The Enterprise Middleware Solution

BEA TUXEDO

Programmer's Guide

BEA TUXEDO Release 6.5
Document Edition 6.5
February 1999

BEA TUXEDO Programmer's Guide

Document Edition	Date	Software Version
6.5	February 1999	BEA TUXEDO Release 6.5

Contents

1. Introduction and Overview

The BEA TUXEDO System Development Environment 1-1
Client Processes 1-2
    Basic Client Operation 1-2
        Client Sending Repeated Service Requests 1-3
    Server Processes and Service Subroutines 1-3
    Basic Server Operation 1-3
    Servers as Requesters 1-5
    The ATMI Primitives 1-6
        An Overview of X/Open's TX Interface 1-8
    Typed Buffers 1-9
        Using VIEW and FML Buffers 1-10
        Relationship Between Some VIEW Buffers and FML 1-10
        Corresponding Data Type Definitions 1-13
        Creating Header Files from View Descriptions 1-14
        Header Files from Field Tables 1-15
    Other Header Files 1-15
    Environment Variables 1-16
    Configuration File 1-17
        Making the Configuration Usable 1-17
    The Bulletin Board 1-18
        Starting and Stopping an Application 1-18
Service Gateway 1-18
    Programming Paradigms 1-18
    Buffer Types 1-19
    Configuration 1-19

Examples 1-19

2. Writing Client Programs

About This Chapter 2-1
Examples Taken from the Sample Application 2-1
Preliminaries 2-2
Client Naming 2-2
Unsolicited Notification 2-3
Security and Client Authentication 2-4
Writing Client Programs with SECURITY Set 2-6
Getting the Security Data 2-7
Joining the Application 2-7
Allocating the TPINIT Buffer 2-10
The Application Key 2-10
Joining and Leaving an Application 2-12
Buffer Management 2-15
Typed Buffers for Messages 2-15
Buffer Types: STRING 2-15
Buffer Types: CARRAY 2-16
Buffer Types: FML and FML32 2-16
Buffer Types: VIEW, VIEW32, X_C_TYPE, and X_COMMON 2-16
Buffer Types: Summary 2-17
ATMI Buffer Primitives 2-18
Allocating a Typed Buffer 2-18
tpalloc Examples 2-20
What About FML Buffer Management Functions? 2-20
Putting Data in the Buffer 2-21
Resizing a Typed Buffer 2-22
Checking for Buffer Type 2-24
Freeing a Typed Buffer 2-25
Service Calls 2-26
Sending Synchronous Messages: tpcall() 2-26
Values for the flags Argument: tpcall() 2-29
Examples of the Use of flags Arguments 2-31
Sending Asynchronous Messages: tpacall() 2-34

Values for the flags Argument: tpacall() 2-35
Getting an Asynchronous Reply: tpgetrply() 2-37
Getting and Setting Priority 2-37
Initiating a Conversational Connection 2-40
Sending a Broadcast Message 2-40
Compiling Client Programs 2-41
The buildclient Command 2-41
The buildclient -o Option 2-41
The buildclient -f and -l Options 2-41

3. Writing Service Routines

Writing Request/Response Services 3-1
Examples Taken from the Sample Application 3-1
Application Service Template 3-2
The TPSVCINFO Structure 3-2
The tpreturn() and tpforward() Functions 3-9
Sending Replies 3-9
Forwarding Requests 3-16
Sending Unsolicited Messages 3-18
Advertising, Unadvertising Services 3-20
System-Supplied Servers and Subroutines 3-22
System-Supplied Servers 3-22
AUTHSVR 3-22
The BEA TUXEDO System main() 3-23
BEA TUXEDO System-Supplied Subroutines 3-24
tpsvrinit() 3-24
tpsvrdone() 3-27
Compiling Subroutines to Build Servers 3-29
The buildserver Command 3-29
The buildserver -o Option 3-29
The buildserver -f and -l Options 3-29
The buildserver -r Option 3-30
The buildserver -s Option 3-30
Using C++ 3-31

4. Conversational Clients and Services

Writing Conversational Clients and Services 4-1
Conversational Mode 4-2
The Connection Descriptor 4-2
Buffer Management 4-2
Joining an Application 4-3
Establishing a Connection 4-3
Values for the flags Argument: tpconnect() 4-4
Sending 4-5
Values for the flags Argument: tpsend() 4-6
Receiving 4-7
Values for the flags Argument: tprecv() 4-8
Ending a Conversation 4-9
Subordinate Calls tpreturn() 4-9
Hierarchy of Connections and tpreturn() 4-10
Ending a Conversation: Summary 4-11
Events and Their Significance 4-11
Disorderly Disconnection 4-12
Request/Response Calls and Conversations 4-13
Configuration Parameters 4-13
Building Conversational Clients and Servers 4-14

5. Global Transactions in BEA TUXEDO System

Introduction 5-1
What Is a Global Transaction? 5-2
ATMI Transaction Primitives 5-3
Explicitly Defining a Global Transaction 5-3
Starting the Transaction 5-5
Terminating the Transaction 5-9
Implicitly Defining a Global Transaction 5-13
In a Client Process 5-13
In a Service Routine 5-13

6. Using the Event Broker

Introduction 6-1

Notification Actions 6-2
User-Defined and System-Defined Events 6-3
Event Broker/Event Monitor Servers 6-4
Programming Interface 6-4
Posting Events 6-5
tppost() Arguments: eventname 6-5
tppost() Arguments: data and len 6-5
tppost() Arguments: flags 6-5
Example of Event Posting 6-6
Subscribing to Events 6-7
tpsubscribe() Arguments: eventexpr 6-7
tpsubscribe() Arguments: filter 6-8
tpsubscribe() Arguments: ctl 6-8
Notification Via Unsolicited Message 6-8
Notification Via Service Call or Reliable Queue 6-9
tpsubscribe() Arguments: flags 6-10
Example of Event Subscription 6-10

7. Error Management

Introduction 7-1
Communicating Errors 7-2
Values of tperrno 7-2
Protocol Errors 7-3
BEA TUXEDO System Errors 7-3
Operating System Errors 7-3
Errors from Invalid Arguments 7-4
Other Possible Error Categories 7-4
No Entry Errors 7-5
Permission Errors 7-5
Resource Manager Errors 7-6
Transaction-Related Errors 7-6
Typed Buffer Errors 7-7
Call Descriptor Errors 7-7
General Communication Call Errors 7-9
Conversational Errors 7-9

Time-Out Errors 7-10
Errors Leading to Abort 7-10
Errors Signaling Heuristic Decisions 7-11
Application-Specific Errors 7-11
How to Deal with Errors 7-11
Fatal Transaction Errors 7-14
Time-Out 7-15
Blocking vs. Transaction Time-Out 7-15
Effect on tpcommit() 7-16
Effect of the TPNOTRAN Flag 7-16
Roles of tpreturn() and tpforward() 7-17
Service in Same Transaction as Caller 7-18
Service in Different Transaction with AUTOTRAN Set 7-18
Service Starts New Explicit Transaction 7-19
Transaction Rules 7-20
Communication Etiquette 7-20
BEA TUXEDO System-Supplied Subroutines 7-21
tpsvrinit() 7-21
tpsvrdone() 7-21
Leaving the Application 7-22
Global Transactions and Resource Managers 7-22
Comprehensive Example 7-23
The Central Event Log 7-27
How the Log Is Named 7-27
What Log Entries Look Like 7-27
How to Write to the Event Log 7-29
Debugging Application Processes 7-30

CHAPTER

1 Introduction and Overview

The BEA TUXEDO System Development Environment

The purpose of this chapter is to describe the environment in which you will be writing code for a BEA TUXEDO system application.

In addition to the C code that expresses the logic of your application, you will be using the Application-Transaction Monitor Interface (ATMI), which refers to the interface between the BEA TUXEDO system transaction monitor and your application. The ATMI primitives are C language functions that resemble UNIX system calls, but they have the specific purpose of implementing the communication among application modules running under the control of the BEA TUXEDO system transaction monitor, including all the associated resources you need.

As you might remember from the *BEA TUXEDO Product Overview*, the BEA TUXEDO system uses an enhanced client-server architecture. Chapters 2 through 7 of this book describe how the ATMI primitives are used in writing and debugging clients and services. This chapter provides some of the context within which you will be doing that work.

Client Processes

A client process takes user input and sends it as a service request to a server process that offers the requested service.

Basic Client Operation

A client process uses one ATMI primitive to join an application, allocates a message buffer by using another ATMI primitive, and uses still others to send the buffer to a server and receive the reply.

The operation of a basic client process can be summarized by the pseudo-code shown in Listing 1-1.

Listing 1-1 Pseudo-code for a 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 statements in Listing 1-1 are implemented with ATMI primitives. Placing user input in a buffer and passing the reply to the user are implemented with C language functions.

When client programs are ready to test, you use the `buildclient`(1) command to compile and link edit them.

Client Sending Repeated Service Requests

A client may send and receive any number of service requests before leaving the application. These can be sent as a series of request/response calls or, if it is important to carry state information from one call to the next, a connection to a conversational server can be set up. The logic within the client program is about the same, but different ATMI primitives are used.

Server Processes and Service Subroutines

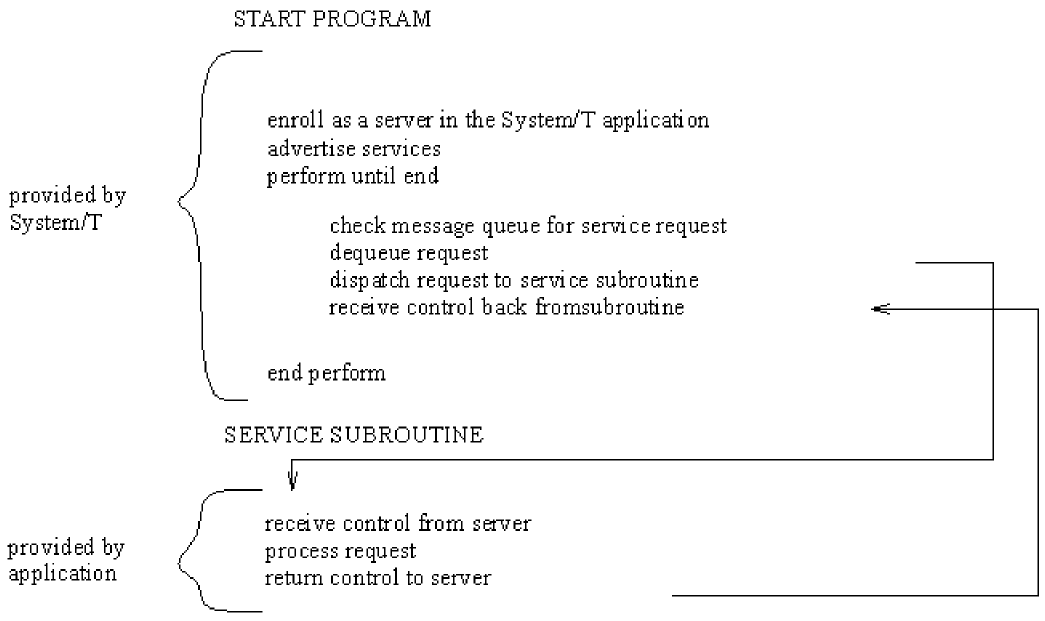
Servers are processes that provide one or more services. They continually check their message queue for service requests and dispatch them to the appropriate service subroutines.

Basic Server Operation

Applications combine their service subroutines with the `main()` that BEA TUXEDO provides in order to build server processes. This system supplied `main`() is a set of predefined functions. It performs server initialization and termination and allocates buffers to receive and dispatch incoming requests to service routines. All of this processing is transparent to the application.

Server and a service subroutine interaction can be summarized by the pseudo-code shown in Figure 1-1.

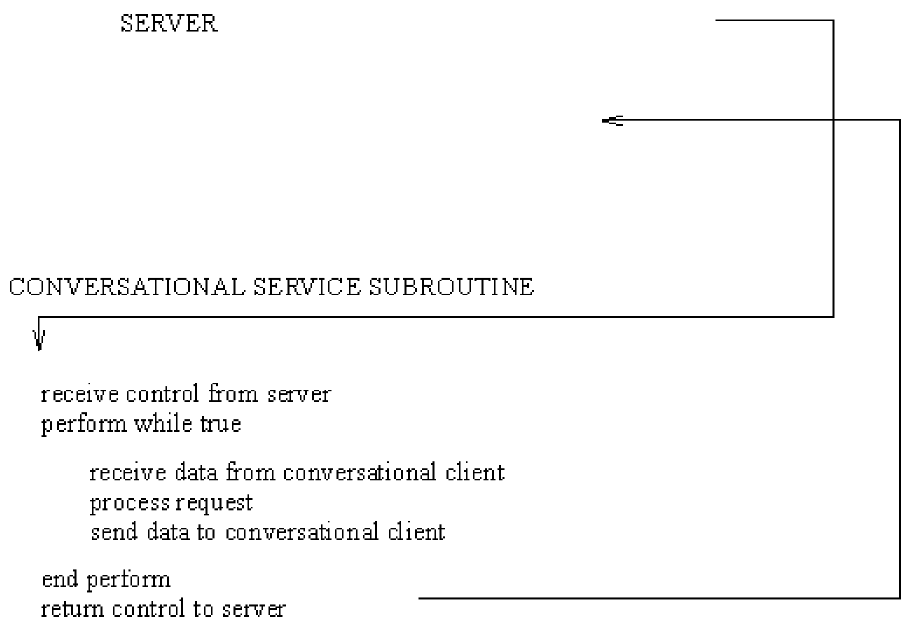
Figure 1-1 Pseudo-code for a Request/Response Server and a Service Subroutine


START PROGRAM
provided by System/T
enroll as a server in the System/T application
advertise services
perform until end
check message queue for service request
dequeue request
dispatch request to service subroutine
receive control back fromsubroutine
end perform
SERVICE SUBROUTINE
provided by application
receive control from server
process request
return control to server

After some initialization a server allocates a buffer, waits until a request message is put on its message queue, dequeues the request, and dispatches it to a service subroutine for processing. If a reply is needed, the reply is considered part of request processing.

The conversational paradigm is somewhat different. Pseudo-code is shown in Figure 1-2.

Figure 1-2 Pseudo-code for a Conversational Service Subroutine


SERVER
CONVERSATIONAL SERVICE SUBROUTINE
receive control from server
perform while true
receive data from conversational client
process request
send data to conversational client
end perform
return control to server

The BEA TUXEDO system-supplied `main()` contains the code needed to enroll as a server, advertise services, allocate buffers, and dequeue request messages. The ATMI primitives are used in service subroutines that process requests. When they are ready to compile and test, service subroutines are link edited with the server `main()` by means of the `buildserver`(1) command to form an executable server.

Servers as Requesters

The serially reusable architecture of servers is particularly significant if the operation requested by the user is logically divisible into several services, or several iterations of the same service. Such operations can be overlapped by having a server assume the role of a client and hand off part of the task to another server as part of fulfilling the original client's request. In such a capacity the server becomes a requester. Both clients and servers can be requesters. In fact, a client can only be a requester. The coding model for such a system is easily accomplished with the routines that are provided by ATMI.

A request/response server can also forward a request to another server. This is different from becoming a requester. In this case, the server does not assume the role of client since no reply is expected by the server that forwards a request. The reply is expected by the original client.

The ATMI Primitives

The Application-Transaction Monitor Interface is a reasonably compact set of primitives used to open and close resources, begin and end transactions, allocate and free buffers, and provide the communication between clients and servers. Table 1-1 summarizes them. Each routine is documented in Section 3C of the *BEA TUXEDO Reference Manual*.

Table 1-1 ATMI Primitives

Group	Name	Operation
Application Interface	`tpinit()`	join an application
	`tpterm()`	leave an application
Buffer Management Interface	`tpalloc()`	allocate a buffer
	`tprealloc()`	re-size a buffer
	`tpfree()`	free a buffer
	`tptypes()`	get buffer type
Request/Response Communication Interface	`tpcall()`	send a request, wait for answer
	`tpacall()`	send request asynchronously
	`tpgetrply()`	get reply after asynchronous call
	`tpcancel()`	cancel communications handle for outstanding reply
	`tpgprio()`	get priority of last request
	`tpsprio()`	set priority of next request

Table 1-1 ATMI Primitives

Group	Name	Operation
Conversational Interface	`tpconnect()`	begin a conversation
	`tpdiscon`	end a conversation
	`tpsend()`	send data in conversation
	`tprecv()`	receive data in conversation
Unsolicited Notification Interface	`tpnotify()`	notify by client id
	`tpbroadcast()`	notify by name
	`tpsetunsol()`	set unsolicited message handling routine
	`tpgetunsol()`	get unsolicited message
	`tpchkunsol()`	check for unsolicited messages
Transaction Management Interface	`tpbegin()`	begin a transaction
	`tpcommit()`	commit the current transaction
	`tpabort()`	abort the current transaction
	`tpgetlev()`	check if in transaction mode
Service Routine Template	`tpservice()`	start a service
	`tpreturn()`	end service routine
	`tpforward()`	forward request and end service routine
Dynamic Advertisement Interface	`tpadvertise()`	advertise a service name
	`tpunadvertise()`	unadvertise a service name
Resource Manager Interface	`tpopen()`	open a resource manager
	`tpclose()`	close a resource manager
Event Broker/ Event Monitor Interface	`tppost()`	post an event
	`tpsubscribe()`	subscribe to an event
	`tpunsubscribe()`	unsubscribe to an event

An Overview of X/Open's TX Interface

In addition to ATMI's transaction management verbs, the BEA TUXEDO system also supports X/Open's TX Interface for defining and managing transactions. Because X/Open used ATMI's transaction demarcation verbs as the base for the TX Interface, the syntax and semantics of the TX Interface are quite similar to ATMI.

Table 1-2 introduces the routines in the TX Interface and highlights the main differences with their corresponding ATMI routines. For maximum portability, the TX routines can be used in place of the ATMI routines shown in Table 1-2.

Table 1-2 TX Verbs

TX Verbs	Corresponding ATMI Verbs	Main Differences
`tx_begin`	`tpbegin`	Timeout value not passed as argument to `tx_begin`. See `tx_set_transaction_timeout`.
`tx_close`	`tpclose`	None
`tx_commit`	`tpcommit`	`tx_commit` can optionally start a new transaction before it returns. This is known as a "chained" transaction.
`tx_info`	`tpgetlev`	`tx_info` returns the settings of transaction characteristics set via the three `tx_set_*` routines.
`tx_open`	`tpopen`	None
`tx_rollback`	`tpabort`	`tx_rollback` supports chained transactions.
`tx_set_commit_return`	`tpscmt`	None
`tx_set_transaction_control`	None	Defines whether the application is using chained or unchained transactions.
`tx_set_transaction_timeout`	`tpbegin`	Transaction timeout parameter separated from `tx_begin`.

The TX interface requires that `tx_open()` be called before using any other TX verbs. Thus, even if a client or a server is not accessing an XA-compliant resource manager, it must call `tx_open()` before it can use `tx_begin()`, `tx_commit()`, and `tx_rollback()` to define transactions.

Listing 1-2 contains an example of how the TX Interface can be used to support chained transactions. Note that `tx_begin()` must be used to start the first of a series of chained transactions. Also, note that before calling `tx_close()`, the application must switch to unchained transactions so that the last `tx_commit()` or `tx_rollback()` does not start a new transaction.

Listing 1-2 Chained Transactions Using TX Verbs

```
tx_open();
tx_set_transaction_control(TX_CHAINED);
tx_set_transaction_timeout(120);
tx_begin();
do_forever {
       do work as part of transaction;
       if (no more work exists)
               tx_set_transaction_control(TX_UNCHAINED);
       if (work done was successful)
               tx_commit();
       else
               tx_rollback();
       if (no more work exists)
               break;
}
tx_close();
```

Typed Buffers

Messages are passed to servers in typed buffers. Why “typed?” Well, different types of data require different software to initialize the buffer, send and receive the data and perhaps encode and decode it, if the buffer is passed between heterogeneous machines. Buffers are designated as being of a specific type so the routines appropriate to the buffer and its contents can be invoked. These issues are typically not of concern to application developers, but more details can be found in `buffer`(3c), `tuxtypes`(5), and `typesw`(5).

The BEA TUXEDO system provides nine buffer types for messages: `STRING`, `CARRAY`, `VIEW`, `VIEW32`, `FML`, `FML32`, `X_OCTET`, `X_COMMON`, and `X_C_TYPE`. Applications can define additional types as needed. Consult the manual pages referred to above and *Administering the BEA TUXEDO System*.

The STRING buffer type is used when the data is an array of characters that terminates with the null character.

The data in a CARRAY buffer is an undefined array of characters, any of which can be null. The CARRAY is not self-describing and the length must always be provided when transmitting this buffer type. The X_OCTET buffer type is equivalent to CARRAY.

The VIEW type is a C structure that the application defines and for which there has to be a view description file. Buffers of the VIEW type must have subtypes, which designate individual data structures. The X_C_TYPE buffer type is equivalent to VIEW. The X_COMMON buffer type is similar to VIEW but is used for both COBOL and C programs so field types should be limited to short, long, and string. The VIEW32 buffer type is similar to VIEW but allows for larger character fields, more fields, and larger overall buffers.

An FML buffer is a proprietary BEA TUXEDO system type of self-defining buffer where each data field carries its own identifier, an occurrence number, and possibly a length indicator. This type provides great flexibility at the expense of some processing overhead in that all data manipulation is done via FML function calls rather than native C statements.

The FML32 data type is similar to FML but allows for larger character fields, more fields, and larger overall buffers.

Using VIEW and FML Buffers

If you are using the VIEW or FML buffer types, some preliminary work is required to create view description files or field table files. In the case of VIEWs, a description file must exist and must be available to client and server processes that use a data structure described in the VIEW. For FML buffers, a field table file containing descriptions of all fields that may be in the buffer must be available.

Relationship Between Some VIEW Buffers and FML

There are two kinds of VIEW buffers. One is based on an FML buffer. The other VIEW buffer is independent; it is simply a C structure. Both types are described in view description files and compiled with viewc(1), the BEA TUXEDO system view compiler. We’re going to talk first about the FML variety.

FML Views

BEA TUXEDO System FML is a family of functions, some of which convert an FML buffer into a C structure or vice versa. The C structure that is derived from the fielded buffer is referred to as an FML VIEW. The reason for converting FML buffers to C structures and back again is that while FML buffers provide data independence and convenience, they do involve processing overhead because they must be manipulated using FML function calls. C structures, while not providing flexibility, offer the performance required for lengthy manipulations on buffer data. If enough manipulation of the data is called for, you can improve the performance of your programs if you transfer fielded buffer data to C structures, operate on the data using normal C functions, and then put the data back into the FML buffer for storage or message transmission.

There are slight differences between a view description of an FML-based view and one that is independent of FML. Listing 1-3 shows a view description file with all of the available data types. The file is myview.v and the structure is based on an FML buffer. Note that the CARRAY1 field has a count of 2 occurrences and has the “C” count flag to indicate that an additional count element should be created in the structure so the application can indicate how many of the occurrences are actually being used. It also has the “L” length flag such that there is a length element (which occurs twice, once for each occurrence of the field) indicating how many of the characters the application has populated.

Listing 1-3 View Description File for FML View

```
VIEW MYVIEW
$ /* View structure */
#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
```

FML Field Table Files

Field table files are always required when using FML records, including the use of FML-dependent VIEWS. A field table file maps the logical name of a field in an FML buffer to a field identifier that uniquely identifies the field.

An example that could be used with the view shown in Listing 1-3 is shown in Listing 1-4.

Listing 1-4 The myview.flds Field Table File

```
# 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   -       -
```

Independent VIEWs

Listing 1-5 shows the view description file, similar to the example in Listing 1-3, but for a VIEW independent from FML.

Listing 1-5 View Description File for Independent Views

```
$ /* 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 in this view description, the format is similar to the FML-dependent view, except that the columns fbname and null in the file are ignored by the view compiler. These columns are not relevant when an FML buffer does not stand behind the view, but it is necessary to place some value (a dash, for example) in these columns to serve as a placeholder.

Corresponding Data Type Definitions

The C float and double fields correspond to COBOL COMP-1 and COMP-2, respectively.

The field types long and short correspond to S9(9) COMP-5 and S9(4) COMP-5 respectively in COBOL. (The use of COMP-5 is for use with MicroFocus COBOL so that the COBOL integer fields match the data format of the corresponding C fields; the data type for VS COBOL II would simply be COMP.)

The dec_t type maps to a COBOL COMP-3 packed decimal field. Packed decimals exist in the COBOL environment as two decimal digits packed into each byte with the low-order half byte used to store the sign. The length of a packed decimal may be 1 to 9 bytes with storage available for 1 to 17 digits and a sign. The dec_t field type is supported within the VIEW definition for the conversion of packed decimals between the C and the COBOL environments. The dec_t field is defined in a VIEW with a size of two numbers separated by a comma. The number to the left of the comma is the total number of bytes that the decimal occupies in COBOL. The number to the right is the number of digits to the right of the decimal point in COBOL. The formula for conversion to the COBOL declaration is:

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

For example, say a size of 6,4 is specified in the VIEW. There are 4 digits to the right of the decimal point, 7 digits to the left, and the last half byte stores the sign. The COBOL application programmer would represent this as 9(7)V9(4), with the V representing the decimal point between the number of digits to each side. Note that there is no dec_t type supported in FML; if FML-dependent VIEWs are used, then the field must be mapped to a C type in the VIEW file (for instance, the packed decimal can be mapped to an FML string field and the mapping functions do the conversion between the formats).

A decimal field can be initialized and accessed in C using the functions described on the decimal(3c) reference page.

Creating Header Files from View Descriptions

View description files are source files. To use the view in a program, you need a header file that defines the structures in the view. You can create a header file from the `myview.v` view description file by invoking the view compiler, `viewc`(1). `viewc` creates two files. One is the header file and the other is the binary version of the source description file, `myview.V`. This binary file must be in the environment when a `VIEW` buffer is allocated. For an `FML`-dependent `VIEW`, the compiler is invoked as follows.

