BEA Tuxedo®

Tutorials for Developing BEA Tuxedo ATMI Applications

Version 10.0
Document Released: September 28, 2007

Contents

3. Tutorial for bankapp, a Full C Application

4. Tutorial for CSIMPAPP, a Simple COBOL Application

5. Tutorial for STOCKAPP, a Full COBOL Application

6. Tutorial for XMLSTOCKAPP: a C and C++ XML Parser Application

CHAPTER 1

Developing a BEA Tuxedo Application

This topic includes the following sections:

Before Developing Your BEA Tuxedo Application

Before you begin developing your BEA Tuxedo Application-to-Transaction Monitor Interface (ATMI) application, it may be helpful to review the various concepts related to its design and the tools that are available to you. These concepts include identifying *clients* or the various ways input from the outside world is gathered and presented to your business for processing, and identifying *servers* or the programs containing the business logic that process the input data. Also important is reviewing the concept of *typed buffers* or how a client program allocates a memory area before sending data to another program. Another concept worth reviewing is that of the BEA Tuxedo *messaging paradigms*. ATMI client programs access the BEA Tuxedo system by calling the ATMI library. Most calls in the ATMI library support these different communication styles available to programmers, such as request/response and conversational. These are the building blocks of every BEA Tuxedo application.

For more information about concepts, such as application queues, event-based communication, and using ATMI, and on the tools available to you, refer to "Basic Architecture of the BEA Tuxedo ATMI Environment" in *Introducing BEA Tuxedo ATMI*. For information about programming an application, refer to *Programming BEA Tuxedo ATMI Applications Using C* and *Programming BEA Tuxedo ATMI Applications Using COBOL*.

Creating a BEA Tuxedo ATMI Client

Creating a BEA Tuxedo client is just like creating any other program in the C or C++ programming language. The BEA Tuxedo system provides you with a C-based programming interface known as the BEA Tuxedo Application-to-Transaction Monitor Interface or ATMI. The ATMI is an easy-to-use interface that enables the rapid development of BEA Tuxedo clients and servers.

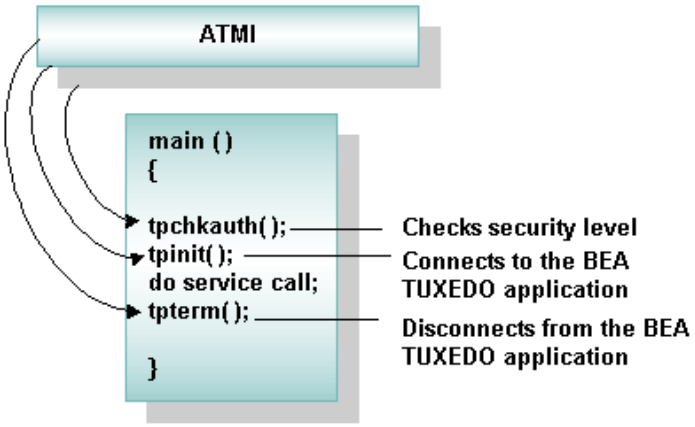
Note: BEA Tuxedo ATMI also supports a COBOL interface. (The examples shown here illustrate the C/C++ API.)

Client Tasks

Clients perform the following basic tasks:

- Clients may need to call `tpchkauth()` to determine the level of security required to join an application. Possible responses are: no security enabled, application password enabled, application authentication enabled, access control lists enabled, link-level encryption, public key encryption, auditing. (This is optional depending on whether you are using security levels.)
- Clients call `tpinit()` to connect to a BEA Tuxedo application. Any required security information is passed to the application as arguments for `tpinit()`.
- Clients perform service requests.
- Clients call `tpterm()` to disconnect from a BEA Tuxedo application.

Figure 1-1 Tasks Performed by a Client



See Also

- "Writing Clients" in *Programming BEA Tuxedo ATMI Applications Using C*
- "Administering Security" in *Using Security in CORBA Applications*
- "Using BEA Tuxedo Messaging Paradigms in Your Application" on page 1-6
- "What Are Typed Buffers?" in *Introducing BEA Tuxedo ATMI*
- *"What You Can Do Using the ATMI"* in *Introducing BEA Tuxedo ATMI*

Creating a BEA Tuxedo ATMI Server

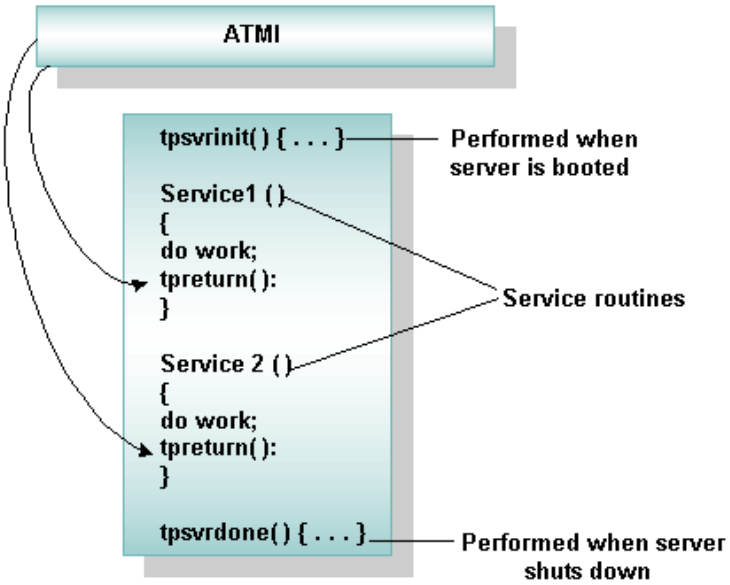
Developers use the ATMI programming interface to create a BEA Tuxedo client and server. However, BEA Tuxedo servers are not written by application developers as complete programs (that is, with a standard `main`). Instead, application developers write a set of specific business functions (known as services) that are compiled along with the BEA Tuxedo binaries to produce a server executable.

When a BEA Tuxedo server is booted, it continues running until it receives a shutdown message. A typical BEA Tuxedo server may perform thousands of service calls before being shut down and rebooted.

Server Tasks

- Application developers write the code and the BEA Tuxedo ATMI servers invoke the `tpsvrinit()` function only when the BEA Tuxedo server is booted. Programmers use this function to open an application resource (such as a database) for later use.
- Application developers write the code and the BEA Tuxedo ATMI servers invoke the `tpsvrdone()` function only when the BEA Tuxedo server is shut down. Programmers use this function to close any application resources opened by `tpsvrinit()`.
- Application developers write the code and the BEA Tuxedo ATMI servers request named application services that process client requests. BEA Tuxedo ATMI clients do not call *servers* by name; they call *services*. A BEA Tuxedo ATMI client does not "know" the location of the server processing its request.
- ATMI servers call the `tpreturn()` function to end a service request and return a buffer, if required, to the calling client.

Figure 1-2 Tasks Performed by a Server



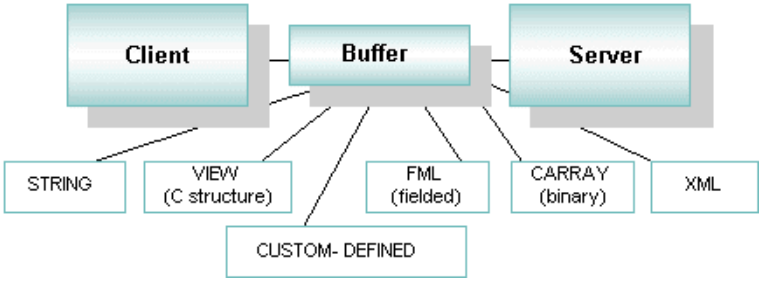
See Also

- "Writing Servers" on page 5-1 in *Programming BEA Tuxedo ATMI Applications Using C*
- "Using BEA Tuxedo Messaging Paradigms in Your Application" on page 1-6
- "What Are Typed Buffers?" in *Introducing BEA Tuxedo ATMI*
- "What You Can Do Using the ATMI" in *Introducing BEA Tuxedo ATMI*

Using Typed Buffers in Your Application

All communication in the BEA Tuxedo system is transmitted through typed buffers. The BEA Tuxedo system offers application developers the choice of many different buffer types to facilitate this communication. All buffers passed through the BEA Tuxedo system have special headers, and must be allocated and freed through the BEA Tuxedo ATMI (`tpalloc()`, `tprealloc()`, and `tpfree()`).

Figure 1-3 Different Types of Buffers



The typed buffers facility allows for generic well-defined processing to be implemented once a buffer type is shared across any type of network and protocol and any type of CPU architecture and operating system supported by the BEA Tuxedo system. The advantage of typed buffers in a distributed environment is that they relieve your clients and servers from the details of preparing data to be transferred between heterogeneous computers linked by various communications networks. This affords an application programmer time to concentrate on their business logic, instead of focusing attention on writing this facility into their own programs.

See Also

- "What Are Typed Buffers?" in *Introducing BEA Tuxedo ATMI*

Using BEA Tuxedo Messaging Paradigms in Your Application

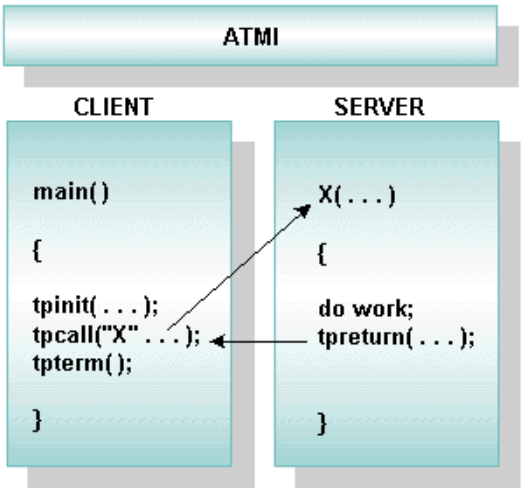
The BEA Tuxedo ATMI offers several communication models that you can use in your application:

- Using the Request/Response Model (Synchronous Calls)
- Using the Request/Response Model (Asynchronous Calls)
- Using Nested Calls
- Using Forwarded Calls
- Using Conversational Communication
- Using Unsolicited Notification
- Using Event-based Communication
- Using Queue-based Communication
- Using Transactions

Using the Request/Response Model (Synchronous Calls)

To make a synchronous call, a BEA Tuxedo ATMI client uses the ATMI function `tpcall()` to send a request to a BEA Tuxedo ATMI server. The function does not invoke a BEA Tuxedo server by name; instead, it invokes a specified service, which is provided by any server that offers the service and is available. The client then waits for the requested service to be performed. Until it receives a reply to its request, the client is not available for any other work. In other words, the client *blocks* until it receives a reply.

Figure 1-4 Using the Synchronous Request/Response Model



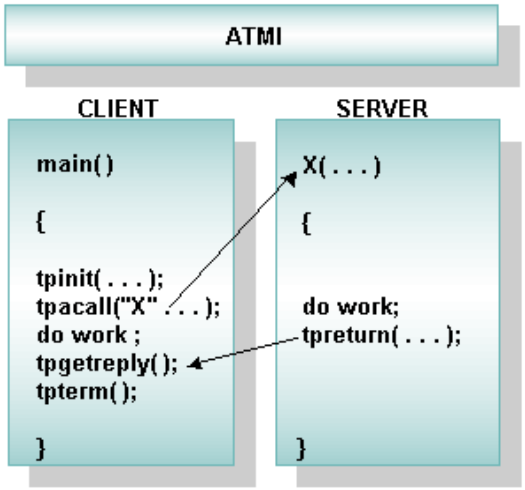
See Also

- "Request/Response Communication" in *Introducing BEA Tuxedo ATMI*

Using the Request/Response Model (Asynchronous Calls)

To make an asynchronous call, a client calls two ATMI functions: the `tpacall(3c)` function, to request a service, and the `tpgetrply(3c)` function, to retrieve the reply. This method is commonly used when a client can perform additional tasks after issuing a request and before receiving a reply.

Figure 1-5 Using Asynchronous Calls



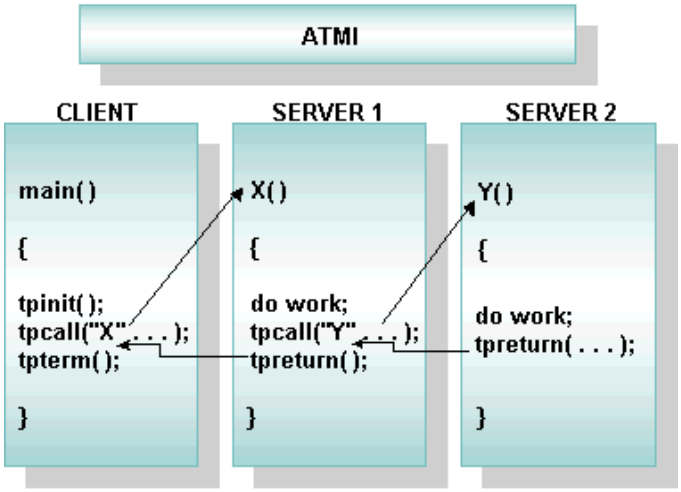
See Also

- "Request/Response Communication" in *Introducing BEA Tuxedo ATMI*

Using Nested Calls

Services can act as BEA Tuxedo ATMI clients and call other BEA Tuxedo services. In other words, you can request a service that, in turn, requests other services. For example, suppose a BEA Tuxedo client calls service X and waits for a reply. Service X then calls service Y and also waits for a reply. When service X receives a reply, it returns the reply to the calling client. This method is efficient because service X can take the reply from service Y, do more work on it, and modify the return buffer before sending a final reply back to the client.

Figure 1-6 Using Nested Calls



See Also

- "Nested Requests" in *Introducing BEA Tuxedo ATMI*

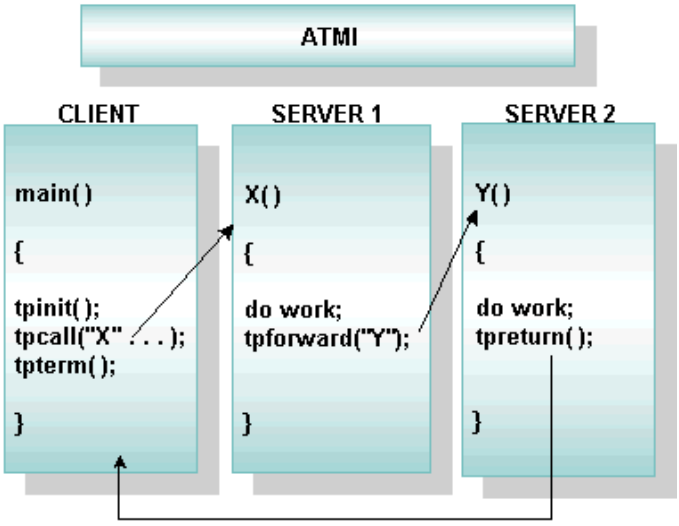
Using Forwarded Calls

With call forwarding, a nested service can return a reply directly to an ATMI client without going through the first service that was called, thereby freeing the first service to handle other requests. This capability is useful when the first service is acting strictly as a delivery agent, without adding data to the reply returned by the nested service.

