Termination

When you are ready to disconnect a client from an application, invoke `tpterm(3c)`. Keep in mind, however, that in a multicontexted application `tpterm()` destroys the current context. All the threads operating on that context are affected. As the application programmer, you must carefully coordinate the use of multiple threads to make sure that `tpterm()` is not called unexpectedly.

It is important to avoid calling `tpterm(3c)` on a context while other threads are still working on that context. If such a call to `tpterm()` is made, the BEA Tuxedo system places the other threads that had been associated with that context in a special invalid context state. When in the invalid context state, most ATMI functions are disallowed. A thread may exit from the invalid context state by calling `tpsetctxt(3c)` or `tpterm()`. Most well designed applications never have to deal with the invalid context state.

Note: The BEA Tuxedo system does not support multithreading in COBOL applications.

Switching Contexts

The following is a summary of the coding steps that might be made by a client that calls services from two contexts.

1. Set the `TUXCONFIG` environment variable to the value required by `firstapp`.
2. Join the first application by calling `tpinit(3c)` with the `TPMULTICONTEXTS` flag set.
3. Obtain a handle to the current context by calling `tpgetctxt(3c)`.
4. Switch the value of the `TUXCONFIG` environment variable to the value required by the `secondapp` context, by calling `tuxputenv()`.
5. Join the second application by calling `tpinit(3c)` with the `TPMULTICONTEXTS` flag set.
6. Get a handle to the current context by calling `tpgetctxt(3c)`.
7. Beginning with the `firstapp` context, start toggling between contexts by calling `tpsetctxt(3c)`.
8. Call `firstapp` services.
9. Switch the client to the `secondapp` context (by calling `tpsetctxt(3c)`) and call `secondapp` services.
10. Switch the client to the `firstapp` context (by calling `tpsetctxt(3c)`) and call `firstapp` services.
11. Terminate the `firstapp` context by calling `tpterm(3c)`.
12. Switch the client to the `secondapp` context (by calling `tpsetctxt(3c)`) and call `secondapp` services.
13. Terminate the `secondapp` context by calling `tpterm(3c)`.

The following sample code provides an example of these steps.

Note: In order to simplify the sample, error checking code is not included.

Listing 10-2 Sample Code for Switching Contexts in a Client

```
#include <stdio.h>
#include "atmi.h"/* BEA Tuxedo header file */

#if defined(__STDC__) || defined(__cplusplus)
main(int argc, char *argv[])
#else
main(argc, argv)
int argc;
char *argv[];
#endif
{

   TPINIT * tpinitbuf;
   TPCONTEXT_T firstapp_contextID, secondapp_contextID;
   /* Assume that TUXCONFIG is initially set to /home/firstapp/TUXCONFIG*/
   /*
    * Attach to the BEA Tuxedo system in multicontext mode.
    */
   tpinitbuf=tpalloc(TPINIT, NULL, TPINITNEED(0));
   tpinitbuf->flags = TPMULTICONTEXTS;

   if (tpinit((TPINIT *) tpinitbuf) == -1) {
      (void) fprintf(stderr, "Tpinit failed\n");
      exit(1);
}

   /*
    * Obtain a handle to the current context.
    */
   tpgetctxt(&firstapp_contextID, 0);

   /*
    * Use tuxputenv to change the value of TUXCONFIG,
    * so we now tpinit to another application.
    */
   tuxputenv("TUXCONFIG=/home/second_app/TUXCONFIG");

   /*
    * tpinit to secondapp.
    */
   if (tpinit((TPINIT *) tpinitbuf) == -1) {
      (void) fprintf(stderr, "Tpinit failed\n");
      exit(1);
   }

   /*
```

```
    * Get a handle to the context of secondapp.
    */
   tpgetctxt(&secondapp_contextID, 0);

   /*
    * Now you can alternate between the two contexts
    * using tpsetctxt and the handles you obtained from
    * tpgetctxt. You begin with firstapp.
    */

   tpsetctxt(firstapp_contextID, 0);

   /*
    * You call services offered by firstapp and then switch
    * to secondapp.
    */

   tpsetctxt(secondapp_contextID, 0);

   /*
    * You call services offered by secondapp.
    * Then you switch back to firstapp.
    */

   tpsetctxt(firstapp_contextID, 0);

   /*
    * You call services offered by firstapp. When you have
    * finished, you terminate the context for firstapp.
    */

   tpterm();

   /*
    * Then you switch back to secondapp.
    */

   tpsetctxt(secondapp_contextID, 0);
   /*
    * You call services offered by secondapp. When you have
      finished, you terminate the context for secondapp and
      end your program.
    */

   tpterm();

   return(0);
}
```

Handling Unsolicited Messages

For each context in which you want to handle unsolicited messages, you must set up an unsolicited message handler or use the process handler default if you have set one up.

If `tpsetunsol(3c)` is called from a thread that is not associated with a context, a per-process default unsolicited message handler for all new `tpinit(3c)` contexts created is established. A specific context may change the unsolicited message handler for that context by calling `tpsetunsol()` again when the context is active. The per-process default unsolicited message handler may be changed by again calling `tpsetunsol()` in a thread not currently associated with a context.

Set up the handler in the same way you set one up for a single-threaded or single-contexted application. See `tpsetunsol(3c)` for details.

You can use `tpgetctxt(3c)` in an unsolicited message handler if you want to identify the context in which you are currently working.

Coding Rules for Transactions in a Multithreaded/Multicontexted ATMI Application

The following consequences of using transactions should be kept in mind while you are writing your application:

- You can have only one transaction in any one context.
- You can have a different transaction for each context.
- All the threads associated with a given context at a given time share the same transaction state (if any) of that context.
- You must synchronize your threads so all conversations and RPC calls are complete before you call `tpcommit(3c)`.
- You can call `tpcommit(3c)` from only one thread in any particular transaction.

See Also

- "How Multithreading and Multicontexting Work in a Client" on page 10-11
- "Writing a Multithreaded ATMI Client" on page 10-45

Writing Code to Enable Multicontexting and Multithreading in an ATMI Server

This topic includes the following sections:

- Coding Rules for a Multicontexted ATMI Server
- Initializing and Terminating ATMI Servers and Server Threads
- Programming an ATMI Server to Create Threads
- Sample Code for Creating an Application Thread in a Multicontexted ATMI Server

Note: The instructions and sample code provided in this section refer to the C library functions provided by the BEA Tuxedo system. (See the *BEA Tuxedo C Function Reference* for details.) Equivalent COBOL routines are not available because multithreading (which is required to create a multicontexted server) is not supported for COBOL applications.

Context Attributes

When writing your code, keep in mind the following considerations about contexts:

- If an application-created server thread exits without changing context before the original dispatched thread exits, then `tpreturn(3c)` or `tpforward(3c)` fails. The execution of a thread exit does not automatically trigger a call to `tpsetctxt(3c)` to change the context to `TPNULLCONTEXT`.
- For all contexts in a process, the same buffer type switch must be used.
- As with any other type of data structure, a multithreaded application must properly make use of BEA Tuxedo buffers, that is, buffers should not be used concurrently in two calls when one of the following may be true:
  - Both calls may use the buffer.
  - Both calls may free the buffer.

- One call may use the buffer and one call may free the buffer.

Coding Rules for a Multicontexted ATMI Server

Keep in mind the following rules for coding multicontexted servers:

- The BEA Tuxedo dispatcher on the server may dispatch the same service and/or different services multiple times, creating a different dispatch context for each service dispatched.
- A server is prohibited from calling `tpinit(3c)` or otherwise acting as a client. If a server process calls `tpinit()`, `tpinit()` returns `-1` and sets `tperrno(5)` to `TPEPROTO`. An application-created server thread may not make ATMI calls before calling `tpsetctxt(3c)`.
- Only a server-dispatched thread may call `tpreturn(3c)` or `tpforward(3c)`.
- A server cannot execute a `tpreturn(3c)` or `tpforward(3c)` if any application-created thread is still associated with any application context. Therefore, before a server-dispatched thread calls `tpreturn()`, each application-created thread associated with that context must call `tpsetctxt(3c)` with the context set to either `TPNULLCONTEXT` or another valid context.

  If this rule is violated, then `tpreturn(3c)` or `tpforward(3c)` writes a message to the user log, indicates `TPESVCERR` to the caller, and returns control to the main server dispatch loop. The threads that had been in the context where the invalid `tpreturn()` was done are placed in an invalid context.
- If there are outstanding ATMI calls, RPC calls, or conversations when `tpreturn(3c)` or `tpforward(3c)` is called, `tpreturn()` or `tpforward()` writes a message to the user log, indicates `TPESVCERR` to the caller, and returns control to the main server dispatch loop.
- A server-dispatched thread may not call `tpsetctxt(3c)`.
- Unlike single-contexted servers, it is permissible for a multicontexted server thread to call a service that is offered only by that same server process.

Initializing and Terminating ATMI Servers and Server Threads

To initialize and terminate your servers and server threads, you can use the default functions provided by the BEA Tuxedo system or you can use your own.

Table 10-1 Default Functions for Initialization and Termination

To . . .	Use the default function
Initialize a server	`tpsvrinit(3c)`
Initialize a server thread	`tpsvrthrinit(3c)`
Terminate a server	`tpsvrdone(3c)`
Terminate a server thread	`tpsvrthrdone(3c)`

Programming an ATMI Server to Create Threads

You may create additional threads within an application server, although most applications using multicontexted servers use only the dispatched server threads created by the system. This section provides instructions for doing so.

Creating Threads

You may create additional threads within an application server by using OS threads functions. These new threads may operate independently of the BEA Tuxedo system, or they may operate in the same context as one of the server-dispatched threads.

Associating Threads with a Context

Initially, application-created server threads are not associated with any server-dispatched context. If called before being initialized, however, most ATMI functions perform an implicit `tpinit(3c)`. Such calls introduce problems because servers are prohibited from calling `tpinit()`. (If a server process calls `tpinit()`, `tpinit()` returns -1 and sets `tperrno(5)` to `TPEPROTO`.)