```
viewc myview.v
```

The header file it creates from the `myview.v` description file is shown in Listing 1-6.

Listing 1-6 Header File Created for FML View

```
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];
};
```

To compile a view description of an independent view, use the `-n` option on the command line, as follows.

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

The header file created is the same with or without the `-n` option. Header files for views must be brought into client programs and service subroutines with `#include` statements.

For use with `VIEW32`, the `viewc32` command should be used.

Header Files from Field Tables

To create a field header file from the field table file, use the `mkfldhdr`(1) command. For example:

```
mkfldhdr myview.flds
```

creates a file called `myview.flds.h` that can be `#include`'d in a service routine or client program so you can refer to fields by their symbolic names. The `myview.flds.h` header file produced by `mkfldhdr` from this field table file is shown in Listing 1-7.

Listing 1-7 The 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 */
#defineCARRAY1    ((FLDID)49270)   /* number: 118 type: carray */
```

For use with `FML32`, the `mkfldhdr32` command should be used.

Other Header Files

If you are using `FML` or `VIEW` typed buffers, `#include` the header files generated from their field table files or view description files as described above.

In addition, all BEA TUXEDO system application programs must `#include` the `atmi.h` header file.

If you are using `FML` buffers, `#include` the `fml.h` header file in your programs.

Environment Variables

Environment variables needed either for clients or service routines associated with a server can be set in `ENVFILE`s that are specified in the configuration file. The environment variables that might have to be set for field tables and view descriptions, for example, are summarized in Table 1-3.

Table 1-3 BEA TUXEDO System Environment Variables

Variable	Contains	Used By
`FIELDTBLS`	comma-separated list of field table file names	client and server processes using `FML` buffers
`FLDTBLDIR`	colon-separated list of directories to be used to find field table files with relative file names	client and server processes using `FML` buffers
`VIEWFILES`	comma-separated list of binary view description files	client and server processes using `VIEW` buffers
`VIEWDIR`	colon-separated list of directories to be used to find binary view description files	client and server processes using `VIEW` buffers

For the `FML32` and `VIEW32` record types, the environment variables are suffixed with `32`, that is, `FLDTBLDIR32`, `FIELDTBLS32`, `VIEWFILES32`, and `VIEWDIR32`.

The `CC` and `CFLAGS` environment variables are used by the `buildclient`(1) and `buildserver`(1) commands. You may want to set them in your environment to make compilation of clients and servers more convenient. Set `CC` to the command that invokes the C compiler. It defaults to `cc`. Set `CFLAGS` to the link edit flags you may want to use on the compile command line. Setting this variable is optional.

The location of the BEA TUXEDO system binary files must be known to your application. It is the convention to install the BEA TUXEDO system software under a root directory whose location is specified in the `TUXDIR` environment variable. `$TUXDIR/bin` must be included in your `PATH` in order for your application to locate the executables for BEA TUXEDO system commands.

Configuration File

The configuration file specifies the configuration of an application to the BEA TUXEDO system. For a BEA TUXEDO system application in production, it is the responsibility of the BEA TUXEDO system administrator to set up a configuration file that defines the application. In the development environment, the responsibility may be delegated to application programmers to create their own.

If you are faced with the task of creating a configuration file, here are some suggestions:

- Borrow a file that already exists. For example, the file `ubbshm` that comes with the sample application is a good starting point.
- Keep it simple. For test purposes, set your application up as a shared memory, single processor system. Use regular UNIX system files for your data.
- Make sure the `IPCKEY` parameter in the configuration file does not conflict with any others that may be in use at your installation. You should probably check this with your BEA TUXEDO system administrator.
- Set the `UID` and `GID` parameters so that you are the owner of the configuration.
- Read the documentation. The configuration file is documented in the `ubbconfig`(5) page in the *BEA TUXEDO Reference Manual* and in the book *Administering the BEA TUXEDO System*.

Making the Configuration Usable

The configuration file is an ASCII file. To make it usable, you have to run `tmloadcf`(1) to convert it to a binary file. The `TUXCONFIG` environment variable must be set to the pathname for the binary file, and exported.

The Bulletin Board

The bulletin board is the BEA TUXEDO system name for a group of data structures in a segment of shared memory that is allocated from information stored in `TUXCONFIG` when the application is booted. Both client and server processes attach to the bulletin board. Part of the bulletin board associates service names with the queue address of servers that advertise that service. Clients send their requests to the name of the service they want to invoke, rather than to a specific address.

All processes that are part of a BEA TUXEDO application share this IPC resource.

Starting and Stopping an Application

Execute the `tmboot`(1) command to bring up an application. The command gets the IPC resources needed by the application, and starts administrative processes and the application servers.

When it is time to bring the application down, execute the `tmshutdown`(1) command. `tmshutdown` stops the servers and releases the IPC resources used by the application, except any that might be used by the database resource manager.

Service Gateway

`GWTUX2TE` and `GWTE2TUX` are BEA TUXEDO system servers that provide connectivity between BEA TUXEDO and BEA TOP END systems. `GWTUX2TE` provides connectivity between BEA TUXEDO clients and BEA TOP END servers. `GWTE2TUX` provides connectivity between BEA TOP END clients and BEA TUXEDO servers. One or both of these gateway servers may be configured.

Programming Paradigms

Gateway servers support request/response messages only. The following BEA TUXEDO client API calls for sending and receiving are allowed:

♦ `tpcall`

- `tpacall` (with or without `TPNOREPLY` flag)
- `tpgetrply`
- `tpforward`

BEA TOP END servers cannot set the `APPL_CONTEXT` flag. If this flag is set, the gateway server dissolves the BEA TOP END dialog and returns an error (`TPESVCFAIL`) to the BEA TUXEDO client.

The following BEA TOP END client API calls are allowed:

- `tp_client_send`
- `tp_client_receive`

Buffer Types

The gateway servers support BEA TUXEDO `CARRAY` (`X_OCTET`) buffers only. Attempts to send other types of buffers from a BEA TUXEDO application generate an error (`TPESVCFAIL`) which is logged by the gateway server.

Configuration

The `GWTUX2TE` and `GWTE2TUX` gateway servers use the BEA TOP END remote client and remote server services. `GWTUX2TE` assumes the role of a BEA TOP END client and makes use of the remote client services. `GWTE2TUX` assumes the role of a BEA TOP END server and makes use of the remote server services. Therefore, you must provide a BEA TOP END remote client/server configuration file on any BEA TUXEDO node running these gateway processes.

Examples

The following example shows how gateway servers are defined in the BEA TUXEDO `UBBCONFIG` file and in the BEA TOP END service definition file.

In this example, a BEA TUXEDO client issues `tpcall` to the `RSERVICE` service. The request is forwarded (via the `GWTUX2TE` gateway) to a BEA TOP END system (`pluto`) and invokes a BEA TOP END service (`RPRODUCT:RFUNC`).

Similarly, a BEA TOP END client issues `tp_client_send`, specifying `LPRODUCT` as the `PRODUCT` and `LFUNC` as the `FUNCTION`. The request is forwarded (via the `GWTE2TUX` gateway) to the BEA TUXEDO system and invokes a BEA TUXEDO service (`LSERVICE`).

Listing 1-8 BEA TUXEDO UBBCONFIG File

```
##############
#UBBCONFIG
*GROUPS
TOPENDGRP  GRPNO=1

#
*SERVERS
GWTE2TUX SRVGRP="TOPENDGRP" SRVID=1001 RESTART=Y MAXGEN=3 GRACE=10
       CLOPT=”-- -f servicedefs -R 30”
GWTUX2TE SRVGRP=”TOPENDGRP” SRVID=1002 RESTART=Y MAXGEN=3 GRACE=10
       MIN=5 MAX=5
       CLOPT=”-- -f servicedefs”
```

Listing 1-9 BEA TOP END Service Definition File

```
############
#service definition file
*TE_LOCAL_SERVICES
DEFAULT: PRODUCT=LPRODUCT
LSERVICE FUNCTION=LFUNC

*TE_REMOTE_SERVICES
RSERVICE PRODUCT=RPRODUCT FUNCTION=RFUNC
```

Listing 1-10 BEA TOP END Remote Configuration File

```
# TOP END remote configuration file
[top end configuration file]
[component type] remote server
[system] pluto
[primary node] //topendmach       5000
```

CHAPTER

2 Writing Client Programs

About This Chapter

The sections that follow describe the ATMI functions that enable a client program to

- ♦ control the client name that is posted in the bulletin board
- ♦ comply with the level of security set for the application
- ♦ enter and leave an application
- ♦ manipulate message buffers
- ♦ communicate with a service and receive replies in request/response mode
- ♦ modify the way a function performs by specifying various options

The chapter ends with information about how to compile client programs.

Examples Taken from the Sample Application

Many of the examples in this chapter are taken from `audit.c`, a client program that is part of the sample application.

Depending on command line options, `audit.c` retrieves either

- ♦ the total account or teller balance for all the branches of the bank, or
- ♦ the account or teller balance for a specified branch

The syntax of the command line is as follows:

```
audit {-a|-t} [bid]
```

`audit` is the name of the executable created when the `audit.c` program is compiled. The `-a` option requests that account balances be retrieved; the `-t` option specifies the teller balance. If no branch identifier, *`bid`*, is included on the command line, the default is to retrieve the total account or teller balance for all the branches of the bank. If the branch identifier is included, a balance of the type specified is retrieved for that branch only.

Preliminaries

Before a client program is ready to join the application, some preliminary processing may be called for to take advantage of BEA TUXEDO system capabilities.

Client Naming

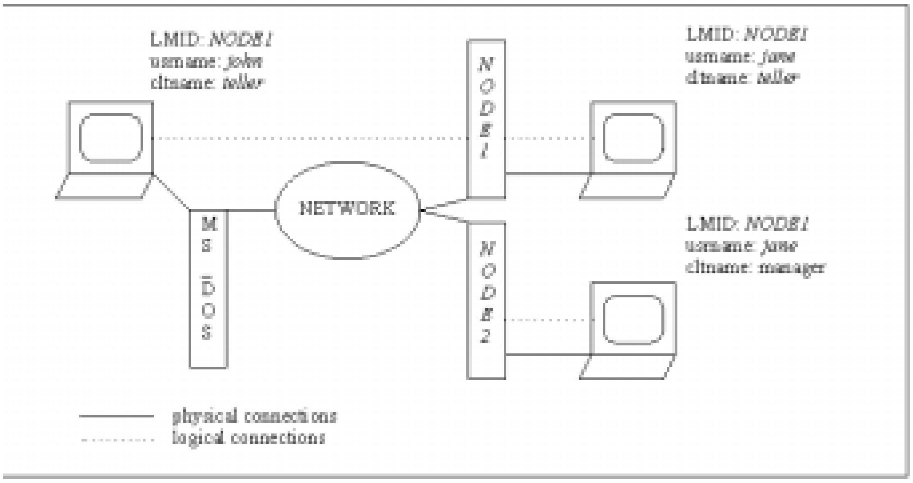
An application can associate both a `usrname` and a `cltname` with an execution of a client process. Values furnished for these names are combined by the BEA TUXEDO system with the logical machine identifier (`LMID`) of the machine where the process runs, in order to establish a unique identification for the process. It is left to the discretion of application developers and programmers to work out ways of acquiring the value for the fields. Once acquired they are passed to `tpinit()` in a `TPINIT` buffer. Some possible ways are shown in later examples.

Note: If the process is running outside the administrative domain of the application, that is, if it is running on a workstation connected to the administrative domain, the `LMID` used is the one for the machine used by the workstation client to access the application.

Once a client process is uniquely identified, client authentication can be implemented, out-of-band messages can be sent to a specific client or to groups of clients via `tpnotify`(3c) and `tpbroadcast`(3c), and detailed statistical information can be gathered via `tmadmin`(1).

Figure 2-1 shows an example of 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 2-1 Client Naming



Unsolicited Notification

Unsolicited notification refers to any communication with a client that is not an expected response to a service request (or an error code). The example that comes to mind is a broadcast message to announce that the world is coming to an end in five minutes. Within the client program there are three things you may want to do to handle such messages:

- set flags in the `TPINIT` buffer to select the method used to detect messages
- if you use the dip-in method, call `tpsetunsol()` to name your message handling function
- if you use the dip-in method, call `tpchkunsol()` to see if any unsolicited messages have been received

The flag values in the `TPINIT` buffer are described below in the section called “Joining the Application.” `tpsetunsol`(3c) and `tpchkunsol`(3c) are shown in examples later in this chapter and are described in the *BEA TUXEDO Reference Manual*.

Security and Client Authentication

The BEA TUXEDO system provides several levels of security:

- Operating System
- Application Password
- User Authentication
- Optional Access Control Lists
- Mandatory Access Control Lists
- Link-Level Encryption

Configuration of the security level is the responsibility of the system administrator and is discussed in the book *Administering the BEA TUXEDO System*. The following paragraphs explain the different levels and discuss what is needed when writing client programs with `SECURITY` set.

Operating System

For platforms that have underlying security mechanisms, this is the first line of defense. The security level is configured to “`NONE`.” This implies, not that there is no security, but that there are no additional mechanisms (for example, a BEA TUXEDO system password) beyond what the platform provides. The BEA TUXEDO system has the notion of an application administrator who configures the application, starts up the application (servers run with the permissions of this administrator), and monitors the running application, making dynamic changes as necessary. Note that this implies that server programs are “trusted” since they run with the administrator’s permissions. This is supported using the underlying operating system login mechanism and read/write permissions on files, directories, and system resources.

Client programs are run directly by the users with their own permissions. However, they normally have access to the administrative configuration file and the interprocess communication mechanisms, such as the Bulletin Board in shared memory, as part of normal processing. This is true whether or not additional BEA TUXEDO system security is configured. For some applications running on platforms supporting it, a more secure approach is to have the files and IPC mechanisms accessible only to the application administrator and to have “trusted” client programs run with the permissions of the administrator (using a `setuid` mechanism). Combining this with BEA

TUXEDO system security will allow the application to "know" who the user is that is making the request. For the most secure environment, only workstation clients should be allowed to access the application; client programs should not be allowed to run on the machines where application server and administrative programs run. BEA TUXEDO system security mechanisms can be used in addition to operating system security to prevent unauthorized access. The additional security can be used to avoid simple violations like someone accessing an unattended terminal. Or it can protect the boundaries of the administrative domain from inter-domain or workstation client access over the network by unauthorized users.

Application Password

This security level requires that every client provide an application password as part of joining the application. The security level is configured to "`APP_PW`." The administrator must provide an application password when this level is configured and this password can also be changed administratively. It is the responsibility of the administrator to inform users of the application what the password is. If this level of security is used, BEA TUXEDO system-supplied client programs, `ud`(1) for example, prompt for the application password. Application-written client programs must include code to obtain the password from a user. The password should not be echoed to the user's terminal. The password is placed in clear text in the `TPINIT` buffer and evaluated when the client calls `tpinit()` to join the application. Code for handling a password is shown in examples later in this chapter.

User Authentication

The third level of BEA TUXEDO system security is based on authenticating each individual user in addition to providing the application password. The security level is set to "`USER_AUTH`." This level involves passing user-specific data to an authentication service. Often, the data is a per-user password. The data is automatically encrypted when passed over the network from workstation clients. The default authentication service, "`AUTHSVC`," is provided by a BEA TUXEDO system-supplied server, `AUTHSVR`. The operation of `AUTHSVR` is described in Chapter 3, "Writing Service Routines." This server can be replaced with an application authentication server with logic specific to the application. (For example, it might access the widely-used Kerberos mechanism for authentication.) With this level of security, authentication but not authorization is provided. That is, the user is checked when joining the application but then is free to execute any services, post events, and access application queues. It is possible for the servers to do application-specific authorization within the logic of the service routines, but there are no hooks for authorization checking for access to events or application queues. The alternative is to use the built-in access control checking.

Optional Access Control Lists

With the use of access control lists (ACLs), the user is not only authenticated when joining the application, but permissions are automatically checked when accessing application entities such as services. ACL security also includes the user-authentication security equivalent to “`USER_AUTH`.” There are two levels of ACL checking. The first ACL security level is simply called “`ACL`.” If “`ACL`” is configured, the Access Control Lists are checked whenever a user attempts to access a service name, queue name, or event name within the application. If there is no ACL associated with the name, the assumption is that permission is granted. This is why this level is considered “optional” ACLs. It allows the administrator to configure access for those resources that need more security, but ACLs need not be configured for services, queues, or events that are accessible to everyone. Some applications may find it necessary to use both system level and application authorization. An ACL can be used to control who can get to a service, and application logic can control data-dependent access (for example, who can handle transactions for more than a million dollars).

Mandatory Access Control Lists

The second ACL security level is “`MANDATORY_ACL`.” This level is similar to “`ACL`,” but an access control list must be configured for every object for which users are to have access. If “`MANDATORY_ACL`” is specified and there is no ACL for the name, permission is denied.

Link-Level Encryption

Users of the BEA TUXEDO system Security Add-On Package (Domestic or International) can establish data privacy for messages moving over the network links that connect machines of BEA TUXEDO system applications (or domains).

Writing Client Programs with SECURITY Set

Two things need to be done for clients that are running in an application with `SECURITY` set: a) getting the security data needed for the specific user, and b) passing this information to the BEA TUXEDO system when joining the application.

Getting the Security Data

The function `tpchkauth`(3c) is provided so a check on the level of security can be done before calling `tpinit()`. This is necessary so that the program can prompt for an application password and possibly user authentication data needed for the `tpinit()` call. `tpchkauth()` is called without arguments and returns one of the following values.

`TPNOAUTH`
: Nothing is required beyond the normal operating system login and file permission security. This is returned for security level "`NONE`."

`TPSYSAUTH`
: An application password is required. The client program should prompt the user to provide the password, and should place it in the *`passwd`* field of the `TPINIT` buffer (described below). This is required for security level "`APP_PW`."

`TPAPPAUTH`
: The application password is required and in addition the client is expected to provide a value to be passed to the authentication service in the *`data`* field of the `TPINIT` buffer. This is returned for security level "`USER_AUTH`," "`ACL`," or "`MANDATORY_ACL`."

Joining the Application

In an application configured with `SECURITY`, it is necessary to pass the security information to the BEA TUXEDO system via a `TPINIT` buffer. The `TPINIT` buffer is a special typed buffer used by a client program to pass client identification and authentication information to the system as the client attempts to join the application. It is defined in the `atmi.h` header file and contains the following fields.

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

The usrname, cltname, and grpname Members of TPINIT

usrname, *cltname*, and *grpname* are all NULL-terminated strings of up to MAXTIDENT characters. MAXTIDENT is defined as 30. *usrname* is a name representing the caller; you might elect to use the operating system user name. *cltname* is a client name whose semantics are application defined. You might use this field to indicate the role of the user when executing the client program. It is also used for selection of specific clients when sending broadcast messages. *grpname* allows a client to be associated with a resource manager group that is defined in the configuration file. This means that a client can access an XA-compliant resource manager as part of a global transaction. If *grpname* is passed as a 0-length string, the client is not associated with a resource manager group and is in the default client group.

The *usrname* and *cltname* fields are associated with the client process when `tpinit()` is called and are used for authentication, broadcast notification, and the retrieval of administrative statistics.

The passwd Member of TPINIT

passwd is a NULL-terminated string of up to 8 characters. It is an application password in unencrypted format that is used by `tpinit()` for validation against the configured application password.

The flags Member of TPINIT

The setting of *flags* is used to indicate the notification mechanism and system access mode to be used. Selections override values specified in the configuration file (with some exceptions explained below). Possible values for *flags* are:

TPU_DIP

Select unsolicited notification by dip-in. This is the default method if nothing is specified in the configuration file. It has the advantage of giving the receiving program more control over when unsolicited messages are handled. The system will detect unsolicited messages for your client process only while you are within ATMI calls. You may want to check for unsolicited messages as part of your regular checking routine following returns from ATMI calls. If you specify this flag (or accept it as the default method), you should include a call to `tpsetunsol()` early in your program. Until the handler for unsolicited messages is known, no messages can be delivered.

`TPU_SIG`
: Select unsolicited notification by signals. This method has the advantage of immediate notification, but has the limitations that you must have the same `uid` as the sending process, and is not available on all platforms (specifically, it is not available with the MS-DOS instantiation of the workstation). If you specify this option but do not qualify for it, the system resets your choice to `TPU_DIP` and calls `userlog()` to note the event.

`TPU_IGN`
: Ignore unsolicited notification.

`TPSA_FASTPATH`
: Specifies a) that ATMI calls within application code can access BEA TUXEDO system internal tables via shared memory, and b) that the shared memory is not protected against access by application code outside of BEA TUXEDO system libraries. Overrides the value in `UBBCONFIG`, except when `NO_OVERRIDE` is specified. This is the default if `SYSTEM_ACCESS` mode is unspecified.

`TPSA_PROTECTED`
: Specifies that ATMI calls within application code can access BEA TUXEDO system internal tables via shared memory but the shared memory is protected against access by application code outside of BEA TUXEDO system libraries. Overrides the value in `UBBCONFIG`, except when `NO_OVERRIDE` is specified.

The datalen and data Members of TPINIT

User-specific data is passed by using the *`datalen`* and *`data`* fields when security is set to “`USER_AUTH`,” “`ACL`,” or “`MANDATORY_ACL`.” *`datalen`* is the length of the user-specific *`data`* that follows. The buffer type switch entry for the `TPINIT` typed buffer sets *`datalen`* based on the total size passed in for the typed buffer (the application data size is the total size less the size of the `TPINIT` structure itself plus the size of the data placeholder as defined in the structure). There is a macro, `TPINITNEED`, provided in `atmi.h`, that calculates the size needed when you call it with the number of bytes of application *`data`* you expect to pass.

`data` is a placeholder for variable length data that is forwarded to an authentication service. *`data`* is always the last element of the structure.

Allocating the TPINIT Buffer

The client program must call `tpalloc()` to allocate the `TPINIT` buffer. You can use the functions described for message typed buffers in the "Buffer Management" section later in this chapter. A sample is shown in Listing 2-1. The intent in this example is to prepare to pass 8 bytes of application-specific *`data`* to `tpinit()`.

Listing 2-1 Allocating a TPINIT Typed Buffer

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

The Application Key

An application key is associated with each client program when it joins the application. You can think of this 32-bit value as the security credential for the client; it identifies the client for security purposes. This value cannot be reset by the client (other than by terminating its association and joining the application as a different user), and cannot be forged. The value is provided to every service invocation as part of the `TPSVCINFO` structure in the *`appkey`* field (see `tpservice`(3c)).

The following list indicates how the application key will be set for various security levels and clients.

- Messages from native BEA TUXEDO system-provided clients that must be run by the administrator, such as `tmadmin`, `dmadmin`, and `tmshutdown`, will have the application key of the administrator (the value is 0x80000000). This is independent of the security level.
- There are three classes of client users in a system with security set to "`NONE`" or "`APP_PW`":

    - Messages from native clients that call `tpinit`(3c) with a client name of `tpsysadm` and are run by the administrator will have the application key of the administrator.
    - Messages from native clients that call `tpinit`(3c) with a client name of `tpsysopr` and are run by the administrator will have the application key of the system operator (the value is 0xC0000000).
    - Other client programs will always have an application key of -1 (there is no distinction between users).
- Multiple users exist in the case where per-user authentication is done (security set to “`USER_AUTH`,” “`ACL`,” or “`MANDATORY_ACL`”):
    - Messages from native clients that call `tpinit`(3c) with a client name of `tpsysadm` and are run by the administrator will have the application key of the administrator, and will not be authenticated.
    - Messages from authenticated clients that call `tpinit`(3c) with a client name of `tpsysadm` will have the application key of the administrator.
    - Messages from authenticated clients that call `tpinit`(3c) with a client name of `tpsysopr` will have the application key of the system operator.
    - For other clients, the key depends on the security level. For “`USER_AUTH`” security, the default `AUTHSVR` returns the configured user identifier. For “`ACL`” or “`MANDATORY_ACL`” security, the `AUTHSVR` returns an application key with the user identifier in the lower 17 bits and the group identifier in the next 14 bits.
- Any message that originates from `tpsvrinit`(3c) or `tpsvrdone`(3c) will have the application key of the administrator. Messages that pass through a server but originate at a client will have the application key of the client.

Joining and Leaving an Application

The two routines discussed in this section allow a client process to join and leave a BEA TUXEDO system application. The syntax of these functions is as follows.

```
int
tpinit(tpinfo)   /* Join a BEA TUXEDO Application */
TPINIT *tpinfo;
```

and

```
int
tpterm()   /* Leave a BEA TUXEDO Application */
```

Before a client can make any service request, it must join the application. If a service request (or any ATMI function) is called before invoking `tpinit()`, then it is invoked automatically with a `NULL` parameter. This implies that the `TPINIT` features mentioned earlier in this chapter cannot be used; the default values are used for client naming, unsolicited notification type, and system access mode, the client cannot be associated with a resource manager group, and an application password cannot be specified. To use these features, the application must explicitly invoke the `tpinit()` function. Once invoked (either implicitly or explicitly), the calling process may initiate requests and receive replies. `tpterm()` removes the process from the application. When `tpterm()` returns successfully, the process must again join the application before communicating with a BEA TUXEDO system server process. A typical client process might begin and end as shown in Listing 2-2.

