BEA Tuxedo

Programming a BEA Tuxedo ATMI Application Using C

BEA Tuxedo Release 8.0
Document Edition 8.0
June 2001

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Programming a BEA Tuxedo ATMI Application Using C

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

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

About This Document

This document explains how to program BEA Tuxedo ATMI applications using the C 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 Buffers,” provides instructions on managing and using typed buffers, including VIEW, FML, and XML buffers.
- Chapter 4, “Writing Clients,” provides instructions on writing and building BEA Tuxedo client applications using the C language. A client process example is provided.
- Chapter 5, “Writing Servers,” provides instructions on writing and building BEA Tuxedo servers using the C 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 managing errors, including both system and application errors.

What You Need to Know

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

This document assumes a familiarity with the BEA Tuxedo platform and C 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:

- *BEA Tuxedo ATMI C Function Reference*
- `compilation(5)` and `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.
italics	Indicates emphasis or book titles.

Convention	Item

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

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *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`*`]...`

`|`	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.

Convention	Item

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

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

CHAPTER

1 Introduction to BEA Tuxedo Programming

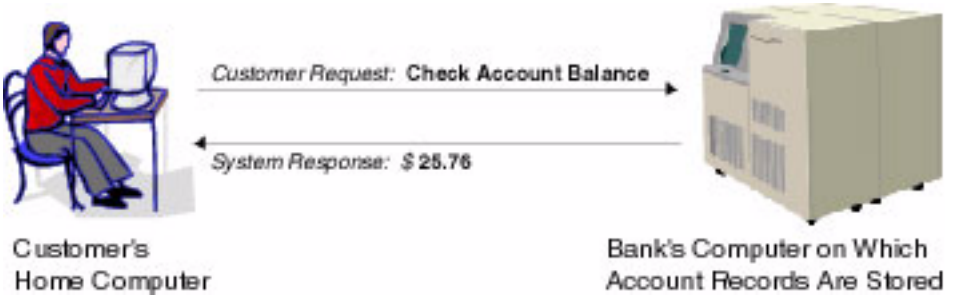
This topic includes the following sections:

- 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) functions.

The BEA Tuxedo distributed application programming environment provides the API functions 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 allocate message buffers and 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 Request/Response Client

```
main()
 {
      allocate a TPINIT buffer
      place initial client identification in buffer
      enroll as a client of the BEA Tuxedo application
      allocate buffer
      do while true {
           place user input in buffer
           send service request
           receive reply
           pass reply to the user }
      leave application
 }
```

Most of the actions described in the above listing are implemented with ATMI functions. Others—placing the user input in a buffer and passing the reply to the user—are implemented with C language functions.

During the "allocate buffer" phase, the client program allocates a memory area, called a *typed buffer*, from the BEA Tuxedo run-time system. A typed buffer is simply a memory buffer with an associated format, for example, a C structure.

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 functions are required for these two approaches.

Before you can execute an ATMI client, you must run the `buildclient` 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` command.

BEA Tuxedo Servers

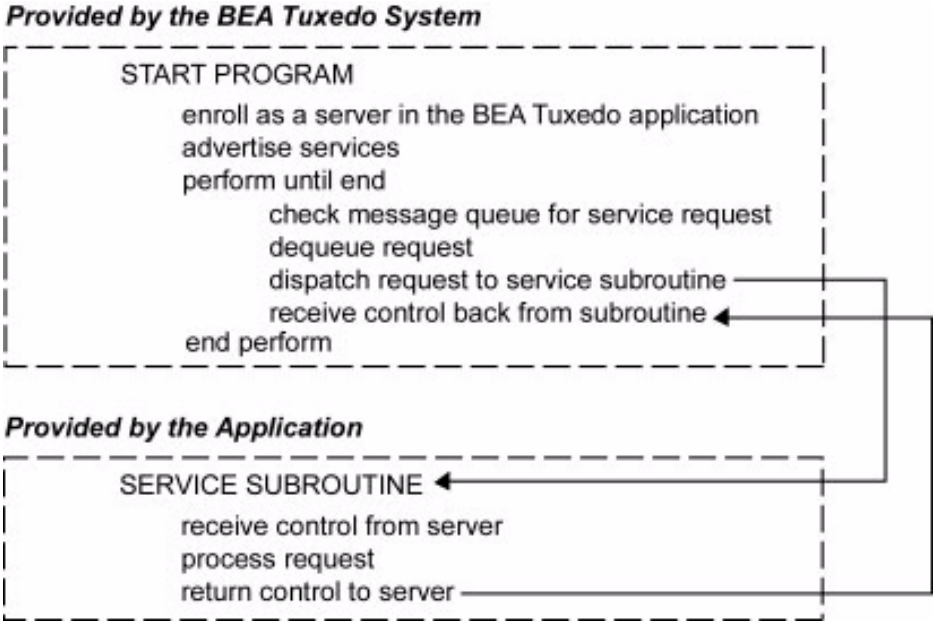
A BEA Tuxedo ATMI *server* is a process that provides one or more *services* to a client. A service is a specific business task that a client may need to perform. Servers receive requests from clients and dispatch them to the appropriate service subroutines.

Basic Server Operation

To build server processes, applications combine their service subroutines with a `main()` process provided by the BEA Tuxedo system. This system-supplied `main()` is a set of predefined functions. It performs server initialization and termination and allocates buffers 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.

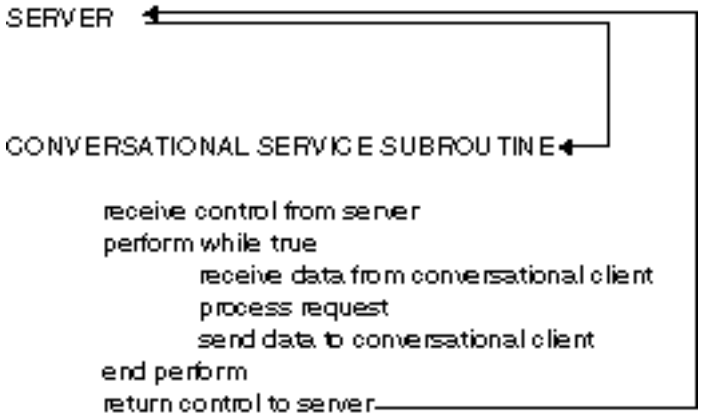
Figure 1-2 Pseudo-code for a Request/Response Server and a Service Subroutine



After initialization, an ATMI server allocates a buffer, 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 `main()` process contains the code needed to enroll a process as an ATMI server, advertise services, allocate buffers, and dequeue requests. ATMI functions 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 `main()` and generate an executable server. To do so, run the `buildserver` 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 functions.

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 C 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 functions are C language functions that resemble operating system calls. They implement communication among application modules running under the control of the BEA Tuxedo system transaction monitor, including all the associated resources you need.

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

Table 1-2 Using the ATMI Function

For a Task Related to . . .	Use This C Function . . .	To . . .	For More Information, Refer to . . .
Buffer management	`tpalloc()`	Create a message buffer	"Managing Typed Buffers" on page 3-1
	`tprealloc()`	Resize a message buffer	
	`tptypes()`	Get a message type and subtype	
	`tpfree()`	Free a message buffer	
Client membership	`tpchkauth()`	Check whether authentication is required	"Writing Clients" on page 4-1
	`tpinit()`	Join an application	
	`tpterm()`	Leave an application	
Multiple application context management	`tpgetctxt(3c)`	Retrieve an identifier for the current thread's context	"Programming a Multithreaded and Multicontexted ATMI Application" on page 10-1
	`tpsetctxt(3c)`	Set the current thread's context in a multicontexted process	

Table 1-2 Using the ATMI Function

For a Task Related to . . .	Use This C Function . . .	To . . .	For More Information, Refer to . . .
Service entry and return	`tpsvrinit()`	Initialize a server	■ "Writing Servers" on page 5-1 ■ "Programming a Multithreaded and Multicontexted ATMI Application" on page 10-1
	`tpsvrdone()`	Terminate a server	
	`tpsvrthrinit()`	Initialize an individual server thread	
	`tpsvrthrdone()`	Termination code for an individual server thread	
	`tpreturn()`	End a service function	
	`tpforward()`	Forward a request	
Dynamic advertisement	`tpadvertise()`	Advertise a service name	"Writing Servers" on page 5-1
	`tpunadvertise()`	Unadvertise a service name	
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	
	`tpgetrply()`	Receive an asynchronous response	
	`tpcancel()`	Cancel an asynchronous request	

Table 1-2 Using the ATMI Function

For a Task Related to . . .	Use This C Function . . .	To . . .	For More Information, Refer to . . .
Conversational communication	`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(3c)`	Enqueue a message to a message queue	*Using the ATMI /Q Component*
	`tpdequeue(3c)`	Dequeue a message from a message queue	
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	
	`tppost()`	Post an event message	
	`tpsubscribe()`	Subscribe to event messages	
	`tpunsubscribe()`	Unsubscribe to event messages	

Table 1-2 Using the ATMI Function

For a Task Related to . . .	Use This C Function . . .	To . . .	For More Information, Refer to . . .
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	
	`tpsuspend()`	Suspend the current transaction	
	`tpresume()`	Resume a transaction	
Resource management	`tpopen(3c)`	Open a resource manager	*Setting Up a BEA Tuxedo Application*
	`tpclose(3c)`	Close a resource manager	

Table 1-2 Using the ATMI Function

For a Task Related to . . .	Use This C Function . . .	To . . .	For More Information, Refer to . . .
Security	tpkey_open(3c)	Open a key handle for digital signature generation, message encryption, or message decryption	*Using Security in CORBA Applications*
	tpkey_getinfo(3c)	Get information associated with a key handle	
	tpkey_setinfo(3c)	Set optional attributes associated with a key handle	
	tpkey_close(3c)	Close a previously opened handle	
	tpsign(3c)	Mark a typed message buffer for generation of a digital signature	
	tpseal(3c)	Mark a typed message buffer for generation of an encryption envelope	
	tpenvelope(3c)	Access the digital signature and recipient information associated with a typed message buffer	
	tpexport(3c)	Convert a typed message buffer into an exportable, machine-independent (externalized) string representation	
	tpimport(3c)	Convert an externalized string representation back into a typed message buffer	

CHAPTER

2 Programming Environment

This topic includes the following sections:

- Updating the UBBCONFIG Configuration File
- Setting Environment Variables
- Including the Required Header Files
- 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 buffers 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 by Functional Category

Functional Category	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.
Compilation	`CC`	Command that invokes the C compiler. Default is `cc`.	`buildclient(1)` and `buildserver(1)` commands.
	`CFLAGS`	Link edit flags to be passed to the C compiler. Link edit flags are optional.	`buildclient(1)` and `buildserver(1)` commands.
Data compression	`TMCMPPRFM`	Level of compression (between 1 and 9).	BEA Tuxedo application programs that perform data compression.
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.

Table 2-2 Programming-related Environment Variables by Functional Category (Continued)

Functional Category	Environment Variable	Defines the . . .	Used by . . .
Buffer management	FIELDTBLS or FIELDTBLS32	Comma-separated list of field table filenames for FML and FML32 typed buffers, respectively. Required only for FML and VIEW types.	FML and FML32 typed buffers 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 typed buffers and FML VIEWs
	VIEWFILES or VIEWFILES32	Comma-separated list of allowable filenames for VIEW and VIEW32 typed buffers, respectively.	VIEW and VIEW32 typed buffers
	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 typed buffers

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*

Including the Required Header Files

The following table summarizes the header files that may need to be specified within the application programs, using the `#include` statement, in order to interface properly with the BEA Tuxedo system.

Table 2-3 Required Header Files

For . . .	You must include . . .
All BEA Tuxedo application programs	`atmi.h` header file supplied by the BEA Tuxedo system
Application programs with `FML` typed buffers	■ Header file generated from the corresponding field table files ■ `fml.h` header file supplied by the BEA Tuxedo system
Application program with VIEW typed buffers	Header file generated from the corresponding view description files

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*

CHAPTER

3 Managing Typed Buffers

This topic includes the following sections:

- Overview of Typed Buffers
- Allocating a Typed Buffer
- Putting Data in a Buffer
- Resizing a Typed Buffer
- Checking for Buffer Type
- Freeing a Typed Buffer
- Using a VIEW Typed Buffer
- Using an FML Typed Buffer
- Using an XML Typed Buffer
- Customizing a Buffer

Overview of Typed Buffers

Before a message can be sent from one process to another, a buffer must be allocated for the message data. BEA Tuxedo ATMI clients use typed buffers to send messages to ATMI servers. A typed buffer is a memory area with a category (type) and optionally a subcategory (subtype) associated with it. Typed buffers 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 buffers require different routines to initialize, send and receive messages, and encode and decode data. Each buffers is designated as a specific type so that the appropriate routines can be called automatically without programmer intervention.

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

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

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

Table 3-1 Typed Buffers

Typed Buffer	Description	Self-Describing	Subtype	Data-Dependent Routing	Encoding/ Decoding
CARRAY	Undefined array of characters, any of which can be NULL. This typed buffer 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
FML (Field Manipulation Language)	Proprietary BEA Tuxedo system type of self-describing buffer in which each data field carries its own identifier, an occurrence number, and possibly a length indicator. Because all data manipulation is done via FML function calls rather than native C statements, the FML buffer offers data-independence and greater flexibility at the expense of some processing overhead. The FML buffer uses 16 bits for field identifiers and lengths of fields. Refer to "Using an FML Typed Buffer" on page 3-22 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 buffers. Refer to "Using an FML Typed Buffer" on page 3-22 for more information.	Yes	No	Yes	Yes

Table 3-1 Typed Buffers (Continued)

Typed Buffer	Description	Self-Describing	Subtype	Data-Dependent Routing	Encoding/Decoding
STRING	Array of characters that terminates with a NULL character. The STRING buffer is self-describing, so the BEA Tuxedo system can convert data automatically when data is exchanged by machines with different character sets.	Yes	No	No	No
VIEW	C 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 buffer. Encoding and decoding are performed automatically if the buffer is passed between machines of different types. Refer to "Using a VIEW Typed Buffer" on page 3-16 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 buffers. Refer to "Using a VIEW Typed Buffer" on page 3-16 for more information.	No	Yes	Yes	Yes
X_C_TYPE	Equivalent to VIEW.	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 Buffer	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 Buffer" on page 3-26 for more information.	No	No	Yes	No
X_OCTET	Equivalent to CARRAY.	No	No	No	No

All buffer types are defined in a file called tmtypesw.c in the $TUXDIR/lib directory. Only buffer types defined in tmtypesw.c are known to your client and server programs. You can edit the tmtypesw.c file to add or remove buffer 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 Buffer" on page 3-16

- "Using an FML Typed Buffer" on page 3-22
- "Using an XML Typed Buffer" on page 3-26
- `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*

Allocating a Typed Buffer

Initially, no buffers are associated with a client process. Before a message can be sent, a client process must allocate a buffer of a supported type to carry a message. A typed buffer is allocated using the `tpalloc(3c)` function, as follows:

```
char*
tpalloc(char *type, char *subtype, long size)
```

The following table describes the arguments to the `tpalloc()` function.

Table 3-2 tpalloc() Function Arguments

Argument	Description
type	Pointer to a valid typed buffer.
subtype	Pointer to the name of a subtype being specified (in the view description file) for a `VIEW`, `VIEW32`, or `X_COMMON` typed buffer. In the cases where a *subtype* is not relevant, assign the NULL value to this argument.

Argument	Description
size	Size of the buffer. The BEA Tuxedo system automatically associates a default buffer size with all typed buffers except CARRAY, X_OCTET, and XML, which require that you specify a size, so that the end of the buffer can be identified. For all typed buffers other than CARRAY, X_OCTET, and XML, if you specify a value of zero, the BEA Tuxedo system uses the default associated with that typed buffer. If you specify a size, the BEA Tuxedo system assigns the larger of the following two values: the specified size or the default size associated with that typed buffer. The default size for all typed buffers other than STRING, CARRAY, X_OCTET, and XML is 1024 bytes. The default size for STRING typed buffers is 512 bytes. There is no default value for CARRAY, X_OCTET, and XML; for these typed buffers you must specify a size value greater than zero. If you do not specify a size, the argument defaults to 0. As a result, the tpalloc() function returns a NULL pointer and sets tperrno to TPEINVAL.

The VIEW, VIEW32, X_C_TYPE, and X_COMMON typed buffers require the *subtype* argument, as shown in the following example.

Listing 3-1 Allocating a VIEW Typed Buffer

```
struct aud *audv;  /* pointer to aud view structure */
 . . .
audv = (struct aud *) tpalloc("VIEW", "aud", sizeof(struct aud));
 . . .
```

The following example shows how to allocate an FML typed buffer. Note that a value of NULL is assigned to the *subtype* argument.

Listing 3-2 Allocating an FML Typed Buffer

```
FBFR *fbfr;  /* pointer to an FML buffer structure */
 . . .
fbfr = (FBFR *)tpalloc("FML", NULL, Fneeded(f, v))
 . . .
```

The following example shows how to allocate a CARRAY typed buffer, which requires that a *size* value be specified.

Listing 3-3 Allocating a CARRAY Typed Buffer

```
char *cptr;
long casize;
. . .
casize = 1024;
cptr = tpalloc("CARRAY", NULL, casize);
. . .
```

Upon success, the tpalloc() function returns a pointer of type char. For types other than STRING and CARRAY, you should cast the pointer to the proper C structure or FML pointer.

If the tpalloc() function encounters an error, it returns the NULL pointer. The following list provides examples of error conditions:

- Failure to specify a *size* value for a CARRAY, X_OCTET, or XML typed buffer
- Failure to specify a *type* (or *subtype* in the case of VIEW)
- Specifying a *type* that is not known to the system
- Failure to join the application before attempting allocation

For a complete list of error codes and explanations of them, refer to tpalloc(3c) in the *BEA Tuxedo ATMI C Function Reference.*

The following listing shows how to allocate a STRING typed buffer. In this example, the associated default size is used as the value of the *size* argument to tpalloc().

Listing 3-4 Allocating a STRING Buffer

```
char *cptr;
 . . .
cptr = tpalloc("STRING", NULL, 0);
. . .
```

See Also

- “Putting Data in a Buffer” on page 3-9
- “Resizing a Typed Buffer” on page 3-11
- `tpalloc(3c)` in the *BEA Tuxedo ATMI C Function Reference*

Putting Data in a Buffer

Once you have allocated a buffer, you can put data in it.

In the following example, a `VIEW` typed buffer called `aud` is created with three members (fields). The three members are `b_id`, the branch identifier taken from the command line (if provided); `balance`, used to return the requested balance; and `ermsg`, used to return a message to the status line for the user. When `audit` is used to request a specific branch balance, the value of the `b_id` member is set to the branch identifier to which the request is being sent, and the `balance` and `ermsg` members are set to zero and the NULL string, respectively.

Listing 3-5 Putting Data in a Message Buffer - Example 1

```
...
audv = (struct aud *)tpalloc("VIEW", "aud", sizeof(struct aud));

/* Prepare aud structure */

audv->b_id = q_branchid;
audv->balance = 0.0;
(void)strcpy(audv->ermsg, "");
...
```

When `audit` is used to query the total bank balance, the total balance at each site is obtained by a call to the `BAL` server. To run a query on each site, a representative branch identifier is specified. Representative branch identifiers are stored in an array named `sitelist[]`. Hence, the `aud` structure is set up as shown in the following example.

Listing 3-6 Placing Data in a Message Buffer - Example 2

```
...
/* Prepare aud structure */

audv->b_id = sitelist[i];/* routing done on this field */
audv->balance = 0.0;
(void)strcpy(audv->ermsg, "");
...
```

The process of putting data into a `STRING` buffer is illustrated in the "Resizing a Buffer" on page 3-12 listing.

See Also

- "Allocating a Typed Buffer" on page 3-6
- "Resizing a Typed Buffer" on page 3-11
- `tpalloc(3c)` in the *BEA Tuxedo ATMI C Function Reference*

Resizing a Typed Buffer

You can change the size of a buffer allocated with `tpalloc()` by using the `tprealloc(3c)` function as follows:

```
char*
tprealloc(char *ptr, long size)
```

The following table describes the arguments to the `tprealloc()` function.

Table 3-3 tprealloc() Function Arguments

Argument	Description
ptr	Pointer to the buffer that is to be resized. This pointer must have been allocated originally by a call to `tpalloc()`. If it was not, the call fails and `tperrno(5)` is set to `TPEINVAL` to signify that invalid arguments have been passed to the function.
size	Long integer specifying the new size of the buffer.

The pointer returned by `tprealloc()` points to a buffer of the same type as the original buffer. You must use the returned pointer to reference the resized buffer because the location of the buffer may have changed.

When you call the `tprealloc()` function to increase the size of the buffer, the BEA Tuxedo system makes new space available to the buffer. When you call the `tprealloc()` function to make a buffer smaller, the system does not actually resize the buffer; instead, it renders the space beyond the specified size unusable. The actual content of the typed buffer remains unchanged. If you want to free up unused space, it is recommended that you copy the data into a buffer of the desired size and then free the larger buffer.

On error, the `tprealloc()` function returns the NULL pointer and sets `tperrno` to an appropriate value. Refer to `tpalloc(3c)` in the *BEA Tuxedo ATMI C Function Reference* for information on error codes.

Warning: If the `tprealloc()` function returns the NULL pointer, the contents of the buffer passed to it may have been altered and may be no longer valid.

The following example shows how to reallocate space for a STRING buffer.

Listing 3-7 Resizing a Buffer

```
#include <stdio.h>
#include “atmi.h”

char instr[100];      /* string to capture stdin input strings */
long s1len, s2len;    /* string 1 and string 2 lengths */
char *s1ptr, *s2ptr;  /* string 1 and string 2 pointers */

main()

{
  (void)gets(instr);              /* get line from stdin */
  s1len = (long)strlen(instr)+1;  /* determine its length */

  join application

  if ((s1ptr = tpalloc(“STRING”, NULL, s1len)) == NULL) {
    fprintf(stderr, “tpalloc failed for echo of: %s\n”, instr);
    leave application
    exit(1);
  }
  (void)strcpy(s1ptr, instr);

  make communication call with buffer pointed to by s1ptr

  (void)gets(instr);       /* get another line from stdin */
  s2len = (long)strlen(instr)+1; /* determine its length */
  if ((s2ptr = tprealloc(s1ptr, s2len)) == NULL) {
    fprintf(stderr, “tprealloc failed for echo of: %s\n”, instr);
    free s1ptr's buffer
    leave application
    exit(1);
  }
  (void)strcpy(s2ptr, instr);

  make communication call with buffer pointed to by s2ptr
  . . .
```

The following example (an expanded version of the previous example) shows how to check for occurrences of all possible error codes.

Listing 3-8 Error Checking for tprealloc()

```
. . .
if ((s2ptr=tprealloc(s1ptr, s2len)) == NULL)
  switch(tperrno) {
  case TPEINVAL:
    fprintf(stderr, "given invalid arguments\n");
    fprintf(stderr, "will do tpalloc instead\n");
      tpfree(s1ptr);
    if ((s2ptr=tpalloc("STRING", NULL, s2len)) == NULL) {
      fprintf(stderr, "tpalloc failed for echo of: %s\n", instr);
     leave application
    exit(1);
    }
    break;
  case TPEPROTO:
    fprintf(stderr, "tried to tprealloc before tpinit;\n");
    fprintf(stderr, "program error; contact product support\n");
     leave application
    exit(1);
  case TPESYSTEM:
    fprintf(stderr,
      "BEA Tuxedo error occurred; consult today's userlog file\n");
     leave application
    exit(1);
  case TPEOS:
    fprintf(stderr, "Operating System error %d
occurred\n",Uunixerr);
    leave application
    exit(1);
  default:
    fprintf(stderr,
      "Error from tpalloc: %s\n", tpstrerror(tperrno));
     break;
}
```

See Also

- "Allocating a Typed Buffer" on page 3-6
- "Putting Data in a Buffer" on page 3-9
- `tprealloc(3c)` in the *BEA Tuxedo ATMI C Function Reference*

Checking for Buffer Type

The tptypes(3c) function returns the type and subtype (if one exists) of a buffer. The tptypes() function signature is as follows:

```
long
tptypes(char *ptr, char *type, char *subtype)
```

The following table describes the arguments to the tptypes() function.

Table 3-4 tptypes() Function Arguments

Argument	Description
ptr	Pointer to a data buffer. This pointer must have been originally allocated by a call to tpalloc() or tprealloc(), it may not be NULL, and it must be cast as a character type; otherwise, the tptypes() function reports an invalid argument error.
type	Pointer to the type of the data buffer. *type* is of character type.
subtype	Pointer to the subtype of the data buffer, if one exists. *subtype* is of character type. For all types other than VIEW, VIEW32, X_C_TYPE, and X_COMMON, upon return the *subtype* parameter points to a character array containing the NULL string.

Upon success, the tptypes() function returns the length of the buffer in the form of a long integer.

In the event of an error, tptypes() returns a value of -1 and sets tperrno(5) to the appropriate error code. For a list of these error codes, refer to the "Introduction to the C Language Application-to-Transaction Monitor Interface," and tpalloc(3c) in the *BEA Tuxedo ATMI C Function Reference*.

You can use the size value returned by tptypes() upon success to determine whether the default buffer size is large enough to hold your data, as shown in the following example.

Listing 3-9 Getting Buffer Size

```
. . .
iptr = (FBFR *)tpalloc("FML", NULL, 0);
ilen = tptypes(iptr, NULL, NULL);
. . .
if (ilen < mydatasize)
     iptr=tprealloc(iptr, mydatasize);
```

See Also

- "Allocating a Typed Buffer" on page 3-6
- `tptypes(3c)` in the *BEA Tuxedo ATMI C Function Reference*

Freeing a Typed Buffer

The `tpfree(3c)` function frees a buffer allocated by `tpalloc()` or reallocated by `tprealloc()`. The `tpfree()` function signature is as follows:

```
void
tpfree(char *ptr)
```

The `tpfree()` function takes only one argument, *`ptr`*, which is described in the following table.

