THE ENTERPRISE MIDDLEWARE SOLUTION

BEA TUXEDO

COBOL Guide

BEA TUXEDO Release 6.5
Document Edition 6.5
February 1999

Copyright

Restricted Rights Legend

Trademarks or Service Marks

BEA TUXEDO COBOL Guide

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

Contents

1 Introduction and a Simple Application

About This Chapter

This chapter contains a tutorial that describes a simple one-client, one-server application called `CSIMPAPP`. An interactive form of this chapter is distributed with the BEA TUXEDO system software.

If you follow the ten steps of the tutorial you will:

- learn how a BEA TUXEDO application is organized
- see how clients and servers are written and compiled
- understand how an application is described in the configuration file
- actually create an executable version of `CSIMPAPP`
- boot, run and shutdown the application

Some Preliminaries

Before you can run this tutorial the BEA TUXEDO system software must be installed so that the files and commands referred to in this chapter are available.

If you are personally responsible for installing the BEA TUXEDO system software, consult the *BEA TUXEDO Installation Guide* for information about how to install the BEA TUXEDO system.

If the installation has already been done by someone else, you need to know the pathname of the root directory of the installed software. You also need to have read and execute permissions on the directories and files in the BEA TUXEDO system directory structure so you can copy `CSIMPAPP` files and execute BEA TUXEDO commands.

The CSIMPAPP Tutorial

`CSIMPAPP` is a very basic BEA TUXEDO system application. It has one client and one server. The server performs only one service; it accepts a string from the client and returns the same string in upper case.

The tutorial consists of ten steps (plus an eleventh step for shutdown) designed to introduce you to the BEA TUXEDO system by showing how an application is developed and by encouraging you to bring the application up and run it. Each of the steps includes one or more smaller steps.

Step 1: Copy the CSIMPAPP Files

1. Make a directory for `CSIMPAPP` and `cd` to it:

   ```
   mkdir CSIMPDIR
   cd CSIMPDIR
   ```

   This is suggested so you will be able to see clearly the `CSIMPAPP` files you have at the start and the additional files you create along the way. Use the standard shell (`/bin/sh`) or the Korn shell; not `csh`.

2. Set and export environment variables

```
TUXDIR=<pathname of the BEA TUXEDO System root directory>
APPDIR=<pathname of your present working directory>
TUXCONFIG=$APPDIR/TUXCONFIG
COBDIR=<pathname of the COBOL compiler directory>
COBCPY=$TUXDIR/cobinclude
COBOPT="-C ANS85 -C ALIGN=8 -C NOIBMCOMP -C TRUNC=ANSI -C OSEXT=cbl"
CFLAGS="-I$TUXDIR/include"
PATH=$PATH:$TUXDIR/bin
LD_LIBRARY_PATH=$COBDIR/coblib:${LD_LIBRARY_PATH}
export TUXDIR APPDIR TUXCONFIG UBBCONFIG COBDIR COBCPY
export COBOPT CFLAGS PATH LD_LIBRARY_PATH
```

   You need `TUXDIR` and `PATH` to be able to access files in the BEA TUXEDO system directory structure and to execute BEA TUXEDO system commands. On SunOS, `/usr/5bin` must be the first directory in your `PATH`. On AIX, `LIBPATH` must be set instead of `LD_LIBRARY_PATH`. On HPUX, `SHLIB_PATH` must be set instead of `LD_LIBRARY_PATH`. You need to set `TUXCONFIG` to be able to load the configuration file as shown in Step 7.

3. Copy the `CSIMPAPP` files.

   ```
   cp $TUXDIR/apps/CSIMPAPP/* .
   ```

   Later on you will be editing some of the files and making them executable, so it is best to begin with a copy of the files rather than the originals delivered with the software.

4. List the files.

   ```
   $ ls
   CSIMPCL.cbl
   CSIMPSRV.cbl
   README
   TPSVRINIT.cbl
   UBBCSIMPLE
   WUBBCSIMPLE
   envfile
   ws
   $
   ```

   The files that make up the application are:

   - `CSIMPCL.cbl`—the source code for the client program
   - `CSIMPSRV.cbl`—the source code for the server program
   - `TPSVRINIT.cbl`—the source code for the server initialization program

- `UBBCSIMPLE`—the ASCII form of the configuration file for the application
- `WUBBCSIMPLE`—the config file for the Workstation example
- `ws`—a directory with `.MAK` files for client programs for three workstation platforms

Step 2: Examine the Client Program

Page through the client program source code:

```
$ more CSIMPCL.cbl
```

The output is shown in Listing 1-1.

Listing 1-1 Source code of CSIMPCL.cbl

```
    IDENTIFICATION DIVISION.
    PROGRAM-ID. CSIMPCL.
    AUTHOR. TUXEDO DEVELOPMENT.
    ENVIRONMENT DIVISION.
    CONFIGURATION SECTION.
    WORKING-STORAGE SECTION.
*****************************************************
* Tuxedo definitions
*****************************************************
    01 TPTYPE-REC.
    COPY TPTYPE.
 *
    01 TPSTATUS-REC.
    COPY TPSTATUS.
 *
    01 TPSVCDEF-REC.
    COPY TPSVCDEF.
 *
    01 TPINFDEF-REC VALUE LOW-VALUES.
    COPY TPINFDEF.
 *****************************************************
 * Log messages definitions
 *****************************************************
    01 LOGMSG.
           05 FILLER   PIC X(8) VALUE "CSIMPCL:".
           05 LOGMSG-TEXT PIC X(50).
    01 LOGMSG-LEN     PIC S9(9) COMP-5.
 *
```

```
   01 USER-DATA-REC    PIC X(75).
   01 SEND-STRING     PIC X(100).
   01 RECV-STRING     PIC X(100).
******************************************************
* Command line arguments
******************************************************
   LINKAGE SECTION.
   01 CMD-LINE.
          05 ARG-LENGTH PIC 9(4) COMP.
          05 ARG.
                 10 ARGS PIC X OCCURS 0 TO 100 DEPENDING
                        ON ARG-LENGTH.
*******************************************************
* Start program with command line args
*******************************************************

   PROCEDURE DIVISION USING CMD-LINE.
START-CSIMPCL.
   MOVE LENGTH OF LOGMSG TO LOGMSG-LEN.
   PERFORM CHECK-ARGS.
   PERFORM DO-TPINIT.
   MOVE ARG TO SEND-STRING.
   PERFORM DO-TPCALL.
   DISPLAY RECV-STRING.
   PERFORM DO-TPTERM.
   PERFORM EXIT-PROGRAM.

*******************************************************
* Check Arguments being passed
*******************************************************
CHECK-ARGS.
   IF ARG-LENGTH = 0
          DISPLAY "Usage: CSIMPCL string"
          PERFORM EXIT-PROGRAM
   END-IF.
   IF ARG-LENGTH = 100
          DISPLAY "Command Line Too Long"
          PERFORM EXIT-PROGRAM
   END-IF.

   MOVE "Started" TO LOGMSG-TEXT.
   PERFORM DO-USERLOG.

******************************************************
* Now register the client with the system.
******************************************************
DO-TPINIT.
   MOVE SPACES TO USRNAME.
   MOVE SPACES TO CLTNAME.
   MOVE SPACES TO PASSWD.
   MOVE SPACES TO GRPNAME.
   MOVE ZERO TO DATALEN.
   SET TPU-DIP TO TRUE.

 CALL "TPINITIALIZE" USING TPINFDEF-REC
          USER-DATA-REC
          TPSTATUS-REC.

   IF NOT TPOK
          MOVE "TPINITIALIZE Failed" TO LOGMSG-TEXT
          PERFORM DO-USERLOG
          PERFORM EXIT-PROGRAM
   END-IF.

*****************************************************
* Issue a TPCALL
*****************************************************
DO-TPCALL.
   MOVE ARG-LENGTH TO LEN.
   MOVE "STRING" TO REC-TYPE.
   MOVE "CSIMPSRV" TO SERVICE-NAME.
   SET TPBLOCK TO TRUE.
   SET TPNOTRAN TO TRUE.
   SET TPNOTIME TO TRUE.
   SET TPSIGRSTRT TO TRUE.
   SET TPCHANGE TO TRUE.

   CALL "TPCALL" USING TPSVCDEF-REC
                 TPTYPE-REC
                 SEND-STRING
                 TPTYPE-REC
                 RECV-STRING
                 TPSTATUS-REC.

   IF NOT TPOK
          MOVE "TPCALL Failed" TO LOGMSG-TEXT
          PERFORM DO-USERLOG
   END-IF.

 *****************************************************
 * Leave TUXEDO
 *****************************************************
 DO-TPTERM.
   CALL "TPTERM" USING TPSTATUS-REC.
   IF NOT TPOK
          MOVE "TPTERM Failed" TO LOGMSG-TEXT
          PERFORM DO-USERLOG
   END-IF.

 *****************************************************
 * Log messages to the userlog
 *****************************************************
```

```
131  DO-USERLOG.
132    CALL "USERLOG" USING LOGMSG
133           LOGMSG-LEN
134           TPSTATUS-REC.
135
136  *****************************************************
137  *Leave Application
138  *****************************************************
139  EXIT-PROGRAM.
140    MOVE "Ended" TO LOGMSG-TEXT.
141    PERFORM DO-USERLOG.
142   STOP RUN.
```

Here are the important things to see in this file:

lines 11, 14, 17, 20	`COPY`	Files needed whenever BEA TUXEDO ATMI calls are used
line 83	`TPINITIALIZE`	The ATMI call used by a client program to join an application.
line 106	`TPCALL`	Sends the message record to the service specified in `SERVICE-NAME`. `TPCALL` waits for a return message. `STRING` is one of the three basic BEA TUXEDO record types. The argument, `LEN IN TPTYPE-REC`, specifies the length of the record contained in `USER-DATA-REC`.
line 122	`TPTERM`	The ATMI call used to leave an application. A call to `TPTERM` is used to leave the application prior to performing a `STOP RUN`.
line 52	`DISPLAY`	This is the successful conclusion of the program. It prints out the message returned from the server.

References

The ATMI calls cited above are documented in the following pages in the *BEA TUXEDO Reference Manual*: `TPINITIALIZE`(3cbl), `TPTERM`(3cbl), `TPCALL`(3cbl), `USERLOG`(3cbl).

Step 3: Compile the Client

1. Run `buildclient` to compile the client program:

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

   where the output file is `CSIMPCL`, and the input source file is `CSIMPCL.cbl`.

2. Check the results:

   ```
   $ ls CSIMPCL*
   CSIMPCL   CSIMPCL.cbl   CSIMPCL.idy   CSIMPCL.int   CSIMPCL.o
   ```

   As can be seen, we now have an executable module called `CSIMPCL`.

References

`buildclient` is documented in `buildclient`(1) in the *BEA TUXEDO Reference Manual*.

Step 4: Examine the Server

1. Page through the server program source code:

   ```
   $ pg CSIMPSRV.cbl
   ```

Listing 1-2 Source code of CSIMPSRV.cbl

```
    IDENTIFICATION DIVISION.
    PROGRAM-ID. CSIMPSRV.
    AUTHOR. BEA TUXEDO DEVELOPMENT.
    ENVIRONMENT DIVISION.
    CONFIGURATION SECTION.
    WORKING-STORAGE SECTION.
*****************************************************
* Tuxedo definitions
*****************************************************
    01 TPSVCRET-REC.
    COPY TPSVCRET.
 *
    01 TPTYPE-REC.
    COPY TPTYPE.
 *
```

```
   01 TPSTATUS-REC.
   COPY TPSTATUS.
*
   01 TPSVCDEF-REC.
   COPY TPSVCDEF.
******************************************************
* Log message definitions
******************************************************
   01 LOGMSG.
            05 FILLER        PIC X(10) VALUE
                   "CSIMPSRV :".
            05 LOGMSG-TEXT   PIC X(50).
   01 LOGMSG-LEN             PIC S9(9) COMP-5.
******************************************************
* User defined data records
******************************************************
   01 RECV-STRING            PIC X(100).
   01 SEND-STRING            PIC X(100).
*
   LINKAGE SECTION.
*
   PROCEDURE DIVISION.
*
 START-FUNDUPSR.
   MOVE LENGTH OF LOGMSG TO LOGMSG-LEN.
   MOVE "Started" TO LOGMSG-TEXT.
   PERFORM DO-USERLOG.

******************************************************
* Get the data that was sent by the client
******************************************************
   MOVE LENGTH OF RECV-STRING TO LEN.
   CALL "TPSVCSTART" USING TPSVCDEF-REC
                 TPTYPE-REC
                 RECV-STRING
                 TPSTATUS-REC.

   IF NOT TPOK
          MOVE "TPSVCSTART Failed" TO LOGMSG-TEXT
          PERFORM DO-USERLOG
          PERFORM EXIT-PROGRAM
   END-IF.

   IF TPTRUNCATE
          MOVE "Data was truncated" TO LOGMSG-TEXT
          PERFORM DO-USERLOG
          PERFORM EXIT-PROGRAM
   END-IF.

   INSPECT RECV-STRING CONVERTING
   "abcdefghijklmnopqrstuvwxyz" TO
```

```
68     "ABCDEFGHIJKLMNOPQRSTUVWXYZ".
69     MOVE "Success" TO LOGMSG-TEXT.
70     PERFORM DO-USERLOG.
71     SET TPSUCCESS TO TRUE.
72     COPY TPRETURN REPLACING
73            DATA-REC BY RECV-STRING.
74
75  ******************************************************
76  * Write out a log err messages
77  ******************************************************
78   DO-USERLOG.
79     CALL "USERLOG" USING LOGMSG
80            LOGMSG-LEN
81            TPSTATUS-REC.
82  ******************************************************
83  * EXIT PROGRAM
84  ******************************************************
85   EXIT-PROGRAM.
86     MOVE "Failed" TO LOGMSG-TEXT.
87     PERFORM DO-USERLOG.
88     SET TPFAIL TO TRUE.
89     COPY TPRETURN REPLACING
90               DATA-REC BY RECV-STRING.
```

Here are the important things to see in this file:

line 49	`TPSVCSTART`	This routine is used to receive the service’s parameters and data. After a successful call, the `RECV-STRING` contains the data sent by the client.
lines 66-68	`INSPECT` statement	Converts the input to uppercase.
line 72	`COPY TPRETURN`	Returns the converted string to the client with `TPSUCCESS` set.
line 79	`USERLOG`	This routine logs messages that are used by the BEA TUXEDO system and applications.

2. Page through the server program source code:

```
$ pg TPSVRINIT.cbl
```

Listing 1-3 Source code of TPSVRINIT.cbl

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. TPSVRINIT.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
*
 01 LOGMSG.
     05 FILLER               PIC X(11) VALUE "TPSVRINIT :".
     05 LOGMSG-TEXT          PIC X(50).
 01 LOGMSG-LEN               PIC S9(9) COMP-5.
 *
  01 TPSTATUS-REC.
  COPY TPSTATUS.
 *********************************************************
  LINKAGE SECTION.
  01 CMD-LINE.
   05 ARGC PIC 9(4) COMP-5.
   05 ARG.
     10 ARGS PIC X OCCURS 0 TO 9999 DEPENDING ON ARGC.
 *
  01 SERVER-INIT-STATUS.
  COPY TPSTATUS.
 ***********************************************************
  PROCEDURE DIVISION USING CMD-LINE SERVER-INIT-STATUS.
  A-000.
    MOVE LENGTH OF LOGMSG TO LOGMSG-LEN.
 ***********************************************************
 * There are no command line parameters in this TPSVRINIT
 ***********************************************************
    IF ARG NOT EQUAL TO SPACES
           MOVE "TPSVRINIT failed" TO LOGMSG-TEXT
           CALL "USERLOG" USING LOGMSG
                  LOGMSG-LEN
                  TPSTATUS-REC
    ELSE
           MOVE "Welcome to the simple service" TO LOGMSG-TEXT
           CALL "USERLOG" USING LOGMSG
                  LOGMSG-LEN
                  TPSTATUS-REC
    END-IF.
 *
  SET TPOK IN SERVER-INIT-STATUS TO TRUE.
 *
  EXIT PROGRAM.
```

This subroutine is called during server initialization, before the server begins processing service requests. A default is provided by the BEA TUXEDO system that writes a message to `USERLOG` indicating that the server has been booted.

References

The ATMI calls and structure cited above are documented in the following pages in the *BEA TUXEDO Reference Manual*: `TPSVCSTART`(3cbl), `TPSVRINIT`(3cbl), `TPRETURN`(3cbl), `USERLOG`(3cbl).

Step 5: Build the Server

1. Run `buildserver` to compile the server program:

```
buildserver -C -o CSIMPSRV -f CSIMPSRV.cbl -f TPSVRINIT.cbl -s CSIMPSRV
```

   where the executable file to be created is named `CSIMPSRV`, and `CSIMPSRV.cbl` and `TPSVRINIT.cbl` are the input source files.

2. Check the results:

```
$ ls
CSIMPCL      CSIMPCL.int   CSIMPSRV.cbl    CSIMPSRV.o     TPSVRINIT.int
CSIMPCL.cbl  CSIMPCL.o     CSIMPSRV.idy    TPSVRINIT.cbl  TPSVRINIT.o
CSIMPCL.idy  CSIMPSRV      CSIMPSRV.int    TPSVRINIT.idy  UBBCSIMPLE
```

   As can be seen, we now have an executable module called `CSIMPSRV`.

References

`buildserver` is documented in `buildserver`(1) in the *BEA TUXEDO Reference Manual*.

Step 6: Edit the Configuration File

1. Edit the file:

Listing 1-4 The CSIMPAPP configuration file

```
#Skeleton UBBCONFIG file for the BEA TUXEDO COBOL Simple
Application.
#Replace the <bracketed> items with the appropriate values.

*RESOURCES
IPCKEY             <Replace with a valid IPC Key>

#Example:
#IPCKEY            123456

MASTER             simple
MAXACCESSERS       5
MAXSERVERS         5
MAXSERVICES        10
MODEL              SHM
LDBAL              N

*MACHINES
DEFAULT:
                   APPDIR="<Replace with the current pathname>"
                   TUXCONFIG="<Replace with TUXCONFIG Pathname>"
                   TUXDIR="<Root directory of BEA TUXEDO (not /)>"
                  ENVFILE="<pathname of file of environment vars>"
#Example:
#                  APPDIR="/home/me/simpapp"
#                  TUXCONFIG="/home/me/simpapp/TUXCONFIG"
#                  TUXDIR="/usr/tuxedo"

<Machine-name>     LMID=simple

#Example:
#usltux            LMID=simple

*GROUPS
GROUP1
                   LMID=simple     GRPNO=1     OPENINFO=NONE

*SERVERS
DEFAULT:
                   CLOPT="-A"

CSIMPSRV           SRVGRP=GROUP1   SRVID=1

*SERVICES
CSIMPSRV
```

2. Change values enclosed in angle brackets to your own local values:

`IPCKEY`	Use a value that will not conflict with any other users.
`TUXCONFIG`	Provide the full pathname of the binary `tuxconfig` file to be created in Step 7.
`TUXDIR`	Provide the full pathname of your BEA TUXEDO system root directory.
`APPDIR`	Provide the full pathname of the directory where you intend to boot the application; in this case, the current directory.
`ENVFILE`	Provide the full pathname for the environment file to be used by `mc`, `viewc`, `tmloadcf`, and so on.
`machine-name`	Provide the machine name as returned by `uname -n`.

3. The pathnames for `TUXCONFIG` and `TUXDIR` must be identical to those you set and exported in Step 1 in "Step 1: Copy the CSIMPAPP Files." The strings must be the actual values; environment variables (such as `$TUXCONFIG`) are not acceptable. Do not forget to remove the angle brackets.

References

The configuration file is documented in `ubbconfig`(5) in the *BEA TUXEDO Reference Manual*.

Step 7: Load the Configuration File

1. Run `tmloadcf` to load the configuration file:

```
$ tmloadcf UBBCSIMPLE
Initialize TUXCONFIG file: /usr/me/CSIMPDIR/TUXCONFIG [y, q] ? y
$
```

2. Check the results:

```
$ ls
CSIMPCL       CSIMPCL.o     CSIMPSRV.int    TPSVRINIT.int
CSIMPCL.cbl   CSIMPSRV      CSIMPSRV.o      TPSVRINIT.o
CSIMPCL.idy   CSIMPSRV.cbl  TPSVRINIT.cbl   TUXCONFIG
CSIMPCL.int    CSIMPSRV.idy   TPSVRINIT.idy    UBBCSIMPLE
```

We see that we now have a file called `TUXCONFIG`. The `TUXCONFIG` file is a new file system under the control of the BEA TUXEDO system.

References

`tmloadcf` is documented in `tmloadcf`(1) in the *BEA TUXEDO Reference Manual*.

Step 8: Boot the Application

Execute `tmboot` to bring up the application:

```
$ tmboot
Boot all admin and server processes? (y/n): y
Booting all admin and server processes in /usr/me/CSIMPDIR/TUXCONFIG

Booting all admin processes ...

exec BBL -A:
       process id=24223 ... Started.

Booting server processes ...

exec CSIMPSRV -A :
      process id=24257 ... Started.
2 processes started.
$
```

BBL is the administrative process that monitors the application shared memory structures. `CSIMPSRV` is our server that runs continuously awaiting requests.

References

`tmboot` is documented in `tmboot`(1) in the *BEA TUXEDO Reference Manual.*

Step 9: Enter a Request

Run the client program to submit a request:

```
$ CSIMPCL "hello world"
HELLO WORLD
```

We are successful!!!

Step 10: Using tmadmin

`tmadmin` is an interactive program that an administrator can use to check an application and make dynamic changes. It requires the `TUXCONFIG` variable to be set. We will show you just two of the many `tmadmin` commands.

1. Enter the command:

   ```
   tmadmin
   ```

   You will see the following lines.

   ```
   tmadmin - Copyright (c) 1987 ATT; 1991 USL. All rights reserved.

   >
   ```

   The greater-than sign (>) is the `tmadmin` prompt.

2. Enter the `printserver(psr)` command to display information about the servers:

```
> psr
a.out Name  Queue Name   Grp Name  ID  RqDone   Load Done  Current Service
----------  ----------   --------  --  ------   ---------  ---------------
BBL         531993       simple     0      0            0  (IDLE)
CSIMPSRV    00001.00001  GROUP1     1      0            0  (IDLE)
>
```

3. Enter the `printservice(psc)` command to display information about the services:

```
> psc
Service Name  Routine Name  a.out Name  Grp Name  ID  Machine #  Done  Status
------------  ------------  ----------  --------  --  -------    ----  -------
ADJUNCTBB     ADJUNCTBB     BBL         simple    0   simple        -  AVAIL
ADJUNCTADMIN  ADJUNCTADMIN  BBL         simple    0   simple        -  AVAIL
CSIMPSRV      CSIMPSRV      CSIMPSRV    GROUP1    1   simple        -  AVAIL
>
```

4. Leave `tmadmin` by entering a `q` at the prompt. You can boot and shut down the application from within `tmadmin`. We have done those functions with shell commands in Steps 8 and 11, respectively.

References

`tmadmin` is documented in `tmadmin`(1) in the *BEA TUXEDO Reference Manual*.

Step 11: Shut Down the Application

1. Run `tmshutdown` to bring the application down:

```
$ tmshutdown
Shutdown all admin and server processes? (y/n): y
Shutting down all admin and server processes in /usr/me/CSIMPDIR/TUXCONFIG

Shutting down server processes ...

Server Id = 1 Group Id = GROUP1 Machine = simple:  shutdown succeeded.

Shutting down admin processes ...