To facilitate call forwarding, a service called by a client uses the `tpforward(3c)` function to pass the request to another service Y. This is the only situation in which a BEA Tuxedo service can end a service call without calling `tpreturn(3c)`.

Call forwarding is transparent to the client. In other words, the same client code is valid for service requests handled by one service and requests handled by more than one service.

Figure 1-7 Using Forwarded Calls



See Also

- "Forwarded Requests" in *Introducing BEA Tuxedo ATMI*

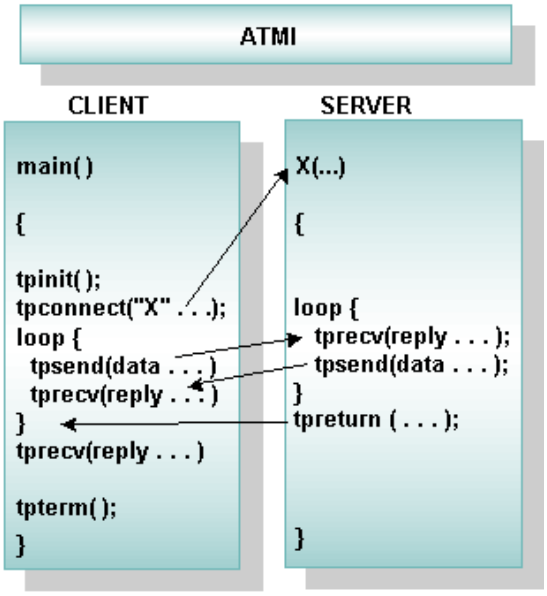
Using Conversational Communication

If multiple buffers need to be sent between a BEA Tuxedo ATMI client and a BEA Tuxedo service in a stateful manner, then the BEA Tuxedo conversation may be a suitable option.

Use BEA Tuxedo conversations judiciously because a server engaged in a conversation is unavailable until the conversation has ended. To implement a conversation, incorporate the following steps into your code:

1. The BEA Tuxedo client starts the conversation with the `tpconnect()` function.
2. The BEA Tuxedo client and the conversational server exchange buffers using the `tpsend()` and `tprecv()` functions. A special flag is set in the service calls to indicate which participant has control of the conversation.
3. The conversation ends in normal conditions, when the server calls `tpreturn()` or the `tpdiscon()` function.

Figure 1-8 Using Conversations



See Also

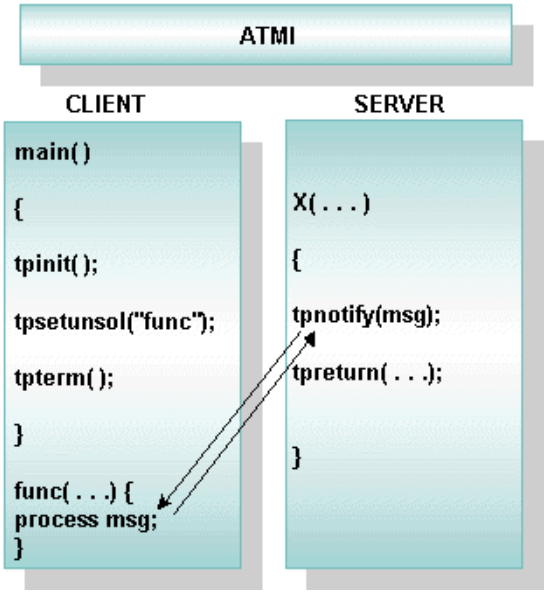
- "Conversational Communication" in *Introducing BEA Tuxedo ATMI*

Using Unsolicited Notification

To enable unsolicited notification, a BEA Tuxedo ATMI client creates an unsolicited message handle using the `tpsetunsol()` function. To send an unsolicited message, a BEA Tuxedo client or server can use either the `tpnotify()` function, to send a message to a single client, or the `tpbroadcast()` function, to send a message to multiple clients at the same time. When a client receives a message, the BEA Tuxedo system calls the client's unsolicited handler function.

In a signal-based system, a client does not have to poll for unsolicited messages. However, in a non-signal based system, a client must check for unsolicited messages using the `tpchkunsol()` function. Whenever a client makes a service request, `tpchkunsol()` is called implicitly.

Figure 1-9 Handling Unsolicited Notification



Note: If you call `tpnotify()` with the `tpack` flag bit set, you will receive an acknowledgement of your request.

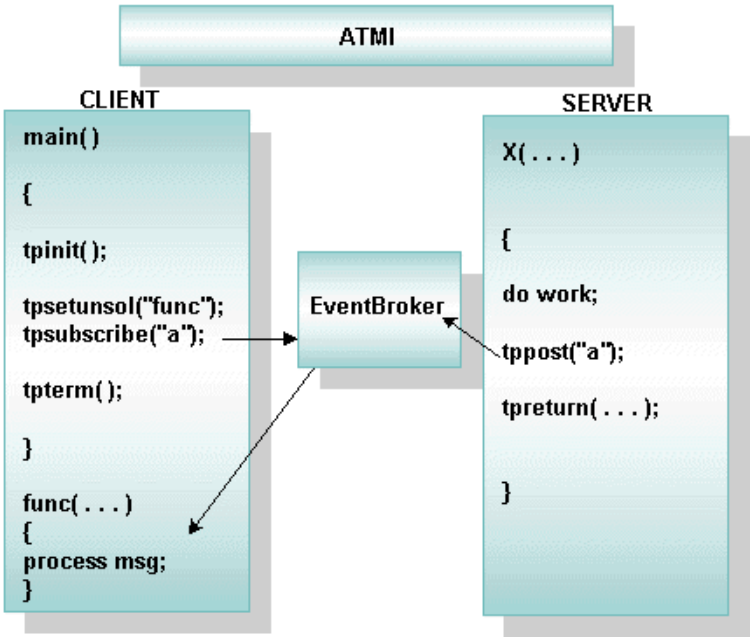
See Also

- "Unsolicited Communication" in *Introducing BEA Tuxedo ATMI*

Using Event-based Communication

In event-based communication, events can also be posted to application queues, log files, and system commands. Any BEA Tuxedo ATMI client can subscribe to a user-defined event using the `tpsubscribe()` function and receive an unsolicited message whenever a BEA Tuxedo service or client issues a `tppost()` function. ATMI clients can also subscribe to system-defined events that are triggered whenever the BEA Tuxedo system detects the event. When a server dies, for example, the `.SysServerDied` event is posted. No application server is needed to post this event, because it is performed by the BEA Tuxedo system.

Figure 1-10 Using Event-based Communication



See Also

- "How Events Are Reported" in *Introducing BEA Tuxedo ATMI*

Using Queue-based Communication

To interface with the /Q system, a BEA Tuxedo client uses two ATMI functions: `tpenqueue()`, to put messages into the queue space, and `tpdequeue()`, to take messages out of the queue space.

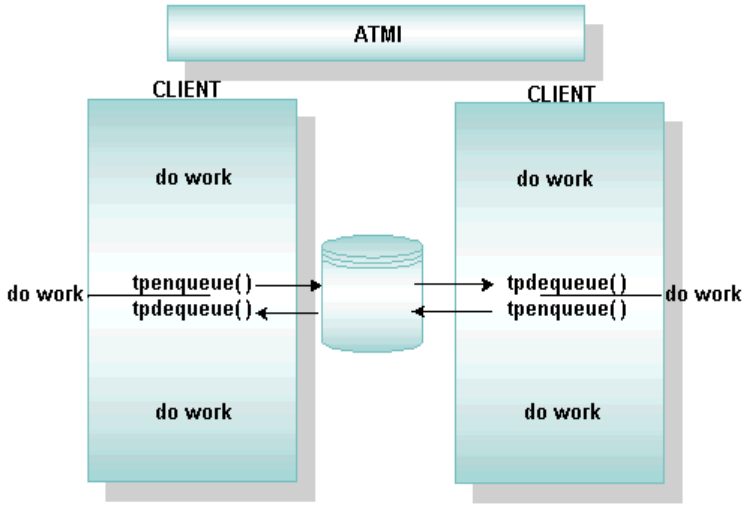
The following model represents peer-to-peer asynchronous messaging. Here, a client enqueues a message to a service using `tpenqueue()`. Optionally, the names of a reply queue and a failure queue can be included in the call to `tpenqueue()`. The client can also specify a correlation identifier value to accompany the message. This value is persistent across queues so that any reply or failure message associated with the queued message can be identified when it is read from the reply or the failure queue.

The client can use the default queue ordering (for example, a time after which the message should be dequeued), or can specify an override of the default queue ordering (asking, for example, that

this message be put at the top of the queue or ahead of another message on the queue). The call to `tpenqueue()` sends the message to the `TMQUEUE` server, the message is queued to stable storage, and an acknowledgment is sent to the client. The acknowledgment is not seen directly by the client, but can be assumed when the client gets a successful return. (A failure return includes information about the nature of the failure.) A message identifier assigned by the queue manager is returned to the application. The identifier can be used to dequeue a specific message. It can also be used in another `tpenqueue()` to identify a message on the queue ahead of the next message to be enqueued.

Before an enqueued message is made available for dequeuing, the transaction in which the message is enqueued must be committed successfully. A client uses `tpdequeue()` to dequeue messages from the queue.

Figure 1-11 Peer-to-Peer Asynchronous Messaging Model

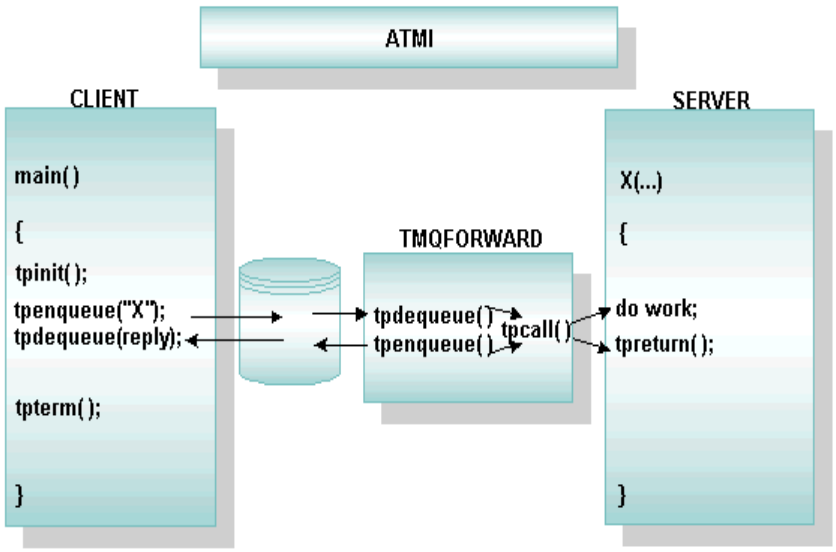


In the following graphic, forwarding a message to another server is illustrated.

The client enqueues a message intended for service X on the server. The service receives this message when it is active and when the handling instructions for the message are met (for example, the message can be encoded to be activated on Friday at 6 PM). Once the service is completed, it returns the reply to the queue space, from which it can be retrieved by the client.

This system of queuing is transparent to services. In other words, the same application code is used for a service, regardless of whether the service is invoked through queuing or direct service invocation using `tp(a)call`.

Figure 1-12 Using Queue Forwarding for Queue-based Service Invocation



See Also

- "Message Queuing Communication" *Introducing BEA Tuxedo ATMI*

Using Transactions

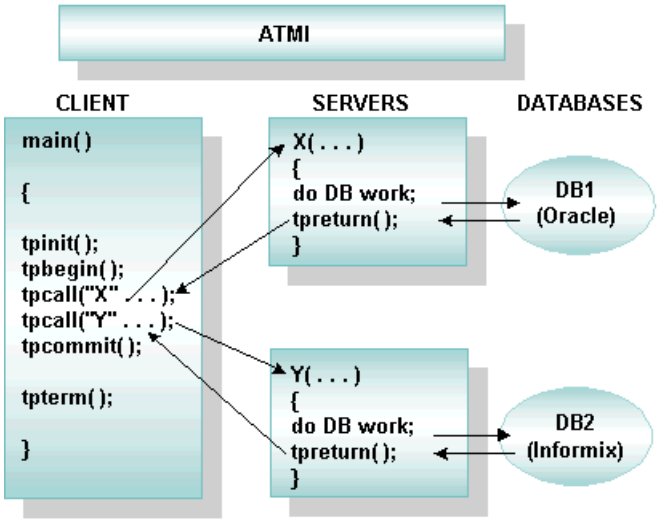
To implement transactions, an application programmer uses three ATMI functions:

- `tpbegin()` to start the transaction.
- `tpcommit()` to start the two-phase commit process.
- `tpabort()` to immediately cancel the transaction.

Any code placed outside the begin and commit/abort sequence is not included in the transaction.

In the following example, a client begins a transaction, requests two services, and then commits the transaction. Because the service requests are made between the beginning and the commitment of the transaction, both services join the transaction.

Figure 1-13 Using Transactions



See Also

- "Tutorial for bankapp, a Full C Application" on page 3-1
- "Tutorial for CSIMPAPP, a Simple COBOL Application" on page 4-1
- "Tutorial for simpapp, a Simple C Application" on page 2-1
- "Tutorial for STOCKAPP, a Full COBOL Application" on page 5-1

CHAPTER 2

Tutorial for simpapp, a Simple C Application

This topic includes the following sections:

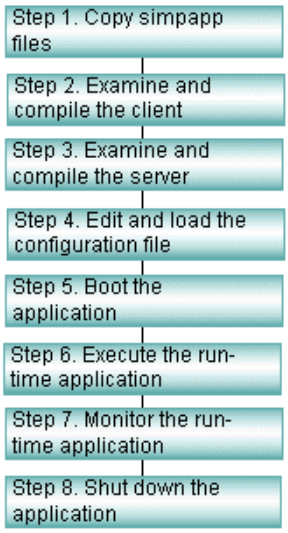
What Is simpapp?

`simpapp` is a sample ATMI application that includes one client and one server. This application is distributed with the BEA Tuxedo software. The server performs only one service: it accepts a lowercase alphabetic string from the client and returns the same string in uppercase.

Preparing simpapp Files and Resources

This topic is a tutorial that leads you, step-by-step, through the process of developing and running a sample BEA Tuxedo ATMI application. The following flowchart summarizes the process. Click on each task for instructions on completing that task.