Table 3-5 tpfree() Function Argument

Argument	Description
`ptr`	Pointer to a data buffer. This pointer must have been allocated originally by a call to `tpalloc()` or `tprealloc()`, it may not be NULL, and it must be cast as a character type; otherwise, the function returns without freeing anything or reporting an error condition.

When freeing an FML32 buffer using tpfree(), the routine recursively frees all embedded buffers to prevent memory leaks. In order to preserve the embedded buffers, you should assign the associated pointer to NULL before issuing the tpfree() routine. When *ptr* is NULL, no action occurs.

The following example shows how to use the tpfree() function to free a buffer.

Listing 3-10 Freeing a Buffer

```
struct aud *audv;  /* pointer to aud view structure */
. . .
audv = (struct aud *)tpalloc("VIEW", "aud", sizeof(struct aud));
. . .
tpfree((char *)audv);
```

See Also

- "Allocating a Typed Buffer" on page 3-6
- "Resizing a Typed Buffer" on page 3-11
- tpfree(3c) in the *BEA Tuxedo ATMI C Function Reference*

Using a VIEW Typed Buffer

There are two kinds of VIEW typed buffers. The first, FML VIEW, is a C structure generated from an FML buffer. The second is simply an independent C structure.

The reason for converting FML buffers into C structures and back again (and the purpose of the FML VIEW typed buffers) is that while FML buffers provide data-independence and convenience, they incur processing overhead because they must be manipulated using FML function calls. C structures, while not providing flexibility, offer the performance required for lengthy manipulations of buffer data. If you need to perform a significant amount of data manipulation, you can improve

performance by transferring fielded buffer data to C structures, operating on the data using normal C functions, and then converting the data back to the FML buffer for storage or message transmission.

For more information on the FML typed buffer and FML file conversion, refer to the *BEA Tuxedo ATMI FML Function Reference*.

To use VIEW typed buffers, 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, the BEA Tuxedo view compiler. Specify the resulting header file in the #include statement for your application program.

Setting Environment Variables for a VIEW Typed Buffer

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

Table 3-6 Environment Variables for a VIEW Typed Buffer

Environment Variable	Description
FIELDTBLS or FIELDTBLS32	Comma-separated list of field table filenames for FML or FML32 typed buffers. 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 buffers. 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 buffer, you must define the C record in a view description file. The view description file includes, a view for each entry, a view that describes the characteristic C structure mapping and the potential FML conversion pattern. The name of the view corresponds to the name of the C language structure.

The following format is used for each structure 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 C structure.

Table 3-7 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 C structure.
fbname	If you will be using the FML-to-VIEW or VIEW-to-FML conversion functions, 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 NULL value ■ S—one-way mapping from fielded buffer to structure ■ F—one-way mapping from structure to fielded buffer ■ N—zero-way mapping ■ C—generate additional field for associated count member (ACM) ■ L—hold number of bytes transferred for STRING and CARRAY

Table 3-7 View Description File Fields (Continued)

Field	Description
size	For STRING and CARRAY buffer types, specifies the maximum length of the value. This field is ignored for all other buffer types.
null	User-specified NULL value, or minus sign (-) to indicate the default value for a field. NULL values are used in VIEW typed buffers to indicate empty C structure members. The default NULL value for all numeric types is 0 (0.0 for dec_t). For character types, the default NULL value is '\0'. For STRING and CARRAY types, the default NULL value is " ". Constants used, by convention, as escape characters can also be used to specify a NULL 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 char NULL values in double or single quotes. The view compiler does not accept unescaped quotes within a user-specified NULL value. You can also specify the keyword NONE in the NULL field of a view member description, which means that there is no NULL 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 buffer. 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 3-11 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 3-12 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 buffer, run the viewc 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 myview.v
```

Note: To compile a VIEW32 typed buffer, run the viewc32 command.

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

```
viewc -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
- 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 header file created by viewc.

Listing 3-13 Header File Created Using the VIEW Compiler

```
struct MYVIEW {
  float   float1;
  double  double1;
  long    long1;
  short   short1;
  int     int1;
  dec_t   dec1;
  char    char1;
  char    string1[20];
  unsigned short L_carray1[2];    /* length array of carray1 */
  short   C_carray1;              /* count of carray1 */
```

```
  char    carray1[2][20];
};
```

The same header file is created for FML-dependent and independent VIEWs.

In order to use a `VIEW` typed buffer in client programs or service subroutines, you must specify the header file in the application `#include` statements.

See Also

- "Using an FML Typed Buffer" on page 3-22
- "Using an XML Typed Buffer" on page 3-26
- `viewc, viewc32(1)` in the *BEA Tuxedo Command Reference*

Using an FML Typed Buffer

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

- Set the appropriate environment variables.
- Describe the potential fields in an FML field table.
- Create an `FML` header file and specify the header file in a `#include` statement in the application.

`FML` functions are used to manipulate typed buffers, including those that convert fielded buffers 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` functions, refer to the *BEA Tuxedo ATMI FML Function Reference*.

Setting Environment Variables for an FML Typed Buffer

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

Table 3-8 FML Typed Buffer Environment Variables

Environment Variable	Description
FIELDTBLS or FIELDTBLS32	Comma-separated list of field table filenames for FML or FML32 typed buffers, 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 buffers and/or FML-dependent VIEWs are used. A field table file maps the logical name of a field in an FML buffer 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 3-9 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 buffer, 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 3-14 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    -       -
```

Creating an FML Header File

In order to use an FML typed buffer 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 buffers, use the mkfldhdr32 command.

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

Listing 3-15 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 Buffer" on page 3-16
- "Using an XML Typed Buffer" on page 3-26
- `mkfldhdr, mkfldhdr32(1)` in the *BEA Tuxedo Command Reference*

Using an XML Typed Buffer

XXML buffers enable BEA Tuxedo applications to use XML for exchanging data within and between applications. BEA Tuxedo applications can send and receive simple XML buffers, and route those buffers to the appropriate servers. All logic for dealing with 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

The programming model for the XML buffer type is similar to that for the CARRAY buffer type: you must specify the length of the buffer with the `tpalloc()` function. The maximum supported size of an XML document is 4 GB.

Formatting and filtering for Events processing (which are supported when a `STRING` buffer type is used) are not supported for the `XML` buffer type. Therefore, the `_tmfilter` and `_tmformat` function pointers in the buffer type switch for `XML` buffers are set to NULL.

The `XML` parser in the BEA Tuxedo system performs the following functions:

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

Data-dependent routing is supported for `XML` buffers. 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 Buffer" on page 3-16
- "Using an FML Typed Buffer" on page 3-22

Customizing a Buffer

You may find that the buffer types supplied by the BEA Tuxedo system do not meet your needs. For example, perhaps your application uses a data structure that is not flat, but has pointers to other data structures, such as a parse tree for an SQL database query. To accommodate unique application requirements, the BEA Tuxedo System supports customized buffers.

To customize a buffer, you need to identify the following characteristics.

Table 3-10 Custom Buffer Type Characteristics

Characteristic	Description
Buffer type	Name of the buffer type, specified by a string of up to eight characters.
Buffer subtype	Name of the buffer subtype, specified by a string of up to 16 characters. The system uses a subtype to identify different processing requirements for buffers of a given type. When the wildcard character (*) is specified as the subtype value, all buffers of a given type can be processed using the same generic routine. Any buffers for which a subtype is defined must appear before the wildcard in the list, in order to be processed correctly.
Default size	Minimum size of the associated buffer type that can be allocated or reallocated. For buffer types that have a value greater than zero and that are sized appropriately, you can specify a buffer size of zero when allocating or reallocating a buffer to use this default size.

The following table defines the list of routines that you may need to specify for each buffer type. If a particular routine is not applicable, you can simply provide a NULL pointer; the BEA Tuxedo system uses default processing, as necessary.

Table 3-11 Custom Buffer Type Routines

Routine	Description
Buffer initialization	Initializes a newly allocated typed buffer.
Buffer reinitialization	Reinitializes a typed buffer. This routine is called after a buffer has been reallocated (that is, assigned a new size).
Buffer uninitialization	Uninitializes a typed buffer. This routine is called just before a typed buffer is freed.
Buffer presend	Prepares the typed buffer for sending. This routine is called before a typed buffer is sent as a message to another client or server. It returns the length of the data to be transmitted.
Buffer postsend	Returns the typed buffer to its original state. This routine is called after the message is sent.
Buffer postreceive	Prepares the typed buffer once it has been received by the application. It returns the length of the application data.
Encode/decode	Performs all the encoding and decoding necessary for the buffer type. A request to encode or decode is passed to the routine, along with input and output buffers and lengths. The format used for encoding is determined by the application and, as with the other routines, it may be dependent on the buffer type.
Routing	Specifies the routing information. This routine is called with a typed buffer, the length of the data for that buffer, a logical routing name configured by an administrator, and a target service. Based on this information, the application must select the server group to which the message should be sent or indicate that the message is not needed.
Filter	Specifies filter information. This routine is called to evaluate an expression against a typed buffer and to return a match if it finds one. If the typed buffer is `VIEW` or `FML`, the `FML` Boolean expressions are used. This routine is used by the EventBroker to evaluate matches for events.
Format	Specifies a printable string for a typed buffer.

Defining Your Own Buffer Types

The application programmer is responsible for the code that manipulates buffers, which allocates and frees space, and sends and receives messages. For applications in which the default buffer types do not meet the needs of the application, other buffer types can be defined, and new routines can be written and then incorporated into the buffer type switch.

To define other buffer types, complete the following steps:

1. Code any switch element routines that may be required.
2. Add your new types and the names of your buffer management modules to `tm_typesw`.
3. Build a new shared object or a DLL. The shared object or DLL must contain your updated buffer type switch and associated functions.
4. Install your new shared object or DLL so that all servers, clients, and executables provided by the BEA Tuxedo system are loaded dynamically at run time.

If your application is using static libraries and you are providing a customized buffer type switch, then you must build a custom server to link in your new type switch. For details, see `buildwsh` (1), `TMQUEUE` (5), or `TMQFORWARD` (5).

The rest of the sections in this topic address the steps listed in the preceding procedure to define a new buffer type in a shared-object or DLL environment. First, however, let's look at the buffer switch that is delivered with the BEA Tuxedo system software. The following listing shows the switch delivered with the system.

Listing 3-16 Default Buffer Type Switch

```
#include <stdio.h>
#include <tmtypes.h>

/* Initialization of the buffer type switch */
static struct tmtype_sw_t tm_typesw[] = {
{
"CARRAY",          /* type */
"",                /* subtype */
0                  /* dfltsize */
},
{
```

```
“STRING",          /* type */
“",                /*subtype */
512,               /* dfltsize */
NULL,              /* initbuf */
NULL,              /* reinitbuf */
NULL,              /* uninitbuf */
_strpresend,       /* presend */
NULL,              /* postsend */
NULL,              /* postrecv */
_strencdec,        /* encdec */
NULL,              /* route */
NULL,              /* filter */
NULL               /* format */
},
{
“FML",             /* type */
“",                /* subtype */
1024,              /* dfltsize */
_finit,            /* initbuf */
_freinit,          /* reinitbuf */
_funinit,          /* uninitbuf */
_fpresend,         /* presend */
_fpostsend,        /*postsend */
_fpostrecv,        /* postrecv */
_fencdec,          /* encdec */
_froute,           /*route */
_ffilter,          /* filter */
_fformat           /* format */
},
{
“FML32",           /* type */
“",                /* subtype */
1024,              /* dfltsize */
_finit32,          /* initbuf */
_freinit32,        /* reinitbuf */
_funinit32,        /* uninitbuf */
_fpresend32,       /* presend */
_fpostsend32,      /* postsend */
_fpostrecv32,      /* postrecv */
_fencdec32,        /* encdec */
_froute32,         /* route */
_ffilter32,        /* filter */
_fformat32         /* format */
},
{
"VIEW",            /* type */
"*",               /* subtype */
1024,              /* dfltsize */
_vinit,            /* initbuf */
```

```
_vreinit,          /* reinitbuf */
NULL,              /* uninitbuf */
_vpresend,         /* presend */
NULL,              /* postsend */
NULL,              /* postrecv */
_vencdec,          /* encdec */
_vroute,           /* route */
_vfilter,          /* filter */
_vformat           /* format */
},
{
"VIEW32",          /* type */
"*"                /* subtype */
1024,              /* dfltsize */
_vinit32,          /* initbuf */
_vreinit32,        /* reinitbuf */
NULL,              /* uninitbuf */
_vpresend32,       /* presend */
NULL,              /* postsend */
NULL,              /* postrecv */
_vencdec32,        /* encdec */
_vroute32,         /* route */
_vfilter32,        /* filter */
_vformat32         /* format */
},
{
"X_OCTET",         /* type */
"",                /* subtype */
0,                 /* dfltsize */
},
{
"'X','_','C','_','T','Y','P','E'",     /* type */
"*",               /* subtype */
1024,              /* dfltsize */
_vinit,            /* initbuf */
_vreinit,          /* reinitbuf */
NULL,              /* uninitbuf */
_vpresend,         /* presend */
NULL,              /* postsend */
NULL,              /* postrecv */
_vencdec,          /* encdec */
_vroute,           /* route */
_vfilter,          /* filter */
_vformat           /* format */
},
{
"'X','_','C','O','M','M','O','N'",   /* type */
"*",               /* subtype */
1024,              /* dfltsize */
```

```
_vinit,             /* initbuf */
_vreinit,           /* reinitbuf */
NULL,               /* uninitbuf */
_vpresend,          /* presend */
NULL,               /* postsend */
NULL,               /* postrecv */
_vencdec,           /* encdec */
_vroute,            /* route */
_vfilter,           /* filter */
_vformat            /* format */
},
{
"XML",              /* type */
"*",                /* subtype */
0,                  /* dfltsize */
NULL,               /* _xinit - not available */
NULL,               /* _xreinit - not available */
NULL,               /* _xuninit - not available */
NULL,               /* _xpresend - not available */
NULL,               /* _xpostsend - not available */
NULL,               /* _xpostrecv - not available */
NULL,               /* _xencdec - not available */
_xroute,            /* _xroute */
NULL,               /* filter - not available */
NULL                /* format - not available */
},
{
""
}
};
```

For a better understanding of the preceding listing, consider the declaration of the buffer type structure that is shown in the following listing.

Listing 3-17 Buffer Type Structure

```
/* The following definitions are in $TUXDIR/include/tmtypes.h */


#define TMTYPELEN      8
#define TMSTYPELEN    16

struct tmtype_sw_t {
  char type[TMTYPELEN];     /* type of buffer */
```

```
  char subtype[TMSTYPELEN];  /* sub-type of buffer */
  long dfltsize;   /* default size of buffer */
  /* buffer initialization function pointer */
  int (_TMDLLENTRY *initbuf) _((char _TM_FAR *, long));
  /* buffer re-initialization function pointer */
  int (_TMDLLENTRY *reinitbuf) _((char _TM_FAR *, long));
  /* buffer un-initialization function pointer */
  int (_TMDLLENTRY *uninitbuf) _((char _TM_FAR *, long));
  /* pre-send buffer manipulation func pointer */
  long (_TMDLLENTRY *presend) _((char _TM_FAR *, long, long));
  /* post-send buffer manipulation func pointer */
  void (_TMDLLENTRY *postsend) _((char _TM_FAR *, long, long));
  /* post-receive buffer manipulation func pointer*/
  long (_TMDLLENTRY *postrecv) _((char _TM_FAR *, long, long));
  /* encode/decode function pointer */
  long (_TMDLLENTRY *encdec) _((int, char _TM_FAR *, long,
    char _TM_FAR *, long));
  /* routing function pointer */
  int (_TMDLLENTRY *route) _((char _TM_FAR *, char _TM_FAR *,
    char _TM_FAR *, long, char _TM_FAR *));
  /* buffer filtering function pointer */
  int (_TMDLLENTRY *filter) _((char _TM_FAR *, long, char _TM_FAR *,
    long));
  /* buffer formatting function pointer */
  int (_TMDLLENTRY *format) _((char _TM_FAR *, long, char _TM_FAR *,
    char _TM_FAR *, long));
  /* this space reserved for future expansion */
  void (_TMDLLENTRY *reserved[10]) _((void));
};
```

The listing for the default buffer type switch shows the initialization of the buffer type switch. The nine default buffer types are shown, followed by a field for naming a subtype. Except for the `VIEW` (and equivalently `X_C_TYPE` and `X_COMMON`) type, subtype is NULL. The subtype for `VIEW` is given as ‘‘*”, which means that the default `VIEW` type puts no constraints on subtypes; all subtypes of type `VIEW` are processed in the same manner.

The next field gives the default (minimum) size of the buffer. For the `CARRAY` (and equivalently `X_OCTET`) type this is given as 0, which means that the routine that uses a `CARRAY` buffer type must `tpalloc()` enough space for the expected `CARRAY`.

For the other types, the BEA Tuxedo system allocates (with a `tpalloc()` call) the space shown in the `dfltsize` field of the entry (unless the size argument of `tpalloc()` specifies a larger size).

The remaining eight fields of entries in the buffer type switch contain the names of switch element routines. These routines are described in the `buffer(3c)` page in the *BEA Tuxedo C Function Reference*. The name of a routine provides a clue to the purpose of the routine. For example, `_fpresend` on the `FML` type is a pointer to a routine that manipulates the buffer before sending it. If no presend manipulation is needed, a NULL pointer may be specified. NULL means no special handling is required; the default action should be taken. See `buffer(3c)` for details.

It is particularly important that you notice the NULL entry at the end of the switch. Any changes that are made must always leave the NULL entry at the end of the array.

Coding Switch Element Routines

Presumably an application that is defining new buffer types is doing so because of a special processing need. For example, let's assume the application has a recurring need to compress data before sending a buffer to the next process. The application could write a presend routine. The declaration for the presend routine is shown in the following listing.

Listing 3-18 Semantics of the Presend Switch Element

```
long
presend(ptr, dlen, mdlen)
char *ptr;

long dlen, mdlen;
```

- *`ptr`* is a pointer to the application data buffer.
- *`dlen`* is the length of the data as passed into the routine.
- *`mdlen`* is the size of the buffer in which the data resides.

The data compression that takes place within your presend routine is the responsibility of the system programmer for your application.

On completion the routine should return the new, hopefully shorter length of the data to be sent (in the same buffer), or a `-1` to indicate failure.

The name given to your version of the presend routine can be any identifier accepted by the C compiler. For example, suppose we name it `_mypresend`.

If you use our `_mypresend` compression routine, you will probably also need a corresponding `_mypostrecv` routine to decompress the data at the receiving end. Follow the template shown in the `buffer`(3c) entry in the *BEA Tuxedo C Function Reference*.

Adding a New Buffer Type to tm_typesw

After the new switch element routines have been written and successfully compiled, the new buffer type must be added to the buffer type switch. To do this task, we recommend making a copy of `$TUXDIR/lib/tmtypesw.c` (the source code for the default buffer type switch). Give your copy a name with a `.c` suffix, such as `mytypesw.c`. Add the new type to your copy. The name of the type can be up to 8 characters in length. Subtype can be null (`""`) or a string of up to 16 characters. Enter the names of your new switch element routines in the appropriate locations, including the `extern` declarations. The following listing provides an example.

Listing 3-19 Adding a New Type to the Buffer Switch

```
#include <stdio.h>
#include <tmtypes.h>

/* Customized the buffer type switch */

static struct tmtype_sw_t tm_typesw[] = {
{
"SOUND",          /* type */
“",               /* subtype */
50000,            /* dfltsize */
snd_init,         /* initbuf */
snd_init,         /* reinitbuf */
NULL,             /* uninitbuf */

snd_cmprs,        /* presend */
snd_uncmprs,      /* postsend */
snd_uncmprs       /* postrecv */
},
{
"FML",            /* type */
"",               /* subtype */
1024,             /* dfltsize */
```

```
_finit,           /* initbuf */
_freinit,         /* reinitbuf */
_funinit,         /* uninitbuf */
_fpresend,        /* presend */
_fpostsend,       /* postsend */
_fpostrecv,       /* postrecv */
_fencdec,         /* encdec */
_froute,          /* route */
_ffilter,         /* filter */
_fformat          /* format */
},
{
""
}
};
```

In the previous listing, we added a new type: `SOUND`. We also removed the entries for `VIEW`, `X_OCTET`, `X_COMMON`, and `X_C_TYPE`, to demonstrate that you can remove any entries that are not needed in the default switch. Note that the array still ends with the NULL entry.

An alternative to defining a new buffer type is to redefine an existing type. Suppose, for the sake of argument, that the data compression for which you defined the buffer type `MYTYPE` was performed on strings. You could substitute your new switch element routines, `_mypresend` and `_mypostrecv`, for the two `_dfltblen` routines in type `STRING`.

Compiling and Linking Your New tm_typesw

To simplify installation, the buffer type switch is stored in a shared object.

Note: On some platforms the term “shared library” is used instead of “shared object.” On the Windows 2000 platform a “dynamic link library” is used instead of a “shared object.” For the purposes of this discussion, however, the functionality implied by all three terms is equivalent, so we use only one term.

This section describes how to make all BEA Tuxedo processes in your application aware of the modified buffer type switch. These processes include application servers and clients, as well as servers and utilities provided by the BEA Tuxedo system.

1. Copy and modify `$TUXDIR/lib/tmtypesw.c`, as described in "Adding a New Buffer Type to tm_typesw" on page 3-36. If additional functions are required, store them in either `tmtypesw.c` or a separate C source file.
2. Compile `tmtypesw.c` with the flags required for shared objects.
3. Link together all object files to produce a shared object.
4. Copy `libbuft.so.71` from the current directory to a directory in which it will be visible to applications, and processed before the default shared object supplied by the BEA Tuxedo system. We recommend using one of the following directories: `$APPDIR`, `$TUXDIR/lib`, or `$TUXDIR/bin` (on a Windows 2000 platform).

Different platforms assign different names to the buffer type switch shared object, to conform to operating system conventions.

Table 3-12 OS-specific Names for the Buffer Type Switch Shared Object

On This Platform . . .	The Name of the Buffer Type Switch Shared Object Is . . .
UNIX System (most SVR4)	`libbuft.so.71`
HP-UX	`libbuft.sl`
Sun OS	`libbuft.so.71`
Windows (16-bit)	`wbuft.dll`
Windows (32-bit)	`wbuft32.dll`
OS/2 (16-bit)	`obuft.dll`
OS/2 (32-bit)	`obuft.dll`

Please refer to the software development documentation for your platform for instructions on building a shared object library.

As an alternative, it is possible to statically link a new buffer type switch in every client and server process, but doing so is more error-prone and not as efficient as building a shared object library.

Compiling and Linking Your New tm_typesw for a 16-bit Windows Platform

If you have modified `tmtypesw.c` on a Windows platform, as described in "Compiling and Linking Your New tm_typesw" on page 3-37, then you can use the commands shown in the following sample code listing to make the modified buffer type switch available to your application.

Listing 3-20 Sample Code in Microsoft Visual C++

```
CL -AL -I..\e\|sysinclu -I..\e\|include -Aw -G2swx -Zp -D_TM_WIN
-D_TMDLL -Od -c TMTYPESW.C
LINK /CO /ALIGN:16 TMTYPESW.OBJ, WBUFT.DLL, NUL, WTUXWS /SE:250 /NOD
/NOE LIBW LDLLCEW, WBUFT.DEF
RC /30 /T /K WBUFT.DLL
```

Data Conversion

The purpose of the `TYPE` parameter in the `MACHINES` section of the configuration file is to group together machines that have the same form of data representation (and use the same compiler) so that data conversion is done on messages going between machines of different `TYPE`s. For the default buffer types, data conversion between unlike machines is transparent to the user (and to the administrator and programmer, for that matter).

If your application defines new buffer types for messages that move between machines with different data representation schemes, you must also write new encode/decode routines to be incorporated into the buffer type switch. When writing your own data conversion routines, keep the following guidelines in mind:

- You should use the semantics of the `_tmencdec` routine shown on the `buffer(3c)` page in the *BEA Tuxedo ATMI C Function Reference*; that is, you should code your routine so that it uses the same arguments and returns the same values on success or failure as the `_tmencdec` routine. When defining new buffer types, follow the procedure provided in "Defining Your Own Buffer Types" on page 3-30 for building servers with services that will use your new buffer type.

The encode/decode routines are called only when the BEA Tuxedo system determines that data is being sent between two machines that are not of the same TYPE.

CHAPTER

4 Writing Clients

This topic includes the following sections:

- Joining an Application
- Using Features of the TPINIT Typed Buffer
- 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 the `tpinit(3c)` function with the following signature:

```
int
tpinit (TPINIT *tpinfo)
```

A client joins an application implicitly by issuing a service request (or any ATMI function) without first calling the `tpinit()`function. In this case, the `tpinit()` function is called by the BEA Tuxedo system on behalf of the client with the `tpinfo` argument set to NULL. The `tpinfo` argument points to a typed buffer with a `TPINIT` type and NULL subtype. The `TPINIT` typed buffer is defined in the `atmi.h` header file and includes the following information:

```
char usrname[MAXTIDENT+2];
char cltname[MAXTIDENT+2];
char passwd[MAXTIDENT+2];
char grpname[MAXTIDENT+2];
long flags;
long datalen;
long data;
```

The following table summarizes the TPINIT data structure fields.

Table 4-1 TPINIT Data Structure Fields

Field	Description
usrname	Name representing the client; used for both broadcast notification and administrative statistics retrieval. The client assigns a value to *usrname* during the call to the tpinit() function. The value is a string of up to MAXTIDENT characters (which defaults to 30 and is configurable by the administrator), and must be terminated by NULL.
cltname	Client name with application-defined semantics: a 30-character NULL-terminated string used for both broadcast notification and administrative statistics retrieval. The client assigns a value to *cltname* during the call to the tpinit() function. The value is a string of up to MAXTIDENT characters (which defaults to 30 and is configurable by the administrator), and must be terminated by NULL. **Note:** The value sysclient is reserved for the *cltname* field.
passwd	Application password in unencrypted format. Used for user authentication. The value is a string of up to 30 characters.
grpname	Associates client with resource manager group. If set to a 0-length string, the client is not associated with a resource manager and is in the default client group. The value of grpname must be the NULL string (0-length string) for Workstation clients. Refer to *Using the ATMI Workstation Component* for more information on Workstation clients.

Field	Description
flags	Indicates both the client-specific notification mechanism and the mode of system access. Controls both multicontext and single-context modes. Refer to "Unsolicited Notification Handling" on page 4-6 or `tpinit()` in the *BEA Tuxedo ATMI C Function Reference* for more information on flags.
datalen	Length of the application-specific data. The buffer type switch entry for the `TPINIT` typed buffer sets this field based on the total size passed in for the typed buffer. The size of the application data is the total size less the size of the `TPINIT` structure itself plus the size of the data placeholder as defined in the structure.
data	Placeholder for variable length data that is forwarded to an application-defined authentication service.

Before it can join the application, the client program must call `tpalloc()` to allocate the `TPINIT` buffer. The following example shows how to allocate a `TPINIT` buffer that will be used to pass eight bytes of application-specific *data* to the `tpinit()` function.

Listing 4-1 Allocating a TPINIT Typed Buffer

```
.
.
.
TPINIT *tpinfo;
.
.
.
if ((tpinfo = (TPINIT *)tpalloc("TPINIT",(char *)NULL,
     TPINITNEED(8))) == (TPINIT *)NULL){
     Error Routine
}
```

Refer to `tpinit()` in the *BEA Tuxedo ATMI C Function Reference* for more information on the `TPINIT` typed buffer.

See Also

- `tpinit(3c)` in the *BEA Tuxedo ATMI C Function Reference*

Using Features of the TPINIT Typed Buffer

The ATMI client must explicitly invoke the `tpinit()` function in order to take advantage of the following features of the `TPINIT` typed buffer:

- 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 the `tpinit()` function via the `tpinfo` buffer argument. 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.

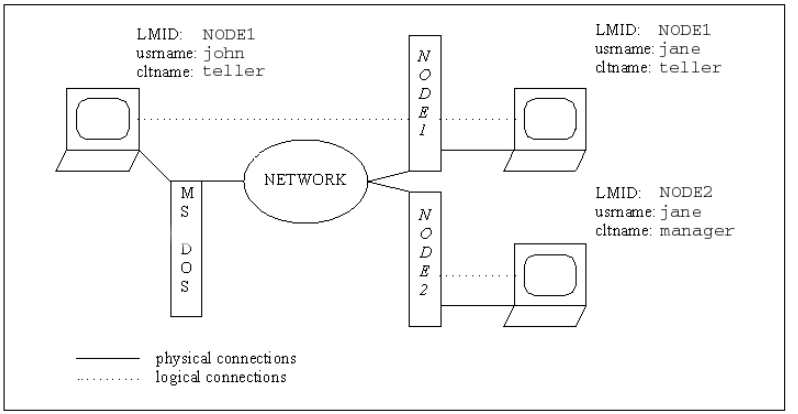
Once a unique identifier for a client process is created:

- 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" on page 8-1 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 function 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()` function 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()`function.

When a client joins an application using the `tpinit()` function, it can control how to handle unsolicited notification messages by defining flags. For client notification, the possible values for `flags` are defined in the following table.

Table 4-2 Client Notification Flags in a TPINIT Typed Buffer

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 function using the tpsetunsol() function, and check for waiting unsolicited messages using the tpchkunsol() function.
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 tperrno(5) is set to TPEINVAL.
TPU_IGN	Ignore unsolicited notification.

Refer to tpinit(3c) in the *BEA Tuxedo ATMI C Function Reference* for more information on the TPINIT typed buffer 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 the tpinit() function. To specify a mode, a client passes one of the following values in the flags field of the TPINIT buffer to the tpinit() function.

Table 4-3 System Access Flags in a TPINIT Typed Buffer

Mode	Description
`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`.
`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 the `TPINIT` buffer.

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 `TPINIT` buffer and evaluated when the client calls `tpinit()` to join the application.

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

You can use the `tpchkauth(3c)` function 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 the `tpchkauth()` function before `tpinit()` 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 the `tpterm(3c)` function. The `tpterm()` function takes no arguments, and returns an integer value that is equal to `-1` on error.

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. Use the following syntax for the `buildclient` command:

`buildclient` *`filename`*`.c -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``.c`	The C 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 C program file (*`file`*`.c`), 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`*	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 C program file (*`file`*`.c`), 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 . . .
`-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 `CC` and `CFLAGS` environment variables to specify an alternative compile command, and to set flags for the compile and link-edit phases, respectively. For more information, refer to "Setting Environment Variables" on page 2-5.

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

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

```
buildclient –o audit –f audit.c
```

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 4-2 Typical Client Process Paradigm

```
main()
{
     check level of security
     call tpsetunsol() to name your handler for TPU_DIP
     get usrname, cltname
     prompt for application password
     allocate a TPINIT buffer
     place values into TPINIT buffer structure members

     if (tpinit((TPINIT *) tpinfo) == -1){
          error routine;
     }

     allocate a message buffer
     while user input exists {
          place user input in the buffer
          make a service call
          receive the reply
          check for unsolicited messages
     }
     free buffers
     . . .
     if (tpterm() == -1){
          error routine;
     }
}
```

On error, `-1` is returned and the application sets the external global variable, `tperrno`, to a value that indicates the nature of the error. `tperrno` is defined in the `atmi.h` header file and documented in `tperrno(5)` in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*. Programmers typically assign an error code to this global variable that reflects the type of error encountered. There is a discussion of the values of `tperrno` in "System Errors" on page 11-1. See "Introduction to the C

Language Application-to-Transaction Monitor Interface" in the *BEA Tuxedo ATMI C Function Reference* for a complete list of error codes that can be returned for each of the ATMI functions.

The following example illustrates how to use the `tpinit()` and `tpterm()` functions. This example is borrowed from, `bankapp`, the sample banking application that is provided with the BEA Tuxedo system.

Listing 4-3 Joining and Leaving an Application

```
#include <stdio.h>          /* UNIX */
#include <string.h>         /* UNIX */
#include <fml.h>            /* BEA Tuxedo System */
#include <atmi.h>           /* BEA Tuxedo System */
#include <Uunix.h>          /* BEA Tuxedo System */
#include <userlog.h>        /* BEA Tuxedo System */
#include "bank.h"           /* BANKING #defines */
#include "aud.h"            /* BANKING view defines */

...

main(argc, argv)
int argc;
char *argv[];

{
     ...
     if (strrchr(argv[0],'/') != NULL)
      proc_name = strrchr(argv[0],'/')+1;
     else
      proc_name = argv[0];
     ...
     /* Join application */
     if (tpinit((TPINIT *) NULL) == -1) {
     (void)userlog("%s: failed to join application\n", proc_name);
        exit(1);
     }
     ...
     /* Leave application */
     if (tpterm() == -1) {
     (void)userlog("%s: failed to leave application\n", proc_name);
      exit(1);
     }
}
```

The previous example shows the client process attempting to join the application with a call to `tpinit()`. If the process encounters an error (that is, if the return code is `-1`), the process writes a descriptive message to the central event log via a call to `userlog()`, which takes arguments similar to the `printf()` C program statement. Refer to `userlog(3c)` in the *BEA Tuxedo ATMI C Function Reference* for more information.

Similarly, when `tpterm()` is called, if an error is encountered, the process writes a descriptive message to the central event log.

CHAPTER

5 Writing Servers

This topic includes the following sections:

- BEA Tuxedo System main( )
- System-supplied Server and Services
- Guidelines for Writing Servers
- Defining a Service
- Example: Checking the Buffer Type
- Example: Checking the Priority of the Service Request
- Terminating a Service Routine
- Advertising and Unadvertising Services
- Building ServersUsing a C++ Compiler

BEA Tuxedo System main( )

To facilitate the development of ATMI servers, the BEA Tuxedo system provides a predefined `main()` routine for server load modules. When you execute the `buildserver` command, the `main()` routine is automatically included as part of the server.

Note: The `main()` routine that the system provides is a closed abstraction; you cannot modify it.

In addition to joining and exiting from an application, the predefined `main()` routine 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 `main()`, 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 function `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`.
  - Allocates a buffer for the request message and 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 buffers and transactions, and handling communication.

Note: Because the system-supplied `main()` accomplishes the work of joining and leaving the application, you should not include calls to the `tpinit()` or `tpterm()` function in your code. If you do, the function encounters an error and returns `TPEPROTO` in `tperrno`. For more information on the `tpinit()` or `tpterm()` function, refer to "Writing Clients" on page 4-1.

System-supplied Server and Services

The `main()` routine 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 functions in an open/close pair must belong to the same set.

In addition to the subroutines described in this topic, the system provides two subroutines called `tpsvrthrinit(3c)` and `tpsvrthrdone(3c)`. For more information, refer to "Programming a Multithreaded and Multicontexted ATMI Application" on page 10-1.

System-supplied Server: AUTHSVR( )

You can use the AUTHSVR(5) server to provide individual client authentication for an application. The tpinit( ) function calls this server when the level of security for the application is TPAPPAUTH.

The service in AUTHSVR looks in the data field of the TPINIT buffer for a user password (not to be confused with the application password specified in the passwd field of the TPINIT buffer). 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, tpinit( ) forwards the data field 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, tpinit( ) encrypts the data before sending it across the network.

System-supplied Services: tpsvrinit( ) Function

When a server is booted, the BEA Tuxedo system main( ) calls tpsvrinit(3c) during its initialization phase, before handling any service requests.

If an application does not provide a custom version of this function within the server, the system uses the default function provided by main( ), 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(3c) function. Refer to “Managing Errors” on page 11-1 for more information on the central event log.

You can use the tpsvrinit( ) function 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() function to start and complete transactions, as described in "Managing Errors" on page 11-1.

Use the following signature to call the tpsvrinit() function:

```
int
tpsvrinit(int argc, char **argv)
```

Receiving Command-line Options

When a server is booted, its first task is to read the server options specified in the configuration file up to the point that it receives an EOF indication. To do so, the server calls the getopt(3) UNIX function. The presence of a double dash (--) on the command line causes the getopt() function to return an EOF. The getopt function places the argv index of the next argument to be processed in the external variable optind. The predefined main() then calls tpsvrinit().

The following code example shows how the tpsvrinit() function is used to receive command-line options.

Listing 5-1 Receiving Command-line Options in tpsvrinit( )

```
tpsvrinit(argc, argv)
int argc;
char **argv;
{
       int c;
       extern char *optarg;
       extern int optind;
       .
       .
       .
       while((c = getopt(argc, argv, "f:x:")) != EOF)
          switch(c){
          .
          .
          .
          }
       .
       .
```

```
        .
}
```

When `main()` calls `tpsvrinit()`, it picks up any arguments that follow the double dash (`--`) on the command line. In the example above, options `f` and `x` each takes an argument, as indicated by the colon. `optarg` points to the beginning of the option argument. The switch statement logic is omitted.

Opening a Resource Manager

The following example illustrates another common use of `tpsvrinit()`: opening a resource manager. The BEA Tuxedo system provides functions to open a resource manager, `tpopen(3c)` and `tx_open(3c)`. It also provides the complementary functions, `tpclose(3c)` and `tx_close(3c)`. Applications that use these functions 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.

Note: If writing a multithreaded server, you must use the `tpsvrthrinit()` function to open a resource manager, as described in "Programming a Multithreaded and Multicontexted ATMI Application" on page 10-1.

These function 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(3c)` function 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( )

```
tpsvrinit()
{

    /* Open database */

    if (tpopen() == -1) {
```

```
        (void)userlog("tpsvrinit: failed to open database: ");
        switch (tperrno) {
          case TPESYSTEM:
             (void)userlog("System error\n");
              break;
          case TPEOS:
             (void)userlog("Unix error %d\n",Uunixerr);
              break;
          case TPEPROTO:
             (void)userlog("Called in improper context\n");
              break;
          case TPERMERR:
             (void)userlog("RM failure\n");
              break;
      }
      return(-1);     /* causes the server to exit */
  }
  return(0);
}
```

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( ) Function

The `tpsvrdone()` function calls `tpclose()` to close the resource manager, similarly to the way `tpsvrinit()` calls `tpopen()` to open it.

Note: If writing a multithreaded server, you must use the `tpsvrthrdone()` command to open a resource manager, as described in “Programming a Multithreaded and Multicontexted ATMI Application” on page 10-1.

Use the following signature to call the `tpsvrdone()` function:

```
void
tpsvrdone()  /* Server termination routine */
```

The `tpsvrdone()` function requires no arguments.

If an application does not define a closing routine for `tpsvrdone()`, the BEA Tuxedo system calls the default routine supplied by `main()`. This routine calls `tx_close()` and `userlog()` to close the resource manager and write to the central event log, respectively. The message sent to the log indicates that the server is about to exit.

`tpsvrdone()` is called after the server has finished processing service requests but before it exits. Because the server is still part of the system, further communication and transactions can take place within the routine, as long as certain rules are followed. These rules are covered in "Managing Errors" on page 11-1.

The following example illustrates how to use the `tpsvrdone()` function to close a resource manager and exit gracefully.

Listing 5-3 Closing a Resource Manager with tpsvrdone( )

```
void
tpsvrdone()
{

     /* Close the database */
     if(tpclose() == -1)
           (void)userlog("tpsvrdone: failed to close database: ");
           switch (tperrno) {
                   case TPESYSTEM:
                           (void)userlog("BEA TUXEDO error\n");
                           break;
                   case TPEOS:
                           (void)userlog("Unix error %d\n",Uunixerr);
                           break;
                   case TPEPROTO:
                           (void)userlog("Called in improper context\n");
                           break;
                   case TPERMERR:
                           (void)userlog("RM failure\n");
                           break;
           }
           return;
    }
    return;
}
```

Guidelines for Writing Servers

Because the communication details are handled by the BEA Tuxedo system `main()` routine, you can concentrate on the application service logic rather than communication implementation. For compatibility with the system-supplied `main()`, 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. Refer to the `tpservice(3c)` reference page in the *BEA Tuxedo ATMI C Function Reference* for more information on these conventions.

- 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 `tpforward()` function. These functions behave similarly to the C language `return` statement except that after they finish executing, control returns to the BEA Tuxedo system's `main()` instead of the calling function.
- 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 `tpforward()`.
- Service routines are invoked with one argument, *`svcinfo`*, which is a pointer to a service information structure (`TPSVCINFO`).

Defining a Service

You must define every service routine as a function that receives one argument consisting of a pointer to a TPSVCINFO structure. The TPSVCINFO structure is defined in the atmi.h header file and includes the following information:

```
char      name[32];
long      flags;
char      *data;
long      len;
int       cd;
int       appkey;
CLIENTID  cltid;
```

The following table summarizes the TPSVCINFO data structure.

Table 5-1 TPSVCINFO Data Structure

Field	Description
name	Specifies, to the service routine, the name used by the requesting process to invoke the service.
flags	Notifies the service if it is in transaction mode or if the caller is expecting a reply. The various ways in which a service can be placed in transaction mode are discussed in "Writing Global Transactions" on page 9-1. The TPTRAN flag indicates that the service is in transaction mode. When a service is invoked through a call to tpcall() or tpacall() with the flags parameter set to TPNOTRAN, the service cannot participate in the current transaction. However, it is still possible for the service to be executed in transaction mode. That is, even when the caller sets the TPNOTRAN communication flag, it is possible for TPTRAN to be set in svcinfo->flags. For an example of such a situation, refer to "Writing Global Transactions" on page 9-1. The *flags* member is set to TPNOREPLY if the service is called by tpacall() with the TPNOREPLY communication flag set. If a called service is part of the same transaction as the calling process, it must return a reply to the caller.

Field	Description
data	Pointer to a buffer that was previously allocated by `tpalloc()` within the `main()`. This buffer is used to receive request messages. However, it is recommended that you also use this buffer to send back reply messages or forward request messages.
len	Contains the length of the request data that is in the buffer referenced by the *data* field.
cd	For conversational communication, specifies the connection descriptor.
appkey	Reserved for use by the application. If application-specific authentication is part of your design, the application-specific authentication server, which is called when a client joins the application, should return a client authentication key as well as an indication of success or failure. 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 a service, the client has already been authenticated. However, the *appkey* field can be used within a service to identify the user invoking the service or some other parameters associated with the user. If this field is not used, the system assigns it a default value of `-1`.
cltid	Structure of type `CLIENTID` used by the system to carry the identification of the client. You should not modify this structure.

When the *data* field in the `TPSVCINFO` structure is being accessed by a process, the following buffer types must agree:

- Type of the request buffer passed by the calling process
- Type of the corresponding buffer code defined within the called service
- Type of the associated buffer type defined for the called service in the configuration file

The following example illustrates a typical service definition. This code is borrowed from the `ABAL` (account balance) service routine that is part of the banking application provided with the BEA Tuxedo software. `ABAL` is part of the `BAL` server.

Listing 5-4 Typical Service Definition

```
#include <stdio.h>      /* UNIX */
#include <atmi.h>       /* BEA Tuxedo System */
#include <sqlcode.h>    /* BEA Tuxedo System */
#include "bank.flds.h"  /* bankdb fields */
#include "aud.h"        /* BANKING view defines */

EXEC SQL begin declare section;
static long branch_id;  /* branch id */
static float bal;       /* balance */
EXEC SQL end declare section;

/*
 * Service to find sum of the account balances at a SITE
 */

void
#ifdef __STDC__
ABAL(TPSVCINFO *transb)

#else

ABAL(transb)
TPSVCINFO *transb;
#endif

{
     struct aud *transv;         /* view of decoded message */

     /* Set pointer to TPSVCINFO data buffer */

     transv = (struct aud *)transb->data;

     set the consistency level of the transaction

     /* Get branch id from message, do query */

     EXEC SQL declare acur cursor for
      select SUM(BALANCE) from ACCOUNT;
     EXEC SQL open acur;         /* open */
     EXEC SQL fetch acur into :bal;   /* fetch */
     if (SQLCODE != SQL_OK) {      /* nothing found */
     (void)strcpy (transv->ermsg,"abal failed in sql aggregation");
      EXEC SQL close acur;
      tpreturn(TPFAIL, 0, transb->data, sizeof(struct aud), 0);
     }
     EXEC SQL close acur;
```

```
     transv->balance = bal;
     tpreturn (TPSUCCESS, 0, transb->data, sizeof(struct aud), 0);
}
```

In the preceding example, the application allocates a request buffer on the client side by a call to `tpalloc()` with the *`type`* parameter set to `VIEW` and the *`subtype`* set to `aud`. The `ABAL` service is defined as supporting the `VIEW` typed buffer. The `BUFTYPE` parameter is not specified for `ABAL` and defaults to `ALL`. The `ABAL` service allocates a buffer of the type `VIEW` and assigns the `data` member of the `TPSVCINFO` structure that was passed to the `ABAL` subroutine to the buffer pointer. The `ABAL` server retrieves the appropriate data buffer by accessing the corresponding *`data`* member, as illustrated in the preceding example.

Note: After the buffer is retrieved, but before the first attempt is made to access the database, the service must specify the consistency level of the transaction. Refer to "Writing Global Transactions" on page 9-1 for more details on transaction consistency levels.

Example: Checking the Buffer Type

The code example in this section shows how a service can access the data buffer defined in the `TPSVCINFO` structure to determine its type by using the `tptypes()` function. (This process is described in "Checking for Buffer Type" on page 3-14.) The service also checks the maximum size of the buffer to determine whether or not to reallocate space for the buffer.

This example is derived from the `ABAL` service that is part of the banking application provided with the BEA Tuxedo software. It shows how the service is written to accept a request either as an `aud VIEW` or an `FML` buffer. If its attempt to determine the message type fails, the service returns a string with an error message plus an appropriate return code; otherwise it executes the segment of code that is appropriate for the buffer type. For more information on the `tpreturn()` function, refer to "Terminating a Service Routine" on page 5-17.

Listing 5-5 Checking for Buffer Type

```
#define TMTYPERR 1 /* return code indicating tptypes failed */
#define INVALMTY 2 /* return code indicating invalid message type */

void
ABAL(transb)

TPSVCINFO *transb;

{
   struct aud *transv; /* view message */
   FBFR *transf;  /* fielded buffer message */
   int repc;  /* tpgetrply return code */
   char typ[TMTYPELEN+1], subtyp[TMSTYPELEN+1]; /* type, subtype of message */
   char *retstr;  /* return string if tptypes fails */

/* find out what type of buffer sent */
   if (tptypes((char *)transb->data, typ, subtyp) == -1) {
      retstr=tpalloc("STRING", NULL, 100);
      (void)sprintf(retstr,
      "Message garbled; tptypes cannot tell what type message\n");
      tpreturn(TPFAIL, TMTYPERR, retstr, 100, 0);
   }
/* Determine method of processing service request based on type */
   if (strcmp(typ, "FML") == 0) {
      transf = (FBFR *)transb->data;
... code to do abal service for fielded buffer ...
 tpreturn succeeds and sends FML buffer in reply
   }
   else if (strcmp(typ, "VIEW") == 0 && strcmp(subtyp, "aud") == 0) {
      transv = (struct aud *)transb->data;
... code to do abal service for aud struct ...
 tpreturn succeeds and sends aud view buffer in reply
   }
   else {
      retstr=tpalloc("STRING", NULL, 100);
     (void)sprintf(retstr,
     "Message garbled; is neither FML buffer nor aud view\n");
     tpreturn(TPFAIL, INVALMTY, retstr, 100, 0);
   }
}
```

Example: Checking the Priority of the Service Request

Note: The `tpgprio()` and `tpsprio()` functions, 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()` function. Then, based on the priority level, the application routes the print job to the appropriate destination printer and pipes the contents of `pbuf->data` to that printer.

The application queries `pbuf->flags` 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()` function, refer to "Terminating a Service Routine" on page 5-17.

Listing 5-6 Checking the Priority of a Received Request

```
#include <stdio.h>
#include "atmi.h"

char *roundrobin();

PRINTER(pbuf)

TPSVCINFO *pbuf;     /* print buffer */

{
char prname[20], ocmd[30];     /* printer name, output command */
long rlen;                     /* return buffer length */
int prio;                      /* priority of request */
FILE *lp_pipe;                 /* pipe file pointer */

prio=tpgprio();
if (prio <= 20)
  (void)strcpy(prname,"bigjobs"); /* send low priority (verbose)
                                     jobs to big comp. center
                                     laser printer where operator
                                     sorts output and puts it
                                     in a bin */
else if (prio <= 60)
  (void)strcpy(prname,roundrobin()); /* assign printer on a
                                        rotating basis to one of
                                        many local small laser printers
```

```
                                        where output can be picked
                                        up immediately; roundrobin() cycles
                                        through list of printers */
else
   (void)strcpy(prname,"hispeed");
                                    /* assign job to high-speed laser
                                       printer; reserved for those who
                                       need verbose output on a daily,
                                       frequent basis */

(void)sprintf(ocmd, "lp -d%s", prname);   /* output lp(1) command */
lp_pipe = popen(ocmd, "w");               /* create pipe to command */
(void)fprintf(lp_pipe, "%s", pbuf->data); /* print output there */
(void)pclose(lp_pipe);                    /* close pipe */

if ((pbuf->flags & TPNOREPLY))
   tpreturn(TPSUCCESS, 0, NULL, 0, 0);
rlen = strlen(prname) + 1;
pbuf->data = tprealloc(pbuf->data, rlen); /* ensure enough space for name */
(void)strcpy(pbuf->data, prname);
tpreturn(TPSUCCESS, 0, pbuf->data, rlen, 0);

char *
roundrobin()

{
static char *printers[] = {"printer1", "printer2", "printer3", "printer4"};
static int p = 0;

if (p > 3)
    p=0;
return(printers[p++]);
}
```

Terminating a Service Routine

The tpreturn(3c), tpcancel(3c), and tpforward(3c) functions 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.
- tpforward() forwards a request to another service for further processing.

Sending Replies

The tpreturn(3c) function marks the end of the service routine and sends a message to the requester. Use the following signature to call the tpreturn() function:

```
void
tpreturn(int rval, int rcode, char *data, long len, long flags)
```

The following table describes the arguments to the tpreturn() function.

Table 5-2 tpreturn( ) Function Arguments

Argument	Description
rval	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 function succeeded. The function stores the reply message in the caller's buffer. If there is a reply message, it is in the caller's buffer. ■ TPFAIL (default)—the service terminated unsuccessfully. The function reports an error message to the client process waiting for the reply. In this case, the client's tpcall( ) or tpgetrply( ) function call fails and the system sets the tperrno(5) variable to TPESVCFAIL to indicate an application-defined failure. If a reply message was expected, it is available in the caller's buffer. ■ TPEXIT—the service terminated unsuccessfully. The function 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.
rcode	Returns an application-defined return code to the caller. The client can access the value returned in *rcode* by querying the tpurcode(5) global variable. The function returns this code regardless of success or failure.

Argument	Description
data	Pointer to the reply message that is returned to the client process. The message buffer must have been allocated previously by `tpalloc()`. If you use the same buffer that was passed to the service in the `SVCINFO` structure, you need not be concerned with buffer allocation or disposition because both are handled by the system-supplied `main()`. You cannot free this buffer using the `tpfree()` command; any attempt to do so quietly fails. You can resize the buffer using the `tprealloc()` function. If you use another buffer (that is, a buffer other than the one passed to the service routine) to return the message, it is your responsibility to allocate it. The system frees the buffer automatically when the application calls the `tpreturn()` function. If no reply message needs to be returned, set this argument to the NULL pointer. **Note:** If no reply is expected by the client (that is, if `TPNOREPLY` was set), the `tpreturn()` function ignores the *`data`* and *`len`* arguments and returns control to `main()`.
len	Length of the reply buffer. The application accesses the value of this argument through the `olen` parameter of the `tpcall()` function or the `len` parameter of the `tpgetrply()` function. Acting as the client, the process can use this returned value to determine whether the reply buffer has grown. If a reply is expected by the client and there is no data in the reply buffer (that is, if the *`data`* argument is set to the NULL pointer), the function sends a reply with zero length, without modifying the client's buffer. The system ignores the value of this argument if the *`data`* argument is not specified.
flag	Currently not used.

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()` functions are described in detail in "Writing Request/Response Clients and Servers" on page 6-1.

When `tpreturn()` is called, control always returns to `main()`. If a service has sent requests with asynchronous replies, it must receive all expected replies or invalidate them with `tpcancel()` before returning control to `main()`. Otherwise, the outstanding replies are automatically dropped when they are received by the BEA Tuxedo system `main()`, 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 buffer referenced by `*odata`. If `tpacall()` is used to send the request, and `tpreturn()` returns successfully, the reply message is available in the `tpgetrply()` buffer that is referenced by `*data`.

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 `tperrno`. In the case of failed messages, the system sets `tperrno` to `TPESVCERR`. This situation takes precedence over the value of the `tpurcode` global variable. If this type of error occurs, no reply data is returned, and both the contents and length of the caller's output buffer remain unchanged.

If `tpreturn()` returns a message in a buffer of an unknown type or a buffer that is not allowed by the caller (that is, if the call is made with *`flags`* set to `TPNOCHANGE`), the system returns `TPEOTYPE` in `tperrno(5)`. In this case, application success or failure cannot be determined, and the contents and length of the output buffer remain unchanged.

The value returned in the `tpurcode(5)` global variable is not relevant if the `tpreturn()` function 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 `tperrno(5)`. In this case, `tperrno(5)` is set to `TPETIME` and the reply data is not sent, leaving the contents and length of the caller's reply buffer 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 buffer for the reply message since it must use the request buffer 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, frees the reply buffer, and sets the *rval* argument of the tpreturn( ) function to TPFAIL. If the call succeeds, the debit balance is retrieved from the reply buffer.

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 buffer. This move is necessary because this occurrence of the field in an FML buffer 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 frees the reply buffer that was allocated in the service routine and sets the *rval* argument to TPSUCCESS, returning the pertinent account information to the status line.

Listing 5-7 tpreturn( ) Function

```
#include <stdio.h>      /* UNIX */
#include <string.h>     /* UNIX */
#include "fml.h"        /* BEA Tuxedo System */
#include "atmi.h"       /* BEA Tuxedo System */
#include "Usysflds.h"   /* BEA Tuxedo System */
#include "userlog.h"    /* BEA Tuxedo System */
#include "bank.h"       /* BANKING #defines */
#include "bank.flds.h"  /* bankdb fields */


/*
 * Service to transfer an amount from a debit account to a credit
 * account
 */

void
#ifdef __STDC__
TRANSFER(TPSVCINFO *transb)

#else
```

```
TRANSFER(transb)
TPSVCINFO *transb;
#endif

{
   FBFR *transf;          /* fielded buffer of decoded message */
   long db_id, cr_id;     /* from/to account id's              */
   float db_bal, cr_bal;  /* from/to account balances          */
   float tamt;            /* amount of the transfer            */
   FBFR *reqfb;           /* fielded buffer for request message*/
   int reqlen;            /* length of fielded buffer          */
   char t_amts[BALSTR];   /* string for transfer amount        */
   char db_amts[BALSTR];  /* string for debit account balance  */
   char cr_amts[BALSTR];  /* string for credit account balance */

/* Set pointr to TPSVCINFO data buffer */
transf = (FBFR *)transb->data;

/* Get debit (db_id) and credit (cr_id) account IDs */

/* must have valid debit account number */
if (((db_id = Fvall(transf, ACCOUNT_ID, 0)) < MINACCT) || (db_id > MAXACCT)) {
   (void)Fchg(transf, STATLIN, 0,"Invalid debit account number",(FLDLEN)0);
   tpreturn(TPFAIL, 0, transb->data, 0L, 0);
}
/* must have valid credit account number */
if ((cr_id = Fvall(transf, ACCOUNT_ID, 1)) < MINACCT || cr_id > MAXACCT) {
   (void)Fchg(transf,STATLIN, 0,"Invalid credit account number",(FLDLEN)0);
   tpreturn(TPFAIL, 0, transb->data, 0L, 0);
}

/* get amount to be withdrawn */
if (Fget(transf, SAMOUNT, 0, t_amts, <  0) 0 || strcmp(t_amts,"") == 0) {
   (void)Fchg(transf, STATLIN, 0, "Invalid amount",(FLDLEN)0);
   tpreturn(TPFAIL, 0, transb->data, 0L, 0);
}
(void)sscanf(t_amts,"%f",tamt);

/* must have valid amount to transfer */
if (tamt = 0.0) {
   (void)Fchg(transf, STATLIN, 0,
      "Transfer amount must be greater than $0.00",(FLDLEN)0);
    tpreturn(TPFAIL, 0, transb->data, 0L, 0);
}

/* make withdraw request buffer */
if ((reqfb = (FBFR *)tpalloc("FML",NULL,transb->len)) == (FBFR *)NULL) {
   (void)userlog("tpalloc failed in transfer\n");
   (void)Fchg(transf, STATLIN, 0,
```

```
      "unable to allocate request buffer", (FLDLEN)0);
   tpreturn(TPFAIL, 0, transb->data, 0L, 0);
}
reqlen = Fsizeof(reqfb);

/* put ID in request buffer */
(void)Fchg(reqfb,ACCOUNT_ID,0,(char *)&db_id, (FLDLEN)0);

/* put amount in request buffer */
(void)Fchg(reqfb,SAMOUNT,0,t_amts, (FLDLEN)0);

/* increase the priority of withdraw call */
if (tpsprio(PRIORITY, 0L) == -1)
   (void)userlog("Unable to increase priority of withdraw\n");

if (tpcall("WITHDRAWAL", (char *)reqfb,0, (char **)&reqfb,
      (long *)&reqlen,TPSIGRSTRT) == -1) {
   (void)Fchg(transf, STATLIN, 0,
         "Cannot withdraw from debit account", (FLDLEN)0);
   tpfree((char *)reqfb);
   tpreturn(TPFAIL, 0,transb->data, 0L, 0);
}

/* get "debit" balance from return buffer */

(void)strcpy(db_amts, Fvals((FBFR *)reqfb,SBALANCE,0));
void)sscanf(db_amts,"%f",db_bal);
if ((db_amts == NULL) || (db_bal < 0.0)) {
   (void)Fchg(transf, STATLIN, 0,
         "illegal debit account balance", (FLDLEN)0);
   tpfree((char *)reqfb);
   tpreturn(TPFAIL, 0, transb->data, 0L, 0);
}

/* put deposit account ID in request buffer */
(void)Fchg(reqfb,ACCOUNT_ID,0,(char *)&cr_id, (FLDLEN)0);

/* put transfer amount in request buffer */
(void)Fchg(reqfb,SAMOUNT,0,t_amts, (FLDLEN)0);

/* Up the priority of deposit call */
if (tpsprio(PRIORITY, 0L) == -1)
     (void)userlog("Unable to increase priority of deposit\n");

/* Do a tpcall to deposit to second account */
if (tpcall("DEPOSIT", (char *)reqfb, 0, (char **)&reqfb,
        (long *)&reqlen, TPSIGRSTRT) == -1) {
  (void)Fchg(transf, STATLIN, 0,
           "Cannot deposit into credit account", (FLDLEN)0);
```

```
  tpfree((char *)reqfb);
  tpreturn(TPFAIL, 0,transb->data, 0L, 0);
}

/* get "credit" balance from return buffer */

(void)strcpy(cr_amts, Fvals((FBFR *)reqfb,SBALANCE,0));
(void)sscanf(cr_amts,"%f",&cr_bal);
if ((cr_amts == NULL) || (cr_bal 0.0)) {
   (void)Fchg(transf, STATLIN, 0,
         "Illegal credit account balance", (FLDLEN)0);
    tpreturn(TPFAIL, 0, transb->data, 0L, 0);
}

/* set buffer for successful return */
(void)Fchg(transf, FORMNAM, 0, "CTRANSFER", (FLDLEN)0);
(void)Fchg(transf, SAMOUNT, 0, Fvals(reqfb,SAMOUNT,0), (FLDLEN)0);
(void)Fchg(transf, STATLIN, 0, "", (FLDLEN)0);
(void)Fchg(transf, SBALANCE, 0, db_amts, (FLDLEN)0);
(void)Fchg(transf, SBALANCE, 1, cr_amts, (FLDLEN)0);
tpfree((char *)reqfb);
tpreturn(TPSUCCESS, 0,transb->data, 0L, 0);
}
```

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(3c)`. If a reply subsequently arrives, it is silently discarded.

Use the following signature to call the `tpcancel()` function:

```
void
tpcancel(int cd)
```

The `cd` (call descriptor) argument identifies the process you want to cancel.

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

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

Listing 5-8 Invalidating a Reply After Timing Out

```
int cd1;
 .
 .
 .
      if ((cd1=tpacall(sname, (char *)audv, sizeof(struct aud),
         TPNOTRAN)) == -1) {
      .
      .
      .
      }
      if (tpgetrply(cd1, (char **)&audv,&audrl, 0) == -1) {
        if (tperrno == TPETIME) {
          tpcancel(cd1);
          .
          .
          .
        }
      }
      tpreturn(TPSUCCESS, 0,NULL, 0L, 0);
```

Forwarding Requests

The `tpforward(3c)` function allows a service to forward a request to another service for further processing.

Use the following signature to call the `tpforward()` function:

```
void
tpforward(char *svc, char *data, long len, long flags)
```

The following table describes the arguments to the `tpreturn()` function.

Table 5-3 tpreturn() Function Arguments

Argument	Description
svc	Character pointer to the name of the service to which the request is to be forwarded.

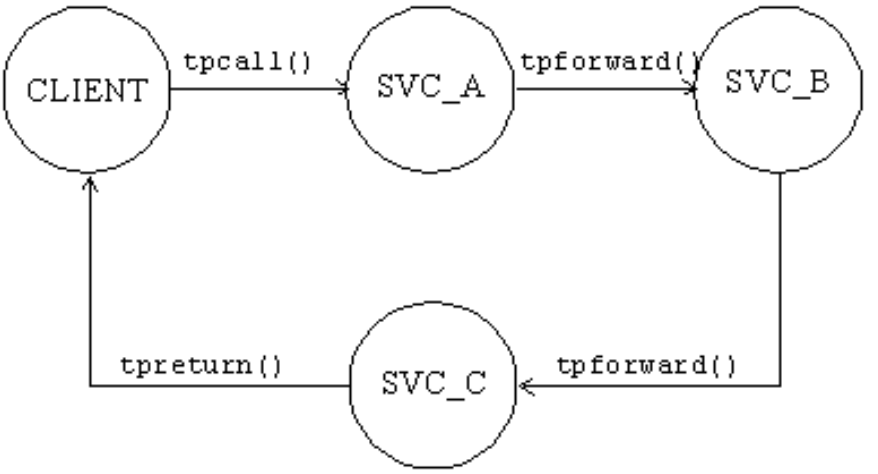
Argument	Description
data	Pointer to the reply message that is returned to the client process. The message buffer must have been allocated previously by `tpalloc()`. If you use the same buffer that was passed to the service in the `SVCINFO` structure, you need not be concerned with buffer allocation or disposition because both are handled by the system-supplied `main()`. You cannot free this buffer using the `tpfree()` command; any attempt to do so quietly fails. You can resize the buffer using the `tprealloc()` function. If you use another buffer (that is, a buffer other than the one that is passed to the service routine) to return the message, it is your responsibility to allocate it. The system frees the buffer automatically when the application calls the `tpreturn()` function. If no reply message needs to be returned, set this argument to the NULL pointer. **Note:** If no reply is expected by the client (that is, if `TPNOREPLY` was set), the `tpreturn()` function ignores the *`data`* and *`len`* arguments and returns control to `main()`.
len	Length of the reply buffer. The application accesses the value of this argument through the `olen` parameter of the `tpcall()` function or the `len` parameter of the `tpgetrply()` function. Acting as the client, the process can use this returned value to determine whether the reply buffer has grown. If a reply is expected by the client and there is no data in the reply buffer (that is, if the *`data`* argument is set to the NULL pointer), the function sends a reply with zero length, without modifying the client's buffer. The system ignores the value of this argument if the *`data`* argument is not specified.
flag	Currently not used.

The functionality of `tpforward()` 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()` function and the last service in the chain (`SVC_C`) provides a reply using the `tpreturn()` function.

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()` function.

When a process calls `tpforward()`, the system-supplied `main()` 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 tpforward() can be used to indicate success up to that point in processing the request. If no application errors have been detected, you can invoke tpforward(), otherwise, you can call tpreturn() with *rval* set to TPFAIL.

The following example is borrowed from the OPEN_ACCT service routine which is part of the ACCT server. This example illustrates how the service sends its data buffer to the DEPOSIT service by calling tpforward(). The code shows how to test the SQLCODE to determine whether the account insertion is successful. If the new account is added successfully, the branch record is updated to reflect the new account, and the data buffer is forwarded to the DEPOSIT service. On failure, tpreturn() is called with *rval* set to TPFAIL and the failure is reported on the status line of the form.

Listing 5-9 tpforward( ) Function

```
 ...
/* set pointer to TPSVCINFO data buffer */
transf = (FBFR *)transb->data;
...
/* Insert new account record into ACCOUNT*/
account_id = ++last_acct;    /* get new account number */
tlr_bal = 0.0;               /* temporary balance of 0 */
EXEC SQL insert into ACCOUNT (ACCOUNT_ID, BRANCH_ID, BALANCE,
ACCT_TYPE, LAST_NAME, FIRST_NAME, MID_INIT, ADDRESS, PHONE) values
(:account_id, :branch_id, :tlr_bal, :acct_type, :last_name,
      :first_name, :mid_init, :address, :phone);
if (SQLCODE != SQL_OK) {    /* Failure to insert */
      (void)Fchg(transf, STATLIN, 0,
          "Cannot update ACCOUNT", (FLDLEN)0);
       tpreturn(TPFAIL, 0, transb->data, 0L, 0);
}

/* Update branch record with new LAST_ACCT */

EXEC SQL update BRANCH set LAST_ACCT = :last_acct where BRANCH_ID = :branch_id;
if (SQLCODE != SQL_OK) {      /* Failure to update */
       (void)Fchg(transf, STATLIN, 0,
           "Cannot update BRANCH", (FLDLEN)0);
        tpreturn(TPFAIL, 0, transb->data, 0L, 0);
}
/* up the priority of the deposit call */
if (tpsprio(PRIORITY, 0L) == -1)
     (void)userlog("Unable to increase priority of deposit\n");

/* tpforward same buffer to deposit service to add initial balance */
tpforward("DEPOSIT", transb->data, 0L, 0);
```

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 function 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 function 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() functions 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(3c) function:

```
int
tpadvertise(char *svcname, void *func)
```

The following table describes the arguments to the tpadvertise() function.

Table 5-4 tpadvertise( ) Function Arguments

Argument	Description
svcname	Pointer to the 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 NULL string is not a valid value. If it is specified, an error (TPEINVAL) results.
func	Pointer to the address of a BEA Tuxedo system function that is called to perform a service. Frequently, this name is the same as the name of the service. The NULL string is not a valid value. If it is specified, an error results.

Unadvertising Services

The tpunadvertise(3c) function 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() function:

```
tpunadvertise(char *svcname)
char *svcname;
```

The tpunadvertise() function contains one argument, which is described in the following table.

Table 5-5 tpunadvertise( ) Function Arguments

Argument	Description
svcname	Pointer to the 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 NULL 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() function. In this example, a server called TLR is programmed to offer only the service called TLR_INIT when booted. After some initialization, TLR_INIT advertises two services called DEPOSIT and WITHDRAW. Both are performed by the tlr_funcs function, and both are built into the TLR server.

After advertising DEPOSIT and WITHDRAW, TLR_INIT unadvertises itself.

Listing 5-10 Dynamic Advertising and Unadvertising

```
extern void tlr_funcs()
 .
 .
 .
 if (tpadvertise("DEPOSIT", (tlr_funcs)(TPSVCINFO *)) == -1)
       check for errors;
 if (tpadvertise("WITHDRAW", (tlr_funcs)(TPSVCINFO *)) == -1)
       check for errors;
 if (tpunadvertise("TLR_INIT") == -1)
       check for errors;
 tpreturn(TPSUCCESS, 0, transb->data,0L, 0);
```

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.

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 -o` *`filename`* `-f` *`filenames`* `-l` *`filenames`* `-s -v`

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

Table 5-6 buildserver Command-line Options

This Option . . .	Allows You to Specify the . . .
`-o` *`filename`*	Name of the executable output file. The default is `a.out`.
`-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 C program file (*`file.`*`c`), 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 C program file (*`file.`*`c`), 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.

Table 5-6 buildserver Command-line Options

This Option . . .	Allows You to Specify the . . .
-s [*service*:]*function*	Name of service or services offered by the server and the name of the function 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 function that performs that service. Alternatively, you can specify any names. To assign names, use the following syntax: *service*:*function*
`-t`	Specifies that the server is coded in a thread-safe manner and may be booted as multithreaded if specified as such in the configuration file.

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 functions are called and which libraries contain references to those functions.

By default, the `buildserver` command invokes the UNIX `cc` command. You can specify an alternative compile command and set your own flags for the compile and link-edit phases, however, by setting the `CC` and `CFLAGS` environment variables, respectively. For more information, refer to “Setting Environment Variables” on page 2-5.

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` function, and `CLOSE_ACCT`, which calls a function of the same name.

```
buildserver –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*

Using a C++ Compiler

There are basically two differences between using a C++ compiler and a C compiler to develop application ATMI servers:

- Different declarations of the service function
- Different use of constructors and destructors

Declaring Service Functions

When declaring a service function for a C++ compiler, you must declare it to have "C" linkage using `extern "C"`. Specify the function prototype as follows:

```
#ifdef __cplusplus
extern "C"
#endif
MYSERVICE(TPSVCINFO *tpsvcinfo)
```

By declaring the name of your service with "C" linkage, you ensure that the C++ compiler will not *modify* the name. Many C++ compilers change the function name to include type information for the parameters and function return.

This declaration also allows you to:

- Link both C and C++ service routines into a single server without indicating the type of each routine.
- Use dynamic service advertisement, which requires accessing the symbol table of the executable to find the function name.

Using Constructors and Destructors

C++ constructors are called to initialize class objects when those objects are created, and destructors are invoked when class objects are destroyed. For automatic (that is, local, non-static) variables that contain constructors and destructors, the constructor is called when the variable comes into scope and the destructor is called when the variable goes out of scope. However, when you call the `tpreturn()` or `tpforward()` function, the compiler performs a non-local goto (using `longjmp`(3)) such that destructors for automatic variables are not called. To avoid this problem, write the application so that you call `tpreturn()` or `tpforward()` from the service routine directly (instead of from any functions that are called from the service routine). In addition, one of the following should be true:

- The service routine should not have any automatic variables with destructors (they should be declared and used in a function called by the service routine).
- Automatic variables should be declared and used in a nested scope (contained within curly brackets {}) in such a way that the scope ends before calling the `tpreturn()` or `tpforward()` function.

In other words, you should define the application so that there are no automatic variables with destructors in scope in the current function or on the stack when the `tpreturn()` or `tpforward()` function is called.

For proper handling of global and static variables that contain constructors and destructors, many C++ compilers require that you compile `main()` using the C++ compiler.

Note: Special processing is included in the `main()` routine to ensure that any constructors are executed when the program starts and any destructors are executed when the program exits.

Because `main()` is provided by the BEA Tuxedo system, you do not compile it directly. To ensure that the file is compiled using C++, you must use the C++ compiler with the `buildserver` command. By default, the `buildserver` command invokes the UNIX `cc` command. You can specify that the `buildserver` command invoke the C++ compiler, instead, by setting the `CC` environment variable to the full path name for the C++ compiler. Also, you can set flags for any options that you want to include on the C++ command line by setting the `CFLAGS` environment variable. For more

information, refer to “Setting Environment Variables” on page 2-5.

CHAPTER

6 Writing Request/Response Clients and Servers

This topic includes the following sections:

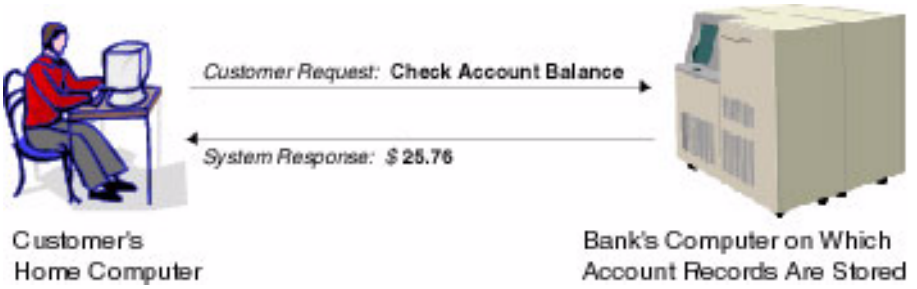
- Overview of Request/Response Communication
- Sending Synchronous Messages
- Sending Asynchronous Messages
- Setting and Getting Message Priorities

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, allocated a buffer, and placed a request for input into that buffer, it can then send the request message to a service subroutine for processing and receive a reply message.

Sending Synchronous Messages

The `tpcall(3c)` function sends a request to a service subroutine and synchronously waits for a reply. Use the following signature to call the `tpcall()` function:

```
int
tpcall(char *svc, char *idata, long ilen, char **odata, long *olen,
long flags)
```

The following table describes the arguments to the `tpcall()` function.

Table 6-1 tpcall( ) Function Arguments

Argument	Description
svc	Pointer to the name of the service offered by your application.

Argument	Description
idata	Pointer that contains the address of the data portion of the request. The pointer must reference a typed buffer that was allocated by a prior call to `tpalloc()`. Note that the `type` (and optionally the `subtype`) of *idata* must match the `type` (and optionally the `subtype`) expected by the service routine. If the types do not match, the system sets `tperrno` to `TPEITYPE` and the function call fails. If the request requires no data, set *idata* to the NULL pointer. This setting means that the parameter can be ignored. If no data is being sent with the request, you do not need to allocate a buffer for *idata*.
ilen	Length of the request data in the buffer referenced by *idata*. If the buffer is a self-defining type, that is, an `FML`, `FML32`, `VIEW`, `VIEW32`, `X_COMMON`, `X_C_TYPE`, or `STRING` buffer, you can set this argument to zero to indicate that the argument should be ignored.
**odata*	Address of a pointer to the output buffer that receives the reply. You must allocate the output buffer using the `tpalloc()` function. If the reply message contains no data, upon successful return from `tpcall()`, the system sets **olen* to zero, and the pointer and the contents of the output buffer remain unchanged. You can use the same buffer for both the request and reply messages. If you do, you must set **odata* to the address of the pointer returned when you allocate the input buffer. It is an error for this parameter to point to NULL.
`olen`	Pointer to the length of the reply data. It is an error for this parameter to point to NULL.
`flags`	Flag options. You can `OR` a series of flags together. If you set this value to zero, the communication is conducted in the default manner. For a list of valid flags and the defaults, refer to `tpcall(3c)` in the *BEA Tuxedo ATMI C Function Reference*.

`tpcall()` waits for the expected reply.

Note: Calling the `tpcall()` function is logically the same as calling the `tpacall()` function immediately followed by `tpgetrply()`, as described in "Sending Asynchronous Messages" on page 6-11.

The request carries the priority set by the system for the specified service (*svc*) unless a different priority has been explicitly set by a call to the `tpsprio()` function (described in "Setting and Getting Message Priorities" on page 6-16).

`tpcall()` returns an integer. On failure, the value of this integer is -1 and the value of `tperrno(5)` is set to a value that reflects the type of error that occurred. For information on valid error codes, refer to `tpcall(3c)` in the *BEA Tuxedo ATMI C 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 buffer 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(3c)` in the *BEA Tuxedo ATMI C Function Reference*.

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

To access the new size of the buffer, use the address returned in the **olen* parameter. To determine whether a reply buffer has changed in size, compare the size of the reply buffer before the call to `tpcall()` with the value of **olen* after its return. If **olen* is larger than the original size, the buffer has grown. If not, the buffer size has not changed.

You should reference the output buffer by the value returned in *odata* after the call because the output buffer may change for reasons other than an increase in buffer size. You do not need to verify the size of request buffers because the request data is not adjusted once it has been allocated.

Note: If you use the same buffer for the request and reply message, and the pointer to the reply buffer has changed because the system adjusted the size of the buffer, then the input buffer pointer no longer references a valid address.

Example: Using the Same Buffer for Request and Reply Messages

The following example shows how the client program, `audit.c`, makes a synchronous call using the same buffer for both the request and reply messages. In this case, using the same buffer is appropriate because the `*audv` message buffer 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 `bal` and `ermsg` fields to zero and the NULL string, respectively, in preparation for the values to be returned by the service.
3. The `svc_name` and `hdr_type` variables represent the service name and the balance type requested, respectively. In this example, these variables represent `account` and `teller`, respectively.

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

```
. . .
/* Create buffer and set data pointer */

audv = (struct aud *)tpalloc("VIEW", "aud", sizeof(struct aud));

       /* Prepare aud structure */

audv->b_id = q_branchid;
audv->balance = 0.0;
(void)strcpy(audv->ermsg, "");

        /* Do tpcall */

if (tpcall(svc_name,(char *)audv,sizeof(struct aud),
        (char **)&audv,(long *)&audrl,0)== -1){
        (void)fprintf (stderr, "%s service failed\n %s: %s\n",
        svc_name, svc_name, audv->ermsg);
        retc = -1;
}
else
       (void)printf ("Branch %ld %s balance is $%.2f\n",
```

```
            audv->b_id, hdr_type, audv->balance);
. . .
```

Example: Testing for Change in Size of Reply Buffer

The following code provides a generic example of how an application test for a change in buffer size after a call to `tpcall()`. In this example, the input and output buffers must remain equal in size.