Server Id = 0 Group Id = simple Machine = simple:  shutdown succeeded.
2 processes stopped.
$
```

2. Check the `ULOG`:

```
$ cat ULOG*
$
140533.usltux!BBL.22964: LIBTUX_CAT:262: std main starting
140540.usltux!CSIMPSRV.22965: COBAPI_CAT:1067: INFO: std main starting
140542.usltux!CSIMPSRV.22965: TPSVRINIT :Welcome to the simple service
140610.usltux!?proc.22966: CSIMPCL:Started
140614.usltux!CSIMPSRV.22965: CSIMPSRV :Started
140614.usltux!CSIMPSRV.22965: CSIMPSRV :Success
140614.usltux!?proc.22966: switch to new log file
/home/usr_nm/CSIMPDIR/ULOG.112592
140614.usltux!?proc.22966: CSIMPCL:Ended
```

Each line of the `ULOG` for this session contains something of interest. First let's look at the format of a `ULOG` line:

```
time (hhmmss).machine_uname!process_name.process_id: log message
```

Now let's look at an individual line:

```
140542. Message from TPSVRINIT in CSIMPSRV
```

References

`tmshutdown` is documented in `tmshutdown`(1) in the *BEA TUXEDO Reference Manual*.

The `USERLOG` is documented in `USERLOG`(3cbl).

Summary

If you have reached this point, you have successfully brought up, run and brought down a BEA TUXEDO system application. You have seen what a client program and a server look like. You have edited a configuration file to refer to your own environment. You have invoked `tmadmin` to check on the activity of your application. In all the applications you may work on in the future the basic elements of client processes, server processes and a configuration file will be present, and you will have all of the BEA TUXEDO shell commands at your fingertips.

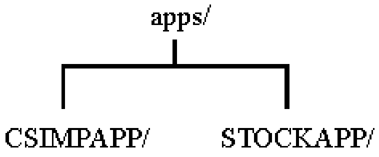
Good luck!

2 STOCKAPP Files

Directory Structure for STOCKAPP

This chapter describes the directory structure that pertains to the COBOL language binding feature under the `apps` directory, which is subordinate to the root directory for your BEA TUXEDO system software. We will also take a look at the files in the `STOCKAPP` directory. The directory structure is shown in Figure 2-1.

Figure 2-1 COBOL Directory structure under apps/



`CSIMPAPP` is described in Chapter 1, “Introduction and a Simple Application.”

Files

Table 2-1 lists the files of the stock application. The left hand column lists the source files delivered with the BEA TUXEDO system software. The center column lists files that are generated when the stock application is built. The right hand column gives a brief summary of the purpose of the file.

Table 2-1 Stock Application Files

Source	Generated	Purpose
BUY.cbl	BUY.o BUY	Client
BUYSR.cbl	BUYSR.o BUYSR	Contains BUY service
ENVFILE		ENVFILE used by tmloadcf
FILES		Descriptive list of all the files in STOCKAPP
FUNDPR.cbl	FUNDPR.o FUNDPR	Client
FUNDPRSR.cbl	FUNDPRSR.o FUNDPRSR	Contains PRICE QUOTE service
FUNDUP.cbl	FUNDUP.o FUNDUP	Client
FUNDUPSR.cbl	FUNDUPSR.o FUNDUPSR	Contains FUND UPDATE service
README		On-line version of the installation and boot procedures
SELL.cbl	SELL.o SELL	Client
SELLSR.cbl	SELLSR.o SELLSR	Contains SELL service
STKVAR		Contains variable settings, except for those within ENVFILE
STOCKAPP.mk		Application makefile
UBBCBSHM	TUXCONFIG	Sample UBBCONFIG file for use in a SHM mode configuration
cust	CUST.cbl cust.V cust.h	View used to define structure passed between the BUY and SELL clients and the BUYSR and SELLSR servers
quote	QUOTE.cbl quote.V quote.h	View used to define structure passed between the FUNDPR and FUNDUP clients and all the servers

Of the files in the directory, eight are `.cbl` files; `BUY.cbl`, `SELL.cbl`, `FUNDPR.cbl` and `FUNDUP.cbl` are client programs; `FUNDUPSR.cbl` is a conversational server; three others are servers or are associated with servers, two are there to generate data or transactions for the application.

The remaining files have various roles; some are files you need in any application, others are present simply to facilitate the use of `STOCKAPP` as an example. In subsequent chapters we will closely examine a number of the files, and give a more complete explanation of their role in the sample application. For now we just want to discuss the `STKVAR` file.

Edit STKVAR to Set Environment Variables

`STKVAR` is a file of environment variables needed by `STOCKAPP`. A complete copy of `STKVAR` is shown in Listing 2-1. The file takes up almost 100 lines, due largely to the extensive comments, but there are only a few that you should be concerned about immediately.

The first line referencing `TUXDIR` ensures that it is set. If it is not, execution of the file fails with the message:

```
TUXDIR: parameter null or not set
```

So set `TUXDIR` to the root directory of your BEA TUXEDO system directory structure, and export it.

As `STKVAR` is delivered, `APPDIR` is set to the directory in which the `STOCKAPP` source files are located: `${TUXDIR}/apps/STOCKAPP`. `APPDIR` is a directory where BEA TUXEDO system looks for your application-specific files. You might prefer to copy the `STOCKAPP` files to a different directory to safeguard the original source files. If you do, then the directory you use should be entered here. It does not have to be under `TUXDIR`.

The other variables specified in `STKVAR` play various roles in the sample application and you will need to be aware of them when you are developing your own application. They will all be mentioned at appropriate places later in this guide. Grouping them all in `STKVAR` is done to show you an example that you may want to adapt at a later time for use with a real application.

When you have made all necessary changes to `STKVAR`, execute `STKVAR` as follows:

```
. ./STKVAR
```

Listing 2-1 STKVAR: Environment Variables for STOCKAPP

```
#ident    "@(#)apps:STOCKAPP/STKVAR
#
# This file sets all the environment variables needed by the TUXEDO software
# to run the STOCKAPP
#
# This directory contains all the TUXEDO software
# System administrator must set this variable
#
TUXDIR=${TUXDIR:?}
#
# This directory contains all the user written code
#
# Contains the full path name of the directory that the application
# generator should place the files it creates
#
APPDIR=${HOME}/STOCKAPP
#
# Environment file to be used by tmloadcf
#
COBDIR=${COBDIR:?}
#
# This directory contains the cobol files needed
# for compiling and linking.
#
LD_LIBRARY_PATH=$COBDIR/coblib:${LD_LIBRARY_PATH}
#
# Add coblib to LD_LIBRARY_PATH
#
ENVFILE=${APPDIR}/ENVFILE
#
# List of field table files to be used by CBLVIEWC, tmloadcf, etc.
#
FIELDTBLS=fields,Usysflds
#
# List of directories to search to find field table files
#
FLDTBLDIR=${TUXDIR}/udataobj:${APPDIR}
#
# Set device for the transaction log; this should match the TLOGDEVICE
# parameter under this site’s LMID in the *MACHINES section of the
# UBBCBSHM file
#
TLOGDEVICE=${APPDIR}/TLOG
#
# Device for the configuration file
#
```

```
UBBCBSHM=$APPDIR/UBBCBSHM
#
# Device for binary file that gives /T all its information
#
TUXCONFIG=${APPDIR}/TUXCONFIG
#
# Set the prefix of the file which is to contain the central user log;
# this should match the ULOGPFX parameter under this site’s LMID in the
# *MACHINES section of the UBBCONFIG file
#
ULOGPFX=${APPDIR}/ULOG
#
# List of directories to search to find view files
#
VIEWDIR=${APPDIR}
#
# List of view files to be used by CBLVIEWC, tmloadcf, etc.
#
VIEWFILES=quote.V,cust.V
#
# Set the COBCPY
#
COBCPY=$TUXDIR/cobinclude
#
# Set the COBOPT
#
COBOPT="-C ANS85 -C ALIGN=8 -C NOIBMCOMP -C TRUNC=ANSI -C OSEXT=cbl"
#
# Set the CFLAGS
#
CFLAGS="-I$TUXDIR/include -I$TUXDIR/sysinclude"
#
# Export all variables just set
#
export TUXDIR APPDIR ENVFILE
export FIELDTBLS FLDTBLDIR TLOGDEVICE
export UBBCBSHM TUXCONFIG ULOGPFX LD_LIBRARY_PATH
export VIEWDIR VIEWFILES COBDIR COBCPY COBOPT CFLAGS
#
# Add TUXDIR/bin to PATH if not already there
#
a="‘echo $PATH | grep ${TUXDIR}/bin‘"
if [ x"$a" = x ]
then
PATH=${TUXDIR}/bin:${PATH}
export PATH
fi
#
# Add APPDIR to PATH if not already there
```

```
#
a="‘echo $PATH | grep ${APPDIR}‘"
if [ x"$a" = x ]
then
PATH=${PATH}:${APPDIR}
export PATH
fi
#
# Add COBDIR to PATH if not already there
#
a="‘echo $PATH | grep ${COBDIR}‘"
if [ x"$a" = x ]
then
PATH=${PATH}:${COBDIR}
export PATH
fi
```

On AIX, `LIBPATH` must be set instead of `LD_LIBRARY_PATH`. On HPUX, `SHLIB_PATH` must be set instead of `LD_LIBRARY_PATH`.

Additional PATH Component for SunOS

If your operating system is SunOS, you need to put `/usr/5bin` at the front of your `PATH`. The following command can be used:

```
PATH=/usr/5bin:$PATH;export PATH
```

Another requirement for SunOS users: use `/bin/sh` rather than `csh` for your shell.

3 STOCKAPP Client Programs

A Look at STOCKAPP Client Programs

This chapter is devoted to the client side of the `STOCKAPP` sample application.

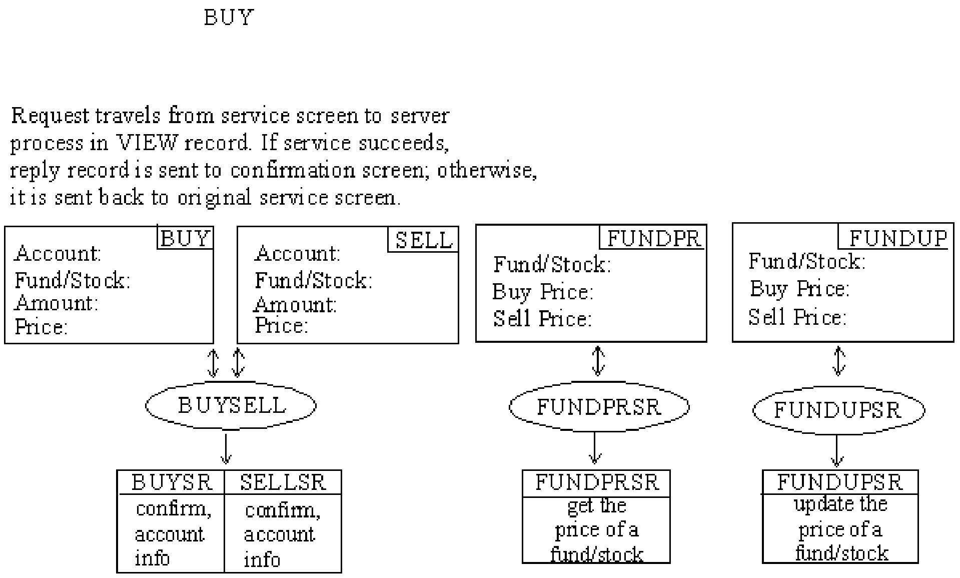
In the client-server architecture of the BEA TUXEDO system, there are two modes of communication:

- Request/response mode, which is characterized by the sending of a single request for a service to be performed by the server and getting back a single response.
- Conversational mode; in this mode a dedicated connection is established between a client (or a server acting like a client) and a server. The connection remains active until terminated. While the connection is active, messages containing service requests and responses can be sent and received between the two participating processes.

System Client Programs

Figure 3-1 shows the hierarchy for `STOCKAPP`. The user selects one of the four service requests. The oval shapes in the illustration represent application services.

Figure 3-1 STOCKAPP Input/Output Hierarchy


BUY
Request travels from service screen to server process in VIEW record. If service succeeds, reply record is sent to confirmation screen; otherwise, it is sent back to original service screen.
BUY
Account:
Fund/Stock:
Amount:
Price:
SELL
Account:
Fund/Stock:
Amount:
Price:
FUNDPR
Fund/Stock:
Buy Price:
Sell Price:
FUNDUP
Fund/Stock:
Buy Price:
Sell Price:
BUYSELL
FUNDPRSR
FUNDUPSR
BUYSR
confirm, account info
SELLSR
confirm, account info
FUNDPRSR
get the price of a fund/stock
FUNDUPSR
update the price of a fund/stock

Record Types

Message records are an essential part of the BEA TUXEDO system, as is the concept of typed records. In the BEA TUXEDO system, a typed record is a record designed to hold a specific data type. Five types are defined: `VIEW`, `STRING`, `CARRAY`, `X_OCTET`, and `X_COMMON`. Applications have the ability to define additional types.

BUY.cbl—A Request/response Client

`BUY.cbl` is an example of a client program. It makes account inquiries that call on the service `BUYSR`. As an executable, it is invoked as follows:

```
BUY
```

BUY.cbl Source Code

Because of space constraints we are not going to print the entire source code of BUY.cbl, but we want to call your attention to the following sections of the program:

```
* Now register the client with the system
* Issue a TPCALL
* Clean up
```

The indicated sections contain all of the places in BUY.cbl where the BEA TUXEDO ATMI calls are used. Note also that BUY.cbl is an example of a program that uses a VIEW typed record and a structure that is defined in the cust file. The source code for the structure can be found in the view description file, cust.V.

Building Client Programs

View description files, of which cust is an example, are processed by the view compiler, viewc(1). viewc has three output files: a COBOL file (CUST.cbl), a binary view description file (cust.V), and a header file (cust.h).

The client programs, BUY.cbl, FUNDPR.cbl, FUNDUP.cbl, and SELL.cbl, are processed by buildclient(1) to compile them and/or link edit them with the necessary BEA TUXEDO libraries.

You can use any of these commands individually, if you choose, but rules for all these steps are included in STOCKAPP.mk.

References

The use of ATMI calls in client programs is covered in Chapter 11, “Writing Client Programs.”

The subject of typed records is covered in Chapter 10, “The BEA TUXEDO System Development Environment,” and Chapter 11, “Writing Client Programs.”

All commands and ATMI calls are described in Sections 1 and 3 of the *BEA TUXEDO Reference Manual*.

CHAPTER

4 STOCKAPP Servers

A Look at STOCKAPP Servers

This chapter describes the servers delivered with `STOCKAPP`, identifies the services coded for the stock application and describes how the services are link edited into servers.

Servers are executable processes that offer one or more services. In the BEA TUXEDO system, they continually accept requests (from processes acting as clients) and dispatch them to the appropriate services. Services are subroutines of COBOL language code written specifically for an application. It is the services accessing a resource manager that provide the functionality for which your BEA TUXEDO system transaction processing application is being developed. Service routines are one part of the application that must be written by the BEA TUXEDO system programmer (user-defined clients being another part).

All the services of `STOCKAPP` use functions provided in the Application Transaction Management Interface (ATMI). These functions allow the services

- to communicate synchronously or asynchronously with other services
- to define global transactions
- to send replies back to clients

This chapter provides

- a description of a service that is part of the stock application
- the relationships between the `STOCKAPP` services and servers
- the `buildserver` command options used to compile and build each server

Service Definitions

There are four services in STOCKAPP. Each STOCKAPP service matches a COBOL function name in the source code of a server as shown in the following list:

BUYSR
: buys a fund/stock record; offered by the BUYSELL server; accepts a VIEW record as input, inserts a CUSTFILE record

SELLSR
: sells a fund/stock record; offered by the BUYSELL server; accepts a VIEW record as input, inserts a CUSTFILE record

FUNDPRSR
: price quote; offered by the PRICEQUOTE server; accepts a VIEW record as input

FUNDUPSR
: fund update; conversational service; offered by FUNDUPDATE server; accepts a VIEW record as input

Building Servers

buildserver is used to put together an executable server. Options identify the names of the output file, the 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 with the -C option invokes the cobcc command. The environment variables ALTCC and ALTCFLAGS 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 illustrated in the examples that follow.

Using the buildserver Command in the STOCKAPP

This section provides the `buildserver` command used in `STOCKAPP.mk` to compile and build each server in the stock application. Refer to the `buildserver`(1) reference page in Section 1 of the *BEA TUXEDO Reference Manual* for complete details.

The BUYSELL Server

The `BUYSELL` server is derived from files that contain the code for the `BUYSR` and `SELLSR` functions. The `BUYSELL` server is first compiled to a `BUYSELL.o` file before supplying it to the `buildserver` command so that any compile-time errors can be clearly identified and dealt with before this step. The `BUYSELL.o` file is created in the following step (done for you in `STOCKAPP.mk`). The `buildserver` command that was used to build the `BUYSELL` server follows:

```
buildserver -C -v -o BUYSELL -s SELLSR -f SELLSR.cbl -s BUYSR -f BUYSR.cbl
```

The explanation of the command line options is as follows:

- The `-C` option is used to build servers with COBOL modules.
- The `-v` option is used to specify the verbose mode. It writes the `cc` command to its standard output.
- The `-s` option is used to specify the service names in the server that are available to be advertised when the server is booted. If the name of the function that performs a service is different from the service name, the function name becomes part of the argument of the `-s` option. In the `STOCKAPP`, the function name is the same as the name of the service so only the service names themselves need to be specified. It is our convention to specify all uppercase for the service name. However, the `-s` option of `buildserver` does allow you to specify an arbitrary name for the processing function for a service within a server. Refer to the `buildserver`(1) manual page for details. It is also possible for the administrator to specify that only a subset of the services that were used to create the server with the `buildserver` command is to be available when the server is booted. For more information, see *Administering the BEA TUXEDO System*.
- 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 `-f` option specifies the files that are used in the link edit phase. Also refer to the `-l` on the `buildserver` manual page. The programming chapters of this guide describe both of these options in some detail, as well. There is a significance to the order in which the files are listed. 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. If these are `.cbl` 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`, the syntax calls for a list enclosed in double quotes. You can use as many `-f` options as you need.
- The `-s` option names the `SELLSR` and `BUYSR` services to be the services that comprise the `BUYSELL` server. The `-o` option assigns the name `BUYSELL` to the executable output file and the `-f` option specifies that the `SELLSR.cbl` and the `BUYSR.cbl` files are to be used in the link edit phase of the build.

Servers Built in STOCKAPP.mk

The preceding section on building a `STOCKAPP` server was included because it is important that you understand how the `buildserver` command is specified. However, in actual practice you are apt to incorporate the build into a makefile and that is the way it is done in `STOCKAPP`. The `STOCKAPP` makefile is discussed in Chapter 5, “The STOCKAPP Makefile.”

References

The writing and debugging of service subroutines using ATMI functions is the main subject of Chapters 12 through 15 of this guide.

Examples of `buildserver` command lines can also be found in these chapters and, of course, in Section 1 of the *BEA TUXEDO Reference Manual*.

CHAPTER

5 The STOCKAPP Makefile

A Look at the STOCKAPP Makefile

STOCKAPP includes a makefile that makes all scripts executable, converts the view description file to binary format, and does all the necessary precompiles, compiles and builds to create the application servers. It can also be used to clean up when you want to make a fresh start.

Editing STOCKAPP.mk

As STOCKAPP.mk is delivered there are a few fields you may want to edit, and some others that may benefit from a little explanation.

TUXDIR

If you look at STOCKAPP.mk, you come to the following comment and to the TUXDIR parameter:

```
#
# Root directory of TUXEDO System. This file must either be edited to set
# this value correctly, or the correct value must be passed via "make -f
# STOCKAPP.mk TUXDIR=/correct/rootdir", or the build of STOCKAPP will fail.
#
TUXDIR=../..
```

The TUXDIR parameter should be set to the absolute pathname of the root directory of your BEA TUXEDO system installation.

APPDIR

You may want to give some thought to the setting of the `APPDIR` parameter. As `STOCKAPP` is delivered, `APPDIR` is set to the directory where the `STOCKAPP` files are located, relative to `TUXDIR`. The section in `STOCKAPP.mk` is as follows:

```
#
# Directory where the STOCKAPP application source and executables live.
# This file must either be edited to set this value correctly, or the
# correct value must be passed via "make -f STOCKAPP.mk
# APPDIR=/correct/appdir", or the build of STOCKAPP will fail.
#
APPDIR=$(TUXDIR)/apps/STOCKAPP
#
```

If you have copied the files to another directory, as is suggested in the `README` file, you should set this parameter to the name of the directory to which you copied the files. When you run the `makefile`, the application will be built in this directory.

Running STOCKAPP.mk

When you have completed the changes you wish to make to `STOCKAPP.mk`, run it with the following command line:

```
nohup make -f STOCKAPP.mk install &
```

Check the `nohup.out` file to make sure the process completed successfully.

CHAPTER

6 Edit STOCKAPP Configuration File

Configuration File for STOCKAPP

A configuration file brings together all the detail about how an application maps to the machines on which it runs. As `STOCKAPP` is delivered, there is a configuration file in the ASCII format described in `ubbconfig`(5). The file called `UBBCBSHM` contains the configuration for an application on a single computer.

The configuration file was delivered with the value of some parameters enclosed in angle brackets (<>). You need to replace these generic values with values that pertain to your installation. All of these fields occur within the `RESOURCES`, `MACHINES` and `GROUPS` sections in the file. In Listing 6-1 we show `UBBCBSHM`. An explanation of the values that need to be replaced follows Listing 6-1.

If you want to enable the application password feature, add this line to the `RESOURCES` section:

```
SECURITY   APP_PW
```

Listing 6-1 Configuration file fields to be replaced

```
      #Copyright (c) 1992 Unix System Laboratories, Inc.
      #All rights reserved
      #Skeleton UBBCONFIG file for the TUXEDO COBOL Sample Application.
      *RESOURCES
      IPCKEY              226164
001   UID                 <user id from id(1)>
002   GID                 <group id from id(1)>
      MASTER              SITE1
      PERM                0660
      MAXACCESSERS        20
      MAXSERVERS          15
      MAXSERVICES         30
      MODEL               SHM
      LDBAL               Y
      MAXGTT              100
      MAXBUFTYPE          16
      MAXBUFSTYPE         32
      SCANUNIT            10
      SANITYSCAN          12
      DBBLWAIT            6
      BBLQUERY            180
      BLOCKTIME           10
      TAGENT              “TAGENT"
      #
      *MACHINES
003   <SITE1’s uname>     LMID=SITE1
004                       TUXDIR="<TUXDIR1>"
005                       APPDIR="<APPDIR1>"
                          ENVFILE="<APPDIR1>/ENVFILE"
                          TUXCONFIG="<APPDIR1>/TUXCONFIG"
                          TUXOFFSET=0
006                       TYPE="<machine type>"
                          ULOGPFX="<APPDIR>/ULOG"
                          MAXWSCLIENTS=5
      #
      *GROUPS
      COBAPI   LMID=SITE1         GRPNO=1
      #
      #
      *SERVERS
      FUNDUPSR SRVGRP=COBAPI      SRVID=1 CONV=Y ENVFILE="<APPDIR1>/ENVFILE"
      FUNDPRSR SRVGRP=COBAPI      SRVID=2 ENVFILE="<APPDIR1>/ENVFILE"
      BUYSELL  SRVGRP=COBAPI      SRVID=3 ENVFILE="<APPDIR1>/ENVFILE"
      #
      #
      *SERVICES
```

Notes to Listing 6-1

The following list describes the nature of the value you must provide for the angle-bracketed values.

Line	Value
001	`UID`—The effective user ID for the owner of the bulletin board IPC structures. In a multiprocessor configuration, the value must be the same on all machines. You avoid problems if this is the same as the owner of the System/T software.
002	`GID`—The effective group ID for the owner of the bulletin board IPC structures. In a multiprocessor configuration, the value must be the same on all machines. Users of the application should share this group ID.
003	`SITE1 name`—The node name of the machine. Use the value produced by the UNIX command: `uname -n`
004	`TUXDIR`—The absolute pathname of the root directory for the BEA TUXEDO system software. Make this a global change to put the value in all occurrences of `<TUXDIR1>` in the file.
005	`APPDIR`—The absolute pathname of the directory where the application runs. Make this a global change to put the value in all occurrences of `<APPDIR1>` in the file.
006	*`machine type`*—This parameter is important in a networked application where machines of different types are present. BEA TUXEDO checks for the value on all communication between machines. Only if the values are different are the message `encode/decode` routines called to convert the data.

References

All of the configuration parameters and their values are described in `ubbconfig`(5) in the *BEA TUXEDO Reference Manual*.

7 Create TUXCONFIG

This chapter describes how to prepare to boot `STOCKAPP`.

As with all the steps since Chapter 1, “Introduction and a Simple Application,” of this guide, you should be in the directory where your `STOCKAPP` files are located and the environment must be set by entering:

```
. ./STKVAR
```

Loading the Configuration File

Once the configuration file has been edited to your satisfaction, it must be loaded to a binary file. The binary configuration file has a file name of `TUXCONFIG`; its pathname relative to `APPDIR` is in the environment variable, `TUXCONFIG`. The file should be created by a person with the effective user ID and group ID of the BEA TUXEDO system administrator, which should be the same as the `UID` and `GID` values in your configuration file. If these conditions are not observed, you may run into permission problems in running `STOCKAPP`. The command line for creating `TUXCONFIG` is:

```
tmloadcf UBBCBSHM
```

There is a `-y` option to suppress prompts that ask if you really want to install `TUXCONFIG` or to overwrite it if it already exists.

If you have specified `SECURITY` as an option for the configuration, `tmloadcf` prompts you to enter an application password. The password you select can be up to 30 characters long. Client processes joining the application will have to come up with the password.

`tmloadcf` parses the ASCII configuration file for syntax errors before it loads it, so if there are errors in the file, the job fails.

References

For instructions on running `tmconfig`, see Chapter 19, “Dynamically Reconfiguring Applications,” in *Administering the BEA TUXEDO System*.

The following page in Section 1 of the *BEA TUXEDO Reference Manual* is important: `tmloadcf`(1).

CHAPTER

8 Boot the Application

This chapter covers booting the application.

Executing tmboot

As with most procedures in this guide, we start by setting the environment. The variables particularly needed by `tmboot` are `TUXCONFIG`, `APPDIR`, and, of course, `TUXDIR`. The command to boot the complete application is simply:

```
tmboot
```

which causes the prompt:

```
Boot all admin and server processes? (y/n): y
```

When you respond `y` to the prompt, you get a running report that starts like this:

```
Booting all admin and server processes in /usr/me/appdir/tuxconfig
Booting all admin processes ...
exec BBL -A:
process id=24223 ... Started.
```

The display continues until all servers in the configuration have been started. It ends with a count of the number started.

There are options that can be used to boot only a portion of the configuration. For example, if the `-A` flag is used, only administrative servers are booted, but with no options specified, everything is booted.

In addition to the report on servers booted, `tmboot` also sends messages to the `ULOG`.

The Userlog: ULOG

We have referred previously to the `ULOG`, but this is the first time it has actually played an important role in the process under discussion. It is called `ULOG` (short for user log) because that is the default prefix; the actual file name of the log is `ULOG` followed by the date in the form: `.`*`mmddyy`*. Log messages can be directed to `ULOG` from user-written modules through a call to `USERLOG`(3cbl), but it is also used heavily by BEA TUXEDO system processes such as `tmboot`.

References

For more information about the `tmboot` command, see Chapter 4, "Starting and Shutting Down Applications," in *Administering the BEA TUXEDO System*.

Chapter 15, "Error Management," contains background information on the user of the userlog. In addition, throughout the guide there are examples of messages being sent to the log.

The following pages in Sections 1 and 3cbl of the *BEA TUXEDO Reference Manual* are important: `tmboot`(1) and `USERLOG`(3cbl).

9 Run STOCKAPP

Run the Application

This chapter covers some of the scripts and commands you can use after `STOCKAPP` has been booted.

We recognize the probability, since you have a system that is active, that you already have set the `STOCKAPP` environment. However, if that is not the case (that is, if you are logging in cold to a running system), you will need to enter the following