Therefore, an application-created server thread must associate itself with an existing context before calling any ATMI functions. To associate an application-created server thread with an existing context, you must write code that implements the following procedure.

1. Server-dispatched-thread_A gets a handle to the current context by calling `tpgetctxt(3c)`.
2. Server-dispatched-thread_A passes the handle returned by `tpgetctxt(3c)` to Application_thread_B.
3. Application_thread_B associates itself with the current context by calling `tpsetctxt(3c)`, specifying the handle received from Server-dispatched-thread_A.
4. Application-created server threads cannot call `tpreturn(3c)` or `tpforward(3c)`. Before the originally dispatched thread calls `tpreturn()` or `tpforward()`, all application-created server threads that have been in that context must switch to `TPNULLCONTEXT` or another valid context.

   If this rule is not observed, then `tpforward(3c)` or `tpreturn(3c)` fails and indicates a service error to the caller.

Sample Code for Creating an Application Thread in a Multicontexted ATMI Server

For those applications with a need to create an application thread in a server, the following code sample shows a multicontexted server in which a service creates another thread to help perform its work. Operating system (OS) threads functions differ from one OS to another. In this sample POSIX and ATMI functions are used.

Notes: In order to simplify the sample, error checking code is not included. Also, an example of a multicontexted server using only threads dispatched by the BEA Tuxedo system is not included because such a server is coded in exactly the same way as a single-contexted server, as long as thread-safe programming practices are used.

Listing 10-3 Code Sample for Creating a Thread in a Multicontexted Server

```
#include <pthread.h>
#include <atmi.h>

void *withdrawalthread(void *);

struct sdata {
             TPCONTEXT_T   ctxt;
             TPSVCINFO     *svcinfoptr;
};

void
TRANSFER(TPSVCINFO *svcinfo)
{
    struct sdata     transferdata;
    pthread_t        withdrawalthreadid;

    tpgetctxt(&transferdata.ctxt, 0);
    transferdata.svcinfoptr = svcinfo;
    pthread_create(&withdrawalthreadid, NULL, withdrawalthread, &transferdata);
    tpcall("DEPOSIT", ...);
    pthread_join(withdrawalthreadid, NULL);
    tpreturn(TPSUCCESS, ...);
}


void *
withdrawalthread(void *arg)
{
    tpsetctxt(arg->ctxt, 0);
    tpopen();
    tpcall("WITHDRAWAL", ...);
    tpclose();
    return(NULL);
}
```

The previous example accomplishes a funds transfer by invoking the DEPOSIT service in the originally dispatched thread, and WITHDRAWAL in an application-created thread. This example is based on the assumption that the resource manager being used allows a mixed model such that multiple threads of a server can be associated with a particular database connection without all threads of the server being associated with that instance. Most resource managers, however, do not support such a model.

A simpler way to code this example is to avoid the use of an application-created thread. To obtain the same concurrency provided by the two calls to tpcall(3c) in the example, substitute two calls to tpacall(3c) and two calls to tpgetrply(3c) in the server-dispatched thread.

See Also

- "How Multithreading and Multicontexting Work in an ATMI Server" on page 10-17

Writing a Multithreaded ATMI Client

This topic includes the following sections:

- Coding Rules for a Multithreaded ATMI Client
- Initializing an ATMI Client to Multiple Contexts
- Getting Replies in a Multithreaded Environment
- Using Environment Variables in a Multithreaded and/or Multicontexted Environment
- Using Per-context Functions and Data Structures in a Multithreaded ATMI Client
- Using Per-process Functions and Data Structures in a Multithreaded ATMI Client
- Using Per-thread Functions and Data Structures in a Multithreaded ATMI Client

- Sample Code for a Multithreaded ATMI Client

Note: The BEA Tuxedo system does not support multithreaded COBOL applications.

Coding Rules for a Multithreaded ATMI Client

Keep in mind the following rules for coding multithreaded clients:

- Once a conversation has been started, any thread in the same process can work on that conversation. Handles and call descriptors are portable within the same context in the same process, but not between contexts or processes. Handles and call descriptors can be used only in the application context in which they are originally assigned.
- Any thread operating in the same context within the same process can invoke `tpgetrply(3c)` to receive a response to an earlier call to `tpacall(3c)`, regardless of whether or not that thread originally called `tpacall()`.
- A transaction can be committed or aborted by only one thread, which may or may not be the same thread that started it.
- All RPC calls and all conversations must be completed before an attempt is made to commit the transaction. If an application calls `tpcommit(3c)` while RPC calls or conversations are outstanding, `tpcommit()` aborts the transaction, returns `-1`, and sets `tperrno(5)` to `TPEABORT`.
- Functions such as `tpcall(3c)`, `tpacall(3c)`, `tpgetrply(3c)`, `tpconnect(3c)`, `tpsend(3c)`, `tprecv(3c)`, and `tpdiscon(3c)` should not be called in transaction mode unless you are sure that the transaction is not already committing or aborting.
- Two `tpbegin(3c)` calls cannot be made simultaneously for the same context.
- `tpbegin(3c)` cannot be issued for a context that is already in transaction mode.
- If you are using a client and you want to connect to more than one domain, you must manually change the value of `TUXCONFIG` or `WSNADDR` before calling `tpinit(3c)`. You must synchronize the setting of the environment variable and the `tpinit()` call if multiple threads may be performing such an action. All application associations in a client must obey the following rules:

- All associations must be made to the same release of the BEA Tuxedo system.
- Either every application association in a particular client must be made as a native client, or every application association must be made as a Workstation client.

- To join an application, a multithreaded Workstation client must always call `tpinit(3c)` with the `TPMULTICONTEXTS` flag set, even if the client is running in single-context mode.

Initializing an ATMI Client to Multiple Contexts

To have a client join more than one context, issue a call to the `tpinit(3c)` function with the `TPMULTICONTEXTS` flag set in the *`flags`* element of the `TPINIT` data structure.

In any one process, either all calls to `tpinit(3c)` must include the `TPMULTICONTEXTS` flag or no call to `tpinit()` may include this flag. The only exception to this rule is that if all of a client's application associations are terminated by successful calls to `tpterm(3c)`, then the process is restored to a state in which the inclusion of the `TPMULTICONTEXTS` flag in the next call to `tpinit()` is optional.

When `tpinit(3c)` is invoked with the `TPMULTICONTEXTS` flag set, a new application association is created and is designated the current association. The BEA Tuxedo domain to which the new association is made is determined by the value of the `TUXCONFIG` or `WSENVFILE/WSNADDR` environment variable.

When a client thread successfully executes `tpinit(3c)` without the `TPMULTICONTEXTS` flag, all threads in the client are placed in the single-context state (`TPSINGLECONTEXT`).

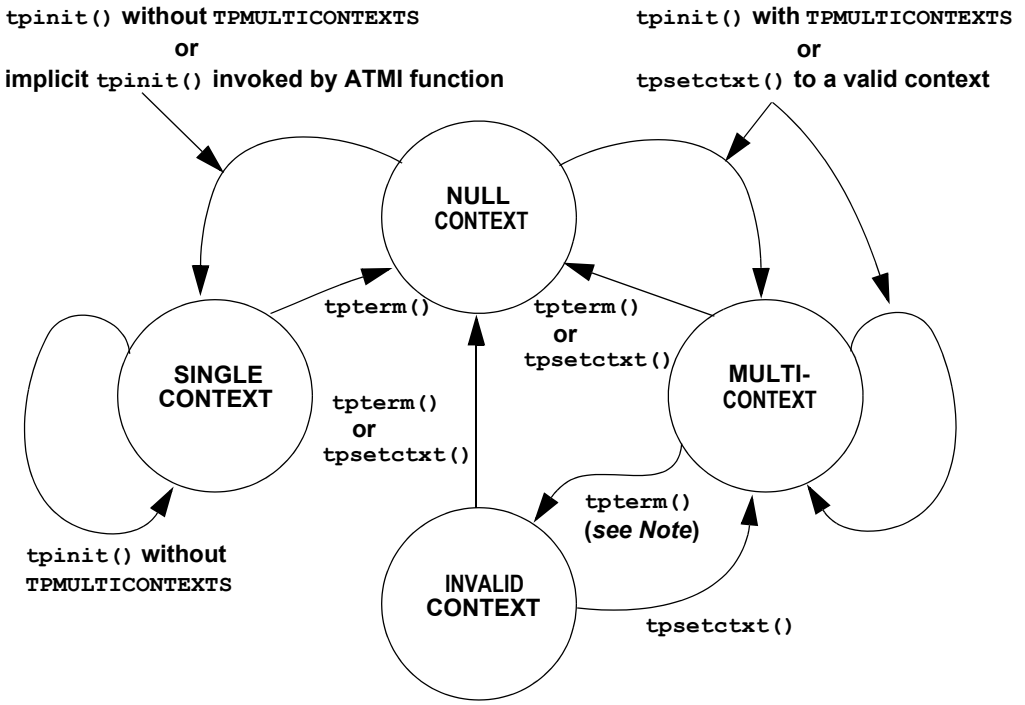
On failure, `tpinit(3c)` leaves the calling thread in its original context (that is, in the context state in which it was operating before the call to `tpinit()`).

Do not call `tpterm(3c)` from a given context if any of the threads in that context are still working. See the table labeled "Multicontext State Transitions" on page 10-48 for a description of the context states that result from calling `tpterm()` under these and other circumstances.

Context State Changes for an ATMI Client Thread

In a multicontext application, calls to various functions result in context state changes for the calling thread and any other threads that are active in the same context as the calling process. The following diagram illustrates the context state changes that result from calls to `tpinit(3c)`, `tpsetctxt(3c)`, and `tpterm(3c)`. (The `tpgetctxt(3c)` function does not produce any context state changes.)

Figure 10-4 Multicontext State Transitions



Note: When `tpterm(3c)` is called by a thread running in the multicontext state (`TPMULTICONTEXTS`), the calling thread is placed in the null context state (`TPNULLCONTEXT`). All other threads associated with the terminated context are switched to the invalid context state (`TPINVALIDCONTEXT`).