Listing 6-2 Testing for Change in Size of the Reply Buffer

```
char *svc, *idata, *odata;
long ilen, olen, bef_len, aft_len;
. . .
if (idata = tpalloc("STRING", NULL, 0) == NULL)
   error

if (odata = tpalloc("STRING", NULL, 0) == NULL)
   error

place string value into idata buffer

ilen = olen = strlen(idata)+1;
. . .
bef_len = olen;
if (tpcall(svc, idata, ilen, &odata, &olen, flags) == -1)
   error

aft_len = olen;

if (aft_len > bef_len){  /* message buffer has grown */

   if (idata = tprealloc(idata, olen) == NULL)
      error
}
```

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 TRANSFER service 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(3c) in the *BEA Tuxedo ATMI C Function Reference*.

Listing 6-3 Sending a Synchronous Message with TPSIGRSTRT Set

```
     /* Do a tpcall to withdraw from first account */

if (tpcall("WITHDRAWAL", (char *)reqfb,0, (char **)&reqfb,
     (long *)&reqlen,TPSIGRSTRT) == -1) {
     (void)Fchg(transf, STATLIN, 0,
     "Cannot withdraw from debit account", (FLDLEN)0);
     tpfree((char *)reqfb);
}
...
     /* Do a tpcall to deposit to second account */

if (tpcall("DEPOSIT", (char *)reqfb, 0, (char **)&reqfb,
     (long *)&reqlen, TPSIGRSTRT) == -1) {
     (void)Fchg(transf, STATLIN, 0,
     "Cannot deposit into credit account", (FLDLEN)0);
     tpfree((char *)reqfb);
}
```

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, `accrcv`, 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 TPNOREPLY | TPNOTRAN" on page 6-13 shows a similar example using an asynchronous message call.

Listing 6-4 Sending a Synchronous Message with TPNOTRAN Set

```
#include <stdio.h>
#include "atmi.h"

main()

{
char *rbuf;                /* report buffer */
long r1len, r2len, r3len;  /* buffer lengths of send, 1st reply,
                              and 2nd reply buffers for report */
join application

if (rbuf = tpalloc("STRING", NULL, 0) == NULL)  /* allocate space
for report */
   leave application and exit program
(void)strcpy(rbuf,
   "REPORT=accrcv DBNAME=accounts"); /* send parms of report */
r1len = strlen(rbuf)+1;              /* length of request */


start transaction
```

```
if (tpcall("REPORT", rbuf, r1len, &rbuf,
   &r2len, 0) == -1)                /* get report print stream */
   error routine
if (tpcall("PRINTER", rbuf, r2len, &rbuf,
   &r3len, TPNOTRAN) == -1)         /* send report to printer */
   error routine
(void)printf("Report sent to %s printer\n",
   rbuf);                           /* indicate which printer */

terminate transaction
free buffer
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, allocated buffers are freed, the client leaves the application, and the program is exited.

Example: Sending a Synchronous Message with TPNOCHANGE Set

The following example shows how the `TPNOCHANGE` communication flag is used to enforce strong buffer type checking by indicating that the reply message must be returned in the same type of buffer that was originally allocated. This example refers to a service routine called `REPORT`. (The `REPORT` service is also shown in "Example: Sending a Synchronous Message with TPNOTRAN Set" on page 6-8.)

In this example, the client receives the reply in a `VIEW` typed buffer called `rview1` and prints the elements in `printf()` statements. The strong type check flag, `TPNOCHANGE`, forces the reply to be returned in a buffer of type `VIEW` and of subtype `rview1`.

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 buffer 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 `VIEW` format without modifying the client process to reflect the change.

Listing 6-5 Sending a Synchronous Message with TPNOCHANGE Set

```
#include <stdio.h>
#include "atmi.h"
#include "rview1.h"

main(argc, argv)
int argc;
char * argv[];

{
char *rbuf;            /* report buffer */
struct rview1 *rrbuf;  /* report reply buffer */
long rlen, rrlen;      /* buffer lengths of send and reply
                          buffers for report */
if (tpinit((TPINIT *) tpinfo) == -1)
   fprintf(stderr, "%s: failed to join application\n", argv[0]);

if (rbuf = tpalloc("STRING", NULL, 0) == NULL) { /* allocate space
for report */
   tpterm();
   exit(1);
}
                       /* allocate space for return buffer */
if (rrbuf = (struct rview1 *)tpalloc("VIEW", "rview1",
sizeof(struct rview1)) \ == NULL{
   tpfree(rbuf);
   tpterm();
   exit(1);
}
(void)strcpy(rbuf, "REPORT=accrcv DBNAME=accounts FORMAT=rview1");
rlen = strlen(rbuf)+1;     /* length of request */
                           /* get report in rview1 struct */
if (tpcall("REPORT", rbuf, rlen, (char **)&rrbuf, &rrlen,
TPNOCHANGE) == -1) {
   fprintf(stderr, "accounts receivable report failed in service
call\n");
   if (tperrno == TPEOTYPE)
      fprintf(stderr, "report returned has wrong view type\n");
   tpfree(rbuf);
   tpfree(rrbuf);
   tpterm();
   exit(1);
}
(void)printf("Total accounts receivable %6d\n", rrbuf->total);
(void)printf("Largest three outstanding %-20s %6d\n", rrbuf->name1,
rrbuf->amt1);
(void)printf("%-20s %6d\n", rrbuf->name2, rrbuf->amt2);
```

```
(void)printf("%-20s %6d\n", rrbuf->name3, rrbuf->amt3);
tpfree(rbuf);
tpfree(rrbuf);
tpterm();
}
```

Sending Asynchronous Messages

This section explains how to:

- Send an asynchronous request using the `tpacall()` function
- Get an asynchronous reply using the `tpgetrply()` function

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 `tpforward()` function is used to pass (or forward) a process from one service to another. For a description of the `tpforward()` function, refer to "Writing Servers" on page 5-1.

Sending an Asynchronous Request

The `tpacall(3c)` function sends a request to a service and immediately returns. Use the following signature to call the `tpacall()` function:

```
int
tpacall(char *svc, char *data, long len, long flags)
```

The following table describes the arguments to the `tpacall()` function.

Table 6-2 tpacall( ) Function Arguments

Argument	Description
svc	Pointer to the name of the service offered by your application.
data	Pointer that contains the address of the data portion of the request. The pointer must reference a typed buffer that was allocated by a prior call to `tpalloc()`. Note that the `type` (and optionally the `subtype`) of *idata* must match the `type` (and optionally the `subtype`) expected by the service routine. If the types do not match, the system sets `tperrno` to `TPEITYPE` and the function call fails. If the request requires no data, set *data* to the NULL pointer. This setting means that the parameter can be ignored. If no data is being sent with the request, you do not need to allocate a buffer for *data*.
len	Length of the request data in the buffer referenced by *data*. If the buffer is a self-defining type, that is, an `FML`, `FML32`, `VIEW`, `VIEW32`, `X_COMMON`, `X_C_TYPE`, or `STRING` buffer, you can set this argument to zero, indicating that the argument should be ignored.
flags	Flag options. You can list a group of flags by using the logical `OR` operator. If you set this value to zero, the communication is conducted in the default manner. For a list of valid flags and defaults, refer to `tpacall(3c)` in the *BEA Tuxedo ATMI C Function Reference*.

The `tpacall()` function sends a request message to the service named in the *svc* parameter and immediately returns from the call. Upon successful completion of the call, the `tpacall()` function returns an integer that serves as a descriptor 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 assigned to the *flags* parameter, the parameter signals to `tpacall()` that a reply is not expected. When this flag is set, on success `tpacall()` returns a value of `0` as the reply descriptor. If subsequently passed to the `tpgetrply()` function, this value becomes invalid. Guidelines for using this flag value correctly when a process is in transaction mode are discussed in "Writing Global Transactions" on page 9-1.

On error, tpacall() returns -1 and sets tperrno(5) 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.

Example: Sending an Asynchronous Message with TPNOTRAN | TPNOREPLY

The following example shows how tpacall() uses the TPNOTRAN and TPNOREPLY flags. This code is similar to the code in “Example: Sending a Synchronous Message with TPNOTRAN Set” on page 6-8. In this case, however, a reply is not expected from the PRINTER service. By setting both TPNOTRAN and TPNOREPLY flags, 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 6-6 Sending an Asynchronous Message with TPNOREPLY | TPNOTRAN

```
#include <stdio.h>
#include "atmi.h"

main()

{
char *rbuf;         /* report buffer */
long rlen, rrlen;   /* buffer lengths of send, reply buffers for
report */

join application

if (rbuf = tpalloc("STRING", NULL, 0) == NULL) /* allocate space
for report */
 error
(void)strcpy(rbuf, "REPORT=accrcv DBNAME=accounts");/* send parms
of report */
rlen = strlen(rbuf)+1;  /* length of request */

start transaction

if (tpcall("REPORT", rbuf, rlen, &rbuf, &rrlen, 0)
   == -1) /* get report print stream */
```

```
   error
if (tpacall("PRINTER", rbuf, rrlen, TPNOTRAN|TPNOREPLY)
   == -1) /* send report to printer */
   error

. . .
commit transaction
free buffer
leave application
}
```

Example: Sending Asynchronous Requests

The following example shows a series of asynchronous calls that make up the total bank balance query. Because the banking application data is distributed among several database sites, an SQL query needs to be executed against each one. The application performs these queries by selecting one branch identifier per database site, and calling the ABAL or TBAL service for each site. The branch identifier is not used in the actual SQL query, but it enables the BEA Tuxedo system to route each request to the proper site. In the following code, the `for` loop invokes `tpacall()` once for each site.

Listing 6-7 Sending Asynchronous Requests

```
audv->balance = 0.0;
(void)strcpy(audv->ermsg, "");

for (i=0; i<NSITE; i++) {

  /* Prepare aud structure */

  audv->b_id = sitelist[i];    /* routing done on this field */

  /* Do tpacall */

  if ((cd[i]=tpacall(sname, (char *)audv, sizeof(struct aud), 0))
     == -1) {
     (void)fprintf (stderr,
        "%s: %s service request failed for site rep %ld\n",
        pgmname, sname, sitelist[i]);
        tpfree((char *)audv);
        return(-1);
  }
}
```

Getting an Asynchronous Reply

A reply to a service call can be received asynchronously by calling the `tpgetrply(3c)` function. The `tpgetrply()` function dequeues a reply to a request previously sent by `tpacall()`.

Use the following signature to call the `tpgetrply()` function:

```
int
tpgetrply(int *cd, char **data, long *len, long flags)
```

The following table describes the arguments to the `tpgetrply()` function.

Table 6-3 tpgetrply( ) Function Arguments

Argument	Description
cd	Pointer to the call descriptor returned by the `tpacall()` function.
**data*	Address of a pointer to the output buffer that receives the reply. You must allocate the output buffer using the `tpalloc()` function. If the reply message contains no data, upon successful return from `tpcall()`, the system sets **data* to zero. The pointer and the contents of the output buffer remain unchanged. You can use the same buffer for both the request and reply messages. If you do, then you must set *odata* to the address of the pointer returned when you allocated the input buffer. It is an error for this parameter to point to NULL.
len	Pointer to the length of the reply data. It is an error for this parameter to point to NULL.
flags	Flag options. You can list a group of flags using the logical `OR` operator. If you set this value to zero, the communication is conducted in the default manner. For a list of valid flags and defaults, refer to `tpcall(3c)`) in the *BEA Tuxedo ATMI C Function Reference*.

By default, the function waits for the arrival of the reply that corresponds to the value referenced by the `cd` parameter. During this waiting interval, a blocking timeout may occur. A time-out occurs when `tpgetrply()` fails and `tperrno(5)` is set to `TPETIME` (unless the *`flags`* parameter is set to `TPNOTIME`).

Setting and Getting Message Priorities

Two ATMI functions allow you to determine and set the priority of a message request: `tpsprio(3c)` and `tpgprio(3c)`. 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(3c)` function enables you to set the priority of a message request.

The `tpsprio()` function 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()` function:

```
int
tpsprio(int prio, long flags);
```

The following table describes the arguments to the `tpsprio()` function.

Table 6-4 tpsprio( ) Function Arguments

Argument	Description
prio	Integer indicating a new priority value. The effect of this argument is controlled by the *flags* parameter. If *flags* is set to 0, *prio* 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 *prio* 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.
flags	Flag indicating whether the value of *prio* 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 6-8 Setting the Priority of a Request Message

```
/* increase the priority of withdraw call */
if (tpsprio(PRIORITY, 0L) == -1)
   (void)userlog("Unable to increase priority of withdraw\n");

if (tpcall("WITHDRAWAL", (char *)reqfb,0, (char **)&reqfb, (long *)
\
       &reqlen,TPSIGRSTRT) == -1) {
   (void)Fchg(transf, STATLIN, 0, "Cannot withdraw from debit
account", \
       (FLDLEN)0);
   tpfree((char *)reqfb);
   tpreturn(TPFAIL, 0,transb->data, 0L, 0);
}
```

Getting a Message Priority

The `tpgprio(3c)` function enables you to get the priority of a message request.

Use the following signature to call the `tpgprio()` function:

```
int
tpgprio();
```

A requester can call the `tpgprio()` function after invoking the `tpcall()` or `tpacall()` function to retrieve the priority of the request message. If a requester calls the function but no request is sent, the function fails, returning `-1` and setting `tperrno(5)` to `TPENOENT`. Upon success, `tpgprio()` 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()` function, 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 6-9 Determining the Priority of a Request After It Is Sent

```
#include <stdio.h>
#include "atmi.h"

main ()
{
int cd1, cd2;              /* call descriptors */
int pr1, pr2;              /* priorities to two calls */
char *buf1, *buf2;         /* buffers */
long buf1len, buf2len;     /* buffer lengths */

join application

if (buf1=tpalloc("FML", NULL, 0) == NULL)
   error
if (buf2=tpalloc("FML", NULL, 0) == NULL)
   error

populate FML buffers with send request
```

```
if ((cd1 = tpacall("service1", buf1, 0, 0)) == -1)
   error
if ((pr1 = tpgprio()) == -1)
   error
if ((cd2 = tpacall("service2", buf2, 0, 0)) == -1)
   error

if ((pr2 = tpgprio()) == -1)\
   error

if (pr1 >= pr2) {   /* base the order of tpgetrplys on priority of
calls */
        if (tpgetrply(&cd1, &buf1, &buf1len, 0) == -1)
           error
        if (tpgetrply(&cd2, &buf2, &buf2len, 0) == -1)
           error
}
else {
        if (tpgetrply(&cd2, &buf2, &buf2len, 0) == -1)
           error
        if (tpgetrply(&cd1, &buf1, &buf1len, 0) == -1)
           error
}
. . .
}
```

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

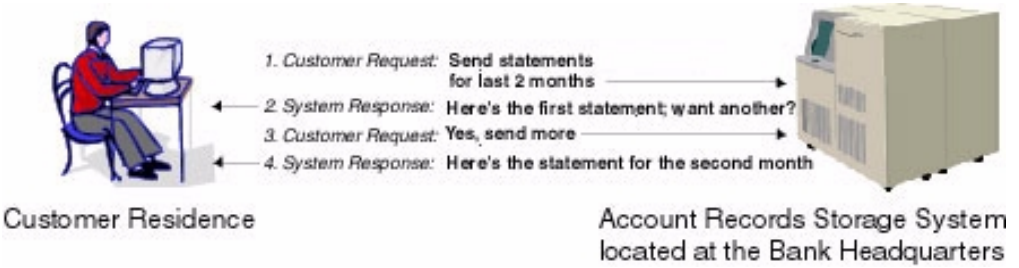
Overview of Conversational Communication

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 buffers. The buffer types must be recognized by the application. For more information on buffer types, refer to "Overview of Typed Buffers" on page 3-2.

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 `tpforward()`.

Joining an Application

A conversational client must join an application via a call to `tpinit()` 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(3c)` function sets up a conversation:

Use the following signature to call the `tpconnect()` function.

```
int
tpconnect(char *name, char *data, long len, long flags)
```

The following table describes the arguments to the tpconnect() function.

Table 7-1 tpconnect( ) Function Arguments

Argument	Description
name	Character pointer to a conversational service name. If you do not specify *name* as a pointer to a conversational service, the call fails with a value of -1 and tperrno is set to the error code TPENOENT.
data	Pointer to a data buffer. When establishing the connection, you can send data simultaneously by setting the *data* argument to point to a buffer previously allocated by tpalloc(). The type and subtype of the buffer must be types recognized by the service being called. You can set the value of *data* to NULL to specify that no data is to be sent. The conversational service being called receives the *data* and *len* pointers via the TPSVCINFO data structure passed to it by main() when the service is invoked. (A request/response service receives the *data* and *len* pointers in the same way.) For more information on the TPSVCINFO data structure, refer to "Defining a Service" on page 5-10.
len	Length of the data buffer. If the buffer is self-defining (for example, an FML buffer), you can set *len* to 0.
flag	Specifies the flag settings. For a complete list of valid flag settings, refer to tpconnect(3c) in the *BEA Tuxedo ATMI C Function Reference*. The system notifies the called service through the flag members of the TPSVCINFO structure.

The BEA Tuxedo system returns a connection descriptor (*cd*) when a connection is established with tpconnect(). The *cd* 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, the tpconnect() function returns a value of -1 and sets tperrno to the appropriate error condition. For a list of possible error codes, refer to tpconnect(3c) in the *BEA Tuxedo ATMI C Function Reference*.

The following example shows how to use the tpconnect() function.

Listing 7-1 Establishing a Conversational Connection

```
#include atmi.h
#define    FAIL    -1
int cd1;        /* Connection Descriptor */
main()
{
 if ((cd = tpconnect("AUDITC",NULL,0,TPSENDONLY)) == -1) {
    error routine
 }
}
```

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(3c) function; the process without control can receive messages using the tprecv(3c) function.

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(3c) function with the following signature:

```
int
tpsend(int cd, char *data, long len, long flags, long *revent)
```

The following table describes the arguments to the tpsend() function.

Table 7-2 tpsend( ) Function Arguments

Argument	Description
cd	Specifies the connection descriptor returned by the `tpconnect()` function identifying the connection over which the data is sent.
data	Pointer to a data buffer. When establishing the connection, you can send data simultaneously by setting the *data* argument to point to a buffer previously allocated by `tpalloc()`. The `type` and `subtype` of the buffer must be types recognized by the service being called. You can set the value of *data* to NULL to specify that no data is to be sent. The conversational service being called receives the *data* and *len* pointers via the `TPSVCINFO` data structure passed to it by `main()` when the service is invoked. (A request/response server receives the *data* and *len* pointers in the same way.) For more information on the `TPSVCINFO` data structure, refer to "Defining a Service" on page 5-10.
len	Length of the data buffer. If the buffer is self-defining (for example, an `FML` buffer), you can set *len* to 0. If you do not specify a value for *data*, this argument is ignored.
revent	Pointer to event value set when an error is encountered (that is, when `tperrno(5)` is set to `TPEEVENT`). For a list of valid event values, refer to `tpsend(3c)` in the *BEA Tuxedo ATMI C Function Reference*.
flag	Specifies the flag settings. For a list of valid flag settings, refer to `tpsend(3c)` in the *BEA Tuxedo ATMI C Function Reference*.

In the event of a failure, the `tpsend()` function returns a value of `-1` and sets `tperrno(5)` to the appropriate error condition. For a list of possible error codes, refer to `tpsend(3c)` in the *BEA Tuxedo ATMI C Function Reference*.

You are not required to pass control each time you issue the `tpsend()` function. 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()` function.

Listing 7-2 Sending Data in Conversational Mode

```
if (tpsend(cd,line,0,TPRECVONLY,revent) == -1) {
              (void)userlog("%s: tpsend failed tperrno %d",
                     argv[0],tperrno);
              (void)tpabort(0);
              (void)tpterm();
              exit(1);
       }
```

Receiving Messages

To receive data sent over an open connection, use the `tprecv(3c)` function with the following signature:

```
int
tprecv(int cd, char **data, long *len, long flags, long *revent)
```

The following table describes the arguments to the `tprecv()` function.

Argument	Description
cd	Specifies the connection descriptor. If a subordinate program issues the call, the *cd* argument should be set to the value specified in the `TPSVCINFO` structure for the program. If the originator program issues the call, the *cd* argument should be set to the value returned by the `tpconnect()` function.

Argument	Description
data	Pointer to a data buffer. The *data* argument must point to a buffer previously allocated by tpalloc(). The *type* and *subtype* of the buffer must be types recognized by the service being called. This value cannot be NULL; if it is, the call fails and tperrno(5) is set to TPEINVAL. The conversational service being called receives the *data* and *len* pointers via the TPSVCINFO data structure passed to it by main() when the service is invoked. (A request/response service receives the *data* and *len* pointers in the same way.) For more information on the TPSVCINFO data structure, refer to "Defining a Service" on page 5-10.
len	Length of the data buffer. If the buffer is self-defining (for example, an FML buffer), you can set *len* to 0. This value cannot be NULL; if it is, the call fails and tperrno(5) is set to TPEINVAL.
revent	Pointer to event value set when an error is encountered (that is, when tperrno is set to TPEEVENT). Refer to tprecv(3c) in the *BEA Tuxedo ATMI C Function Reference* for a list of valid event values.
flag	Specifies the flag settings. Refer to tprecv(3c) in the *BEA Tuxedo ATMI C Function Reference* for a list of valid flags.

Upon success, the **data* argument points to the data received and *len* contains the size of the buffer. If *len* is greater than the total size of the buffer before the call to tprecv(), the buffer size has changed and *len* indicates the new size. A value of 0 for the *len* argument indicates that no data was received.

The following example shows how to use the tprecv() function.

Listing 7-3 Receiving Data in Conversation

```
if (tprecv(cd,line,len,TPNOCHANGE,revent) != -1) {
          (void)userlog("%s: tprecv failed tperrno %d revent %ld",
               argv[0],tperrno,revent);
            (void)tpabort(0);
            (void)tpterm();
```

```
            exit(1);
}
```

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()` function 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.

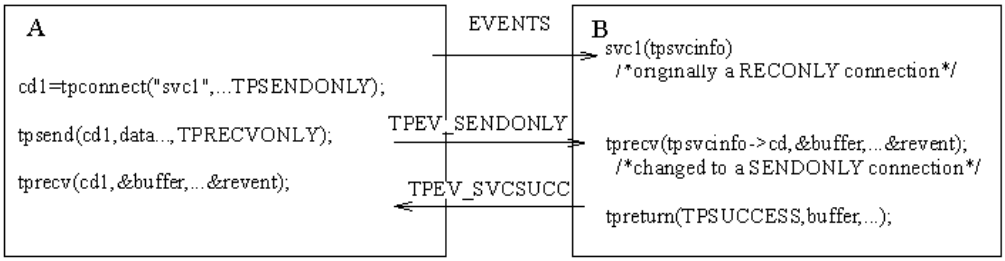
The first example shows how to terminate a simple conversation between two components. The second example illustrates a more complex scenario, with a hierarchical set of conversations.

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 Terminated Gracefully



The program flow is as follows:

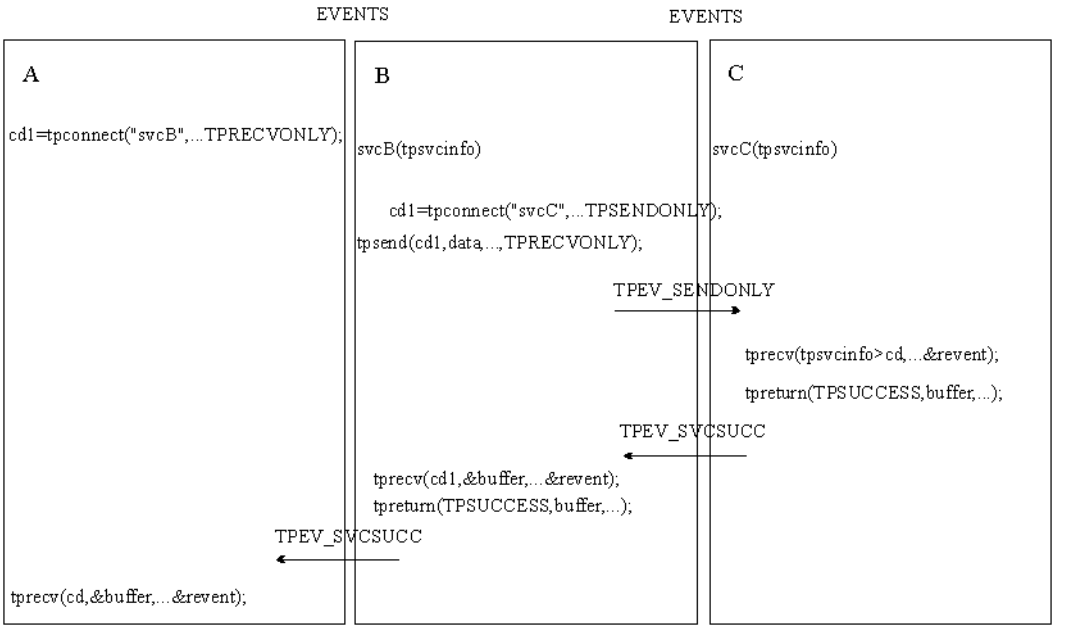
1. A sets up the connection by calling `tpconnect()` with the `TPSENDONLY` flag set, indicating that process B is on the receiving end of the conversation.
2. A turns control of the connection over to B by calling `tpsend()` with the `TPRECVONLY` flag set, resulting in the generation of a `TPEV_SENDONLY` event.
3. The next call by B to `tprecv()` returns a value of -1, sets `tperrno(5)` to `TPEEVENT`, and returns `TPEV_SENDONLY` in the *`revent`* argument, indicating that control has passed to B.
4. B calls `tpreturn()` with *`rval`* 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 *`rval`* 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 `tpforward()`.

Executing a Disorderly Disconnect

The only way in which a disorderly disconnect can be executed is through a call to the `tpdiscon(3c)` function (which is equivalent to "pulling the plug" on a connection). This function 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()` function.

Use the following signature to call the `tpdiscon()` function:

```
int
tpdiscon(int cd)
```

The *cd* argument specifies the connection descriptor returned by the `tpconnect()` function when the connection is established.

The `tpdiscon()` function generates a `TPEV_DISCONIMM` event for the service at the other end of the connection, rendering the *cd* 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 *cd*, the function fails with an error code of `TPEBADDESC`.

For a list and descriptions of all event and error codes, refer to `tpdiscon(3c)` in the *BEA Tuxedo ATMI C Function Reference*.

Building Conversational Clients and Servers

Use the following commands to build conversational clients and servers:

- `buildclient()` as described in “Building Clients” on page 4-10
- `buildserver()` as described in “Building Servers” on page 5-32

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 functions for which they are returned, and a detailed description of each.

Table 7-3 Conversational Communication Events

Event	Received By	Description
`TPEV_SENDONLY`	`tprecv()`	Control of the connection has been passed; this process can now call `tpsend()`.
`TPEV_DISCONIMM`	`tpsend()`, `tprecv()`, `tpreturn()`	The connection has been torn down and no further communication is possible. The `tpdiscon()` function 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.

Table 7-3 Conversational Communication Events

Event	Received By	Description
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 buffer, 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.

CHAPTER

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
- Subscribing to Events
- Unsubscribing from Events
- Posting Events
- Example of Event Subscription

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 buffer 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()` function.
Service call	Servers may receive event notification messages as input to service routines, just as if they were sent by the `tpacall()` function.
Reliable queue	Event notification messages may be stored in a BEA Tuxedo system reliable queue, using the `tpenqueue(3c)` function. Event notification buffers are stored until requests for buffer contents are issued. A BEA Tuxedo system client or server process may call the `tpdequeue(3c)` function to retrieve these notification buffers, or alternately `TMQFORWARD(5)` may be configured to automatically dispatch a BEA Tuxedo system service routine that retrieves a notification buffer. 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 buffers, 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 buffer to an application-defined event name.
2. The posted buffer 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 function, use the `tpsetunsol(3c)` function with the following signature:

```
int
tpsetunsol(*myfunc)
```

The following table describes the single argument that can be passed to the `tpsetunsol()` function.

Table 8-2 tpsetunsol( ) Function Argument

Argument	Description
myfunc	Pointer to a function that conforms to the prototype of a call-back function. In order to conform, the function must accept the following three parameters: ■ *data*—points to the typed buffer that contains the unsolicited message ■ *len*—length of the buffer ■ *flags*—currently not used

When a client receives an unsolicited notification, the system dispatches the call-back function with the message. To minimize task disruption, you should code the unsolicited message handler function to perform only minimal processing tasks, so it can return quickly to the waiting process.

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(3c)`, or by an identifier received with a previously processed message, using `tpnotify(3c)`. 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(3c)` function 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 `tpinit()` and have not yet made a call to `tpterm()`.

Use the following signature to call the `tpbroadcast()` function:

```
int
tpbroadcast(char *lmid, char *usrname, char *cltname, char *data, long len, long
flags)
```

The following table describes the arguments to the `tpbroadcast()` function.

Table 8-3 tpbroadcast( ) Function Arguments

Argument	Description
lmid	Pointer to the logical machine identifier for the client. A value of NULL acts as a wildcard, so that a message can be directed to groups of clients.
usrname	Pointer to the username 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.
cltname	Pointer to the 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.
data	Pointer to the content of a message.
len	Size of the message buffer. If *data* points to a self-defining buffer type, for example, FML, then *len* can be set to 0.
flags	Flag options. Refer to tpbroadcast(3c) in the *BEA Tuxedo ATMI C Function Reference* for information on available flags.

The following example illustrates a call to tpbroadcast() for which all clients are targeted. The message to be sent is contained in a STRING buffer.

Listing 8-1 Using tpbroadcast( )

```
char *strbuf;

if ((strbuf = tpalloc("STRING", NULL, 0)) == NULL) {
         error routine
         }

(void) strcpy(strbuf, "hello, world");

if (tpbroadcast(NULL, NULL, NULL, strbuf, 0, TPSIGRSTRT) == -1)
             error routine
```

Broadcasting Messages by Identifier

The tpnotify(3c) function 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() function:

```
int
tpnotify(CLIENTID *clientid, char *data, long len, long flags)
```

The following table describes the arguments to the tpnotify() function.

Table 8-4 tpnotify( ) Function Arguments

Argument	Description
clientid	Pointer to a CLIENTID structure that is saved from the TPSVCINFO structure that accompanied the request to this service.
data	Pointer to the content of the message.
len	Size of the message buffer. If *data* points to a self-defining buffer type, for example, FML, then *len* can be set to 0.
flags	Flag options. Refer to tpnotify(3c) in the *BEA Tuxedo ATMI C Function Reference* for information on available flags.

Checking for Unsolicited Messages

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

```
int
tpchkunsol()
```

The function takes no arguments.

If any messages are pending, the system invokes the unsolicited message handling function that was specified using tpsetunsol(). Upon completion, the function returns either the number of unsolicited messages that were processed or -1 on error.

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

Subscribing to Events

The tpsubscribe(3c) function 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() function:

```
long handle
tpsubscribe (char *eventexpr, char *filter, TPEVCTL *ctl, long flags)
```

The following table describes the arguments to the tpsubscribe() function.

Table 8-5 tpsubscribe( ) Function Arguments

Argument	Description
eventexpr	Pointer to a set of one or more events to which a process can 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 *BEA Tuxedo ATMI C Function Reference*). 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 any uppercase letter between A and Z. ■ ".*(ERR|err).*"—the caller is subscribing to all user events with names that contain either err or ERR, such as the account_error and ERROR_STATE events, respectively.

Argument	Description
filter	Pointer to a string containing a Boolean filter rule that must be evaluated successfully before the EventBroker posts the event. Upon receiving an event to be posted, the EventBroker applies the filter rule, if one exists, to the posted event's data. If the data passes the filter rule, the EventBroker invokes the notification method specified; otherwise, the EventBroker 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 discriminate among the events about which they are notified. For example, a poster can post an event for withdrawals greater than $10,000, but a subscriber may want to specify a higher threshold for being notified, such as $50,000. Or, a subscriber may want to be notified of large withdrawals made by specific customers. Filter rules are specific to the typed buffers to which they are applied. For more information on filter rules, refer to `tpsubscribe(3c)` in the *BEA Tuxedo ATMI C Function Reference*.
ctl	Pointer to a flag for controlling how a subscriber is notified of an event. Valid values include: ■ NULL—sends unsolicited messages. Refer to "Notification via Unsolicited Message" on page 8-11 for more information. ■ Pointer to a valid `TPEVCTL` structure—sends information based on the `TPEVCTL` structure. Refer to "Notification via Service Call or Reliable Queue" on page 8-11 for more information.
flags	Flag options. For more information on available flag options, refer to `tpsubscribe(3c)` in the *BEA Tuxedo ATMI C Function Reference*.

You can subscribe to both system- and application-defined events using the `tpsubscribe()` function.

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

Notification via Unsolicited Message

If a subscriber is a BEA Tuxedo system client process and *ctl* is NULL, when the event to which the client has subscribed is posted, the EventBroker sends an unsolicited message to the subscriber as follows. When an event name is posted that evaluates successfully against *eventexpr*, the EventBroker tests the posted data against the associated filter rule. If the data passes the filter rule (or if there is no filter rule for the event), then the subscriber receives an unsolicited notification along with any data posted with the event.

In order to receive unsolicited notifications, the client must register an unsolicited message handling routine using the `tpsetunsol()` function.

ATMI clients receiving event notification via unsolicited messages should remove their subscriptions from the EventBroker list of active subscriptions before exiting. This is done using the `tpunsubscribe()` function.

Notification via Service Call or Reliable Queue

Event notification via *service call* enables you to program actions that can be taken in response to specific conditions in your application without human intervention. Event notification via *reliable queue* ensures that event data is not lost. It also provides the subscriber the flexibility of retrieving the event data at any time.

If the subscriber (either a client or a server process) wants event notifications sent to service routines or to stable-storage queues, then the *ctl* parameter of `tpsubscribe()` must point to a valid `TPEVCTL` structure.

The `TPEVCTL` structure contains the following elements:

```
long   flags;
char   name1[32];
char   name2[32];
TPQCTL qctl;
```

The following table summarizes the TPEVCTL typed buffer data structure.

Table 8-6 TPEVCTL Typed Buffer Format

Field	Description
flags	Flag options. For more information on flags, refer to tpsubscribe(3c) in the *BEA Tuxedo ATMI C Function Reference*.
name1	Character string of 32 characters or fewer.
name2	Character string of 32 characters or fewer.
qctl	TPQCTL structure. For more information, refer to tpsubscribe(3c) in the *BEA Tuxedo ATMI C Function Reference*.

Unsubscribing from Events

The tpunsubscribe(3c) function enables a BEA Tuxedo system ATMI client or server to unsubscribe from an event.

Use the following signature to call the tpunsubscribe() function:

```
int
tpunsubscribe (long subscription, long flags)
```

The following table describes the arguments to the tpunsubscribe() function.

Table 8-7 tpunsubscribe( ) Function Arguments

Argument	Description
subscription	Subscription handle returned by a call to tpsubscribe().
flags	Flag options. For more information on available flag options, refer to tpunsubscribe(3c) in the *BEA Tuxedo ATMI C Function Reference*.

Posting Events

The tppost(3c) function enables a BEA Tuxedo ATMI client or server to post an event.

Use the following signature to call the tppost() function:

```
tppost(char *eventname, char *data, long len, long flags)
```

The following table describes the arguments to the tppost() function.

Table 8-8 tppost( ) Function Arguments

Argument	Description
eventname	Pointer to an event name containing up to 31 characters plus NULL. The first character cannot be a dot (".") because the dot is reserved as the first character in names of BEA Tuxedo system-defined events. When defining event names, keep in mind that subscribers can use wildcard capabilities to subscribe to multiple events with a single function call. Using the same prefix for a category of related event names can be helpful.
data	Pointer to a buffer previously allocated using the tpalloc() function.
len	Size of data buffer that should be posted with the event. If *data* points to a buffer of a type that does not require a length to be specified (for example, an FML fielded buffer) or if you set it to NULL, the *len* argument is ignored and the event is posted with no data.
flags	Flag options. For more information on available flag options, refer to tppost(3c) in the *BEA Tuxedo ATMI C Function Reference*.

The following example illustrates an event posting taken from the BEA Tuxedo system sample application bankapp. This example is part of the WITHDRAWAL service. One of the functions of the WITHDRAWAL service is checking for withdrawals greater than $10,000 and posting an event called BANK_TLR_WITHDRAWAL.

Listing 8-2 Posting an Event with tppost( )

```
.
.
.
/* Event logic related */
static float evt_thresh = 10000.00 ; /* default for event threshold */
static char  emsg[200] ;  /* used by event posting logic */
.
.
.
/* Post a BANK_TLR_WITHDRAWAL event ? */
if (amt < evt_thresh) {
     /* no event to post */
     tpreturn(TPSUCCESS, 0,transb->data , 0L, 0);
}
/* prepare to post the event */
if ((Fchg (transf, EVENT_NAME, 0, "BANK_TLR_WITHDRAWAL", (FLDLEN)0) == -1) ||
(Fchg (transf, EVENT_TIME, 0, gettime(), (FLDLEN)0) == -1) ||
(Fchg (transf, AMOUNT, 0, (char *)&amt, (FLDLEN)0) == -1)) {
     (void)sprintf (emsg, "Fchg failed for event fields: %s",
     Fstrerror(Ferror)) ;
}
/* post the event */
else if (tppost ("BANK_TLR_WITHDRAWAL", /* event name */
(char *)transf, /* data */
0L,   /* len */
TPNOTRAN | TPSIGRSTRT) == -1) {
/* If event broker is not reachable, ignore the error */
     if (tperrno != TPENOENT)
     (void)sprintf (emsg, "tppost failed: %s", tpstrerror (tperrno));
}
```

This example simply posts the event to the EventBroker to indicate a noteworthy occurrence in the application. Subscription to the event by interested clients, who can then take action as required, is done independently.

Example of Event Subscription

The following example illustrates a portion of a bankapp application server that subscribes to BANK_TLR_.* events, which includes the BANK_TLR_WITHDRAWAL event shown in the previous example, as well as any other event names beginning with BANK_TLR_. When a matching event is posted, the application notifies the subscriber via a call to a service named WATCHDOG.

Listing 8-3 Subscribing to an Event with tpsubscribe( )

```
.
.
.
/* Event Subscription handles */
static long sub_ev_largeamt = 0L ;
.
.
.
/* Preset default for option 'w' - watchdog threshold */
(void)strcpy (amt_expr, "AMOUNT > 10000.00") ;
.
.
.
/*
 * Subscribe to the events generated
 * when a "large" amount is transacted.
 */
evctl.flags = TPEVSERVICE ;
(void)strcpy (evctl.name1, "WATCHDOG") ;
/* Subscribe */
sub_ev_largeamt = tpsubscribe ("BANK_TLR_.*",amt_expr,&evctl,TPSIGRSTRT) ;
if (sub_ev_largeamt == -1L) {
     (void)userlog ("ERROR: tpsubscribe for event BANK_TLR_.* failed: %s",
     tpstrerror(tperrno)) ;
     return -1 ;
}
.
.
.
{
/* Unsubscribe to the subscribed events */
if (tpunsubscribe (sub_ev_largeamt, TPSIGRSTRT) == -1)
```

```
    (void)userlog ("ERROR: tpunsubscribe to event BANK_TLR_.* failed: %s",
    tpstrerror(tperrno)) ;
    return ;
}
/*
* Service called when a BANK_TLR_.* event is posted.
*/
void
#if defined(__STDC__) || defined(__cplusplus)
WATCHDOG(TPSVCINFO *transb)
#else
WATCHDOG(transb)
TPSVCINFO *transb;
#endif
{
FBFR *transf; /* fielded buffer of decoded message  */
/* Set pointr to TPSVCINFO data buffer  */
transf = (FBFR *)transb->data;
/* Print the log entry  to stdout */
(void)fprintf (stdout, "%20s|%28s|%8ld|%10.2f\n",
Fvals (transf, EVENT_NAME, 0),
Fvals (transf, EVENT_TIME, 0),
Fvall (transf, ACCOUNT_ID, 0),
*( (float *)CFfind (transf, AMOUNT, 0, NULL, FLD_FLOAT)) );
/* No data should be returned by the event subscriber's svc routine */
tpreturn(TPSUCCESS, 0,NULL, 0L, 0);
}
```

CHAPTER

9 Writing Global Transactions

This topic includes the following sections:

- What Is a Global Transaction?
- Starting the Transaction
- Suspending and Resuming a 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.

Note: If a transaction includes calls to `tpcall()`, `tpacall()`, or `tpconnect()` for which the *`flags`* parameter is explicitly set to `TPNOTRAN`, the operations performed by the called service do not become part of that transaction. In this case, the calling process does not invite the called service to be a participant in the current transaction. As a result, services performed by the called process are not affected by the outcome of the current transaction. If `TPNOTRAN` is set for a call that is directed to a service in an XA-compliant server group, the call may be executed outside of transaction mode or in a separate transaction, depending on how the service is configured and coded. For more information, refer to “Implicitly Defining a Global Transaction” on page 9-17.

Starting the Transaction

To start a global transaction, use the `tpbegin(3c)` function with the following signature:

```
int
tpbegin(unsigned long timeout, long flags)
```

The following table describes the arguments to the `tpbegin()` function

Table 9-1 tpbegin( ) Function Arguments

Field	Description
timeout	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 *timeout* parameter delays system detection and reporting of errors. The system uses the *timeout* 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 *timeout* 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 *timeout* 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 *timeout* parameter to a value that is greater than the scanning unit. If you set the *timeout* 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 *timeout* parameter with your application administrator to make sure the value you assign to the *timeout* parameter is compatible with the values assigned to your system parameters.
flags	Currently undefined; must be set to 0.

Any process may call tpbegin() unless the process is already in transaction mode or is waiting for outstanding replies. If tpbegin() is called in transaction mode, the call fails due to a protocol error and tperrno(5) 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 Defining a Global Transaction - High-level View

```
. . .
if (tpbegin(timeout,flags) == -1)
 error routine
program statements
. . .
if (tpcommit(flags) == -1)
 error routine
```

The following example provides a more detailed view of how to define a transaction. This example is excerpted from audit.c, a client program included in bankapp, the sample banking application delivered with the BEA Tuxedo system.

Listing 9-2 Defining a Global Transaction - Detailed View

```
#include <stdio.h>          /* UNIX */
#include <string.h>         /* UNIX */
#include <atmi.h>           /* BEA Tuxedo System */
#include <Uunix.h>          /* BEA Tuxedo System */
#include <userlog.h>        /* BEA Tuxedo System */
#include "bank.h"           /* BANKING #defines */
#include "aud.h"            /* BANKING view defines */

#define INVI 0              /* account inquiry */
#define ACCT 1              /* account inquiry */
#define TELL 2              /* teller inquiry */

static int sum_bal _((char *, char *));
static long sitelist[NSITE] = SITEREP;    /* list of machines to audit */
static char pgmname[STATLEN];        /* program name = argv[0] */
static char result_str[STATLEN]; /* string to hold results of query */
```

```
main(argc, argv)
int argc;
char *argv[];
{
        int aud_type=INVI;          /* audit type -- invalid unless specified */
        int clarg;                  /* command line arg index from optind */
        int c;                      /* Option character */
        int cflgs=0;                /* Commit flags, currently unused */
        int aflgs=0;                /* Abort flags, currently unused */
        int nbl=0;                  /* count of branch list entries */
        char svc_name[NAMELEN];     /* service name */
        char hdr_type[NAMELEN];     /* heading to appear on output */
        int retc;                   /* return value of sum_bal() */
        struct aud *audv;           /* pointer to audit buf struct */
        int audrl=0;                /* audit return length */
        long q_branchid;            /* branch_id to query */

. . .        /* Get Command Line Options and Set Variables */

/* Join application */

if (tpinit((TPINIT *) NULL) == -1) {
        (void)userlog("%s: failed to join application\n", pgmname);
        exit(1);
}

/* Start global transaction */

if (tpbegin(30, 0) == -1) {
        (void)userlog("%s: failed to begin transaction\n", pgmname);
        (void)tpterm();
        exit(1);
}

if (nbl == 0) {    /* no branch id specified so do a global sum */
  retc = sum_bal(svc_name, hdr_type);    /* sum_bal routine not shown */

} else {

        /* Create buffer and set data pointer */

        if ((audv = (struct aud *)tpalloc("VIEW", "aud", sizeof(struct aud)))
               == (struct aud *)NULL) {
               (void)userlog("audit: unable to allocate space for VIEW\n");
                exit(1);
        }

        /* Prepare aud structure */
```

```
        audv->b_id = q_branchid;
        audv->balance = 0.0;
        audv->ermsg[0] = '\0';

        /* Do tpcall */

        if (tpcall(svc_name,(char *)audv,sizeof(struct aud),
           (char **)audv,(long *)audrl,0) == -1){
                    (void)fprintf (stderr,"%s service failed\n%s:  %s\n",
                    svc_name, svc_name, audv->ermsg);
                    retc = -1;

        }else {

                    (void)sprintf(result_str,"Branch %ld %s balance is $%.2f\n",
                    audv->b_id, hdr_type, audv->balance);
        }
                    tpfree((char *)audv);
}

/* Commit global transaction */

if (retc < 0)        /* sum_bal failed so abort */
       (void) tpabort(aflgs);
else {
         if (tpcommit(cflgs) == -1) {
                 (void)userlog("%s: failed to commit transaction\n", pgmname);
                 (void)tpterm();
                 exit(1);
       }
        /*print out results only when transaction has committed successfully*/
        (void)printf("%s",result_str);
}

/* Leave application */

if (tpterm() == -1) {
        (void)userlog("%s: failed to leave application\n", pgmname);
        exit(1);
}
```

If a transaction times out, a call to `tpcommit()` causes the transaction to be aborted. As a result, `tpcommit()` fails and sets `tperrno(5)` to `TPEABORT`.

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

Listing 9-3 Testing for Transaction Timeout

```
if (tpbegin(30, 0) == -1) {
  (void)userlog("%s: failed to begin transaction\n", argv[0]);
  tpterm();
  exit(1);
}
. . .
communication calls
. . .
if (tperrno == TPETIME){
    if (tpabort(0) == -1) {
      check for errors;
}
else if (tpcommit(0) == -1){
      check for errors;
}
. . .
```

Note: When a process is in transaction mode and makes a communication call with *flags* set to 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.

Suspending and Resuming a Transaction

At times, it may be desirable to temporarily remove a process from an incomplete transaction and allow it to initiate a different transaction by calling tpbegin() or tpresume(). For example, suppose a server wants to log a request to the database central event log, but does not want the logging activity to be rolled back if the transaction aborts.

The BEA Tuxedo system provides two functions that allow a client or server to suspend and resume a transaction in such situations: tpsuspend(3c) and tpresume(3c). Using these functions, a process can:

1. Temporarily suspend the current transaction by calling `tpsuspend()`.
2. Start a separate transaction. (In the preceding example, the server writes an entry to the event log.)
3. Commit the transaction started in step 2.
4. Resume the original transaction by calling `tpresume()`.

Suspending a Transaction

Use the `tpsuspend(3c)` function to suspend the current transaction. Use the following signature to call the `tpsuspend()` function:

```
int
tpsuspend(TPTRANID *t_id,long flags)
```

The following table describes the arguments to the `tpsuspend()` function.

Table 9-2 tpsuspend( ) Function Arguments

Field	Description
`*t_id`	Pointer to the transaction identifier.
`flags`	Currently not used. Reserved for future use.

You cannot suspend a transaction with outstanding asynchronous events. When a transaction is suspended, all modifications previously performed are preserved in a pending state until the transaction is committed, aborted, or timed out.

Resuming a Transaction

To resume the current transaction, use the `tpresume(3c)` function with the following signature.