```
. ./STKVAR
```

to set your environment for `STOCKAPP.`

Running the audit Client Program

The `BUY.cbl` client program was described in Chapter 3, “STOCKAPP Client Programs” To execute the program, enter the command line as follows:

```
BUY
```

Using tmadmin

This book is not the place to go into an extensive description of the BEA TUXEDO system administrative interface, `tmadmin`. We simply want to encourage you to use it while `STOCKAPP` is running in order to see the kind of information you can produce with `tmadmin` subcommands.

Shutting STOCKAPP Down

When you want to bring `STOCKAPP` down, the command

```
tmshutdown
```

(or the `shutdown` command of `tmadmin`), entered without arguments, will cause all application servers, gateway servers, TMSs, and administrative servers to be shut down and their associated IPC resources to be removed.

The `shutdown` command must be issued from the `MASTER` node.

References

For an extensive discussion on using the `tmadmin` command-line interface for administration, see Chapter 14, “Monitoring a Running System,” in *Administering the BEA TUXEDO System*.

The following pages in Section 1 of the *BEA TUXEDO Reference Manual* are important: `tmadmin`(1) and `tmshutdown`(1).

10 The BEA TUXEDO System Development Environment

Introduction

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 COBOL 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 and your application. The ATMI calls are COBOL calls that have the specific purpose of implementing the communication among application modules running under the control of BEA TUXEDO, 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. The remaining chapters of this book describe how the ATMI calls 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 call to join an application, another to send the data structure to a server and still others to receive the reply.

The operation of a basic client process can be summarized by the pseudo-code shown in Listing 10-1.

Listing 10-1 Pseudo-code for a Client

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

Most of the statements in Listing 10-1 are implemented with ATMI calls. Placing user input in a `DATA-REC` and passing the reply to the user are implemented with COBOL calls.

When client programs are ready to test, you use the `buildclient -C` 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 calls are used.

Server Processes and Service Subroutines

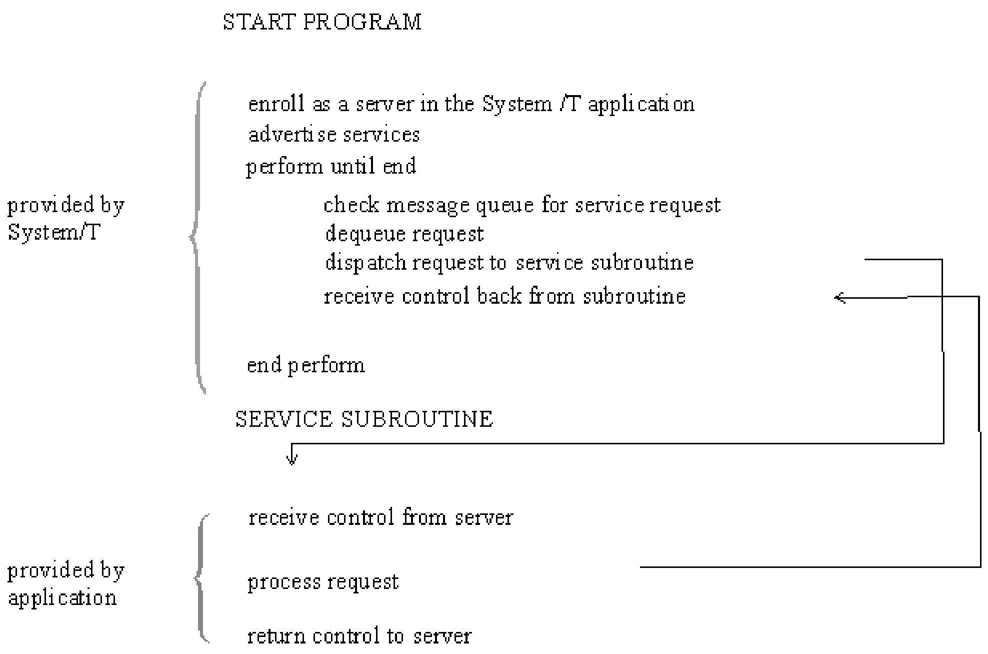
Servers are processes that provide one or more services. They continually check their message queue for service requests and dispatch them to the appropriate service subroutines.

Basic Server Operation

Applications combine their service subroutines with the controlling program that the BEA TUXEDO system provides in order to build server processes. This system supplied controlling program is a set of predefined routines. It performs server initialization and termination and places user input in data structures 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 10-1.

Figure 10-1 Pseudo-code for a Request/Response Server and a Service Subroutine

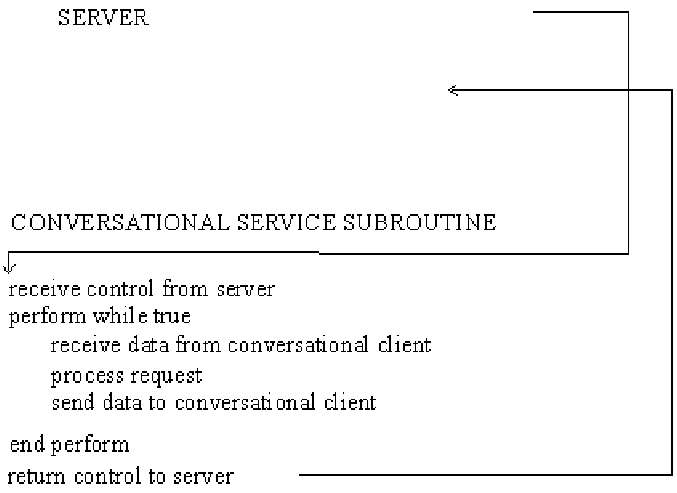


```
START PROGRAM

 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 from subroutine

 end perform
SERVICE SUBROUTINE

 receive control from server

 process request

 return control to server
```

After some initialization a server 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 10-2.

Figure 10-2 Pseudo-code for a 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 controlling program contains the code needed to enroll as a server, advertise services and dequeue request messages. The ATMI calls are used in service subroutines that process requests. When they are ready to compile and test, service subroutines are link edited with the server by means of the `buildserver -C` 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 Calls

The Application-Transaction Monitor Interface is a reasonably compact set of calls used to open and close resources, begin and end transactions, and provide the communication between clients and servers. Table 10-1 summarizes them. Each routine is documented on its own page in the *BEA TUXEDO Reference Manual*.

Table 10-1 ATMI Calls

Group	Name	Operation
Application Interface	`TPINITIALIZE`	join an application client
	`TPTERM`	leave an application client
Request/response	`TPCALL`	send a request, wait for answer
Communication Interface	`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
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

Table 10-1 ATMI Calls

Group	Name	Operation
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	`TPSVCSTART`	start a service
	`TPRETURN`	end service routine
	`TPFORWAR`	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
Events See `EVENTS`(5) and the reference pages listed in the next column.	`TPSUBSCRIBE`	subscribe to events
	`TPPOST`	post events
	`TPUNSUBSCRIB`	unsubscribe events

An Overview of X/Open's TX Interface

In addition to ATMI's transaction management verbs, BEA TUXEDO 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 10-2 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 10-1.

Table 10-2 TX Calls

TX Verbs	Corresponding ATMI Verbs	Main Differences
TXBEGIN	TPBEGIN	Timeout value not passed as argument to TXBEGIN. See TXSETTIMEOUT.
TXCLOSE	TPCLOSE	None
TXCOMMIT	TPCOMMIT	TXCOMMIT can optionally start a new transaction before it returns. This is known as a "chained" transaction.
TXINFORM	TPGETLEV	TXINFORM returns the settings of transaction characteristics set via the three TXSET* routines.
TXOPEN	TPOPEN	None
TXROLLBACK	TPABORT	TXROLLBACK supports chained transactions.
TXSETCOMMITRET	TPSCMT	None
TXSETTRANCTL	None	Defines whether the application is using chained or unchained transactions.
TXSETTIMEOUT	TPBEGIN	Transaction timeout parameter separated from TXBEGIN.

There are two points to keep in mind when using the TX Interface. First, the TX interface requires that TXOPEN 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 TXOPEN before it can use TXBEGIN, TXCOMMIT, and TXROLLBACK to define transactions.

The second rule concerns the default TPSVRINIT and TPSVRDONE routines provided with BEA TUXEDO. If an application writer wants to use the TX Interface in service routines, then the default BEA TUXEDO system TPSVRINIT and TPSVRDONE routines should not be used. This is because these routines call TPOPEN and TPCLOSE which would preclude the use of TX verbs in service routines. Thus, application writers should supply their own TPSVRINIT and TPSVRDONE routines that call TXOPEN and TXCLOSE.

Listing 10-2 is an example of how the TX Interface can be used to support chained transactions. Note that `TXBEGIN` must be used to start the first of a series of chained transactions. Also, note that before calling `TXCLOSE`, the application must switch to unchained transactions so that the last `TXCOMMIT` or `TXROLLBACK` does not start a new transaction.

Listing 10-2 Chained Transaction Example

```
CALL "TXOPEN" USING TX-RETURN-STATUS.
SET TXCHAINED TO TRUE.
CALL "TXSETTRANCTL" USING TX-INFO-AREA
                   TX-RETURN-STATUS.
MOVE 120 TRANSACTION-TIMEOUT.
CALL "TXSETTIMEOUT" USING TX-INFO-AREA
                   TX-RETURN-STATUS.
do forever
           do work as part of transaction.
           if no more work exists
                        SET TXCHAINED TO FALSE.
                        CALL "TXSETTRANCTL" USING TX-INFO-AREA
                                  TX-RETURN-STATUS.
           if work done was successful
                        CALL "TXCOMMIT" USING TX-RETURN-STATUS.
           else
                        CALL "TXROLLBACK" USING TX-RETURN-STATUS.
           if no more work exists
                        leave
end do
CALL "TXCLOSE" USING TX-RETURN-STATUS.
```

Typed Records

Messages are passed to servers in typed records, actually pairs of records. Why “typed?” Well, different types of data require different software to initialize the record, send and receive the data and perhaps encode and decode it, if the record is passed between heterogeneous machines. Records are designated as being of a specific type so the routines appropriate to the record 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) in the *BEA TUXEDO Reference Manual.*

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

The STRING record type allows an arbitrary number of characters which may not contain LOW-VALUE characters anywhere within the record but could be at the end of the record. When sending data, LEN IN *TPTYPE-REC* must contain the number of bytes to be transferred.

The data in a CARRAY record type allows an arbitrary number of characters which may contain LOW-VALUE characters. When sending data, LEN IN *TPTYPE-REC* must contain the number of bytes to be transferred. The X_OCTET record type is equivalent to CARRAY.

The VIEW type is a COBOL data structure that the application defines and for which there has to be a view description file. Records of the VIEW type must have subtypes, that designate individual data structures. The X_COMMON record type is similar to VIEW but is used for both COBOL and C programs so field types should be limited to PIC S9(4) COMP-5, PIC S9(9) COMP-5, and PIC X(any-length). The VIEW32 record type is similar to VIEW but allows for larger character fields, more fields, and larger overall records.

An FML record is a proprietary BEA TUXEDO system type of self-defining buffer where each data field carries its own identifier, an implied 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. The FML function calls are not available from COBOL. COBOL procedures are provided with procedures to initialize an FML record, and convert FML records to/from VIEW records. This is used primarily for applications that have COBOL programs communicating with C programs that use FML records.

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. The BEA TUXEDO system view compiler program, viewc, is used with the -C option to produce one or more COBOL COPY files (one per view) from a source viewfile. These COPY files contain Data Description Records, which may be used in the LINKAGE SECTION or the WORKING STORAGE section of the DATA DIVISION according to the demands of the program.

For FML buffers, a field table file containing descriptions of all fields that may be in the buffer must be available.

Relationship Between 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 COBOL record or vice versa. The COBOL record that is derived from the fielded buffer is referred to as an FML VIEW. FML buffers must be converted to COBOL records for manipulation since the FML are functions not available to COBOL programs. The VIEW is then converted back into an FML buffer for message transmission to a C program that expects an FML buffer.

There are slight differences between a view description of an FML-based view and one that is independent of FML. Listing 10-3 shows a view description file with all of the available data types. Note that the CARRAY1 field has a count of two occurrences and has the "C" count flag to indicate that an additional count element should be created in the record 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 10-3 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 10-3 is shown in Listing 10-4.

Listing 10-4 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 10-5 shows the view description file, similar to the example in Listing 10-3, but for a VIEW independent from FML.

Listing 10-5 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 COBOL application programmer should define float and double fields in the application as COMP-1 and COMP-2, respectively.

The UNIX 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).

Creating COBOL COPY Files from View Descriptions

View description files are source files. To use the `VIEW` in a program, you need a COBOL `COPY` file that defines the data structures in the view. You can create a COBOL `COPY` file from the `myview.v` view description file by invoking the view compiler, `viewc`.

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

Note that the `-n` option is specified only if the `VIEW` is independent of any `FML` definition. `viewc -C` creates three files. One is the COBOL `COPY` file, `MYVIEW.cbl`, another is the header file, `myview.h`, for C routines that share the same view, 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` record is defined.

The COBOL `COPY` file created from `myview.v` is shown in Listing 10-6.

Listing 10-6 Resulting MYVIEW COBOL Copy File

```
*    VIEWFILE: "myview.v"
*    VIEWNAME: "MYVIEW"
  05 FLOAT1                      USAGE IS COMP-1.
  05 DOUBLE1                     USAGE IS COMP-2.
  05 LONG1                       PIC S9(9) USAGE IS COMP-5.
  05 SHORT1                      PIC S9(4) USAGE IS COMP-5.
  05 FILLER                      PIC X(02).
  05 INT1                        PIC S9(9) USAGE IS COMP-5.
  05 DEC1.
    07 DEC-EXP                   PIC S9(4) USAGE IS COMP-5.
    07 DEC-POS                   PIC S9(4) USAGE IS COMP-5.
    07 DEC-NDGTS                 PIC S9(4) USAGE IS COMP-5.
*     DEC-DGTS is the actual packed decimal value
    07 DEC-DGTS                  PIC S9(1)V9(16) COMP-3.
    07 FILLER                    PIC X(07).
  05 CHAR1                       PIC X(01).
  05 STRING1                     PIC X(20).
  05 FILLER                      PIC X(01).
  05 L-CARRAY1 OCCURS 2 TIMES    PIC 9(4) USAGE IS COMP-5.
*    LENGTH OF CARRAY1
  05 C-CARRAY1                   PIC S9(4) USAGE IS COMP-5.
*    COUNT OF CARRAY1
  05 CARRAY1 OCCURS 2 TIMES      PIC X(20).
  05 FILLER                      PIC X(02).
```

COBOL `COPY` files for views must be brought into client programs and service subroutines with `COPY` statements. In Listing 10-6 note that there are some FILLER fields. These are created by the view compiler so that the alignment of fields in `COBOL` matches the alignment in `C`. Also, note the format of the packed decimal value, `DEC1`. It is composed of 5 fields; the `DEC-EXP`, `DEC-POS`, `DEC-NDGTS` and `FILLER` fields are used only in `C` (they are defined in the dec_t type) but are included in the `COBOL` record for filler; they should not be used by the `COBOL` application programmer. The actual packed decimal value is stored in the `DEC-DGTS` value; this is the value that should be set and/or accessed by the `COBOL` programmer. All of the `ATMI` primitives take care of correctly populating the `DEC-DGTS` in packed decimal format before the record is passed to the `COBOL` program from a `C` program, and convert back to the dec_t type when passed from the `COBOL` program to a `C` program. The only restriction is that a `COBOL` program cannot directly pass the record to a `C` function without going through the `ATMI` interface (the decimal formats won't match).

Also note that there is an `L-CARRAY1` length field that occurs twice, once for each occurrence of `CARRAY1` and there is also the `C-CARRAY1` count field.

`viewc` also creates a C version of the header file which can be used if an application desires to mix C and COBOL service and/or client programs.

FML/VIEW Conversion

The `FML` function interface consists of about eighty 16-bit primitives and the same number for `FML32`. This interface was designed for use with the C language. Instead of providing a `COBOL` version of this interface, `COBOL` procedures are provided to convert a received `FML` buffer to a `COBOL` record for processing, and then convert the record back to `FML`.

If a `COBOL` client or server is the originator of an `FML` message, the record must be initialized using the `FINIT` procedure. `FINIT` takes the `FML` record (suitably aligned on a full-word boundary) and `FML-LENGTH` in an `FMLINFO` record which is set to length of the `FML` record. The initialization is shown in Listing 10-7. If an `FML` record is received in a program, it is automatically initialized (unless `TPNOCHANGE` is set). That means that if a program first receives an `FML` record instead of being the originator of the message, it is unnecessary to call `FINIT`.

Listing 10-7 FML/VIEW Conversion

```
WORKING-STORAGE SECTION.
*RECORD TYPE AND LENGTH
 01 TPTYPE-REC.
        COPY TPTYPE.
*STATUS OF CALL
 01 TPSTATUS-REC.
        COPY TPSTATUS.
* SERVICE CALL FLAGS/RECORD
 01 TPSVCDEF-REC.
        COPY TPSVCDEF.
* TPINIT FLAGS/RECORD
 01 TPINFDEF-REC.
        COPY TPINFDEF.
* FML CALL FLAGS/RECORD
 01 FML-REC.
        COPY FMLINFO.
*
*
* APPLICATION FML RECORD - ALIGNED
 01 MYFML.
     05 FBFR-DTA OCCURS 100 TIMES   PIC S9(9) USAGE IS COMP-5.
* APPLICATION VIEW RECORD
 01 MYVIEW.
        COPY MYVIEW.

.....

* MOVE DATA INTO MYVIEW

.....

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

* Convert VIEW to FML Record
 SET FUPDATE TO TRUE.
 MOVE "MYVIEW" TO VIEWNAME.
 CALL "FVSTOF" USING MYFML MYVIEW FML-REC.
 IF NOT FOK
        MOVE "FVSTOF Failed" TO LOGMSG-TEXT
        PERFORM DO-USERLOG
        PERFORM EXIT-PROGRAM
 END-IF.
* CALL THE SERVICE USING THE FML RECORD
```

```
 MOVE "FML" TO REC-TYPE IN TPTYPE-REC.
 MOVE SPACES TO SUB-TYPE IN TPTYPE-REC.
 MOVE LENGTH OF MYFML TO LEN.
 CALL "TPCALL" USING TPSVCDEF-REC
        TPTYPE-REC
        MYFML
        TPTYPE-REC
        MYFML
        TPSTATUS-REC.
 IF NOT TPOK
        MOVE "TPCALL MYFML Failed" TO LOGMSG-TEXT
        PERFORM DO-USERLOG
        PERFORM EXIT-PROGRAM
 END-IF.
* CONVERT THE FML RECORD BACK TO MYVIEW
 CALL "FVFTOS" USING MYFML MYVIEW FML-REC.
 IF NOT FOK
        MOVE "FVFTOS Failed" TO LOGMSG-TEXT
        PERFORM DO-USERLOG
        PERFORM EXIT-PROGRAM
 END-IF.
```

The `FVSTOF` procedure is used to convert an `FML` record to a `VIEW` record. The view is defined by including the copy file generated by the view compiler. The `FML-REC` record provides the `VIEWNAME` and the `FML-MODE` transfer mode which can be set to `FUPDATE, FOJOIN FJOIN` or `FCONCAT`. The actions of these modes are the same as those described in `Fupdate`(3fml), `Fojoin`(3fml), `Fjoin`(3fml), and `Fconcat`(3fml).

The `FVFTOS` procedure is used to convert a `VIEW` record into an `FML` record. The parameters are the same as for `FVSTOF` procedure but the `FML-MODE` need not be set. Fields are copied from the fielded buffer into the structure based on the element descriptions in the view. If a field in the fielded buffer has no corresponding element in the `COBOL` record, it is ignored. If an element specified in the `COBOL` record has no corresponding field in the fielded buffer, a null value is copied into the element. The null value used is definable for each element in the view description. To store multiple occurrences in the `COBOL` record, the record element should be defined with `OCCURS`. If the buffer has fewer occurrences of the field than there are occurrences of the element, the extra element slots are assigned null values. On the other hand, if the buffer has more occurrences of the field than there are occurrences of the element, the surplus occurrences are ignored.

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

Upon successful completion, `FML-STATUS` is set to `FOK`. On error, `FML-STATUS` is set to a non-zero value (see the reference manual pages).

Environment Variables

Environment variables needed either for clients or service routines associated with a server can be set in ENVFILEs that are specified in the configuration file. The environment variables, for example, that need to be set for view descriptions are summarized in Table 10-3.

Table 10-3 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 records
VIEWDIR	colon separated list of directories to be used to find binary view description files can be found	client and server processes using VIEW records

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

The ALTCC and ALTCFLAGS environment variables are used by the buildclient and buildserver commands when run with the -C option for COBOL. You may want to set them in your environment to make compilation of clients and servers more convenient. Set ALTCC to the command that invokes the COBOL compiler. It defaults to cobcc. Set ALTCFLAGS to the link edit flags you may want to use on the compile command line. Setting these variables are optional.

Set COBOPT to the arguments you may want to use on the compile command line. This variable is also optional. Set COBCPY to the directories that contain a set of the COBOL COPY files to be used by the compiler. Set COBDIR to the directory that contains the COBOL compiler software.

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 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 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 `ubbconfig`(5) in the *BEA TUXEDO Reference Manual* and in *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 UNIX shared memory.

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, 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.

CHAPTER

11 Writing Client Programs

Introduction

This chapter describes the ATMI routines that enable a client program to do the following:

- 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 records
- communicate with a service and receive replies in request/response mode
- modify the way a routine performs by specifying various options

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

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

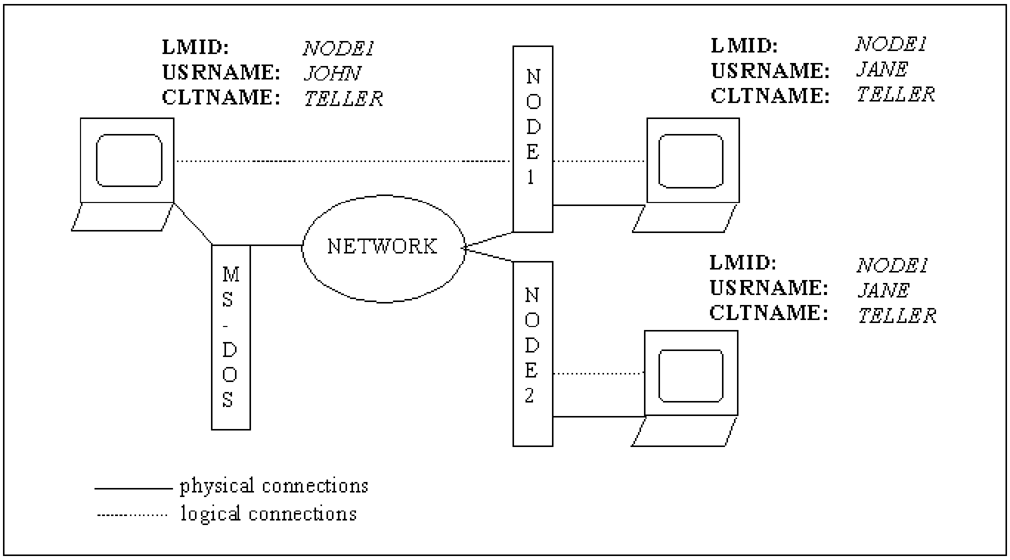
An application can associate both a `USRNAME` and a `CLTNAME` with an execution of a client process. Values furnished for these names are combined by the BEA TUXEDO system with the logical machine identifier (`LMID`) of the machine where the process runs, in order to establish a unique identification 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 `TPINITIALIZE` in a *TPINFDEF-REC* record. 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` and `TPBROADCAST` and detailed statistical information can be gathered via `tmadmin`(1).

Figure 11-1 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 routine.

Figure 11-1 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:

- select settings in the *TPINFDEF-REC* record to select the method used to detect messages
- if you use the dip-in method, call `TPSETUNSOL` to name your message handling routine
- if you use the dip-in method, call `TPCHKUNSOL` to see if any unsolicited messages have been received
- if you use the dip-in method, call `TPGETUNSOL` to get any unsolicited messages

The setting values are described below in “The TPINFDEF-REC Record.” `TPSETUNSOL` and `TPCHKUNSOL` are shown in examples later in this chapter and are described in Section 3cbl of the *BEA TUXEDO Reference Manual.*

Security Strategy

The BEA TUXEDO system provides five incremental levels of security.

Operating System

For platforms that have underlying security mechanisms, this is the first line of defense. The security level is configured to "`NONE`" (configuration is discussed below). This implies, not that there is no security, but that there are no additional mechanisms (for example, the BEA TUXEDO system application 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 such, 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.

The 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 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 *TPINFDEF-REC* record and evaluated when the client calls `TPINITIALIZE` to join the application.

See "Writing Client Programs" in the *BEA TUXEDO Programmer's Guide* for examples of code for handling a password.

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. This 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 "Writing Service Routines" in the *BEA TUXEDO Programmer's Guide*. 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.

A routine, `TPCHKAUTH`, is provided so the level of security can be checked before calling `TPINITIALIZE`. `TPCHKAUTH` returns a value corresponding to:

`TPNOAUTH`

normal UNIX login and file permission security

`TPSYSAUTH`

an application password is required. The client program should place it in the `PASSWD` field of the *TPINFDEF-REC* record.

`TPAPPAUTH`

the application password is required. In addition, the client is expected to provide a value to be passed to the application-specific authentication service in the `DATALEN` field of the *TPINFDEF-REC* record.

The TPINFDEF-REC Record

The *TPINFDEF-REC* record is a special BEA TUXEDO system typed record used by a client program to pass client identification and authentication information to the system as the client attempts to join the application. It is defined in the COBOL `COPY` file and contains the following fields:

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

The USRNAME, CLTNAME and GRPNAME Members of TPINFDEF-REC

`USRNAME`, `CLTNAME` and `GRPNAME` are all strings of up to `MAXTIDENT` characters. `MAXTIDENT` is defined as 30. `USRNAME` is a name representing the caller; you might elect to use the number returned by `getuid`(2). `CLTNAME` is a client name whose semantics are application defined. `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. Currently, `GRPNAME` must be passed as `SPACES`, 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 `TPINITIALIZE` is called and are used for both broadcast notification and the retrieval of administrative statistics.

The PASSWD Member of TPINFDEF-REC

`PASSWD` is a `SPACES` string of up to `MAXTIDENT` characters. It is an application password in unencrypted format that is used by `TPINITIALIZE` for validation against the application password stored in the `TUXCONFIG` file.

The Settings Members of TPINFDEF-REC

The settings members of *TPINFDEF-REC* are 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 settings values 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 setting (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 ATMI calls within application code can access BEA TUXEDO system internal tables via shared memory and 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 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 Member of TPINFDEF-REC

DATALEN is the length of the application-specific data that will be sent to the authentication service. For native clients, it is not encoded by the system; it is passed to the authentication service as the client program provides it. For workstation clients, client authentication is handled by the system; it is passed over the network in encrypted form.

Joining and Leaving an Application

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

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

and

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

Before a client can make any service request, it must join the application. If a service request (or any ATMI routine) is called before invoking TPINITIALIZE, then it is invoked automatically with a SPACES parameter. This implies that the features mentioned above 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 TPINITIALIZE routine. 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 illustrated in Listing 11-1.

Listing 11-1 Typical Client Process Paradigm

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

The arguments to `TPINITIALIZE` are a structure, *TPINFDEF-REC*, that is defined in the COBOL `COPY` file, the user data and a status structure, *TPSTATUS-REC*, that is also defined in the COBOL `COPY` file.

`TPTERM` does not take an argument. Both routines return `TP-STATUS IN` *TPSTATUS-REC* set to `[TPOK]` upon success. On error, the command fails and sets `TP-STATUS`, to a value that indicates the nature of the error. *TPSTATUS-REC* is defined in the COBOL `COPY` file. There is a discussion of the values of `TP-STATUS` in Chapter 15, “Error Management.” The complete list of error codes that can be returned for each of the ATMI routines can also be found on the manual pages that describe the routine and `INTRO`(3cbl) in the *BEA TUXEDO Reference Manual*.

An example of `TPINITIALIZE` and `TPTERM` is shown in Listing 11-2.

Listing 11-2 Joining and Leaving the Application