The following table lists all possible context state changes produced by calling `tpinit(3c)`, `tpsetctxt(3c)`, and `tpterm(3c)`.

Table 10-2 Context State Changes for a Client Thread

When this function is executed . . .	Then a thread in this context state results in . . .			
	Null Context	**Single Context**	**Multicontext**	**Invalid Context**
`tpinit(3c)` without `TPMULTICONTEXTS`	Single context	Single context	Error	Error
`tpinit(3c)` with `TPMULTICONTEXTS`	Multicontext	Error	Multicontext	Error
`tpsetctxt(3c)` to `TPNULLCONTEXT`	Null	Error	Null	Null
`tpsetctxt(3c)` to context 0	Error	Single context	Error	Error
`tpsetctxt(3c)` to context > 0	Multicontext	Error	Multicontext	Multicontext
Implicit `tpinit(3c)`	Single context	N/A	N/A	Error
`tpterm(3c)` in this thread	Null	Null	Null	Null
`tpterm(3c)` in a different thread of this context	N/A	Null	Invalid	N/A

Getting Replies in a Multithreaded Environment

`tpgetrply(3c)` receives responses only to requests made via `tpacall(3c)`. Requests made with `tpcall(3c)` are separate and cannot be retrieved with `tpgetrply()` regardless of the multithreading or multicontexting level.

tpgetrply(3c) operates in only one context, which is the context in which it is called. Therefore, when you call tpgetrply() with the TPGETANY flag, only handles generated in the same context are considered. Similarly, a handle generated in one context may not be used in another context, but the handle may be used in any thread operating within the same context.

When tpgetrply(3c) is called in a multithreaded environment, the following restrictions apply:

- If a thread calls tpgetrply(3c) for a specific handle while another thread in the same context is already waiting in tpgetrply() for the same handle, tpgetrply() returns -1 and sets tperrno to TPEPROTO.
- If a thread calls tpgetrply(3c) for a specific handle while another thread in the same context is already waiting in tpgetrply() with the TPGETANY flag, the call returns -1 and sets tperrno(5) to TPEPROTO.

  The same behavior occurs if a thread calls tpgetrply(3c) with the TPGETANY flag while another thread in the same context is already waiting in tpgetrply() for a specific handle. These restrictions protect a thread that is waiting on a specific handle from having its reply taken by a thread waiting on any handle.
- At any given time, only one thread in a particular context can wait in tpgetrply(3c) with the TPGETANY flag set. If a second thread in the same context invokes tpgetrply() with the TPGETANY flag while a similar call is outstanding, this second call returns -1 and sets tperrno(5) to TPEPROTO.

Using Environment Variables in a Multithreaded and/or Multicontexted Environment

When a BEA Tuxedo application is run in an environment that is multicontexted and/or multithreaded, the following considerations apply to the use of environment variables:

- A process initially inherits its environment from the operating system environment. On platforms that support environment variables, such variables make up a per-process entity. Therefore, applications that depend on per-context environment settings should use the tuxgetenv(3c) function instead of an OS function.

Note: The environment is initially empty for those operating systems that do not recognize an operating system environment.

- Many environment variables are read by the BEA Tuxedo system only once per process or once per context and then cached within the BEA Tuxedo system. Changes to such variables once cached in the process have no effect.

Caching is done on a . . .	For environment variables such as . . .
Per-context basis	TUXCONFIG
	FIELDTBLS and FIELDTBLS32
	FLDTBLDIR and FLDTBLDIR32
	ULOGPFX
	VIEWDIR and VIEWDIR32
	VIEWFILES and VIEWFILES32
	WSNADDR
	WSDEVICE
	WSENV
Per-process basis	TMTRACE
	TUXDIR
	ULOGDEBUG

- The `tuxputenv(3c)` function affects the environment for the entire process.
- When you call the `tuxreadenv(3c)` function, it reads a file containing environment variables and adds them to the environment for the entire process.
- The `tuxgetenv(3c)` function returns the current value of the requested environment variable in the current context. Initially, all contexts have the same environment, but the use of environment files specific to a particular context can cause different contexts to have different environment settings.
- If a client intends to initialize to more than one domain, the client must change the value of the `TUXCONFIG`, `WSNADDR`, or `WSENVFILE` environment variable to

the proper value before each call to `tpinit(3c)`. If such an application is multithreaded, a mutex or other application-defined concurrency control will probably be needed to ensure that:

- The appropriate environment variable is reset.
- The call to `tpinit(3c)` is made without the environment variable being reset by any other thread.

- When a client initializes to the system, the `WSENVFILE` and/or machine environment file is read and affects the environment in that context only. The previous environment for the process as a whole remains for that context to the extent that it is not overridden within the environment file(s).

Using Per-context Functions and Data Structures in a Multithreaded ATMI Client

The following ATMI functions affect only the application contexts in which they are called:

- `tpabort(3c)`
- `tpacall(3c)`
- `tpadmcall(3c)`
- `tpbegin(3c)`
- `tpbroadcast(3c)`
- `tpcall(3c)`
- `tpcancel(3c)`
- `tpchkauth(3c)`
- `tpchkunsol(3c)`
- `tpclose(3c)`
- `tpcommit(3c)`
- `tpconnect(3c)`
- `tpdequeue(3c)`
- `tpdiscon(3c)`

- tpenqueue(3c)
- tpforward(3c)
- tpgetlev(3c)
- tpgetrply(3c)
- tpinit(3c)
- tpnotify(3c)
- tpopen(3c)
- tppost(3c)
- tprecv(3c)
- tpresume(3c)
- tpreturn(3c)
- tpscmt(3c)
- tpsend(3c)
- tpservice(3c)
- tpsetunsol(3c)
- tpsubscribe(3c)
- tpsuspend(3c)
- tpterm(3c)
- tpsubscribe(3c)
- tx_begin(3c)
- tx_close(3c)
- tx_commit(3c)
- tx_info(3c)
- tx_open(3c)
- tx_rollback(3c)
- tx_set_commit_return(3c)
- tx_set_transaction_control(3c)
- tx_set_transaction_timeout(3c)
- userlog(3c)

Note: For `tpbroadcast(3c)`, the broadcast message is identified as having come from a particular application association. For `tpnotify(3c)`, the notification is identified as having come from a particular application association. See "Using Per-process Functions and Data Structures in a Multithreaded Client" for notes about `tpinit(3c)`.

If `tpsetunsol(3c)` is called from a thread that is not associated with a context, a per-process default unsolicited message handler for all new `tpinit(3c)` contexts created is established. A specific context may change the unsolicited message handler for that context by calling `tpsetunsol()` again when the context is active. The per-process default unsolicited message handler may be changed by again calling `tpsetunsol()` in a thread not currently associated with a context.

- The `CLIENTID`, client name, username, transaction ID, and the contents of the `TPSVCINFO` data structure may differ from context to context within the same process.
- Asynchronous call handles and connection descriptors are valid in the contexts in which they are created. The unsolicited notification type is specific per-context. Although signal-based notification may not be used with multiple contexts, each context may choose one of three options:
  - Ignoring unsolicited messages
  - Using dip-in notification
  - Using dedicated thread notification

Using Per-process Functions and Data Structures in a Multithreaded ATMI Client

The following BEA Tuxedo functions affect the entire process in which they are called:

- `tpadvertise(3c)`
- `tpalloc(3c)`
- `tpconvert(3c)`—the requested structure is converted, although it is probably relevant to only a subset of the process.
- `tpfree(3c)`
- `tpinit(3c)`—to the extent that the per-process `TPMULTICONTEXTS` mode or single-context mode is established. See also "Using Per-context Functions and Data Structures in a Multithreaded ATMI Client" on page 10-52.
- `tprealloc(3c)`
- `tpsvrdone(3c)`
- `tpsvrinit(3c)`
- `tptypes(3c)`
- `tpunadvertise(3c)`
- `tuxgetenv(3c)`—if the OS environment is per-process.
- `tuxputenv(3c)`—if the OS environment is per-process.
- `tuxreadenv(3c)`—if the OS environment is per-process.
- `Usignal(3c)`

The determination of single-context mode, multicontext mode, or uninitialized mode affects an entire process. The buffer type switch, the view cache, and environment variable values are also per-process functions.

Using Per-thread Functions and Data Structures in a Multithreaded ATMI Client

Only the calling thread is affected by the following:

- `CATCH`
- `tperrordetail(3c)`
- `tpgetctxt(3c)`
- `tpgprio(3c)`
- `tpsetctxt(3c)`
- `tpsprio(3c)`
- `tpstrerror(3c)`
- `tpstrerrordetail(3c)`
- `TRY(3c)`
- `Uunix_err(3c)`

The `Ferror, Ferror32(5)`, `tperrno(5)`, `tpurcode(5)`, and `Uunix_err` variables are specific to each thread.

The identity of the current context is specific to each thread.

Sample Code for a Multithreaded ATMI Client

The following example shows a multithreaded client using ATMI calls. Threads functions differ from one operating system to another. In this example, POSIX functions are used.

Note: In order to simplify this example, error checking code has not been included.