Figure 2-1 simpapp Development Process



Before You Begin

Before you can run this tutorial, the BEA Tuxedo ATMI client and server software must be installed so that the files and commands referred to are available. If the installation has already been done by someone else, you need to know the pathname of the directory in which the software is installed (`TUXDIR`). You also need to have read and write permissions on the directories and files in the BEA Tuxedo directory structure so you can copy `simpapp` files and execute BEA Tuxedo commands.

About This Tutorial

The instructions for the `simpapp` tutorial are based on a UNIX system platform. While specific platform instructions for the UNIX operating system environment remain largely the same, instructions for performing tasks (such as copying `simpapp` files or setting environment

variables) on non-UNIX platforms (such as Windows 2003) may be different. For this reason, the examples used in the tutorial may or may not provide reliable procedures for your platform.

What You Will Learn

After you complete this tutorial, you will be able to understand the tasks ATMI clients and servers can perform, edit a configuration file for your own environment, and invoke `tmadmin` to check on the activity of your application. You will understand the basic elements of all BEA Tuxedo applications—client processes, server processes, and a configuration file—and you will know how to use BEA Tuxedo system commands to manage your application.

Step 1: How to Copy the simpapp Files

Note: The following instructions are based on a UNIX system platform. Instruction for non-UNIX platforms, such as Windows 2003, may be different. Examples used in the sample applications may vary significantly, depending on the specific platform.

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

   ```
   mkdir simpdir
   cd simpdir
   ```

   Note: This step is suggested so you can see the `simpapp` 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; do not use `csh`.

2. Set and export environment variables:

   ```
   TUXDIR=pathname of the BEA Tuxedo system root directory
   TUXCONFIG=pathname of your present working directory/tuxconfig
   PATH=$PATH:$TUXDIR/bin
   LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$TUXDIR/lib
     export TUXDIR TUXCONFIG 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 Sun Solaris, `/usr/5bin` must be the first directory in your `PATH`. With AIX on the RS/6000, use `LIBPATH` instead of `LD_LIBRARY_PATH`. On HP-UX on the HP 9000, use `SHLIB_PATH` instead of `LD_LIBRARY_PATH`.

   You need to set `TUXCONFIG` to be able to load the configuration file, described in "Step 4: Editing and Loading the Configuration File" on page 2-10.

3. Copy the `simpapp` files:

```
cp $TUXDIR/samples/atmi/simpapp/* .
```

Note: It is best to begin with a copy of the files rather than the originals delivered with the software because you will edit some of the files to make them executable.

4. List the files:

```
$ ls
   README       env        simpapp.nt  ubbmp     wsimpcl
   README.as400 setenv.cmd simpcl.c    ubbsimple
   README.nt    simpapp.mk simpserv.c  ubbws
$
```

Note: Except for the `README` files, the other files are variations of `simp*.*` and `ubb*` files for non-UNIX system platforms. The `README` files provide explanations of the other files.

The three files that are central to the application are:

- `simpcl.c`—the source code for the client program.
- `simpserv.c`—the source code for the server program.
- `ubbsimple`—the text form of the configuration file for the application.

See Also

- "What Is simpapp?" on page 2-1

Step 2: Examining and Compiling the Client

How to Examine the Client

Review the ATMI client program source code:

```
$ more simpcl.c
```

The output is shown in the following listing.

Listing 2-1 Source Code of simpcl.c

```
1     #include <stdio.h>
2     #include "atmi.h"           /* TUXEDO */
3
4
```

```
5
6
7     #ifdef __STDC__
8     main(int argc, char *argv[])
9
10    #else
11
12    main(argc, argv)
13    int argc;
14    char *argv[];
15    #endif
16
17    {
18
19             char *sendbuf, *rcvbuf;
20             int sendlen, rcvlen;
21             int ret;
22
23             if(argc != 2) {
24                      fprintf(stderr, "Usage: simpcl string\n");
25                      exit(1);
26             }
27             /* Attach to BEA TUXEDO as a Client Process */
28             if (tpinit((TPINIT *) NULL) == -1) {
29                      fprintf(stderr, "Tpinit failed\n");
30                      exit(1);
31             }
32             sendlen = strlen(argv[1]);
33            if((sendbuf = (char *)tpalloc("STRING", NULL, sendlen+1))== NULL){
34                      fprintf(stderr,"Error allocating send buffer\n");
35                      tpterm();
36                      exit(1);
37             }
38            if((rcvbuf = (char *)tpalloc("STRING", NULL, sendlen+1))== NULL){
39                      fprintf(stderr,"Error allocating receive buffer\n");
40                      tpfree(sendbuf);
41                      tpterm();
42                      exit(1);
43             }
44             strcpy(sendbuf, argv[1]);
45             ret = tpcall("TOUPPER", sendbuf, NULL, &rcvbuf, &rcvlen, 0);
46             if(ret == -1) {
47                   fprintf(stderr, "Can't send request to service TOUPPER\n");
48                      fprintf(stderr, "Tperrno = %d, %s\n", tperrno,
49                                tmemsgs[tperrno]);
50                      tpfree(sendbuf);
51                      tpfree(rcvbuf);
52                      tpterm();
53                      exit(1);
```

```
54              }
55              printf("Returned string is: %s\n", rcvbuf);
56
57              /* Free Buffers & Detach from BEA TUXEDO */
58              tpfree(sendbuf);
59              tpfree(rcvbuf);
60              tpterm();
61      }
```

Table 2-1 Significant Lines in the simpcl.c Source Code

Line(s)	File/Function	Purpose
2	`atmi.h`	Header file required whenever BEA Tuxedo ATMI functions are used.
28	`tpinit()`	The ATMI function used by a client program to join an application.
33	`tpalloc()`	The ATMI function used to allocate a typed buffer. `STRING` is one of the five basic BEA Tuxedo buffer types; `NULL` indicates there is no subtype argument. The remaining argument, `sendlen + 1`, specifies the length of the buffer plus 1 for the null character that ends the string.
38	`tpalloc()`	Allocates another buffer for the return message.
45	`tpcall()`	Sends the message buffer to the `TOUPPER` service specified in the first argument. Also includes the address of the return buffer. `tpcall()` waits for a return message.
35, 41, 52, 60	`tpterm()`	The ATMI function used to exit an application. A call to `tpterm()` is used to exit the application before exiting in response to an error condition (lines 36, 42, and 53). The final call to `tpterm()` (line 60) is issued after the message has been printed.

Table 2-1 Significant Lines in the simpcl.c Source Code (Continued)

Line(s)	File/Function	Purpose
40, 50, 51, 58, 59	`tpfree()`	Frees allocated buffers. `tpfree()` is the functional opposite of `tpalloc()`.
55	`printf()`	The successful conclusion of the program. It prints out the message returned from the server.

How to Compile the Client

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

   ```
   buildclient -o simpcl -f simpcl.c
   ```

   The output file is `simpcl` and the input source file is `simpcl.c`.

2. Check the results:

   ```
   $ ls -l
   total 97
   -rwxr-x--x 1 usrid   grpid  313091 May 28 15:41 simpcl
   -rw-r----- 1 usrid   grpid    1064 May 28 07:51 simpcl.c
   -rw-r----- 1 usrid   grpid     275 May 28 08:57 simpserv.c
   -rw-r----- 1 usrid   grpid     392 May 28 07:51 ubbsimple
   ```

   As can be seen, we now have an executable module called `simpcl`. The size of `simpcl` may vary.

See Also

- "What Is simpapp?" on page 2-1
- `buildclient(1)` in *BEA Tuxedo Command Reference*
- *BEA Tuxedo ATMI C Function Reference*

Step 3: Examining and Compiling the Server

How to Examine the Server

Review the ATMI server program source code.

```
$ more simpserv.c
```

Listing 2-2 Source Code of simpserv.c

```
   */

/* #ident"@(#) apps/simpapp/simpserv.c$Revision: 1.1 $" */
1  #include <stdio.h>
2  #include <ctype.h>
3  #include <atmi.h>/* TUXEDO Header File */
4  #include <userlog.h>/* TUXEDO Header File */
5  /* tpsvrinit is executed when a server is booted, before it begins
   processing requests.  It is not necessary to have this function.
   Also available is tpsvrdone (not used in this example), which is
   called at server shutdown time.

9  */
10 #if defined(__STDC__) || defined(__cplusplus)

12 tpsvrinit(int argc, char *argv[])
13 #else
14 tpsvrinit(argc, argv)
15 int argc;
16 char **argv;
17 #endif
18     {
19        /* Some compilers warn if argc and argv aren't used.
20         */
21        argc = argc;
22        argv = argv;
23        /* userlog writes to the central TUXEDO message log */
24         userlog("Welcome to the simple server");
25        return(0);
26 }
27 /* This function performs the actual service requested by the client.
      Its argument is a structure containing, among other things, a pointer
       to the data buffer, and the length of the data buffer.
30 */
31 #ifdef __cplusplus
32 extern "C"
33 #endif
34 void
35 #if defined(__STDC__) || defined(__cplusplus)
36 TOUPPER(TPSVCINFO *rqst)
37 #else
38 TOUPPER(rqst)
39 TPSVCINFO *rqst;
40 #endif
41  {
```

```
42       int i;
43
44       for(i = 0; i < rqst->len-1; i++)
45               rqst->data[i] = toupper(rqst->data[i]);
46       /* Return the transformed buffer to the requestor. */
47       tpreturn(TPSUCCESS, 0, rqst->data, 0L, 0);
48 }
```

Table 2-2 Significant Parts of the simpserv.c Source Code

Line(s)	File/Function	Purpose
Whole file		A BEA Tuxedo server does not contain a `main()`. The `main()` is provided by the BEA Tuxedo system when the server is built.
12	`tpsvrinit()`	This subroutine is called during server initialization, that is, before the server begins processing service requests. A default subroutine (provided by the BEA Tuxedo system) writes a message to `USERLOG` indicating that the server has been booted. `userlog(3c)` is a log used by the BEA Tuxedo system and can be used by applications.
38	`TOUPPER()`	The declaration of a service (the only one offered by `simpserv`). The sole argument expected by the service is a pointer to a `TPSVCINFO` structure, which contains the data string to be converted to uppercase.
45	`for loop`	Converts the input to uppercase by repeated calls to `TOUPPER`.
49	`tpreturn()`	Returns the converted string to the client with the `TPSUCCESS` flag set.

How to Compile the Server

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

```
buildserver -o simpserv -f simpserv.c -s TOUPPER
```

The executable file to be created is named `simpserv` and `simpserv.c` is the input source file. The `-s TOUPPER` option specifies the service to be advertised when the server is booted.

2. Check the results:

```
$ ls -l
total 97
-rwxr-x--x 1  usrid grpid 313091    May 28 15:41 simpcl
-rw-r----- 1  usrid grpid   1064    May 28 07:51 simpcl.c
-rwxr-x--x 1  usrid grpid 358369    May 29 09:00 simpserv
-rw-r----- 1  usrid grpid    275    May 28 08:57 simpserv.c
-rw-r----- 1  usrid grpid    392    May 28 07:51 ubbsimple
```

You now have an executable module called `simpserv`.

See Also

- "What Is simpapp?" on page 2-1
- `buildserver(1)` in *BEA Tuxedo Command Reference*
- *BEA Tuxedo ATMI C Function Reference*

Step 4: Editing and Loading the Configuration File

How to Edit the Configuration File

1. In a text editor, familiarize yourself with `ubbsimple`, which is the configuration file for `simpapp`.

Listing 2-3 The simpapp Configuration File

```
1$
2
3 #Skeleton UBBCONFIG file for the BEA Tuxedo Simple Application.
4 #Replace the <bracketed> items with the appropriate values.
5 RESOURCES
6 IPCKEY            <Replace with valid IPC Key greater than 32,768>
7
8  #Example:
9
10 #IPCKEY         62345
```

```
11
12 MASTER          simple
13 MAXACCESSERS    5
14 MAXSERVERS      5
15 MAXSERVICES     10
16 MODEL           SHM
17 LDBAL           N
18
19 *MACHINES
20
21 DEFAULT:
22
23                          APPDIR="<Replace with the current pathname>"
24              TUXCONFIG="<Replace with TUXCONFIG Pathname>"
25         TUXDIR="<Root directory of Tuxedo (not /)>"
26 #Example:
27 #       APPDIR="/usr/me/simpdir"
28 #       TUXCONFIG="/usr/me/simpdir/tuxconfig"
29 #       TUXDIR="/usr/tuxedo"
30
31 <Machine-name>  LMID=simple
32 #Example:
33 #tuxmach        LMID=simple
34 *GROUPS
35 GROUP1
36              LMID=simple   GRPNO=1  OPENINFO=NONE
37
38 *SERVERS
39  DEFAULT:
40              CLOPT="-A"
41  simpserv    SRVGRP=GROUP1 SRVID=1
42 *SERVICES
43  TOUPPER
```

2. For each <string> (that is, for each string shown between angle brackets), substitute an appropriate value.

How to Load the Configuration File

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

```
$ tmloadcf ubbsimple
Initialize TUXCONFIG file: /usr/me/simpdir/tuxconfig [y, q] ? y
$
```

2. Check the results:

```
$ ls -l
total 216
-rwxr-x--x 1 usrid grpid 313091    May 28 15:41 simpcl
-rw-r----- 1 usrid grpid   1064    May 28 07:51 simpcl.c
-rwxr-x--x 1 usrid grpid 358369    May 29 09:00 simpserv
-rw-r----- 1 usrid grpid    275    May 28 08:57 simpserv.c
-rw-r----- 1 usrid grpid 106496    May 29 09:27 tuxconfig
-rw-r----- 1 usrid grpid    382    May 29 09:26 ubbsimple
```

You now have a file called `TUXCONFIG`. The `TUXCONFIG` file is a new file under the control of the BEA Tuxedo system.

See Also

- "What Is simpapp?" on page 2-1
- `tmloadcf(1)` in the *BEA Tuxedo Command Reference*
- `UBBCONFIG(5)` in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*

Step 5: How to Boot the Application

1. 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/simpdir/tuxconfig
Booting all admin processes ...
   exec BBL -A:
   process id=24223 ... Started.

Booting server processes ...

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

The BBL is the administrative process that monitors the shared memory structures in the application. `simpserv` is the `simpapp` server that runs continuously, awaiting requests.

See Also

- "What Is simpapp?" on page 2-1

- tmboot(1) in the *BEA Tuxedo Command Reference*
- "How to Boot the Application" in *Administering a BEA Tuxedo Application at Run Time*

Step 6: How to Execute the Run-time Application

To execute your simpapp, have the client submit a request.