```
IDENTIFICATION DIVISION.
PROGRAM-ID. FIG1-3.
AUTHOR. TUXEDO DEVELOPMENT.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
*
WORKING-STORAGE SECTION.
*****************************************************
* Tuxedo definitions
*****************************************************
01 TPSTATUS-REC.
COPY TPSTATUS.
*
01 TPINFDEF-REC.
COPY TPINFDEF.
*****************************************************
* Log messages definitions
*****************************************************
01 LOGMSG.
     05 FILLER         PIC X(10) VALUE "FIG12-3 =>".
     05 LOGMSG-TEXT    PIC X(50).
01 LOGMSG-LEN          PIC S9(9) COMP-5.
*
01 USER-DATA-REC  PIC X(75).
******************************************************
PROCEDURE DIVISION.
START-HERE.
MOVE LENGTH OF LOGMSG TO LOGMSG-LEN.
*****************************************************
* Now register the client with the system.
*****************************************************
MOVE SPACES TO USRNAME.
MOVE SPACES TO CLTNAME.
MOVE SPACES TO PASSWD.
MOVE SPACES TO GRPNAME.
MOVE ZERO TO DATALEN.
SET TPU-DIP TO TRUE.
*
CALL "TPINITIALIZE" USING TPINFDEF-REC
         USER-DATA-REC
         TPSTATUS-REC.
IF NOT TPOK
        MOVE "TPINITIALIZE FAILED" TO LOGMSG-TEXT
        PERFORM DO-USERLOG
        PERFORM EXIT-PROGRAM.
*****************************************************
```

```
* Application specific code
*****************************************************
. . .
*****************************************************
*Leave Application
*****************************************************
CALL "TPTERM" USING TPSTATUS-REC.
IF  NOT TPOK
    MOVE "TPTERM FAILED" TO LOGMSG-TEXT
    PERFORM DO-USERLOG.
EXIT-PROGRAM.
STOP RUN.
*****************************************************
* Log messages to the userlog
*****************************************************
DO-USERLOG.
CALL "USERLOG" USING LOGMSG
      LOGMSG-LEN
      TPSTATUS-REC.
```

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

Record Management

Before messages can be sent between processes, a record must be defined for the message data. The following sections describe the record types supported by BEA TUXEDO and how records are tested for type using routines in the ATMI.

Typed Records for Messages

BEA TUXEDO is delivered with eight message record types defined:

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

Note: A ninth type, `X_C_TYPE`, is defined but should not be used from COBOL.

The eight record 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 eight defined types are available to your application programs.

The `tmtypesw.c` file can be edited to add or remove record types. Information about how to do this can be found in *Administering the BEA TUXEDO System*. Only record 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.

Record Types: STRING

The `STRING` record type is what is conventionally understood as a string in the C language. It is an arbitrary number of characters which may not contain `LOW-VALUE` characters anywhere within the record but may be at the end of the record. Data dependent routing is not provided for this record type. If routing routines are desired, they must be written as part of the application. Encoding and decoding is provided for this record type.

Record Types: CARRAY

The `CARRAY` record type (and equivalently `X_OCTET`) is an arbitrary number of characters which may contain `LOW-VALUE` characters. The application defines the semantics; it is not interpreted by BEA TUXEDO. Data dependent routing is not provided for this record type. If routing routines are desired, they must be written as part of the application. No encoding or decoding is provided for a `CARRAY` record when crossing machine boundaries since the bytes are not interpreted by the system.

Record Types: FML and FML32

Records of the FML type are very flexible buffers that hold field identifier/field value pairs. 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.

In C, FML functions are used to manipulate FML typed buffers. These functions are not available in COBOL. However, functions are provided to initialize an FML buffer, to convert FML buffers to COBOL records, VIEWs, and back again.

The FML32 type is similar to the FML type but supports larger character fields, more fields, and larger overall records. It is also used on conversion to/from VIEW32 records. The FML32 buffer type uses environment variables suffixed with “32”, that is, FIELDTBLS32 and FLDTBLDIR32. The primary use of FML32 in COBOL is simply to work with C programs that are using VIEW32 or FML32 typed buffers.

Record Types: VIEW, X_COMMON and VIEW32

Records of the VIEW type (and equivalently X_COMMON) are COBOL data structures that the application defines. The data structure is passed between processes in a VIEW typed record of a specific subtype. The process for defining a VIEW record was described in Chapter 10, “The BEA TUXEDO System Development Environment.” A VIEW can be one derived from a fielded buffer (type FML) or one defined independently of a fielded buffer. The ATMI primitives all take both types of VIEW buffer, 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 Chapter 10, “The BEA TUXEDO System Development Environment.” Both types of VIEW buffer support data dependent routing and automatic encoding and decoding when the buffer is passed between unlike machines.

The comparison of how to create and use the two VIEW types is summarized in Table 11-1.

Table 11-1 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 -C` compiler without the `-n` option to compile the description file	use the `viewc -C` 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 the copy file `FMLINFO`, the copy file created from the view compiler in the programs that define `FML` and `VIEW` buffers	include the copy file created from the view compiler in the programs that define `VIEW` buffers

An `X_COMMON` record should contain only

```
PIC S9(4) COMP-5 (short)
PIC S9(9) COMP-5 (long)
and
PIC (character)
```

fields, which are common to both the COBOL and C languages.

The `VIEW32` record 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”, that is, `FIELDTBLS32`, `FLDTBLDIR32`, `VIEWFILES32`, and `VIEWDIR32`. The primary use of `VIEW32` in COBOL is simply to work with C programs that are using `VIEW32` or `FML32` typed buffers.

Record Types: Summary

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

ATMI Record Calls

It is important for the BEA TUXEDO programmer to know what record types are required and expected by the application. The ATMI routines take `REC-TYPE` and `SUB-TYPE`, both in *TPTYPE-REC*, as arguments. For the types provided by BEA TUXEDO, the `REC-TYPE` specifies the type of record that is to be sent. The `SUB-TYPE` argument has meaning only when `REC-TYPE` is `VIEW`, `VIEW32`, or `X_COMMON`. In this case, the `SUB-TYPE` is the name of the specific data structure defined as a `VIEW`, or `X_COMMON`. In the other record types, the `SUB-TYPE` argument is `SPACES`. `LEN IN` *TPTYPE-REC* specifies the amount of data to send and the amount received.

Service Calls

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

The `TPCALL` routine sends a request to a service subroutine and synchronously waits for its reply.

The `TPACALL` routine sends a request to a service and immediately returns. The reply to the service call is asynchronously received by calling the `TPGETRPLY` routine.

Sending Synchronous Messages: TPCALL

TPCALL is used to send synchronous messages. The syntax of this routine is:

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

TPCALL sends a request to the service that is specified in its first parameter, SERVICE-NAME IN *TPSVCDEF-REC*. The service named in SERVICE-NAME must be one offered in your application. TPCALL waits for the expected reply. It is logically the same as calling the TPACALL routine immediately followed by TPGETRPLY. The request carries the priority that is set by the system for the service specified in SERVICE-NAME unless a different priority has been explicitly set by a call to TPSPRIO.

The parameter of the routine, *IDATA-REC*, contains the data portion of the request and LEN IN *ITPTYPE-REC* specifies how much of *IDATA-REC* to send. Note that the REC-TYPE and SUB-TYPE, both in *ITPTYPE-REC*, must match the type (and subtype) expected by the service routine. If the types do not match, the system sets TP-STATUS to TPEITYPE and the routine call fails.

If the record is a self-defining type, that is, a VIEW, VIEW32, FML, FML32, or X_COMMON record, LEN IN *ITPTYPE-REC* is ignored and can be set to zero. If REC-TYPE IN *ITPTYPE-REC* is STRING and LEN IN *ITPTYPE-REC* is 0, then the request is sent with no data portion. If the request requires no data, set REC-TYPE IN *ITPTYPE-REC* to SPACES. This causes the *IDATA-REC* and LEN IN *ITPTYPE-REC* parameters to be ignored.

The next two parameters indicate the record that is to receive the reply message, *ODATA-REC*, and the length of the reply data, `LEN IN` *OTPTYPE-REC*. If the reply message sent back contains no data portion, upon successful return from `TPCALL`, `LEN IN` *OTPTYPE-REC* will be set to zero, and the contents of the output record will remain unchanged. It is an error for `LEN IN` *OTPTYPE-REC* to be zero on input.

The same record can be used for both the request and reply message. If this is the case, then *ODATA-REC* must be `REDEFINED` to *IDATA-REC*.

Listing 11-3 shows a client program making a synchronous call using the same record for both the request and reply message. Using the same record is appropriate in this particular case, since the `AUDV-REC` message record has been set up to accommodate both request and reply information in the same record. The `B-ID` field is queried by the service but not overwritten and the `BALANCE` field has been initialized to zero in anticipation of the value to be returned by the service. The `SERVICE-NAME` variable represents the service name requested.

Listing 11-3 Using the Same Record for Request and Reply Messages

```
WORKING-STORAGE SECTION.
*****************************************************
* Tuxedo definitions
*****************************************************
  01  TPTYPE-REC.
  COPY TPTYPE.
*
  01 TPSTATUS-REC.
  COPY TPSTATUS.
*
  01  TPSVCDEF-REC.
  COPY TPSVCDEF.
*****************************************************
* Log messages definitions
*****************************************************
  01  LOGMSG.
     05  FILLER           PIC X(6) VALUE  "FIG =>".
     05  LOGMSG-TEXT      PIC X(50).
  01  LOGMSG-LEN          PIC S9(9)  COMP-5.
*
  01  USER-DATA-REC       PIC X(75).
*****************************************************
* This VIEW record (audv) will be sent to the server
*****************************************************
  01 AUDV-REC.
  COPY AUDV.
*
*****************************************************
  PROCEDURE DIVISION.
  START-FIG.
  MOVE LENGTH OF LOGMSG TO LOGMSG-LEN.
****************************************************
*  Prepare the audv record
****************************************************
  MOVE "BRANCH" TO B-ID IN AUDV-REC.
  MOVE 0 TO BALANCE IN AUDV-REC.
  MOVE LENGTH OF AUDV-REC TO LEN.
  MOVE "VIEW" TO REC-TYPE.
  MOVE "audv" TO SUB-TYPE.
  MOVE "SOMESERVICE" TO SERVICE-NAME.
  SET TPBLOCK TO TRUE.
  SET TPNOTRAN TO TRUE.
  SET TPNOTIME TO TRUE.
  SET TPSIGRSTRT TO TRUE.
  SET TPNOCHANGE TO TRUE.
  CALL "TPCALL" USING TPSVCDEF-REC
              TPTYPE-REC
              AUDV-REC
              TPTYPE-REC
              AUDV-REC
              TPSTATUS-REC.
  IF NOT TPOK
         MOVE "Service Failed" TO LOGMSG-TEXT
         PERFORM DO-USERLOG
         PERFORM EXIT-PROGRAM.
  DISPLAY BRANCH and BALANCE
  . . .
```

Note: For an example in which different records are used for input and output see Listing 12-2 in Chapter 12, "Writing Service Routines."

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

Values for the Settings: TPCALL

The last argument that `TPCALL` takes is *TPSTATUS-REC*. The settings in the *TPSTATUS-REC* argument can change the operation of the communication call in some way by allowing additional flexibility to the application. Valid settings are:

`TPNOTRAN`
: If the client process is in transaction mode when it calls `TPCALL`, and the setting is `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 14, "Global Transactions in the BEA TUXEDO System." Either `TPNOTRAN` or `TPTRAN` must be set.

`TPTRAN`
: If the client process is in transaction mode when it calls `TPCALL`, and the setting is `TPTRAN`, the service that is invoked by the call will be part of the transaction; that is, the operations that the service performs are part of the caller's transaction. Either `TPNOTRAN` or `TPTRAN` must be set.

`TPNOCHANGE`
: By using this value, the calling program is indicating that it wants the message returned in the same type of record that was originally defined as the output record. In other words when this setting is set, the type of record returned to the caller must be the same as `REC-TYPE IN` *OTPTYPE-REC* and `SUB-TYPE IN` *OTPTYPE-REC*. This is known as strong record type checking. Either `TPNOCHANGE` or `TPCHANGE` must be set.

`TPCHANGE`
: This setting allows a record type to be different than the original one so long as the caller recognizes the type. In this case, the record type, `REC-TYPE IN` *OTPTYPE-REC*, changes to the received record type. This is known as weak type checking. Either `TPNOCHANGE` or `TPCHANGE` must be set.

`TPNOBLOCK`
: `TPNOBLOCK` concerns the action a routine 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 records 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 routine call fails and `TP-STATUS` is set to `TPETIME`. When the valid setting is

TPNOBLOCK, if a blocking condition exists, the call fails immediately and the request message is not sent. In this case, TP-STATUS is set to TPEBLOCK. Note that TPCALL is a dual routine in that it both sends a request and receives a reply. When TPNOBLOCK is set, it affects only the send part of the routine; if all the server queues are filled or the internal records into which the message records 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 setting does not immunize the call from blocking while it waits. Either TPNOBLOCK or TPBLOCK must be set.

TPBLOCK

When the valid setting is TPBLOCK, if a blocking condition exists, the caller blocks until the condition changes or a timeout occurs, either transaction or blocking. Either TPNOBLOCK or TPBLOCK must be set.

TPNOTIME

By setting 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 setting has no effect; it is subject to the transaction time-out limit. The timing out of transactions will be discussed in Chapter 14, “Global Transactions in the BEA TUXEDO System.” Either TPNOTIME or TPTIME must be set.

TPTIME

TPTIME indicates that you are telling the system to receive the blocking time-out if a blocking condition exists and the blocking time is reached. Either TPNOTIME or TPTIME must be set.

TPSIGRSTRT

Another valid setting is TPSIGRSTRT. This value concerns the action to take if there is a signal interrupt. When TPSIGRSTRT is set, the call is automatically made again. As a result, in the event that a signal interrupts the underlying system call, the routine call is reissued. When TPSIGRSTRT is not set and there is a signal interrupt, the routine call fails and TP-STATUS returns TPGOTSIG. Either TPSIGRSTRT or TPNOSIGRSTRT must be set.

TPNOSIGRSTRT

When TPNOSIGRSTRT is set and there is a signal interrupt, the call is not restarted and the routine call fails. Either TPSIGRSTRT or TPNOSIGRSTRT must be set.

TPCALL sets TP-STATUS to TPOK upon success. On failure, the value of TP-STATUS 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 15, "Error Management." 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 record errors (TPEITYPE, TPEOTYPE), time-out (TPETIME), and protocol errors (TPEPROTO) among others. They are all discussed in Chapter 15, "Error Management," and are listed on the INTRO and TPCALL manual pages. The communication of these failures will also be explained in the discussion of the TPRETURN routine in Chapter 12, "Writing Service Routines."

Examples of the Use of Settings

The next three figures give examples of TPCALL using the communication settings in various scenarios.

The example shown in Listing 11-4 is based on a service which assumes the role of a client when it calls on the services of WITHDRAWAL and DEPOSIT. In the example, we have set the communication setting to TPSIGRSTRT in these service calls to give the transaction a better chance of committing.

Listing 11-4 Sending a Synchronous Message with TPSIGRSTRT Set

```
   WORKING-STORAGE SECTION.
*****************************************************
* Tuxedo definitions
*****************************************************
   01 TPTYPE-REC.
   COPY TPTYPE.
*
   01 TPSTATUS-REC.
   COPY TPSTATUS.
 *
   01 TPSVCDEF-REC.
   COPY TPSVCDEF.
*****************************************************
* This VIEW record (audv) will be sent to the server
*****************************************************
   01 AUDV-REC.
   COPY AUDV.
*
```

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

Listing 11-5 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 11-6 shows a similar example using an asynchronous message call.

Listing 11-5 Sending a Synchronous Message with TPNOTRAN Set

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

In Listing 11-5 where *error processing* has been indicated, it should include printing an error message, aborting the transaction, leaving the application, and exiting the program.

Listing 11-5 also illustrates the use of the `TPNOCHANGE` communication setting to enforce strong record type checking. The strong record type checking, `TPNOCHANGE` is used to force the reply to be returned in a record of type `STRING`. A possible reason for this check is to guard against errors that may occur in the `REPORT` service subroutine in processing the request that could result in a reply record of an incorrect type. Another, 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 `STRING` format without modifying the client process to reflect the change.

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 routine is:

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

The `TPACALL` routine sends a request message to the service named in `SERVICE-NAME IN` *TPSVCDEF-REC* and immediately returns from the call. The three parameters, *DATA-REC*, `LEN IN` *TPTYPE-REC*, and the settings in *TPSTATUS-REC*, have the same semantics as *IDATA-REC*, `LEN IN` *ITPTYPE-REC*, and the settings in *TPSTATUS-REC* of the `TPCALL` routine. Upon successful completion of the call, `TPACALL` returns a value in `COMM-HANDLE IN` *TPSVCDEF-REC* which serves as a communications handle that can be used to get the correct reply for the sent request. While `TPACALL` is in transaction mode (the topic of Chapter 14, “Global Transactions in the 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 Settings: TPACALL

The communication settings that `TPACALL` takes as valid for *TPSTATUS-REC* pertain to the send part of the communication. As a result, the setting value `TPNOCHANGE` is removed since it concerns the output record which is not present in this call, and the values `TPNOREPLY` and `TPREPLY` are 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 set, it signals to `TPACALL` that a reply is not expected. Guidelines for using this setting correctly when a process is in transaction mode are discussed in Chapter 14, “Global Transactions in the

BEA TUXEDO System." When `TPNOREPLY` is set, on success `TPACALL` returns the value of `0` in `COMM-HANDLE`, an invalid communications handle, where `0` cannot be used by `TPGETRPLY`. Either `TPNOREPLY` or `TPREPLY` must be set.

`TPREPLY`

If the value `TPREPLY` is set, it signals to `TPACALL` that a reply is expected. When `TPREPLY` is set, on success `TPACALL` returns a valid communications handle in `COMM-HANDLE`. Either `TPNOREPLY` or `TPREPLY` must be set.

An example of `TPACALL` using the `TPNOREPLY|TPNOTRAN` setting is shown in Listing 11-6. This example is similar to the one presented above in Listing 11-5. In this case, however, a reply is not expected from the `PRINTER` service. By setting both of these settings, the client is indicating that no reply is expected and the `PRINTER` service is not to be a participant in the current transaction. Chapter 15 fully discusses this situation. Refer to the "Transaction Rules" section.

Listing 11-6 Sending an Asynchronous Message with TPNOTRAN or TPNOREPLY

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

On error, `TPACALL` sets `TP-STATUS` to a value that reflects the nature of the error. `TPACALL` returns many of the same error codes as `TPCALL`. 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 15, “Error Management.”

Getting an Asynchronous Reply: TPGETRPLY

`TPGETRPLY` is the complementary routine to `TPACALL`. It receives a reply from a request previously sent by `TPACALL`. The syntax of this routine is:

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

`TPGETRPLY` takes the value of the communication handle returned by `TPACALL` in `COMM-HANDLE IN` *TPSVCDEF-REC*. In the default case, the routine waits for the arrival of the reply that corresponds to the value contained in `COMM-HANDLE`. In waiting for this specific reply, a blocking time-out may occur. A time-out means that `TPGETRPLY` fails and `TP-STATUS` is set to `TPETIME` (unless `TPNOTIME` is set).

The second and third arguments to `TPGETRPLY`, *DATA-REC* and `LEN IN` *TPTYPE-REC*, have identical semantics to those of the *ODATA-REC* and `LEN IN` *OTPTYPE-REC* parameters of the `TPCALL` routine.

Getting and Setting Priority

ATMI provides two routines 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 routines is:

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

and

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

The TPGPRIO routine can be called by a requester after invoking the TPCALL or TPACALL routine to retrieve the priority of the request message just sent. If it was called and no request was sent, the routine fails and sets TP-STATUS to TPENOENT. Upon success, TPGPRIO sets TP-STATUS to TPOK and returns an integer value in the range of 1 to 100, 100 being the highest priority value, in PRIORITY IN *TPPRIDEF-REC*. If the priority has not been explicitly set by using the TPSPRIO routine, 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 11-7 for an example of retrieving the priority of a message that was sent off in an asynchronous call.

Listing 11-7 Determining the Priority of the Sent Request