```
int
tpresume(TPTRANID *t_id,long flags)
```

The following table describes the arguments to the `tpresume()` function:

Table 9-3 tpresume( ) Function Arguments

Field	Description
`*t_id`	Pointer to the transaction identifier.
`flags`	Currently not used. Reserved for future use.

It is possible to resume a transaction from a process other than the one that suspended it, subject to certain restrictions. For a list of these restrictions, refer to `tpsuspend(3c)` and `tpresume(3c)` in the *BEA Tuxedo ATMI C Function Reference*.

Example: Suspending and Resuming a Transaction

The following example shows how to suspend one transaction, start and commit a second transaction, and resume the initial transaction. For the sake of simplicity, error checking code has been omitted.

Listing 9-4 Suspending and Resuming a Transaction

```
DEBIT(SVCINFO *s)
{
       TPTRANID t;
       tpsuspend(&t,TPNOFLAGS);  /* suspend invoking transaction*/

       tpbegin(30,TPNOFLAGS); /* begin separate transaction */
       Perform work in the separate transaction.
       tpcommit(TPNOFLAGS);   /* commit separate transaction */

       tpresume(&t,TPNOFLAGS);   /* resume invoking transaction*/

       .
       .
       .
       tpreturn(. . . );
}
```

Terminating the Transaction

To end a global transaction, call `tpcommit(3c)` to commit the current transaction, or `tpabort(3c)` to abort the transaction and roll back all operations.

Note: If `tpcall()`, `tpacall()`, or `tpconnect()` is called by a process that has explicitly set the *`flags`* argument to `TPNOTRAN`, the operations performed by the called service do not become part of the current transaction. In other words, when you call the `tpabort()` function, the operations performed by these services are not rolled back.

Committing the Current Transaction

The `tpcommit(3c)` function 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()` function:

```
int
tpcommit(long flags)
```

Although the `flags` argument is not used currently, you must set it to zero to ensure compatibility with future releases.

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 `tperrno(5)` is set to `TPEPROTO`, indicating a protocol error. If the second or third condition is false, the call fails and `tperrno()` 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 function cancels the transaction, returns -1, and sets `tperrno(5)` to `TPEABORT`. The results are the same if `tpcommit()` is called for a transaction that has already timed out: `tpcommit()` returns -1 and sets `tperrno()` to `TPEABORT`. Refer to "Managing Errors" on page 11-1 for more information on transaction errors.

Two-phase Commit Protocol

When the `tpcommit()` function 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()` function. 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 `tperrno(5)` 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`.

If you later want to override the setting in the configuration file, you can do so by calling the `tpscmt()` function with its *`flags`* argument set to either `TP_CMT_LOGGED` or `TP_CMT_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 `tpcommit()`, 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(3c)` function 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()` function:

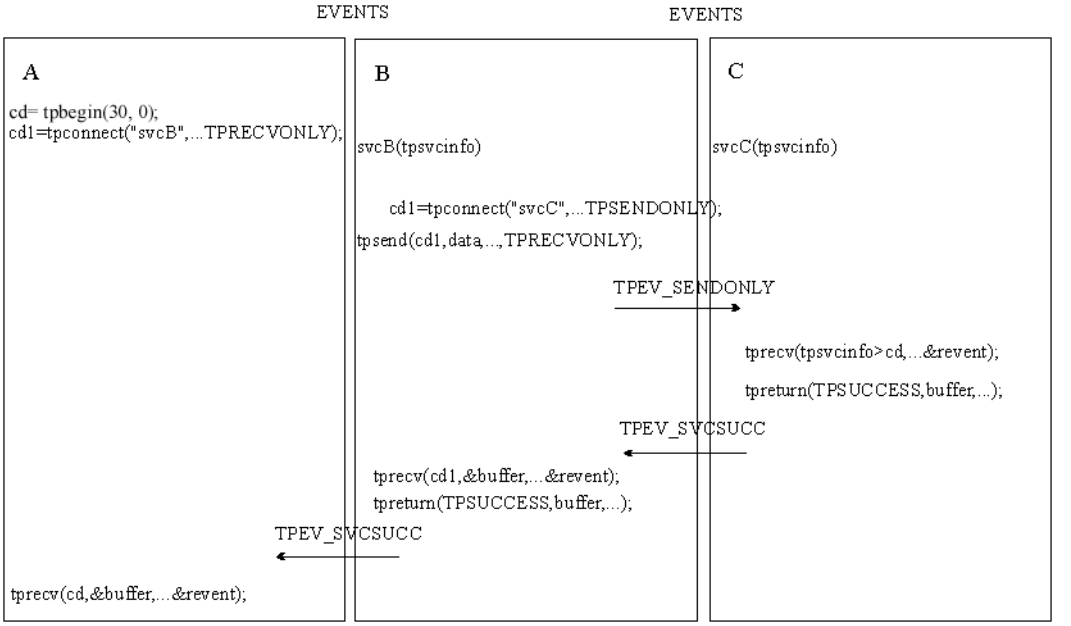
```
int
tpabort(long flags)
```

Although the *`flags`* argument is not used currently, you must set it to zero to ensure compatibility with future releases.

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 18). Then the code checks for participant failures by testing for errors that can be returned on a communication call (lines 19-34).

Listing 9-5 Testing for Participant Success or Failure

```
001    #include <stdio.h>
002    #include "atmi.h"
003
004    main()
005    {
006    char *sbuf, *rbuf;
007    long slen, rlen;
008    if (tpinit((TPINIT *) NULL) == -1)
009        error message, exit program;
010    if (tpbegin(30, 0) == -1)
011        error message, tpterm, exit program;
012    if ((sbuf=tpalloc("STRING", NULL, 100)) == NULL)
013        error message, tpabort, tpterm, exit program;
014    if ((rbuf=tpalloc("STRING", NULL, 2000)) == NULL)
015        error message, tpfree sbuf, tpabort, tpterm, exit program;
016    (void)strcpy(sbuf, "REPORT=accrcv DBNAME=accounts");
017    slen=strlen(sbuf);
018    if (tpcall("REPORT", sbuf, slen, &rbuf, &rlen, 0) == -1) {
019        switch(tperrno) {
020        case TPESVCERR:
021            fprintf(stderr,
022                "REPORT service's tpreturn encountered problems\n");
023            break;
024        case TPESVCFAIL:
025            fprintf(stderr,
026               "REPORT service TPFAILED with return code of %d\n", tpurcode);
027            break;
028        case TPEOTYPE:
029            fprintf(stderr,
030                "REPORT service's reply is not of any known data type\n");
031            break;
032        default:
```

```
033             fprintf(stderr,
034                 "REPORT service failed with error %d\n", tperrno);
035             break;
036         }
037         if (tpabort(0) == -1){
038             check for errors;
039         }
040     }
041     else
042         if (tpcommit(0) == -1)
043             fprintf(stderr, "Transaction failed at commit time\n");
044     tpfree(rbuf);
045     tpfree(sbuf);
046     tpterm();
047     exit(0);
048     }
```

Implicitly Defining a Global Transaction

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

- Explicitly, by calling ATMI functions, as described in “Starting the Transaction” on page 9-3.
- Implicitly, from within a service routine

This section describes the second method.

Implicitly Defining a Transaction in a Service Routine

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 has its *flags* parameter set to TPNOTRAN.

  If the *flags* argument is 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 the flags argument 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 *flags* member of the service information structure returns TPTRAN when queried. For example, if the call is made with the communication *flags* member set to TPNOTRAN | TPNOREPLY, and the service automatically starts a transaction when called, the *flags* member of 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 *flags* set to 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

When a process in transaction mode requests a service from another process, the latter process becomes part of the transaction, unless specifically instructed not to join it.

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 tperrno(5) 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 *flags* field 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(3c) function.

Use the following signature to call the `tpgetlev()` function:

```
int
tpgetlev()  /*  Get current transaction level  */
```

The `tpgetlev()` function requires no arguments. It returns 0 if the caller is not in a transaction, and 1 if it is.

The following code sample is a variation of the `OPEN_ACCT` service that shows how to test for transaction level using the `tpgetlev()` function (line 12). If the process is not already in transaction mode, the application starts a transaction (line 14). If `tpbegin()` fails, a message is returned to the status line (line 16) and the *`rcode`* argument of `tpreturn()` is set to a code that can be retrieved in the global variable `tpurcode(5)` (lines 1 and 17).

Listing 9-6 Testing Transaction Level

```
001 #define BEGFAIL    3    /* tpurcode setting for return if tpbegin fails */

002 void
003 OPEN_ACCT(transb)

004 TPSVCINFO *transb;

005 {
  ... other declarations ...
006 FBFR *transf;    /* fielded buffer of decoded message */
007 int dotran;      /* checks whether service tpbegin/tpcommit/tpaborts */

008 /* set pointer to TPSVCINFO data buffer */

009 transf = (FBFR *)transb->data;

010 /* Test if transaction exists; initiate if no, check if yes */

011 dotran = 0;
012 if (tpgetlev() == 0) {
013     dotran = 1;
014      if (tpbegin(30, 0) == -1) {
015         Fchg(transf, STATLIN, 0,
016             "Attempt to tpbegin within service routine failed\n");
017         tpreturn(TPFAIL, BEGFAIL, transb->data, 0, 0);
018     }
019 }
    . . .
```

If the AUTOTRAN parameter is set to Y, you do not need to call the tpbegin(), and tpcommit() or tpabort() transaction functions 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 (see lines 7 and 10-19). 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 9-17 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 Appendix A, "Platform Data Sheets," in *Installing the BEA Tuxedo System*.

Platform-specific Considerations for Multithreaded/Multicontexted Applications

Many platforms have idiosyncratic requirements for multithreaded and multicontexted applications. Appendix A, "Platform Data Sheets," in *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?

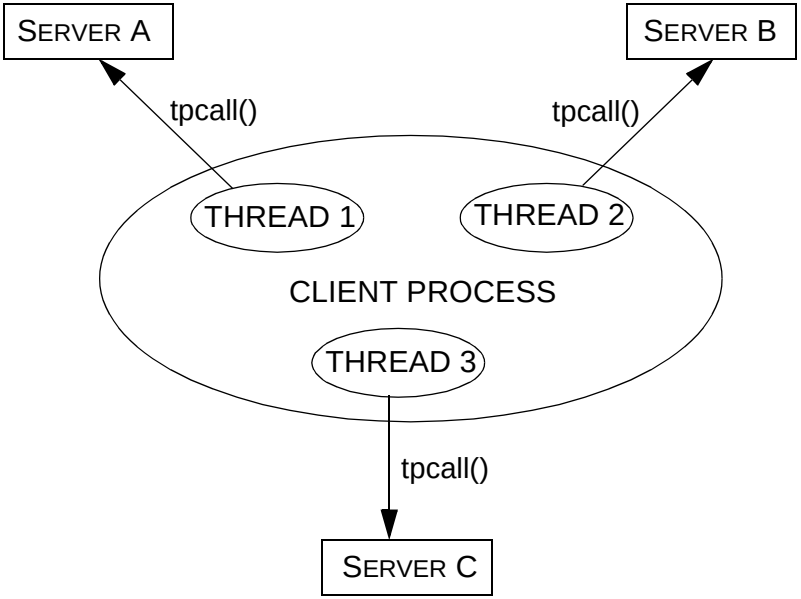
Multithreading is the inclusion of more than one unit of execution in a single process. In a multithreaded application, multiple simultaneous calls can be made from the same process. For example, an individual process is not limited to one outstanding `tpcall()`.

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.

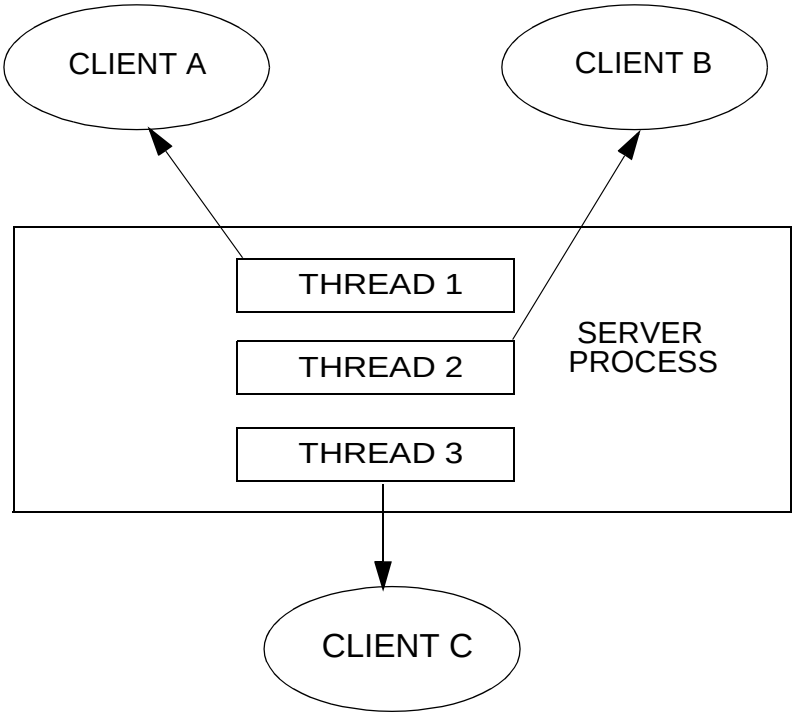
Figure 10-1 Sample Multithreaded Process



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.

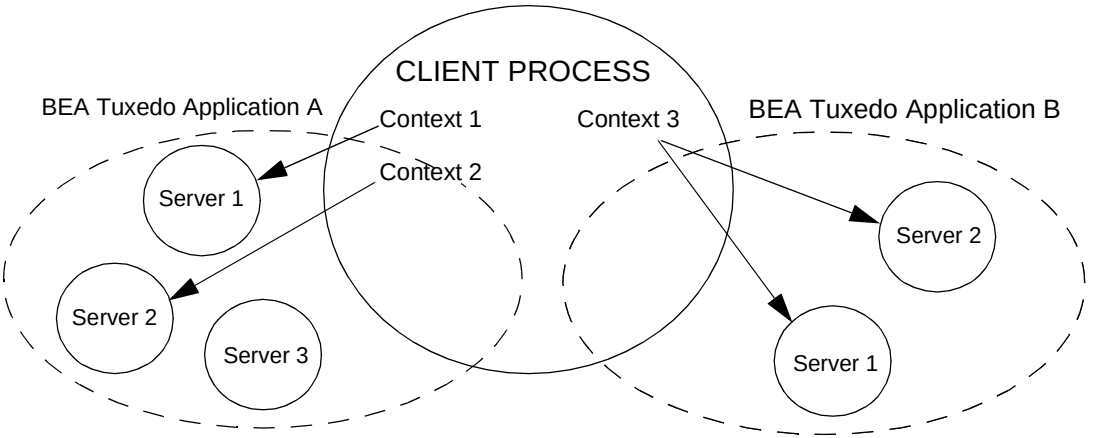
For a more complete list of the characteristics of a context, see “Context Attributes” in one of the following sections:

- “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()`.
- 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()` 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()` for a synchronous request or `tpacall()` 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()`, 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()`.

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()` 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()` 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() 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() is called from a thread that is not associated with a context, a per-process default unsolicited message handler for all new tpinit() 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() 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()`. 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()` to join the application.
> 0	> 1	1. The BEA Tuxedo system creates a thread dispatcher. 2. The dispatcher calls `tpsvrinit()` 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()`.

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()`.
- 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()` or `tpforward()`. 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 Appendix A, "Platform Data Sheets," in *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()` 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()` in one thread and `tpgetrply()` 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()` 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()` or `tpforward()` 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()` 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()` 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()` 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()`, 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()`. 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()` 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()` 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()` 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()`.
12. Switch the client to the `secondapp` context (by calling `tpsetctxt(3c)`) and call `secondapp` services.
13. Terminate the `secondapp` context by calling `tpterm()`.

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()` is called from a thread that is not associated with a context, a per-process default unsolicited message handler for all new `tpinit()` 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()` 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()`.
- You can call `tpcommit()` 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()` or `tpforward()` 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()` 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()` or `tpforward()`.
- A server cannot execute a `tpreturn()` or `tpforward()` 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()` or `tpforward()` 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()` or `tpforward()` 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()`. 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()` or `tpforward()`. 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()` or `tpreturn()` 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()` in the example, substitute two calls to `tpacall()` and two calls to `tpgetrply()` 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()` to receive a response to an earlier call to `tpacall()`, 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()` while RPC calls or conversations are outstanding, `tpcommit()` aborts the transaction, returns `-1`, and sets `tperrno(5)` to `TPEABORT`.
- Functions such as `tpcall()`, `tpacall()`, `tpgetrply()`, `tpconnect()`, `tpsend()`, `tprecv()`, and `tpdiscon()` should not be called in transaction mode unless you are sure that the transaction is not already committing or aborting.
- Two `tpbegin()` calls cannot be made simultaneously for the same context.
- `tpbegin()` 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()`. 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()` 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()` function with the `TPMULTICONTEXTS` flag set in the *`flags`* element of the `TPINIT` data structure.

In any one process, either all calls to `tpinit()` 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()`, 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()` 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()` without the `TPMULTICONTEXTS` flag, all threads in the client are placed in the single-context state (`TPSINGLECONTEXT`).

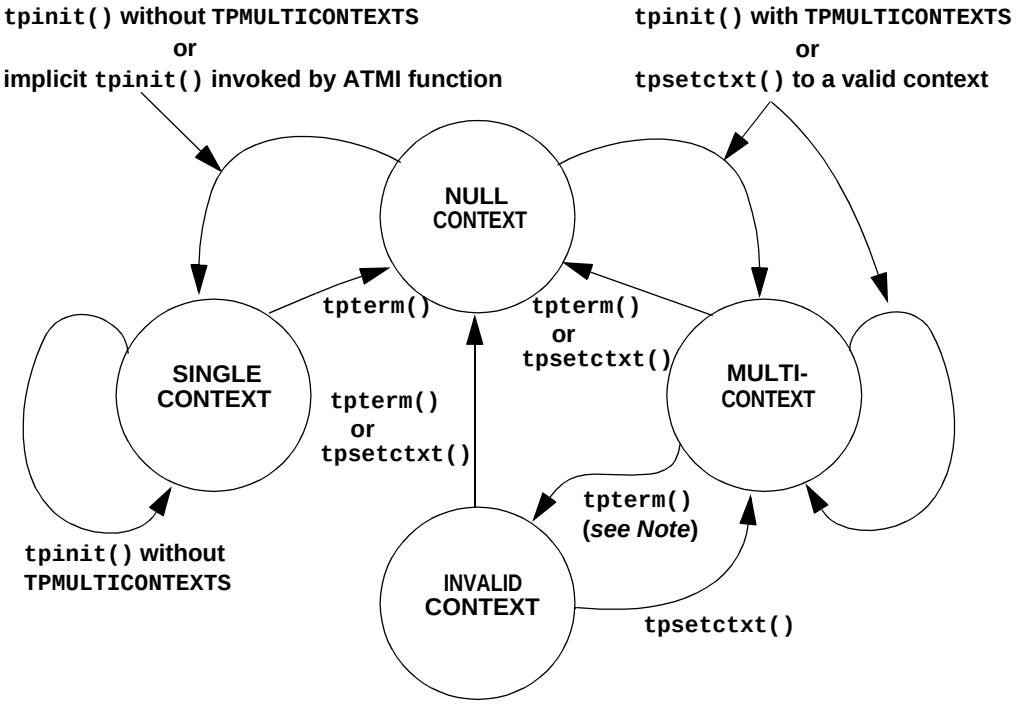
On failure, `tpinit()` leaves the calling thread in its original context (that is, in the context state in which it was operating before the call to `tpinit()`).

Do not call `tpterm()` 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()`, `tpsetctxt(3c)`, and `tpterm()`. (The `tpgetctxt(3c)` function does not produce any context state changes.)

Figure 10-4 Multicontext State Transitions



Note: When `tpterm()` 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()`, `tpsetctxt(3c)`, and `tpterm()`.

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()` without `TPMULTICONTEXTS`	Single context	Single context	Error	Error
`tpinit()` 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()`	Single context	N/A	N/A	Error
`tpterm()` in this thread	Null	Null	Null	Null
`tpterm()` in a different thread of this context	N/A	Null	Invalid	N/A

Getting Replies in a Multithreaded Environment

`tpgetrply()` receives responses only to requests made via `tpacall()`. Requests made with `tpcall()` are separate and cannot be retrieved with `tpgetrply()` regardless of the multithreading or multicontexting level.

tpgetrply() 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() is called in a multithreaded environment, the following restrictions apply:

- If a thread calls tpgetrply() 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() 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() 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() 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()`. 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()` 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()`
- `tpacall()`
- `tpadmcall(3c)`
- `tpbegin()`
- `tpbroadcast()`
- `tpcall()`
- `tpcancel()`
- `tpchkauth()`
- `tpchkunsol()`
- `tpclose(3c)`
- `tpcommit()`
- `tpconnect()`
- `tpdequeue(3c)`
- `tpdiscon()`

- tpenqueue(3c)
- tpforward()
- tpgetlev()
- tpgetrply()
- tpinit()
- tpnotify()
- tpopen(3c)
- tppost()
- tprecv()
- tpresume()
- tpreturn()
- tpscmt(3c)
- tpsend()
- tpservice(3c)
- tpsetunsol()
- tpsubscribe()
- tpsuspend()
- tpterm()
- tpunsubscribe()
- 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()`, 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()`.

If `tpsetunsol()` is called from a thread that is not associated with a context, a per-process default unsolicited message handler for all new `tpinit()` 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()`
- `tpalloc()`
- `tpconvert(3c)`—the requested structure is converted, although it is probably relevant to only a subset of the process.
- `tpfree()`
- `tpinit()`—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()`
- `tpsvrdone()`
- `tpsvrinit()`
- `tptypes()`
- `tpunadvertise()`
- `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()`
- `tpsetctxt(3c)`
- `tpsprio()`
- `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( ) returns -1 and sets tperrno to TPEPROTO.

Calls to tpinit( ) Without TPMULTICONTEXTS

When tpinit( ) 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( ) 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
- Debugging Application Processes
- Comprehensive Example

System Errors

The BEA Tuxedo system uses the `tperrno(5)` variable to supply information to a process when a function fails. All ATMI functions that normally return an integer or pointer return `-1` or `NULL`, respectively, on error and set `tperrno()` to a value that describes the nature of the error. When a function does not return to its caller, as in the case of `tpreturn()` or `tpforward()`, which are used to terminate a service routine, the only way the system can communicate success or failure is through the variable `tperrno()` in the requester.

The tperrordetail(3c) and tpstrerrordetail(3c) functions can be used to obtain additional detail about an error in the most recent BEA Tuxedo system call on the current thread. tperrordetail() returns an integer (with an associated symbolic name) which is then used as an argument to tpstrerrordetail() to retrieve a pointer to a string that contains the error message. The pointer can then be used as an argument to userlog(3c) or fprintf(). For a list of the symbolic names that can be returned, refer to tperrordetail(3c) in the *BEA Tuxedo ATMI C Function Reference*.

tpurcode(5) is used to communicate user-defined conditions only. The system sets the value of tpurcode to the value of the *rcode* argument of tpreturn(). The system sets tpurcode, regardless of the value of the *rval* argument of tpreturn(), unless an error is encountered by tpreturn() or a transaction timeout occurs.

The codes returned in tperrno(5) represent categories of errors, which are listed in the following table.

Table 11-1 tperrno Error Categories

Error Category	tperrno Values
Abort	TPEABORT[2]
BEA Tuxedo system[1]	TPESYSTEM
Call descriptor	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

Error Category	tperrno Values
Permission	TPEPERM
Protocol[1]	TPEPROTO
Queueing	TPEDIAGNOSTIC
Release compatibility	TPERELEASE
Resource manager	TPERMERR
Timeout	TPETIME
Transaction	TPETRAN[2]
Typed buffer mismatch	TPEITYPE and TPEOTYPE

1. Applicable to all ATMI functions for which failure is reported by the value returned in `tperrno(5)`.
2. Refer to "Fatal Transaction Errors" on page 11-22 for more information on this error category.

As footnote 1 shows, four categories of errors are reported by `tperrno(5)` and are applicable to all ATMI functions. The remaining categories are used only for specific ATMI functions.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 11-22.

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 `tperrno(5)`. For more information, refer to the “Central Event Log” on page 11-31. Because these errors occur in the system, rather than in the application, you may need to consult the system administrator to correct them.

Call Descriptor Errors

Call descriptor errors occur as a result of exceeding the maximum limit of call descriptors or referencing an invalid value. Asynchronous and conversational calls return `TPELIMIT` when the maximum number of outstanding call descriptors has been exceeded. `TPEBADDESC` is returned when an invalid call descriptor value is specified for an operation.

Call descriptor errors occur only during asynchronous calls or conversational calls. (Call descriptors are not used for synchronous calls.) Asynchronous calls depend on call descriptors to associate replies with the corresponding requests. Conversational send and receive functions depend on call descriptors to identify the connection; the call that initiates the connection depends on the availability of a call descriptor.

Troubleshooting of call descriptor errors can be done by checking for specific errors at the application level.

Limit Errors

The system allows up to 50 outstanding call descriptors (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 call descriptors 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 call descriptor can become invalid and, if referenced, cause an error to be returned to tperrno(5) in either of two situations:

- A call descriptor is used to retrieve a message, which may be a failed message (TPEBADDESC).
- An attempt is made to reuse a stale call descriptor (TPEBADDESC).

A call descriptor might become stale, for example, in the following circumstances:

- When the application calls tpabort() or tpcommit() and transaction replies (sent without the TPNOTRAN flag) 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 descriptor and the descriptor becomes stale.

Conversational Errors

When an unknown descriptor is specified for conversational services, the tpsend(), tprecv(), and tpdiscon() functions 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 *revent* parameter passed to the function call 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 `tperrno(5)` when an attempt is made to perform an operation that results in a duplicate object. The following table lists the functions that may return the `TPEMATCH` error code and the associated cause

Function	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 functions, refer to the *BEA Tuxedo ATMI C 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 `tperrno(5)`: `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 `tperrno(5)`. The system determines the error by the arguments that are passed to `tpreturn()` and the processing that is performed by this function.

If `tpreturn()` encounters an error in processing or handling arguments, the system returns an error to the original requester and sets `tperrno(5)` to `TPESVCERR`. The receiver determines that an error has occurred by checking the value of `tperrno()`. The system does not send the data from the `tpreturn()` function, and if the failure occurred on `tpgetrply()`, it renders the call descriptor invalid.

If `tpreturn()` does not encounter the `TPESVCERR` error, then the value returned in *`rval`* determines the success or failure of the call. If the application specifies `TPFAIL` in the *`rval`* parameter, the system returns `TPESVCFAIL` in `tperrno(5)` and sends the data message to the caller. If *`rval`* is set to `TPSUCCESS`, the system returns successfully to the caller, `tperrno()` 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 its *`flags`* parameter to `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 with *`flags`* set to `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 the *flags* parameter for the communication functions is set to TPSIGRSTRT, the calls do not fail and the system does not return the TPGOTSIG error code in tperrno(5).

Invalid Argument Errors

Invalid argument errors indicate that an invalid argument was passed to a function. Any ATMI function that takes arguments can fail if you pass it arguments that are invalid. In the case of a function that returns to the caller, the function fails and causes tperrno(5) to be set to TPEINVAL. In the case of tpreturn() or tpforward(), the system sets tperrno() to TPESVCERR for either the tpcall() or tpgetrply() function 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 functions.

MIB Error

The tpadmcall(3c) function returns TPEMIB in tperrno(5) in the event an administrative request fails. *outbuf* is updated and returned to the caller with FML32 fields indicating the cause of the error. For more information on the cause of the error, refer to MIB(5) and TM_MIB(5) in *File Formats, Data Descriptions, MIBs, and System Processes Reference*.

No Entry Errors

No entry errors result from a lack of entries in the system tables or the data structure used to identify buffer types. The meaning of the no entry type error, TPENOENT, depends on the function that is returning it. The following table lists the functions that return this error and describes various causes of error.

Table 11-2 No Entry Errors

Function	Cause
tpalloc()	The system does not know about the type of buffer requested. For a buffer type and/or subtype to be known, there must be an entry for it in a type switch data structure that is defined in the BEA Tuxedo system libraries. Refer to tuxtypes(5) and typesw(5) in the *File Formats, Data Descriptions, MIBs, and System Processes Reference* for more information. On an application level, ensure that you have referenced a known type; otherwise, check with the system administrator.
tpinit()	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 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 the service name because the name is not currently advertised by the calling process.
tpenqueue(3c) tpdequeue(3c)	The system cannot access the queue space because the associated TMQUEUE(5) server is not available. Refer to the *File Formats, Data Descriptions, MIBs, and System Processes Reference* for more information.

Function	Cause

`tppost()`
`tpsubscribe()`
`tpunsubscribe()`
	The system cannot access the BEA Tuxedo system Event Broker. Refer to “Writing Event-based Clients and Servers” on page 8-1 for more information.

Operating System Errors

Operating system errors indicate that an operating system call has failed. The system returns `TPEOS` in `tperrno(5)`. 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 `tpinit()` call fails, returning `TPEPERM` in `tperrno(5)`. 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 function 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 `tperrno(5)`.

Refer to "Introduction to the C Application-Transaction Monitor Interface" in the *BEA Tuxedo ATMI C Function Reference* for more information.

Queuing Error

The `tpenqueue(3c)` or `tpdequeue(3c)` function returns `TPEDIAGNOSTIC` in `tperrno(5)` if the enqueuing or dequeuing on a specified queue fails. The reason for failure can be determined by the diagnostic returned via the *`ctl`* buffer. For a list of valid *`ctl`* flags, refer to `tpenqueue(3c)` or `tpdequeue(3c)` in the *BEA Tuxedo ATMI C Function Reference*

Release Compatibility Error

The BEA Tuxedo system returns `TPERELEASE` in `tperrno(5)` 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(3c)` function (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(3c)` and `tpclose(3c)`, in which case the system returns the value of `TPERMERR` in `tperrno(5)`. 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 *`timeout`* 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. When you set the *`flags`* parameter of a communication call to `TPNOTIME`, the flag setting applies to blocking timeouts only. If a process is in transaction mode, blocking timeouts are not performed and the `TPNOTIME` flag setting is not relevant.

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 call descriptor to an asynchronous transaction reply (specified without the TPNOTRAN flag) 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 the *flags* parameter was not set to TPNOBLOCK.

Note: If you set the TPNOBLOCK flag, 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 11-19.

Transaction Errors

For information on transactions and the non-fatal and fatal errors that can occur, refer to "Transaction Considerations" on page 11-19.

Typed Buffer Errors

Typed buffer errors are returned when requests or replies to processes are sent in buffers of an unknown type. The `tpcall()`, `tpacall()`, and `tpconnect()` functions return `TPEITYPE` when a request data buffer is sent to a service that does not recognize the type of the buffer.

Processes recognize buffer 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 buffers 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 buffer types. An application can set up the buffer type data structure (referred to as a buffer 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()`, `tpgetrply()`, `tpdequeue(3c)`, and `tprecv()` functions return `TPEOTYPE` when a reply message is sent in a buffer that is not recognized or not allowed by the caller. In the latter case, the buffer type is included in the type switch, but the type returned does not match the type that was allocated to receive the reply and a change in buffer type is not allowed by the caller. The caller indicates this preference by setting *`flags`* to `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 buffer 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 buffer 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 `tpurcode` to the value of the *rcode* argument of `tpreturn()`. For more information about `tpreturn(3c)` or `tpurcode(5)`, refer to the *BEA Tuxedo ATMI C Function Reference* and the *File Formats, Data Descriptions, MIBs, and System Processes Reference*, respectively.

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. Specifically, you should:

- Test to determine whether a `-1` or `NULL` value has been returned (depending on the function call).
- Invoke code that contains a switch statement that tests for specific values of `tperrno(5)` and performs the appropriate application logic.

The ATMI supports three functions, `tpstrerrordetail(3c)`, `tpstrerror(3c)`, and `Fstrerror, Fstrerror32(3fml)`, for retrieving the text of an error message from the message catalogs for the BEA Tuxedo system and FML. The functions return pointers to the appropriate error messages. Your program can use a pointer to direct the referenced text to `userlog(3c)` or to another destination. For details, refer to `tpstrerrordetail(3c)` and `tpstrerror(3c)` in the *BEA Tuxedo ATMI C Function Reference*, and `Fstrerror, Fstrerror32(3fml)` in the *BEA Tuxedo ATMI FML Function Reference*.

The following example shows a typical method of handling errors. The `atmicall()` function in this example represents a generic ATMI call. Note the code after the switch statement (line 21): it shows how `tpurcode` can be used to interpret an application-defined return code.

Listing 11-1 Handling Errors