```
$ simpcl "hello, world"
Returned string is: HELLO, WORLD
```

See Also

- "What Is simpapp?" on page 2-1

Step 7: How to Monitor the Run-time Application

As the administrator, you can use the tmadmin command interpreter to check an application and make dynamic changes. To run tmadmin, you must have the TUXCONFIG environment variable set.

tmadmin can interpret and run over 50 commands. For a complete list, see tmadmin(1). The following uses two of the tmadmin commands.

1. Enter the following command:

   ```
   $ tmadmin
   ```

   The following lines are displayed:

   ```
   tmadmin - Copyright (c) 1999 BEA Systems, Inc. All rights reserved.
   >
   ```

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

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

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

See Also

- "What Is simpapp?" on page 2-1
- `tmadmin(1)` in the *BEA Tuxedo Command Reference*

Step 8: How to Shut Down the Application

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

```
$ tmshutdown
Shutdown all admin and server processes? (y/n): y
Shutting down all admin and server processes in /usr/me/simpdir/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*
$
113837.tuxmach!tmloadcf.10261: CMDTUX_CAT:879: A new file system has been
created. (size = 32 4096-byte blocks)
113842.tuxmach!tmloadcf.10261: CMDTUX_CAT:871: TUXCONFIG file
/usr/me/simpdir/tuxconfig has been created
113908.tuxmach!BBL.10768: LIBTUX_CAT:262: std main starting
113913.tuxmach!simpserv.10925: LIBTUX_CAT:262: std main starting
113913.tuxmach!simpserv.10925: Welcome to the simple server
114009.tuxmach!simpserv.10925: LIBTUX_CAT:522: Default tpsvrdone()
function used.
114012.tuxmach!BBL.10768: CMDTUX_CAT:26: Exiting system
```

See Also

- "What Is simpapp?" on page 2-1

- `tmshutdown(1)` in the *BEA Tuxedo Command Reference*
- `userlog(3c)` in the *BEA Tuxedo ATMI C Function Reference*
- "How to Shut Down Your Application" in *Administering a BEA Tuxedo Application at Run Time*
- "What Is the User Log (ULOG)?" in *Administering a BEA Tuxedo Application at Run Time*

CHAPTER 3

Tutorial for bankapp, a Full C Application

This topic includes the following sections:

What Is bankapp?

bankapp is a sample ATMI banking application that is provided with the BEA Tuxedo software. The application performs the following banking functions: opens and closes accounts, retrieves account balances, deposits or withdraws money from an account, and transfers monies from one account to another.

About This Tutorial

This tutorial leads you, step-by-step, through the procedure you must perform to develop the bankapp application. Once you have "developed" bankapp through this tutorial, you will be ready to start developing applications of your own.

The bankapp tutorial is presented in three sections:

- Preparing bankapp Files and Resources
- Running bankapp

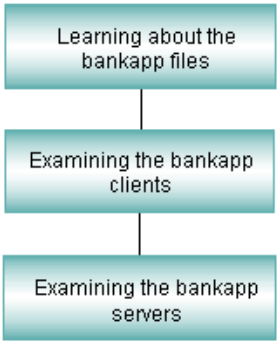
Note: This information has been written for UNIX and Windows 2003 system users with some experience in application development, administration, or system programming. We assume some familiarity with the BEA Tuxedo software.

Familiarizing Yourself with bankapp

Instructions in this sample application are automated for your convenience through shell scripts that work in a UNIX or Windows 2003 environment: `RUNME.sh` and `RUNME.cmd`. The associated `readme` files discuss how to run these files. Go through these files to understand the procedure more thoroughly and then follow these step-by-step instructions to help you set up and manage a distributed application.

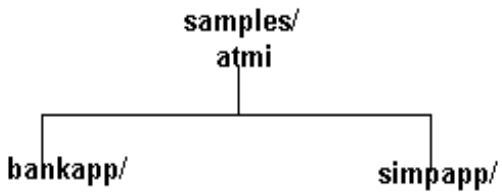
`bankapp` uses a demo relational database delivered with the software that enables you to use the sample application. Various commands and SQL code within the sample application (included for demo purposes only) provide access to the database.

This documentation provides a tour of the files, client, and services that make up the `bankapp` application. Click on any of the following activities for more information about that part of the tour.



Learning About the bankapp Files

The files that make up the bankapp application are delivered in a directory called bankapp, which is positioned as follows:



Exploring the Banking Application Files

The bankapp directory contains the following files:

- Five source files for service subroutines using embedded SQL statements
- Eight C source files
- One request/response client program (audit)
- One conversational server (AUDIT)
- One conversational client (auditcon)
- Three servers (or files associated with servers)
- Two files that generate data or transactions for the application
- Miscellaneous files
- Generic BEA Tuxedo application files (that is, files needed in any BEA Tuxedo application)
- Makefile for various add-ons
- Files provided to facilitate the use of bankapp as an example

The following table lists the files of the banking application. The table lists the source files delivered with the BEA Tuxedo software, files that are generated when the bankapp.mk is run, and a summary of the contents of each file.

Table 3-1 Description of the Banking Application Files

Source File	Generated File	Contents
ACCT.ec	ACCT.c, ACCT.o, ACCT	Contains two services: OPEN_ACCT and CLOSE_ACCT to open and close accounts.
ACCTMGR.c	ACCTMGR	A server that subscribes to events and logs notifications. Contains WATCHDOG and Q_OPENACCT_LOG services.
AUDITC.c	AUDITC	Contains a conversational server that handles service requests from the client auditcon.
BAL.ec	BAL.c, BAL.o, BAL	Contains a set of services: ABAL, TBAL, ABAL_BID, and TBAL_BID that allow the audit client to obtain bank-wide or branch-wide account or teller balances.
BALC.ec	BALC.c, BALC.o, BALC	Contains two services: ABALC_BID, and TBALC_BID. These services are the same as TBAL_BID and ABAL_BID, except that TPSUCCESS is returned when a branch ID is not found, which allows auditcon to continue running.
bankmgr.c	bankmgr	A client program that subscribes to events of special interest.
BTADD.ec	BTADD.c, BTADD.o, BTADD	Contains two services: BR_ADD and TLR_ADD for adding branches and tellers to the database.
cracl.sh	-	A shell script that creates access control lists (ACL) to demonstrate the access control security level.
crqueue.sh	-	A shell script that creates application queues for use in event notification.
crusers.sh	-	A shell script that creates groups and users to demonstrate the authentication security level.
event.flds	-	A field table file used in the event feature.
FILES	-	A descriptive list of all the files in bankapp.
README	-	Installation and boot procedures for all platforms except Windows 2003.
README.nt	-	Installation and boot procedures for the Windows 2003 platform.

Table 3-1 Description of the Banking Application Files

Source File	Generated File	Contents
README2	-	Documentation of additions to bankapp that demonstrate new features. The file is located in the samples/atmi/bankapp directory.
README2.nt	-	Documentation of additions to bankapp that demonstrate new features for the Windows 2003 platform. The file is located in the samples\atmi\bankapp directory.
RUNME.cmd		An interactive script to build, configure, boot, and shut down the application for Windows 2003.
RUNME.sh	-	An interactive script to build, configure, boot, and shut down the application for UNIX.
showq.sh!	-	A shell script that displays the status and contents of a message queue.
TLR.ec	TLR.c, TLR.o, TLR	Contains three services: WITHDRAWAL, DEPOSIT, and INQUIRY.
usrevtf.sh	-	A shell script that creates an ENVFILE for the BEA Tuxedo server TMUSREVT.
XFER.c	XFER.o, XFER	Contains TRANSFER service.
aud.v	aud.V, aud.h	An FML view used to define the structure passed between the audit client and the BAL server.
appinit.c	appinit.o	Contains customized versions of tpsvrinit() and tpsvrdone() for all servers other than TLR.
audit.c	audit.o, audit	A client that obtains bank-wide or branch-wide account and teller balances via the ABAL, TBAL, ABAL_BID, and TBAL_BID services.
auditcon.c	auditcon	An interactive version of audit that uses conversations and four services: ABAL, TBAL, ABALC_BID, and TBALC_BID.
bankapp.mk	-	An application makefile for UNIX.
bankapp.nt	-	An application makefile for Windows 2003.

Table 3-1 Description of the Banking Application Files

Source File	Generated File	Contents
bank.flds	bank.flds.h	A field table file containing bank database fields and auxiliary FML fields used by servers.
bank.h	-	Contains data definitions pertinent to multiple C programs in the application.
bankvar	-	Contains some environment variables for bankapp. Other environment variables are defined in ENVFILE, but because ENVFILE is set from within bankvar, you can control the entire environment for your application through bankvar.
crbank.sh	crbank	A shell script that creates databases for all banks when bankapp is run in SHM mode.
crbankdb.sh	crbankdb	A shell script that creates a database for one server group.
crtlog.sh	crtlog, TLOG	A shell script that creates a UDL and a TLOG on the master site and a UDL on the non-master sites.
driver.sh	driver	A shell script that drives the application by piping FML buffers with transaction requests through ud(1).
envfile.sh	envfile, ENVFILE	A shell script that creates ENVFILE for use by tmloadcf.
gendata.c	gendata	A program that generates ud-readable requests to add ten branches, thirty tellers, and two hundred accounts.
gentran.c	gentran	A program that generates ud-readable transaction requests from four services: DEPOSIT, WITHDRAWAL, TRANSFER, and INQUIRY.
populate.sh	populate	A shell script that populates the database by piping FML buffers with requests to add branches, tellers, and accounts through ud(1).
ubbmp	TUXCONFIG	A sample UBBCONFIG file for use in an MP mode configuration.
ubbshm	TUXCONFIG	A sample UBBCONFIG file for use in a SHM-mode configuration.

Table 3-1 Description of the Banking Application Files

Source File	Generated File	Contents
`util.c`	`util.o`	A set of functions, such as `getstrl()`, that are commonly used by services.
`bankclt.c`	`bankclt`	Client for `bankapp`.

See Also

- "Familiarizing Yourself with bankapp" on page 3-2

Examining the bankapp Clients

What Is the bankclt.c File?

The `bankclt` file contains the client program that requests BEA Tuxedo services from the `bankapp` application. This client program is text-based and provides the following options:

- Balance Inquiry
- Withdrawal
- Deposit
- Transfer
- Open Account
- Close Account
- Exit Application

Each of these options, except *Exit Application*, calls a subroutine that performs the following tasks:

1. Obtains the user input from the keyboard using the `get_account()`, `get_amount()`, `get_socsec()`, `get_phone()`, and the `get_val()` functions.
2. Puts the values into a global FML buffer (`*fbfr`). (Some functions add more fields than others. This is dependent on the information needed by the servers.)

3. Enables routines that make a request to the BEA Tuxedo system through the do_tpcall() function to invoke the required service. The following table lists the functions and the services they invoke.

Table 3-2 Services Called by Function

Function Name	Input FML Fields	Output FML Fields	Service Name
BALANCE()	ACCOUNT_ID	SBALANCE	INQUIRY
WITHDRAWAL()	ACCOUNT_ID SAMOUNT	SBALANCE	WITHDRAWAL
DEPOSIT()	ACCOUNT_ID SAMOUNT	SBALANCE	DEPOSIT
TRANSFER()	ACCOUNT_ID (0)[1] ACCOUNT_ID (1) SAMOUNT	SBALANCE (0) SBALANCE (1)	TRANSFER
OPEN_ACCT()	LAST_NAME FIRST_NAME MID_INIT SSN ADDRESS PHONE ACCT_TYPE BRANCH_ID SAMOUNT	ACCOUNT_ID SBALANCE	OPEN_ACCT
CLOSE_ACCT()	ACCOUNT_ID	SBALANCE	CLOSE_ACCT

[1]The number in parentheses is the FML occurrence number for that field.

4. After the call completes successfully, each function gets the fields it needs from the returned FML buffer and prints the results.

The do_tpcall() function (that begins on line 447 in bankclt.c) follows:

Listing 3-1 do_tpcall() in bankclt.c

```
/*
 * This function does the tpcall to Tuxedo.
```

```
 */
static int
do_tpcall(char *service)
{

        long len;
        char *server_status;
        /* Begin a Global transaction */
        if (tpbegin(30, 0) == -1) {
                (void)fprintf(stderr, "ERROR: tpbegin failed (%s)\n",
                        tpstrerror(tperrno));
                return(-1);
        }
        /* Request the service with the user data */
        if (tpcall(service, (char *)fbfr, 0, (char **)&fbfr, &len,
                        0) == -1) {
                if(tperrno== TPESVCFAIL && fbfr != NULL &&
                    (server_status=Ffind(fbfr,STATLIN,0,0)) != 0) {
                        /* Server returned failure */
                        (void)fprintf(stderr, "%s returns failure
                        (%s)\n",
                                service,server_status);
                }
                else {
                        (void)fprintf(stderr,
                                "ERROR: %s failed (%s)\n", service,
                                tpstrerror(tperrno));
                }
                /* Abort the transaction */
                (void) tpabort(0);
                return(-1);
        }
        /* Commit the transaction */
        if(tpcommit(0) < 0) {
                (void)fprintf(stderr, "ERROR: tpcommit failed
                        (%s)\n",
                        tpstrerror(tperrno));
```

```
                return(-1);
        }
        return(0);
}
```

The do_tpcall() function performs the following tasks:

- Begins a global transaction by calling tpbegin(), which ensures that all work is done as a single unit.
- Calls tpcall() with the requested service name (char *service) and the supplied FML buffer (the global *fbfr pointer).
- If tpcall() fails with a server-indicated failure (TPSVCERR), it prints the message from the server in the STATLIN FML field. The transaction is rolled back with tpabort() and it returns -1.
- If tpcall() fails with any other error, it prints the error message and rolls back the transaction with tpabort() and returns -1.
- If tpcall() succeeds, it commits the transaction using tpcommit() and returns 0.

Note: The unsolfcn() function is invoked if there is an unsolicited message to the client. It only supports STRING buffer types and prints the message.

How ud(1) Is Used in bankapp

bankapp uses the BEA Tuxedo program ud(1), which allows fielded buffers to be read from standard input and sent to a service. In the sample application, ud is used by both the populate and driver programs:

- In populate, a program called gendata passes service requests to ud with customer account information to be entered in the bankapp database.
- In driver, the data flow is similar, but the program is gentran and the purpose is to pass transactions to the application to simulate an active system.

A Request/Response Client: audit.c

audit is a request/response client program that makes branch-wide or bank-wide balance inquiries, using the services: ABAL, TBAL, ABAL_BID, and TBAL_BID. You can invoke it in two ways:

- audit [-a | -t]—prints the bank-wide total value of all accounts, or bank-wide cash supply of all tellers. Option -a or -t must be specified to indicate whether account balances or teller balances are to be tallied.
- audit [-a | -t] branch_ID—prints the branch-wide total value of all accounts, or branch-wide cash supply of all tellers, for branch denoted by branch_ID. Option -a or -t must be specified to indicate whether account balances or teller balances are to be tallied.

The source code for audit contains two major parts: main() and a subroutine called sum_bal(). BEA Tuxedo ATMI functions are used in both parts. The program uses a VIEW typed buffer and a structure that is defined in the aud.h header file. The source code for the structure can be found in the view description file, aud.v.

The following pseudo-code shows the algorithm for the program.

Listing 3-2 audit Pseudo-code