```
   WORKING-STORAGE SECTION.
*****************************************************
* Tuxedo definitions
*****************************************************
   01 TPTYPE-REC-1.
   COPY TPTYPE.
   01 TPTYPE-REC-2.
   COPY TPTYPE.
*
   01 TPSTATUS-REC.
   COPY TPSTATUS.
*
   01 TPSVCDEF-REC-1.
   COPY TPSVCDEF.
   01 TPSVCDEF-REC-2.
   COPY TPSVCDEF.
*
   01 TPPRIDEF-REC-1.
   COPY TPPRIDEF.
   01 TPPRIDEF-REC-2.
   COPY TPPRIDEF.
*****************************************************
   01 DATA-REC-1     PIC X(100) VALUE SPACES.
   01 DATA-REC-2     PIC X(100) VALUE SPACES.
******************************************************
   PROCEDURE DIVISION.
START-FIG.
   . . .
   join application
   populate DATA-REC1 and DATA-REC2 with send request
   . . .
   MOVE "CARRAY" TO REC-TYPE IN TYPE-REC-1.
   MOVE 100 TO LEN IN TYPE-REC-1.
   MOVE "SERVICE1" TO SERVICE-NAME IN TPSVCDEV-REC-1.
   SET TPTRAN TO TRUE IN TPSVCDEV-REC-1.
   SET TPBLOCK TO TRUE IN TPSVCDEV-REC-1.
   SET TPNOTIME TO TRUE IN TPSVCDEV-REC-1.
   SET TPSIGRSTRT TO TRUE IN TPSVCDEV-REC-1.
   SET TPREPLY TO TRUE IN TPSVCDEV-REC-1.
   CALL "TPACALL" USING TPSVCDEF-REC-1
                  TPTYPE-REC-1
                  DATA-REC-1
                  TPSTATUS-REC.
   IF NOT TPOK
      error processing
   CALL "TPGPRIO" USING TPPRIDEF-REC-1 TPSTATUS-REC
   IF NOT TPOK
      error processing
   MOVE "CARRAY" TO REC-TYPE IN TYPE-REC-2.
   MOVE 100 TO LEN IN TYPE-REC-2.
   MOVE "SERVICE2" TO SERVICE-NAME IN TPSVCDEV-REC-2.
   SET TPTRAN TO TRUE IN TPSVCDEV-REC-2.
   SET TPBLOCK TO TRUE IN TPSVCDEV-REC-2.
   SET TPNOTIME TO TRUE IN TPSVCDEV-REC-2.
   SET TPSIGRSTRT TO TRUE IN TPSVCDEV-REC-2.
   SET TPREPLY TO TRUE IN TPSVCDEV-REC-2.
   CALL "TPACALL" USING TPSVCDEF-REC-2
                  TPTYPE-REC-2
                  DATA-REC-2
                  TPSTATUS-REC.
   IF NOT TPOK
      error processing
   CALL "TPGPRIO" USING TPPRIDEF-REC-2 TPSTATUS-REC
   IF NOT TPOK
      error processing
   IF PRIORITY IN TPSVCDEF-REC-1 >= PRIORITY IN TPSVCDEF-REC-2
            PERFORM DO-GETREPLY1
            PERFORM DO-GETREPLY2
   ELSE
                 PERFORM DO-GETREPLY2
                 PERFORM DO-GETREPLY1
   END-IF.
   . . .
   leave application
DO-GETRPLY1.
   SET TPGETHANDLE TO TRUE IN TPSVCDEV-REC-1.
   SET TPCHANGE TO TRUE IN TPSVCDEV-REC-1.
   SET TPBLOCK TO TRUE IN TPSVCDEV-REC-1.
   SET TPNOTIME TO TRUE IN TPSVCDEV-REC-1.
   SET TPSIGRSTRT TO TRUE IN TPSVCDEV-REC-1.
   CALL "TPGETRPLY" USING TPSVCDEF-REC-1
                    TPTYPE-REC-1
                    DATA-REC-1
                    TPSTATUS-REC.
   IF NOT TPOK
            error processing
DO-GETRPLY2
   SET TPGETHANDLE TO TRUE IN TPSVCDEV-REC-2.
   SET TPCHANGE TO TRUE IN TPSVCDEV-REC-2.
   SET TPBLOCK TO TRUE IN TPSVCDEV-REC-2.
   SET TPNOTIME TO TRUE IN TPSVCDEV-REC-2.
   SET TPSIGRSTRT TO TRUE IN TPSVCDEV-REC-2.
   CALL "TPGETRPLY" USING TPSVCDEF-REC-2
                    TPTYPE-REC-2
                    DATA-REC-2
                    TPSTATUS-REC.
   IF NOT TPOK
              error processing
```

It is also possible to use TPGPRIO to retrieve the priority of the request just received by the service. This is illustrated in Listing 12-3 in Chapter 12, "Writing Service Routines."

With the TPSPRIO routine, 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 of the very next request only that is sent by TPCALL or TPACALL or forwarded by a service subroutine. Forwarding requests will be discussed later in Chapter 12, "Writing Service Routines." This routine takes two parameters, *TPPRIDEF-REC* and *TPSTATUS-REC*, and the second one indicates how the first one is to be interpreted. The first member, PRIORITY IN *TPPRIDEF-REC*, 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 member to be treated as a relative value, the settings must be set to 0. If TPABSOLUTE is set, the priority value of the next request that is sent out will receive the absolute value of the integer contained in PRIORITY. The absolute value of PRIORITY must be in the range of 1 to 100. If the value is not in this range, the system uses the default value, 50. If TPRELATIVE is set, the priority value of the next request is sent out at the relative value of the integer contained in PRIORITY.

Listing 11-8 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 11-8 Setting the Priority of a Request Message

```
    WORKING-STORAGE SECTION.
*****************************************************
* Tuxedo definitions
*****************************************************
    01 TPTYPE-REC.
       COPY TPTYPE.
*
    01 TPSTATUS-REC.
       COPY TPSTATUS.
*
    01 TPSVCDEF-REC.
       COPY TPSVCDEF.
*
    01 TPPRIDEF-REC.
       COPY TPPRIDEF.
*****************************************************
    01 DATA-REC               PIC X(100) VALUE SPACES.
******************************************************
    PROCEDURE DIVISION.
 START-FIG.
    . . .
    join application
    . . .
    MOVE 30 TO PRIORITY.
    SET TPRELATIVE TO TRUE.
    CALL "TPSPRIO" USING TPPRIDEF-REC TPSTATUS-REC
    IF NOT TPOK
       error processing
    MOVE "CARRAY" TO REC-TYPE.
    MOVE 100 TO LEN.
    MOVE "WITHDRAWAL" TO SERVICE-NAME.
    SET TPTRAN TO TRUE .
    SET TPBLOCK TO TRUE .
    SET TPNOTIME TO TRUE .
    SET TPSIGRSTRT TO TRUE .
    SET TPREPLY TO TRUE .
    CALL "TPACALL" USING TPSVCDEF-REC
                   TPTYPE-REC
                   DATA-REC
                   TPSTATUS-REC.
    IF NOT TPOK
       error processing
    . . .
    leave application
```

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 13, "Conversational Clients and Services," describes this topic in detail.

Sending a Broadcast Message

The `TPBROADCAST` routine 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 12, "Writing Service Routines."

Handling Unsolicited Notification

The three routines in this section allow a client to handle unsolicited messages. They are `TPGETUNSOL`, `TPSETUNSOL` and `TPCHKUNSOL`. The syntax for `TPSETUNSOL` is:

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

`TPSETUNSOL` allows a client to identify the routine that should be invoked when an unsolicited message is received by the BEA TUXEDO libraries. Prior to the first call to `TPSETUNSOL`, any unsolicited messages received by the BEA TUXEDO libraries on behalf of the client are logged and ignored. A call to `TPSETUNSOL` with a function number, `CURR-ROUTINE`, set to `0` has the same effect. The method used by the system for notification and detection is determined by the application default, which can be overridden on a per-client basis (see `TPINITIALIZE`).

The routine number passed, in `CURR-ROUTINE`, on the call to `TPSETUNSOL` selects one of 16 predefined routines. The routine names must be `_tm_dispatch1` through `_tm_dispatch8` for C routines that provide unsolicited message handling and `TMDISPATCH9` through `TMDISPATCH16` for COBOL routines that provide the same message handling. The routine `_tm_dispatch1` through `_tm_dispatch8` must conform to the parameter definition described in `tpsetunsol`(3c). Routines `TMDISPATCH9` through `TMDISPATCH16` must use `TPGETUNSOL` to receive the data.

Listing 11-9 is an example of a client setting a COBOL unsolicited function.

Listing 11-9 Setting an Unsolicited Function

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

The syntax of `TPGETUNSOL` is:

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

`TPGETUNSOL` gets unsolicited messages that were sent via `TPBROADCAST` or `TPNOTIFY`. This routine may be called only from an unsolicited message handler.

Upon successful return, `LEN IN` *TPTYPE-REC* contains the actual number of bytes moved into *DATA-REC*. `REC-TYPE` and `SUB-TYPE`, both in *TPTYPE-REC*, contain the data's type and sub-type, respectively. If the message is larger than *DATA-REC*, then *DATA-REC* will contain only as many bytes as will fit in the record. The remainder of the message is discarded and sets `TPTRUNCATE`. If `LEN` is `0`, upon successful completion, then the message has no data portion and *DATA-REC* was not modified.

Listing 11-10 is an example of a COBOL program receiving an unsolicited message.

Listing 11-10 Receiving an Unsolicited Message

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  TMDISPATCH9.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.  USL-486.
 OBJECT-COMPUTER.  USL-486.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
*
 01 TPTYPE-REC.
    COPY TPTYPE.
*
 01 TPSTATUS-REC.
    COPY TPSTATUS.
*
 01  DATA-REC                  PIC X(1000).
*
 PROCEDURE DIVISION.
*
 A-000.
*
   MOVE "CARRAY" TO REC-TYPE.
   MOVE 1000  TO LEN.
   CALL "TPGETUNSOL" USING TPTYPE-REC
                     DATA-REC
                     TPSTATUS-REC.
   IF  NOT TPOK
             error processing
*
   Process message
   DISPLAY "TPGETUNSOL IS TPOK".
   DISPLAY "MESSAGE IS" DATA-REC.
   DISPLAY "LENGTH IS" LEN.
   EXIT PROGRAM.
*
```

The syntax of `TPCHKUNSOL` is:

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

TPCHKUNSOL is used by a client to trigger checking for unsolicited messages. Calls to this routine in a client using signal-based notification do nothing and return immediately. Calls to this routine can result in calls to an application-defined unsolicited message handling routine by the BEA TUXEDO system libraries.

Upon successful completion, TPCHKUNSOL sets TP-STATUS to [TPOK] and returns the number of unsolicited messages dispatched in *MSG-NUM*.

Listing 11-11 is an example of a COBOL program checking for the arrival of an unsolicited message.

Listing 11-11 Arrival of an Unsolicited Message

```
*
* Check for unsolicited messages
*
    CALL "TPCHKUNSOL" USING MESS-NUM
                      TPSTATUS-REC.
*
    IF TPOK
         IF MESS-NUM IS = 0
            No messages were processed by the
            unsolicited function
    ELSE
         MESS-NUM  number of messages were
            processed by the unsolicited function
    END-IF
    ELSE
       process error
   END-IF
```

Compiling Client Programs

To compile your client programs you have several methods to choose from. You can use regular COBOL 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 (`.cbl`) 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. When compiling a `COBOL` client, the `-C` option must be used to indicate that the language is `COBOL`. This ensures that the correct language libraries are included in linking the program.

`buildclient` with the `-C` option invokes the `cobcc` command. The environment variables `ALTCC` and `ALTCFLAGS` can be set to name an alternative compile command and to set flags for the compile and link edit phases. The default value for `ALTCC` is `cobcc`.

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 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 routine 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 `.cbl` and `.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 BUY executable program. The environment variable ALTCC is set to cobcc. The environment variable ALTCFLAGS is set to -I $TUXDIR/include.

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

The buildclient -r Option

The -r option is used to specify which resource manager access libraries should be link edited with the executable client. The choice is specified with a string from the $TUXDIR/udataobj/RM file. Only one string can be specified. The database routines 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 the BEA TUXEDO system, this file must be updated to include the information about the resource manager. For more information, refer to buildtms(1) in the *BEA TUXEDO Reference Manual* and *Administering the BEA TUXEDO System.*

CHAPTER

12 Writing Service Routines

Writing Request/Response Services

The preceding chapter discussed the ATMI calls that can be used to write client programs. In this chapter, some of the same routines are revisited in the context of the service subroutines. As you may recall, services are COBOL subroutines that are linked together with the BEA TUXEDO system-provided controlling program 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 13, “Conversational Clients and Services.”

Note: You have probably noticed that we refer to the service routines described in this chapter as request/response. The service can receive exactly one request and send at most one reply.

Application Service Template

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

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 begin by calling the `TPSVCSTART` routine.
- service routines must terminate by calling either the `TPRETURN` or `TPFORWAR` routine.
- 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 `TPFORWAR`.
- service routines are invoked with *TPSVCDEF-REC*, which is a service information data structure, the user data record, *TPTYPE-REC*, used whenever sending or receiving application data, and *TPSTATUS-REC*, which is used by the ATMI routines for return codes and setting definitions.

The following sections examine these concepts more closely.

The TPSVCSTART Routine

TPSVCSTART is the very first routine to be called when writing a service routine. It is an error to issue any other call within a service routine before calling TPSVCSTART. The syntax of this routine is:

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

The TPSVCDEF-REC Structure

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

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

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

The 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
- give the communications handle, if this is a conversational connection
- provide the client key for authentication
- carry the identifier for the client originating the call

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

The Settings of TPSVCDEF-REC

The TPNOTRAN and TPTRAN settings of the structure are 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 14, “Global Transactions in the BEA TUXEDO System.” If the setting is TPTRAN, it indicates that the service is in transaction mode. When a service is called by TPCALL or TPACALL with a setting of 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 TPNOTRAN, it is possible for TPTRAN to be set. The case that allows this to happen is discussed in Chapter 14, “Global Transactions in the BEA TUXEDO System.”

TPNOREPLY is set if the service was called by TPACALL with the TPNOREPLY communication setting set. It is possible for both the settings to be set. 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 APPKEY Member of TPSVCDEF-REC

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 CLIENTID Member of TPSVCDEF-REC

The `CLIENTID` is used by the system to carry the identification of the client. You should not make changes in this field.

Accessing Data that Comes with the Request

When accessing the request data to be placed in *DATA-REC,* the service must be coded to expect the data to be in a record of the type defined for the service in the configuration file. `LEN IN` *TPTYPE-REC* contains the maximum number of bytes that should be moved. `LEN` is not allowed to be `0` on input.

Upon successful return, *DATA-REC* contains the data received and `LEN` contains the actual number of bytes moved. If the length of the message is greater than *DATA-REC*, *DATA-REC* will receive only as much of the message as possible and `TPTRUNCATE` is set in *TPTYPE-REC*. If `LEN` is `0`, no data was received and *DATA-REC* remains unchanged. `REC-TYPE` and `SUB-TYPE`, both in *TPTYPE-REC,* contain the type and subtype for the service called, which in turn must agree with the typed record as defined for that service in the configuration file.

Listing 12-1 illustrates a typical service definition.

Listing 12-1 Typical Service Definition

```
    IDENTIFICATION DIVISION.
    PROGRAM-ID. BUYSR.
    AUTHOR. TUXEDO DEVELOPMENT.
    ENVIRONMENT DIVISION.
    CONFIGURATION SECTION.
    SOURCE-COMPUTER. USL-486.
    OBJECT-COMPUTER. USL-486.
*
    INPUT-OUTPUT SECTION.
    . . .
******************************************************
* Tuxedo definitions
******************************************************
    01 TPSVCRET-REC.
    COPY TPSVCRET.
*
    01 TPTYPE-REC.
     COPY TPTYPE.
*
    01 TPSTATUS-REC.
    COPY TPSTATUS.
*
    01 TPSVCDEF-REC.
    COPY TPSVCDEF.
******************************************************
* Log message definitions
******************************************************
    01 LOGMSG.
            05 LOGMSG-TEXT        PIC X(50).
*
    01 LOGMSG-LEN                 PIC S9(9) COMP-5.
******************************************************
* User defined data records
******************************************************
    01 CUST-REC.
    COPY CUST.
*
    LINKAGE SECTION.
*
    PROCEDURE DIVISION.
*
 START-BUYSR.
    MOVE LENGTH OF LOGMSG TO LOGMSG-LEN.
    OPEN files or DATABASE
******************************************************
* Get the data that was sent by the client
```

```
******************************************************
    MOVE "Server Started" TO LOGMSG-TEXT.
    PERFORM DO-USERLOG.
    MOVE LENGTH OF CUST-REC TO LEN IN TPTYPE-REC.
    CALL "TPSVCSTART" USING TPSVCDEF-REC
                      TPTYPE-REC
                      CUST-REC
                    T  PSTATUS-REC.
    IF TPTRUNCATE
        MOVE "Input data exceeded CUST-REC length" TO LOGMSG-TEXT
        PERFORM DO-USERLOG
        PERFORM A-999-EXIT.
    IF NOT TPOK
        MOVE "TPSVCSTART Failed" TO LOGMSG-TEXT
        PERFORM DO-USERLOG
        PERFORM A-999-EXIT.
    IF REC-TYPE NOT = "VIEW"
        MOVE "REC-TYPE in not VIEW" TO LOGMSG-TEXT
        PERFORM DO-USERLOG
        PERFORM A-999-EXIT.
    IF SUB-TYPE NOT = "cust"
        MOVE "SUB-TYPE in not cust" TO LOGMSG-TEXT
        PERFORM DO-USERLOG
        PERFORM A-999-EXIT.
     . . .
     set consistency level of the transaction
     . . .
******************************************************
* Exit
******************************************************
  A-999-EXIT.
     MOVE "Exiting" TO LOGMSG-TEXT.
     PERFORM DO-USERLOG.
     SET TPFAIL TO TRUE.
     COPY TPRETURN REPLACING TPSVCRET-REC BY TPSVCRET-REC
     TPTYPE-REC BY TPTYPE-REC
     DATA-REC BY CUST-REC
     TPSTATUS-REC BY TPSTATUS-REC.
******************************************************
* Write to userlog
******************************************************
  DO-USERLOG.
     CALL "USERLOG" USING LOGMSG
              LOGMSG-LEN
              TPSTATUS-REC.
```

In the above example, the request record on the client side was originally sent with REC-TYPE set to VIEW and the SUB-TYPE set to cust. The BUYSR service is defined in the configuration file as a service that knows about the VIEW typed record. BUYSR is able to retrieve the data record by accessing the CUST-REC record as illustrated in the above example. Note that after this record is retrieved and before the first database access is made, the consistency level of the transaction is specified. Refer to Chapter 14, “Global Transactions in the BEA TUXEDO System,” for more details on transaction consistency levels.

Checking The Priority of the Service Request

Listing 12-2 shows the fictitious PRINTSR service testing the priority level of the request just received by invoking the TPGPRIO routine. Based on the priority level, the print job is routed to the appropriate destination printer, RNAME. The contents of INPUT-REC are sent to that printer. Also, the TPSVCDEF-REC settings are 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 12-2 Determining the Priority of the Received Request

```
    IDENTIFICATION DIVISION.
    PROGRAM-ID. PRINTSR.
    AUTHOR. TUXEDO DEVELOPMENT.
    ENVIRONMENT DIVISION.
    CONFIGURATION SECTION.
    SOURCE-COMPUTER. USL-486.
    OBJECT-COMPUTER. USL-486.
*
    INPUT-OUTPUT SECTION.
    . . .
******************************************************
* Tuxedo definitions
******************************************************
    01 TPSVCRET-REC.
    COPY TPSVCRET.
*
    01 TPTYPE-REC.
    COPY TPTYPE.
*
    01 TPSTATUS-REC.
    COPY TPSTATUS.
*
    01 TPSVCDEF-REC.
    COPY TPSVCDEF.
*
    01 TPPRIDEF-REC.
    COPY TPPRIDEF.
******************************************************
* Log message definitions
******************************************************
    01 LOGMSG.
            05 FILLER           PIC S9(9) VALUE
                    "TP-STATUS=".
            05 LOG-TP-STATUS    PIC S9(9).
            05 LOGMSG-TEXT      PIC X(50).
*
    01 LOGMSG-LEN  PIC S9(9) COMP-5.
******************************************************
* User defined data records
******************************************************
    01 INPUT-REC           PIC X(1000).
    01 PRNAME              PIC X(20).
*
    LINKAGE SECTION.
*
    PROCEDURE DIVISION.
*
 START-PRINTSR.
    MOVE LENGTH OF LOGMSG TO LOGMSG-LEN.
    OPEN files or DATABASE
******************************************************
* Get the data that was sent by the client
******************************************************
    MOVE ZERO to TP-STATUS.
    MOVE "Server Started" TO LOGMSG-TEXT.
    PERFORM DO-USERLOG.
    MOVE LENGTH OF INPUT-REC TO LEN.
    CALL "TPSVCSTART" USING TPSVCDEF-REC
                      TPTYPE-REC
                      INPUT-REC
                      TPSTATUS-REC.
    IF NOT TPOK
             MOVE "TPSVCSTART Failed" TO LOGMSG-TEXT
             PERFORM DO-USERLOG
             SET TPFAIL TO TRUE.
             PERFORM A-999-EXIT.
    . . .
    Check other parameters
    CALL "TPGPRIO" USING TPPRIDEF-REC
                   TPSTATUS-REC.
    IF NOT TPOK
             MOVE "TPGPRIO Failed" TO LOGMSG-TEXT
             PERFORM DO-USERLOG
             SET TPFAIL TO TRUE.
             PERFORM A-999-EXIT.
    IF PRIORITY < 20
             MOVE "BIGJOBS" TO RNAME
    ELSE IF PRIORITY < 60
             MOVE "MEDJOBS" TO RNAME
    ELSE
             MOVE "HIGHSPEED" TO RNAME.
    . . .
    Print INPUT-REC on RNAME printer
    . . .
    IF TPNOREPLY
             MOVE SPACES TO REC-TYPE
             MOVE 0 TO LEN
             SET TPSUCCESS TO TRUE
             PERFORM A-999-EXIT
    IF TPREPLY
             MOVE "STRING" TO REC-TYPE
             MOVE LENGTH OF PRNAME TO LEN
             SET TPSUCCESS TO TRUE
             PERFORM A-999-EXIT.
******************************************************
* Exit
******************************************************
 A-999-EXIT.
     MOVE "Exiting" TO LOGMSG-TEXT.
     PERFORM DO-USERLOG.
     SET TPSUCCESS TO TRUE.
     COPY TPRETURN REPLACING TPSVCRET-REC BY TPSVCRET-REC
              TPTYPE-REC buTPTYPE-REC
              DATA-REC BY PRNAME
              TPSTATUS-REC BY TPSTATUS-REC.
******************************************************
* Write to userlog
******************************************************
 DO-USERLOG.
     MOVE TP-STATUS TO LOG-TP-STATUS.
     CALL "USERLOG" USING LOGMSG
               LOGMSG-LEN
               TPSTATUS-REC.
```

The TPRETURN and TPFORWAR Routines

TPRETURN and TPFORWAR are routines 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 routine, 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 the controlling program. If the service has sent requests with asynchronous replies, it must receive all expected replies or invalidate them with TPCANCEL before returning control to the controlling program; otherwise the outstanding replies are automatically dropped when they are received by the BEA TUXEDO system's controlling program, and an error is returned to the caller.

The TPRETURN routine, 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 *ODATA-REC* record. If TPACALL was used to send the request, on success from TPRETURN, the reply message is available in TPGETRPLY's *DATA-REC* record. The syntax of this routine is:

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

Currently the settings are not used.

TPRETURN Arguments: TP-RETURN-VAL IN TPSVCRET-REC

The `TP-RETURN-VAL IN` *TPSVCRET-REC* 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 routine succeeded, and if there is a reply message, it is in the caller's record. If the service terminated unsuccessfully (that is, if the logic of the application set `TP-RETURN-VAL IN` *TPSVCRET-REC* to `TPFAIL`), an error is reported to the client process waiting for the reply. The client's `TPCALL` or `TPGETRPLY` routine call will fail and `TP-STATUS` 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 record. If `TPEXIT` is set in `TP-RETURN-VAL IN` *TPSVCRET-REC*, the functionality of `TPFAIL` is performed, but the server exits after the reply is sent back to the client. Note that if `TP-RETURN-VAL` 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 14, "Global Transactions in the BEA TUXEDO System."

The preceding discussion concerns the effect of `TP-RETURN-VAL` 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 set in `TP-STATUS`. In case of *failed* messages, `TP-STATUS` is set to `TPESVCERR`. This situation overrides the effect of the value of `TP-RETURN-VAL`. If this type of error occurs, no reply data is returned, and the contents of the caller's output record and its length remain unchanged.

If `TPRETURN` sends back a message in a record whose type is not known or not allowed by the caller (that is, the call was made with a setting of `TPNOCHANGE`), `TPEOTYPE` is returned in `TP-STATUS`. Application success or failure cannot be determined and the contents of the caller's output record and its length remain unchanged.

Also, the value returned in `TP-RETURN-VAL` 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 `TP-STATUS.` `TP-STATUS` is set to `TPETIME` and the reply data is not sent leaving the contents and length of the caller's reply record unchanged. There are two types of time-outs in the BEA TUXEDO system. Blocking time-out was discussed when explaining the `TPNOBLOCK` and `TPNOTIME` communication settings. The other type of time-out, transaction time-out, is discussed in Chapter 14, "Global Transactions in the BEA TUXEDO System."

TPRETURN Arguments: APPL-CODE IN TPSVCRET-REC

The `APPL-CODE IN` *TPSVCRET-REC* parameter can be used to return to the caller an application-defined return code. The client can access the value returned in `APPL-CODE` by querying `APPL-RETURN-CODE IN` *TPSTATUS-REC*. This code is sent regardless of application success or failure; that is, it is returned in the case of `TPSUCCESS` or `TPESVCFAIL`. As indicated, no reply messages can be sent in the other error cases.

TPRETURN Arguments: DATA-REC and LEN IN TPTYPE-REC

DATA-REC is the reply message that is to be returned to the client process with the length of the message specified by `LEN IN` *TPTYPE-REC*. If the record is self-defining (for example a `VIEW` record), `LEN` is ignored and can be set to `0`. If `REC-TYPE IN` *TPTYPE-REC* is `STRING` and `LEN` is `0`, then the request is sent with no data portion. If the reply message does not have a data part, `REC-TYPE` is `SPACES`, and *DATA-REC* and `LEN` are ignored. If a reply is expected by the client, and there is no data in the reply record, then a reply with no data portion is sent to the client. If no reply is expected, that is, `TPNOREPLY` was set, `TPRETURN` ignores any data passed to it and simply returns control to the controlling program; the server process is then free to process another request.

TPRETURN Example

Listing 12-3 shows the `TRANSFER` service which makes synchronous calls to the `WITHDRAWAL` and `DEPOSIT` services. If the call to `WITHDRAWAL` should fail, `Cannot withdraw from debit account` is written to the status line of the form, the reply record is freed and the `TP-RETURN-VAL IN` *TPSVCRET-REC* parameter to `TPRETURN` is set to `TPFAIL`. If the call succeeds, the debit balance is retrieved from the reply record.