Listing 10-4 Sample Code for a Multithreaded Client

```
#include <stdio.h>
#include <pthread.h>
#include <atmi.h>
```

```
TPINIT   *   tpinitbuf;
int          timeout=60;
pthread_t    withdrawalthreadid, stockthreadid;
TPCONTEXT_T  ctxt;
void * stackthread(void *);
void * withdrawalthread(void *);

main()
{
tpinitbuf = tpalloc(TPINIT, NULL, TPINITNEED(0));
/*
 * This code will perform a transfer, using separate threads for the
 * withdrawal and deposit.   It will also get the current
 * price of BEA stock from a separate application, and calculate how
 * many shares the transferred amount can buy.
 */

tpinitbuf->flags = TPMULTICONTEXTS;

/* Fill in the rest of tpinitbuf.  */
tpinit(tpinitbuf);

tpgetctxt(&ctxt, 0);
tpbegin(timeout, 0);
pthread_create(&withdrawalthreadid, NULL, withdrawalthread, NULL);
tpcall("DEPOSIT", ...);

/* Wait for the withdrawal thread to complete. */
pthread_join(withdrawalthreadid, NULL);

tpcommit(0);
tpterm();

/* Wait for the stock thread to complete. */
pthread_join(stockthreadid, NULL);

/* Print the results. */
printf("$%9.2f has been transferred \
from your savings account to your checking account.\n", ...);

printf("At the current BEA stock price of $%8.3f, \
you could purchase %d shares.\n", ...);

exit(0);
}
```

```
void *
stockthread(void *arg)
{

     /* The other threads have now called tpinit(), so resetting TUXCONFIG can
      * no longer adversely affect them.
     */

     tuxputenv("TUXCONFIG=/home/users/xyz/stockconf");
     tpinitbuf->flags = TPMULTICONTEXTS;
     /* Fill in the rest of tpinitbuf. */
     tpinit(tpinitbuf);
     tpcall("GETSTOCKPRICE", ...);
     /* Save the stock price in a variable that can also be accessed in main(). */
     tpterm();
     return(NULL);
}



void *
withdrawalthread(void *arg)
{
/* Create a separate thread to get stock prices from a different
 * application.
*/



     pthread_create(&stockthreadid, NULL, stockthread, NULL);
     tpsetctxt(ctxt, 0);
     tpcall("WITHDRAWAL", ...);
     return(NULL);
}
```

See Also

- "How Multithreading and Multicontexting Work in a Client" on page 10-11
- "Preliminary Guidelines for Programming a Multithreaded/Multicontexted ATMI Application" on page 10-28
- "Writing Code to Enable Multicontexting in an ATMI Client" on page 10-31

Writing a Multithreaded ATMI Server

Multithreaded servers are almost always multicontexted, as well. For information about writing a multithreaded server, see “Writing Code to Enable Multicontexting and Multithreading in an ATMI Server” on page 10-40.

Compiling Code for a Multithreaded/Multicontexted ATMI Application

The programs provided by the BEA Tuxedo system for compiling or building executables, such as `buildserver(1)` and `buildclient(1)`, automatically include any required compiler flags. If you use these tools, then you do not need to set any flags at compile time.

If, however, you compile your `.c` files into `.o` files before doing a final compilation, you may need to set platform-specific compiler flags. Such flags must be set consistently for all code linked into a single process.

If you are creating a multithreaded server, you must run the `buildserver(1)` command with the `-t` option. This option is mandatory for multithreaded servers; if you do not specify it at build time and later try to boot the new server with a configuration file in which the value of `MAXDISPATCHTHREADS` is greater than 1, a warning message is recorded in the user log and the server reverts to single-threaded operation.

To identify any operating system-specific compiler parameters that are required when you compile `.c` files into `.o` files in a multithreaded environment, run `buildclient(1)` or `buildserver(1)` with the `-v` option set on a test file.

See Also

- "Writing Code to Enable Multicontexting in an ATMI Client" on page 10-31
- "Writing Code to Enable Multicontexting and Multithreading in an ATMI Server" on page 10-40
- "Writing a Multithreaded ATMI Client" on page 10-45

Testing a Multithreaded/Multicontexted ATMI Application

This topic includes the following sections:

- Testing Recommendations for a Multithreaded/Multicontexted ATMI Application
- Troubleshooting a Multithreaded/Multicontexted ATMI Application
- Error Handling for a Multithreaded/Multicontexted ATMI Application

Testing Recommendations for a Multithreaded/Multicontexted ATMI Application

We recommend following these recommendations during testing of your multithreaded and/or multicontexted code:

- Use a multiprocessor.
- Use a multithreaded debugger (if your operating system vendor offers one).
- Run stress tests to introduce a variety of timing conditions.

Troubleshooting a Multithreaded/Multicontexted ATMI Application

When you need to investigate possible causes of errors, we recommend that you start by checking whether and how the TPMULTICONTEXTS flag has been set. Errors are frequently introduced by failures to set this flag or to set it properly.

Improper Use of the TPMULTICONTEXTS Flag to tpinit( )

If a process includes the TPMULTICONTEXTS flag in a state for which this flag is not allowed (or omits TPMULTICONTEXTS in a state that requires it), then tpinit(3c) returns -1 and sets tperrno to TPEPROTO.

Calls to tpinit( ) Without TPMULTICONTEXTS

When tpinit(3c) is invoked without TPMULTICONTEXTS, it behaves as it does when called in a single-contexted application. When tpinit() has been invoked once, subsequent tpinit() calls without the TPMULTICONTEXTS flag succeed without further action. This is true even if the value of the TUXCONFIG or WSNADDR environment variable in the application has been changed. Calling tpinit() without the TPMULTICONTEXTS flag set is not allowed in multicontext mode.

If a client has not joined an application and tpinit(3c) is called implicitly (as a result of a call to another function that calls tpinit()), then the BEA Tuxedo system interprets the action as a call to tpinit() without the TPMULTICONTEXTS flag for purposes of determining which flags may be used in subsequent calls to tpinit().

For most ATMI functions, if a function is invoked by a thread that is not associated with a context in a process already operating in multicontext mode, the ATMI function fails with tperrno(5)=TPEPROTO.

Insufficient Thread Stack Size

On certain operating systems, the operating system default thread stack size is insufficient for use with the BEA Tuxedo system. Compaq Tru64 UNIX and UnixWare are two operating systems for which this is known to be the case. If the default thread stack size parameter is used, applications on these platforms dump core when a function with substantial stack usage requirements is called by any thread other than the main thread. Often the core file that is created does not give any obvious clues to the fact that an insufficient stack size is the cause of the problem.

When the BEA Tuxedo system is creating threads on its own, such as server-dispatched threads or a client unsolicited message thread, it can adjust the default stack size parameter on these platforms to a sufficient value. However, when an application is creating threads on its own, the application must specify a sufficient stack size. At a minimum, a value of 128K should be used for any thread that will access the BEA Tuxedo system.

On Compaq Tru64 UNIX and other systems on which POSIX threads are used, a thread stack size is specified by invoking `pthread_attr_setstacksize()` before calling `pthread_create()`. On UnixWare, the thread stack size is specified as an argument to `thr_create()`. Consult your operating system documentation for further information on this subject.

Error Handling for a Multithreaded/Multicontexted ATMI Application

Errors are reported in the user log. For each error, whether in single-context mode or multicontext mode, the following information is recorded:

process_ID.thread_ID.context_ID

See Also

- "How Multithreading and Multicontexting Work in a Client" on page 10-11
- "How Multithreading and Multicontexting Work in an ATMI Server" on page 10-17
- "Preliminary Guidelines for Programming a Multithreaded/Multicontexted ATMI Application" on page 10-28

CHAPTER

11 Managing Errors

This topic includes the following sections:

- System Errors
- Application Errors
- Handling Errors
- Transaction Considerations
- Central Event Log

System Errors

The BEA Tuxedo system uses TP-STATUS IN TPSTATUS-REC to supply information to a process when a routine fails. All ATMI calls set TP-STATUS to a value that describes the nature of the error. When a call does not return to its caller, as in the case of TPRETURN or TPFORWAR, which are used to terminate a service routine, the only way the system can communicate success or failure is through TP-STATUS in the requester.

APPL-RETURN-CODE is used to communicate user-defined conditions only. The system sets the value of APPL-RETURN-CODE to the value of APPL-CODE IN TPSVCRET-REC during TPRETURN. The system sets APPL-RETURN-CODE, regardless of the value of APPL-RETURN-CODE IN TPSTATUS-REC during TPRETURN, unless an error is encountered by TPRETURN or a transaction timeout occurs.

The codes returned in TP-STATUS represent categories of errors, which are listed in the following table.

Table 11-1 TP-STATUS Error Categories

Error Category	TP-STATUS Values
Abort	TPEABORT[2]
BEA Tuxedo system[1]	TPESYSTEM
Communication handle	TPELIMIT and TPEBADDESC
Conversational	TPEVENT
Duplicate operation	TPEMATCH
General communication	TPESVCFAIL, TPESVCERR, TPEBLOCK, and TPGOTSIG
Heuristic decision	TPEHAZARD[2] and TPEHEURISTIC[2]
Invalid argument[1]	TPEINVAL
MIB	TPEMIB
No entry	TPENOENT
Operating system[1]	TPEOS
Permission	TPEPERM
Protocol[1]	TPEPROTO
Queueing	TPEDIAGNOSTIC
Release compatibility	TPERELEASE
Resource manager	TPERMERR
Timeout	TPETIME
Transaction	TPETRAN[2]
Typed record mismatch	TPEITYPE and TPEOTYPE

1. Applicable to all ATMI calls for which failure is reported by the value returned in TP-STATUS.

2. Refer to “Fatal Transaction Errors” on page 11-18 for more information on this error category.

As footnote 1 shows, four categories of errors are reported by TP-STATUS and are applicable to all ATMI calls. The remaining categories are used only for specific ATMI calls.The following sections describe some error categories in detail.

Abort Errors

For information on the errors that lead to abort, refer to “Fatal Transaction Errors” on page 11-18.

BEA Tuxedo System Errors

BEA Tuxedo system errors indicate problems at the *system level*, rather than at the application level. When BEA Tuxedo system errors occur, the system writes messages explaining the exact nature of the errors to the central event log, and returns TPESYSTEM in TP-STATUS. For more information, refer to the “Central Event Log” on page 11-26. Because these errors occur in the system, rather than in the application, you may need to consult the system administrator to correct them.

Communication Handle Errors

Communication handle errors occur as a result of exceeding the maximum limit of communication handles or referencing an invalid value. Asynchronous and conversational calls return TPELIMIT when the maximum number of outstanding communication handles has been exceeded. TPEBADDESC is returned when an invalid communication handle value is specified for an operation.