Listing 2-2 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;
     }
}
```

The argument to `tpinit()` is a pointer to a structure `TPINIT`, that is `typedef`'d in the `atmi.h` header file. If you use a buffer, a `TPINIT` typed buffer must be allocated via `tpalloc()` before calling `tpinit()`.

`tpterm()` does not take an argument. Both functions return an integer. On error, the value of the returned integer is `-1` and the external global variable, `tperrno`, is set to a value that indicates the nature of the error. `tperrno` is defined in the `atmi.h` header file and documented on the `tperrno`(5) reference page. The convention is to 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 Chapter 7, “Error Management.” The complete list of error codes that can be returned for each of the ATMI functions can also be found on the reference pages that describe the function and the `intro`(3c) reference page in the *BEA TUXEDO Reference Manual*.

An example of `tpinit()` and `tpterm()` is shown in Listing 2-3. It is taken from the `audit.c` client program in the banking application.

Listing 2-3 Joining and Leaving the Application

```
#include <stdio.h>          /* UNIX */
#include <string.h>         /* UNIX */
#include <fml.h>            /* BEA TUXEDO */
#include <atmi.h>           /* BEA TUXEDO */
#include <Uunix.h>          /* BEA TUXEDO */
#include <userlog.h>        /* BEA TUXEDO */
#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 an error is encountered (that is, if the return code is `-1`), a message is written to the central event log via a call to `userlog()`. The `userlog()` function takes arguments similar to `printf()` and is documented in the `userlog`(3c) reference page in the *BEA TUXEDO Reference Manual*. As explained, the client process is invoked by entering its name at the prompt with the mandatory `-a` or `-t` option. Its name is captured in `argv[0]` and is placed in the global variable `proc_name` which gets written to the event log as part of the message. A similar explanation applies to the call to `tpterm()`.

Buffer Management

Before messages can be sent between processes, a buffer must be allocated for the message data. The following sections describe the buffer types supported by the BEA TUXEDO system and how buffers are allocated, changed in size, tested for type, and freed using ATMI functions.

Typed Buffers for Messages

The BEA TUXEDO system is delivered with nine message buffer types defined:

```
STRING CARRAY FML VIEW X_COMMON X_C_TYPE X_OCTET FML32 VIEW32
```

The buffer types are defined in `tmtypesw.c` (which can be found in `$TUXDIR/lib/tmtypesw.c`, with documentation in `tuxtypes`(5)). When the BEA TUXEDO system software is built, `tmtypesw.o` is archived in the BEA TUXEDO system libraries that are automatically linked in when the `buildclient` and `buildserver` commands are invoked, so the nine defined types are available to your application programs.

The `tmtypesw.c` file can be edited to add or remove buffer types. Information about how to do this can be found in the book *Administering the BEA TUXEDO System*. Only buffer types defined in `tmtypesw.c` can be known to your client or server programs. The `ubbconfig`(5) `BUFTYPE` parameter can be used to specify the types and subtypes a given service can know about.

Buffer Types: STRING

The `STRING` buffer type is what is conventionally understood as a string in the C language. It is a character array terminated by the null character. Data dependent routing is not provided for this buffer type. If routing functions are desired, they must be written as part of the application. Encoding and decoding is provided for this buffer type.

Buffer Types: CARRAY

The CARRAY buffer type (and equivalently X_OCTET) is an array of characters, any of which can be the null character. The application defines the semantics of the array; it is not interpreted by the BEA TUXEDO system. Data dependent routing is not provided for this buffer type. If routing functions are desired, they must be written as part of the application. No encoding or decoding is provided for a CARRAY buffer when crossing machine boundaries since the bytes are not interpreted by the system.

Buffer Types: FML and FML32

FML buffers offer the advantages of data independence and flexibility; fields may be present or absent, or may have multiple occurrences. Also, FML buffers interface well with both the BEA TUXEDO system DBMS and the DES. The BEA TUXEDO system DBMS supports fielded records in database files, and the mio client process of the BEA TUXEDO system DES uses fielded buffers for input and output data. In addition, this data type provides the functionality of data dependent routing. Automatic encoding and decoding is done if the buffer is passed between machines of different types.

FML functions are used to manipulate FML typed buffers. These functions include some that convert fielded buffers to C structures and back again, thus providing both the performance gains of C structures for lengthy field manipulations and the flexibility of fielded buffers. A C structure that is derived from a fielded buffer is called a VIEW.

FML32 is similar to FML but allows for larger string and character fields, more fields, and larger overall buffers. The FML32 buffer type uses environment variables suffixed with “32”, for example, FIELDTBLS32 and FLDTBLDIR32. FML32 functions (like their FML counterparts but with a “32” suffix) are used to manipulate these buffers. Functions are also provided to convert between 16-bit and 32-bit FML buffers (assuming that the limits are not exceeded), and functions are available to convert between FML32 and VIEW32 buffers.

Buffer Types: VIEW, VIEW32, X_C_TYPE, and X_COMMON

Buffers of the VIEW type (and equivalently X_C_TYPE and X_COMMON) are C structures. The C structure is passed between processes in a VIEW typed buffer of a specific subtype. It can be one derived from a fielded buffer or one defined independently of a fielded buffer. The ATMI buffer management primitives for allocating, resizing, and freeing a VIEW buffer are the same for both types, but there are differences in the way the two types of VIEWS themselves are defined and in how they are handled within your programs. These differences were described in the section titled, “The BEA TUXEDO System Development Environment,” in Chapter 1. Both types of VIEW buffer support data dependent routing and automatic encoding and decoding when the buffer is passed between unlike machines.

A comparison of how to create and use the two VIEW types is summarized in Table 2-1.

Table 2-1 Comparison of Two VIEW Types

	FML-dependent VIEW	**FML-independent VIEW**
Creating	create the view description file with FML information in it	create the view description file without FML information in it
	use the `viewc` compiler without the `-n` option to compile the description file	use the `viewc` compiler with the `-n` option to compile the description file
Using	set and export FIELDTBLS, FLDTBLDIR, VIEWFILES, VIEWDIR in the ENVFILE for the machine the client process is running on	set and export VIEWFILES and VIEWDIR in the ENVFILE for the machine the client process is running on
	`#include fml.h`, the header file created from the field table file, and the header file created from the view compiler in the programs that define FML and VIEW buffers	`#include` the header file created from the view compiler in the programs that define VIEW buffers

Buffers of type `X_COMMON` should contain only short, long, and character fields, which are common to both the COBOL and C languages.

The `VIEW32` type is similar to the `VIEW` type but supports larger character fields and bigger records. It is also used for conversion to/from FML32 records. The `VIEW32` buffer type uses environment variables suffixed with “`32`”, for example, `FIELDTBLS32`, `FLDTBLDIR32`, `VIEWFILES32`, and `VIEWDIR32`.

Buffer Types: Summary

Although system configuration and defining buffer types are application design issues rather than programming issues, the above discussion has been included to explain how processes know about the various buffer types so you can allocate buffers correctly for the communication calls between processes.

ATMI Buffer Primitives

It is important for the BEA TUXEDO system programmer to know what buffer types are required and expected by the application. The ATMI functions that allocate, resize, and free the buffers take the buffer *`type`* and *`subtype`* as arguments. For the types provided by BEA TUXEDO, the *`subtype`* argument has meaning only when *`type`* is `VIEW`, `VIEW32`, `X_C_TYPE`, or `X_COMMON`. In this case, the *`subtype`* is the name of the specific C structure defined as a `VIEW`. In the other buffer types, the *`subtype`* argument is `NULL`.

Allocating a Typed Buffer

Initially, a client process does not have any buffers. Before a message can be sent, the client process must allocate a buffer of a supported type to carry the message. A typed buffer is allocated by using the `tpalloc()` function. The syntax of this function is:

```
char*
tpalloc(type, subtype, size) /* Allocate a new data buffer */
char *type, *subtype;
long size;
```

The three arguments the function takes are *type*, *subtype*, and *size*. The value of *type* must be a type known to BEA TUXEDO.

The `VIEW`, `VIEW32`, `X_C_TYPE`, and `X_COMMON` buffers require the *subtype* argument. (See Listing 2-4.) In the cases where a subtype is not relevant, assign the `NULL` value to this argument. This is illustrated in Listing 2-5, Listing 2-6, and Listing 2-7.

Listing 2-4 Allocating a VIEW Buffer

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

Listing 2-5 shows the allocation of an FML typed buffer.

Listing 2-5 Allocating an FML Buffer

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

The *size* argument can be set to zero for all the BEA TUXEDO system-supplied types except for CARRAY. If *size* is not specified (that is, if it is set to zero), BEA TUXEDO uses a default size that is defined for each buffer type. If the *size* argument is specified, the size of the buffer will be the larger of the specified size or the default size. The default size for STRING is 512 bytes, and it is 1024 bytes for FML, FML32, VIEW, X_C_TYPE, X_COMMON, and VIEW32.

For a CARRAY a *size* greater than zero must be specified (see example in Listing 2-6); the default size is 0 and this causes tpalloc() to return a NULL pointer and set tperrno to TPEINVAL.

Note that in cases of error, tpalloc() always returns the NULL pointer. Other causes for error include failure to specify a value for type (or subtype in the case of VIEW), specifying a type that is not known to the system, and failing to join the application before attempting allocation. Refer to the tpalloc(3c) reference page for the complete list of error codes and their explanation.

Upon success, tpalloc() returns a pointer of type char. For types other than STRING and CARRAY, you should cast it to the proper C structure pointer or to an FML pointer. (See Listing 2-4 and Listing 2-5.)

Listing 2-4 shows the allocation of a VIEW typed buffer. It is taken from the audit.c client program in the banking application. The aud structure is the VIEW typed buffer that is defined in Chapter 1, “Introduction and Overview.”

tpalloc Examples

Listing 2-6 shows the allocation of a `CARRAY` typed buffer. The value of *`casize`* must not be zero.

Listing 2-6 Allocating a CARRAY Buffer

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

Listing 2-7 shows the allocation of a `STRING` typed buffer. In the example, the default size defined by the system is used as the value for the `size` argument to `tpalloc()`.

Listing 2-7 Allocating a STRING Buffer

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

What About FML Buffer Management Functions?

If you've been looking at the *BEA TUXEDO FML Programmer's Guide*, especially the section in Chapter 5 called "Buffer Allocation and Initialization," you probably realize it is also possible to manage `FML` buffers by the routines described there. However, if the buffers are to be used in the communication calls in the ATMI interface, they must be managed by the routines described on the `tpalloc`(3c) reference page. Specifically, this means that `Falloc()`, `Frealloc()`, and `Ffree()` should be replaced by `tpalloc()`, `tprealloc()`, and `tpfree()`. `Finit()` is not needed because `tpalloc()` automatically initializes the buffer. Also, since the `FML` typed buffer is given a default size by the system, `Fneeded()` should be used only when you wish to assign the buffer a specific size that is larger than the default size as is shown above in Listing 2-5. The `f` and `v` arguments to `Fneeded` are integer values that represent the number of fields and the space for field values in bytes required for the fielded buffer. All the other `FML` functions described in Chapter 5 of the *BEA TUXEDO FML Programmer's Guide* can be used with all `FML` typed buffers regardless of how the buffers were allocated.

Putting Data in the Buffer

Once the buffer has been allocated, data can be put in it. The aud VIEW typed buffer has three members (fields). They are b_id, the branch identifier taken from the command line (if given); 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 query a specific branch balance, the b_id member is set to the branch identifier to be queried, and the balance and ermsg members are set to zero and the null string, respectively. This is illustrated in Listing 2-8.

Listing 2-8 Placing 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 illustrated in Listing 2-9.

Listing 2-9 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, "");
...
```

An example of code that puts data into a STRING buffer is part of Listing 2-10.

Resizing a Typed Buffer

It is possible to resize the buffer that is initially allocated by `tpalloc()` if you want to use the same buffer to input and send messages of different sizes. The function you would use in this case is `tprealloc()`. The syntax of the function is as follows.

```
char*
tprealloc(ptr, size) /* Change a data buffer's size */
char *ptr;
long size;
```

For example, if a buffer has been allocated as type `STRING`, it is possible to reallocate a buffer of a different size but of the same type, as illustrated in Listing 2-10.

Listing 2-10 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
  . . .
}
```

As illustrated, `tprealloc()` takes two parameters, a pointer to the buffer that is to be resized and a long integer that tells the function the new size of the buffer. The pointer passed to `tprealloc()` must have originally been allocated by a call to `tpalloc()`; otherwise the call will fail and `tperrno` will be set to `TPEINVAL` to signify that invalid arguments have been passed to the function. The pointer returned by `tprealloc()` will point 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. The contents of the buffer, up to the smaller of the two sizes, remains unchanged. When `tprealloc()` is called to make a buffer larger, new space is available beyond the existing contents. When `tprealloc()` is called to make a buffer smaller, the buffer does not actually become smaller; space beyond the specified size is unusable. If you really want to free up the unused space, you must copy the data into a buffer of the appropriate size and free the larger buffer.

`tprealloc()` returns the NULL pointer on error and sets `tperrno` as indicated on the `tpalloc`(3c) reference page. When `tprealloc()` returns the NULL pointer, the contents of the buffer passed to it may have been altered and may be no longer valid.

Listing 2-11 shows an expanded version of the example in Listing 2-10 that could be used to check for all error codes `tprealloc()` can return.

Listing 2-11 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);
}
```

Checking for Buffer Type

The `tptypes()` function takes a pointer to a data buffer as its first argument and returns the type and subtype (if there is one) for that buffer in its other two arguments. It returns a long integer which, on success, is the length of the buffer. The syntax of this function is:

```
long
tptypes(ptr, type, subtype)  /* Determine a data */
char *ptr, *type, *subtype;  /* buffer's type and subtype */
```

The pointer you supply to this function must point to a buffer originally allocated or reallocated by `tpalloc()` or `tprealloc()`, otherwise it will fail complaining of invalid arguments. If the type is not `VIEW`, the subtype parameter will point to a character array containing the null string upon return from the function call. All three of the parameters of `tptypes()` are pointers to character. The first parameter is the pointer to the typed buffer and must be non-null. Be sure to cast it as a pointer to a character before passing it to this function since it is expecting a character pointer. The second and third parameters return the type and subtype of the buffer pointed to by the first parameter. The second parameter must be a character array of at least `TM_TYPELEN` characters, and the third parameter must be a character array of at least `TM_STYPELEN` characters. On success, the size of the buffer is returned. On error, `tptypes()` returns `-1` and sets `tperrno` to an error code that signifies the problem. All the possible codes are listed on the `intro`(3c) and `tpalloc`(3c) reference pages.

In addition to the fragment shown below (in Listing 2-12), an example of `tptypes()` can be found in Listing 3-2 in Chapter 3, "Writing Service Routines." It demonstrates a service routine checking the type of buffer received.

The size value returned by `tptypes()` can be used to determine if the default buffer size is large enough to hold your data.

Listing 2-12 Getting Buffer Size

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

Freeing a Typed Buffer

To free a buffer allocated by `tpalloc()` or reallocated by `tprealloc()`, use the `tpfree()` function. The syntax of this function is:

```
void
tpfree(ptr) /* Free a data buffer */
char *ptr;
```

The argument to this function is a pointer previously returned by the `tpalloc()` or `tprealloc()` function. If `tpfree()` is given a pointer that does not point to a buffer obtained from `tpalloc()` or `tprealloc()`, it returns without freeing anything, and it does not return an error condition. `tpfree()` expects a character pointer as its only parameter. Listing 2-13 shows an example of its use.

Listing 2-13 Freeing a Buffer

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

Service Calls

Once a client process has joined the application, allocated a buffer, and placed the input data request in it, it can then send the request message to a service subroutine for processing and receive a reply message. The next sections discuss the ATMI functions that allow processes that are acting as clients to send message requests to services and receive replies either synchronously or asynchronously.

The `tpcall()` function sends a request to a service subroutine and synchronously waits for its reply.

The `tpacall()` function sends a request to a service and immediately returns. The reply to the service call is asynchronously received by calling the `tpgetrply()` function.

Sending Synchronous Messages: tpcall()

`tpcall()` is used to send synchronous messages. The syntax of this function is:

```
int
tpcall(svc, idata, ilen, odata, olen, flags) /* Send service request */
char *svc, *idata;                           /* and await its reply */
long ilen;
char **odata;
long *olen, flags;
```

`tpcall()` sends a request to the service that is specified in its first parameter, *svc*. The service named in *svc* must be one offered in your application. `tpcall()` waits for the expected reply. It is logically the same as calling the `tpacall()` function immediately followed by `tpgetrply()`. The request carries the priority that is set by the system for the service specified in *`svc`* unless a different priority has been explicitly set by a call to `tpsprio()`.

The second parameter of the function, *`idata`*, is a 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 subtype) of *`idata`* must match the type (and subtype) expected by the service routine. If the types do not match, the system sets `tperrno` to `TPEITYPE` and the function call fails.

The third parameter, *ilen*, specifies the length of the request data in the buffer pointed to 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, *ilen* is ignored and can be set to zero. If the request requires no data, set *idata* to the NULL pointer. This causes the *ilen* parameter to be ignored. If no data is being sent with the request, there is no need to allocate a buffer for *idata*.

The next two parameters are the address of a pointer to the output buffer, *odata*, and a pointer to the length of the reply data, *olen*. The output buffer, ***odata*, must have been allocated by a previous call to tpalloc(). This buffer is used to receive the reply message. If the reply message sent back contains no data portion, upon successful return from tpcall(), **olen* will be set to zero, and the pointer and the contents of the output buffer will remain unchanged. It is an error for either ***odata* or **olen* to point to NULL.

The same buffer can be used for both the request and reply message. If this is the case, then *odata* must be set to the *address* of the pointer returned from allocating the input buffer.

Listing 2-14 shows the client program, audit.c, making a synchronous call using the same buffer for both the request and reply message. Using the same buffer is appropriate in this particular case, since the *audv message buffer has been set up to accommodate both request and reply information in the same buffer. The b_id field is queried by the service but not overwritten and the *bal* and *ermsg* fields have been initialized to zero and the null string, respectively, in anticipation of the values to be returned by the service. The svc_name and hdr_type variables represent the service name and the balance type (account or teller) requested.

Listing 2-14 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);
. . .
```

Note: For an example in which different buffers are used for input and output, see Listing 3-2 in Chapter 3, “Writing Service Routines.”

Buffers used for receiving messages can grow upon receipt of the message if the message proves to be too large for the allocated buffer. BEA TUXEDO guarantees that a received message will fit into the buffer by growing the buffer automatically. However, it is necessary for the programmer to test for size changes of reply buffers in order to determine their actual sizes. The new size is accessible by the address returned in the *olen* parameter. To determine if a reply buffer 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 has not changed in size. 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 increase in buffer size. This scenario does not apply to request buffers since there is no possibility that the request data will grow upon placing it in the buffer. Note that if you use the same buffer for the request and reply message, and the pointer to the reply buffer changed because the buffer grew, then the input buffer pointer no longer references a valid address.

Listing 2-15 offers a generic example of an application testing for a change in buffer size after a call to `tpcall()`. The logic exercised in this particular example is that the input and output buffers must remain equal in size.

Listing 2-15 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
}
```

Values for the flags Argument: tpcall()

The last argument that `tpcall()` takes is *`flags`*. The values given to the *`flags`* argument can change the operation of the communication call in some way, allowing additional flexibility to the application. If *`flags`* is set to `0`, the communication is conducted in the default manner.

`TPNOTRAN`

If the client process is in transaction mode when it calls `tpcall()`, and *`flags`* is set to `TPNOTRAN`, the service that is invoked by the call will not be part of the transaction; that is, the operations that the service performs are not part of the caller's transaction. There's more on this subject in Chapter 5, "Global Transactions in BEA TUXEDO System."

`TPNOCHANGE`

By using this value, the calling program is indicating that it wants the message returned in the same type of buffer that was originally allocated as the output buffer. In other words, when this flag is set, the type of buffer returned to the caller must be the same as the one pointed to by **`odata`*. This is known as strong type checking. The default is to allow a buffer type to be different than the original one so long as the caller recognizes the type. In this case, the buffer type for **`odata`* changes to the received buffer type. This is known as weak type checking. A call to `tptypes()` informs the recipient of the new buffer type.

TPNOBLOCK

TPNOBLOCK concerns the action a function call takes if a blocking condition exists. Callers of the communication routines typically block when waiting for a reply to arrive although they may also block when trying to send a request if all server queues or internal buffers are full. A default blocking time-out period is defined for the application in the configuration file. It specifies the amount of time a caller should wait for a blocking condition to subside when one exists. If the condition persists beyond this limit, the function call fails and tperrno is set to TPETIME. When the value of *flags* is set to TPNOBLOCK, if a blocking condition exists, the call fails immediately and the request message is not sent. In this case, tperrno is set to TPEBLOCK. Note that tpcall() is a dual function in that it both sends a request and receives a reply. When TPNOBLOCK is set, it affects only the send part of the function; if all the server queues are filled or the internal buffers into which the message buffers are copied are full, the call will not block but immediately return. However, if it must wait for the reply (which is usually the case), this flag setting does not immunize the call from blocking while it waits.

TPNOTIME

By setting *flags* to TPNOTIME, you are telling the system to ignore the blocking time-out limit because the caller is willing to wait indefinitely for the blocking condition to subside. However, if the caller is in transaction mode, this flag has no effect; it is subject to the transaction time-out limit. The timing out of transactions is discussed in Chapter 5, "Global Transactions in BEA TUXEDO System."

TPSIGRSTRT

Another valid value for the *flags* argument is TPSIGRSTRT, which concerns the action to take if there is a signal interrupt. When *flags* is set to this value, the call is automatically made again. As a result, if a signal interrupts the underlying system call, the function call is reissued. When *flags* is not set to this value and there is a signal interrupt, the function call fails and tperrno returns TPGOTSIG.

Flag values can be or'd together.

tpcall() returns an integer. On failure, the value of this integer is -1 and the value of tperrno is set to an appropriate value reflecting the type of error that occurred. Some of the causes for error have already been discussed, while others have transaction implications and will be introduced in Chapter 5, "Global Transactions in BEA TUXEDO System." In general, communication calls may fail for a variety of errors. Many of the errors returned on communication calls can be fixed on an application level. They include application defined errors (TPESVCFAIL), errors in processing return arguments (TPESVCERR), typed buffer errors (TPEITYPE, TPEOTYPE), time-out (TPETIME), and protocol errors (TPEPROTO), among others. They are all discussed in Chapter 7, “Error Management,” and are listed on the intro(3c) and tpcall(3c) reference pages. The communication of these failures will also be explained in the discussion of the tpreturn() function in Chapter 3, “Writing Service Routines.”

Examples of the Use of flags Arguments

The next three examples show tpcall() using the communication flags in various scenarios.

Listing 2-16 is based on the TRANSFER service, which is part of the XFER server process of bankapp. The TRANSFER service assumes the role of a client when it calls on the services of WITHDRAWAL and DEPOSIT. In the example, we have set the communication flag to TPSIGRSTRT in these service calls to give the transaction a better chance of committing.

Listing 2-16 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);
}
```

Listing 2-17 illustrates a communication call that suppresses transaction mode. It is being made to a service that is not affiliated with a resource manager and it would be an error to allow it to participate in the transaction. Specifically in this example, an accounts receivable report, accrcv is to be printed against a database named accounts. The service routine `REPORT` interprets the parameters and sends the byte stream for the completed report as a reply. The client, shown here, uses `tpcall()` to send the byte stream to a service called `PRINTER` that prints out the byte stream to the appropriate printer for this client. It receives a reply from the `PRINTER` service naming the printer that was chosen to print the report to make it convenient for the user to pick up the hard copy. Listing 2-19 shows a similar example using an asynchronous message call.