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

Listing 12-3 How to Use TPRETURN

```
   IDENTIFICATION DIVISION.
   PROGRAM-ID. TRANSFER.
   AUTHOR. TUXEDO DEVELOPMENT.
   ENVIRONMENT DIVISION.
   CONFIGURATION SECTION.
   SOURCE-COMPUTER. USL-486.
   OBJECT-COMPUTER. USL-486.
*
   INPUT-OUTPUT SECTION.
   . . .
*****************************************************
* Tuxedo definitions
*****************************************************
   01  TPSVCRET-REC.
   COPY TPSVCRET.
*
   01  TPTYPE-REC.
   COPY TPTYPE.
*
   01 TPSTATUS-REC.
   COPY TPSTATUS.
*
   01  TPSVCDEF-REC.
   COPY TPSVCDEF.
*****************************************************
* User defined data records
*****************************************************
   01 TRANS-REC.
      COPY TRANS-AMOUNT.
*
    LINKAGE SECTION.
*
    PROCEDURE DIVISION.
*
   START-TRANSFER.
*****************************************************
* Get the data that was sent by the client
*****************************************************
    MOVE LENGTH OF TRANS-REC TO LEN.
    CALL "TPSVCSTART" USING TPSVCDEF-REC
                      TPTYPE-REC
                      TRANS-REC
                      TPSTATUS-REC.
    IF NOT TPOK
             MOVE "Transaction Encountered An Error" TO STATUS-LINE
             SET TPFAIL TO TRUE.
             COPY TPRETURN REPLACING TPSVCRET-REC BY TPSVCRET-REC
                      TPTYPE-REC BY TPTYPE-REC
                      DATA-REC BY TRANS-REC
                      TPSTATUS-REC BY TPSTATUS-REC.
    ELSE
             . . . Check other parameters
******************************************************
* must have a valid debit and credit account number
******************************************************
    CALL "FIND-ACCOUNT-FUNCTION" USING TRANS-DEBIT-ACCOUNT IN  TRANS-REC.

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

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

    IF TRANS-CREDIT-ACCOUNT is not valid
             MOVE "Invalid Credit Account Number"
             TO STATUS-LINE IN TRANS-REC
             SET TPFAIL TO TRUE
             COPY TPRETURN REPLACING
                       DATA-REC BY TRANS-REC.
******************************************************
*  Check amount to transfer
******************************************************
    IF TRANS-AMOUNT IN TRANS-REC < 0
             MOVE "Invalid Transfer Amount Requested"
                       TO STATUS-LINE IN TRANS-REC
             SET TPFAIL TO TRUE
             COPY TPRETURN REPLACING
                       DATA-REC BY TRANS-REC.
******************************************************
*  Make Withdrawal using another service
******************************************************
    MOVE "WITHDRAWAL" TO SERVICE-NAME.
    . . . set other TPCALL parameters
    CALL "TPCALL" USING . . .
    IF NOT TPOK
             MOVE "Cannot withdraw from debit account"
                       TO STATUS-LINE IN TRANS-REC
             SET TPFAIL TO TRUE
             COPY TPRETURN REPLACING
                       DATA-REC BY TRANS-REC.
******************************************************
*  Make Deposit using another service
******************************************************
    MOVE "DEPOSIT" TO SERVICE-NAME.
    . . . set other TPCALL parameters
    CALL "TPCALL" USING . . .
    IF NOT TPOK
             MOVE "Cannot Deposit into credit account"
```

```
                   TO STATUS-LINE IN TRANS-REC
         SET TPFAIL TO TRUE
         COPY TPRETURN REPLACING
                   DATA-REC BY TRANS-REC.
. . .
MOVE "Transfer completed" TO STATUS-LINE IN TRANS-REC
. . . MOVE all the data into TRANS-REC needed by the client
SET TPSUCCESS TO TRUE
COPY TPRETURN REPLACING
                       DATA-REC BY TRANS-REC.
```

Invalidating Handles: TPCANCEL

If a service calling `TPGETRPLY` fails with `TPETIME` and decides not to wait any longer, it can invalidate the handle 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` setting); within a transaction `TPABORT` does the same job of invalidating the transaction communications handle. Listing 12-4 shows the code.

Listing 12-4 Invalidate a Reply after Timing Out

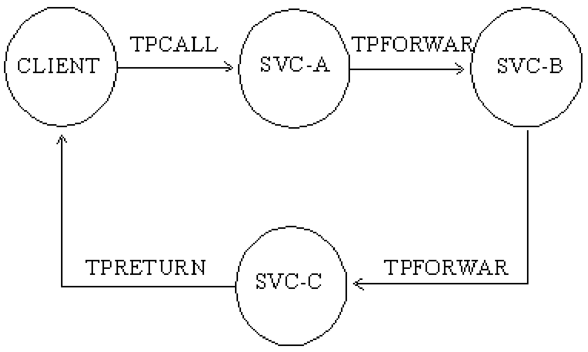
```
. . .  Set up parameters to TPACALL
SET TPNOTRAN TO TRUE.
CALL "TPACALL" USING TPSVCDEF-REC
                   TPTYPE-REC
                   DEBIT-REC
                   TPSTATUS-REC.
IF  NOT TPOK
       error processing
. . .
CALL "TPGETRPLY" USING TPSVCDEF-REC
                   TPTYPE-REC
   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

Forwarding Requests

The TPFORWAR routine 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 12-1 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 routine.

TPFORWAR is identical to TPRETURN in that when it is called, the controlling program regains control, and the server process is free to do more work. The syntax of this routine is:

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

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

TPFORWAR Arguments

The name of the service to which the request is to be forwarded is specified in *TPSVCDEF-REC*. The request record is its third parameter, *DATA-REC*, and the length of the request data is available in `LEN IN` *TPTYPE-REC*. These two parameters share the same meanings as the corresponding ones specified for `TPRETURN`.

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 setting set or forwards the request the call will not fail since the caller is not waiting on itself.

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

TPFORWAR Example

The example in Listing 12-5 is a service routine which shows what the service would look like if it used a call to `TPFORWAR` to send its data record to the `DEPOSIT` service. If the new account is added successfully, the branch record is updated to reflect the new account. On success, the data record gets forwarded to the `DEPOSIT` service. On failure, `TPRETURN` is called with `TP-RETURN-VAL IN` *TPSVCRET-REC* set to `TPFAIL` and the failure reported to the status line of the form.

Listing 12-5 How to Use TPFORWAR

```
    . . .
******************************************************
* Get the data that was sent by the client
******************************************************
    MOVE LENGTH OF TRANS-REC TO LEN.
    CALL "TPSVCSTART" USING TPSVCDEF-REC
                      TPTYPE-REC
                      TRANS-REC
                      TPSTATUS-REC.
    IF NOT TPOK
             MOVE "Transaction Encountered An Error" TO STATUS-LINE
             SET TPFAIL TO TRUE.
             COPY TPRETURN REPLACING
                      DATA-REC BY TRANS-REC.
    ELSE
             . . . Check other parameters
******************************************************
* Insert new account record
******************************************************
    CALL "ADD-NEW-ACCOUNT-FUNCTION" USING TRANS-ACCOUNT IN TRANS-REC.
    IF Adding New Account Failed
             MOVE "Account not added" TO STATUS-LINE IN TRANS-REC
             SET TPFAIL TO TRUE
             COPY TPRETURN REPLACING
                       DATA-REC BY TRANS-REC.
******************************************************
* Forward record to the DEPOSIT service to add initial
* balance into account
******************************************************
    MOVE "DEPOSIT" TO SERVICE-NAME.
    . . . set other TPFORWAR parameters
    COPY TPFORWAR REPLACING
                       DATA-REC BY TRANS-REC.
```

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.

Table 12-1 Unsolicited Messages

	Initiator	Receiver
`TPBROADCAST`	client, server	client
`TABLE`	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 `TPINITIALIZE` and have not yet made a call to `TPTERM`). `TPBROADCAST` can be called in both client and service routines. The syntax of the routine is:

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

`LMID`, `USRNAME`, and `CLTNAME`, all in *TPBCTDEF-REC,* are identifiers used to select the target list of clients. A value of `SPACES` 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-REC* argument identifies the data portion of the message up to the length specified by the `LEN IN` *TPTYPE-REC* argument. If the record is self-defining, for example, a `VIEW` record, `LEN` is ignored and can be set to `0`. The settings can be:

`TPNOBLOCK`
: If a blocking condition exists, don't send the message. Either `TPNOBLOCK` or `TPBLOCK` must be set.

`TPBLOCK`
: The calling program blocks until data is available to receive. Either `TPNOBLOCK` or `TPBLOCK` must be set.

`TPNOTIME`
: Wait indefinitely; do not time out. Either `TPNOTIME` or `TPTIME` must be set.

`TPTIME`
: Timeout if a blocking condition exists and the blocking time is reached. Either `TPNOTIME` or `TPTIME` must be set.

`TPSIGRSTRT`
: When a signal interrupts any underlying system calls, the call is reissued. If this setting is not set, a signal causes `TPBROADCAST` to fail with the `TPGOTSIG` error code. Either `TPSIGRSTRT` or `TPNOSIGRSTRT` must be set.

`TPNOSIGRSTRT`
: When a signal interrupts any underlying system calls, then the interrupted call is not restarted and the call fails. Either `TPSIGRSTRT` or `TPNOSIGRSTRT` must be set.

TPBROADCAST Example

Listing 12-6 shows an example of a call to TPBROADCAST where all clients are targeted. The message to be sent is in a STRING record.

Listing 12-6 Using TPBROADCAST

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

TPNOTIFY Arguments

TPNOTIFY can be called only from a service. The syntax of the routine is:

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

CLIENTID contains a client identifier saved from the TPSVCDEF-REC 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 by 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-REC*, LEN IN *TPTYPE-REC* and settings 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) in the *BEA TUXEDO Reference Manual*). 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) manual 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 routine with a name different from that of the advertised service to be called to process the service request. The BEA TUXEDO administrator can use the advertise and unadvertise commands of tmadmin(1) to control the services offered by servers.

The TPADVERTISE and TPUNADVERTISE routines 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:

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

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

PROGRAM-NAME is the name of a BEA TUXEDO service routine 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. *PROGRAM-NAME* is not permitted to be `SPACES`.

TPADVERTISE Example

Listing 12-7 shows an example of `TPADVERTISE` that is based on the following hypothetical situation:

- SERVER `TLR` is specified to offer only the service `TLRINIT` when booted.
- After some initialization, `TLRINIT` advertises services `DEPOSIT` and `WITHDRAW` both performed by routine `TLRFUNCS`, and both built into the `TLR` server.
- On return from advertising the two services, `TLRINIT` unadvertises itself.

Listing 12-7 Dynamic Advertising and Unadvertising

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

TPUNADVERTISE

TPUNADVERTISE, of course, is called to remove a service from the service table of the bulletin board. The syntax is:

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

The only argument is a name to the *SERVICE-NAME* being unadvertised. An example is included above in Listing 12-7.

System-supplied Servers and Subroutines

The BEA TUXEDO system is delivered with a basic client authentication service: `AUTHSVR`.

System-Supplied Server: AUTHSVR

`AUTHSVR`(5) can be used to provide individual client authentication for an application. It is called by `TPINITIALIZE` when the level of security for the application is `TPAUTH`, `USER_AUTH`, `ACL`, or `MANDATORY_ACL`.

The service in `AUTHSVR` looks in the *USER-DATA-REC* record for a user password (not to be confused with the application password in the `PASSWD` field of the *TPINFDEF-REC* record). The string in *USER-DATA-REC* 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 *USER-DATA-REC* record is sent along by `TPINITIALIZE` 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, `TPINITIALIZE` encrypts the data before sending it across the network.

The BEA TUXEDO System Controlling Program

To speed the development of servers the BEA TUXEDO system provides a predefined controlling program routine for server load modules. This controlling program is automatically included when the `buildserver -C` command is executed.

The predefined controlling 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 controlling program, or a subset specified by the BEA TUXEDO administrator in the configuration file.
- processes command line arguments up to the double dash (`--`) that indicates the end of system-recognized arguments.
- calls the routine `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
  - 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 record for the request message and dispatches the service; that is, calls the service subroutine
  - 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 controlling program 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, record 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 controlling program doing the work of joining and leaving the application, it is an error for services to make calls to the `TPINITIALIZE` or `TPTERM` routines. This error returns `TPEPROTO` in `TP-STATUS.`

In addition to the above functionality, there are two user exits in the controlling program 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 the controlling program, `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 controlling program 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 15, “Error Management.” For now, simply understand that it is a UNIX System file to which processes can write messages by calling the `USERLOG` routine. Coming as it does near the beginning of the system supplied controlling program, `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 the BEA TUXEDO system and the server exits gracefully. `TPSVRINIT` can also start and complete transactions, as discussed in Chapter 14, “Global Transactions in the BEA TUXEDO System.”

The syntax of this routine is:

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

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. The options are passed through *ARGC*, which contains the number of arguments that have been passed, and *ARGV*, which contains the arguments separated by a single `SPACE` character. The predefined controlling program then calls `TPSVRINIT`.

Listing 12-8 shows an example of a `TPSVRINIT` coded to receive command line options.

Listing 12-8 Receiving Command Line Options in TPSVRINIT

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

```
*
 PROCEDURE DIVISION USING CMD-LINE SERVER-INIT-STATUS.
**********************************************************
* ARGC indicates the number of arguments and ARGV contains the
* arguments separated by a single SPACE.
**********************************************************
  A-START.
*
   . . . INSPECT the ARGV line and process arguments
   IF arguments are invalid
         SET TPEINVAL IN SERVER-INIT-STATUS TO TRUE.
   ELSE arguments are OK continue
         SET TPOK IN SERVER-INIT-STATUS TO TRUE.
*
   EXIT PROGRAM.
```

Using TPSVRINIT to Open a Resource Manager

Listing 12-9 shows a code fragment that illustrates another common use of `TPSVRINIT`: opening a resource manager. The BEA TUXEDO system provides a routine to generically open a resource manager, `TPOPEN`. It also provides the complementary routine, `TPCLOSE`. The details of these ATMI calls can be found in Section 3cbl of 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` routine to write to the central event log.

Listing 12-9 Opening a resource manager in TPSVRINIT

```
  IDENTIFICATION DIVISION.
  PROGRAM-ID. TPSVRINIT.
  ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  SOURCE-COMPUTER. USL-486.
  OBJECT-COMPUTER. USL-486.
*
  DATA DIVISION.
  WORKING-STORAGE SECTION.
    01 TPSTATUS-REC.
      COPY TPSTATUS.
    01 LOGMSG         PIC X(50).
    01 LOGMSG-LEN    PIC S9(9) COMP-5.
*
  LINKAGE SECTION.
    01 CMD-LINE.
      05 ARGC PIC 9(4) COMP-5.
      05 ARGV.
           10 ARGS PIC X OCCURS 0 TO 9999 DEPENDING ON ARGC.
    01 SERVER-INIT-STATUS.
     COPY TPSTATUS.
*
  PROCEDURE DIVISION USING CMD-LINE SERVER-INIT-STATUS.
  A-START.
    . . . INSPECT the ARGV line and process arguments
    IF arguments are invalid
              MOVE "Invalid Arguments Passed" TO LOGMSG
              PERFORM EXIT-NOW.
    ELSE arguments are OK continue

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

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 TPCLOSE to close the resource manager in a manner analogous to the way TPSVRINIT and TPOPEN 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 TPCLOSE 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. 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 14, "Global Transactions in the BEA TUXEDO System." The syntax of this routine is:

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

The following example shows the typical way in which TPSVRDONE is used.

Listing 12-10 Closing a resource manager in TPSVRDONE

```
  IDENTIFICATION DIVISION.
  PROGRAM-ID. TPSVRDONE.
  ENVIRONMENT DIVISION.
  CONFIGURATION SECTION.
  SOURCE-COMPUTER. USL-486.
  OBJECT-COMPUTER. USL-486.
*
  DATA DIVISION.
  WORKING-STORAGE SECTION.
   01 TPSTATUS-REC.
      COPY TPSTATUS.
   01 LOGMSG                PIC X(50).
   01 LOGMSG-LEN            PIC S9(9) COMP-5.
   01 SERVER-DONE-STATUS.
      COPY TPSTATUS.
  PROCEDURE DIVISION.
  A-START.
   CALL "TPCLOSE" USING TPSTATUS-REC.
   IF NOT TPOK
```

```
          MOVE "TPCLOSE Failed" TO LOGMSG
   ELSE IF TPESYSTEM
          MOVE "System /T error has occurred" TO LOGMSG
   ELSE IF TPEOS
          MOVE "An Operating System error has occurred" TO LOGMSG
   ELSE IF TPEPROTO
         MOVE "TPCLOSE was called in an improper Context" TO LOGMSG
   ELSE IF TPERMERR
          MOVE "Resource manager Failed to Open" TO LOGMSG
          PERFORM EXIT-NOW.
   SET TPOK IN SERVER-DONE-STATUS TO TRUE.
   EXIT PROGRAM.
  EXIT-NOW.
   SET TPEINVAL IN SERVER-DONE-STATUS TO TRUE
   MOVE 50 LOGMSG-LEN.
   CALL "USERLOG" USING LOGMSG
                  LOGMSG-LEN
                  TPSTATUS-REC.
   EXIT PROGRAM.
```

Compiling Subroutines to Build Servers

To compile your service subroutines you have the same freedom you had in compiling clients. You can use regular COBOL 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 (`.cbl`) files. In any event, when you invoke `buildserver -C` 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` is used to put together an executable server with the BEA TUXEDO systems's controlling program. 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 application in a variety of ways. When compiling a `COBOL` server, the `-C` option must be used to indicate that the language is `COBOL`. This ensures that the correct language libraries are included in linking the program.

`buildserver` invokes the `cobcc` command. The environment variables `ALTCC` and `ALTCFLAGS` can be set to name an alternative compile command and to set settings 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 routine 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 `.cbl` 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 routines 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 the BEA TUXEDO system, this file must be updated to include the information about the resource manager. Refer to the `buildtms`(1) reference page and *Administering the BEA TUXEDO System* for more information.

The buildserver -s Option

The `-s` option is used to specify the service names included in the server and the name of the routines that perform each service. Normally, the routine 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 `BUYSR` service would be processed by routine `BUYSR()`. The following represents the command line to create the `BUYSELL` server.

```
buildserver -C -o BUYSELL \
     -s SELLSR -f SELLSR.cbl \
     -s BUYSR -f BUYSR.cbl
```

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 *Administering the BEA TUXEDO System*.

CHAPTER

13 Conversational Clients and Services

Introduction

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 11, "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 12, "Writing Service Routines,"):

- ♦ It is part of a server identified in the configuration file as offering only conversational services.
- ♦ It is prohibited from invoking `TPFORWAR`.

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`, a setting of `TPEVENT IN` *TPSTATUS-REC*, 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 Communications Handle

A communications handle, `COMM-HANDLE IN` *TPSVCDEF-REC*, is returned when a connection is established with `TPCONNECT` or `TPSVCSTART`. `COMM-HANDLE` is used to identify subsequent message transmissions with a particular conversation. A client or conversational service can have more than one conversation active simultaneously. The maximum number is ten. A client process can have up to ten connections open, all outgoing. A service process can have one incoming connection and up to nine outgoing connections.

Record Management

Data is passed in typed records just as in request/response mode. The record types must be recognized by the application; they must be defined with ATMI routines as described in Chapter 11, “Writing Client Programs.”

Joining an Application

Conversational clients must join the application via a call to `TPINITIALIZE` before attempting to establish a connection to a service. The procedure for joining the application is described in Chapter 11, “Writing Client Programs.”

Establishing a Connection

`TPCONNECT` is the ATMI routine used to set up a conversation. The syntax is

```
01 TPSVCDEF-REC.
     COPY TPSVCDEF.

01 TPTYPE-REC.
     COPY TPTYPE.

01 DATA-REC.
     COPY User Data.

01 TPSTATUS-REC.
     COPY TPSTATUS.

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

`SERVICE-NAME IN` *TPSVCDEF-REC* must contain the name of a service posted in the bulletin board by a conversational server. If `SERVICE-NAME` is not a reference to a conversational service, the call fails and `TP-STATUS IN` *TPSTATUS-REC* 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 through the *DATA-REC* with the length of the data specified by `LEN IN` *TPTYPE-REC*. The `REC-TYPE` and `SUB-TYPE` of the data contained in *DATA-REC* must be a type recognized by the service being called. If no data is being sent, `REC-TYPE` is `SPACES`, and *DATA-REC* and `LEN` are ignored. If the record is self-defining (for example, a VIEW record), `LEN` is ignored and can be set to `0`. The conversational service being called receives the *DATA-REC* and `LEN` 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 values for the settings.

Values for the Settings: TPCONNECT

As with other ATMI routines, the behavior of the called program can be controlled by settings of `TPCONNECT`. Eight of the settings are identical to their use in `TPCALL` and are described in the section titled “Values for the Settings: TPCALL” in Chapter 11. They are:

TPNOTRAN	TPNOBLOCK	TPNOTIME	TPSIGRSTRT
TPTRAN	TPBLOCK	TPTIME	TPNOSIGRSTRT

New valid settings 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 `TPSENDONLY` setting of `TPSENDRECV-FLAG IN` *TPSVCDEF-REC*; `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 `TPRECVONLY` setting of `TPSENDRECV-FLAG IN` *TPSVCDEF-REC*; `TPSENDONLY` and `TPRECVONLY` are mutually exclusive; one or the other must be specified.

As mentioned above, on successful completion `TPCONNECT` returns a `COMM-HANDLE IN` *TPSVCDEF-REC* that is used in all subsequent calls of the conversation. Your call to `TPCONNECT` should be coded something like that shown in Listing 13-1.

Listing 13-1 Establishing a Conversational Connection

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

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

Sending

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 `TPRECVONLY` setting value of the `TPCONNECT` call; `TPRECVONLY` means control is given to the called service. After `TPCONNECT` returns successfully, data is sent across the open connection with the `TPSEND` routine.

The syntax of `TPSEND` is:

```
01 TPSVCDEF-REC.
     COPY TPSVCDEF.

01 TPTYPE-REC.
     COPY TPTYPE.

01 DATA-REC.
     COPY User Data.

01 TPSTATUS-REC.
     COPY TPSTATUS.

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

`COMM-HANDLE IN` *TPSVCDEF-REC* is the communications handle returned by `TPCONNECT` or `TPSVCSTART` that identifies the connection over which to send the data. *DATA-REC* and `LEN IN` *TPTYPE-REC* are, respectively, a structure that contains the data and the length of the data to be sent. The same rules apply to *DATA-REC* and `LEN` that have been outlined earlier: the record must be of a type recognized by the program that receives it and length can be `0` if the record is self-defining. There is no requirement that data be sent.

Values for the Settings: TPSEND

There are eight valid settings for `TPSEND`. The following six settings have the same meanings described in “Values for the Settings: TPCALL” in Chapter 11.

```
TPNOBLOCK          TPNOTIME         TPSIGRSTRT
TPBLOCK            TPTIME           TPNOSIGRSTRT
```

The other settings are like ones that are used in `TPCONNECT`, but have added significance in this routine.

`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. Either `TPRECVONLY` or `TPSENDONLY` must be set.

`TPSENDONLY`
: Signals the intent of the calling program to retain control of the connection. Either `TPRECVONLY` or `TPSENDONLY` must be set.

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 13-2 shows `TPSEND` used in a code fragment.