```
 main()
{
     Parse command-line options with getopt();
     Join application with tpinit();
     Begin global transaction with tpbegin();
     If (branch_ID specified) {
               Allocate buffer for service requests with tpalloc();
               Place branch_ID into the aud structure;
               Do tpcall() to "ABAL_BID" or "TBAL_BID";
               Print balance for branch_ID;
               Free buffer with tpfree();
     }
     else /* branch_ID not specified */
               call subroutine sum_bal();
     Commit global transaction with tpcommit();
     Leave application with tpterm();
}
sum_bal()
}
     Allocate buffer for service requests with tpalloc();
     For (each of several representative branch_ID's,
```

```
            one for each site)
            Do tpacall() to "ABAL" or "TBAL";
    For (each representative branch_ID) {
            Do tpgetrply() wtith TPGETANY flag set
                    to retrieve replies;
            Add balance to total;
            Print total balance;
    }
    Free buffer with tpfree();
}
```

Following is a summary of the two main parts of the `audit` source code.

In the programs `main()`:

1. /* Join application */
2. /* Start global transaction */
3. /* Create buffer and set data pointer */
4. /* Do tpcall */
5. /* Commit global transaction */
6. /* Leave application /*

In the subroutine `sum_bal`:

1. /* Create buffer and set data pointer */
2. /* Do `tpacall` */
3. /* Do `tpgetrply` to retrieve answers to questions */

A Conversational Client: auditcon.c

`auditcon` is a conversational version of the `audit` program. The source code for `auditcon` uses the ATMI functions for conversational communication: `tpconnect()` to establish the connection between the client and service, `tpsend()` to send a message, and `tprecv()` to receive a message.

The following pseudo-code shows the algorithm for the program.

Listing 3-3 auditcon Pseudo-code

```
main()
{
        Join the application
        Begin a transaction
        Open a connection to conversational service AUDITC
        Do until user says to quit: {
                Query user for input
                Send service request
                Receive response
                Print response on user's terminal
                Prompt for further input
        }

        Commit transaction
        Leave the application
}
```

A Client that Monitors Events: bankmgr.c

`bankmgr` is included with `bankapp` as an example of a client that is designed to run constantly. It subscribes to application-defined events of special interest, such as the opening of a new account or a withdrawal above $10,000. (The `bankmgr.c` client is more fully described in the `README2` file of `bankapp` and in the `bankmgr.c` code itself.)

See Also

- "Familiarizing Yourself with bankapp" on page 3-2
- "What You Can Do Using the ATMI" in *Introducing BEA Tuxedo ATMI*
- "What Are the BEA Tuxedo ATMI Messaging Paradigms?" in *Introducing BEA Tuxedo ATMI*
- "What Is bankapp?" on page 3-1
- "What Are Typed Buffers?" in *Introducing BEA Tuxedo ATMI*
- "Using Event-based Communication" on page 1-12
- *BEA Tuxedo Command Reference*

- *BEA Tuxedo ATMI C Function Reference*

Examining the bankapp Servers and Services

This topic provides the following information:

- A description of the servers and services delivered with `bankapp`.
- A description of how the services are link-edited into servers.
- Pseudo-code for each service that is either accessed by the BEA Tuxedo `bankclt.c`, or the application client, `audit.c`.
- Descriptions of the relationships between the `bankapp` services and servers.
- Descriptions of the `buildserver(1)` command options used to compile and build each server with the `main()` defined by the BEA Tuxedo system.
- An alternative method for structuring 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 C language code written specifically for an application. BEA Tuxedo's applications are written to make services available and capable of accessing resource managers. Service routines must be written by BEA Tuxedo application programmers.

All `bankapp` services are coded in C with embedded SQL except for the `TRANSFER` service, which does not interact directly with the database. The `TRANSFER` service is offered by the `XFER` server and is a C program (that is, its source file is a `.c` file rather than a `.ec` file).

All `bankapp` services of `bankapp` use functions provided in the Application-to-Transaction Management Interface (ATMI) for performing the following tasks:

- Managing typed buffers
- Communicating synchronously or asynchronously with other services
- Defining global transactions
- Generically accessing a resource manager
- Sending replies back to clients

bankapp Request/Response Servers

Five bankapp servers operate in request/response mode. Four of the five use embedded SQL statements to access a resource manager; the names of the source files for these servers (located in the bankapp sample application subdirectory), include a .ec filename extension.

The fifth server, XFER, for transfer, makes no calls to the resource manager itself; it calls the WITHDRAWAL and DEPOSIT services (offered by the TLR server) to transfer funds between accounts. The source file for XFER is a .c file, because XFER makes no resource manager calls and contains no embedded SQL statements.

This Server	Provides this Functionality
BTADD.ec	Allows branch and teller records to be added to the appropriate database from any site.
ACCT.ec	Provides customer representative services, namely the opening and closing of accounts (OPEN_ACCT and CLOSE_ACCT).
TLR.ec	Provides teller services, namely WITHDRAWAL, DEPOSIT, and INQUIRY. Each TLR process identifies itself as an actual teller in the TELLER file, via the user-defined -T option on the server's command line.
XFER.c	Provides fund transfers for accounts anywhere in the database.
BAL.ec	Calculates the account for all branches of the database or for a specified branch.

bankapp Conversational Server

AUDITC is an example of a conversational server. It offers one service, which is also called AUDITC. The conversational client, auditcon, establishes a connection to AUDITC and sends it requests for auditing information.

AUDITC evaluates requests and calls an appropriate service (ABAL, TBAL, ABAL_BID, or TBAL_BID) to get the appropriate information. When a reply is received from the service called, AUDITC sends it back to auditcon. A service in a conversational server can make calls to request/response services. It can also initiate connections to other conversational servers, but this functionality is not provided by AUDITC.

bankapp Services

bankapp offers 12 request/response services. The name of each bankapp service matches the name of a C function in the source code of a server.

This Service	Offered by this Server	With this Input	Performs this Function
BR_ADD	BTADD	FML buffer	• Adds a new branch record
TLR_ADD	BTADD	FML buffer	• Adds a new teller record
OPEN_ACCT	ACCT	FML buffer	• Inserts a record into the ACCOUNT file and calls DEPOSIT to add the initial balance • Chooses an ACCOUNT_ID for a new account based on the *BRANCH_ID* of the teller involved
CLOSE_ACCT	ACCT	FML buffer	• Deletes an ACCOUNT record • Validates ACCOUNT_ID • Calls WITHDRAWAL to remove the final balance
WITHDRAWAL	TLR	FML buffer	• Subtracts an amount from the specified branch, teller, and account balance • Validates the ACCOUNT_ID and SAMOUNT fields • Checks that funds are available from account and tell
DEPOSIT	TLR	FML buffer	• Adds an amount to specified branch, teller, and account balances • Validates the ACCOUNT_ID and SAMOUNT fields

This Service	Offered by this Server	With this Input	Performs this Function
INQUIRY	TLR	FML buffer	• Retrieves an account balance • Validates ACCOUNT_ID
TRANSFER	XFER	FML buffer	• Issues a tpcall() requesting WITHDRAWAL followed by one requesting DEPOSIT
ABAL	BAL	VIEW buffer of aud.v	• Calculates account balances for all branches on a given site
TBAL	BAL	VIEW buffer of aud.v as input	• Calculates the teller balances for all branches on a given site
ABAL_BID	BAL	VIEW buffer of aud.v as input	• Calculates the account balances for a specific BRANCH_ID
TBAL_BID	BAL	VIEW buffer of aud.v as input	• Calculates the teller balances for a specific BRANCH_ID

Algorithms of bankapp Services

The following listings show pseudo-code for the algorithms used for the bankapp services: BR_ADD, TLR_ADD, OPEN_ACCT, CLOSE_ACCT, WITHDRAWAL, DEPOSIT, INQUIRY, TRANSFER, ABAL, TBAL, ABAL_BID, and TBAL_BID. You can use them as road maps through the source code of the bankapp servers.

Listing 3-4 BR_ADD Pseudo-code

```
void BR_ADD (TPSVCINFO *transb)
{
        -set pointer to TPSVCINFO data buffer;
        -get all values for service request from field buffer;
        -insert record into BRANCH;
        -tpreturn() with success;
}
```

Listing 3-5 TLR_ADD Pseudo-code

```
void TLR_ADD (TPSVCINFO *transb)
{
        -set pointer to TPSVCINFO data buffer;
        -get all values for service request from fielded buffer;
        -get TELLER_ID by reading branch's LAST_ACCT;
        -insert teller record;
        -update BRANCH with new LAST_TELLER;
        -tpreturn() with success;
}
```

Listing 3-6 OPEN_ACCT Pseudo-code

```
void OPEN_ACCT(TPSVCINFO *transb)
{
    -Extract all values for service request from fielded buffer using Fget()
     and Fvall();
     -Check that initial deposit is positive amount and tpreturn() with
     failure if not;
    -Check that branch ID is a legal value and tpreturn() with failure if it
     is not;
     -Set transaction consistency level to read/write;
    -Retrieve BRANCH record to choose new account based on branch's LAST_ACCT
     field;
     -Insert new account record into ACCOUNT file;
     -Update BRANCH record with new value for LAST_ACCT;
    -Create deposit request buffer with tpalloc(); initialize it for FML with
     Finit();
     -Fill deposit buffer with values for DEPOSIT service request;
       -Increase priority of coming DEPOSIT request since call is from a
service;
     -Do tpcall() to DEPOSIT service to add amount of initial balance;
     -Prepare return buffer with necessary information;
     -Free deposit request buffer with tpfree();
      tpreturn() with success;
}
```

Listing 3-7 CLOSE_ACCT Pseudo-code

```
void CLOSE_ACCT(TPSVCINFO *transb)
{
   -Extract account ID from fielded buffer using Fvall();
   -Check that account ID is a legal value and tpreturn() with failure if it
    is not;
   -Set transaction consistency level to read/write;
   -Retrieve ACCOUNT record to determine amount of final withdrawal;
   -Create withdrawal request buffer with tpalloc(); initialize it for FML
    with Finit();
   -Fill withdrawal buffer with values for WITHDRAWAL service request;
   -Increase priority of coming WITHDRAWAL request since call is from
    a service;
   -Do tpcall() to WITHDRAWAL service to withdraw balance of account;
   -Delete ACCOUNT record;
   -Prepare return buffer with necessary information;
   -Free withdrawal request buffer with tpfree();
   tpreturn with success;
}
```

Listing 3-8 WITHDRAWAL Pseudo-code

```
void WITHDRAWAL(TPSVCINFO *transb)
{
    -Extract account id and amount from fielded buffer using Fvall() and
Fget();
   -Check that account id is a legal value and tpreturn() with failure if not;
        -Check that withdraw amount (amt) is positive and tpreturn() with
failure
    if not;
   -Set transaction consistency level to read/write;
   -Retrieve ACCOUNT record to get account balance;
   -Check that amount of withdrawal does not exceed ACCOUNT balance;
   -Retrieve TELLER record to get teller's balance and branch id;
   -Check that amount of withdrawal does not exceed TELLER balance;
   -Retrieve BRANCH record to get branch balance;
   -Check that amount of withdrawal does not exceed BRANCH balance;
   -Subtract amt to obtain new account balance;
   -Update ACCOUNT record with new account balance;
   -Subtract amt to obtain new teller balance;
   -Update TELLER record with new teller balance;
   -Subtract amt to obtain new branch balance;
   -Update BRANCH record with new branch balance;
   -Insert new HISTORY record with transaction information;
```

```
       -Prepare return buffer with necessary information;
        tpreturn with success;
    }
```

Listing 3-9 DEPOSIT Pseudo-code

```
    void DEPOSIT(TPSVCINFO *transb)
    {
        -Extract account id and amount from fielded buffer using Fvall() and
Fget();
      -Check that account ID is a legal value and tpreturn() with failure if not;
      -Check that deposit amount (amt) is positive and tpreturn() with failure if
       not;
       -Set transaction consistency level to read/write;
       -Retrieve ACCOUNT record to get account balance;
       -Retrieve TELLER record to get teller's balance and branch ID;
       -Retrieve BRANCH record to get branch balance;
       -Add amt to obtain new account balance;
       -Update ACCOUNT record with new account balance;
       -Add amt to obtain new teller balance;
       -Update TELLER record with new teller balance;
       -Add amt to obtain new branch balance;
       -Update BRANCH record with new branch balance;
       -Insert new HISTORY record with transaction information;
       -Prepare return buffer with necessary information;
       tpreturn() with success;
    }
```

Listing 3-10 INQUIRY Pseudo-code

```
    void INQUIRY(TPSVCINFO *transb)
    {
       -Extract account ID from fielded buffer using Fvall();
      -Check that account ID is a legal value and tpreturn() with failure if not;
       -Set transaction consistency level to read only;
       -Retrieve ACCOUNT record to get account balance;
       -Prepare return buffer with necessary information;
       tpreturn() with success;
    }
```

Listing 3-11 TRANSFER Pseudo-code

```
    void TRANSFER(TPSVCINFO *transb)
    {
       -Extract account ID's and amount from fielded buffer using Fvall()
        and Fget();
       -Check that both account IDs are legal values and tpreturn() with
        failure if not;
       -Check that transfer amount is positive and tpreturn() with failure if
        it is not;
       -Create withdrawal request buffer with tpalloc(); initialize it for
        FML with
        Finit();
         -Fill withdrawal request buffer with values for WITHDRAWAL service
request;
       -Increase priority of coming WITHDRAWAL request since call is from
        a service;
       -Do tpcall() to WITHDRAWAL service;
       -Get information from returned request buffer;
      -Reinitialize withdrawal request buffer for use as deposit request buffer
         with Finit();
       -Fill deposit request buffer with values for DEPOSIT service request;
       -Increase priority of coming DEPOSIT request;
       -Do tpcall() to DEPOSIT service;
       -Prepare return buffer with necessary information;
       -Free withdrawal/deposit request buffer with tpfree();
       tpreturn() with success;
    }
```

Listing 3-12 ABAL Pseudo-code