Listing 2-17 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
}
```

In Listing 2-17, where *error routine* has been indicated, it should include printing an error message, aborting the transaction, freeing allocated buffers, leaving the application, and exiting the program.

Listing 2-18 illustrates the use of the `TPNOCHANGE` communication flag to enforce strong buffer type checking. This example refers to the same `REPORT` service that is used above in Listing 2-17. In this one, the reply is received in a `VIEW` typed buffer called rview1 and the elements are printed in `printf()` statements. The strong type check flag, `TPNOCHANGE`, is used to force 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 in processing the request that could result in 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, someone could have changed the `REPORT` service to standardize all replies in some other `VIEW` format without modifying the client process to reflect the change.

Listing 2-18 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: tpacall()

This section discusses the sending of asynchronous messages where the sender of the request does not wait for the reply. The first half of this communication is performed by `tpacall()`. The syntax of this function is:

```
int
tpacall(svc, data, len, flags) /* Send service request */
char *svc, *data;
long len, flags;
```

The `tpacall()` function sends a request message to the service named in the *`svc`* parameter and immediately returns from the call. The next three parameters, *`data`*, *`len`*, and *`flags`*, have the same semantics as *`idata`*, *`ilen`*, and *`flags`* of the `tpcall()` function. Upon successful completion of the call, `tpacall()` returns an integer that serves as a descriptor used to get the correct reply for the sent request. While `tpacall()` is in transaction mode (topic of Chapter 5, “Global Transactions in BEA TUXEDO System,”), there may be no outstanding replies when the transaction commits; that is, within a given transaction, for each request sent expecting a reply, a corresponding reply must eventually be received.

Values for the flags Argument: tpacall()

The communication flags that `tpacall()` takes as values for the *`flags`* argument pertain to the send part of the communication. As a result, the flag value `TPNOCHANGE` is removed since it concerns the output buffer which is not present in this call, and the value `TPNOREPLY` is added since the receive part is not implicit to this communication call. When `tpcall()` is used, the fact that a reply is expected is implicit. `tpacall()` represents only the sending part of `tpcall()`, and it is possible to indicate whether a reply is expected or not.

`TPNOREPLY`

If the value `TPNOREPLY` is assigned to the *`flags`* parameter, it signals to `tpacall()` that a reply is not expected. Guidelines for using this flag value correctly when a process is in transaction mode are discussed in Chapter 5, "Global Transactions in BEA TUXEDO System." When this flag is set, on success `tpacall()` returns the value of `0` as the reply descriptor, where `0` cannot be used by `tpgetrply()`.

An example of `tpacall()` using the `TPNOREPLY|TPNOTRAN` flags is shown in Listing 2-19. This example is similar to the one presented above. In this case, however, a reply is not expected from the `PRINTER` service. By setting both of these flags, the client is indicating that no reply is expected and the `PRINTER` service is not to be a participant in the current transaction. Chapter 7 fully discusses this situation. Refer to the section called "Transaction Rules."

Listing 2-19 Sending an Asynchronous Message with TPNOTRAN|TPNOREPLY

```
#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
}
```

On error, `tpacall()` returns `-1` and sets `tperrno` to a value that reflects the nature of the error. `tpacall()` returns many of the same error codes as `tpcall()`. Again, the differences are based on the fact that one represents a synchronous call and the other an asynchronous call. These errors are discussed at length in Chapter 7, "Error Management."

Listing 2-20 illustrates a series of asynchronous calls being made that make up the total bank balance query. Since the banking application data is distributed among several database sites, an SQL query needs to be executed against each one. The audit client chooses to do this by selecting representative branch identifiers (that is, one branch identifier per database site), and calling the ABAL or TBAL service for each one. The representative branch identifier is not used in the actual SQL query, but it does cause the BEA TUXEDO system to route the request to the proper database site. In the following code, the for-loop invokes `tpacall`() once for each site. We'll see this same logic handled in a different way in Chapter 4, "Conversational Clients and Services."

Listing 2-20 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: tpgetrply()

`tpgetrply()` is the complementary function to `tpacall()`. It dequeues a reply from a request previously sent by `tpacall()`. The syntax of this function is:

```
int
tpgetrply(cd, data, len, flags)  /* Receive reply to service request */
int *cd;                         /* Call Descriptor */
char **data;
long *len, flags;
```

`tpgetrply()` takes the address of the call descriptor returned by `tpacall()` as its first parameter, *cd*. In the default case, the function waits for the arrival of the reply that corresponds to the value pointed to by the *cd* parameter. In waiting for this specific reply, a blocking time-out may occur. A time-out means that `tpgetrply()` fails and `tperrno` is set to `TPETIME` (unless its *flags* parameter is set to `TPNOTIME`).

The second and third parameters to `tpgetrply()`, *data* and *len*, have identical semantics to those of the *odata* and *olen* parameters of the `tpcall()` function. *data* contains the address of a pointer that was previously assigned by a call to `tpalloc()`.

Getting and Setting Priority

ATMI provides two functions that allow you to determine and set the priority of the message request. The priority affects how the request is dequeued by the server. Servers dequeue requests with the highest priorities first. The syntax of these functions is:

```
int
tpgprio(); /* Get service request priority */
```

and

```
int
tpsprio(prio, flags); /* Set service request priority */
int prio;
long flags;
```

The `tpgprio()` function can be called by a requester after invoking the `tpcall()` or `tpacall()` function to retrieve the priority of the request message just sent. If it was called and no request was sent, the function fails returning `-1` and setting `tperrno` to `TPENOENT`. Upon success, `tpgprio()` returns an integer value in the range of 1 to 100, 100 being the highest priority value. If the priority has not been explicitly set by using the `tpsprio()` function, the value of the priority will be that of the service routine that handles the request. The priority of the service is assigned the system default value of 50 unless it has been specifically defined to some other value by the administrator. See Listing 2-21 for an example of determining the priority of a message that was sent in an asynchronous call.

Listing 2-21 Determining the Priority of the Sent Request

```
#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
}
. . .
}
```

It is also possible to use this function to retrieve the priority of the request just received by the service. This is illustrated in Listing 3-3 in Chapter 3, "Writing Service Routines."

With the `tpsprio()` function, the programmer can override the priority level the request would normally inherit from the service to which it is dispatched. When `tpsprio()` is called, it affects the priority level only of the very next request that is sent by `tpcall()` or `tpacall()` or forwarded by a service subroutine. This function takes two parameters; the second one indicates how the first one is to be interpreted. The first parameter, *`prio`*, is an integer. In the default situation, its sign indicates whether the request's priority should be incremented or decremented in relation to the existing priority. For the first parameter to be treated as a relative value, the second parameter, *`flags`*, must be set to `0`. If it is set to `TPABSOLUTE`, the priority value of the next request that is sent out will receive the absolute value of the integer contained in the *`prio`* parameter. The absolute value of *`prio`* must be in the range of 1 to 100. If the value is not in this range, the system uses the default value, 50.

Listing 2-22 shows an excerpt from the `TRANSFER` service acting as a client process to call services of `WITHDRAWAL`. It invokes `tpsprio()` to increase the priority of the request message it sends in its synchronous call to `WITHDRAWAL`. It does so to prevent the request from being queued for the `WITHDRAWAL` service (and later the `DEPOSIT` service) after already having waited on the `TRANSFER` queue.

Listing 2-22 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);
}
```

Initiating a Conversational Connection

The discussion in this chapter has centered around how client programs initiate a request/response service request. Client programs can also connect to conversational servers by using `tpconnect()` instead of `tpcall()` or `tpacall()`. Chapter 4, “Conversational Clients and Services,” describes that in detail.

Sending a Broadcast Message

The `tpbroadcast()` function is used to send an unsolicited message to registered clients within the application. It is mentioned in this chapter on client programs because it can be called by clients. A more complete discussion of its use can be found in Chapter 3, “Writing Service Routines.”

Compiling Client Programs

To compile your client programs you have several methods to choose from. You can use regular C Compilation System utilities to make object files. The object files can be kept as individual files or collected into an archive file. If you prefer, you can retain your programs as source (`.c`) files. In any event, when you invoke `buildclient` to produce an executable client, you specify your input files on the command line with the `-f` option.

The buildclient Command

`buildclient`(1) is used to put together an executable client program. Options identify the name of the output file, input files provided by the application, and various libraries.

`buildclient` invokes the UNIX `cc` command. The environment variables `CC` and `CFLAGS` can be set to name an alternative compile command and to set flags for the compile and link edit phases.

The buildclient -o Option

The `-o` option is used to assign a name to the executable output file. If no name is provided, the file is named `a.out`.

The buildclient -f and -l Options

The `-f` and `-l` options are used to specify files to be used in the link edit phase. The files specified in the `-f` (first) option are brought in before the BEA TUXEDO system libraries, whereas the files specified in the `-l` (last) option are brought in after these libraries. There is a significance to the order of the options. The order is dependent on function references and in what libraries the references are resolved. Input files should be listed ahead of libraries that might be used to resolve their references. If input files are `.c` files, they are first compiled. Object files can be either separate `.o` files or groups of files in archive (`.a`) files. If more than a single file name is given as an argument to a `-f` or `-l` option, the syntax calls for a list enclosed in double quotes. You can use as many `-f` and `-l` options as you need.

The following represents the command line that was used to create the `audit` executable program. The environment variable `CC` is set to `cc` and the environment variable `CFLAGS` is set to `-I $TUXDIR/include`.

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

3 Writing Service Routines

Writing Request/Response Services

The preceding chapter discussed the ATMI primitives that can be used to write client programs. In this chapter, some of the same functions are revisited in the context of the service subroutines. As you may recall, services are C subroutines that are linked together with the BEA TUXEDO system-provided `main`() to create executable server programs.

In this chapter the discussion covers only services that operate in a request/response mode. Conversational clients and servers are the subject of Chapter 4, “Conversational Clients and Services.”

Examples Taken from the Sample Application

Most of the examples shown are taken from the services of the banking application.

Application Service Template

Since the communication details are taken care of by BEA TUXEDO system's `main()`, the programmer can concentrate on the application logic rather than communication implementation. For services to be compatible with the `main()` provided, they must adhere to certain conventions. These conventions are referred to as the service template for coding service routines; they are described here and on the `tpservice`(3c) reference page of the *BEA TUXEDO Reference Manual*.

Request/response services have the following characteristics:

- A request/response service can receive only one request at a time and can send only one reply.
- When servicing a request, it works only on that request and can accept another only after it has sent its reply to the requester or has 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 control returns to BEA TUXEDO system's `main()` instead of the calling function.
- When communicating with another server via `tpacall()`, the initiating service must wait for all outstanding replies or must 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.

The following sections examine these concepts more closely.

The TPSVCINFO Structure

The typical service routine is defined as a function receiving one argument that is a pointer to a structure. This service information structure is `typdef`'d as `TPSVCINFO` in the `atmi.h` header file and includes the following members:

```
char      name[32]; /* service name being invoked */
long      flags;    /* describes service attributes */
char      *data;    /* request data */
long      len;      /* request data length */
int       cd;       /* connection descriptor if (flags & TPCONV) true */
int       appkey;   /* application authentication client key */
CLIENTID  cltid;    /* client identifier for originating client */
```

The members of the structure

- ♦ indicate to the service routine the name with which it was invoked
- ♦ tell the service attributes about itself or the caller
- ♦ point to the request data
- ♦ indicate the length of the request data
- ♦ give the connection descriptor, if this is a conversational connection
- ♦ provide the client key for authentication
- ♦ carry the identifier for the client originating the call

The name Member of TPSVCINFO

The *name* member of the structure indicates to the service routine the name that the requesting process used to invoke the service.

The flags Member of TPSVCINFO

The *flags* member of the structure is used to let the service know if it is in transaction mode or if the caller is expecting a reply. The various ways a service can be placed in transaction mode are discussed in Chapter 5, “Global Transactions in BEA TUXEDO System.” If the value of *flags* is `TPTRAN`, it indicates that the service is in transaction mode. When a service is called by `tpcall`() or `tpacall`() with the *flags* parameter set to `TPNOTRAN`, it indicates that the service cannot participate in the current transaction, but it is still possible for the service to be in transaction mode. So even when the caller sets the `TPNOTRAN` communication flag, it is possible for `TPTRAN` to be set in `svcinfo->flags`. The case that allows this to happen is discussed in Chapter 5, “Global Transactions in BEA TUXEDO System.” The *flags* member is set to `TPNOREPLY` if the service was called by `tpacall`() with the `TPNOREPLY` communication flag set. It is possible for the *flags* member to be set to both of these values. When this represents a valid situation is discussed in the next chapter. However, if a called service is part of the same transaction as the calling process, it must return a reply to the caller.

The data and len Members of TPSVCINFO

The *data* member points to a buffer that was previously allocated by `tpalloc`() within the server `main()`; this buffer is used to receive the request message. The *len* member contains the length of the request data that is in the buffer pointed to by *data*. It is recommended that you use this buffer to send back the reply message or forward the request message. This is further discussed when explaining the proper usage of the `tpreturn`() and `tpforward`() functions. The contents of the buffer get overwritten each time the service routine is invoked regardless of whether the buffer is used as the message buffer for returning or forwarding the reply.

The appkey Member of TPSVCINFO

The use of this member is left to the application to decide. If application-specific authentication is part of your design, the application-specific authentication server, which is called at the time a client joins the application, should return a client authentication key as well as a success/failure indication. (This is the logic of the BEA TUXEDO system default `AUTHSVC` service.) The key is held by the system on behalf of the client and is passed to subsequent service requests in the *appkey* field. By the time the key is passed to the service, the client has already passed authentication, but the *appkey* field can be used within the service to identify in some way the user invoking the service or some other parameters associated with the user. If not used, the value is set to `-1` by the system.

The cltid Member of TPSVCINFO

The *cltid* member is a structure of type `CLIENTID`. It is used by the system to carry the identification of the client. You should not make changes in this structure.

Accessing Data that Comes with the Request

When accessing the request data pointed to by *data*, the service must be coded to expect the data to be in a buffer of the type defined for the service in the configuration file. For everything to be interpreted correctly by the system, the type and subtype of the request buffer passed by the calling process must agree with the type that is coded for the service called which, in turn, must agree with the typed buffer as defined for that service in the configuration file.

Listing 3-1 illustrates a typical service definition; this one is taken from the ABAL (account balance) service routine. `ABAL` is part of the `BAL` server.

Listing 3-1 Typical Service Definition

```
#include <stdio.h>      /* UNIX */
#include <atmi.h>       /* TUXEDO */
#include <sqlcode.h>    /* TUXEDO */
#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 Listing 3-1, the request buffer on the client side was originally allocated by a call to tpalloc() with the *type* parameter set to VIEW and the *subtype* set to aud. The ABAL service is defined in the configuration file as a service that knows about the VIEW typed buffer. (This is by implication; the BUFTYPE parameter is not specified for ABAL, which means it defaults to ALL.) ABAL's server main() allocated a buffer of the VIEW type and assigned the pointer to this buffer to the data member of the TPSVCINFO structure that was passed to the ABAL subroutine. ABAL is able to retrieve the data buffer by accessing the data member as illustrated in the above example. Note that after this buffer is retrieved and before the first database access is made, the consistency level of the transaction is specified. Refer to the "Global Transactions and Resource Managers" and the "Comprehensive Example" sections in Chapter 7 for more details on transaction consistency levels.

Checking the Buffer Type

Listing 3-2 shows the service accessing the data buffer to determine its type. This service knows about more than one buffer type and invokes the tptypes() ATMI function primitive to determine the buffer type of the received request. It also finds out the maximum size of the buffer so it knows whether to reallocate the buffer size or not. This example is derived from the ABAL service. It represents what the subroutine would look like if it accepted its request either as an aud VIEW or an FML buffer. If its attempt to determine the message type fails, it sends back 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. The tpreturn() function is discussed after priority; it is included in this example for completeness.

Listing 3-2 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);
   }
}
```

Checking the Priority of the Service Request

Listing 3-3 shows the fictitious `PRINTER` service testing the priority level of the request just received by invoking the `tpgprio()` function. Based on the priority level, the print job is routed to the appropriate destination printer. The contents of `pbuf->data` are piped to that printer. Also, `pbuf->flags` is queried to see if a reply is expected. If one is expected, the name of the destination printer is returned to the client. Again, the use of `tpreturn()` is explained in the next section.

Listing 3-3 Determining the Priority of the 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++]);
}
```

The tpreturn() and tpforward() Functions

`tpreturn()` and `tpforward()` are functions that indicate that a service routine has completed; they either send a reply back to the calling client or forward a request to another service for further processing.

Sending Replies

The primary function of a service routine is to process a request and return the reply to a client process. In performing this function, a service can in turn act as a requester and make request calls to other services with `tpcall()` or `tpacall()`. When `tpreturn()` is called, control always returns to `main()`. If the 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 BEA TUXEDO system's `main()`, and an error is returned to the caller.

The `tpreturn()` function, besides marking the end of the service routine, also causes the reply message to be sent to the requester. If the client invoked the service with `tpcall()`, after a successful call to `tpreturn()`, the reply message is available in the buffer pointed to by **odata*. If `tpacall()` was used to send the request, on success from `tpreturn()`, the reply message is available in the `tpgetrply()` buffer that is pointed to by **data*. The syntax of this function is:

```
void
tpreturn(rval, rcode, data, len, flags) /* End service routine */
int rval, rcode;
char *data;
long len, flags;
```

Currently the *flags* argument is not used.

tpreturn() Arguments: rval

The *rval* parameter can be set to TPSUCCESS, TPFAIL, or TPEXIT. This value indicates whether the service has completed successfully or not on an application-level. These conditions are communicated to the calling client in the following way. When set to TPSUCCESS, the calling function succeeded, and if there is a reply message, it is in the caller's buffer. If the service terminated unsuccessfully, (that is, the logic of the application set *rval* to TPFAIL), an error is reported to the client process waiting for the reply. The client's tpcall() or tpgetrply() function call will fail and the tperrno variable will be set to TPESVCFAIL to indicate an application-defined failure. In the case of this kind of failure, if a reply message was expected, it will be available in the caller's buffer. If TPEXIT is set in *rval*, the functionality of TPFAIL is performed, but the server exits after the reply is sent back to the client. Note that if *rval* is not set, the default value of TPFAIL is assigned to this parameter. The impact of the value of this parameter when a process is in transaction mode is discussed in Chapter 5, "Global Transactions in BEA TUXEDO System."

The preceding discussion concerns the effect of *rval* if application-defined errors are the only ones that occur. If, however, tpreturn() encounters errors while processing its arguments, it sends a *failed* message (if a reply is expected) to the calling process. This is detected by the caller by the value placed in tperrno. In case of *failed* messages, tperrno is set to TPESVCERR. This situation overrides the effect of the value of *rval*. If this type of error occurs, no reply data is returned, and the contents of the caller's output buffer and its length remain unchanged.

If tpreturn() sends back a message in a buffer whose type is not known or not allowed by the caller (that is, the call was made with *flags* set to TPNOCHANGE), TPEOTYPE is returned in tperrno. Application success or failure cannot be determined, and the contents of the caller's output buffer and its length remain unchanged.

Also, the value returned in *rval* is not relevant in the case when tpreturn() is invoked and a time-out occurs for the call waiting on the reply. This situation overrides all others in determining the value that is returned in tperrno. tperrno 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 time-outs in BEA TUXEDO. Blocking time-out was discussed when explaining the TPNOBLOCK and TPNOTIME communication flags. The other type of time-out, transaction time-out, is discussed in Chapter 5, "Global Transactions in BEA TUXEDO System."

tpreturn() Arguments: rcode

The *rcode* parameter can be used to return to the caller an application-defined return code. The client can access the value returned in *rcode* by querying the `tpurcode` global variable. This code is sent regardless of application success or failure; that is, it is returned in the case of success or `TPESVCFAIL`. As indicated, no reply messages can be sent in the other error cases.

tpreturn() Arguments: data and len

data points to the reply message that is to be returned to the client process. The message buffer must have been allocated by a previous call to `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 since they are handled by the system supplied `main`(). In fact, it is not possible to free this buffer in the service subroutine. Any attempt to free the buffer using `tpfree`() quietly fails, achieving nothing. However, this buffer can be grown by a service routine with a call to `tprealloc`(). BEA TUXEDO treats the original buffer the same whether it has been resized or not. If a buffer other than the one that was passed to the service routine is used to return the message, it is up to the programmer to allocate it by invoking the `tpalloc`() function within the service routine. The buffer obtained in this way is automatically freed by `tpreturn`(). If the reply message does not have a data part, no buffer is required; simply set *data* to the NULL pointer. The *`len`* parameter indicates the amount of data in the reply buffer, and it is this value that can be accessed in the *`olen`* parameter of the `tpcall`() or the *len* parameter of the `tpgetrply`() function. As indicated earlier, the process acting as the client can use this returned value to test to see if the reply buffer has grown. If a reply is expected by the client, and there is no data in the reply buffer, that is, *data* is set to the NULL pointer, a reply with zero length is sent to the client. The pointer to and the contents of the client's buffer remain unchanged. When the *data* pointer is NULL, `tpreturn`() ignores the `len` parameter. If no reply is expected, that is, `TPNOREPLY` was set, `tpreturn`() ignores the buffer and length parameters and simply returns control to `main`(); the server process is then free to process another request.

tpreturn() Example

Listing 3-4 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 different buffer for the reply message since it must use the contents of the request buffer for the calls to both the `WITHDRAWAL` and the `DEPOSIT` services. If the call to WITHDRAWAL should fail, cannot withdraw is written to the status line of the form, the reply buffer is freed, and the *rval* parameter to tpreturn() is set to TPFAIL. If the call succeeds, the debit balance is retrieved from the reply buffer.

Note: The "to-account id" retrieved in the variable cr_id in Listing 3-4 is moved to the zeroth occurrence of the ACCOUNT_ID field in the transf fielded buffer. It is necessary to assign it to this position since it is this occurrence of a field in an FML buffer that is used for data dependent routing. Refer to the book *Administering the BEA TUXEDO System.*

A similar scenario is followed for the call to DEPOSIT. On success, the service frees the reply buffer that was allocated within the service routine and sets *rval* to TPSUCCESS and returns the pertinent account information to the status line.

Listing 3-4 How to Use tpreturn()

```
#include <stdio.h>      /* UNIX */
#include <string.h>     /* UNIX */
#include "fml.h"        /* TUXEDO */
#include "atmi.h"       /* TUXEDO */
#include "Usysflds.h"   /* TUXEDO */
#include "userlog.h"    /* TUXEDO */
#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: tpcancel()

If a service calling `tpgetrply`() fails with `TPETIME` and decides not to wait any longer, it can invalidate the descriptor with a call to `tpcancel`(). If the reply ever does arrive, it is silently discarded. `tpcancel`() cannot be used for transaction replies (request was done without the TPNOTRAN flag); within a transaction, `tpabort`() does the same job of invalidating the transaction call descriptor. Listing 3-5 shows the code.

Listing 3-5 Invalidate a Reply after Timing Out

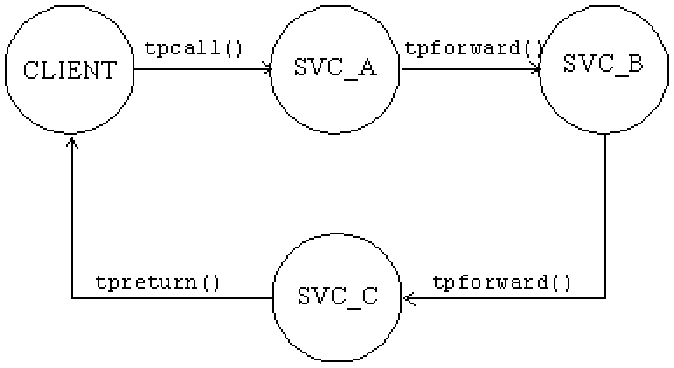
```
int cd1;
 .
 .
 .
      if ((cd1=tpacall(sname, (char *)audv, sizeof(struct aud),
         TPNOTRAN)) == -1) {
      .
      .
      .
      }
      if (tpgetrply(cd1, (char **)&audv,&audrl, 0) == -1) {
        if (tperrno == TPETIME) {
          tpcancel(cd1);
          .
          .
          .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

Forwarding Requests

The `tpforward`() function allows a service to forward a request to another service for further processing. This differs from a service call in that the service that forwards the request does not ever expect a reply. The reply is owed to the process that originated the request, and the responsibility for providing the reply has been passed to the service to which the request has been forwarded. It becomes the responsibility of the last server in the forward chain to send the reply back by invoking `tpreturn`(). The process that made the initial service call is the client and will be waiting for a reply.

The following figure gives you an idea of what a forward chain might look like. The request is initiated with a `tpcall`() and the eventual reply is provided by the `tpreturn`() that is invoked by the last service in the chain.

Figure 3-1 Forwarding a Request



Service routines can forward requests at specified priorities in the same manner that client processes send requests. You may recall that this is accomplished by invoking the `tpsprio`() function.

`tpforward`() is identical to `tpreturn`() in that when it is called, `main`() regains control, and the server process is free to do more work. The syntax of this function is:

```
void
tpforward(svc, data, len, flags) /* Forward request */
char *svc, *data;
long len, flags;
```

tpforward() Arguments

The first parameter of `tpforward`(), *svc*, is a character pointer that references the name of the service to which the request is to be forwarded. The request buffer is pointed to by its second parameter, *data*, and the length of the request data is available in *len*. These two parameters and the remaining one, *flags*, share the same meanings as the corresponding ones specified for `tpreturn`(). Recall that, at present, *flags* has no defined values.

Note: When acting as a client, a server process is not allowed to request services from itself when a reply is expected. If the only available instance of the desired service is offered by the server process making the request, the call will fail indicating that a recursive call would have been made. However, if the service routine sends the request with the `TPNOREPLY` communication flag set or forwards the request, the call will not fail since the caller is not waiting on 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, call `tpreturn`() with *rval* set to `TPFAIL`.

tpforward() Example

Listing 3-6 is taken from the `OPEN_ACCT` service routine which is part of the `ACCT` server. It shows what the service would look like if it used a call to `tpforward`() to send its data buffer to the `DEPOSIT` service. The example illustrates testing of the `SQLCODE` to see if the account insertion was successful. If the new account is added successfully, the branch record is updated to reflect the new account. On success, the data buffer gets forwarded to the `DEPOSIT` service. On failure, `tpreturn()` is called with *rval* set to `TPFAIL` and the failure reported to the status line of the form.

Listing 3-6 How to Use tpforward()

```
 ...
/* 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);
```

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 (`tpbroadcast()`) or by an identifier received with a previously processed message (`tpnotify()`). Messages sent via `tpbroadcast()` can originate either in a service or in another client. Messages sent via `tpnotify()` can originate only in a service, as shown in the following table.

	Initiator	**Receiver**
`tpbroadcast()`	client, server	client
`tpnotify()`	server	client

tpbroadcast() Arguments

`tpbroadcast()` allows a message to be sent to registered clients of the application. (Registered clients are those that have successfully made a call to `tpinit()` and have not yet made a call to `tpterm()`.) The syntax of the function is:

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

lmid, *usrname*, and *cltname* are pointers to identifiers used to select the target list of clients. A value of `NULL` for any of these arguments acts as a wildcard for that argument, so the message can be directed to groups of clients or to the entire universe.

The *data* argument points to the content of the message up to the length specified by the *len* argument. If *data* points to a self-defining buffer type, for example, an `FML` buffer, *len* can be `0`. The *flags* argument can be:

`TPNOBLOCK`
: If a blocking condition exists, don't send the message.

`TPNOTIME`
: Wait indefinitely; do not time out.

`TPSIGRSTRT`
: When a signal interrupts any underlying system calls, the call is reissued. If this flag is not set, a signal causes `tpbroadcast()` to fail with the `TPGOTSIG` error code.

tpbroadcast() Example

Listing 3-7 shows an example of a call to `tpbroadcast()` where all clients are targeted. The message to be sent is in a `STRING` buffer.

Listing 3-7 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
```

tpnotify() Arguments

`tpnotify`() can be called only from a service. The syntax of the function is:

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

**clientid* is a pointer to a `CLIENTID` structure saved from the `TPSVCINFO` structure that accompanied the service request to this service. Thus it can be seen that `tpnotify`() is used to direct an out-of-band message to the client process that called the service. This is not the same as the reply to the service request that would be sent when the service calls `tpreturn`() (or when a conversational service calls `tpsend`() to send a reply to the client), nor is it any part of a transaction, if one is in progress. It is used in cases where the service encounters information in processing that needs to be passed to the unsolicited message handler for the application.

The **data*, *len*, and *flags* arguments are the same as they are for `tpbroadcast`().

Advertising, Unadvertising Services

When servers are booted, they advertise the services they offer based on the specification in their `CLOPT` parameter in the configuration file. The default specification calls for the server to advertise all services with which it was built; this is the meaning of the `-A` option. (See `ubbconfig`(5) or `servopts`(5).) When a service is advertised, it takes up a service table entry in the bulletin board. This can lead an application to decide to boot servers to offer some subset of their available services. As the `servopts`(5) reference page makes clear, the `-s` option allows a comma-separated list of services to be specified by service name. It also allows, with the `-s` *services:func* notation, for a function with a name different from that of the advertised service to be called to process the service request. The BEA TUXEDO system administrator can use the `advertise` and `unadvertise` commands of `tmadmin`(1) to control the services offered by servers.