Communication handle errors occur only during asynchronous calls or conversational calls. (Call descriptors are not used for synchronous calls.) Asynchronous calls depend on communication handles to associate replies with the corresponding requests. Conversational send and receive routines depend on communication handles to identify the connection; the call that initiates the connection depends on the availability of a communication handle.

Communication handle errors can be done by checking for specific errors at the application level.

Limit Errors

The system allows up to 50 outstanding communication handles (replies) per context (or BEA Tuxedo application association). This limit is enforced by the system; it cannot be redefined by your application.

The limit for communication handles for simultaneous conversational connections is more flexible than the limit for replies. The application administrator defines the limit in the configuration file. When the application is not running, the administrator can modify the MAXCONV parameter in the RESOURCES section of the configuration file. When the application is running, the administrator can modify the MACHINES section dynamically. Refer to tmconfig, wtmconfig(1) in the *BEA Tuxedo Command Reference* for more information.

Invalid Descriptor Errors

A communication handle can become invalid and, if referenced, cause an error to be returned to TP-STATUS in either of two situations:

- A communication handle is used to retrieve a message, which may be a failed message (TPEBADDESC).
- An attempt is made to reuse a stale communication handle (TPEBADDESC).

A communication handle might become stale, for example, in the following circumstances:

- When the application calls TPABORT or TPCOMMIT and transaction replies (sent without TPNOTRAN) remain to be retrieved.

- A transaction times out. When the timeout is reported by a call to TPGETRPLY, no message is retrieved using the specified handle and the handle becomes stale.

Conversational Errors

When an unknown handle is specified for conversational services, the TPSEND, TPRECV, and TPDISCON routines return TPEBADDESC.

When TPSEND and TPRECV fail with a TPEEVENT error after a conversational connection is established, an event has occurred. Data may or may not be sent by TPSEND, depending on the event. The system returns TPEEVENT in the TPEVENT member of TPSTATUS-REC and the course of action is dictated by the particular event.

For a complete description of conversational events, refer to "Understanding Conversational Communication Events" on page 7-13.

Duplicate Object Error

The TPEMATCH error code is returned in TP-STATUS when an attempt is made to perform an operation that results in a duplicate object. The following table lists the routines that may return the TPEMATCH error code and the associated cause.

Routine	Cause
TPADVERTISE	The *svcname* specified is already advertised for the server but with a function other than *func*. Although the function fails, *svcname* remains advertised with its current function (that is, *func* does not replace the current function name).
TPRESUME	The *tranid* points to a transaction identifier that another process has already resumed. In this case, the caller's state with respect to the transaction is not changed.
TPSUBSCRIBE	The specified subscription information has already been listed with the EventBroker.

For more information on these routines, refer to the *BEA Tuxedo ATMI COBOL Function Reference*.

General Communication Call Errors

General communication call errors can occur during any communication calls, regardless of whether those calls are synchronous or asynchronous. Any of the following errors may be returned in TP-STATUS: TPESVCFAIL, TPESVCERR, TPEBLOCK, or TPGOTSIG.

TPESVCFAIL and TPESVCERR Errors

If the reply portion of a communication fails as a result of a call to TPCALL or TPGETRPLY, the system returns TPESVCERR or TPSEVCFAIL to TP-STATUS. The system determines the error by the arguments that are passed to TPRETURN and the processing that is performed by this call.

If TPRETURN encounters an error in processing or handling arguments, the system returns an error to the original requester and sets TP-STATUS to TPESVCERR. The receiver determines that an error has occurred by checking the value of TP-STATUS. The system does not send the data from the TPRETURN call, and if the failure occurred on TPGETRPLY, it renders the call handle invalid.

If TPRETURN does not encounter the TPESVCERR error, then the value returned in TP-RETURN-VAL determines the success or failure of the call. If the application specifies TPFAIL in the TP-RETURN-VAL, the system returns TPESVCFAIL in TP-STATUS and sends the data message to the caller. If TP-RETURN-VAL is set to TPSUCCESS, the system returns successfully to the caller, TP-STATUS is not set, and the caller receives the data.

TPEBLOCK and TPGOTSIG Errors

The TPEBLOCK and TPGOTSIG error codes may be returned at the request or the reply end of a message and, as a result, can be returned for all communication calls.

The system returns TPEBLOCK when a blocking condition exists and the process sending a request (synchronously or asynchronously) indicates, by setting TPPNOBLOCK that it does not want to wait on a blocking condition. A blocking condition can exist when a request is being sent if, for example, all the system queues are full.

When TPCALL indicates a no blocking condition, only the sending part of the communication is affected. If a call successfully sends a request, the system does not return TPEBLOCK, regardless of any blocking situation that may exist while the call waits for the reply.

The system returns TPEBLOCK for TPGETRPLY when a call is made TPNOBLOCK and a blocking condition is encountered while TPGETRPLY is awaiting the reply. This may occur, for example, if a message is not currently available.

The TPGOTSIG error indicates an interruption of a system call by a signal; this situation is not actually an error condition. If TPSIGRSTRT is set, the calls do not fail and the system does not return the TPGOTSIG error code in TP-STATUS.

Invalid Argument Errors

Invalid argument errors indicate that an invalid argument was passed to a routine. Any ATMI call that takes arguments can fail if you pass it arguments that are invalid. In the case of a call that returns to the caller, the call fails and causes TP-STATUS to be set to TPEINVAL. In the case of TPRETURN or TPFORWAR, the system sets TP-STATUS to TPESVCERR for either the TPCALL or TPGETRPLY call that initiated the request and is waiting for results to be returned.

You can correct an invalid argument error at the *application level* by ensuring that you pass only valid arguments to routines.

No Entry Errors

No entry errors result from a lack of entries in the system tables or the data structure used to identify record types. The meaning of the no entry type error, TPENOENT, depends on the call that is returning it. The following table lists the calls that return this error and describes various causes of error.

Table 11-2 No Entry Errors

Call	Cause
TPINITIALIZE	The calling process cannot join the application because there is no space left in the bulletin board to make an entry for it. Check with the system administrator.
TPCALL TPACALL	The calling process references a service called SERVICE-NAME IN TPSVCDEF-REC that is not known to the system since there is no entry for it in the bulletin board. On an application level, ensure that you have referenced the service correctly; otherwise, check with the system administrator.
TPCONNECT	The system cannot connect to the specified name because the service named does not exist or it is not a conversational service.
TPGPRIO	The calling process seeks a request priority when no request has been made. This is an application-level error.
TPUNADVERTISE	The system cannot unadvertise SERVICE-NAME IN TPSVCDEF-REC because the name is not currently advertised by the calling process.

Operating System Errors

Operating system errors indicate that an operating system call has failed. The system returns TPEOS in TP-STATUS. On UNIX systems, the system returns a numeric value identifying the failed system call in the global variable Uunixerr. To resolve operating system errors, you may need to consult your system administrator.

Permission Errors

If a calling process does not have the correct permissions to join the application, the TPINITIALIZE call fails, returning TPEPERM in TP-STATUS. Permissions are set in the configuration file, outside of the application. If you encounter this error, check with the application administrator to make sure the necessary permissions are set in the configuration file.

Protocol Errors

Protocol errors occur when an ATMI call is invoked, either in the wrong order or using an incorrect process. For example, a client may try to begin communicating with a server before joining the application. Or TPCOMMIT may be called by a transaction participant instead of the initiator.

You can correct a protocol error at the *application level* by enforcing the rules of order and proper usage of ATMI calls.

To determine the cause of a protocol error, answer the following questions:

- Is the call being made in the correct order?
- Is the call being made by the correct process?

Protocol errors return the TPEPROTO value in TP-STATUS.

Refer to "Introduction to the COBOL Application-Transaction Monitor Interface" in the *BEA Tuxedo ATMI COBOL Function Reference* for more information.

Queuing Error

The TPENQUEUE(3cbl) or TPDEQUEUE(3cbl) routine returns TPEDIAGNOSTIC in TP-STATUS if the enqueuing or dequeuing on a specified queue fails. The reason for failure can be determined by the diagnostic returned via the *ctl* record. For a list of valid *ctl* flags, refer to TPENQUEUE(3cbl) or TPDEQUEUE(3cbl) in the *BEA Tuxedo ATMI COBOL Function Reference*.

Release Compatibility Error

The BEA Tuxedo system returns TPERELEASE in TP-STATUS if a compatibility issue exists between multiple releases of a BEA Tuxedo system participating in an application domain.

For example, the TPERELEASE error may be returned if the TPACK flag is set when issuing the TPNOTIFY(3cbl) routine (indicating that the caller blocks until an acknowledgment message is received from the target client), but the target client is using an earlier release of the BEA Tuxedo system that does not support the TPACK acknowledgement protocol.

Resource Manager Errors

Resource manager errors can occur with calls to TPOPEN(3cbl) and TPCLOSE(3cbl), in which case the system returns the value of TPERMERR in TP-STATUS. This error code is returned for TPOPEN when the resource manager fails to open correctly. Similarly, this error code is returned for TPCLOSE when the resource manager fails to close

correctly. To maintain portability, the BEA Tuxedo system does not return a more detailed explanation of this type of failure. To determine the exact nature of a resource manager error, you must interrogate the resource manager.

Timeout Errors

The BEA Tuxedo system supports timeout errors to establish a limit on the amount of time that the application waits for a service request or transaction. The BEA Tuxedo system supports two types of configurable timeout mechanisms: blocking and transaction.

A *blocking timeout* specifies the maximum amount of time that an application waits for a reply to a service request. The application administrator defines the blocking timeout for the system in the configuration file.

A *transaction timeout* defines the duration of a transaction, which may involve several service requests. To define the transaction timeout for an application, pass the `T-OUT` argument to `TPBEGIN`.

The system may return timeout errors on communication calls for either blocking or transaction timeouts, and on `TPCOMMIT` for transaction timeouts only. In each case, if a process is in transaction mode and the system returns `TPETIME` on a failed call, a transaction timeout has occurred.

By default, if a process is not in transaction mode, the system performs blocking timeouts.