```
001   #include <stdio.h>
002   #include "atmi.h"
003
004   main()
005
006   {
007   int rtnval;
008
009   if (tpinit((TPINIT *) NULL) == -1)
010      error message, exit program;
011   if (tpbegin(30, 0) == -1)
012      error message, tpterm, exit program;
013
014   allocate any buffers,
015   make atmi calls
016   check return value
017
018   rtnval = atmicall();
019
020   if (rtnval == -1) {
021       switch(tperrno) {
022       case TPEINVAL:
023           fprintf(stderr, "Invalid arguments were given to
atmicall\n");
024          fprintf(stderr, "e.g., service name was null or flags
wrong\n");
025           break;
026       case ...:
027           fprintf(stderr, ". . .");
028           break;
029
030  Include all error cases described in the atmicall(3) reference
031   page.
032   Other return codes are not possible, so there should be no
033   default within the switch statement.
034
035   if (tpabort(0) == -1) {
036      char *p;
037      fprintf(stderr, "abort was attempted but failed\n");
038      p = tpstrerror(tperrno);
039      userlog("%s", p);
040   }
041   }
042   else
043   if (tpcommit(0) == -1)
044   fprintf(stderr, "REPORT program failed at commit time\n");
```

```
045
046   The following code fragment shows how an application-specific
047    return code can be examined.
048    .
049    .
050    .
051    ret = tpcall("servicename", (char*)sendbuf, 0, (char
**)&rcvbuf, &rcvlen, \
052    (long)0);
053    .
054    .
055    .
056    (void) fprintf(stdout, "Returned tpurcode is: %d\n",
tpurcode);
057
058
059    free all buffers
060    tpterm();
061    exit(0);
062    }
```

The values of `tperrno(5)` 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 `tpurcode`.

The following example shows how to use the `tpstrerrordetail(3c)` function to obtain additional detail when an error is encountered.

Listing 11-2 Handling Errors Using tpstrerrordetail( )

```
001   #include <stdio.h>
002   #include <string.h>
003   #include <atmi.h>/* BEA Tuxedo  Header File */
004   #define LOOP_ITER 100
005   #if defined(__STDC__) || defined(__cplusplus)
006   main(int argc, char *argv[])
007   #else
008   main(argc, argv)
009   int argc;
010   char *argv[];
011   #endif
012   {
```

```
013    char *sendbuf, *rcvbuf;
014    long sendlen, rcvlen;
015    int ret;
016    int i;
017    if(argc != 2) {
018           (void) fprintf(stderr, "Usage: simpcl string\n");
019           exit(1);
020    }
021    /* Attach to BEA Tuxedo System as a Client Process */
022    if (tpinit((TPINIT *) NULL) == -1) {
023           (void) fprintf(stderr, "Tpinit failed\n");
024           exit(1);
025    }
026    sendlen = strlen(argv[1]);
027
028    /* Allocate STRING buffers for the request and the reply */
029
030    if((sendbuf = (char *) tpalloc("STRING", NULL, sendlen+1))
== NULL) {
031           (void) fprintf(stderr,"Error allocating send
buffer\n");
032           tpterm();
033           exit(1);
034    }
035
036    if((rcvbuf = (char *) tpalloc("STRING", NULL, sendlen+1)) ==
NULL) {
037           (void) fprintf(stderr,"Error allocating receive
buffer\n");
038           tpfree(sendbuf);
039           tpterm();
040           exit(1);
041    }
042
043   for( i=0; i<LOOP_ITER; i++) {
044    (void) strcpy(sendbuf, argv[1]);
045
046    /* Request the service TOUPPER, waiting for a reply */
047    ret = tpcall("TOUPPER", (char *)sendbuf, 0, (char
**)&rcvbuf, &rcvlen, (long)0);
048
049    if(ret == -1) {
050           (void) fprintf(stderr, "Can't send request to service
TOUPPER\n");
051           (void) fprintf(stderr, "Tperrno = %d, %s\n", tperrno,
tpstrerror(tperrno));
052
053           ret = tperrordetail(0);
054           if(ret == -1) {
```

```
055                 (void) fprintf(stderr, "tperrodetail()
failed!\n");
056                 (void) fprintf(stderr, "Tperrno = %d, %s\n",
tperrno, tpstrerror(tperrno));
057           }
058           else if (ret != 0) {
059                   (void) fprintf( stderr, "errordetail:%s\n",
060                                 tpstrerrordetail( ret, 0));
061           }
062           tpfree(sendbuf);
063           tpfree(rcvbuf);
064           tpterm();
065           exit(1);
066    }
067    (void) fprintf(stdout, "Returned string is: %s\n", rcvbuf);
068   }
069
070    /* Free Buffers & Detach from System/T */
071    tpfree(sendbuf);
072    tpfree(rcvbuf);
073    tpterm();
074    return(0);
```

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 the *flags* parameter of `tpacall()` to `TPNOTRAN` or `TPNOREPLY`.
- A service must retrieve all asynchronous transaction replies before calling `tpreturn()` or `tpforward()`. 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 the `TPNOTRAN` flag) 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 the `TPNOTRAN` flag set so that the results of the communication are preserved after the transaction is rolled back.
- If a transaction has timed out:
  - The descriptor for the timed-out call becomes stale and any further reference to it returns `TPEBADDESC`.
  - Further calls to `tpgetrply()` or `tprecv()` for any outstanding descriptors return a global state of transaction timeout; the system sets `tperrno(5)` to `TPETIME`.
  - Asynchronous calls can be made with the *flags* parameter of `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 descriptor 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 descriptors (made without the `TPNOTRAN` flag) 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 tperrno(5). The precise meaning of such an error, however, depends on the function that is returning it. The following table lists the functions that return transaction errors and describes possible causes of them.

Table 11-3 Transaction Errors

Function	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()	The function was called for a transaction reply after a request was made without the TPNOTRAN flag.
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.

Function	Cause
`tpconnect()`, `tppost()`, `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 `TPNOTRAN` flag 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; `tperrno(5)` is set to `TPESVCERR`.
  - The *`rval`* argument to `tpreturn()` was set to `TPFAIL`; `tperrno(5)` is set to `TPESVCFAIL`.
  - The *`type`* or *`subtype`* of the reply buffer is not known or not allowed by the caller and, as a result, success or failure cannot be determined; `tperrno(5)` is set to `TPEOTYPE`.
- The transaction times out; `tperrno(5)` is set to `TPETIME`.

- tpcommit() is called by a participant rather than by the originator of a transaction; tperrno(5) 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 call descriptors before explicitly aborting the transaction. However, because these descriptors 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 the *flags* parameter set to 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 the *flags* parameter to TPNOREPLY, TPNOBLOCK, or TPNOTRAN.

Heuristic Decision Errors

The tpcommit() function 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 11-21, 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 11-21 for more information on timeouts.

Effect on the tpcommit() Function

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 tperrno(5) 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.

Effect on the TPNOTRAN Flag

When a process is in transaction mode and makes a communication call with *flags* set to 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 the TPNOTRAN flag, refer to "tpreturn( ) and tpforward( ) Functions" on page 11-25.

tpreturn( ) and tpforward( ) Functions

If you call a process while running in transaction mode, tpreturn( ) and tpforward( ) 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 tpforward( ) should be called until all outstanding descriptors for the communication calls made within the service have been retrieved. If you call tpreturn( ) with outstanding descriptors for which *rval* 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 descriptors for which *rval* 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 *rval* by the application.

You can use tpforward() 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 tpforward(); otherwise, the system invokes tpreturn() with TPFAIL. If you call tpforward() improperly, the system considers the call a processing error and returns a failed message to the requester.

tpterm( ) Function

Use the tpterm() function to remove a client context from an application.

If the client context is in transaction mode, the call fails with TPEPROTO returned in tperrno(5), 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 function 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 functions in your application code. Failure to follow this transaction rule causes indeterminate results. You can use the tpgetlev() function to determine whether a process is already in a global transaction before calling the resource manager's transaction function.

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 `tpforward()`, 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.

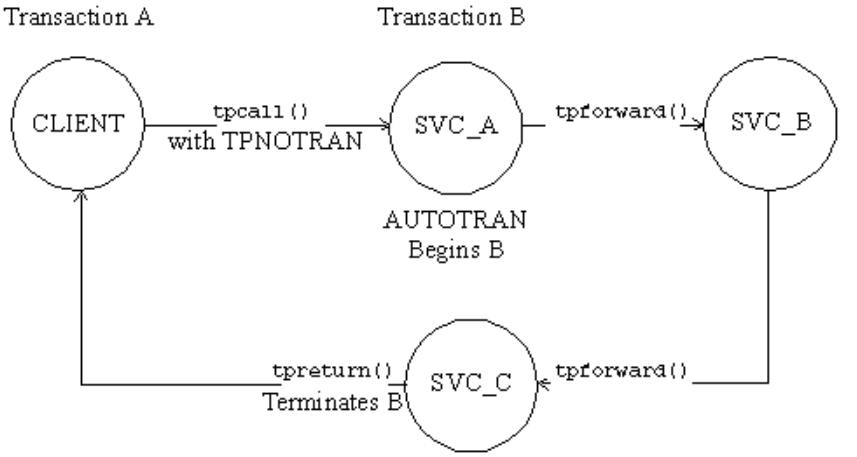
- The success or failure of the called process affects the current transaction. If any fatal transaction errors are encountered by the participant, the current transaction is marked “abort-only.”
- Whether or not the tasks performed by a successful participant are applied depends on the fate of the transaction. In other words, if the transaction is aborted, the work of all participants is reversed.
- The TPNOREPLY flag 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 the TPNOTRAN flag 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 tpforward(), 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 tpforward( ) and tpreturn() with AUTOTRAN” on page 11-29.)
- 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 tpforward( ) 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 *rval*, regardless of the outcome of the transaction.
- Typically, the system returns processing errors, buffer 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()`, `tpsvrdone()` `tpsvrthrinit(3c)`, and `tpsvrthrdone(3c)`, must follow certain rules when used in transactions.

Note: `tpsvrthrinit(3c)` and `tpsvrthrdone(3c)` can be specified for multithreaded applications only. `tpsvrinit()` and `tpsvrdone()` can be specified for both threaded and non-threaded applications.

The BEA Tuxedo system server calls `tpsvrinit()` or `tpsvrthrinit(3c)` during initialization. Specifically, `tpsvrinit()` or `tpsvrthrinit(3c)` is called after the calling process becomes a server but before it starts handling service requests. If `tpsvrinit()` or `tpsvrthrinit(3c)` 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()` or `tpsvrthrinit(3c)` 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()` or `tpsvrthrdone(3c)` 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()` or `tpsvrthrdone(3c)` 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()` or `tpsvrthrdone(3c)`, 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(3c)` function.

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(3c)`. 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 `userlog(3c)` error message file on each machine. The `userlog(3c)` function 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(3c)`

  - Context ID of the thread calling `userlog(3c)`

- Message text

  The text of each message is preceded by the catalog name and number of that message.

- Optional arguments in `printf`(3S) format

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):

```
userlog("Unknown User '%s' \n", usrnm);
```

The resulting log entry appears as follows:

```
162214.mach1!security.23451: Unknown User 'abc'
```

In this example, the process ID for security is `23451`, and the variable `usrnm` contains the value `abc`.

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: LIBSEC_CAT: 999: Unknown User 'abc'
```

In this case, the message catalog name is `LIBSEC_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 'abc'
```

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 variable of type `char *` and use the variable name as the argument to the call.
- Specify the literal text of the message within double quotes, as the argument to the `userlog(3c)` call, as shown in the following example:

```
.
.
.
/* Open the database to be accessed by the transactions.*/
if(tpopen() == -1) {
      userlog("tpsvrinit: Cannot open database %s,
tpstrerror(tperrno)");
      return(-1);
}
.
.
.
```

In this example, the message is sent to the central event log if `tpopen(3c)` returns `-1`.

The `userlog(3c)` signature is similar to that of the UNIX System `printf`(3S) function. The format portion of both functions can contain literal strings and/or conversion specifications for a variable number of arguments.

Debugging Application Processes

Although you can use `userlog(3c)`statements to debug application software, it is sometimes necessary to use a debugger command for more complex problem solving.

The following sections describe how to debug an application on UNIX and Windows 2000 platforms.

Debugging Application Processes on UNIX Platforms

The standard UNIX system debugging command is `dbx`(1). For complete information about this tool, refer to `dbx`(1) in a UNIX system reference manual. If you use the `-g` option to compile client processes, you can debug those processes using the procedures described on the `dbx`(1) reference page.

To run the dbx command, enter the following:

```
dbx client
```

To execute a client process:

1. Set any desired breakpoints in the code.
2. Enter the dbx command.
3. At the dbx prompt (*), type the run subcommand (r) and any options you want to pass to the client program's main().

The task of debugging server programs is more complicated. Normally a server is started using the tmboot command, which starts the server on the correct machine with the correct options. When using dbx, it is necessary to run a server directly rather than through the tmboot command. To run a server directly, enter the r (short for run) subcommand after the prompt displayed by the dbx command.

The BEA Tuxedo tmboot(1) command passes undocumented command-line options to the server's predefined main(). To run a server directly, you must pass these options, manually, to the r subcommand. To find out which options need to be specified, run tmboot with the -n and -d 1 options. The -n option instructs tmboot not to execute a boot; -d 1 instructs it to display level 1 debugging statements. By default, the -d 1 option returns information about all processes. If you want information about only one process, you can specify your request accordingly with additional options. For more information, refer to the *BEA Tuxedo Command Reference*.

The output of tmboot -n -d 1 includes a list of the command-line options passed by tmboot to the server's main(), as shown in the following example:

```
exec server -g 1 -i 1 -u sfmax -U /tuxdir/appdir/ULOG -m 0 -A
```

Once you have the list of required command-line options, you are ready to run the server program directly, with the r subcommand of dbx(1). The following command line is an example:

```
*r -g 1 -i 1 -u sfmax -U /tuxdir/appdir/ULOG -m 0 -A
```

You may not use dbx(1) to run a server that is already running as part of the configuration. If you try to do so, the server exits gracefully, indicating a duplicate server in the central event log.

Debugging Application Processes on Windows 2000 Platforms

On a Windows 2000 platform, a graphical debugger is provided as part of the Microsoft Visual C++ environment. For complete information about this tool, refer to the Microsoft Visual C++ reference manual.

To invoke the Microsoft Visual C++ debugger, enter the `start` command as follows:

```
start msdev -p process_ID
```

Note: For versions of the Microsoft Visual C++ debugger that are earlier than 5.0, enter the `start` command as follows:

```
start msdev -p process_id
```

To invoke the debugger and automatically enter a process, specify the process name and arguments on the `start` command line, as follows:

```
start msdev filename argument
```

For example, to invoke the debugger and enter the `simpcl.exe` process with the `ConvertThisString` argument, enter the following command:

```
start msdev simpcl.exe ConvertThisString
```

When a user-mode exception occurs, you are prompted to invoke the default system debugger to examine the location of the program failure and the state of the registers, stacks, and so on. By default, `Dr. Watson` is used in the Windows 2000 environment uses as the default debugger for user-mode exception failures, while the kernel debugger is used in the Win32 SDK environment.

To modify the default debugger used by the Windows 2000 system for user-mode exception failures, perform the following steps:

1. Run `regedit` or `regedt32`.
2. Within the `HKEY_LOCAL_MACHINE` subtree, navigate to `\SOFTWARE\Microsoft\Windows\CurrentVersion\AeDebug`
3. Double-click on the `Debugger` key to advance into the registry string editor.
4. Modify the existing string to specify the debugger of your choice.

For example, to request the debugger supplied with the Microsoft Visual C++ environment, enter the following command:

```
msdev.exe -p %ld -e %ld
```

Note: For versions of the Microsoft Visual C++ debugger that are earlier than 5.0, enter the following command:

```
msvc.exe -p %ld -e %ld
```

Comprehensive Example

Transaction integrity, message communication, and resource access are the major requirements of an Online-Transaction-Processing (OLTP) application.

This section provides a code sample that illustrates the ATMI transaction, buffer management, and communication routines operating together with SQL statements that access a resource manager. The example is borrowed from the `ACCT` server that is part of the BEA Tuxedo banking application (`bankapp`) and illustrates the `CLOSE_ACCT` service.

The example shows how the `set transaction` statement (line 49) is used to set the consistency level and access mode of the transaction before the first SQL statement that accesses the database. (When read/write access is specified, the consistency level defaults to high consistency.) The SQL query determines the amount to be withdrawn in order to close the account based on the value of the `ACCOUNT_ID` (lines 50-58).

`tpalloc()` allocates a buffer for the request message to the `WITHDRAWAL` service, and the `ACCOUNT_ID` and the amount to be withdrawn are placed in the buffer (lines 62-74). Next, a request is sent to the `WITHDRAWAL` service via a `tpcall()` call (line 79). An SQL `delete` statement then updates the database by removing the account in question (line 86).

If all is successful, the buffer allocated in the service is freed (line 98) and the `TPSVCINFO` data buffer that was sent to the service is updated to indicate the successful completion of the transaction (line 99). Then, if the service was the initiator, the transaction is automatically committed. `tpreturn()` returns `TPSUCCESS`, along with

the updated buffer, to the client process that requested the closing of the account. Finally, the successful completion of the requested service is reported on the status line of the form.

After each function call, success or failure is determined. If a failure occurs, the buffer allocated in the service is freed, any transaction begun in the service is aborted, and the TPSVCINFO buffer is updated to show the cause of failure (lines 80-83). Finally, tpreturn() returns TPFAIL and the message in the updated buffer is reported on the status line of the form.

Note: When specifying the consistency level of a global transaction in a service routine, take care to define the level in the same way for all service routines that may participate in the same transaction.

Listing 11-3 ACCT Server

```
001   #include <stdio.h>              /* UNIX */
002   #include <string.h>             /* UNIX */
003   #include <fml.h>                /* BEA Tuxedo System */
004   #include <atmi.h>               /* BEA Tuxedo System */
005   #include <Usysflds.h>           /* BEA Tuxedo System */
006   #include <sqlcode.h>            /* BEA Tuxedo System */
007   #include <userlog.h>            /* BEA Tuxedo System */
008   #include "bank.h"               /* BANKING #defines */
009   #include "bank.flds.h"          /* bankdb fields */
010   #include "event.flds.h"         /* event fields */
011
012
013   EXEC SQL begin declare section;
014   static long account_id;                    /* account id */
015   static long branch_id;                     /* branch id  */
016   static float bal, tlr_bal;                 /* BALANCE    */
017   static char acct_type;                     /* account type*/
018   static char last_name[20], first_name[20]; /* last name, first name */
019   static char mid_init;                      /* middle initial */
020   static char address[60];                   /* address    */
021   static char phone[14];                     /* telephone */
022   static long last_acct;                     /* last account branch gave */
023   EXEC SQL end declare section;

024   static FBFR *reqfb;             /* fielded buffer for request message */
025   static long reqlen;             /* length of request buffer */
026   static char amts[BALSTR];       /* string representation of float */

027   code for OPEN_ACCT service
```

```
028   /*
029    * Service to close an account
030    */

031   void
032   #ifdef __STDC__
033   LOSE_ACCT(TPSVCINFO *transb)

034   #else

035   CLOSE_ACCT(transb)
036   TPSVCINFO *transb;
037   #endif

038   {
039      FBFR *transf;              /* fielded buffer of decoded message */

040      /* set pointer to TPSVCINFO data buffer */
041         transf = (FBFR *)transb->data;

042      /* must have valid account number */
043      if (((account_id = Fvall(transf, ACCOUNT_ID, 0)) < MINACCT) ||
044       (account_id > MAXACCT)) {
045         (void)Fchg(transf, STATLIN, 0, "Invalid account number", (FLDLEN)0);
046         tpreturn(TPFAIL, 0, transb->data, 0L, 0);
047      }

048      /* Set transaction level */
049      EXEC SQL set transaction read write;

050      /* Retrieve AMOUNT to be deleted */
051      EXEC SQL declare ccur cursor for
052       select BALANCE from ACCOUNT where ACCOUNT_ID = :account_id;
053      EXEC SQL open ccur;
054      EXEC SQL fetch ccur into :bal;
055      if (SQLCODE != SQL_OK) {               /* nothing found */
056       (void)Fchg(transf, STATLIN, 0, getstr("account",SQLCODE), (FLDLEN)0);
057       EXEC SQL close ccur;
058       tpreturn(TPFAIL, 0, transb->data, 0L, 0);
059      }

060      /* Do final withdrawal */

061      /* make withdraw request buffer */
062     if ((reqfb = (FBFR *)tpalloc("FML",NULL,transb->len)) == (FBFR *)NULL) {
063       (void)userlog("tpalloc failed in close_acct\n");
064       (void)Fchg(transf, STATLIN, 0,
065           "Unable to allocate request buffer", (FLDLEN)0);
```

```
066       tpreturn(TPFAIL, 0, transb->data, 0L, 0);
067      }
068      reqlen = Fsizeof(reqfb);
069      (void)Finit(reqfb,reqlen);

070      /* put ID in request buffer */
071      (void)Fchg(reqfb,ACCOUNT_ID,0,(char *)&account_id, (FLDLEN)0);

072      /* put amount into request buffer */
073      (void)sprintf(amts,"%.2f",bal);
074      (void)Fchg(reqfb,SAMOUNT,0,amts, (FLDLEN)0);

075      /* increase the priority of this withdraw */
076      if (tpsprio(PRIORITY, 0L) == -1)
077       (void)userlog("Unable to increase priority of withdraw");

078      /* tpcall to withdraw service to remove remaining balance */
079      if (tpcall("WITHDRAWAL", (char *)reqfb, 0L, (char **)&reqfb,
080       (long *)&reqlen,TPSIGRSTRT) == -1) {
081       (void)Fchg(transf, STATLIN, 0,"Cannot make withdrawal", (FLDLEN)0);
082       tpfree((char *)reqfb);
083       tpreturn(TPFAIL, 0,transb->data, 0L, 0);
084      }

085      /* Delete account record */

086      EXEC SQL delete from ACCOUNT where current of ccur;
087      if (SQLCODE != SQL_OK) {               /* Failure to delete */
088       (void)Fchg(transf, STATLIN, 0,"Cannot close account", (FLDLEN)0);
089        EXEC SQL close ccur;
090        tpfree((char *)reqfb);
091        tpreturn(TPFAIL, 0, transb->data, 0L, 0);
092      }
093      EXEC SQL close ccur;

094      /* prepare buffer for successful return */
095      (void)Fchg(transf, SBALANCE, 0, Fvals(reqfb,SAMOUNT,0), (FLDLEN)0);
096      (void)Fchg(transf, FORMNAM, 0, "CCLOSE", (FLDLEN)0);
097      (void)Fchg(transf, STATLIN, 0, " ", (FLDLEN)0);
098      tpfree((char *)reqfb);
099      tpreturn(TPSUCCESS, 0, transb->data, 0L, 0);
100   }
```