The `tpadvertise`() and `tpunadvertise`() functions allow that dynamic control to be exercised within a service of a request/response server or conversational server to advertise or unadvertise a service. The limitation is that the service to be advertised (or unadvertised) must be available within the same server as the service making the request.

tpadvertise() Arguments

The syntax of `tpadvertise` is:

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

**svcname* is a pointer to a character string of 15 characters or less that names the service to be advertised. Names longer than 15 characters are truncated; a `NULL` value causes an error, `[TPEINVAL]`.

func is the address of a BEA TUXEDO system service function that is called to perform the service (of course, it is not uncommon that this name is the same as the name of the service). *func* is not permitted to be `NULL`.

tpadvertise() Example

Listing 3-8 shows an example of `tpadvertise`() that is based on the following hypothetical situation (this is an extension to an existing `bankapp` server):

`SERVER TLR` is specified to offer only the service `TLR_INIT` when booted.

After some initialization, `TLR_INIT` advertises services `DEPOSIT` and `WITHDRAW`, both performed by function `tlr_funcs`, and both built into the `TLR` server.

On return from advertising the two services, `TLR_INIT` unadvertises itself.

Listing 3-8 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);
```

tpunadvertise()

`tpunadvertise`(), of course, is called to remove a service from the service table of the bulletin board. The syntax is:

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

The only argument is a pointer to the *svcname* being unadvertised. An example is included above in Listing 3-8.

System-Supplied Servers and Subroutines

The BEA TUXEDO system is delivered with a server that provides a basic client authentication service: `AUTHSVR`. A standard `main`() routine and two subroutines called by `main`() are also provided.

System-Supplied Servers

The servers described in this section are intended to save you the trouble of coding services to do routine tasks.

AUTHSVR

`AUTHSVR`(5) can be used to provide individual client authentication for an application. It is called by `tpinit`(3c) 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 in the *passwd* field of the `TPINIT` buffer). The string in *data* is checked against the `/etc/passwd` file (by default; the application can specify a different file to be checked). When used by a native site client, the *data* field is sent along by `tpinit`() as it is received. This means that if the application wants the password to be encrypted, the client program must be coded accordingly. When used by a workstation client, `tpinit`() encrypts the data before sending it across the network.

The BEA TUXEDO System main()

To speed the development of servers, the BEA TUXEDO system provides a predefined `main`() routine for server load modules. This `main`() is automatically included when the `buildserver`(1) command is executed.

The predefined `main`() routine does the following:

- runs the process immune to hangups (ignores the UNIX system `SIGHUP` signal)
- arranges for cleanup on receipt of the standard UNIX 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 (`--`) that indicates the end of system-recognized arguments.
- calls the function `tpsvrinit`() to process any command line arguments occurring after the `--` and optionally to open the resource manager. Such arguments are for application-specific initialization.
- until ordered to halt, checks its request queue for service request messages
- until ordered to halt, when a service request message arrives on the request queue:
  - if the `-r` option was 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

- until ordered to halt, when the service has returned from processing its input:
    - if the `-r` option was specified, records the ending time of the service request
    - updates statistics
    - updates the bulletin board to indicate that the server is `IDLE`; that is, ready for work
    - checks its queue for the next service request
- when the server is about to halt, calls `tpsvrdone`() to perform any required user shutdown operations.

The `main`() that the system provides is a closed abstraction and can not be modified by the programmer. As indicated in the previous list items, it takes care of all the details concerning entrance into and exit from an application, buffer and transaction management, and communication. It leaves the programmer free to implement the application through the logic of the service subroutines. Note that as a result of the system supplied `main`() doing the work of joining and leaving the application, it is an error for services to make calls to the `tpinit`() or `tpterm`() functions. This error returns TPEPROTO in `tperrno`.

In addition to the above functionality, there are two user exits in `main`() that allow the programmer to do various initialization and exiting activities. The next sections explain how these two system supplied subroutines are used.

BEA TUXEDO System-Supplied Subroutines

There are two subroutines of `main`(), `tpsvrinit`() and `tpsvrdone`(), that are provided with the BEA TUXEDO system software. The default versions can be modified to suit your application.

tpsvrinit()

When a server is booted, the BEA TUXEDO system `main`() calls `tpsvrinit`() during its initialization phase before it handles any service requests. If an application does not provide this routine in a server, the default one is called that opens the resource manager and makes an entry in the central event log indicating that the server has successfully started. The central event log is discussed in Chapter 7, “Error Management.” For now, simply understand that it is a UNIX System file to which processes can write messages by calling the `userlog`(3c) function. Coming as it does near the beginning of the system-supplied `main`(), `tpsvrinit`() can be used for any initialization purposes that might be needed by an application. Two possibilities are illustrated here: receiving command line options and opening a database.

Note that although not shown in the following examples, message communication can also be performed within this routine. However, `tpsvrinit`() fails if it returns with asynchronous replies pending. In addition, the replies are ignored by BEA TUXEDO, and the server exits gracefully. `tpsvrinit`() can also start and complete transactions, but this is discussed in Chapter 7, “Error Management.”

The syntax of this function is:

```
int
tpsvrinit(argc, argv)  /* Server initialization routine */
int argc;
char **argv;
```

Using tpsvrinit() to Receive Command Line Options

When a server is booted, before calling the `tpsvrinit()` routine, it reads the options specified for it in the configuration file. Using the UNIX function `getopt`(3C) (see a UNIX System programmer’s reference manual), it reads options up to the point where it receives an EOF indication. The presence of a double dash (`--`) on the command line causes `getopt` to return an EOF. `getopt` places the *argv* index of the next argument to be processed in the external variable `optind`. The predefined `main`() then calls `tpsvrinit`().

Listing 3-9 shows an example of a `tpsvrinit`() coded to receive command line options.

Listing 3-9 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 the BEA TUXEDO system's `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 take an argument, as indicated by the colon. `optarg` points to the beginning of the option argument. We have omitted the switch statement logic.

Using tpsvrinit() to Open a Resource Manager

Listing 3-10 shows a code fragment that illustrates another common use of `tpsvrinit()`: opening a resource manager. BEA TUXEDO provides functions to open a resource manager, `tpopen()` and `tx_open()`. It also provides the complementary functions, `tpclose()` and `tx_close()`. The details of these ATMI primitives can be found in the *BEA TUXEDO Reference Manual.* Applications that use these calls to open and close their resource managers are portable in this respect. They work by accessing the resource manager instance-specific information that is available in the configuration file. These 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. In the example that follows, the code does not pick up command line options, but there is no reason it could not both pick up options and open the database. Also, note the use of the `userlog`(3c) function to write to the central event log.

Listing 3-10 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 /T 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);
}
```

If an error occurs during the initialization activities, `tpsvrinit()` can be coded to permit the server to exit gracefully before the server starts processing service requests.

tpsvrdone()

Using tpsvrdone() to Close a Resource Manager

As might be expected, `tpsvrdone()` can call on the services of `tx_close()` to close the resource manager in a manner analogous to the way `tpsvrinit()` and `tx_open()` are used to open it. If the application does not define a closing routine for `tpsvrdone()`, the BEA TUXEDO system calls the default version, which calls `tx_close()` and `userlog()` to close the resource manager and write to the central event log. The message to the log indicates that the server is about to exit.

Note: Applications choosing to write their own versions of `tpsvrinit`() and `tpsvrdone`() should remember that the default versions of these two routines call `tx_open`() and `tx_close`(), respectively. If the application writes a new version of `tpsvrinit`() that calls `tpopen`() rather than `tx_open`(), they should also write a new version of `tpsvrdone`() that calls `tpclose`(). In other words, the open/close pairs have to be from the same set.

`tpsvrdone`() is called after the server has finished processing service requests but before it exits. Since the server is still part of the system, further communication and transactions can take place within the routine. The rules that must be followed to do this properly are covered in Chapter 7, "Error Management." The syntax of this function is:

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

Listing 3-11 shows the typical way in which `tpsvrdone`() is used.

Listing 3-11 Closing a Resource Manager in 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;
}
```

Compiling Subroutines to Build Servers

To compile your service subroutines, you have the same freedom you had in compiling clients. You can use regular C Compilation System utilities to make object files. The object files can be kept as individual files or collected into an archive file. If you prefer, you can retain them as source (`.c`) files. In any event, when you invoke `buildserver` to produce an executable server, you specify them on the command line with the `-f` option. This applies to new versions of `tpsvrinit`() and `tpsvrdone`() as well as your application subroutines.

The buildserver Command

`buildserver`(1) is used to put together an executable server with the BEA TUXEDO system's `main`(). Options identify the name of the output file, input files provided by the application, and various libraries that permit you to run a BEA TUXEDO system application in a variety of ways.

`buildserver` invokes the `cc` command. The environment variables `CC` and `CFLAGS` can be set to name an alternative compile command and to set flags for the compile and link edit phases. The key `buildserver` command line options are described in the paragraphs that follow.

The buildserver -o Option

The `-o` option is used to assign a name to the executable output file. If no name is provided, the file is named `SERVER`.

The buildserver -f and -l Options

The `-f` and `-l` options are used to specify files to be used in the link edit phase. The files specified in the `-f` option are brought in before the BEA TUXEDO system and resource manager libraries (first), whereas the files specified in the `-l` option are brought in after these libraries (last). There is a significance to the order of the options. The order is dependent on function references and in what libraries the references are resolved. Source modules should be listed ahead of libraries that might be used to resolve their references. Any `.c` files are first compiled. Object files can be either separate `.o` files or groups of files in archive (`.a`) files. If more than a single file name is given as an argument to a `-f` or `-l` option, the syntax calls for a list enclosed in double quotes. You can use as many `-f` and `-l` options as you need.

The buildserver -r Option

The `-r` option is used to specify which resource manager access libraries should be link edited with the executable server. The choice is specified with a string from the `$TUXDIR/udataobj/RM` file. Only one string can be specified. The database functions in your service are the same regardless of which library is used.

All valid strings that name resource managers are contained in the `$TUXDIR/udataobj/RM` file. When integrating a new resource manager into BEA TUXEDO, this file must be updated to include the information about the resource manager. Refer to the `buildtms`(1) reference page and the book *Administering the BEA TUXEDO System* for more information.

If you are using the ATMI transaction primitives, `tpbegin`() and `tpcommit`()/`tpabort`(), you should build your servers using the `buildserver` command.

The buildserver -s Option

The `-s` option is used to specify the service names included in the server and the name of the functions that perform each service. Normally, the function name is the same as the name of the service. In the sample program, our convention is to specify all uppercase for the service name. For example, the `OPEN_ACCT` service would be processed by function `OPEN_ACCT`(). However, the `-s` option of `buildserver` does allow you to specify an arbitrary name for the processing function for a service within a server. For example, the command

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

specifies that the `NEW_ACCT` request is to be processed by a function called `OPEN_ACCT`(), while service request `CLOSE_ACCT` is to be processed by the function `CLOSE_ACCT`().

However, it is possible for the administrator to specify that only a subset of the services that were used to create the server with the `buildserver` command are to be advertised when the server is booted. Refer to the book *Administering the BEA TUXEDO System*.

Using C++

There are not many differences in using a C++ compiler instead of a C compiler to develop application servers. The two areas affected are the declaration of the service function, and the use of constructors and destructors.

When declaring a service function, it must be declared to have “C” linkage using `extern “C”`. That is, the function prototype should be as follows:

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

Declaring the name with “C” linkage ensures that the C++ compiler will not *mangle* the name (many C++ compilers change the function name to include type information for the parameters and function return). This allows for the linking of both C and C++ service routines into a single server without the application programmer indicating what type each routine is. This also allows use of dynamic service advertisement, which requires accessing the symbol table of the executable to find the function name.

C++ constructors are called to initialize class objects when they are created, and destructors are invoked when class objects are destroyed. For automatic (local, non-static) variables that have 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 `tpreturn`() or `tpforward`() is called, it does a non-local goto (using `longjmp`(3)) such that destructors for automatic variables will not be called. To avoid this problem, the application should be written such that `tpreturn`() or `tpforward`() is called from the service routine (instead of any functions called from the service routine). In addition, either 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) or they should be declared and used in a nested scope (within curly brackets `{}`) such that the scope ends before calling `tpreturn`() or `tpforward`(). Summarizing, there should be no automatic variables with destructors in scope in the current function or on the stack when `tpreturn`() or `tpforward`() is called.

For proper handling of global and static variables that have constructors and destructors, many C++ compilers require that the `main`() must be compiled using the C++ compiler (special processing is included in the `main`() to ensure that the constructors are executed when the program starts and the destructors are executed when the program exits). Since the `main()` is provided by the BEA TUXEDO system, the application programmer does not compile it directly. To ensure that the file is compiled using C++, the `buildserver` command must use the C++ compiler. This is done by setting the CC environment variable to the full pathname for the C++ compiler and setting the `CFLAGS` environment variable to any options to be provided on the C++ command line.

4 Conversational Clients and Services

Writing Conversational Clients and Services

This chapter covers the subject of conversational clients and services.

A conversational client differs in the following ways from a request/response client (described in Chapter 2, "Writing Client Programs,"):

- ♦ It initiates a request for service by using `tpconnect`() rather than `tpcall`() or `tpacall`().
- ♦ It passes the service request to a conversational server.

A conversational service differs in the following ways from a request/response service (described in Chapter 3, "Writing Service Routines,"):

- ♦ It is part of a server identified in the configuration file as offering only conversational services.
- ♦ It is prohibited from making a call to `tpforward`().

Both 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 the connection.
- ♦ They use `tpsend`() and `tprecv`() calls to send and receive data in conversations.

Conversational Mode

In the conversational mode of communication, a half-duplex connection is established between the client (or initiator) and a server. Control of the connection can be passed back and forth between the initiator and the subordinate server. At any point in the conversation, the process that has control can send messages; the process that does not have control can only receive. The connection remains up until an event occurs that tears it down. One event, `TPEV_SENDONLY`, notifies the receiving program that control of the connection has been passed to it and it can successfully call `tpsend`(). Other events are notifications that something significant has occurred; they have the result of either bringing the conversation to a normal conclusion or precipitating a disorderly disconnection.

The Connection Descriptor

A connection descriptor, `cd`, is returned 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 have more than one conversation active simultaneously. The maximum number is 64.

Buffer Management

Data is passed in typed buffers just as in request/response mode. The buffer types must be recognized by the application; they must be allocated with ATMI functions as described in Chapter 2, “Writing Client Programs,” and in `tpalloc`(3c) in the *BEA TUXEDO Reference Manual*.

Joining an Application

Conversational clients must join the application via a call to `tpinit()` prior to attempting to establish a connection to a service. The procedure for joining the application is described in Chapter 2, "Writing Client Programs."

Establishing a Connection

`tpconnect`() is the ATMI function used to set up a conversation. The syntax is:

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

name must point to the name of a service posted in the bulletin board by a conversation server. If *name* is not a pointer to a conversational service, the call fails with a `-1` and `tperrno` is set to the error code `TPENOENT`. If the calling program has already reached the maximum number of active connections allowed, the call will fail with the error code `TPELIMIT`.

Data can be sent at the same time the connection is being established by having *data* point to a buffer previously allocated by `tpalloc`(). The type and subtype of the buffer pointed to by *`data`* must be a type recognized by the service being called. If no data is being sent, *`data`* can be set to `NULL`. *`len`* is used to specify how much of the buffer to send. If the buffer is self-defining (for example, an FML buffer), *`len`* can be set to `0`. 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. So far, this should sound a lot like what happens when a request/response service is invoked, because it is. Differences begin to appear when we consider options for the *`flags`* argument.

Values for the flags Argument: tpconnect()

As with other ATMI functions, the behavior of the called program can be controlled by values of the *`flags`* argument of `tpconnect`(). Four of the values are identical to their use in `tpcall`() and are described in “Values for the flags Argument: tpcall()” in Chapter 2, “Writing Client Programs.” They are:

```
TPNOTRAN    TPNOBLOCK
TPNOTIME    TPSIGRSTRT
```

New valid *`flags`* options are:

`TPSENDONLY`

The calling program retains control of the connection, and the called service is permitted only to receive. The called service learns of this through the `flags` member of its `TPSVCINFO` structure; `TPSVCINFO->flags == TPRECVONLY`. `TPSENDONLY` and `TPRECVONLY` are mutually exclusive; one or the other must be specified.

`TPRECVONLY`

Control of the connection is being passed to the called service, and the called service can only send. The called service learns of this through the `flags` member of its `TPSVCINFO` structure; `TPSVCINFO->flags == TPSENDONLY`. `TPSENDONLY` and `TPRECVONLY` are mutually exclusive; one or the other must be specified.

As mentioned above, on successful completion `tpconnect`() returns a connection descriptor that is used in all subsequent calls of the conversation. Your call to `tpconnect`() should be coded something like that shown in Listing 4-1.

Listing 4-1 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

After the conversational connection is set up, communication between the client (or initiator) and the service is accomplished with send/receive calls. The connection is half-duplex. That means communication can be in only one direction at a time. The process that has control of the connection can send; the process that does not have control can receive. Initially, control is decided by the originator and is specified by the `TPSENDONLY` or `TPRECVONLY` flag value of the `tpconnect()` call; `TPSENDONLY` means control is retained by the originator, `TPRECVONLY` means control is given to the called service. After `tpconnect()` returns successfully, data is sent across the open connection with the `tpsend()` function.

The syntax of `tpsend()` is:

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

cd is the connection descriptor returned by `tpconnect()` that identifies the connection over which to send the data. **data* and *len* are, respectively, a pointer to a buffer created by `tpalloc()`, and the length of the data to be sent. The same rules apply to *data* and *len* that have been outlined earlier: The buffer must be of a type recognized by the program that receives it and length can be `0` if the buffer is self-defining. There is no requirement that data be sent. If the data pointer is `NULL`, *len* is ignored.

Values for the flags Argument: tpsend()

There are four valid values for the *flags* argument of tpsend(). Three of them:

TPNOBLOCK
TPNOTIME
TPSIGRSTRT

have the same meaning described in Chapter 2 (in "Values for the flags Argument: tpcall()" section). The fourth value is like one that is used in tpconnect(), but has added significance in this function.

TPRECVONLY
: Signals the intent of the calling program to issue no more tpsend() calls at the moment and to pass control of the connection over to the other side of the connection. When the called program receives the data, it also receives a TPEV_SENDONLY event at the address pointed to by *revent*.

It is not a requirement that control be passed each time the tpsend() call is made. The process authorized to make tpsend() calls on the connection can make as many calls as necessary before turning over control of the connection. In fact, the logic of the conversational program may be such that one side of the conversation retains control of the connection throughout the life of the conversation.

Listing 4-2 shows tpsend() used in a code fragment.

Listing 4-2 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

The function used to receive data sent over an open connection is `tprecv`(). The syntax is:

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

`cd` is the connection descriptor. If the function is being issued from a subordinate program (that is, not the originator of the connection), *`cd`* is in the `TPSVCINFO` structure for the program. If `tprecv`() is being issued by the originator, *`cd`* is the descriptor returned by `tpconnect`(). When the call is made, **`data`* is a pointer to the address of a previously `tpalloc`'d buffer and *`len`* is a pointer to the size of the buffer. *len*, *`data`*, and **`data`* are not allowed to be `NULL`. The call fails and `tperrno` is set to `TPEINVAL`.

Upon successful return, **data* 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`(), it indicates the buffer's new size. If *len* is 0, no data was received.

If an event exists for *cd*, `tprecv`() returns a `-1` and `tperrno` is set to `TPEEVENT`. The event type is returned in revent. With events `TPESVCSUCC`, `TPESVCFAIL`, and `TPESENDONLY`, data can be received. These three events are all normal completions of the `tprecv`() call, so it is not correct to assume the `-1` return value means the call has failed. A more complete discussion of events can be found in "Events and Their Significance" later in this chapter.

Values for the flags Argument: tprecv()

tprecv() has four valid flags. Three of them are described in Chapter 2 (in the “Values for the flags Argument: tpcall()” section). They are:

- TPNOCHANGE
- TPNOTIME
- TPSIGRSTRT

The fourth valid flag value is TPNOBLOCK.

When the flag is set, tprecv() does not wait for data to arrive. If data is available, fine; tprecv() gets the data and returns. If data is not available, the call fails and tperrno is set to TPEBLOCK. When the flag is not set, tprecv() waits and does not return until data arrives or a timeout occurs.

Listing 4-3 shows a fragment of code using tprecv().

Listing 4-3 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);
}
```

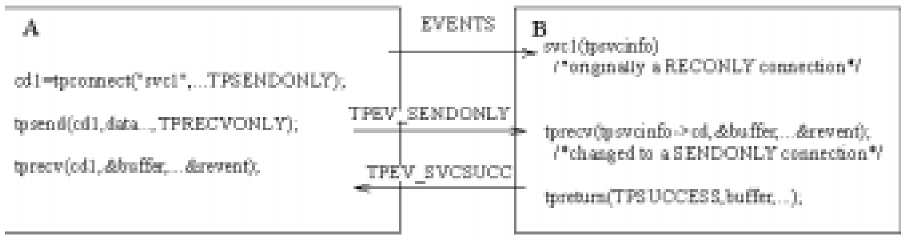
Ending a Conversation

There are three ways in which the connection can be taken down in an orderly fashion and the conversation ended normally. Figure 4-1 and Figure 4-2 show two scenarios that help to illustrate how conversations are ended where global transactions are not involved. The third approach of ending a conversation where a transaction is involved is shown in Chapter 5, “Global Transactions in BEA TUXEDO System.”

Subordinate Calls tpreturn()

Figure 4-1 shows a simple A to B conversation. The connection is set up initially with a call to `tpconnect()` with the `TPSENDONLY` flag set. In due course, A turns control of the connection over to B by calling `tpsend()` with the `TPRECVONLY` flag set. This generates a `TPEV_SENDONLY` event. The next call by B to `tprecv()` returns a `-1`, `tperrno` is set to `TPEEVENT`, and revent shows the event `TPEV_SENDONLY`. B knows from the `TPEV_SENDONLY` event that it now controls the connection. Subsequently, B calls `tpreturn()` with `rval` set to `TPSUCCESS`. This generates a `TPEV_SVCSUCC` event for A. The call to `tpreturn()` also brings down the connection. When A calls `tprecv()` and learns of the event, it recognizes that the conversation has been terminated. Data can be received on this call to `tprecv()` even if the event is `TPEV_SVCFAIL`. In this illustration, A can be either a client or a server, B can only be a server.

Figure 4-1 Simple SENDONLY Connection and Return



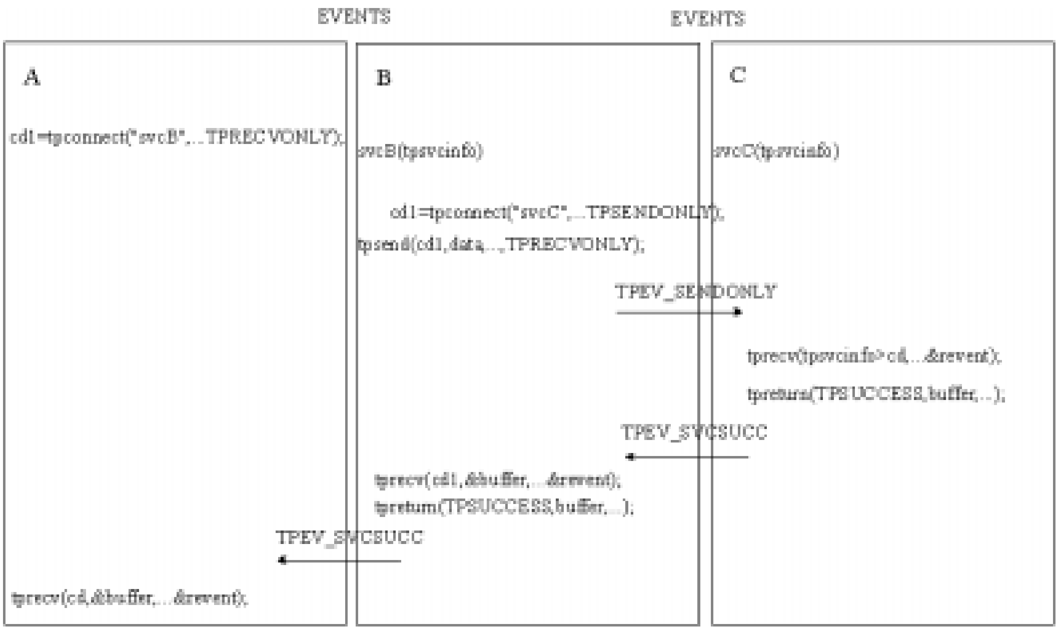
```
cd1=tpconnect("svc1",...TPSENDONLY);
tpsend(cd1,data...,TPRECVONLY);
tprecv(cd1,&buffer,...&revent);
```

```
svc1(tpsvcinfo)
  /*originally a RECONLY connection*/
tprecv(tpsvcinfo->cd,&buffer,...&revent);
  /*changed to a SENDONLY connection*/
tpreturn(TPSUCCESS,buffer,...);
```

Hierarchy of Connections and tpreturn()

Figure 4-2 shows a hierarchy of connections. The scenario applies to a service in a conversation, B, that has initiated a connection to a second service, 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 will fail, a `TPEV_SVCERR` event will be posted on all open connections, and the connections will be closed in a disorderly manner. The proper sequence is for B to call `tpsend`() with the `TPRECVONLY` flag set on the connection to C, turning control of the B-C connection over to C. C can then call `tpreturn`() with *`rval`* set to `TPSUCCESS`, `TPFAIL` or `TPEXIT`, as appropriate. B can then call `tpreturn`(), posting an event (either `TPEV_SVCSUCC` or `TPEV_SVCFAIL`) for A. Both connections are terminated normally.