If a process is not in transaction mode and a blocking timeout occurs on an asynchronous call, the communication call that blocked fails, but the call descriptor is still valid and may be used on a reissued call. Other communication is not affected.

When a transaction timeout occurs, the communication handle to an asynchronous transaction reply (specified without `TPNOTRAN`) becomes stale and may no longer be referenced.

`TPETIME` indicates a blocking timeout on a communication call if the call was not made in transaction mode or if `TPNOBLOCK` was not set.

Note: If you set TPNOBLOCK, a blocking timeout cannot occur because the call returns immediately if a blocking condition exists.

For additional information on handling timeout errors, refer to "Transaction Considerations" on page 11-15.

Transaction Errors

For information on transactions and the non-fatal and fatal errors that can occur, refer to "Transaction Considerations" on page 11-15.

Typed Record Errors

Typed record errors are returned when requests or replies to processes are sent in records of an unknown type. The TPCALL and TPACALL calls return TPEITYPE when a request data record is sent to a service that does not recognize the type of the record.

Processes recognize record types that are identified in both the configuration file and the BEA Tuxedo system libraries that are linked into the process. These libraries define and initialize a data structure that identifies the typed records that the process recognizes. You can tailor the library to each process, or an application can supply its own copy of a file that defines the record types. An application can set up the record type data structure (referred to as a record type switch) on a process-specific basis. For more information, see tuxtypes(5) and typesw(5) in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*.

The TPCALL and TPGETRPLY calls return TPEOTYPE when a reply message is sent in a record that is not recognized or not allowed by the caller. In the latter case, the record type is included in the type switch, but the type returned does not match the record that was allocated to receive the reply and a change in record type is not allowed by the caller. The caller indicates this preference by setting TPNOCHANGE. In this case, strong type checking is enforced; the system returns TPEOTYPE when it is violated. By default, weak type checking is used. In this case, a record type other than the type originally

allocated may be returned, as long as that type is recognized by the caller. The rules for sending replies are that the reply record must be recognized by the caller and, if strong type checking has been indicated, you must observe it.

Application Errors

Within an application, you can pass information about user-defined errors to calling programs using the *rcode* argument of TPRETURN. Also, the system sets the value of APPL-RETURN-CODE to the value of APPL-CODE IN TPSVCRET-REC during TPRETURN. For more information about TPRETURN(3cbl), refer to the *BEA Tuxedo ATMI COBOL Function Reference*.

Handling Errors

Your application logic should test for error conditions for the calls that have return values, and take appropriate action when an error occurs.

The following example shows a typical method of handling errors. The term ATMICALL(3) is used in this example to represent a generic ATMI call.

Listing 11-1 Handling Errors

```
. . .
CALL "TPINITIALIZE" USING TPINFDEF-REC
                    USR-DATA-REC
                    TPSTATUS-REC.
IF NOT TPOK
   error message, EXIT PROGRAM
CALL "TPBEGIN" USING TPTRXDEF-REC
               TPSTATUS-REC.
IF NOT TPOK
   error message, EXIT PROGRAM

   Make atmi calls
   Check return values

IF  TPEINVAL
      DISPLAY "Invalid arguments were given."
IF  TPEPROTO
      DISPLAY "A call was made in an improper context."
. . .
```

```
   Include all error cases described in the ATMICALL(3)
   reference page. Other return codes are not possible,
   so there is no need to test them.
. . .
   continue
```

The values of TP-STATUS provide details about the nature of each problem and suggest the level at which it can be corrected. If your application defines a list of error conditions specific to your processing, the same can be said for the values of APPL-RETURN-CODE IN TPSTATUS-REC.

Transaction Considerations

The following sections describe how various programming features work when used in transaction mode. The first section provides rules of basic communication etiquette that should be observed in code written for transaction mode.

Communication Etiquette

When writing code to be run in transaction mode, you must observe the following rules of basic communication etiquette:

- Processes that are participants in the same transaction must require replies for all requests. To include a request that requires no reply, set TPACALL to TPNOTRAN or TPNOREPLY.
- A service must retrieve all asynchronous transaction replies before calling TPRETURN or TPFORWAR. This rule must be observed regardless of whether the code is running in transaction mode.
- The initiator must retrieve all asynchronous transaction replies (made without TPNOTRAN) before calling TPCOMMIT.

- Replies must be retrieved for asynchronous calls that expect replies from non-participants of the transaction, that is, replies to requests made with TPACALL in which the transaction, but not the reply, is suppressed.
- If a transaction has not timed out but is marked "abort-only," any further communication should be performed with TPNOTRAN set so that the results of the communication are preserved after the transaction is rolled back.
- If a transaction has timed out:
  - The handle for the timed-out call becomes stale and any further reference to it returns TPEBADDESC.
  - Further calls to TPGETRPLY or TPRECV for any outstanding handles return a global state of transaction timeout; the system sets TP-STATUS to TPETIME.
  - Asynchronous calls can be made with TPACALL set to TPNOREPLY, TPNOBLOCK, or TPNOTRAN.
- Once a transaction has been marked "abort-only" for reasons other than timeout, a call to TPGETRPLY returns whatever value represents the local state of the call; that is, it returns either success or an error code that reflects the local condition.
- Once a handle is used with TPGETRPLY to retrieve a reply, or with TPSEND or TPRECV to report an error condition, it becomes invalid and any further reference to it returns TPEBADDESC. This rule is always observed, regardless of whether the code is running in transaction mode.
- Once a transaction is aborted, all outstanding transaction call handles (made without TPNOTRAN) become stale, and any further references to them return TPEBADDESC.

Transaction Errors

The following sections describe transaction-related errors.

Non-fatal Transaction Errors

When transaction errors occur, the system returns TPETRAN in TP-STATUS. The precise meaning of such an error, however, depends on the call that is returning it. The following table lists the calls that return transaction errors and describes possible causes of them.

Table 11-3 Transaction Errors

Call	Cause
TPBEGIN	Usually caused by a transient system error that occur during an attempt to start the transaction. The problem may clear up with a repeated call.
TPCANCEL	Returns TPETRAN when called from a transaction.
TPRESUME	The BEA Tuxedo system is unable to resume a global transaction because the caller is currently participating in work outside the global transaction with one or more resource managers. All such work must be completed before the global transaction can be resumed. The caller's state with respect to the local transaction is unchanged.
TPCONNECT, TPCALL, and TPACALL	A call was made in transaction mode to a service that does not support transactions. Some services belong to server groups that access a database management system (DBMS) that, in turn, support transactions. Other services, however, do not belong to such groups. In addition, some services that support transactions may require interoperation with software that does not. For example, a service that prints a form may work with a printer that does not support transactions. Services that do not support transactions may not function as participants in a transaction. The grouping of services into servers and server groups is an administrative task. In order to determine which services support transactions, check with your application administrator. You can correct transaction-level errors at the application level by enabling the setting TPSVCDEF-REF or by accessing the service for which an error was returned outside of the transaction.

Fatal Transaction Errors

When a fatal transaction error occurs, the application should explicitly abort the transaction by having the initiator call TPABORT. Therefore, it is important to understand the errors that are fatal to transactions. Three conditions cause a transaction to fail:

- The initiator or a participant in the transaction causes it to be marked "abort-only" for one of the following reasons:
  - TPRETURN encounters an error while processing its arguments; TP-STATUS is set to TPESVCERR.
  - The TP-RETURN-VAL to TPRETURN was set to TPFAIL; TP-STATUS is set to TPESVCFAIL.
  - The type of the reply record is not known or not allowed by the caller and, as a result, success or failure cannot be determined; TP-STATUS is set to TPEOTYPE.
- The transaction times out; TP-STATUS is set to TPETIME.
- TPCOMMIT is called by a participant rather than by the originator of a transaction; TP-STATUS is set to TPEPROTO.

The only protocol error that is fatal to transactions is calling TPCOMMIT from the wrong participant in a transaction. This error can be corrected in the application during the development phase.

If TPCOMMIT is called after an initiator/participant failure or transaction timeout, the result is an implicit abort error. Then, because the commit failed, the transaction should be aborted.

If the system returns TPESVCERR, TPESVCFAIL, TPEOTYPE, or TPETIME for any communication call, the transaction should be aborted explicitly with a call to TPABORT. You need not wait for outstanding communication handles before explicitly aborting the transaction. However, because these communication handles are considered stale after the call is aborted, any attempt to access them after the transaction is terminated returns TPEBADDESC.

In the case of TPESVCERR, TPESVCFAIL, and TPEOTYPE, communication calls continue to be allowed as long as the transaction has not timed out. When these errors are returned, the transaction is marked abort-only. To preserve the results of any further

work, you should call any communication functions with TPNOTRAN. By setting this flag, you ensure that the work performed for the transaction marked “abort-only” will not be rolled back when the transaction is aborted.

When a transaction timeout occurs, communication can continue, but communication requests cannot:

- Require replies
- Block
- Be performed on behalf of the caller’s transaction

Therefore, to make asynchronous calls, you must set TPNOREPLY, TPNOBLOCK, or TPNOTRAN.

Heuristic Decision Errors

The TPCOMMIT call may return TPEHAZARD or TPEHEURISTIC, depending on how TP-COMMIT-CONTROL is set.

If you set TP-COMMIT-CONTROL to TP-CMT-LOGGED, the application obtains control before the second phase of a two-phase commit is performed. In this case, the application may not be aware of a heuristic decision that occurs during the second phase.

TPEHAZARD or TPEHEURISTIC can be returned in a one-phase commit, however, if a single resource manager is involved in the transaction and it returns a heuristic decision or a hazard indication during a one-phase commit.

If you set TP_COMMIT_CONTROL to TP_CMT_COMPLETE, then the system returns TPEHEURISTIC if any resource manager reports a heuristic decision, and TPEHAZARD if any resource manager reports a hazard. TPEHAZARD specifies that a participant failed during the second phase of commit (or during a one-phase commit) and that it is not known whether a transaction completed successfully.