Listing 13-2 Sending Data in Conversational Mode

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

Receiving

The routine used to receive data sent over an open connection is TPRECV. The syntax is:

```
01 TPSVCDEF-REC.
     COPY TPSVCDEF.

01 TPTYPE-REC.
     COPY TPTYPE.

01 DATA-REC.
     COPY User Data.

01 TPSTATUS-REC.
     COPY TPSTATUS.

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

If the routine is being issued from a subordinate program (that is, not the originator of the connection), COMM-HANDLE, the communications handle, is in the *TPSVCDEF-REC* structure for the program. If TPRECV is being issued by the originator, COMM-HANDLE is the handle returned by TPCONNECT. When the call is made, *DATA-REC* specifies where the data is to be placed, LEN IN *TPTYPE-REC* contains the maximum number of bytes and REC-TYPE IN *TPTYPE-REC* and SUB-TYPE IN *TPTYPE-REC* have the data's type and sub-type. LEN is not allowed to be 0 on input. If it is, the call fails and TP-STATUS is set to TPEINVAL.

Upon successful return, *DATA-REC* contains the data received and LEN contains the actual number of bytes moved. If the length of the message is greater than *DATA-REC*, *DATA-REC* will receive as much of the message as possible and set TPTRUNCATE. If LEN is 0, no data was received and *DATA-REC* remains unchanged.

If an event exists for COMM-HANDLE, TPRECV returns TP-STATUS set to TPEEVENT. The event type is returned in TPEVENT. With events TPESVCSUCC, TPESVCFAIL, and TPESENDONLY data can be received. A more complete discussion of events can be found in “Events and Their Significance” later in this chapter.

Values for the Settings: TPRECV

TPRECV has eight valid settings. Six are described in Chapter 11 (in the section called “Values for the Settings: TPCALL”). They are:

TPNOCHANGE	TPNOTIME	TPSIGRSTRT
TPCHANGE	TPTIME	TPNOSIGRSTRT

The last two valid settings are:

TPNOBLOCK
: The calling program waits for data to arrive to receive it. If data is available, fine; TPRECV gets the data and returns. If data is not available, the call fails and TP-STATUS is set to TPEBLOCK. TPNOBLOCK or TPBLOCK must be set.

TPBLOCK
: The calling program blocks until data is available to receive. TPNOBLOCK or TPBLOCK must be set.

Listing 13-3 shows a fragment of code using TPRECV.

Listing 13-3 Receiving Data in Conversation

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

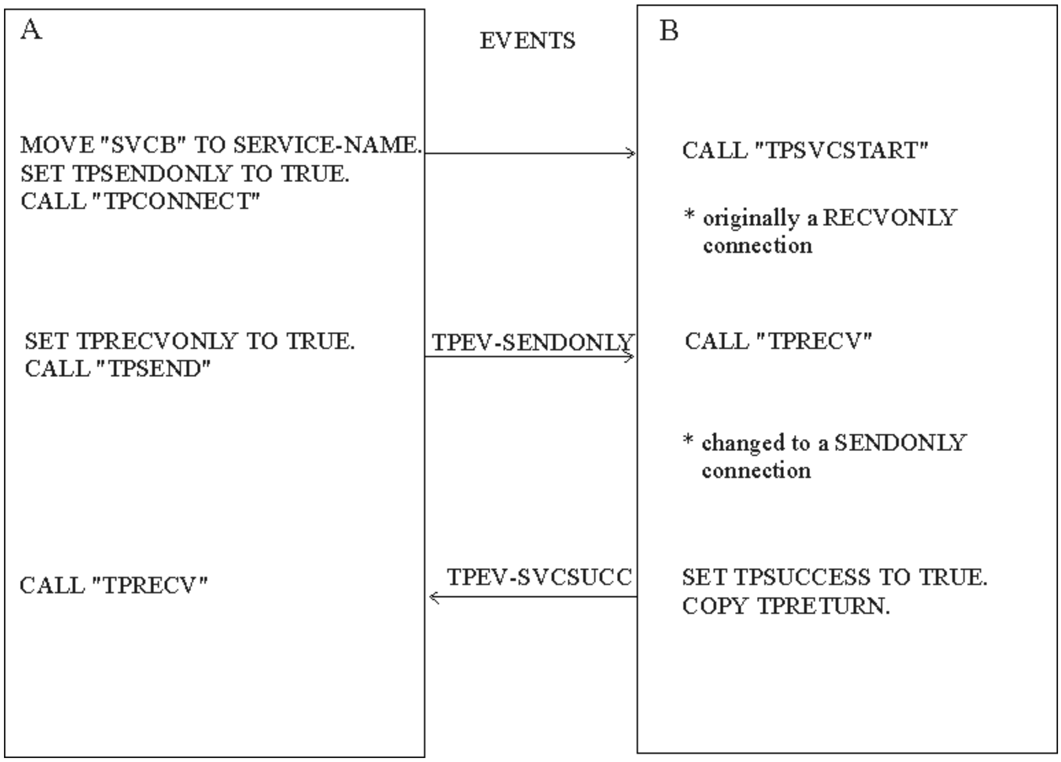
Ending a Conversation

There are three ways in which the connection can be taken down in an orderly fashion and the conversation ended normally. Figure 13-1 and Figure 13-2 show two scenarios that illustrate how conversations are ended when global transactions are not involved. (For a description of ending a conversation when a transaction is involved, see Chapter 14, “Global Transactions in the BEA TUXEDO System.”)

Subordinate Calls TPRETURN

Figure 13-1 shows a simple “A-to-B” conversation. The connection is set up initially with a call to `TPCONNECT` with the `TPSENDONLY` setting of the `TPSENDRECV-FLAG IN` *TPSVCDEF-REC* set. In due course, A turns control of the connection over to B by calling `TPSEND` with a valid setting of `TPRECVONLY.` This generates a `TPEV-SENDONLY` event. The next call by B to `TPRECV` returns `TP-STATUS IN` *TPSTATUS-REC* set to `TPEEVENT` and `TPEVENT` set to `TPEV-SENDONLY`. B knows from the `TPEV-SENDONLY` event that it now controls the connection. Subsequently, B calls `TPRETURN` with `TP-RETURN-VAL IN` *TPSVCRET-REC* set to `TPSUCCESS`. This generates a `TPEV-SVCSUCC` event setting for `TPEVENT IN` *TPSTATUS-REC* 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 be only a server.

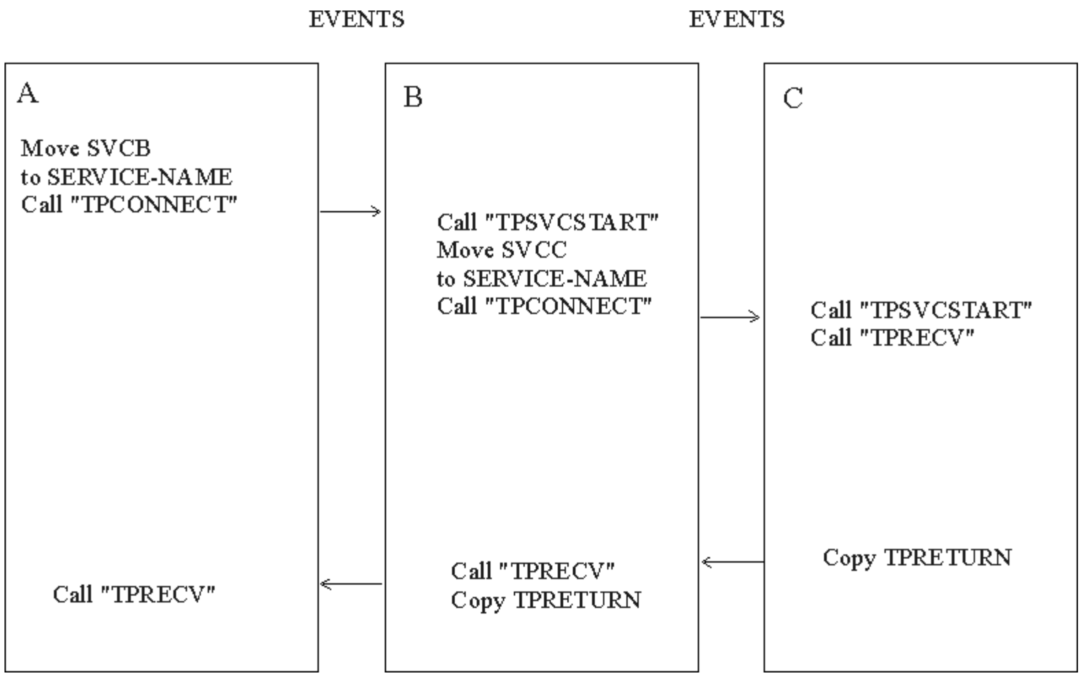
Figure 13-1 Simple SENDONLY Connection and Return



Hierarchy of Connections and TPRETURN

Figure 13-2 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 setting for `TPEVENT IN` *TPSTATUS-REC* 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` setting on the connection to C, turning control of the B-C connection over to C. C can then call `TPRETURN` with `TP-RETURN-VAL IN` *TPSVCRET-REC* set to `TPSUCCESS`, `TPFAIL`, or `TPEXIT`, as appropriate. B can then call `TPRETURN`, setting an event (either `TPEV-SVCSUCC` or `TPEV-SVCFAIL`) for A. Both connections are terminated normally.

Figure 13-2 Connection Hierarchy



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 in which 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 13-1, 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 13-2, above.

Events and Their Significance

There are five events recognized in conversational communication. All five can be posted for TPRECV; three can be posted for TPSEND.Table 13-1 summarizes these events.

Table 13-1 Conversational Communication Events

Event	Received 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.
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 record, 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 name of the `TPDISCON` routine suggests that this routine is the opposite of `TPCONNECT`, but this is not the case; `TPDISCON` 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:

```
01 TPSVCDEF-REC.
     COPY TPSVCDEF.

01 TPSTATUS-REC.
     COPY TPSTATUS.

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

`COMM-HANDLE IN` *`TPSVCDEF-REC`* is the communications handle returned by `TPCONNECT`.

`TPDISCON` generates a `TPEV-DISCONIMM` event setting of `TPEVENT IN` *TPSTATUS-REC* for the service at the other end of the connection and the `COMM-HANDLE` 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 `COMM-HANDLE`, it fails with `TP-STATUS` set to `[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 example of connection hierarchies shown earlier in Figure 13-2, 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 `TPFORWAR`.

Configuration Parameters

Some parameters in the configuration file apply only to conversational processing. As noted in the “Configuration File” section of Chapter 10, “The BEA TUXEDO System Development Environment,” 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.

You need to know about the following parameters:

`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. For an application under development, the default value is probably adequate.

`CONV` = `{ Y | N }`
: is a parameter in the `SERVERS` section. Connections can be made only 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 & 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 ten.

`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 won’t 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 11 and 12, `buildclient` and `buildserver`, 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 cannot use the same name.

CHAPTER

14 Global Transactions in the 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 calls 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` routine allows you to explicitly 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` routine. A process can determine if it is currently working on behalf of a transaction by calling the `TPGETLEV` routine. The rest of this chapter will explain these routines 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 routine calls that are described in Section 3cbl of the *BEA TUXEDO Reference Manual*.

ATMI Transaction Primitives

More specifically, the ATMI routines enable the application programmer to begin and terminate transactions and to test if a client or service routine is currently in a transaction. The ATMI calls, `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` routine. A two-phase commit protocol is used by the BEA TUXEDO system to coordinate the commitment, rollback, and recovery of global transactions. This protocol will be further discussed later in the chapter.

When the `TPGETLEV` routine is invoked, it returns a setting in *TPTRXLEV-REC* that indicates if the caller is within a transaction `(TP-IN-TRAN)` or not `(TP-NOT-IN-TRAN)`.

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` routine. 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 routines have the following syntax:

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

```
*
   CALL "TPBEGIN" USING TPTRXDEF-REC TPSTATUS-REC.
*
   CALL "TPCOMMIT" USING TPTRXDEF-REC TPSTATUS-REC.
*
   CALL "TPABORT" USING TPTRXDEF-REC TPSTATUS-REC.
```

A high-level view of defining a transaction is shown in Listing 14-1.

Listing 14-1 Delineating a Transaction

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

The 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 routines. Any process may call `TPBEGIN` except if

- it is already in transaction mode or
- it is waiting for any outstanding replies.

With reference to the second point, it is an error to make the sequence of calls shown in Listing 14-2.

Listing 14-2 Error - Starting a Transaction with an Outstanding Reply

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

If `TPBEGIN` is called with either of these two conditions existing, the call will fail because of an error in protocol and `TP-STATUS` 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 explicitly set 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. The value of T-OUT 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 T-OUT 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 human 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 TP-STATUS as illustrated in Listing 14-3. 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 in TP-STATUS and the transaction is implicitly aborted.

The value assigned to the T-OUT 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 T-OUT 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 T-OUT may need to be coordinated with your system administrator to be sure it makes sense with regard to the system parameters.

Listing 14-3 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 14-3 Testing for Transaction Time Out

```
. . .
MOVE 30 TO T-OUT.
CALL "TPBEGIN" USING TPTRXDEF-REC TPSTATUS-REC.
IF  NOT TPOK
    MOVE "Failed to BEGIN a transaction" TO LOG-REC-TEXT.
    MOVE 29 to LOG-REC-LEN
    CALL "USERLOG" USING
                   LOG-REC-TEXT
                   LOG-REC-LEN
                   TPSTATUS-REC
    CALL "TPTERM" USING
                  TPSTATUS-REC
    PERFORM A-999-EXIT.
. . .
       communication CALL statements
. . .
IF  TPETIME
    CALL "TPABORT" USING
                   TPTRXDEF-REC
                   TPSTATUS-REC
IF  NOT TPOK
       error processing
ELSE
   CALL "TPCOMMIT" USING
                   TPTRXDEF-REC
                   TPSTATUS-REC
   IF  NOT TPOK
        error processing
```

Note 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 15, “Error Management.”

The example in Listing 14-4 illustrates how to define a transaction.

Listing 14-4 Defining a Transaction

```
 DATA DIVISION.
 WORKING-STORAGE SECTION.
*
 01  TPTYPE-REC.
 COPY TPTYPE.
*
 01 TPSTATUS-REC.
 COPY TPSTATUS.
*
 01 TPINFDEF-REC.
 COPY TPINFDEF.
*
 01  TPSVCDEF-REC.
 COPY TPSVCDEF.
*
 01  TPTRXDEF-REC.
 COPY TPTRXDEF.
*
 01 LOG-REC              PIC X(30) VALUE " ".
 01 LOG-REC-LEN          PIC S9(9)  COMP-5.
*
 01 USR-DATA-REC         PIC X(16).
*
 01 AUDV-REC.
     05  AUDV-BRANCH-ID      PIC S9(9) COMP-5.
     05  AUDV-BALANCE        PIC S9(9) COMP-5.
     05  AUDV-ERRMSG         PIC X(60).
*
 PROCEDURE DIVISION.
*
 A-000.
  . . .
* Get Command Line Options  set Variables (Q-BRANCH)
  MOVE SPACES TO USRNAME.
  MOVE SPACES TO CLTNAME.
  MOVE SPACES TO PASSWD.
  MOVE SPACES TO GRPNAME.
  CALL "TPINITIALIZE" USING TPINFDEF-REC
                      USR-DATA-REC
                      TPSTATUS-REC.
  IF  NOT TPOK
      MOVE "Failed to join application" TO LOG-REC
      MOVE 26 TO LOG-REC-LEN
       CALL "USERLOG" USING LOG-REC
                      LOG-REC-LEN
```

```
                        TPSTATUS-REC
            PERFORM A-999-EXIT.
      * Start global transaction
        MOVE 30 TO T-OUT.
        CALL "TPBEGIN" USING TPTRXDEF-REC TPSTATUS-REC.
        IF  NOT TPOK
            MOVE 29 to LOG-REC-LEN
            MOVE "Failed to begin a transaction" TO LOG-REC
            CALL "USERLOG" USING LOG-REC
                           LOG-REC-LEN
                           TPSTATUS-REC
            PERFORM DO-TPTERM.
      * Set up record
        MOVE Q-BRANCH TO AUDV-BRANCH-ID.
        MOVE ZEROS TO AUDV-BALANCE.
        MOVE SPACES TO AUDV-ERRMSG.
      * Set up TPCALL records
        MOVE "GETBALANCE" TO SERVICE-NAME.
        MOVE "VIEW" TO REC-TYPE.
        MOVE LENGTH OF AUDV-REC TO LEN.
        SET TPBLOCK TO TRUE.
        SET TPTRAN IN TPSVCDEF-REC TO TRUE.
        SET TPNOTIME TO TRUE.
        SET TPSIGRSTRT TO TRUE.
        SET TPCHANGE TO TRUE.
      *
        CALL "TPCALL" USING TPSVCDEF-REC
                      TPTYPE-REC
                      AUDV-REC
                      TPTYPE-REC
                      AUDV-REC
                      TPSTATUS-REC.
        IF  NOT TPOK
            MOVE 19 to LOG-REC-LEN
            MOVE "Service call failed" TO LOG-REC
            CALL "USERLOG" USING LOG-REC
                           LOG-REC-LEN
                           TPSTATUS-REC
            PERFORM DO-TPABORT
            PERFORM DO-TPTERM.
      * Commit global transaction
        CALL "TPCOMMIT" USING TPTRXDEF-REC
                        TPSTATUS-REC
        IF  NOT TPOK
            MOVE 16 to LOG-REC-LEN
            MOVE "Failed to commit" TO LOG-REC
            CALL "USERLOG" USING LOG-REC
                      LOG-REC-LEN
                      TPSTATUS-REC
            PERFORM DO-TPTERM.
      * Show results only when transaction has completed successfully
```

```
  DISPLAY "BRANCH=" Q-BRANCH.
  DISPLAY "BALANCE=" AUDV-BALANCE.
  PERFORM DO-TPTERM.
* Abort the transaction
 DO-TPABORT.
  CALL "TPABORT" USING TPTRXDEF-REC
                 TPSTATUS-REC
  IF  NOT TPOK
      MOVE 26 to LOG-REC-LEN
      MOVE "Failed to abort transaction" TO LOG-REC
      CALL "USERLOG" USING LOG-REC
                     LOG-REC-LEN
                     TPSTATUS-REC.
* Leave the application
 DO-TPTERM.
  CALL "TPTERM" USING TPSTATUS-REC.
  IF  NOT TPOK
      MOVE 27 to LOG-REC-LEN
      MOVE "Failed to leave application" TO LOG-REC
      CALL "USERLOG" USING LOG-REC
                     LOG-REC-LEN
                     TPSTATUS-REC.
                     EXIT PROGRAM.
*
 A-999-EXIT.
*
 EXIT PROGRAM.
```

Terminating the Transaction

As already indicated, a transaction is terminated by a call to either `TPCOMMIT` or `TPABORT`. 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 communications handles of any outstanding replies. 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 replies outstanding

If either condition is not true, the call fails and `TP-STATUS` 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 replies, the transaction is aborted and those reply descriptors become invalid.

TPCOMMIT Initiates the 2-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 TP-RETURN-VAL parameter that can be set to either TPSUCCESS or TPFAIL. If TPFAIL has been returned, TPCOMMIT fails, TP-STATUS 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 15, “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 routine TPSCMT can be called with *TPCMTDEF-REC* 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

Listing 14-5 shows a client making a synchronous call to a fictitious `REPORT` service (line 24). It demonstrates testing for errors that can be returned on a communication call that indicate participant failure (lines 30-42).

Listing 14-5 Testing for Participant Success or Failure

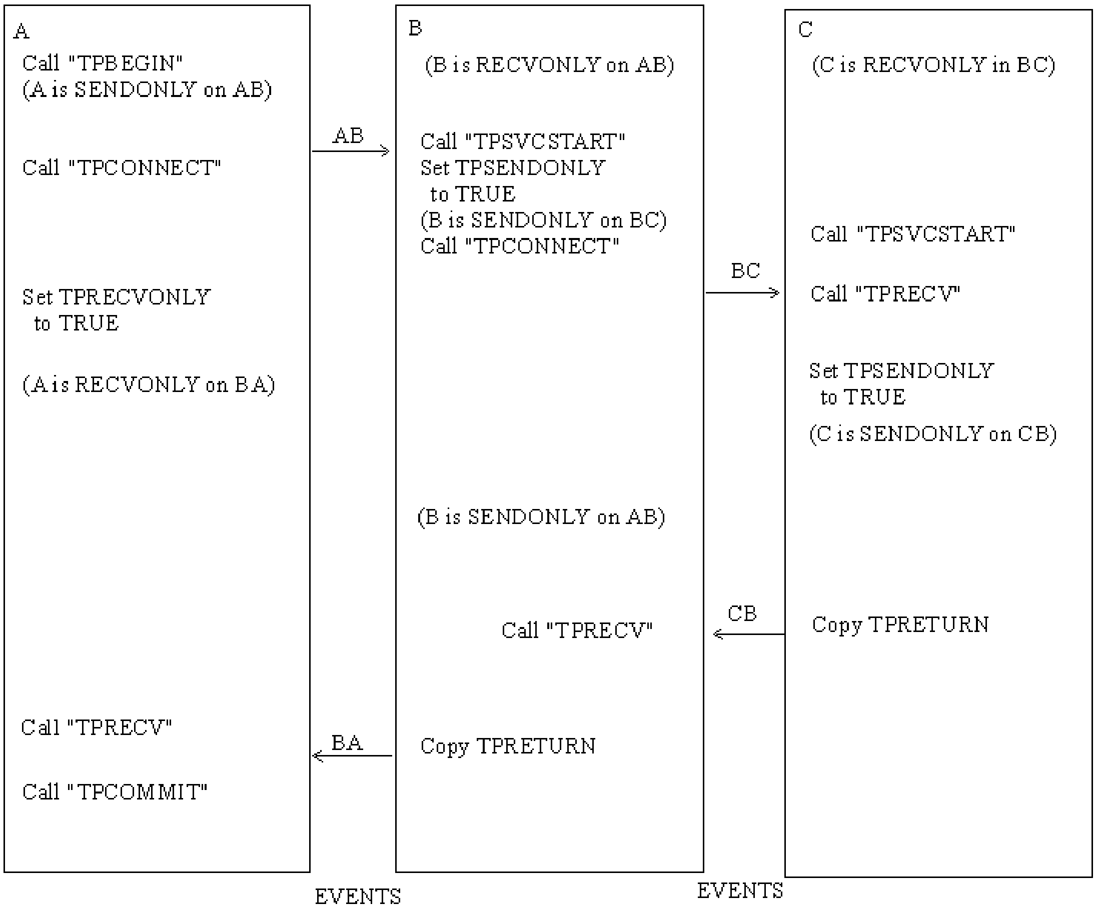
```
01   . . .
02   CALL "TPINITIALIZE" USING TPINFDEF-REC
03   USR-DATA-REC
04   TPSTATUS-REC.
05   IF  NOT TPOK
06       error message,
07       EXIT PROGRAM .
08   MOVE 30 TO T-OUT.
09   CALL "TPBEGIN" USING TPTRXDEF-REC TPSTATUS-REC.
10   IF  NOT TPOK
11       error message,
12       PERFORM DO-TPTERM.
13  * Set up record
14   MOVE "REPORT=accrcv DBNAME=accounts" TP-RECORD.
15   MOVE 27 TO LEN.
16   MOVE "REPORTS" TO SERVICE-NAME.
17   MOVE "STRING" TO REC-TYPE.
18   SET TPBLOCK TO TRUE.
19   SET TPTRAN IN TPSVCDEF-REC TO TRUE.
20   SET TPNOTIME TO TRUE.
21   SET TPSIGRSTRT TO TRUE.
22 SET TPCHANGE TO TRUE.
23  *
24   CALL "TPCALL" USING TPSVCDEF-REC
25                 TPTYPE-REC
26                 TP-RECORD
27                 TPTYPE-REC
28                 TP-RECORD
29                 TPSTATUS-REC.
30   IF  TPOK
31        PERFORM DO-TPCOMMIT
32        PERFORM DO-TPTERM.
33  * Check return status
34    IF  TPESVCERR
35    DISPLAY "REPORT service’s TPRETURN encountered problems"
36    ELSE IF  TPESVCFAIL
37    DISPLAY "REPORT service FAILED with return code=" APPL-RETURN-CODE
38    ELSE IF  TPEOTYPE
```

```
  DISPLAY "REPORT service's reply is not of any known REC-TYPE"
*
  PERFORM DO-TPABORT
  PERFORM DO-TPTERM.
* Commit global transaction
 DO-TPCOMMIT.
  CALL "TPCOMMIT" USING TPTRXDEF-REC
                  TPSTATUS-REC
  IF  NOT TPOK
      error message
* Abort the transaction
 DO-TPABORT.
  CALL "TPABORT" USING TPTRXDEF-REC
                 TPSTATUS-REC
  IF  NOT TPOK
      error message
* Leave the application
 DO-TPTERM.
  CALL "TPTERM" USING TPSTATUS-REC.
  IF  NOT TPOK
      error message
  EXIT PROGRAM.
```

Committing a Transaction in Conversational Mode

Figure 14-1 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 14-1, each subordinate service must call `TPRETURN` when it no longer has replies outstanding. A `TPEV-SVCSUCC` or `TPEV-SVCFAIL` event setting for `TPEVENT IN` *TPSTATUS-REC* 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 14-1 Connection Hierarchy: Transaction Mode



Implicitly Defining a Global Transaction

Besides using the ATMI calls explicitly to start and end a transaction, it is possible for a global transaction to be started in a service routine. 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 set `TPTRAN-FLAG IN` *TPSVCDEF-REC* 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 `AUTOTRAN=N` (or not set), the called process is not placed in transaction mode.
  - ♦ if `AUTOTRAN=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 setting of `TPNOTRAN` and the setting member of the service information structure to return `TPTRAN` when queried.

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 setting of `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 `TP-STATUS` to `TPEPROTO` to indicate that the routine 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."

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 settings 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` routine. The syntax of this routine is:

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

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

Listing 14-6 is an example of a service that shows testing for transaction level using the `TPGETLEV` routine (line 3). If the process is not in transaction mode, it starts one (line 5). If `TPBEGIN` fails, a message is returned to the status line (line 9) and `APPL-CODE IN` *TPSVCRET-REC* is set to a code that can be retrieved in `APPL-RETURN-CODE IN` *TPSTATUS-REC* (line 11 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 routines. For example, in the fragment shown in Listing 14-6, if the 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 (line 4).

Listing 14-6 Testing Transaction Level

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

CHAPTER

15 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 routine 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 routine calls 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 calls affect the use of local transaction-defining routines (that is, routines 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 field `TP-STATUS` of *TPSTATUS-REC*. In an environment of concurrent processes, this is a key way to inform processes if their routine calls have succeeded or not. All the ATMI routines set `TP-STATUS` to a value that reveals the nature of the error. In cases where the routine does not return to its caller, as in the case of `TPRETURN` or `TPFORWAR`, since they are called to terminate a service routine, the only way to communicate success or failure is through `APPL-RETURN-CODE IN` *TPSTATUS-REC* in the requester.

`APPL-RETURN-CODE IN` *TPSTATUS-REC* can also be used to communicate user-defined conditions. The value in `APPL-RETURN-CODE` is set from the value placed in `APPL-CODE IN` *TPSVCRET-REC* during `TPRETURN`. This code is sent regardless of the setting of `TP-RETURN-VAL IN` *TPSVCRET-REC* unless an error is encountered by `TPRETURN` or a transaction time-out occurs.

Values of TP-STATUS

The setting returned by `TP-STATUS IN` *TPSTATUS-REC* represent categories of errors. All the ATMI routines whose failure is reported by the setting returned by `TP-STATUS` have the four basic categories of

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

Protocol Errors

Protocol errors occur because an ATMI routine was called in an incorrect context. Refer to the `INTRO`(3cbl) reference page. This type of error usually happens for one of the following reasons:

- ♦ The ATMI call is being made in the wrong order.
- ♦ The ATMI call is being made by the wrong process.

A transaction participant rather than the initiator calling `TPCOMMIT` is a protocol error because the participant is the wrong process to be calling `TPCOMMIT`. This type of 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, try to discover the exact error in the context of the semantics of the specific call and ask 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 last major section in this chapter explains this log in detail. Since these are system errors rather than application errors, the systems 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 routines that take arguments can fail if invalid arguments are passed to them. In the case where the routine returns to the caller, the routine fails and sets `TPEINVAL`. In the case of `TPRETURN` or `TPFORWAR` if this type of error is discovered while processing the arguments, `TPESVCERR` is set for the routine 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 record types (`TPENOENT`)
- errors from incorrect permission to enter the application (`TPEPERM`)
- resource manager errors (`TPERMERR`)
- transaction related errors (`TPETRAN`)
- errors from mismatching of typed records (`TPEITYPE` and `TPEOTYPE`)
- errors that apply only to asynchronous communication calls or conversational calls because they involve communications handles (`TPELIMIT` and `TPEBADDESC`)
- errors that can occur as a result of the communication calls in general (`TPESVCFAIL`, `TPESVCERR`, `TPEBLOCK`, and `TPEGOTSIG`)
- 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 routine call is returning it. The routine that allows you to join the application, TPINITIALIZE, the routine that unlists an advertised service, TPUNADVERTISE, and the communication routines, TPCALL, TPACALL, TPCONNECT and TPGPRIO are the routines that return this error. The following table lists the routines and specifies the reason for the failure in each case.