Figure 4-2 Connection Hierarchy


EVENTS
EVENTS
A
B
C
cd1=tpconnect("svcB",...TPRECVONLY);
svcB(tpsvcinfo)
svcC(tpsvcinfo)
cd1=tpconnect("svcC",...TPSENDONLY);
tpsend(cd1,data,...,TPRECVONLY);
TPEV_SENDONLY
tprecv(tpsvcinfo->cd,...&revent);
tpreturn(TPSUCCESS,buffer,...);
TPEV_SVCSUCC
tprecv(cd1,&buffer,...&revent);
tpreturn(TPSUCCESS,buffer,...);
TPEV_SVCSUCC
tprecv(cd,&buffer,...&revent);

Ending a Conversation: Summary

It is an error to end a conversation with connections still open. Either `tpcommit`() or `tpreturn`() will fail in a disorderly manner.

To summarize the ways a conversation can be ended in an orderly manner:

- ♦ If the connection originated in a server, the originator turns over control of the connection to the called process. That process can then call `tpreturn`(). This is illustrated in Figure 4-1 above.
- ♦ A subordinate process can call `tpreturn`(). The subordinate must have control of the connection and must make the call to `tpreturn`() before the originator does. This is illustrated in Figure 4-2 above.

In each case, the subordinate has control and calls `tpreturn`().

Events and Their Significance

There are five events recognized in conversational communication. All five can be posted for `tprecv`(), three of the five can be posted for `tpsend`(). Table 4-1 summarizes them.

Table 4-1 Conversational Communication Events

Event	Rec'By	Meaning
`TPEV_SENDONLY`	`tprecv`()	control of the connection has been passed; this process can now call `tpsend`()
`TPEV_DISCONIMM`	`tpsend`() `tprecv`() `tpreturn`()	a disorderly disconnect; the connection has been torn down; no further communication is possible; posted by `tpdiscon`() in the originator of the connection, and posted to all open connections when `tpreturn` is called while connections to subordinate services remain open. All connections are closed in a disorderly fashion. If a transaction exists, it is aborted.

Table 4-1 Conversational Communication Events

Event	Rec'By	Meaning
TPEV_SVCERR	tpsend()	received by the originator of the connection, usually indicates the subordinate program has issued a tpreturn without having control of the connection
	tprecv()	received by the originator of the connection, indicates the subordinate program has 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, indicates the subordinate program has 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, indicates the subordinate service finished unsuccessfully (tpreturn was called with TPFAIL or TPEXIT)
TPEV_SVCSUCC	tprecv()	received by the originator of the connection, indicates the subordinate service finished successfully, that is, called tpreturn() with TPSUCCESS

Disorderly Disconnection

The tpdiscon() function has an innocent sound to it, as though it was the logical opposite of tpconnect(), but it is really the equivalent of pulling the plug on the connection. It can be called only by the initiator of a conversation.

The syntax is simple:

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

cd is the connection descriptor returned by tpconnect().

`tpdiscon`() generates a `TPEV_DISCONIMM` event for the service at the other end of the connection, and the *`cd`* is no longer valid. If a transaction is in progress, it is aborted. Data may be lost. If `tpdiscon`() is called from a service that was not the originator of the connection identified by *`cd`*, it fails with an error code of `TPEBADDESC`.

The preferred way of bringing down a connection is for the subordinate to call `tpreturn`().

Request/Response Calls and Conversations

There is nothing that prevents a conversational service from making request/response calls if it needs to communicate with another service. In the examples of connection hierarchies shown in Figure 4-2 above, the calls from B to C could have been made with `tpcall`() or `tpacall`() instead of `tpconnect`(). Remember, however, that conversational services are not permitted to make calls to `tpforward`().

Configuration Parameters

There are some parameters in the configuration file that pertain only to conversational processing. As noted in Chapter 1 (in the "Configuration File" section), the BEA TUXEDO system administrator normally is responsible for setting up the production version of the configuration file for the application, but you may need to set some parameters in your own development configuration.

Here are the parameters you need to know about:

`MAXCONV`
: sets the maximum number of simultaneous conversations for a single machine. The range is from 0 to 32,767. The default is 10 when conversational servers are specified. The parameter can be specified in the `RESOURCES` section for all machines in the configuration and can be overridden in the `MACHINES` section for each machine. It is quite probable that for an application under development the default is adequate.

`CONV = { Y/N }`
: is a parameter in the `SERVERS` section. Connections can only be made to servers that have this value set to `Y`. If it is set to `N` or left unspecified, a `tpconnect`() call to a service of the server will fail.

`MIN` and `MAX`

: are parameters in the `SERVERS` section that specify 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 initially. When a `tpconnect`() call is made to a service offered by that server, the system starts up a second copy. As each copy is called a new one is spawned up to a limit of 10.

`MAXSERVERS`

: specifies the high-water mark for all servers of the configuration. This figure needs to take into account the `MAX` values for all conversational servers. You probably will not need to worry about this for an application under development, but it could be something that needs attention when the application reaches the production stage. The parameter is in the `RESOURCES` section.

Building Conversational Clients and Servers

The utilities described in Chapters 2 and 3, `buildclient`(1) and `buildserver`(1), are used for building conversational clients and servers.

Conversational servers must be built only with conversational services; that is, mixing of conversational services and request/response services in the same server is not allowed.

Conversational services and request/response services can not use the same name.

CHAPTER

5 Global Transactions in BEA TUXEDO System

Introduction

The purpose of this chapter is to explain the concept of global transactions and how to define and manage them in your application using the ATMI primitives for transaction management.

A global transaction is a transaction that allows work involving more than one resource manager and spanning more than one physical site to be treated as one logical unit. The `tpbegin`() function allows you explicitly to start a transaction. The process that calls `tpbegin`() is the initiator of the transaction and must complete it by calling `tpcommit`() or `tpabort`(). 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 the participants. They may affect the outcome of the transaction by the value they return when they invoke the `tpreturn`() function. A process can determine if it is currently working on behalf of a transaction by calling the `tpgetlev`() function. The rest of this chapter will explain these functions in detail.

What Is a Global Transaction?

Before we get into how you can write applications that define and manage global transactions, this section gives you some idea as to what is meant by a transaction that is under the control of a transaction monitor.

The BEA TUXEDO system manages global transactions. As already indicated, a global transaction is one that can execute in more than one server, accessing data from more than one resource manager. A global transaction may be composed of several local transactions, each accessing a single resource manager. A local transaction accesses a single database or file and is controlled by the resource manager responsible for performing concurrency control and atomicity of updates at that distinct database. A given local transaction may be either successful or unsuccessful in completing its access.

A global transaction is always treated as a specific sequence of operations that is characterized by the four properties of atomicity, consistency, isolation, and durability. That is, it is a logical unit of work in which:

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

The BEA TUXEDO system is responsible for managing the status of the global transaction and making the decision as to whether or not a global transaction should be committed or rolled back. Global transactions are explicitly defined and controlled by the ATMI function primitives that can be found on their respective reference pages in the *BEA TUXEDO Reference Manual* and are the topic of this chapter. More specifically, the ATMI functions enable the application programmer to begin and terminate transactions and to test if a client or service routine is currently in a transaction.

ATMI Transaction Primitives

The ATMI primitives `tpbegin()`, `tpcommit()`, and `tpabort()` are used to explicitly begin and end a transaction. The initiator of a transaction uses `tpbegin()` to mark its beginning. After specifying the operations (service requests) to be applied to the resource as part of this transaction, the initiator can then call either `tpcommit()` or `tpabort()` to mark its completion. The calls to initiate and terminate a transaction delineate the operations within the transaction. If the transaction is completed with a call to `tpcommit()`, the changes made as a result of the transaction are applied to the resource and become permanent. `tpabort()` causes the resource to be in the consistent state at the start of the transaction. That is, any changes made to the resource are rolled back. Any of the participants of a transaction can cause the global transaction to fail by communicating their local failure to the initiator through the `tpreturn()` function. A two-phase commit protocol is used by BEA TUXEDO to coordinate the commitment, rollback, and recovery of global transactions. This protocol will be further discussed later in the chapter.

When the `tpgetlev()` function is invoked, it returns a 1 or a 0 that indicates if the caller is within a transaction (1) or not (0).

Explicitly Defining a Global Transaction

Global transactions can be defined in either client or server processes. To explicitly define a global transaction, call the `tpbegin()` function. Follow it by the program statements that are to be in transaction mode. Terminate the statements by a call to `tpcommit()` or `tpabort()`.

The three functions have the following syntax:

```
int
tpbegin(timeout, flags)        /* Begin transaction  */
unsigned long timeout;
long flags;

int
tpcommit(flags)             /* Commit current transaction  */
long flags;

int
tpabort(flags)              /* Abort current transaction  */
long flags;
```

A high-level view of defining a transaction is shown in Listing 5-1.

Listing 5-1 Delineating a Transaction

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

The process that makes the call to `tpbegin()`, the initiator, must also be the one that terminates it by invoking either `tpcommit()` or `tpabort()`. There is no limit to the number of sequential transactions that a process may define using these functions. Any process may call `tpbegin()` except if it is already in transaction mode. If `tpbegin()` is called in transaction mode, the call will fail because of an error in protocol and `tperrno` will be set to `TPEPROTO`. If the process is in transaction mode, the transaction is unaffected by the failure.

Any service subroutines that are called within the transaction delimiters of `tpbegin()` and `tpcommit()/tpabort()` become part of the current transaction. However, if `tpcall()` or `tpacall()` have the *`flags`* parameter explicitly set to `TPNOTRAN`, the operations performed by the called service do not become part of that transaction. This in effect means that the calling process is not inviting the called service to be a participant in the current transaction. As a result, any services performed by the called process will not be affected by the outcome of the current transaction. It should be noted here that a call made with `TPNOTRAN` set that is directed to a service in an XA-compliant server group may produce unexpected results. See the discussion under "Implicitly Defining a Global Transaction" later in this chapter.

Starting the Transaction

The transaction is started by a call to `tpbegin()`. `tpbegin()` takes two parameters, only one of which is used at the present time. *`timeout`* specifies the amount of time in seconds a transaction has before timing out; *`flags`* is currently undefined and must be set to 0. The value of the *`timeout`* parameter indicates the least amount of time in seconds that a transaction should be given before timing out. If `0` is specified for this parameter, the transaction is given the maximum number of seconds allowed by the system before timing out (that is, the time-out value will equal the maximum value for an unsigned long as defined by the system).

Note: The use of 0 or unrealistically large values for the *`timeout`* parameter delays system detection and reporting of errors. A time-out value is used to ensure response to service requests within a reasonable time, and to terminate transactions that have encountered problems such as network failures prior to commit. For a transaction in which a person is waiting for a response, a small value, often less than 30 seconds, is best. In a production system, the time-out value should be 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.

If a transaction times out, it is aborted. You can determine if a transaction has timed out by testing the value of `tperrno` as illustrated in Listing 5-2. Note that if the transaction timed out and it goes untested, a call to `tpcommit()` will still cause the transaction to be aborted. In this case, `tpcommit()` fails and returns `TPEABORT` tpcommit() in `tperrno` and the transaction is implicitly aborted.

The value assigned to the *timeout* parameter should be consistent with the `SCANUNIT` parameter set by the BEA TUXEDO system administrator in the configuration file. The system parameter specifies the frequency with which timed-out transactions and blocked calls are looked for. Its value represents an interval of time between periodic scans to find old transactions and timed out blocking calls within service requests. The *`timeout`* parameter should be set to a value that is greater than the scanning unit. If the time-out value were smaller, there would be some discrepancy between the time the transaction timed out and its discovery. The default value for `SCANUNIT` is 10 seconds. The value you give to *`timeout`* may need to be coordinated with your system administrator to be sure it makes sense with regard to the system parameters.

Listing 5-2 illustrates the starting of a transaction with the time-out value set to 30 seconds followed by a check to see if a timeout occurred.

Listing 5-2 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 that a transaction is still subject to timing out even when a process calls on another with the TPNOTRAN communication flag set. This will be further discussed in Chapter 7, "Error Management."

The example in Listing 5-3 is excerpted from the audit.c client program of the banking application.

Listing 5-3 Defining a Transaction

```
#include <stdio.h>          /* UNIX */
#include <string.h>         /* UNIX */
#include <atmi.h>           /* TUXEDO */
#include <Uunix.h>          /* TUXEDO */
#include <userlog.h>        /* TUXEDO */
#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);
}
```

Terminating the Transaction

As already indicated, a transaction is terminated by a call to either `tpcommit()` or `tpabort()`. Both have the *flags* parameter defined. However, while *flags* is not currently used, you must set this parameter to zero to ensure compatibility with future releases. When `tpcommit()` returns successfully, all changes to the resource as a result of the current transaction become permanent. `tpabort()` is called to indicate an abnormal condition and explicitly aborts the transaction and invalidates the call descriptors of any outstanding transactional replies (calls done with the TPNOTRAN flag will not be invalidated). None of the changes that were produced as a result of the transaction are applied to the resource. For `tpcommit()` to succeed, the following two 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 transaction replies (calls made without the TPNOTRAN flag) outstanding.

If either condition is not true, the call fails and `tperrno` is set to `TPEPROTO` indicating an error in protocol. If a participant calls `tpcommit()` or `tpabort()`, the transaction is unaffected. 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.

tpcommit Initiates the Two-Phase Commit

When `tpcommit()` is called, it initiates the two-phase commit protocol mentioned earlier. This protocol, as the name suggests, has two parts. In the first, each participating resource manager indicates a readiness to commit. In the second, the initiator gives permission to commit. The process that calls `tpcommit()` must be the initiator of the transaction. As the initiator, this process starts the commit processing in which the participants (the other server processes that took part in the transaction) communicate their success or failure. This can be made known to the initiator by `tpreturn()` through the *rval* parameter that can be set to either `TPSUCCESS` or `TPFAIL`. If `TPFAIL` has been returned, `tpcommit()` fails, `tperrno` is set to `TPEABORT`, and the transaction is implicitly aborted. All the work that is performed by every process that participated in that transaction is undone. More will be said about the transaction role of `tpreturn()` and `TPFORWAR()` in Chapter 7, “Error Management.”

Setting When tpcommit() Should Return

When more than one machine is involved in a transaction, the application can elect to specify that `tpcommit()` should return successfully when all participants have indicated a readiness to commit; that is, when phase 1 of the two-phase commit has been logged as complete by all participants. The alternative choice is to have `tpcommit()` wait until all participants have finished phase 2 of the two-phase commit. The `CMTRET` parameter in the RESOURCES section of `UBBCONFIG` can be set to either `LOGGED` or `COMPLETE` to control this characteristic. The function `TPSCMT()` can be called with its *`flags`* argument set to either `TP_CMT_LOGGED` or `TP_CMT_COMPLETE` to override the setting in the configuration file.

The idea behind this option is that most of the time when all participants in a global transaction have logged successful completion of phase 1, they will not fail to complete phase 2. By setting `TP_COMMIT_CONTROL` to `LOGGED` you allow slightly faster return of calls to `tpcommit()`, but you run the slight risk that a participant (probably on a remote node) 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 would probably prefer to allow for the time required for all participants fully to complete the two-phase commit process. If you are counting beans, you may prefer to have the application run as fast as possible even knowing you may be a few beans off over a period of time.

Testing for Participant Errors

A client making a synchronous call to the fictitious `REPORT` service (line 18) is shown in Listing 5-4. It demonstrates testing for errors that can be returned on a communication call that indicate participant failure (lines 19-34).

Listing 5-4 Testing for Participant Success or Failure

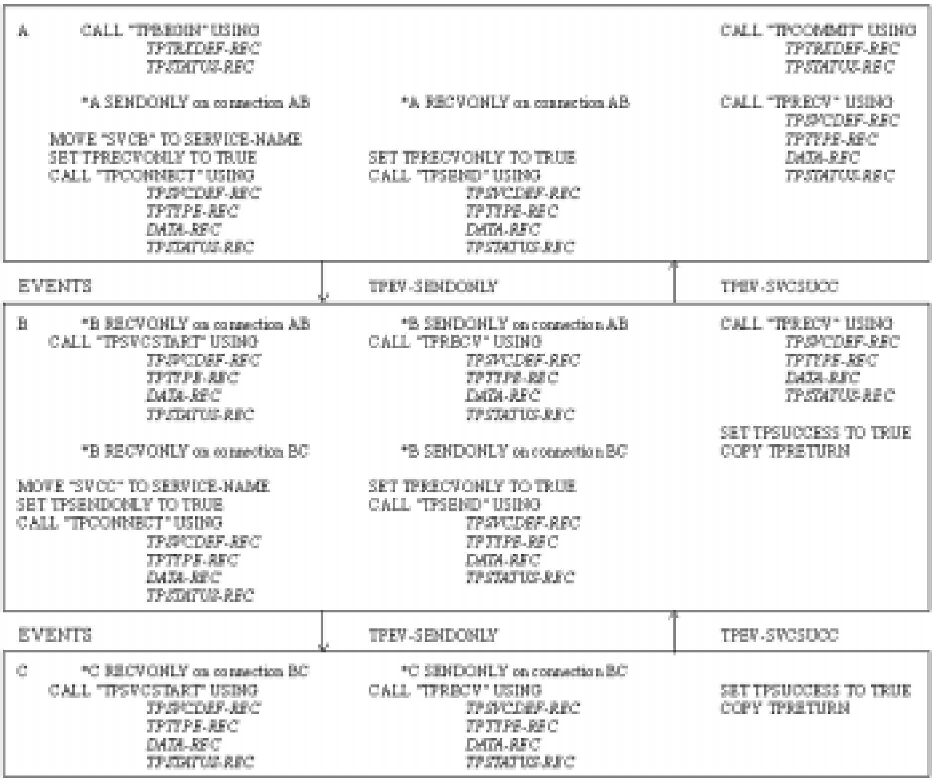
```
#include <stdio.h>
#include "atmi.h"

main()
{
char *sbuf, *rbuf;
long slen, rlen;
if (tpinit((TPINIT *) NULL) == -1)
    error message, exit program;
if (tpbegin(30, 0) == -1)
    error message, tpterm, exit program;
if ((sbuf=tpalloc("STRING", NULL, 100)) == NULL)
    error message, tpabort, tpterm, exit program;
if ((rbuf=tpalloc("STRING", NULL, 2000)) == NULL)
    error message, tpfree sbuf, tpabort, tpterm, exit program;
(void)strcpy(sbuf, "REPORT=accrcv DBNAME=accounts");
slen=strlen(sbuf);
if (tpcall("REPORT", sbuf, slen, &rbuf, &rlen, 0) == -1) {
    switch(tperrno) {
    case TPESVCERR:
        fprintf(stderr,
            "REPORT service's tpreturn encountered problems\n");
        break;
    case TPESVCFAIL:
        fprintf(stderr,
           "REPORT service TPFAILED with return code of %d\n", tpurcode);
        break;
    case TPEOTYPE:
        fprintf(stderr,
            "REPORT service's reply is not of any known data type\n");
        break;
    }
    if (tpabort(0) == -1){
    check for errors;
    }
}
else
    if (tpcommit(0) == -1)
        fprintf(stderr, "REPORT failed at commit time\n");
tpfree(rbuf);
tpfree(sbuf);
tpterm();
exit(0);
}
```

Committing a Transaction in Conversational Mode

Figure 5-1 shows a conversational connection hierarchy that includes a global transaction. The originator of a connection in transaction mode (process A that called `tpbegin()` followed by `tpconnect()`) can call `tpcommit()` after all services have called `tpreturn().` If a hierarchy of connections exists as it does in Figure 5-1, each subordinate service must call `tpreturn()` when it no longer has replies outstanding. A `TPEV_SVCSUCC` or `TPEV_SVCFAIL` event is sent back up the hierarchy to the process that began the transaction. If all subordinates return successfully, the client (process A) completes the transaction; otherwise the transaction is aborted.

Figure 5-1 Connection Hierarchy: Transaction Mode



```
A       CALL "TPBEGIN" USING                                                   CALL "TPCOMMIT" USING
                   TPTRXDEF-REC                                                          TPTRXDEF-REC
                   TPSTATUS-REC                                                          TPSTATUS-REC

        *A SENDONLY on connection AB          *A RECVONLY on connection AB     CALL "TPRECV" USING
                                                                                         TPSVCDEF-REC
    MOVE "SVCB" TO SERVICE-NAME                                                          TPTYPE-REC
    SET TPRECVONLY TO TRUE                SET TPRECVONLY TO TRUE                         DATA-REC
    CALL "TPCONNECT" USING                CALL "TPSEND" USING                            TPSTATUS-REC
                   TPSVCDEF-REC                          TPSVCDEF-REC
                   TPTYPE-REC                            TPTYPE-REC
                   DATA-REC                              DATA-REC
                   TPSTATUS-REC                          TPSTATUS-REC

EVENTS                                    TPEV-SENDONLY                        TPEV-SVCSUCC

B       *B RECVONLY on connection AB          *B SENDONLY on connection AB     CALL "TPRECV" USING
    CALL "TPSVCSTART" USING               CALL "TPRECV" USING                            TPSVCDEF-REC
                   TPSVCDEF-REC                          TPSVCDEF-REC                    TPTYPE-REC
                   TPTYPE-REC                            TPTYPE-REC                      DATA-REC
                   DATA-REC                              DATA-REC                        TPSTATUS-REC
                   TPSTATUS-REC                          TPSTATUS-REC
                                                                               SET TPSUCCESS TO TRUE
        *B RECVONLY on connection BC          *B SENDONLY on connection BC     COPY TPRETURN

MOVE "SVCC" TO SERVICE-NAME               SET TPRECVONLY TO TRUE
SET TPSENDONLY TO TRUE                    CALL "TPSEND" USING
CALL "TPCONNECT" USING                                   TPSVCDEF-REC
                   TPSVCDEF-REC                          TPTYPE-REC
                   TPTYPE-REC                            DATA-REC
                   DATA-REC                              TPSTATUS-REC
                   TPSTATUS-REC

EVENTS                                    TPEV-SENDONLY                        TPEV-SVCSUCC

C       *C RECVONLY on connection BC          *C SENDONLY on connection BC
    CALL "TPSVCSTART" USING               CALL "TPRECV" USING                  SET TPSUCCESS TO TRUE
                   TPSVCDEF-REC                          TPSVCDEF-REC          COPY TPRETURN
                   TPTYPE-REC                            TPTYPE-REC
                   DATA-REC                              DATA-REC
                   TPSTATUS-REC                          TPSTATUS-REC
```

Implicitly Defining a Global Transaction

Besides using the ATMI primitives explicitly to start and end a transaction, it is possible for a global transaction to be started in two other ways.

In a Client Process

The BEA TUXEDO system provides a predefined client program that is part of its Data Entry System. The program is called `mio` and is the form handler for the system. Through the form specification language, `UFORM`, it is possible to cause a transaction to be started in `mio` whenever a service request is generated. The `TRANMODE` parameter in the `FORM` statement of the `UFORM` specification language allows you to have the client process initiate a transaction for each service. In this case, the programmer does not make explicit calls to `tpbegin()` or `tpcommit()/tpabort()` to delineate the transaction. If the `TRANMODE` parameter of the `FORM` statement has been set to `TRAN`, `mio` automatically starts and ends the transaction. The application logic to decide whether to commit or roll back the transaction is built into `mio`.

In a Service Routine

In another special case, a service routine can be placed in transaction mode through the system parameter, `AUTOTRAN`, in the configuration file. If `AUTOTRAN` is set to `Y`, a transaction is automatically started in the service subroutine when a request message is received from another process. Let’s look at some variations on this theme.

- If a process is not in transaction mode and calls on the services of another process, the system parameter is consulted for the called service, and if it is set to start a transaction, one will be initiated with the call.
- If a process is in transaction mode and calls on the services of another process, it places the called process in transaction mode through the “rule of propagation” and the system parameter is not consulted.
- If a process is in transaction mode and calls on the services of another process, but the caller has its *`flags`* parameter set to `TPNOTRAN`, the services performed by the called process are not part of the current transaction (suppresses propagation rule). The system parameter will be consulted and
  - if it is set to `N` (or not set), the called process is not placed in transaction mode.
  - if it is set to `Y`, the service is placed in transaction mode, but this is a new transaction.

Because a service can automatically be placed in transaction mode, it is possible for the call to be made with the communication flag set to `TPNOTRAN` and the `flags` member of the service information structure to return `TPTRAN` when queried. For example, if the call is made with the communication flags set to `TPNOTRAN|TPNOREPLY` and the service automatically starts a transaction when called, the `flags` member of the information structure will be set to `TPTRAN|TPNOREPLY`.

What a Service in an XA-Compliant Server Group Expects

A service that is part of an XA-compliant server group is generally written to perform some operation via the group's resource manager, which automatically opened the associated database when the application was booted. In the normal case, the service expects to do its work within a transaction. If a service like this is called with the caller's communication flags set to `TPNOTRAN`, the results of the ensuing database operation may be a little strange.

The solution is to write your 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=Y`. Another precaution is to test early in the service code to see what the transaction level is.

Testing Whether a Transaction Has Begun

In order correctly to interpret the error messages that can occur, it is important to know if a process is in transaction mode or not. It is an error for a process that is already in transaction mode to make a call to `tpbegin()`. `tpbegin()` will fail and set `tperrno` to `TPEPROTO` to indicate that the function was invoked while the caller was already in a transaction. However, the transaction will not be affected.

It might be helpful to think of transaction mode as something that is propagated unless specifically suppressed. When one process in transaction mode calls on the services of another process, that process acquires the same "condition." If `mio` has been placed in transaction mode through the form specification language, all the service subroutines it calls upon may be placed in transaction mode.

Service subroutines can be written so that they test to see if they are already in transaction mode before invoking `tpbegin()`. Testing transaction level can be done by querying the `flags` member of the service information structure that is passed to the service routine. If its value is set to `TPTRAN`, the service is in transaction mode. Also, this information can be retrieved by calling the `tpgetlev()` function. The syntax of this function is:

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

`tpgetlev()` returns `0` if the caller is not in a transaction and `1` if it is.

Listing 5-5 is a variation of the `OPEN_ACCT` service that shows testing for transaction level using the `tpgetlev()` function (line 12). If the process is not in transaction mode, it starts one (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` (line 17 and line 1).