Transaction Timeouts

As described in "Transaction Errors" on page 11-16, two types of timeouts can occur in a BEA Tuxedo application: blocking and transaction. The following sections describe how various programming features are affected by transaction timeouts. Refer to "Transaction Errors" on page 11-16 for more information on timeouts.

TPCOMMIT Call

What is the state of a transaction if a timeout occurs after a call to TPCOMMIT? If the transaction timed out and the system knows that it was aborted, the system reports these events by setting TP-STATUS to TPEABORT. If the status of the transaction is unknown, the system sets the error code to TPETIME.

When the state of a transaction is in doubt, you must query the resource manager. First, verify whether or not any of the changes that were part of the transaction were applied. Then you can determine whether the transaction was committed or aborted.

TPNOTRAN

When a process is in transaction mode and makes a communication call with TPNOTRAN, it prohibits the called service from becoming a participant in the current transaction. Whether the service request succeeds or fails has no impact on the outcome of the transaction. The transaction can still timeout while waiting for a reply that is due from a service, whether it is part of the transaction or not.

For additional information on using TPNOTRAN, refer to "TPRETURN and TPFORWAR Calls" on page 11-21.

TPRETURN and TPFORWAR Calls

If you call a process while running in transaction mode, TPRETURN and TPFORWAR place the service portion of the transaction in a state that allows it to be either committed or aborted when the transaction completes. You can call a service several times on behalf of the same transaction. The system does not fully commit or abort the transaction until the initiator of the transaction calls TPCOMMIT or TPABORT.

Neither TPRETURN nor TPFORWAR should be called until all outstanding handles for the communication calls made within the service have been retrieved. If you call TPRETURN with outstanding handles for which TP-RETURN-VAL is set to TPSUCCESS, the system encounters a protocol error and returns TPESVCERR to the process waiting on TPGETRPLY. If the process is in transaction mode, the system marks the caller as "abort-only." Even if the initiator of the transaction calls TPCOMMIT, the system implicitly aborts the transaction. If you call TPRETURN with outstanding handles for which TP-RETURN-VAL is set to TPFAIL, the system returns TPESVCFAIL to the process waiting on TPGETRPLY. The effect on the transaction is the same.

When you call TPRETURN while running in transaction mode, this function can affect the result of the transaction by the processing errors that it encounters or that are retrieved from the value placed in TP-RETURN-VAL by the application.

You can use TPFORWAR to indicate that success has been achieved up to a particular point in the processing of a request. If no application errors have been detected, the system invokes TPFORWAR; otherwise, the system invokes TPRETURN with TPFAIL. If you call TPFORWAR improperly, the system considers the call a processing error and returns a failed message to the requester.

tpterm( ) Function

Use the TPTERM call to remove a client context from an application.

If the client context is in transaction mode, the call fails with TPEPROTO returned in TP-STATUS, and the client context remains part of the application and in transaction mode.

When the call is successful, the client context is allowed no further communication or participation in transactions because the current thread of execution is no longer part of the application.

Resource Managers

When you use an ATMI call to define transactions, the BEA Tuxedo system executes an internal call to pass any global transaction information to each resource manager participating in the transaction. When you call `TPCOMMIT` or `TPABORT`, for example, the system makes internal calls to direct each resource manager to commit or abort the work it did on behalf of the caller's global transaction.

When a global transaction has been initiated, either explicitly or implicitly, you should not make explicit calls to the resource manager's transaction calls in your application code. Failure to follow this transaction rule causes indeterminate results. You can use the `TPGETLEV` call to determine whether a process is already in a global transaction before calling the resource manager's transaction call.

Some resource managers allow programmers to configure certain parameters (such as the transaction consistency level) by specifying options available in the interface to the resource managers themselves. Such options are made available in two forms:

- Resource manager-specific function calls that can be used by programmers of distributed applications to configure options.
- Hard-coded options incorporated in the transaction interface supplied by the provider of the resource manager.

Consult the documentation for your resource managers for additional information.

The method of setting options varies for each resource manager. In the BEA Tuxedo System SQL resource manager, for example, the `set transaction` statement is used to negotiate specific options (consistency level and access mode) for a transaction that has already been started by the BEA Tuxedo system.

Sample Transaction Scenarios

The following sections provide some considerations for the following transaction scenarios:

- Called Service in Same Transaction as Caller
- Called Service in Different Transaction with AUTOTRAN Set
- Called Service That Starts a New Explicit Transaction

Called Service in Same Transaction as Caller

When a caller in transaction mode calls another service to participate in the current transaction, the following facts apply:

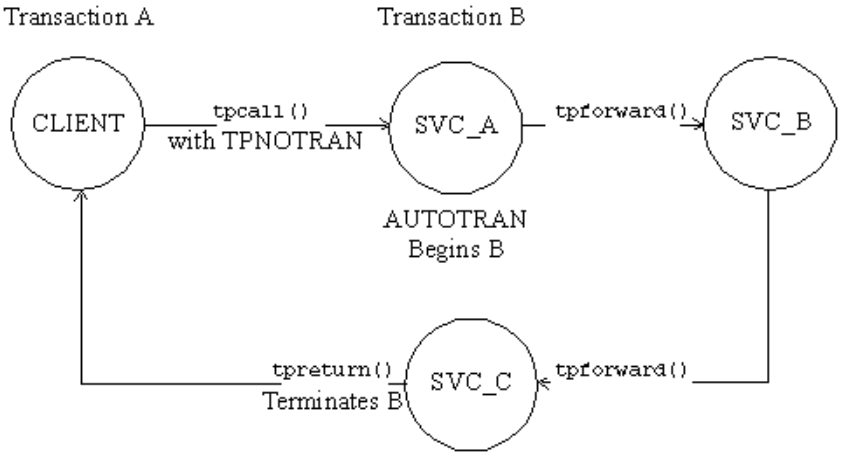
- `TPRETURN` and `TPFORWAR`, when called by the participating service, place that service's portion of the transaction in a state from which it can be either aborted or committed by the initiator.
- The success or failure of the called process affects the current transaction. If any fatal transaction errors are encountered by the participant, the current transaction is marked "abort-only."
- Whether or not the tasks performed by a successful participant are applied depends on the fate of the transaction. In other words, if the transaction is aborted, the work of all participants is reversed.
- `TPNOREPLY` cannot be used when calling another service to participate in the current transaction.

Called Service in Different Transaction with AUTOTRAN Set

If you issue a communication call with TPNOTRAN set and the called service is configured such that a transaction automatically starts when the service is called, the system places both the calling and called processes in transaction mode, but the two constitute different transactions. In this situation, the following facts apply:

- TPRETURN plays the initiator's transaction role: it terminates the transaction in the service in which the transaction was automatically started. Alternatively, if the transaction is automatically started in a service that terminates with TPFORWAR, the TPRETURN call issued in the last service in the forward chain plays the initiator's transaction role: it terminates the transaction. (For an example, refer to the figure called "Transaction Roles of TPFORWAR and TPRETURN with AUTOTRAN" on page 11-25.)
- Because it is in transaction mode, TPRETURN is vulnerable to the failure of any participant in the transaction, as well as to transaction timeouts. In this scenario, the system is more likely to return a failed message.
- The state of the caller's transaction is not affected by any failed messages or application failures returned to the caller.
- The caller's own transaction may timeout as the caller waits for a reply.
- If no reply is expected, the caller's transaction cannot be affected in any way by the communication call.

Figure 11-1 Transaction Roles of TPFORWAR and TPRETURN with AUTOTRAN



Called Service That Starts a New Explicit Transaction

If a communication call is made with TPNOTRAN, and the called service is not automatically placed in transaction mode by a configuration option, the service can define multiple transactions using explicit calls to TPBEGIN, TPCOMMIT, and TPABORT. As a result, the transaction can be completed before a call is issued to TPRETURN.

In this situation, the following facts apply:

- TPRETURN plays no transaction role; that is, the role of TPRETURN is always the same, regardless of whether transactions are explicitly defined in the service routine.
- TPRETURN can return any value in TP-RETURN-VAL, regardless of the outcome of the transaction.
- Typically, the system returns processing errors, record type errors, or application failure, and follows the normal rules for TPESVCFAIL, TPEITYPE/TPEOTYPE, and TPESVCERR.
- The state of the caller's transaction is not affected by any failed messages or application failures returned to the caller.

- The caller is vulnerable to the possibility that its own transaction may time out as it waits for its reply.
- If no reply is expected, the caller's transaction cannot be affected in any way by the communication call.

BEA TUXEDO System-supplied Subroutines

The BEA Tuxedo system-supplied subroutines, TPSVRINIT and TPSVRDONE, must follow certain rules when used in transactions.

The BEA Tuxedo system server calls TPSVRINIT during initialization. Specifically, TPSVRINIT is called after the calling process becomes a server but before it starts handling service requests. If TPSVRINIT performs any asynchronous communication, all replies must be retrieved before the function returns; otherwise, the system ignores all pending replies and the server exits. If TPSVRINIT defines any transactions, they must be completed with all asynchronous replies retrieved before the function returns; otherwise, the system aborts the transaction and ignores all outstanding replies. In this case, the server exits gracefully.

The BEA Tuxedo system server abstraction calls TPSVRDONE after it finishes processing service requests but before it exits. At this point, the server's services are no longer advertised, but the server has not yet left the application. If TPSVRDONE initiates communication, it must retrieve all outstanding replies before it returns; otherwise, pending replies are ignored by the system and the server exits. If a transaction is started within TPSVRDONE, it must be completed with all replies retrieved; otherwise, the system aborts the transaction and ignores the replies. In this case, too, the server exits.

Central Event Log

The central event log is a record of significant events in your BEA Tuxedo application. Messages about these events are sent to the log by your application clients and services via the USERLOG(3cbl) routine.

Any analysis of the central event log must be provided by the application. You should establish strict guidelines for the events that are to be recorded in the `USERLOG(3cbl)`. Application debugging can be simplified by eliminating trivial messages.

For information on configuring the central event log on the Windows 2000 platform, refer to *Using BEA Tuxedo ATMI on Windows*.