```
    void ABAL(TPSVCINFO *transb)
    {
       -Set transaction consistency level to read only;
       -Retrieve sum of all ACCOUNT file BALANCE values for the
         database of this server group (A single ESQL
         statement is sufficient);
       -Place sum into return buffer data structure;
       tpreturn( ) with success;
    }
```

Listing 3-13 TBAL Pseudo-code

```
void TBAL(TPSVCINFO *transb)
{
   -Set transaction consistency level to read only;
   -Retrieve sum of all TELLER file BALANCE values for the
     database of this server group (A single ESQL
     statement is sufficient);
   -Place sum into return buffer data structure;
   tpreturn( ) with success;
}
```

Listing 3-14 ABAL_BID Pseudo-code

```
void ABAL_BID(TPSVCINFO *transb)
{
   -Set transaction consistency level to read only;
   -Set branch_ID based on transb buffer;
   -Retrieve sum of all ACCOUNT file BALANCE values for records
     having BRANCH_ID = branch_ID (A single ESQL
     statement is sufficient);
   -Place sum into return buffer data structure;
   tpreturn( ) with success;
}
```

Listing 3-15 TBAL_BID Pseudo-code

```
void TBAL_BID(TPSVCINFO *transb)
{
   -Set transaction consistency level to read only;
   -Set branch_ID based on transb buffer;
   -Retrieve sum of all TELLER file BALANCE values for records
     having BRANCH_ID = branch_ID (A single ESQL
     statement is sufficient);
   -Place sum into return buffer data structure;
   tpreturn( ) with success;
}
```

Utilities Incorporated into Servers

Two C subroutines are included among the source files for bankapp: appinit.c and util.c:

- appinit.c contains application-specific versions of the tpsvrinit() and tpsvrdone() subroutines. tpsvrinit() and tpsvrdone() are subroutines included in the standard BEA Tuxedo ATMI main(). The default version of tpsvrinit() calls two functions: tpopen(), to open the resource manager, and userlog(), to post a message that the server has started. The default version of tpsvrdone() also calls two functions: tpclose(), to close the resource manager, and userlog(), to post a message that the server is about to shut down. Any application subroutines named tpsvrinit() and tpsvrdone() can be used in place of the default subroutines, thus enabling an application to provide initialization and pre-shutdown procedures of its own.
- util.c contains a subroutine called getstr(), which is used in bankapp to process SQL error messages.

Alternative Way to Code Services

In the bankapp source files all the services were incorporated into files that are referred to as the source code for servers. These files have the same names as the bankapp servers, but are not really servers because they do not contain a main() section. A standard main() is provided by BEA Tuxedo ATMI at buildserver time.

An alternative organization for a BEA Tuxedo system application is to keep each service subroutine in a separate file. Suppose, for example, that you want to use this alternative structure for the TLR server. The TLR.ec file contains three services that you maintain in three separate .ec files: INQUIRY.ec, WITHDRAW.ec, and DEPOSIT.ec. Follow these steps.

1. Compile each .ec file into a .o file.
2. Run the buildserver command specifying each .o file with a separate invocation of the -f option.

```
buildserver -r TUXEDO/SQL \
            -s DEPOSIT -s WITHDRAWAL -s INQUIRY \
            -o TLR \
            -f DEPOSIT.o -f WITHDRAW.o -f INQUIRY.o \
            -f util.o -f -lm
```

Note: The backslash in the preceding command-line entry is a documentation convention that indicates a line break for presentation purposes only. You should enter the command and options on one line

As this example illustrates, you do not need to code all the service functions in a single source file. In other words, a server does not need to exist as a source program file at all. It can be derived from various source files and exist as a server executable through the files specified on the `buildserver` command line. This can give you greater flexibility in building servers.

See Also

- "Familiarizing Yourself with bankapp" on page 3-2
- "What You Can Do Using the ATMI" in *Introducing BEA Tuxedo ATMI*
- `buildserver(1)` in the *BEA Tuxedo Command Reference*
- *BEA Tuxedo Command Reference*

Preparing bankapp Files and Resources

This documentation leads you through the procedures you must complete to create the files and other resources you need to run `bankapp`.

Click on each task for instructions on completing that task.

Step 1: How to Set the Environment Variables

The environment variables are defined in the `bankvar` file. The file is large (approximately 185 lines) because it includes extensive comments.

1. In a text editor, familiarize yourself with the `bankvar` file.
   - The first key line checks whether `TUXDIR` is set. If it is not set, execution of the file fails with the following message:

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

2. Set `TUXDIR` to the root directory of your BEA Tuxedo system directory structure, and export it.

3. Another line in `bankvar` sets `APPDIR` to the directory `${TUXDIR}/samples/atmi/bankapp`, which is the directory where `bankapp` source files are located. `APPDIR` is a directory where the BEA Tuxedo system looks for your application-specific files. You may prefer to copy the

bankapp files to a different directory to safeguard the original source files. If you do, enter the directory there. It does not have to be under TUXDIR.

4. Set a value for DIPCKEY. This is an IPCKEY for a BEA Tuxedo system database. The value of DIPCKEY must be different from the value of the BEA Tuxedo system IPCKEY specified in the UBBCONFIG file.

   Note: Other variables specified in bankvar play various roles in the sample application; you need to be aware of them when you are developing your own application. By including them all in bankvar, we provide you with a "template" that you may want to adapt at a later time for use with a real application.

5. When you have made all the required changes to bankvar, execute bankvar as follows:

   ```
   . ./bankvar
   ```

Listing 3-16 bankvar: Environment Variables for bankapp

```
# Copyright (c) 1997, 1996 BEA Systems, Inc.
# Copyright (c) 1995, 1994 Novell, Inc.
# Copyright (c) 1993, 1992, 1991, 1990 Unix System Laboratories, Inc.
# All rights reserved
#
# This file sets all the environment variables needed by the TUXEDO software
# to run the bankapp
#
# This directory contains all the TUXEDO software
# System administrator must set this variable
#
if [ -z "${TUXDIR}" ] ; then
 if [ ! -z "${ROOTDIR}" ] ; then
  TUXDIR=$ROOTDIR
  export TUXDIR
 fi
 fi
TUXDIR=${TUXDIR:?}
#
# Reset LANG if necessary
#
if [ ! -d ${TUXDIR}/locale/C -a -d ${TUXDIR}/locale/english_us ] ; then
 export LANG
 LANG=english_us.ascii
fi
#
# 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=${TUXDIR}/apps/bankapp
#
# This path contains the shared objects that are dynamically linked at
# runtime in certain environments, e.g., SVR4.
#
LD_LIBRARY_PATH=${TUXDIR}/lib:${LD_LIBRARY_PATH}
#
# Set the path to shared objects in HP-UX
#
SHLIB_PATH=${TUXDIR}/lib:${SHLIB_PATH}
#
# Set the path to shared objects in AIX
#
LIBPATH=${TUXDIR}/lib:/usr/lib:${LIBPATH}
#
# Logical block size; Database Administrator must set this variable
#
BLKSIZE=512
#
# Set default name of the database to be used by database utilities
# and database creation scripts
#
DBNAME=bankdb
#
# Indicate whether database is to be opened in share or private mode
#
DBPRIVATE=no
#
# Set Ipc Key for the database; this MUST differ from the UBBCONFIG
# *RESOURCES IPCKEY parameter
#
DIPCKEY=80953
#
# Environment file to be used by tmloadcf
#
ENVFILE=${APPDIR}/ENVFILE
#
# List of field table files to be used by mc, viewc, tmloadcf, etc.
#
FIELDTBLS=Usysflds,bankflds,creditflds,eventflds
#
FIELDTBLS32=Usysfl32,evt_mib,tpadm
#
# List of directories to search to find field table files
#
FLDTBLDIR=${TUXDIR}/udataobj:${APPDIR}
```

```
#
FLDTBLDIR32=${TUXDIR}/udataobj:${APPDIR}
#
# Universal Device List for database
#
FSCONFIG=${APPDIR}/bankdl1
#
# Network address, used in MENU script
#
NADDR=
#
# Network device name
#
NDEVICE=
#
# Network listener address, used in MENU script
#
NLSADDR=
#
# Make sure TERM is set
#
TERM=${TERM:?}
#
# Set device for the transaction log; this should match the TLOGDEVICE
# parameter under this site's LMID in the *MACHINES section of the
# UBBCONFIG file
#
TLOGDEVICE=${APPDIR}/TLOG
#
# Device for binary file that gives the BEA Tuxedo system all its information
#
TUXCONFIG=${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
#
# System name, used by RUNME.sh
#
UNAME=
#
# List of view files to be used by viewc, tmloadcf, etc.
#
VIEWFILES=aud.V
#
VIEWFILES32=mib_views,tmib_views
#
```

```
# List of directories to search to find view files
#
VIEWDIR=${TUXDIR}/udataobj:${APPDIR}
#
VIEWDIR32=${TUXDIR}/udataobj:${APPDIR}
#
# Specify the Q device (if events included in demo)
#
QMCONFIG=${APPDIR}/qdevice
#
# Export all variables just set
#
export TUXDIR APPDIR BLKSIZE DBNAME DBPRIVATE DIPCKEY ENVFILE
export LD_LIBRARY_PATH SHLIB_PATH LIBPATH
export FIELDTBLS FLDTBLDIR FSCONFIG MASKPATH OKXACTS TERM
export FIELDTBLS32 FLDTBLDIR32
export TLOGDEVICE TUXCONFIG ULOGPFX
export VIEWDIR VIEWFILES
export VIEWDIR32 VIEWFILES32
export QMCONFIG
#
# 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

#
# Check for other machine types bin directories
#
for DIR in /usr/5bin /usr/ccs/bin /opt/SUNWspro/bin
do
       if [ -d ${DIR} ] ; then
 PATH="${DIR}:${PATH}"
fi
done
```

Note: If your operating system is Sun Solaris, you must do two things: use `/bin/sh` rather than `csh` for your shell place; and specify `/usr/5bin` at the beginning of your `PATH`, as follows.

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

See Also

- "Preparing bankapp Files and Resources" on page 3-24

Step 2: Building Servers in bankapp

`buildserver`(1) puts together an executable ATMI server built on the BEA Tuxedo ATMI `main()`. 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` invokes the `cc` command. The environment variables `CC` and `CFLAGS` can be set to name an alternative compile command and to set flags for the compile and link edit phases. The `buildserver` command is used in `bankapp.mk` to compile and build each server in the banking application. The following sections describe the six `bankapp` servers:

- How to Build ACCT Server
- How to Build the BAL Server
- How to Build the BTADD Server
- How to Build the TLR Server
- How to Build the XFER Server
- Step 3: Editing the bankapp Makefile

How to Build ACCT Server

The `ACCT` server is derived from a file called `ACCT.ec` that contains the code for the `OPEN_ACCT` and `CLOSE_ACCT` functions. It is created in two steps. `ACCT.ec` is first compiled to an `ACCT.o` file, which is then specified to the `buildserver` command so that any compile-time errors can be identified and resolved.

1. Create the ACCT.o file (performed for you in bankapp.mk):
   - The .c file is generated as follows: esql ACCT.ec.
   - The .o file is generated as follows: cc -I $TUXDIR/include -c ACCT.c.
   - The ACCT server is created by running the following buildserver command line.

```
buildserver -r TUXEDO/SQL \
            -s OPEN_ACCT -s CLOSE_ACCT \
            -o ACCT \
            -f ACCT.o -f appinit.o -f util.o
```

   Note: The backslash in the preceding command-line entry is a documentation convention that indicates a line break for presentation purposes only. You should enter the command and options on one line.

Following is an explanation of the buildserver command-line options:

- 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 the string TUXEDO/SQL.
- The -s option is used to specify the names of the services in the server that are available to be advertised when the server is booted. If the name of a function that performs a service is different from the corresponding service name, the function name becomes part of the argument to the -s option.
  In bankapp, the function name is always the same as the name of the corresponding service so only the service names themselves need to be specified. It is our convention to spell all service names in all uppercase. For example, the OPEN_ACCT service is processed by the function OPEN_ACCT(). However, the -s option to buildserver does allow you to specify an arbitrary name for the processing function for a service within a server. Refer to the buildserver(1) reference page for details. It is also possible for an administrator to specify that only a subset of the services used to create the server with the buildserver command is to be available when the server is booted.
- 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. (For related information, see the description of the -l option on the buildserver(1) reference page.) The order in which the files are listed depends on function references and the libraries in which those references are resolved. Source modules should be listed before libraries that might be used to resolve their references. If these are .c files, they are first compiled. (In the example above, appinit.o and util.o have been already compiled.) Object files can be either separate .o files or groups of files in archive (.a) files. If more than one filename is given as an argument to the -f option, the list must be enclosed in double quotation

marks. Although the -f option takes only one file or one list of files (enclosed in double quotation marks) as an argument, you can include the -f option as many times as necessary on a single command line.

To summarize, the options specified on the buildserver command line used to create the ACCT server performed the following functions:

- The -r option specifies the BEA Tuxedo system SQL resource manager.
- The -s option names the OPEN_ACCT and CLOSE_ACCT services (which are defined by functions of the same name in the ACCT.ec file) to be the services that make up the ACCT server.
- The -o option assigns the name ACCT to the executable output file.
- The -f option specifies that the ACCT.o, appinit.o, and util.o files are to be used in the link-edit phase of the build.

  Note: The appinit.c file contains the system-supplied tpsvrinit() and tpsvrdone(). (Refer to tpservice(3c) reference pages for an explanation of how these routines are used.)

How to Build the BAL Server

The BAL server is derived from a file called BAL.ec that contains the code for the ABAL, TBAL, ABAL_BID, and TBAL_BID functions. As with ACCT.ec, the BAL.ec is first compiled to a BAL.o file before being supplied to the buildserver command so that any compile-time errors can be identified and resolved.

1. Modify the buildserver command used to create the BAL server as follows:

```
buildserver -r TUXEDO/SQL \
            -s ABAL -s TBAL -s ABAL_BID -s TBAL_BID\
            -o BAL \
            -f BAL.o -f appinit.o
```

   Note: The backslash in the preceding command-line entry is a documentation convention that indicates a line break for presentation purposes only. You should enter the command and options on one line.

   – Use the -r option to specify the BEA Tuxedo system SQL resource manager.

   – Use the -s option to name the ABAL, TBAL, ABAL_BID, TBAL_BID services that make up the BAL server. The functions in the BAL.ec file that define these services have identical names.

– Use the -o option to assign the name BAL to the executable server.

– Use the -f option to specify that the BAL.o and the appinit.o files are to be used in the link-edit phase.

How to Build the BTADD Server

The BTADD server is derived from a file called BTADD.ec that contains the code for the BR_ADD and TLR_ADD functions. The BTADD.ec is compiled to a BTADD.o file before being supplied to the buildserver command.

1. Modify the buildserver command used to create the BTADD server as follows:

```
buildserver -r TUXEDO/SQL \
            -s BR_ADD -s TLR_ADD \
            -o BTADD \
            -f BTADD.o -f appinit.o
```

Note: The backslash in the preceding command-line entry is a documentation convention that indicates a line break for presentation purposes only. You should enter the command and options on one line.

– Use the -r option to specify the BEA Tuxedo system SQL resource manager.

– Use the -s option to name the BR_ADD and TLR_ADD services that make up the BTADD server. The functions in the BTADD.ec file that define these services have identical names.

– Use the -o option to assign the name BTADD to the executable server.

– Use the -f option to specify that the BTADD.o and appinit.o files are to be used in the link-edit phase.

How to Build the TLR Server

The TLR server is derived from a file called TLR.ec that contains the code for the DEPOSIT, WITHDRAWAL, and INQUIRY functions. The TLR.ec is also compiled to a TLR.o file before being supplied to the buildserver command.

1. Modify the buildserver command used to create the TLR server as follows:

```
buildserver -r TUXEDO/SQL \
            -s DEPOSIT -s WITHDRAWAL -s INQUIRY \
            -o TLR \
            -f TLR.o -f util.o -f -lm
```

Note: The backslash in the preceding command-line entry is a documentation convention that indicates a line break for presentation purposes only. You should enter the command and options on one line.

– Use the `-r` option to specify the BEA Tuxedo system SQL resource manager.

– Use the `-s` option to name the `DEPOSIT`, `WITHDRAWAL`, and `INQUIRY` services that make up the `TLR` server. The functions in the `TLR.ec` file that define these services have identical names.

– Use the `-o` option to assign the name `TLR` to the executable server.

– Use the `-f` option to specify that the `TLR.o` and the `util.o` files are to be used in the link-edit phase.

Note: In this example, the `-f` option is used to pass an option (`-lm`) to the `cc` command, which is invoked by `buildserver`. The `-lm` argument to `-f` causes the math libraries to be linked in at compile time.

(Refer to `cc`(1) in the *UNIX System V User's Reference Manual* for a complete list of compile-time options.)

How to Build the XFER Server

The `XFER` server is derived from a file called `XFER.c` that contains the code for the `TRANSFER` function. The `XFER.c` is also compiled to an `XFER.o` file before being supplied to the `buildserver` command.

1. Modify the `buildserver` command used to create the `XFER` server as follows:

```
buildserver -r TUXEDO/SQL \
            -s TRANSFER \
            -o XFER \
            -f XFER.o -f appinit.o
```

Note: The backslash in the preceding command-line entry is a documentation convention that indicates a line break for presentation purposes only. You should enter the command and options on one line.

– Use the `-r` option to specify the BEA Tuxedo system SQL resource manager.

– Use the `-s` option to name the `TRANSFER` service that makes up the `XFER` server. The function in the `XFER.c` file that defines the `TRANSFER` service has the identical name.

– Use the `-o` option to assign the name `XFER` to the executable server.

– Use the -f option to specify that the XFER.o and the appinit.o files are to be used in the link-edit phase.

Servers Built in the bankapp.mk File

The topics on creating the bankapp servers are important to your understanding of how the buildserver command is specified. However, in actual practice you are apt to incorporate the build into a makefile; that is the way it is done in bankapp.

Step 3: Editing the bankapp Makefile

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

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

How to Edit the TUXDIR Parameter

1. Review bankapp.mk, about 40 lines into the file, where you come to the following comment and 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
# bankapp.mk TUXDIR=/correct/tuxdir", or the build of bankapp will fail.
#
TUXDIR=../..
```

2. Set the TUXDIR parameter to the absolute pathname of the root directory of your BEA Tuxedo system installation.

How to Edit the APPDIR Parameter

1. Review the setting of the APPDIR parameter. As bankapp is delivered, APPDIR is set to the directory in which the bankapp files are located, relative to TUXDIR. The following section of bankapp.mk defines and describes the setting of APPDIR.

```
#
# Directory where the bankapp application source and executables reside.
# This file must either be edited to set this value correctly, or the
# correct value must be passed via "make -f bankapp.mk
```

```
# APPDIR=/correct/appdir", or the build of bankapp will fail.
#
APPDIR=$(TUXDIR)/samples/atmi/bankapp
#
```

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

How to Set the Resource Manager Parameters

By default, `bankapp` is set up to use the BEA Tuxedo/SQL as the database resource manager. This arrangement is based on the assumption that you have the BEA Tuxedo system database on your system. If this is not the case, you should set the RM parameter to the name of your resource manager as listed in `TUXDIR/udataobj/RM`.

```
#
# Resource Manager
#
RM=TUXEDO/SQL
#
```

Note: The BEA Tuxedo SQL resource manager is included for demonstration purposes only.

How to Run the bankapp.mk File

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

   ```
   nohup make -f bankapp.mk &
   ```

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

   Note: `bankvar` sets a number of parameters that are referenced when `bankapp.mk` is run.

See Also

- "Preparing bankapp Files and Resources" on page 3-24

Step 4: Creating the bankapp Database

This documentation describes the interface between `bankapp` and a resource manager, typically a database management system, and how to create the database for `bankapp`. `bankapp` is

designed to use the BEA Tuxedo/SQL facilities of the BEA Tuxedo system database, which is an XA-compliant resource manager.

Note: The BEA Tuxedo SQL resource manager is included for demonstration purposes only.

How you create the `bankapp` database depends on whether you bring up the application on a single processor (SHM mode) or on a network of more than one processor (MP mode).

How to Create the Database in SHM Mode

1. Set the environment by typing the following:

   ```
   . ./bankvar
   ```

2. Execute `crbank`. `crbank` calls `crbankdb` three times, changing some environment variables each time, so that you end up with three database files on a single machine. As a result, you can simulate the multi-machine environment of the BEA Tuxedo system without a network of machines.

How to Create the Database in MP Mode

1. Set the environment by typing the following:

   ```
   . ./bankvar
   ```

   Note: You may have already set your environment variables. For detailed instructions, see "How to Set Environment Variables."

2. Run `crbankdb` to create the database for this site.

3. On each additional machine in your BEA Tuxedo system network, edit `bankvar` to provide the pathname for the `FSCONFIG` variable that is used for that site in the configuration file, `ubbmp`. Then repeat steps 1 and 2.

See Also

- "Preparing bankapp Files and Resources" on page 3-24

Step 5: Preparing for an XA-Compliant Resource Manager

To run `bankapp` with an alternative XA-compliant resource manager, you must modify various files. This section describes the following:

- How to Change the bankvar File
- How to Change the bankapp Services
- How to Change the bankapp.mk File
- How to Change crbank and crbankdb
- How to Change the Configuration File

How to Change the bankvar File

1. Review the following environment variables that are assigned the values shown here, by default, to create the BEA Tuxedo system database:

   ```
   BLKSIZE=512
   DBNAME=bankdb
   DBPRIVATE=no
   DIPCKEY=80953
   FSCONFIG=${APPDIR}/bankdl1
   ```

   Note: These environment variables pertain to the BEA Tuxedo system only; you may need to set different environment variables or other mechanisms depending on your specific database management system requirements.

2. Change the value of these variables as needed to create the database for your resource manager.

How to Change the bankapp Services

Because all database access in `bankapp` is performed with embedded SQL statements, if your new resource manager supports SQL, you should have no problem. The utility `appinit.c` includes calls to `tpopen()` and `tpclose()`.

How to Change the bankapp.mk File

1. Edit the RM parameter in `bankapp.mk` to name the new resource manager.
2. Ensure that the following entry is in the RM file.

   ```
   $TUXDIR/udataobj/RM
   ```

3. If necessary, change the name of the SQL compiler and its options. The name of the source file may or may not include .ec. You may have to specify a non-default for compiling the resulting .c file.

How to Change crbank and crbankdb

1. crbank may be ignored by your alternate resource manager. Its only functions are to reset variables and to run crbankdb three times.

2. crbankdb, on the other hand, requires close attention. The following code listing is the beginning of the crbankdb script. It is followed by an explanation of parts of the code that do not work with a resource manager that is not supplied with the BEA Tuxedo system.

Listing 3-17 Excerpt from the crbankdb Script

```
#Copyright (c) BEA Systems, Inc.
#All rights reserved
#
# Create device list
#
dbadmin<<!
echo
crdl
# Replace the following line with your device zero entry
${FSCONFIG}        0         2560
!
#
# Create database files, fields, and secondary indices
#
sql<<!
echo
create database ${DBNAME} with (DEVNAME='${FSCONFIG}',
      IPCKEY=${DIPCKEY}, LOGBLOCKING=0, MAXDEV=1,
      NBLKTBL=200,       NBLOCKS=2048,  NBUF=70,       NFIELDS=80,
      NFILES=20,         NFLDNAMES=60,  NFREEPART=40,  NLCKTBL=200,
      NLINKS=80,         NPREDS=10,     NPROCTBL=20,   NSKEYS=20,
      NSWAP=50,          NTABLES=20,    NTRANTBL=20,   PERM='0666',
      STATISTICS='n'
)


create table BRANCH (
      BRANCH_ID             integer not null,
      BALANCE               real,
      LAST_ACCT             integer,
```

```
      LAST_TELLER           integer,
      PHONE                 char(14),
      ADDRESS               char(60),
      primary key(BRANCH_ID)
) with (
      FILETYPE='hash',      ICF='PI',     FIELDED='FML',
      BLOCKLEN=${BLKSIZE},  DBLKS=8,      OVBLKS=2
)
!
```

The first 40 lines give you an idea of what needs to be changed and what may be kept unchanged. As you can see, `crbankdb` is made up of two documents that provide input to the `dbadmin` and `sql` shell commands. The first `here` file is passed to the BEA Tuxedo system command `dbadmin` to create a device list for the database.

This command does not work with non-BEA Tuxedo resource managers. Other commands may be needed to create table spaces and/or grant the correct privileges.

How to Change the Configuration File

In the `GROUPS` section, specify appropriate values (that is, values that are recognized by your resource manager) for the `TMSNAME` and `OPENINFO` parameters.

How to Integrate bankapp with Oracle (XA RM) for a Windows 2003 Platform

1. Edit the `nt\bankvar.cmd` and supply suitable values for the following environment variables:

   `TUXDIR : Root directory for the BEA TUXEDO system installation`

   `APPDIR : Application directory in which bankapp files are located`

   `ORACLE_HOME : Root directory of the Oracle8 installation`

   `ORACLE_SID : Oracle System ID`

   `BLK_SIZE: Logical block size`

   `DBNAME: default name of the database to be used by database utilities and database creation scripts`

   `DBPRIVATE: indicates whether database is to be opened in share or private mode (yes or no)`

```
FSCONFIG:Universal Device List for database

PATH=%TUXDIR%\bin;%TUXDIR%\include;%TUXDIR%\lib;%ORACLE_HOME%\bin;%P
ATH%

INCLUDE=%ORACLE_HOME%\rdbms80\xa;
%ORACLE_HOME%\pro80\c\include;%include%

NLSPATH=%TUXDIR%\locale\C

PLATFORM=inwnt40

LIB=%TUXDIR%\lib; %ORACLE_HOME%\pro80\lib\msvc;
%ORACLE_HOME%\rdbms80\xa; %lib%;
```

2. Run the script to set up the environment:

   ```
   >bankvar
   ```

3. Edit the `TUXDIR\udataobj\RM` file as follows:

   – Append the following line to the `$TUXDIR\udataobj\RM` file:

   ```
   Oracle_XA;xaosw;%ORACLE_HOME%\pro80\lib\msvc\sqllib80.lib
   %ORACLE_HOME%\RDBMS80\XA\xa80.lib
   ```

   or if Oracle exists over the network:

   – Map the machine to a drive, for example, F.

   – Append the following line to the `$TUXDIR\udataobj\RM` file:

   ```
   Oracle_XA;xaosw;f:\orant\pro80\lib\msvc\sqllib80.lib
   f:\orant\RDBMS80\XA\xa80.lib
   ```

   – Remove any previous entry of Oracle_XA in the `RM` file>

4. Build the Transaction Manager Server for Oracle8:

   ```
   cd $APPDIR
   buildtms -r Oracle_XA -o TMS_ORA
   ```

5. Edit the `nt\bankapp.mak` file as indicated in the following table.

Task	Value
Specify values for the following environment variables.	`TUXDIR`=Root directory for the BEA Tuxedo system installation
	`APPDIR`=Application directory in which `bankapp` files are located
	`RM=Oracle_XA`

Task	Value
	`ORACLE_LIBS=$(ORACLE_HOME)\PRO80\LIB`
	`RMNAME=Oracle_XA`
	`SQLPUBLIC=$(ORACLE_HOME)\PRO80\C\INCLUDE`
	`CFLAGS=$(HOST) -DNOWHAT=1 $(CGFLAGS) $(DFML32)`
	`CGFLAGS=-DWIN32 -W3 -MD -nologo`
	`ORACLE_DIR=$(ORACLE_HOME)\bin`
	`INCDIR=$(TUXDIR)\include`
	`CC=cl`
In the `.ec.c` section, *Edit rules for creating C programs from embedded SQL programs*, (use the proc compiler), set the following values.	`set TUXDIR=$(TUXDIR) & $(ORACLE_DIR)\proc80 mode=ansi release_cursor=yes include=$(SQLPUBLIC) include=$(INCDIR) $(SQL_PLATFORM_INC) -c iname=$*.ec`
In the `.c.obj` section, *Edit rule for creating object files from C programs*, set the following values.	`$(CC) -c $(CFLAGS) $(SQLPUBLIC) $(INCLUDE) $*.c`

6. Update the `*.ec` files. Use Oracle SQL commands.
7. Run the makefile:

```
copy nt\bankapp.mak to %APPDIR%
nmake -f bankapp.mak
```

8. Edit `nt\ubbshm` as follows:

```
USER_ID=0
GROUP_ID=0
UNAME_SITE1=nodename returned by hostname
TUXDIR=same as specified in bankvar
APPDIR=same as specified in bankvar
```

9. In the `GROUPS` section of the configuration file, enter the following changes:

```
TMSNAME=TMS_ORA
BANKB1 GRPNO=1
OPENINFO="Oracle_XA:Oracle_XA+Acc=P/user1/PaSsWd1+SesTm=0+LogDir=."
```

```
[
Oracle_XA +
 required fields:
 Acc=P/oracle_user_id/oracle_password +
 SesTm=Session_time_limit (maximum time a transaction can be inactive) +
 optional fields:
 LogDir=logdir (where XA library trace file is located) +
 MaxCur=maximum_#_of_open cursors +
 SqlNet=connect_string (if Oracle exists over the network)
 (eg. SqlNet=hqfin@NEWDB indicates the database with sid=NEWDB accessed
at host hqfin by TCP/IP)
]
BANKB2 GRPNO=2
OPENINFO="Oracle_XA:Oracle_XA+Acc=P/user2/PaSsWd2+SesTm=0+LogDir=."
BANKB3 GRPNO=3
OPENINFO="Oracle_XA:Oracle_XA+Acc=P/user3/PaSsWd3+SesTm=0+LogDir=."
```

10. Create the BEA Tuxedo configuration binary file:

    ```
    tmloadcf -y nt/ubbshm
    ```

11. Create the device list and the `TLOG` device on the master machine:

    ```
    crtlog -m
    ```

12. Start up the Oracle database instance if not already started.

13. Boot the BEA Tuxedo system servers:

    ```
    tmboot -y
    ```

14. Ensure that the view `v$XATRANS$` exists on the database. (The view `V$XATRANS$` should have been created during the XA library installation.)