If the `AUTOTRAN` configuration parameter discussed above is set to `Y`, you avoid the overhead of testing for transaction level and the need of explicitly calling the `tpbegin()` and `tpcommit()/tpabort()` transaction functions. For example, in the fragment shown in Listing 5-5, if `OPEN_ACCT` service is always to be called in transaction mode, the system parameters `AUTOTRAN` and `TRANTIME` can be set in the configuration file, eliminating the need to define the transaction or determine its existence within the programming code (lines 7 and 10-19).

Listing 5-5 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 }
    . . .
```

CHAPTER

6 Using the Event Broker

Introduction

The Event Broker is a BEA TUXEDO subsystem that receives event posting messages, filters them, and distributes them to subscribers. A poster is a BEA TUXEDO system process that detects when an event of interest has occurred and reports (posts) it to the Event Broker. A subscriber is a BEA TUXEDO system process that requests that some notification action be taken when a matching event is posted.

This concept of an "anonymous" broker that receives and distributes messages provides another client-server communication paradigm to BEA TUXEDO. Instead of a one-to-one relationship between a service requester and a service provider, an arbitrary number of posters can post a message buffer for an arbitrary number of subscribers. The posters simply post events, without having to know who receives the information and what is done about it. The subscribers can get whatever information they are interested in from the Event Broker, without having to know who posted it, and they can be notified and take action in a variety of ways.

Typically, Event Broker applications are designed to handle exception events. The application designer has to decide what events in the application need to be monitored. In a banking application, for example, an event might be posted for an unusually large withdrawal transaction; but it would not be particularly useful to post an event for every withdrawal transaction. And not all users would need to subscribe to that event; perhaps just the branch manager would need to be notified.

Following this introduction to the Event Broker's features and some guidelines on its use, this chapter explains how to post events, how to subscribe to events, and gives some examples.

Notification Actions

When an event is posted, the Event Broker may be configured to invoke one or more of these notification actions for clients or servers who have subscribed:

- Unsolicited notification message—Clients may receive event notification messages in their unsolicited message handling routine, just as if they were sent by `tpnotify`(3c).
- Service call—Servers may receive event notification messages as input to service routines, just as if they were sent by `tpacall`(3c).
- Reliable queue—Event notification messages may be stored in a BEA TUXEDO system reliable queue, using `tpenqueue`(3c). The event notification buffers are stored until requested. A BEA TUXEDO system client or server process may call `tpdequeue`(3c) to retrieve these notification buffers, or alternately `TMQFORWARD`(5) may be configured to automatically dispatch a BEA TUXEDO system service routine.

Further information on the use of these notification actions is given later in this chapter.

In addition, the following notification actions may be configured by the system administrator only, using the BEA TUXEDO system administrative API to create an `EVENT_MIB`(5) entry:

- Invoke a system command.
- Write a message to the system's log file on disk.

For information on the `EVENT_MIB`(5), see the *BEA TUXEDO Reference Manual*.

User-Defined and System-Defined Events

The Event Broker is used to monitor and report application-defined events such as a large cash withdrawal as mentioned earlier for a banking application. BEA TUXEDO itself detects and posts certain pre-defined events related to system warnings and failures. This is done by the Event Monitor feature, which is a part of the general-purpose Event Broker communication mechanism. For example, system-generated events report on configuration changes, state changes, connection failures, and machine partitioning. A list of the system-generated events detected by the Event Monitor is given in the `EVENTS`(5) reference page in the *BEA TUXEDO Reference Manual*.

Note that a leading dot (".") in the event name is used to distinguish system-generated events from application-defined events.

System-generated events are defined in advance by BEA TUXEDO system code and as such do not have to be posted. System-generated events can be subscribed to by clients and servers just as application-defined events are. However, just as application-defined events should be used for exceptional conditions, subscriptions to system-generated events should be used mainly by system administrators, not by every client in the application.

When incorporating the features offered by the Event Broker/Event Monitor into your application, remember that it is not intended to be a mechanism for high volumes of postings going to many subscribers. Don't try to post an event for every activity that occurs, and don't think that everyone needs to subscribe. In an overload condition, system performance could be affected and notifications could be dropped. To minimize that possibility, the system administrator should ensure that the UNIX IPC resources are carefully tuned as explained in the *BEA TUXEDO Installation Guide*.

Event Broker/Event Monitor Servers

The Event Broker server is `TMUSREVT`(5). This is a BEA TUXEDO system-provided server that processes event report message buffers and acts as an Event Broker to filter and distribute them. The BEA TUXEDO system administrator must boot one or more of these servers to activate event brokering.

`TMSYSEVT`(5) is the BEA TUXEDO system-provided server that acts as an Event Broker for system-generated events. `TMSYSEVT` and `TMUSREVT` are similar, but separate servers have been provided to allow the system administrator to have a different replication strategy for processing system event notifications. This is discussed in *Administering the BEA TUXEDO System*.

Programming Interface

Event Broker programming interfaces are available to all BEA TUXEDO system server and client processes, including Workstation and COBOL. Basic operations are as follows:

- A client or server *posts* a buffer to an application-defined event name.
- This buffer is then transmitted to any number of processes that have *subscribed* to the event.
- Subscribers may be notified in a variety of ways, as discussed above, and events may be filtered. Notification and filtering are configured through the programming interface described in this chapter as well as through the BEA TUXEDO system administrative API.

Posting Events

To post an event, a BEA TUXEDO client or server calls `tppost`(3c). The input is an event name, buffer pointer, buffer length, and flags.

The syntax of the `tppost`() function is:

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

tppost() Arguments: eventname

The `tppost`() *`eventname`* can contain up to 31 characters plus a null character. The first character cannot be a dot ("."), as this character is reserved as the starting character for BEA TUXEDO system-generated events.

When choosing event names, keep in mind that subscribers can use wild card 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.

tppost() Arguments: data and len

The `tppost`() *`data`* argument must point to a buffer previously allocated by `tpalloc`(3c), and *`len`* should specify the amount of data in the buffer that should be posted with the event. Note that if *`data`* points to a buffer of a type that does not require a length to be specified (for example, an FML fielded buffer), then *`len`* is ignored.

If *`data`* is `NULL`, *`len`* is ignored and the event is posted with no data.

tppost() Arguments: flags

`tppost`() can be used with a number of flags, for example, to determine how transaction timeouts and blocking timeouts are to be handled. For details, see the `tppost`(3c) reference page in the *BEA TUXEDO Reference Manual*.

Example of Event Posting

Listing 6-1 shows an example of event posting taken from the BEA TUXEDO system sample application `bankapp`. This example is part of the `WITHDRAWAL` service. One of the things that the `WITHDRAWAL` service does is check for withdrawals greater than $10,000.00. In this example, an event called `BANK_TLR_WITHDRAWAL` is posted whenever such a withdrawal is made.

Listing 6-1 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));
}
```

Note that this example simply posts the event to the Event Broker: something noteworthy has occurred in the application. Subscription to the event by interested client(s), who can then take action as needed, is done independently.

Subscribing to Events

To subscribe to an event, a BEA TUXEDO system client or server calls `tpsubscribe`(3c). The input is an event name(s), optional filter rules, and flags to specify the notification method. As mentioned earlier in this chapter, several notification methods are available: unsolicited notification message, service call, and reliable queue. (Other notification methods can be configured by the system administrator using the BEA TUXEDO system administrative API.)

The syntax of the `tpsubscribe`() function is:

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

Both system-generated events and application-defined events can be subscribed to with `tpsubscribe`().

For purposes of subscriptions (and for `MIB` updates), service routines executing in a BEA TUXEDO system server process are considered to be trusted code. This is a recent enhancement to BEA TUXEDO system security; it permits some operations that might have been prohibited in earlier versions.

tpsubscribe() Arguments: eventexpr

The event or set of events being subscribed to is named by *`eventexpr`*, a null-terminated string of up to 255 characters containing a regular expression. Regular expressions are of the form specified in `recomp`(3c). For example:

- If *`eventexpr`* is `"\\..*"`, the caller is subscribing to all system-generated events.
- If *`eventexpr`* is `"\\.SysServer.*"`, the caller is subscribing to all system-generated events related to servers.
- If *`eventexpr`* is `"[A-Z].*"`, the caller is subscribing to all user events starting with A-Z.
- If *`eventexpr`* is `".*(ERR|err).*"`, the caller is subscribing to all user events containing either the substring `ERR` or the substring `err` (for example, `account_error` and `ERROR_STATE` events would both qualify).

tpsubscribe() Arguments: filter

The `tpsubscribe()` *`filter`* argument, if present, is a string containing a boolean filter rule that must be evaluated successfully before the Event Broker posts the event. Upon receiving an event to be posted, the Event Broker applies the filter rule, if one exists, to the posted event's data. If the data passes the filter rule, the Event Broker invokes the notification method specified; otherwise, the Event Broker does not invoke the notification method. The caller can subscribe to the same event multiple times with different filter rules.

By using the event filtering capability, subscribers can be more discriminating about the events they are notified of. For example, a poster can post an event for withdrawals greater than $10,000.00, as illustrated above, but a subscriber may want to specify a higher threshold for being notified, such as $50,000.00. Or, a subscriber may want to be notified of large withdrawals only if made by customers with specified IDs.

Filter rules are specific to the typed buffers to which they are applied. See the `tpsubscribe`(3c) reference page in the *BEA TUXEDO Reference Manual* for further information on filter rules.

tpsubscribe() Arguments: ctl

The *`ctl`* argument to `tpsubscribe`() controls how the subscriber is notified of the event.

Notification Via Unsolicited Message

If the subscriber is a BEA TUXEDO system client process and *`ctl`* is `NULL`, then the Event Broker sends an unsolicited message to the subscriber when the event to which it is subscribed is posted. Basic operation is as follows. When an event name is posted that evaluates successfully against *`eventexpr`*, the Event Broker 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 (via `tpsetunsol`(3c)) an unsolicited message handling routine.

Clients receiving event notification via unsolicited messages should remove their subscriptions from the Event Broker's list of active subscriptions before exiting. This is done using the `tpunsubscribe`(3c) function, as shown later in this chapter.

Notification Via Service Call or Reliable Queue

Event notification via service call gives you the ability to program responses to specific conditions in your application and take action without human intervention. An example is given later in this chapter. Event notification via reliable queue ensures that event data will not get lost and also gives 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 to go to service routines or to stable-storage queues, then the *`ctl`* parameter of `tpsubscribe`() must point to a valid `TPEVCTL` structure. This structure contains the following elements:

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

The following is a list of valid bits for the *ctl->flags* element for controlling options for event subscriptions.

`TPEVSERVICE`

When this flag bit is set, event notifications are sent to the BEA TUXEDO system service routine named in *`ctl->name1`*. Basic operation is as follows. When an event name is posted that evaluates successfully against *`eventexpr`*, the Event Broker 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 a service request is sent to *`ctl->name1`* along with any data posted with the event. The service name in *`ctl->name1`* can be any valid BEA TUXEDO system service name and it may or may not be active at the time the subscription is made. Service routines invoked by the Event Broker should return with no reply data (that is, they should call `tpreturn`(3c) with a `NULL` data argument). Any data passed to `tpreturn`() will be dropped.

`TPEVQUEUE`

When this flag bit is set, event notifications are enqueued to the queue space named in *`ctl->name1`* and the queue named in *`ctl->name2`*. Basic operation is as follows. When an event name is posted that evaluates successfully against *`eventexpr`*, the Event Broker 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 Event Broker enqueues a message to the specified queue space/queue name along with any data posted with the event. The queue space and queue name can be any valid BEA TUXEDO system queue space and queue name, either of which may or may not exist at the time the subscription is made. In the TPEVCTL structure, *`ctl->qctl`* can contain options further directing the Event Broker's enqueuing of the posted event. These are the same options that are used for control of the `tpenqueue`(3c) function. For example, the `TPQTOP` option can be used to place a message at the top of the queue. For further information on these options, see the `tpenqueue`(3c) reference page in the *BEA TUXEDO Reference Manual*.

`TPEVSERVICE` and `TPEVQUEUE` are mutually exclusive flags. For information on other flag bits that can be used with either `TPEVSERVICE` or `TPEVQUEUE`, see the `tpsubscribe`(3c) reference page in the *BEA TUXEDO Reference Manual*.

tpsubscribe() Arguments: flags

`tpsubscribe`() can be used with a number of flags, for example, to determine how transaction timeouts and blocking timeouts are to be handled. For details, see the `tpsubscribe`(3c) reference page in the *BEA TUXEDO Reference Manual*.

Example of Event Subscription

Listing 6-2 shows part of a `bankapp` application server that subscribes to `BANK_TLR_.*` events, which would include the `BANK_TLR_WITHDRAWAL` event from the previous example as well as any other event names beginning with `BANK_TLR_`. When a matching event is posted, the subscriber is notified via a service call to a service named `WATCHDOG`.

Listing 6-2 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);
}
```

In the above example, note the use of `tpunsubscribe()` before leaving the application. This removes the event subscription from the Event Broker's list of active subscriptions. For information on the options available for this function, see the `tpunsubscribe`(3c) reference page in the *BEA TUXEDO Reference Manual*.

CHAPTER

7 Error Management

Introduction

The purpose of this chapter is to review the transaction and communication concepts discussed in the preceding chapters with the focus on how to manage and interpret error conditions correctly.

What are the means used by the BEA TUXEDO system to communicate to the application that a function call has failed, allowing the programmer to implement the appropriate logic? What are the various scenarios for determining whether to commit or abort a transaction? What errors are fatal to transactions? How does transaction mode affect the concept of time-out and what are the implications? How does transaction mode affect the roles of the function primitives and how they may be used? What operations are part of one transaction and what are the determining factors? Does the fate of one transaction ever determine the fate of another? What communication rules must be followed between processes within and not within the same transaction? How do global transaction primitives affect the use of local transaction-defining functions (that is, functions used to explicitly mark the beginning and end of a local transaction) that may be part of the Data Manipulation Language (DML) that is native to the resource manager?

Many of these subjects have been touched upon already in earlier chapters. Now let's attempt to bring them together to explain the functionality of the ATMI, showing how the various pieces fit together following consistent rules that create an environment that combines message communication with transaction integrity.

Communicating Errors

The following discussion concerns how the BEA TUXEDO system communicates errors to the application developer. It is couched in terms of categories of errors and whether they are application or system-based. Hopefully, this discussion will give you more insight as to what errors to expect, what effect they have on transactions, and what kind of control you as a programmer have over them.

Throughout the guide, there has been a continual reference to the global variable `tperrno`. In an environment of concurrent processes, this is a key way to inform processes if their function calls have succeeded or not. All the ATMI functions that normally return an integer or pointer, return `-1` or `NULL` on error and set `tperrno` to a value that reveals the nature of the error. In cases where the function does not return to its caller, as in the case of `tpreturn()` or `tpforward()` since they are called to terminate a service routine, the only way to communicate success or failure is through the global variable, `tpurcode`, in the requester.

The global variable `tpurcode` can also be used to communicate user-defined conditions. The value in `tpurcode` is set from the value placed in the *`rcode`* argument of `tpreturn()`. This code is sent regardless of the value of the *`rval`* argument of `tpreturn()` unless an error is encountered by `tpreturn()` or a transaction time-out occurs.

Values of tperrno

The codes returned in `tperrno` represent categories of errors. All the ATMI functions whose failure is reported by the value returned in `tperrno` have the four basic categories of

- protocol errors (`TPEPROTO`)
- BEA TUXEDO system errors (`TPESYSTEM`)
- operating system errors (`TPEOS`)
- errors from invalid arguments (`TPEINVAL`)

Protocol Errors

Protocol errors occur because an ATMI function was called in an incorrect context. Refer to the `intro`(3c) reference page. This type of error usually happens for one of two reasons. Either the ATMI call is being made

- ♦ in the wrong order
- ♦ or by the wrong process.

For example, a client attempting to begin communication before joining the application illustrates an error in protocol because these operations are being attempted in the wrong order.

A transaction participant rather than the initiator calling `tpcommit()` is another protocol error because the participant is the wrong process to be calling `tpcommit()`.

A protocol error is one that is totally correctable at the application level by enforcing the rules of order and propriety associated with the ATMI calls (that is, by making calls in the correct order by the appropriate processes).

Since each ATMI call can return a protocol error, attempt to discover the exact error in the context of the semantics of the specific call and ask the two questions:

- ♦ Is this call being made in the correct order?
- ♦ Is this call being made by the correct process?

BEA TUXEDO System Errors

When BEA TUXEDO system errors occur, messages explaining their exact nature are written to the central event log. The section entitled “The Central Event Log” later in this chapter explains this log in detail. Since these are system errors rather than application errors, the system administrator may be needed to help correct them.

Operating System Errors

Operating system errors indicate that a system call has failed. A numeric value identifying the failed system call is returned in the global variable, `Uunixerr`. Operating system errors are seldom application errors; systems administrators may need to be called on to correct them.

Errors from Invalid Arguments

All of the ATMI functions that take arguments can fail if invalid arguments are passed to them. In the case where the function returns to the caller, the function fails and causes `tperrno` to be set to `TPEINVAL`. In the case of `tpreturn()` or `tpforward()`, if this type of error is discovered while processing the arguments, `tperrno` is set to `TPESVCERR` for the function waiting on the call; that is, either `tpcall()` or `tpgetrply()`. This is an application error and is correctable by the programmer.

Other Possible Error Categories

In addition to the four basic categories just discussed, others include

- errors from lack of entries in system tables or the data structure used to identify buffer types (`TPENOENT`)
- errors from lack of permission to enter the application (`TPEPERM`)
- resource manager errors (`TPERMERR`)
- transaction related errors (`TPETRAN`)
- errors from mismatching of typed buffers (`TPEITYPE` and `TPEOTYPE`)
- errors that apply only to asynchronous communication calls or conversational calls because they involve call descriptors (`TPELIMIT` and `TPEBADDESC`)
- errors that can occur as a result of the communication calls in general (`TPESVCFAIL`, `TPESVCERR`, `TPEBLOCK`, and `TPGOTSIG`)
- transaction and blocking time-out errors (`TPETIME`)
- errors from calling `tpcommit()` when the transaction should have been explicitly aborted (`TPEABORT`)
- errors that signal that a heuristic decision was (or may have been) taken (`TPEHAZARD`, `TPEHEURISTIC`)

No Entry Errors

The no entry type error, `TPENOENT`, has more than one meaning and depends on which function call is returning it. The following table lists the functions and specifies the reason for the failure in each case.

Function	Explanation of TPNOENT Error
`tpalloc()`	The type of buffer asked for is not known to the system. 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 the `tuxtypes`(5) and `typesw`(5) reference pages. On an application level, make sure you have referenced a known type correctly, otherwise see your 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. See your system administrator.

`tpcall()`
`tpacall()`
	The calling process is referencing a service that is not known to the system since there is no entry for it in the bulletin board. On an application level, make sure you have referenced the service correctly, otherwise see your system administrator.

`tpconnect()`	Cannot connect to *name* because it does not exist or is not a conversational service
`tpgprio()`	The calling process is asking for a request priority when no request has been made. The system has no current entry for a request. This is an application error.
`tpunadvertise()`	Cannot unadvertise the service name because it is not currently advertised by the calling process

`tpenqueue()`
`tpdequeue()`
	Cannot access the *qspace* because it is not available (the associated `TMQUEUE`(5) server is not available)

`tppost()`
`tpsubscribe()`
`tpunsubscribe()`
	Cannot access the BEA TUXEDO system event broker

Permission Errors

The only ATMI function that returns this type of error is `tpinit()`. If the calling process does not have the correct permissions to enter the application, this call fails returning `TPEPERM`. Permissions are set in the configuration file and as such the correction of this error is outside of your application. See the BEA TUXEDO system administrator if you encounter this error.

Resource Manager Errors

These errors can occur with calls to `tpopen()` and `tpclose()`, and they return the value of `TPERMERR` in `tperrno`. The meaning of the BEA TUXEDO system error code is intentionally vague in this case so as not to hinder portability. The exact nature of the error must be determined by interrogating the resource manager in its own specific manner. Obviously when this error code is returned for `tpopen()`, it indicates that the problem has to do with a failure on the part of the resource manager to open correctly and for `tpclose()`, to close correctly.

Transaction-Related Errors

When this type of error occurs, `TPETRAN` is returned in `tperrno`. `tpbegin()`, `tpcancel()`, `tpresume()`, `tpconnect()`, `tppost()`, and the `tpcall()`/`tpacall()` functions can return this error code. For `tpbegin()`, it usually means some transient system error occurred when attempting to start the transaction that may clear up with a repeated call.

`tpcancel()` returns this error code when called for a transaction reply (the request was done without the `TPNOTRAN` flag).

For `tpresume()`, it means that the BEA TUXEDO system is unable to resume the global transaction because the caller is currently participating in work outside any global transaction with one or more resource managers. All such work must be completed before a global transaction can be resumed. The caller's state with respect to the local transaction is unchanged.

For the other functions, it means a call was made in transaction mode to a service that does not support transactions. What does this mean? Some services belong to server groups that access a DBMS that can support transactions, whereas other services may be responsible for printing out a form and accessing a printer that knows nothing about transactions. The configuration of services into servers and server groups is an administrative task. In order to determine which services support transactions, ask your system administrator. This is an application error. For the communication call to such a service to succeed, the `TPNOTRAN` flag must be set. In other words, you may not ask a service that does not support transactions to be a participant in the transaction. If you desire the service, it can be asked for only if the `TPNOTRAN` flag is explicitly set or if you access the service outside of your transaction.

Typed Buffer Errors

Typed buffer errors are returned as a result of sending requests or replies to processes in typed buffers that are unfamiliar to them. `TPEITYPE` is returned by `tpcall()`, `tpacall()`, and `tpconnect()` when the request data buffer is sent to a service that does not know about this type. What does this mean? The buffer types that processes know about are determined both by the configuration file and by the BEA TUXEDO system libraries that have been linked into the process. These libraries define and initialize a data structure that identifies the typed buffers that the process is to know about. The library can be tailored to each process. Also, an application can supply its own copy of a file that defines buffer types. An application can set up the buffer type data structure (referred to as a buffer type switch) on a per process basis. Refer to the `tuxtypes`(5) and `typesw`(5) reference pages. This is an administrative decision and is mentioned here to clarify what is meant by a process knowing about a typed buffer. The rule for sending requests is that you must always send a request in a typed buffer that a service knows about; this information can be obtained from your system administrator.

`TPEOTYPE` is returned by `tpcall()`, `tpgetrply()`, `tpdequeue()`, and `tprecv()` when the reply message is sent in a buffer that is not known or not allowed by the caller. What does this mean? Not known has the same semantics as previously explained for the request buffer. Not allowed means that although the process knows of the existence of this buffer type, the type returned to it does not match the type of the buffer it allocated to receive the reply and the caller is not allowing for a change in buffer type. The caller indicates this preference by setting *flags* to `TPNOCHANGE`. In this case, strong type checking is enforced, returning `TPEOTYPE` when violated. The default is to have weak type checking, allowing a different buffer type to be returned as long as it is known to the caller. Again, the rule for sending replies is that the reply buffer must be known to the caller and you must observe strong type checking if it has been indicated.

Call Descriptor Errors

The errors discussed in this section can occur only when making asynchronous calls or conversational calls because they involve the misuse of call descriptors. Asynchronous calls depend on call descriptors to identify replies with their corresponding requests. Conversational sends and receives depend on call descriptors to identify the connection; the call that initiates the connection depends on the availability of a call descriptor. There are two things that the BEA TUXEDO system doesn't like you to do with call descriptors:

- exceed your limit (TPELIMIT)
- reference one that has become invalid (TPEBADDESC)

The limit for outstanding call descriptors (replies) has been defined for the system as fifty and is a non-tunable parameter. The only way to change it is to recompile the system. The maximum number of descriptors allowed should be ample for your application, but this limit is system-defined and cannot be redefined by your application.

The limit for call descriptors for simultaneous conversational connections is defined in the configuration file and is more flexible than the limit for replies. The MAXCONV parameter in the RESOURCES section of the configuration file can be changed when the application is not running; it can be dynamically changed in the MACHINES section when the application is running. (See tmconfig(1).)

There are two general ways that a call descriptor can become invalid. If a call descriptor has been used to retrieve a message (including a failed message) and an attempt is made to reuse it, the system complains that you cannot reuse the descriptor and returns TPEBADDESC in tperrno.

Sometimes a condition occurs where you can no longer reference a call descriptor although it has never been used to retrieve a message. In this case we refer to the descriptor as having become stale and any attempt to reference it causes TPEBADDESC to be returned. One of the conditions that causes this to happen is calling tpabort() or tpcommit() when there are still transaction replies (replies for requests sent without the TPNOTRAN flag) to be retrieved. The outstanding descriptors for these transaction replies are considered stale. Another condition that causes this to happen is transaction time-out. When it is reported on the call to tpgetrply(), no message is retrieved with that descriptor, and any further reference to it is invalid because it is considered stale. This error can be corrected at the application level.

General Communication Call Errors

These errors can occur when making communication calls but have nothing to do with the nature of the call being synchronous or asynchronous.

The communication errors, `TPESVCERR` and `TPESVCFAIL`, are the result of the reply part of communication. They can be returned as a result of a call to `tpcall()` or `tpgetrply()` and they are determined by the arguments passed to and the processing done by `tpreturn()`. If `tpreturn()` encounters an error in processing or handling arguments, it will cause a `failed` message to be sent to the caller. This `failed` message is detected by the receiver with `tperrno` being set to `TPESVCERR`. The caller's data is not sent, and if the failure was on `tpgetrply()`, the call descriptor becomes invalid. If an error of this nature is not encountered by `tpreturn()`, then the value placed in *rval* determines the success or failure of the call. If the application logic placed the value `TPFAIL` in this parameter, `TPESVCFAIL` is returned in `tperrno` and the data message is sent to the caller.

The error codes `TPEBLOCK` and `TPGOTSIG` can happen on the request or the reply end of message communication. As a result, it can be returned for all three of the request/response communication calls. `TPEBLOCK` is returned when a blocking condition exists and the process sending a request either synchronously or asynchronously has indicated that it does not want to wait on a blocking condition by setting its *flags* parameter to `TPNOBLOCK`. A blocking condition can exist when sending a request if, for example, all the queues of the desired service are full. When `tpcall()` indicates a no blocking condition, it affects only the sending part of the communication. If the call successfully sends the request, `TPEBLOCK` will not be returned regardless of any blocking situation that may exist while the call waits for the reply. `TPEBLOCK` is returned for `tpgetrply()` when the call is made with *flags* set to `TPNOBLOCK` and a blocking condition is encountered while awaiting the reply; for example, if a message is not currently available.

`TPGOTSIG` really does not flag an error condition but indicates when a signal interrupts a BEA TUXEDO system call. If the communication functions set their *flags* parameter to `TPSIGRSTRT`, the calls will not fail and this code will not be returned in `tperrno`.

Conversational Errors

Once a conversational connection has been established, `tpsend()` and `tprecv()` can fail with a `TPEEVENT` error. An event has occurred. No data is sent by `tpsend()`. The event type is returned in the *revent* member of `TPSVCINFO`. A course of action is dictated by the particular event.

In conversational services `tpsend()`, `tprecv()`, and `tpdiscon()` return `TPEBADDESC` when an unknown descriptor is specified.

Time-Out Errors

Time-out errors can occur for one of two reasons:

- ♦ the maximum length of time a blocking call may remain blocked until the caller regains control has exceeded the amount of time it was allotted, that is, a blocking time-out occurred
- ♦ the duration of a transaction from start to finish has exceeded the amount of time it was allotted, that is, a transaction time-out occurred

As a result, this error can be returned on communication calls for either blocking or transaction time-out and on `tpcommit()` for transaction time-out only. In every case, if a process is in transaction mode and `TPETIME` is returned on a failed call, it means a transaction time-out has occurred.

`TPETIME` indicates a blocking time-out on a communication call if

- ♦ the call was not made in transaction mode and
- ♦ the call was not made with *flags* set to `TPNOBLOCK`

You may recall that if this flag is set, a blocking time-out cannot occur because the call returns immediately if a blocking condition exists.

Blocking time-out is a value set by the administrator of the system and is defined in the configuration file. Transaction time-out is defined by the application by the first argument passed to `tpbegin()`.

Further implications concerning the concept of time-out will be discussed in the section “Time-Out” later in this chapter.

Errors Leading to Abort

Errors by a participant in a transaction can cause `tpcommit()` to fail returning the error code of `TPEABORT` in `tperrno`. The transaction is implicitly aborted because of the failure and should be explicitly aborted. There are two ways that this error code can be returned:

- ♦ if a transaction has been marked abort-only by the initiator or one of the participants, or
- ♦ the transaction timed out and its status is known to be aborted

Errors Signaling Heuristic Decisions

Based on how `TP_COMMIT_CONTROL` is set, `tpcommit()` may return `TPEHAZARD` or `TPEHEURISTIC`. If `TP_COMMIT_CONTROL` is set to `TP_CMT_LOGGED`, the application gets control before the second phase of the two-phase commit is done, so it may not hear about a heuristic that occurs during the second phase. (Note that `TPEHAZARD` or `TPEHEURISTIC` can be returned if only 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 `TP_COMMIT_CONTROL` is set to `TP_CMT_COMPLETE`, then `TPEHEURISTIC` is returned if any of the resource managers reports a heuristic decision, and `TPEHAZARD` is returned if any of the involved resource managers reports a hazard. `TPEHAZARD` simply means that a participant failed during the second phase of commit (or during a one-phase commit) and we can't know if it completed the transaction successfully or unsuccessfully.

Application-Specific Errors

The previous sections dealt with the various categories into which system errors may fall. Your application can set up a method whereby you can pass information about user-defined errors to calling programs.

The mechanism involves use of the `rcode` argument of `tpreturn`(3) and the global variable `tpurcode`(5).

How to Deal with Errors

Your application logic should test for error conditions after the calls that have return values, and take suitable steps in the face of them. You may want to test if `-1` or `NULL` (depending on which the call returns) has been returned after a function call. In the event that it has been, you may invoke a function that contains a `switch` statement to test for specific values of `tperrno` and perform the appropriate application logic in each case.

Two routines, `tpstrerror`(3c) and `Fstrerror`(3fml), are provided to retrieve the text of an error message from the message catalogs for the BEA TUXEDO system and FML, respectively. The routines return a pointer to the error message. Your program can use the pointer to direct the text to `userlog`(3c) or to another destination. An example is shown in Listing 7-1.

Listing 7-1 illustrates a general way of dealing with errors. The term *`atmicall`*`()` is used in this example generically to represent an ATMI function call.

The code following the `switch` statement in Listing 7-1 illustrates how `tpurcode` can be used to disclose an application-defined code.

Listing 7-1 How to Deal with Errors

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

extern int tperrno;
extern int tpurcode;

main()

{
int rtnval;

if (tpinit((TPINIT *) NULL) == -1)
   error message, exit program;
if (tpbegin(30, 0) == -1)
   error message, tpterm, exit program;

allocate any buffers,
make atmi calls
check return value

rtnval = atmicall();

if (rtnval == -1) {
    switch(tperrno) {
    case TPEINVAL:
        fprintf(stderr, "Invalid arguments were given to atmicall\n");
        fprintf(stderr, "e.g., service name was null or flags wrong\n");
        break;
    case ...:
        fprintf(stderr, ". . .");
        break;
Include all error cases described in the atmicall(3) reference page.
Other return codes are not possible, so there should be no default within
the switch statement.

if (tpabort(0) == -1) {
   char *p;
   fprintf(stderr, "abort was attempted but failed\n");
   p = tpstrerror(tperrno);
   userlog("%s", p);
}
}
else
if (tpcommit(0) == -1)
fprintf(stderr, "REPORT program failed at commit time\n");

The following code fragment shows how an application-specific
return code can be examined.
.
.
.
ret = tpcall(“servicename”, (char*)sendbuf, 0, (char **)&rcvbuf, &rcvlen, \
(long)0);
.
.
.
(void) fprintf(stdout, “Returned tpurcode is: %d\n”, tpurcode);


free all buffers
tpterm();
exit(0);
}
```