Table 15-1 Error Routines

Function	Explanation
TPINITIALIZE	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 systems administrator.
TPCALL	The calling process is referencing a service, SERVICE-NAME IN *TPSVCDEF-REC* that is not known to the system since there is no entry for it in the bulletin board. On an application level, make sure you have referenced the service correctly; otherwise, see your systems administrator.
TPACALL	Same as TPCALL.
TPCONNECT	Cannot connect to SERVICE-NAME IN *TPSVCDEF-REC* 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

Permission Errors

The only ATMI routine that returns this error is TPINITIALIZE. 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 administrator if it is encountered.

Resource Manager Errors

These errors can occur with calls to TPOPEN and TPCLOSE and they return a setting of TPERMERR. 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 TP-STATUS. TPBEGIN, TPCANCEL and the TPCALL/TPACALL routines 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 from within a transaction.

In the case of the communication routines, 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 systems administrator. This is an application error. For the communication call to such a service to succeed, the TPNOTRAN setting for *TPSVCDEF-REC* 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 setting is explicitly set or if you access the service outside of your transaction.

Typed Record Errors

Typed record errors are returned as a result of sending processes requests or replies in typed records that are unfamiliar to them. TPEITYPE is returned by TPCALL and TPACALL when the request data record is sent to a service that does not know about this type. What does this mean? The record types that processes know about are determined both by the configuration file and by the BEA TUXEDO libraries that have been linked into the process. These libraries define and initialize a data structure to the typed records 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 record types. An application can set up the record type data structure (referred to as a record type switch) on a per process basis. Refer to the `tuxtypes`(5) and `typesw`(5) manual pages. This is an administrative decision and is mentioned here to clarify what is meant by a process *knowing* about a typed record. The rule for sending requests is that you must always send a request in a typed record that a service knows about; this information can be obtained from your systems administrator.

`TPEOTYPE` is returned by `TPCALL` and `TPGETRPLY` when the reply message is sent in a record 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 record. Not allowed means that although the process knows of the existence of this record type, the type returned to it does not match the type of the record it allocated to receive the reply and the caller is not allowing for a change in record type. The caller indicates this preference by setting `TPNOCHANGE`. In this case, strong type checking is enforced, returning `TPEOTYPE` when violated. The default is to have weak type checking allowing a different record type to be returned as long as it is known to the caller. Again, the rule for sending replies is that the reply record must be known to the caller and you must observe strong type checking if it has been indicated.

Communication Handle Errors

The errors discussed in this section can occur only when making asynchronous calls or conversational calls because they involve the misuse of the communication handle, `COMM-HANDLE IN` *TPSVCDEF-REC*. Asynchronous calls depend on communication handles to identify replies with their corresponding requests. Conversational sends and receives depend on communication handles to identify the connection; the call that initiates the connection depends on the availability of a communications handle. There are two things that the BEA TUXEDO system does not like you to do with communication handles.

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

The limit for outstanding communication handles (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 handles allowed should be ample for your application, but this limit is system-defined and cannot be redefined by your application.

The limit for communications handles 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 communications handle can become invalid. If a communications handle 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 handle and returns TPEBADDESC.

Sometimes a condition occurs where you can no longer reference a communications handle although it has never been used to retrieve a message. In this case we refer to the handle 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 replies to be retrieved. The outstanding handles for these 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 handle, 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 only 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 TP-STATUS being set to TPESVCERR. The caller's data is not sent, and if the failure was on TPGETRPLY, the communications handle becomes invalid. If an error of this nature is not encountered by TPRETURN, then the setting for TP-RETURN-VAL determines the success or failure of the call. If the application logic set TPFAIL, TPESVCFAIL is returned and the data message is sent to the caller.

The error codes TPEBLOCK and TPEGOTSIG 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 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 set to TPNOBLOCK and a blocking condition is encountered while awaiting the reply; for example, if a message is not currently available.

TPEGOTSIG really does not flag an error condition but indicates when a signal interrupts a BEA TUXEDO call. If the communication routines set TPSIGRSTRT, the calls will not fail and this code will not be returned.

Conversational Errors

Once a conversational connection has been established, TPSEND and TPRECV can fail with a TPEEVENT error. No data is sent by TPSEND. The event type is returned in the TPEVENT member of *TPSTATUS-REC*. A course of action is dictated by the particular event.

In conversational services TPSEND, TPRECV and TPDISCON return TPEBADDESC when an unknown handle 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 TPNOBLOCK set

You may recall that if TPNOBLOCK 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. 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.
- If the transaction timed out and its status is known to be aborted.

Heuristic Decision Errors

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. If TP-COMMIT-CONTROL is set to TP-CMT-COMPLETE, the application finds out about heuristics, but may still get back TPEHAZARD. Since TPEHAZARD simply means that a participant failed during the second phase, we cannot know if it completed the transaction successfully or unsuccessfully.

How to Deal with Errors

Listing 15-1 illustrates a general way of dealing with errors. The term *ATMICALL(3)* is used in this example to generically represent an ATMI routine call.

Listing 15-1 How to Deal with Errors

```
. . .
CALL "TPINITIALIZE" USING TPINFDEF-REC
                    USR-DATA-REC
                    TPSTATUS-REC.
IF NOT TPOK
   error message, EXIT PROGRAM
CALL "TPBEGIN" USING TPTRXDEF-REC
               TPSTATUS-REC.
IF NOT TPOK
   error message, EXIT PROGRAM
   Make atmi calls
   Check return values
IF  TPEINVAL
      DISPLAY "Invalid arguments were given."
IF  TPEPROTO
      DISPLAY "A call was made in an improper context."
. . .
   Include all error cases described in the INTRO(3cbl)
   reference page. Other return codes are not possible, so
   there should be no need to test them.
. . .
   continue
```

The specific settings of *TPSTATUS-REC* give you more insight into the nature of the problem and the level on which it can be corrected.

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 members (`TPESVCERR`).
  - The `TP-RETURN-VAL` argument of `TPRETURN` was set to `TPFAIL` (`TPESVCFAIL`).
  - The type or subtype of the reply record 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 `TPNOTRAN` set. 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 setting of `TPNOREPLY`, `TPNOBLOCK` or `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. In addition, the following sections address the following issues:

- 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 by the `T-OUT IN` *TPTRXDEF-REC* argument to `TPBEGIN`. By default, if a process is not in transaction mode, blocking time-outs are performed. When the 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` setting 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 communications handle 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 communications handle to an asynchronous reply 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 `TP-STATUS`. 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 a setting of `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 TPFORWAR.” 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 TPFORWAR

If a process is called in transaction mode, TPRETURN and TPFORWAR 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 TPFORWAR 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 TP-RETURN-VAL 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 TP-RETURN-VAL 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 TP-RETURN-VAL. Calling TPFORWAR can be used to indicate success up to that point in processing the request. If no application errors have been detected TPFORWAR is invoked, otherwise TPRETURN with TPFAIL. If TPFORWAR 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 `TPFORWAR`, 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

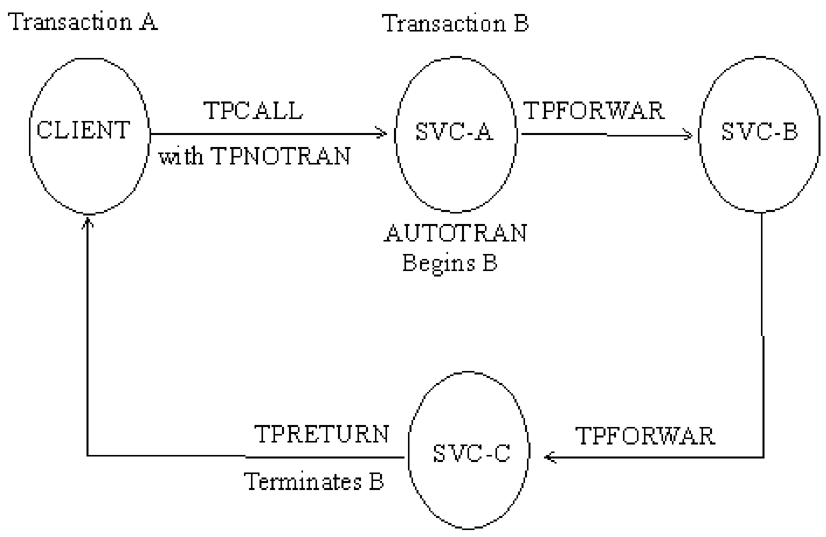
Service in Different Transaction with AUTOTRAN Set

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 `TPFORWAR`, the `TPRETURN` in the last service in the forward chain plays the initiator's transaction role to terminate the transaction. Refer to Figure 15-1.
- 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 to transaction time-out. As a result, `TPRETURN` 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 15-1 Transaction Roles of TPFORWAR 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 TP-RETURN-VAL regardless of the outcome of the transaction.
- Typically the errors returned will be processing errors, record 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 `TPACALL` is set to `TPNOTRAN` or `TPNOREPLY`
- a service must retrieve all asynchronous replies before calling `TPRETURN` or `TPFORWAR` (this applies regardless of transaction mode)
- the initiator must retrieve all asynchronous replies 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 `TPNOTRAN` 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 `TP-STATUS` to `TPETIME`
  - asynchronous calls can be made with `TPACALL` set to `TPNOREPLY` or `TPNOBLOCK` or `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 communications handles 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 the BEA TUXEDO system 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 the BEA TUXEDO system 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 the BEA TUXEDO system 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 TP-STATUS, 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 calls to define transactions. The BEA TUXEDO system makes an internal call to pass the global transaction information to each resource manager participating in the transaction. When `TPCOMMIT` or `TPABORT` is called, the BEA TUXEDO system 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 calls in your application code. Failure to follow this transaction rule will give indeterminate results.

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

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.

The Central Event Log

The central event log is a UNIX System file to which you can send messages from BEA TUXEDO clients and services. Writing to the central event log is accomplished through the `USERLOG` routine. 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.

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` routine 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

Each log entry consists of a tag and message text.

- A tag is made up of the following:
  - time of day (*`hhmmss`*)
  - name of the machine (the name that is returned by `uname -n`)
  - name and process-ID of the process calling `USERLOG`
- The message text for BEA TUXEDO system messages is preceded by the message catalog name, message number, and classification level.

For example, if the call

```
01 LOG-REC PIC X(15) VALUE "UNKNOWN USER ".
01 LOGREC-LEN PIC S9(9) VALUES IS 13.
CALL "USERLOG" USING LOG-REC LOGREC-LEN TPSTATUS-REC.
```

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

```
162214.mach1!security.23451: UNKNOWN USER
```

assuming `23451` is the process ID for `security`.

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

```
162214.mach1!security.23451: COBAPI_CAT: 999: UNKNOWN USER
```

where `COBAPI_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
```

How to Write to the Event Log

You can either have the error message you wish to write to the log in a record and use the record name as the argument to the call, or include the message as a literal within quotation marks as the argument to the call, as is shown in the example below.

```
01 TPSTATUS-REC.
   COPY TPSTATUS.
01 LOGMSG     PIC X(50).
01 LOGMSG-LEN PIC S9(9) COMP-5.
. . .
CALL "TPOPEN" USING TPSTSTUS-REC.
IF NOT TPOK
        MOVE "TPSVRINIT: Cannot Open Data Base" TO LOGMSG
        MOVE 43 LOGMSG-LEN
        CALL "USERLOG" USING LOGMSG
                       LOGMSG-LEN
                       TPSTATUS-REC.
. . .
```

In this example, the message is sent to the central event log if `TPOPEN` is not successful.

CHAPTER

16 Workstation COBOL Language Binding Feature

Introduction

This chapter specifically covers the use of the COBOL language binding feature of the Workstation on the following workstation platforms:

- UNIX
- MS-DOS
- Windows
- OS/2

The material in this chapter is intended to supplement the material presented in the programming chapters of this guide and the *BEA TUXEDO Workstation Guide*.

UNIX

Programming Consideration with UNIX Clients

This section covers items specific to writing and building BEA TUXEDO COBOL client programs to run under UNIX.

Writing Client Programs

COBOL client programs for UNIX workstations are the same as COBOL client programs within the BEA TUXEDO administrative domain. You do have available all of the ATMI functions.

Building Client Programs

Workstation client programs are compiled and link edited with the `buildclient` command. If you are building a UNIX Workstation client on the native node, use the `-w` option. This specifies that the client should be built using the workstation libraries. On a native node, where both native and workstation libraries are present, the default is to use the native libraries. The `-w` option ensures that the correct libraries for a workstation client are used. On a workstation, where only the workstation libraries are present, it is not necessary to use the `-w`.

Listing 16-1 shows an example of the `buildclient` command line on the native node.

Listing 16-1 Example of UNIX buildclient Command Lines

```
ALTCC=cobcc ALTCFLAGS="-I /APPDIR/include"
COBCPY=$TUXDIR/cobinclude
COBOPT="-C ANS85 -C ALIGN=8 -C NOIBMCOMP -C TRUNC=ANSI -C OSEXT=cbl"
export COBOPT COBCPY ALTCC ALTCFLAGS
buildclient -C -w -o empclient -f name.cbl -f "userlib1.a userlib2.a"
```

The `-o` option provides a name for your `a.out` file. Input files are specified with a `-f` *firstfiles* option in Listing 16-1 to indicate that they are called in ahead of system libraries. `buildclient` needs `TUXDIR` to locate system libraries. `CC` defaults to `cc`, but can be set to another compiler as in the example.

Environment Variables

Workstation clients make use of several environment variables. The following are checked by `TPINITIALIZE` when the workstation client attempts to join the application:

`WSENVFILE`
: names a file containing environment variable settings to be set in the client's environment.

`WSNADDR`
: specifies the network address of the workstation listener process through which the client gains access to the application. Use the value specified in the application configuration file for the workstation listener to be called. If the value begins with the characters `0x`, it is interpreted as a string of hex-digits, otherwise it is interpreted as ASCII characters.

`WSDEVICE`
: is the device name to be used to access the network and is not required by all transport layer interfaces.

`WSTYPE`
: is used within `TPINITIALIZE` when invoked by a workstation client to negotiate encode/decode responsibilities with the native site. An unspecified `WSTYPE` always causes encoding, even if it is also unspecified on the native site. The only way to ensure that encode/decode is turned off is to explicitly specify the same `WSTYPE` value for both sites.

`WSRPLYMAX`
: is used by `TPINITIALIZE` to set the maximum amount of core memory that ATMI uses for buffering application replies before they are dumped to disk. The system default limit for this is 32,000 bytes. The available memory on your machine is the key factor in deciding whether you should use `WSRPLYMAX` to set a lower limit. Writing replies to disk causes a substantial reduction in performance.

Other environment variables may be needed by Workstation COBOL clients on a UNIX workstation depending on what BEA TUXEDO features are being used.

Note: MicroFocus COBOL does not support shared objects on UNIX 3.2. `LIBNSL.a` is delivered as a shared object and is required by `buildclient` when linking a workstation client. As a result, Workstation for UNIX 3.2 is not supported.

DOS

Programming Considerations with MS-DOS Clients

This section covers items specific to writing and building BEA TUXEDO COBOL client programs to run under MS-DOS.

Writing Client Programs

COBOL client programs for MS-DOS workstations are the same as COBOL client programs within the BEA TUXEDO administrative domain. You have available all of the ATMI functions.

Building Client Programs

The COBOL source files that call ATMI functions must be compiled with the COBOL compiler using LITLINK option. Workstation client object files are link edited with the `buildclt` command. While the syntax of the command is straightforward, the usage varies according to the compilation system used. Listing 16-2 shows a sample `buildclt` command line.

Listing 16-2 Example of MS-DOS buildclt Command Lines

```
COBCPY=C:\TUXEDO\COBINC
COBDIR=C:\COBOL\LBR;C:\COBOL\EXEDLL
PATH=C:\C700\BIN;C:\COBOL\EXEDLL;...
TUXDIR=C:\tuxedo
INCLUDE=C:\TUXEDO\INCLUDE;C:\NET\TOOLKIT\INCLUDE;C:\C700\INCLUDE
LIB=C:\NET\TOOLKIT\LIB;C:\C700\LIB;C:\TUXEDO\LIB;C:\COBOL\LIB
buildclt -C -o EMP.EXE -f EMP+MFC7INTF+C7DOSIF+C7DOSLB \
     -f "/NOE/NOI/SE:300/CO/ST:10000"  -l "LLIBSOCK LLIBCE"
```

`buildclt` has the following options:

`-o` *`name`*
: the file name of the executable file being created. The default is `client.exe`.

`-f` *`firstfiles`*
: one or more object files to be included before the BEA TUXEDO libraries. `-f` can also be used to pass options to the compiler or linker. If more than one file name is specified, the names are separated by white space and the list is enclosed in quotation marks. The `-f` option can appear more than once.

`-l` *`libfiles`*
: specifies libraries to be included after the BEA TUXEDO libraries. If more than one file name is specified, the names are separated by white space and the list is enclosed in quotation marks. The `-l` option can appear more than once.

After the client programs have been developed and tested they can be moved to the MS-DOS workstations where they will be available to users.

Environment Variables

Workstation clients make use of several environment variables. The following are checked by `TPINITIALIZE` when the client attempts to join the application:

`WSENVFILE`
: names a file containing environment variable settings to be set in the client's environment. All of the other environment variables needed by client programs can be contained in this file.

`WSNADDR`
: specifies the network address of the workstation listener process through which the client gains access to the application. Use the value specified in the application configuration file for the workstation listener to be called. If the value begins with the characters `0x`, it is interpreted as a string of hex-digits, otherwise it is interpreted as ASCII characters.

`WSTYPE`
: is used within `TPINITIALIZE` when invoked by a workstation client to negotiate encode/decode responsibilities with the native site. An unspecified `WSTYPE` always causes encoding, even if it is also unspecified on the native site. The only way to ensure that encode/decode is turned off is to specify the same `WSTYPE` value for both sites.

`WSRPLYMAX`
: is used by `TPINITIALIZE` to set the maximum amount of core memory that ATMI uses for buffering application replies before they are dumped to disk. The system default limit for this is 32,000 bytes. The available memory on your machine is the key factor in deciding whether you should use `WSRPLYMAX` to set a lower limit. Writing replies to disk causes a substantial reduction in performance.

Other environment variables may be needed by Workstation COBOL clients on an MS-DOS workstation depending on what BEA TUXEDO features are being used.

Windows

Programming Considerations with the Windows DLL

This section covers items specific to writing and building BEA TUXEDO system client programs to run under Microsoft Windows. They are intended to supplement the material presented in the programming chapters of this guide and the *BEA TUXEDO Workstation Guide*.

Writing Client Programs

The ATMI calls used in Windows client programs are the same as those described in the programming chapters of this guide.

Building Client Programs

The COBOL source files that call ATMI functions must be compiled with the COBOL compiler using LITLINK option. Workstation client object files are link edited with the `buildclt` command. While the syntax of the command is straightforward, the usage varies according to the compilation system used. Listing 16-3 shows a sample `buildclt` command line.

Listing 16-3 Example of Windows buildclt Command Lines

```
COBCPY=C:\TUXEDO\COBINC
COBDIR=C:\COBOL\LBR;C:\COBOL\EXEDLL
PATH=C:\COBOL\EXEDLL;...
TUXDIR=C:\tuxedo
LIB=C:\NET\TOOLKIT\LIB;C:\MSVC\LIB;C:\TUXEDO\LIB;C:\COBOL\LIB
buildclt -C \-W \-o EMP.EXE \-f EMP \
        -f "/NOD/NOI/NOE/CO/SE:300" -d EMP.DEF -l WLIBSOCK

For Windows NT:

buildclt -C -W -o EMP.EXE  \
        -f empobj -d emp.def
```

`buildclt` has the following options:

`-W`
: specifies that the client should be built using Windows libraries.

`-d` *`deffile`*
: specifies the module definition file used for linking a Windows program.

`-o` *`name`*
: the file name of the executable file being created. The default is `client.exe`.

`-f` *`firstfiles`*
: one or more object files to be included before the BEA TUXEDO libraries. `-f` can also be used to pass options to the compiler or linker. If more than one file name is specified, the names are separated by white space and the list is enclosed in quotation marks. The `-f` option can appear more than once.

`-l` *`libfiles`*
: specifies libraries to be included after the BEA TUXEDO libraries. If more than one file name is specified, the names are separated by white space and the list is enclosed in quotation marks. The `-l` option can appear more than once.

Listing 16-4 is the module definition file used in the Windows `buildclt` command line.

Listing 16-4 Example of a Windows Module Definition File

```
NAME            EMP
DESCRIPTION     "EMPLOYEE CLIENT ATMI"
EXETYPE         WINDOWS
CODE            PRELOAD MOVEABLE DISCARDABLE
DATA            PRELOAD FIXED MULTIPLE
HEAPSIZE        15000
STACKSIZE       15000
EXPORTS         WordProc
```

Building ACCEPT/DISPLAY Clients

To build an executable client for an ACCEPT/DISPLAY application (like CSIMPAPP, for example), use the procedure shown in Listing 16-5.

Listing 16-5 Building ACCEPT/DISPLAY clients

```
a) compile the COBOL module and create a file.obj
         cobol file.cbl omf(obj) litlink;
b) use the following link statement
         link FILE+cblwinaf,,,\
         wcobatmi+cobws+wtuxws+ \
         lcobol+lcoboldw+cobw+cobfp87w+  \
         wlibsock,FILE.def  /nod/noe;
   For Windows NT the link statement is:
         cbllink -oEMP.exe EMP.obj  \
         cobws.lib ncobatmi.lib wtuxws32.lib  \
         libcmt.lib user32.lib
```

Blocking Network Behavior

Refer to the *BEA TUXEDO Workstation Guide*.

Restoring the Environment

Refer to the *BEA TUXEDO Workstation Guide*.

OS/2

Programming Considerations with OS/2 Clients

This section covers items specific to writing and building BEA TUXEDO System COBOL client programs to run under OS/2. They are intended to supplement the material presented in the programming chapters of this guide and the *BEA TUXEDO Workstation Guide*.

Writing Presentation Manager Client Programs

The ATMI calls used in Presentation Manager client programs are the same as those described in the programming chapters of this guide. They must, however, be incorporated into Presentation Manager modules.

Blocking Network Behavior

Refer to the *BEA TUXEDO Workstation Guide*.

Building Client Programs

The COBOL source files that call ATMI functions must be compiled with the COBOL compiler using LITLINK options and must use the OPTLINK calling convention. There is an example of the use of the OPTLINK calling convention in `$TUXDIR/apps/CSIMPAPP/ws/os2/csimpcl.cbl`. Workstation client object files are link edited with the `buildclt` command. While the syntax of the command is straightforward, the usage varies according to the compilation system used. Listing 16-6 shows a sample `buildclt`.

Listing 16-6 Example of OS/2 Presentation Manager buildclt Command Lines

```
COBCPY=C:\TUXEDO\COBINC
COBDIR=C:\COBOL\LBR;C:\COBOL\EXEDLL
PATH=C:\COBOL\EXEDLL;...
TUXDIR=C:\tuxedo

LIB=C:\TCPIP\LIB;C:\IBMCPP\LIB;C:\TOOLKT2\OS2LIB;C:\TUXEDO\LIB;C:\COBOL\LIB
buildclt -C -P -o emp.exe -f emp.obj -d emp.def
```

`buildclt` has the following options:

`-P`
: specifies that the client should be built using OS/2 Presentation Manager libraries.

`-o` *`name`*
: the file name of the executable file being created. The default is `client.exe`.

`-d` *`deffile`*
: specifies the module definition file used for linking a Windows program.

`-f` *`firstfiles`*
: one or more object files to be included before the BEA TUXEDO libraries. `-f` can also be used to pass options to the compiler or linker. If more than one file name is specified, the names are separated by white space and the list is enclosed in quotation marks. The `-f` option can appear more than once.

`-l` *`libfiles`*
: specifies libraries to be included after the BEA TUXEDO libraries. If more than one file name is specified, the names are separated by white space and the list is enclosed in quotation marks. The `-l` option can appear more than once.

Listing 16-7 shows the module definition file used in the OS/2 Presentation Manager `buildclt` command line.

Listing 16-7 Example of an OS/2 Presentation Manager Module Definition File

```
NAME            EMP WINDOWAPI
PROTMODE
EXETYPE         OS2
HEAPSIZE        15000
STACKSIZE       15000
EXPORTS        EMPWNDPROC
```

Listing 16-8 shows a sample OS/2 character-mode `buildclt` command line.

Listing 16-8 Example of OS/2 Character-Mode buildclt Command Lines

```
COBCPY=C:\TUXEDO\COBINC
COBDIR=C:\COBOL\LBR;C:\COBOL\EXEDLL
PATH=C:\COBOL\EXEDLL;...
TUXDIR=C:\tuxedo
LIB=C:\TCPIP\LIB;C:\MSVC\LIB;C:\TUXEDO\LIB;C:\COBOL\LIB
buildclt -C -O -o emp.exe -f emp.obj
```

`buildclt` has the following option:

`-O`

specifies that the client should be built using OS/2 character-mode libraries.