Log Name

The application administrator defines (in the configuration file) the absolute pathname that is used as the prefix of the name of the error message file on each machine. The `USERLOG(3cbl)` routine creates a date—in the form *`mmddyy`*, representing the month, day, and year—and adds this date to the pathname prefix, forming the full filename of the central event log. A new file is created daily. Thus, if a process sends messages to the central event log on succeeding days, the messages are written into different files.

Log Entry Format

Entries in the log consist of the following components:

- Tag consisting of:
  - Time of day (*`hhmmss`*)
  - Machine name (for example, the name returned by the `uname`(1) command on a UNIX system)
  - Name, process ID, and thread ID (which is 0 on platforms that do not support threads) of the thread calling `USERLOG(3cbl)`
  - Context ID of the thread calling `USERLOG(3cbl)`
- Message text

  The text of each message is preceded by the catalog name and number of that message.

For example, suppose that a security program executes the following call at `4:22:14pm` on a UNIX machine called `mach1` (as returned by the `uname` command):

```
01 LOG-REC PIC X(15) VALUE "UNKNOWN USER ".
01 LOGREC-LEN PIC S9(9) VALUES IS 13.
CALL "USERLOG" USING LOG-REC LOGREC-LEN TPSTATUS-REC.
```

The resulting log entry appears as follows:

```
162214.mach1!security.23451: UNKNOWN USER
```

In this example, the process ID for security is `23451`.

If the preceding message was generated by the BEA Tuxedo system (rather than by the application), it might appear as follows:

```
162214.mach1!security.23451: COBAPI_CAT: 999: UNKNOWN USER
```

In this case, the message catalog name is `COBAPI_CAT` and the message number is `999`.

If the message is sent to the central event log while the process is in transaction mode, other components are added to the tag in the user log entry. These components consist of the literal string `gtrid` followed by three long hexadecimal integers. The integers uniquely identify the global transaction and make up what is referred to as the global transaction identifier, that is, the `gtrid`. This identifier is used mainly for administrative purposes, but it also appears in the tag that prefixes the messages in the central event log. If the system writes the message to the central event log in transaction mode, the resulting log entry appears as follows:

```
162214.mach1!security.23451: gtrid x2 x24e1b803 x239:
   UNKNOWN USER
```

Writing to the Event Log

To write a message to the event log, you must perform the following steps:

- Assign the error message you wish to write to the log to a record and use the record name as the argument to the call.
- Specify the literal text of the message within double quotes, as the argument to the `USERLOG(3cbl)` call, as shown in the following example:

```
01 TPSTATUS-REC.
   COPY TPSTATUS.
01 LOGMSG     PIC X(50).
01 LOGMSG-LEN PIC S9(9) COMP-5.
. . .
CALL "TPOPEN" USING TPSTSTUS-REC.
```

```
IF NOT TPOK
        MOVE "TPSVRINIT: Cannot Open Data Base" TO LOGMSG
        MOVE 43 LOGMSG-LEN
        CALL "USERLOG" USING LOGMSG
                       LOGMSG-LEN
                       TPSTATUS-REC.
. . .
```

In this example, the message is sent to the central event log if `TPOPEN(3cbl)` returns `-1`.

CHAPTER

12 COBOL Language Bindings for the Workstation Component

This topic includes the following sections:

- UNIX Bindings
- Microsoft Windows Bindings

Refer to *Using the BEA Tuxedo Workstation Component* for more information on the Workstation platform.

UNIX Bindings

The following sections describe how to write and build client programs, and set appropriate environment variables when developing, in COBOL, a BEA Tuxedo application on a UNIX platform.

Writing Client Programs

You can develop COBOL client programs for a UNIX platform in the same way that you develop COBOL clients in the BEA Tuxedo administrative domain. All ATMI calls are available.

Building Client Programs

To compile and link-edit Workstation client programs, use the `buildclient(1)` command. If you are building a UNIX Workstation client on the native node, use the `-w` option to have the client built using the Workstation libraries.

If you are building a client on a native node, and both native and Workstation libraries are present, the native libraries are used by default. In this case, specifying the `-w` option ensures that the correct libraries for a Workstation client are used.

On a workstation, where only the Workstation libraries are present, it is not necessary to specify `-w`.

The following example shows how to use the `buildclient` command on a native node.

Listing 12-1 Example of Running buildclient on a UNIX Platform

```
ALTCC=cobcc ALTCFLAGS="-I /APPDIR/include"
COBCPY=$TUXDIR/cobinclude
COBOPT="-C ANS85 -C ALIGN=8 -C NOIBMCOMP -C TRUNC=ANSI -C OSEXT=cbl"
export COBOPT COBCPY ALTCC ALTCFLAGS
buildclient -C -w -o empclient -f name.cbl -f "userlib1.a userlib2.a"
```

The `-o` option enables you to specify a name for your output file. Input files specified with the `-f` option are linkedited before system libraries.

As illustrated, the `TUXDIR` environment variable must be used to ensure that the `buildclient` command can locate system libraries. Be sure that you have defined `TUXDIR`. The `CC` environment variable defaults to `cc`, but can be set to another compiler through `ALTCC`.

Setting Environment Variables

Workstation clients make use of several environment variables.

The following table lists the environment variables that are checked by TPINITIALIZE when a Workstation client attempts to join an application.

Table 12-1 Environment Variables Checked by TPINITIALIZE on a UNIX Platform

Environment Variable	Description
WSENVFILE	Name of a file containing environment variable settings to be used in the client's environment.
WSNADDR	Network address of the Workstation listener process through which the client gains access to the application. Use the value specified in the application configuration file for the Workstation listener to be called. If the value begins with the characters 0x, the system interprets it as a string of hexadecimal digits; otherwise, the system interprets it as ASCII characters.
WSDEVICE	Name of the device to be used to access the network. Not required by all transport layer interfaces.
WSTYPE	Workstation type. Used by TPINITIALIZE when that call is invoked by a Workstation client to negotiate encode/decode responsibilities with the native site. If you do not specify WSTYPE, the system performs encoding, even if WSTYPE is not specified on the native site, either. You must explicitly specify the same WSTYPE value for both the native and Workstation client sites to ensure that the encode/decode feature is turned off.
WSRPLYMAX	Maximum amount of core memory that the ATMI uses for buffering application replies before dumping them to disk. Used by TPINITIALIZE. The default system limit is 256,000 bytes. Whether you should use WSRPLYMAX to set a lower limit depends on the amount of memory available on your machine. Writing replies to disk causes a substantial reduction in performance.

Environment Variable	Description
WSFADDR	The network address used by the Workstation client when connecting to the Workstation listener or Workstation handler. This variable, along with the WSFRANGE variable, determines the range of TCP/IP ports to which a Workstation client will attempt to bind before making an outbound connection. This address must be a TCP/IP address.
WSFRANGE	The range of TCP/IP ports to which a Workstation client process attempts to bind before making an outbound connection. The WSFADDR parameter specifies the base address of the range. The default is 1.

Other environment variables may be needed by Workstation COBOL clients on a UNIX workstation, depending on which components of the BEA Tuxedo system are being used.

Note: MicroFocus delivers LIBNSL.a as a shared object, which is required by buildclient when linking a Workstation client. Because MicroFocus COBOL does not support shared objects on UNIX 3.2, Workstation for UNIX 3.2 is not supported.

Microsoft Windows Bindings

The following sections describe how to write and build client programs, build ACCEPT/DISPLAY clients, block network behavior, and restore the network environment when developing, in COBOL, a BEA Tuxedo application for the Microsoft Windows platform.

Writing Client Programs

All program-specific ATMI calls are available.

Building Client Programs

To compile the COBOL source files that call the ATMI, you must use the COBOL compiler with the `LITLINK` option. To linkedit the Workstation client object files, use the `buildclient(1)` command. While the syntax of the command is straightforward, the usage varies according to the compilation system used.

The following example shows how to use the `buildclient` command.

Listing 12-2 Example of Running buildclient on a Windows Platform

```
COBCPY=C:\TUXEDO\COBINC
COBDIR=C:\COBOL\LBR;C:\COBOL\EXEDLL
PATH=C:\COBOL\EXEDLL;...
TUXDIR=C:\tuxedo
LIB=C:\NET\TOOLKIT\LIB;C:\MSVC\LIB;C:\TUXEDO\LIB;C:\COBOL\LIB
buildclient -C -o EMP.EXE -f EMP -f "/NOD/NOI/NOE/CO/SE:300" -l WLIBSOCK

For Windows NT:

buildclient -C -o EMP.EXE  -f empobj
```

The following table describes the `buildclient` command options used in the preceding example.

Table 12-2 buildclient Command Options for Windows Platform

Option	Description
`-o` *name*	Name of the executable file being created. The default is `client.exe`.
`-f` *firstfiles*	One or more object files to be included before the BEA Tuxedo libraries. You can use the `-f` option to pass options to the compiler or linker. To specify more than one filename, enter a list of files after `-f`, using white space to separate filenames and double quotation marks around the list. You can also specify multiple filenames using multiple occurrences of the `-f` option on the command line.

Option	Description
`-l libfiles`	Libraries to be included after the BEA Tuxedo libraries. To specify more than one filename, you must separate the names by white space and enclose the list in quotation marks. You can also specify multiple filenames using multiple occurrences of the `-l` option on the command line.

Building ACCEPT/DISPLAY Clients

The following example shows how to build an executable client for an `ACCEPT/DISPLAY` application, such as `CSIMPAPP`.

Listing 12-3 Building ACCEPT/DISPLAY clients

```
a) compile the COBOL module and create a file.obj
         cobol file.cbl omf(obj) litlink;
b) use the following link statement
         link FILE+cblwinaf,,,\
         wcobatmi+cobws+wtuxws+ \
         lcobol+lcoboldw+cobw+cobfp87w+  \
         wlibsock,FILE.def  /nod/noe;
   For Windows NT the link statement is:
         cbllink -oEMP.exe EMP.obj  \
         cobws.lib ncobatmi.lib wtuxws32.lib  \
         libcmt.lib user32.lib
```