The specific values of `tperrno` give you more insight into the nature of the problem and on what level it can be corrected.

If your application has defined a list of error conditions specific to your processing, the same can be said for `tpurcode`.

Fatal Transaction Errors

In managing transactions, it is important to understand which errors prove fatal to transactions. When these errors are encountered, transactions should be explicitly aborted on the application level by having the initiator of the transaction call `tpabort()`. Basically, there are three conditions that cause a transaction to fail. They are:

- the initiator or a participant of the transaction caused it to be marked abort-only for one of the following reasons:
    - `tpreturn()` encountered an error while processing its arguments (`TPESVCERR`)
    - the *rval* argument of `tpreturn()` was set to `TPFAIL` (`TPESVCFAIL`)
    - the type or subtype of the reply buffer is not known or allowed by the caller and, as a result, success or failure cannot be determined (`TPEOTYPE`)
- the transaction timed out (`TPETIME`)
- `tpcommit()` was called by a participant rather than by the originator of a transaction (`TPEPROTO`)

If `TPESVCERR`, `TPESVCFAIL`, `TPEOTYPE`, or `TPETIME` is returned for any of the communication calls, the transaction should be explicitly aborted with a call to `tpabort()`. If there are still outstanding descriptors, there is no need to wait for them before explicitly aborting the transaction. However, any attempt to access these descriptors after the transaction has been terminated will return `TPEBADDESC` since they are considered stale after the call.

Note that in the case of `TPESVCERR`, `TPESVCFAIL`, and `TPEOTYPE`, communication calls are still allowed as long as the transaction has not timed out. With the return of these errors, the transaction has been marked abort-only. In order for any further work to have any lasting effect, the communication calls should be made with the *flags* parameter set to `TPNOTRAN`. In this way, the work performed for the transaction that has been marked abort-only will not be rolled back when the transaction is aborted.

When a transaction time-out occurs, communication can continue, but it must be conducted with the following conditions enforced. The communication requests

- cannot require replies
- cannot block
- and cannot be performed on behalf of the caller’s transaction

This means asynchronous calls can be made with the *flags* parameter set to `TPNOREPLY|TPNOBLOCK|TPNOTRAN`.

Calling `tpcommit()` from the wrong participant in a transaction represents the only protocol error that is fatal to transactions. This error can be corrected on the application level during the development phase.

Calling `tpcommit()` when there is initiator/participant failure or transaction time-out represents the implicit abort error discussed earlier in the section “Errors Leading to Abort.” Because the commit failed, the transaction should be aborted.

Time-Out

As already indicated, there are two possible types of time-out that can occur in the BEA TUXEDO system. The effect of time-out on communication calls is different depending on the type that occurred. Also, the following issues are addressed in the following sections.

- What happens if a transaction times out while committing?
- Do calls to services that are not part of your transaction use time on your transaction clock?

Blocking vs. Transaction Time-Out

We have defined blocking time-out as exceeding the amount of time a call can wait for a blocking condition to clear up. Transaction time-out occurs when a transaction takes longer than the amount of time defined for it in the *timeout* argument to `tpbegin()`. By default, if a process is not in transaction mode, blocking time-outs are performed. When the *flags* parameter of a communication call is set to `TPNOTIME`, it applies to blocking time-outs only. If a process is in transaction mode, blocking time-out and the `TPNOTIME` flag are not relevant. The process is sensitive to transaction time-out only as it has been defined for it when the transaction was started. What are the implications of the two different types of time-out with concern to communication calls?

If a process is not in transaction mode and a blocking time-out occurs on an asynchronous call, the communication call that blocked will fail, but the call descriptor is still valid and may be used on a re-issued call. Further communication in general is unaffected.

In the case of transaction time-out, the call descriptor to an asynchronous transaction reply (done without the `TPNOTRAN` flag) becomes stale and may no longer be referenced. The only further communication allowed is the one case described earlier of no reply, no blocking, and no transaction.

Effect on tpcommit()

What is the state of a transaction if time-out occurs after the call to `tpcommit()`? It is unknown; the transaction can have either succeeded or failed. If the transaction timed out and the system knows that it was aborted, this is communicated to you by the error code `TPEABORT` returned in `tperrno`. If the status of the transaction is unknown, `TPETIME` is the error code. When the state of the transaction is in doubt, you must query the resource to see if any of the changes that were part of that transaction have been applied to it in order to discover whether the transaction committed or aborted.

Effect of 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 of that transaction and as such the service's success or failure cannot influence the outcome of that transaction. This will be discussed in greater detail in the next section, "Roles of tpreturn() and tpforward()." However, if the caller is expecting a reply, its transaction clock is still ticking away while the services that generate the reply are being performed. As a result, the transaction can time out while waiting for a reply that is due from a service that is not part of that transaction.

Roles of tpreturn() and tpforward()

If a process is called in transaction mode, `tpreturn()` and `tpforward()` place the service's portion of the transaction in a state where it can be either committed or aborted when the transaction is completed by its initiator. A service may be called several times on behalf of the same transaction. It is not fully committed or aborted 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 `tpreturn()` is called with outstanding descriptors with *rval* set to `TPSUCCESS`, this constitutes a protocol error and is returned as `TPESVCERR` to the process waiting on `tpgetrply()`. If the process is in transaction mode, it will cause the caller's current transaction to be marked internally as abort-only. Even if the initiator of the transaction should call `tpcommit()`, the transaction is aborted implicitly. If `tpreturn()` is called with outstanding descriptors with *rval* set to `TPFAIL`, `TPESVCFAIL` is returned to the process waiting on `tpgetrply()`. The effect on the transaction is the same.

It is always the case that when `tpreturn()` is called in transaction mode, it can determine the fate of that transaction either from the processing errors it encounters or from the value the application places in *rval*. Calling `tpforward()` can be used to indicate success up to that point in processing the request. If no application errors have been detected, `tpforward()` is invoked, otherwise `tpreturn()` with `TPFAIL`. If `tpforward()` is called improperly, it is considered a processing error and a `failed` message is returned to the requester.

Many of the ideas presented here have already been discussed in earlier sections, but they bear repeating. The following sections highlight various possible scenarios involving the transaction role of `tpreturn()` as well as the communication rules.

Service in Same Transaction as Caller

This is the straightforward case of the caller in transaction mode that calls another service to participate in the current transaction. What are the implications?

- `tpreturn()` and `tpforward()`, when called by the participating service, place that service's portion of the transaction in a state where it can be either aborted or committed by the initiator.
- The success or failure of the called process affects the current transaction. If any of the errors that prove fatal to transactions are encountered by the participant, the current transaction is marked abort-only.
- The lasting effect of the work done by a successful participant is dependent on the fate of the transaction; that is, if the transaction is aborted, the work of all participants is undone.
- The `TPNOREPLY` flag cannot be used when calling another service to participate in the current transaction.

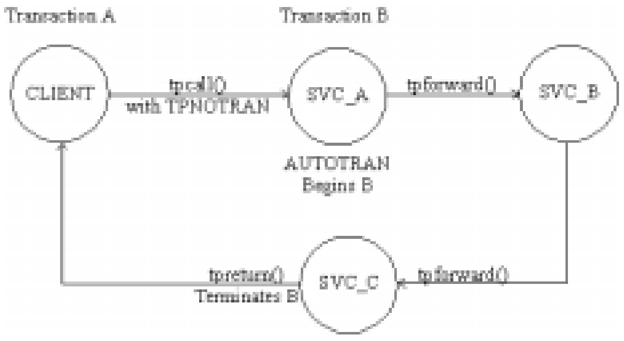
Service in Different Transaction with AUTOTRAN Set

If a communication call is made with the `TPNOTRAN` flag set and the called service is configured so that a transaction will automatically get started when it is called, these processes will both be in transaction mode but they will be in different transactions. What are the implications?

- `tpreturn()` plays the initiator's transaction role to terminate the transaction in the service where the transaction was automatically started. Alternatively, if the transaction is automatically started in a service that terminates with `tpforward()`, the `tpreturn()` in the last service in the forward chain plays the initiator's transaction role to terminate the transaction. Refer to Figure 7-1.
- Because it is in transaction mode, `tpreturn()` is also vulnerable to failure and is subject to the failure of any participant in the transaction as well as transaction time-out and as a result is more likely to send a *failed* message to the caller.
- Any *failed* messages or application failures returned to the caller do not affect the state of the caller's transaction.

- The caller is vulnerable to its own transaction timing 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.

Figure 7-1 Transaction Roles of tpforward() and tpreturn() with AUTOTRAN



Service Starts 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 as many transactions as it wants with explicit calls to `tpbegin()`, `tpcommit()`, and `tpabort()`. As a result, the transaction is already completed before the call to `tpreturn()`. What are the implications?

- `tpreturn()` plays no transaction role; that is, the role of `tpreturn()` would be exactly the same whether transactions were explicitly defined within the service routine or not.
- `tpreturn()` can send any value back in *rval* regardless of the outcome of the transaction.
- Typically, the errors returned will be processing errors, buffer type errors, or application failure, and the normal rules for `TPESVCFAIL`, `TPEITYPE`/`TPEOTYPE`, and `TPESVCERR` are followed.
- Any `failed` messages or application failures returned to the caller do not affect the state of the caller's transaction.
- The caller is vulnerable to its own transaction timing 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.

Transaction Rules

Certain rules are in effect when processes perform in transaction mode. Many of them have been touched upon already; but now, by way of summary, let’s bring them together and discuss them in one place.

Communication Etiquette

The basic communication etiquette that must be observed while in transaction mode is as follows:

- ♦ Processes that are participants in the same transaction must require replies for their requests.
- ♦ Requests requiring no reply can be made only if the *flags* parameter of `tpacall` is set to `TPNOTRAN|TPNOREPLY`.
- ♦ A service must retrieve all asynchronous transaction replies before calling `tpreturn()` or `tpforward` (this applies regardless of transaction mode).
- ♦ The initiator must retrieve all asynchronous transaction replies (made without the `TPNOTRAN` flag) before calling `tpcommit()`.
- ♦ The asynchronous replies that must be retrieved include those that are expected from non-participants of the transaction, that is, replies expected for requests made with `tpacall` suppressing the transaction but not the reply.
- ♦ If a transaction has not timed out but is marked abort-only, further communication should be performed with the `TPNOTRAN` flag set so that the work done as a result of the communication has lasting effect 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 will return `TPEBADDESC`
  - ♦ further calls to `tpgetrply()` or `tprecv()` for any outstanding descriptors will return the global state of transaction time-out by setting `tperrno` to `TPETIME`
  - ♦ asynchronous calls can be made with the *flags* parameter of `tpacall()` set to `TPNOREPLY|TPNOBLOCK|TPNOTRAN`

- Once a transaction has been marked abort-only for reasons other than time-out, a call to `tpgetrply()` will return whatever represents the local state of the call, that is, it can either return success or an error code that represents 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 will return `TPEBADDESC` (this applies regardless of transaction mode).
- Once a transaction is aborted, all outstanding transaction call descriptors (made without the TPNOTRAN flag) become stale, and any further reference to them will return `TPEBADDESC`.

BEA TUXEDO System-Supplied Subroutines

In both the standard subroutines, namely `tpsvrinit()` and `tpsvrdone()`, transactions may be defined and communication may be performed. What rules must they follow?

tpsvrinit()

The BEA TUXEDO system server abstraction calls `tpsvrinit()` during initialization. This routine is called after the process has become a server but before it handles service requests. If `tpsvrinit()` performs any asynchronous communication, all replies must be retrieved before returning, or BEA TUXEDO will ignore all pending replies and the server exits. If `tpsvrinit()` defines any transactions, they must be completed with all asynchronous replies retrieved before returning, or BEA TUXEDO will abort the transaction and ignore the outstanding replies. The server exits gracefully.

tpsvrdone()

The BEA TUXEDO system server abstraction calls `tpsvrdone()` after it has finished processing service requests but before it exits. Its services are no longer advertised, but it has not yet left the application. If `tpsvrdone()` initiates communication, it must retrieve all outstanding replies before it returns, or the pending replies will be ignored by the BEA TUXEDO system and the server exits. If a transaction has been started within this subroutine, it must be completed with all replies retrieved, or BEA TUXEDO will abort the transaction and ignore the replies. The server exits.

Leaving the Application

`tpterm()` is used to remove a client from an application. What transaction rules must it obey? If the client is in transaction mode, the call fails with `TPEPROTO` returned in `tperrno`, and the client is still part of the application and in transaction mode. When the call is successful, no further communication or participation in transactions is allowed because the process is no longer part of the application.

Global Transactions and Resource Managers

An interesting point arises when using the ATMI transaction primitives to define transactions. BEA TUXEDO makes an internal call to pass the global transaction information to each resource manager participating in the transaction. When `tpcommit()` or `tpabort()` is called, BEA TUXEDO makes internal calls to direct each resource manager to commit or abort the work they did on behalf of the caller's global transaction. When you write service routines in a DTP environment, you need not and should not make resource manager-specific calls to start, commit, or abort transactions. When a global transaction has been initiated either explicitly or implicitly, you should not make explicit calls to the resource manager's transaction primitives in your application code. Failure to follow this transaction rule will give indeterminate results.

This represents a good occasion to use the transaction primitive, `tpgetlev()`, to determine if a process is already in a global transaction before calling the resource manager's transaction primitive.

Some resource managers offer specific options in their interface. (For example, a resource manager might offer various transaction consistency levels or flags.) Some resource manager providers offer programmers of distributed applications the opportunity to negotiate these options using resource manager-specific calls; in other resource managers these options are hard-coded in the version of the transaction interface supplied by the resource manager provider. Documentation for the resource managers you are using should be consulted for further information on this subject.

In the BEA TUXEDO system SQL resource manager, 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. The method of setting such options will vary for other resource managers.

Comprehensive Example

Transaction integrity, message communication, and resource access represent the major needs of an On-line-Transaction-Processing (OLTP) application.

Listing 7-2 shows the ATMI transaction, buffer management, and communication routines working together with the SQL statements that access a resource manager. The example is taken from the ACCT server that is part of the banking application and illustrates the CLOSE_ACCT service.

The example illustrates the use of the `set transaction` statement (line 49) to set the consistency level and access mode of the transaction (when read/write access is specified the consistency level defaults to high consistency) before the first SQL statement that accesses the database. 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()` is invoked to allocate 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). This is followed by a `tpcall()` to the `WITHDRAWAL` service (line 79). An SQL `delete` statement updates the database by removing the account in question (line 86).

If all is successful, the buffer allocated within the service is freed (line 98), the `TPSVCINFO` data buffer that was sent to the service is updated to indicate the successful completion of the transaction (line 99); the transaction is automatically committed if the service was the initiator. `tpreturn()` returns `TPSUCCESS` and the updated buffer to the client process making the request to close the account. The successful completion is reported to the status line of the form.

After each function call, success or failure is determined. In the case of failure, the buffer allocated within the service is freed, the transaction is aborted if started in the service, and the `TPSVCINFO` buffer is updated to show the cause of failure (lines 80-83). `tpreturn()` returns `TPFAIL` and the message in the updated buffer is reported to the status line of the form.

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

Listing 7-2 ACCT Server

```
001   #include <stdio.h>              /* UNIX */
002   #include <string.h>             /* UNIX */
003   #include <fml.h>                /* TUXEDO */
004   #include <atmi.h>               /* TUXEDO */
005   #include <Usysflds.h>           /* TUXEDO */
006   #include <sqlcode.h>            /* TUXEDO */
007   #include <userlog.h>            /* TUXEDO */
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    */
```

```
void
#ifdef __STDC__
LOSE_ACCT(TPSVCINFO *transb)

#else

CLOSE_ACCT(transb)
TPSVCINFO *transb;
#endif

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

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

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

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

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

   /* Do final withdrawal */

   /* make withdraw request buffer */
   if ((reqfb = (FBFR *)tpalloc("FML",NULL,transb->len)) == (FBFR *)NULL) {
    (void)userlog("tpalloc failed in close_acct\n");
    (void)Fchg(transf, STATLIN, 0,
        "Unable to allocate request buffer", (FLDLEN)0);
    tpreturn(TPFAIL, 0, transb->data, 0L, 0);
   }
   reqlen = Fsizeof(reqfb);
   (void)Finit(reqfb,reqlen);
```

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

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

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

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

    /* Delete account record */

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

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

The Central Event Log

The central event log is a UNIX system file to which you can send messages from BEA TUXEDO system clients and services. Writing to the central event log is accomplished through the `userlog`(3c) function. The central event log simply provides a record of events considered worth recording. Any organized analysis of the central event log must be provided by the application. Application developers are encouraged to establish fairly strict guidelines for events to be recorded in the `userlog`(3c). Application debugging is made easier if the log is not flooded with trivial messages.

How the Log Is Named

One of the system parameters set up by the administrator determines the absolute pathname prefix of the userlog error message file on each machine. The `userlog()` function concatenates the month, day, and year in the form *`mmddyy`* to the prefix to form the full file name of the central event log. That means that if a process sends a message to the central event log on succeeding days, the message is written into different files.

What Log Entries Look Like

Entries on the log consist of:

- ♦ a tag made up of the
  - ♦ time of day (*`hhmmss`*)
  - ♦ the name of the machine (the name that is returned by `uname`)
  - ♦ the name and process-ID of the process calling `userlog()`
- ♦ the message text—For BEA TUXEDO system messages, text is preceded by the message catalog name and message number.
- ♦ optional arguments in `printf`(3S) format

For example, if the call:

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

is made at 4:22:14pm by the `security` program, on a machine where `uname` returns the value `mach1`, the resulting log entry will look like this:

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

assuming `23451` is the process ID for `security`, and that the variable `usrnm` contains the value `abc`.

If the above message was generated by BEA TUXEDO (as opposed to the application), it might look like this:

```
162214.mach1!security.23451: LIBSEC_CAT: 999: Unknown User 'abc'
```

where `LIBSEC_CAT: 999:` represents a message catalog name and message number.

If the message was sent to the central event log while the process is in transaction mode, the user log entry will have additional components in the tag. These components consist of the literal `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. This identifier is used mainly for administrative purposes, but it does make an appearance in the tag that prefixes the messages in the central event log. If the foregoing message is written to the central event log in transaction mode, the resulting log entry will look like this:

```
162214.mach1!security.23451: gtrid x2 x24e1b803 x239:
Unknown User 'abc'
```

How to Write to the Event Log

You can either have the error message you wish to write to the log in a variable of type `char *` and use the variable name as the argument to the call, or include the message as a literal within quotation marks as the argument to the call, as shown in the example below.

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

In this example, the message is sent to the central event log if `tpopen()` returns a negative number.

`userlog()` is similar to the UNIX System command `printf`(3S). That is, the format portion can contain literals and/or conversion specifications for a variable number of arguments.

Debugging Application Processes

While it is possible to use `userlog()` statements to help debug application software, it is sometimes necessary to use a debugger command for more complex debugging.

The standard UNIX system debugging command is `sdb`(1). Refer to a UNIX System programmer's reference manual. Client processes compiled with the `-g` option are debugged in the conventional manner explained on the `sdb` reference page. The syntax of the `sdb` command can take the following form:

```
sdb -W client - directory_list
```

For complete syntactical information, refer to the reference page. To run the client process:

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

The debugging of server programs is more complicated. Normally, servers are started using the `tmboot` command, which starts the server on the correct machine with the correct options. When using `sdb`, it is necessary to run the server directly rather than through the `tmboot` command. The BEA TUXEDO system `tmboot(1)` command passes undocumented command line options to the server's predefined `main()`. When you want to run your server, you will need to pass it these options as well. To obtain these options, run `tmboot` with the `-n` and `-d 1` options. Refer to Section 1 of the *BEA TUXEDO Reference Manual*. The `-n` option tells `tmboot` not to perform the actual execution; `-d 1` tells it to print out debugging level one statements. You can pass other options as well to `tmboot` in order to get information on a particular process rather than all of them. The output from `tmboot` will look something like the following, revealing the command line options it passes to the server's `main()`:

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

When you want to run your server program using `sdb`, you must pass the options following the word `server` to its `run` (`r`) subcommand. As a result, the `run` command will look like the following:

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

Also note that the server you are attempting to run from `sdb` must not already be running as part of the configuration, or the server will exit gracefully indicating a duplicate server in the central event log.