15. If the `v$XATRANS$` view has not been created, create it as follows:

    – Ensure that the environment variables `ORACLE_HOME` and `ORACLE_SID` are set.

    – Log in to the database as user `SYS`:

    ```
    Execute the sql script ${ORACLE_HOME}/RDBMS80/ADMIN/XAVIEW.sql
    ```

    – Grant select privileges to this view for all Oracle account applications that will use the XA library.

16. Create the `bankapp` database and database objects for Oracle RM:

    – Log in to any of the Oracle utilities SQL*plus or SQL*DBA as any Oracle user.

    – `notepad crbank-ora8.sql`

– When Oracle8 is installed, a sample database is created. You can use this database for the `bankapp` application. The `sql` script provided, creates a new tablespace in the database to hold all the database objects of `bankapp`.The script prompts for the Oracle system user password as well as a full path name of a file to use as the new tablespace.

– Edit `crbank-ora8.sql` as follows:

```
WHENEVER OSERROR EXIT ;
/*Obtain the password for user "system" */
PROMPT
PROMPT
PROMPT -- Some of the operations require "system" user privileges
PROMPT -- Please specify the Oracle "system" user password
PROMPT
ACCEPT syspw CHAR PROMPT 'system passwd:' HIDE ;
CONNECT system/&syspw ;
SHOW user ;
PROMPT
/* Create a new tablespace in the default DB for use with "bankapp" */
DROP TABLESPACE bank1
    INCLUDING CONTENTS
CASCADE CONSTRAINTS;
PROMPT
PROMPT
PROMPT -- Will create a 3MB tablespace for bankapp ;
PROMPT -------- Please specify full pathname below for Datafile ;
PROMPT -------- Ex: %ORACLE_HOME%/dbs/bankapp.dbf
PROMPT
ACCEPT datafile CHAR PROMPT 'Datafile:' ;

CREATE TABLESPACE bank1
    DATAFILE '&datafile'  SIZE 3M REUSE
    DEFAULT STORAGE (INITIAL 10K   NEXT 50K
            MINEXTENTS 1           MAXEXTENTS 120
            PCTINCREASE 5)
    ONLINE;

/***************** Create a user called "user1" ***************/

DROP USER user1 CASCADE;

PROMPT Creating user "user1"

CREATE USER user1 IDENTIFIED by PaSsWd1
    DEFAULT TABLESPACE bank1
    QUOTA UNLIMITED ON bank1 ;

GRANT CREATE SESSION TO user1 ;
GRANT CREATE TABLE TO user1 ;
```

```
CONNECT user1/PaSsWd1 ;
SHOW user ;

PROMPT Creating database objects for user "user1" ;
PROMPT Creating table "branch" ;

    CREATE TABLE branch (
          branch_id       NUMBER NOT NULL PRIMARY KEY,
          balance         NUMBER,
          last_acct       NUMBER,
          last_teller     NUMBER,
          phoneCHAR(14),
          address         CHAR(60)

 )
          STORAGE         (INITIAL 5K   NEXT 2K
                          MINEXTENTS 1 MAXEXTENTS 5  PCTINCREASE 5) ;

PROMPT Creating table "account" ;

    CREATE TABLE account (
          account_id      NUMBER NOT NULL PRIMARY KEY,
          branch_id       NUMBER NOT NULL,
          ssn             CHAR(12) NOT NULL,
          balance         NUMBER,
          acct_type       CHAR,
          last_name       CHAR(20),
          first_name      CHAR(20),
          mid_init        CHAR,
          phoneCHAR(14),
          address         CHAR(60)

 )
          STORAGE         (INITIAL 50K NEXT 25K
                          MINEXTENTS 1 MAXEXTENTS 50   PCTINCREASE 5) ;

PROMPT Creating table "teller" ;
     CREATE TABLE teller (
          teller_id       NUMBER NOT NULL PRIMARY KEY,
          branch_id       NUMBER NOT NULL,
          balance         NUMBER,
          last_name       CHAR(20),
          first_name      CHAR(20),
          mid_init        CHAR
 )
          STORAGE         (INITIAL 5K  NEXT 2K
                          MINEXTENTS 1  MAXEXTENTS 5    PCTINCREASE 5) ;
PROMPT Creating table "history" ;
    CREATE TABLE history (
          account_id      NUMBER NOT NULL,
```

```
        teller_id       NUMBER NOT NULL,
        branch_id       NUMBER NOT NULL,
        amount          NUMBER

)
        STORAGE         (INITIAL 400K   NEXT 200K
                        MINEXTENTS 1   MAXEXTENTS 5     PCTINCREASE 5) ;
```

17. Write the code to create *user2* and *user3* with passwords *PaSsWd2* and *PaSsWd3*, respectively, following the method described in the above steps:

    ```
    SQL*plus> start $APPDIR/ crbank-ora8.sql
    ```

18. Populate the database:

    ```
    nt\populate
    ```

19. Generate transactions against the database:

    ```
    driver
    ```

20. Run the `bankapp` client:

    ```
    run
    ```

21. Shut down the application:

    ```
    tmshutdown -y
    ```

See Also

- "Preparing bankapp Files and Resources" on page 3-24

Step 6: How to Edit the Configuration File

A configuration file defines how an application runs. `bankapp` is delivered with two configuration files in the text format described in `UBBCONFIG(5)`: `ubbshm`, which defines an application on a single computer, and `ubbmp`, which defines a networked application.

Initialization scripts are provided in the sample applications. In addition, you can generate completed configuration files by `.sh` for any number up to 10 for your configuration and machines.

1. In a text editor, familiarize yourself with the `ubbshm` and `ubbmp` configuration files for `bankapp`.

Listing 3-18 ubbmp Configuration File

```
     #Copyright (c) 1999 BEA Systems, Inc.
     #All rights reserved


     *RESOURCES
     IPCKEY               80952
001  UID                  <user id from id(1)>
002  GID                  <group id from id(1)>
     PERM                 0660
     MAXACCESSERS         40
     MAXSERVERS           35
     MAXSERVICES          75
     MAXCONV              10
     MAXGTT               20
     MASTER               SITE1,SITE2
     SCANUNIT             10
     SANITYSCAN           12
     BBLQUERY             180
     BLOCKTIME            30
     DBBLWAIT          6
     OPTIONS           LAN,MIGRATE
     MODEL             MP
     LDBAL             Y
     ##SECURITY        ACL
     #
     *MACHINES
003  <SITE1's uname>  LMID=SITE1
004                   TUXDIR="<TUXDIR>"
005                   APPDIR="<APPDIR>"
                      ENVFILE="<APPDIR>/ENVFILE"
                      TLOGDEVICE="<APPDIR>/TLOG"
                      TLOGNAME=TLOG
                      TUXCONFIG="<APPDIR>/tuxconfig"
006                   TYPE="<machine type>"
                      ULOGPFX="<APPDIR>/ULOG"
007  <SITE2's uname>  LMID=SITE2
                      TUXDIR="<TUXDIR>"
                      APPDIR="<APPDIR>"
                      ENVFILE="<APPDIR>/ENVFILE"
                      TLOGDEVICE="<APPDIR>/TLOG"
                      TLOGNAME=TLOG
                      TUXCONFIG="<APPDIR>/tuxconfig"
                      TYPE="<machine type>"
                      ULOGPFX="<APPDIR>/ULOG"
     #
     *GROUPS
```

```
#
# Group for Authentication Servers
#
Group for Application Queue (/Q) Servers
#
##QGRP1      LMID=SITE1    GRP=102
##       TMSNAME=TMS_QM    TMSCOUNT=2
##       OPENINFO=”TUXEDO/QM:<APPDIR>/qdevice:QSP_BANKAPP”
#
# Group for Event Broker Servers
#
##EVBGRP1     LMID=SITE1    GRPNO=104


DEFAULT: TMSNAME=TMS_SQL   TMSCOUNT=2
BANKB1   LMID=SITE1        GRPNO=1


008  OPENINFO="TUXEDO/SQL:<APPDIR>/bankdl1:bankdb:readwrite"
     BANKB2   LMID=SITE2         GRPNO=2
     OPENINFO="TUXEDO/SQL:<APPDIR>/bankdl2:bankdb:readwrite"
     *NETWORK
009  SITE1     NADDR="<network address of SITE1>"
010            NLSADDR="<network listener address of SITE1>"
011  SITE2     NADDR="<network address of SITE2>"
012            NLSADDR="<network listener address of SITE2>"
```

2. To enable the application password feature, add the following line to the RESOURCES section of ubbshm or ubbmp:

   ```
   SECURITY       APP_PW
   ```

3. In both configuration files, you may notice that the values of some parameters are enclosed in angle brackets (< >). Values shown in angle brackets are generic; you need to replace them with values that pertain to your installation. All of these fields occur within the RESOURCES, MACHINES, and GROUPS sections in both files. In ubbmp, the NETWORK section also has values you must replace. The following table shows the ubbmp through the NETWORK section and illustrates all the changes you need to make in the RESOURCES, MACHINES, and GROUPS sections if you are bringing up a single-machine application.

Table 3-3 Explanation of Values

Line	String to Be Replaced	Description
001	`UID`	The effective user ID (UID) for the owner of the bulletin board IPC structures. In a multiprocessor configuration, the value must be the same on all machines. To avoid problems, use the same UID as that of the owner of the BEA Tuxedo system software.
002	`GID`	The effective group ID (GID) 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 name of the machine. (For UNIX platforms, use the value produced by the UNIX command: `uname -n`)
004	`TUXDIR`	The absolute path name of the root directory for the BEA Tuxedo software. Replace all occurrences of <*TUXDIR*> in the file with the specified path name.
005	`APPDIR`	The absolute path name of the directory in which the application runs. Make this a global change so that all occurrences of <*APPDIR*> in the file are replaced by the specified path name.
006	machine type	An identifying string used in networked applications that include machines of different types. The BEA Tuxedo system checks the value of machine type for each machine communicating with another. If the system identifies two machines with different machine types trying to communicate, it invokes the message encode and decode routines to convert the data being transmitted to a form recognizable by both machines.
007	`SITE2` name	The name of the second machine. (For UNIX platforms, use the value produced by the UNIX command: `uname -n`)
008	`OPENINFO`	The statement here and in the following entry are in a format understood by BEA Tuxedo system resource managers. They need to be changed (or removed) to meet the requirements of other resource managers.

Table 3-3 Explanation of Values (Continued)

Line	String to Be Replaced	Description
009	Network address of SITE1	The full address of the network listener for the BRIDGE process on this machine.
010	Network listener address of SITE1	The address of the network listener for the tlisten process on this machine.
011	Network address of SITE2	The full address of the network listener for the BRIDGE process on this machine. This value must be different on each machine.
012	Network listener address of SITE2	The address of the network listener for the tlisten process on this machine.

See Also

- "Preparing bankapp Files and Resources" on page 3-24
- UBBCONFIG(5) in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*
- "What Is the Configuration File?" in *Setting Up a BEA Tuxedo Application*

Steps 7 and 8: Creating a Binary Configuration File and Transaction Log File

Before Creating the Binary Configuration File

Before creating the binary configuration file, you need to be in the directory in which your bankapp files are located and you must set the environment variables. Complete the following tasks.

1. Go to the directory in which your bankapp files are located.
2. Set the environment variables by entering

```
. ./bankvar
```

Note: If you bring up bankapp in SHM mode, you do not have to create the tlisten process or create a transaction log on another machine.

How to Load the Configuration File

Once you have finished editing the configuration file, you must load it into a binary file on your MASTER machine. The name of the binary configuration file is TUXCONFIG; its path name is defined in the TUXCONFIG environment variable. 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 this requirement is not met, you may have permission problems in running bankapp.

1. To create TUXCONFIG, enter the following command:

   ```
   tmloadcf ubbmp
   ```

   While the configuration file is being loaded, you are prompted several times to confirm that you want to install this configuration, even if doing so means an existing configuration file must be overwritten. If you want to suppress such prompts, include the -y option on the command line.

2. If you want the amount of IPC resources needed by your application to be calculated by the BEA Tuxedo system, include the -c option on the command line.

TUXCONFIG can be installed only on the MASTER machine; it is propagated to other machines by tmboot when the application is booted.

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 are required to supply the password.

tmloadcf parses the text configuration file (UBBCONFIG) for syntax errors before it loads it, so if there are errors in the file, the job fails.

How to Create the Transaction Log (TLOG) File

The TLOG is the transaction log used by the BEA Tuxedo system in the management of global transactions. Before an application can be booted, an entry for the TLOG must be created in every file on every machine in the application, and a file for the log itself must be created on the MASTER machine.

bankapp provides a script called crtlog that creates a device list and a TLOG for you. The device list is created using the TLOGDEVICE variable from bankvar.

1. To create your TLOG and device list, enter the command on the MASTER machine as follows:

   ```
   crtlog -m
   ```

   Note: In a production environment, the device list may be the same as that used for the database.

2. On all other machines, do not specify -m; when the system is booted, the BBL on each non-MASTER machine creates the log.

If you are using a non-XA resource manager, you do not need a transaction log.

See Also

- "Preparing bankapp Files and Resources" on page 3-24

Step 9: How to Create a Remote Service Connection on Each Machine

tlisten is the listener process that provides a remote service connection for processes such as tmboot between machines in a BEA Tuxedo application. It must be installed on all the machines in your network as defined in the NETWORK section of the configuration file.

Instructions for starting tlisten are provided in the "Starting the tlisten Process" in *Installing the BEA Tuxedo System*.

1. We recommend starting a separate tlisten process for bankapp. To do so, enter the following command:

   ```
   tlisten -l nlsaddr
   ```

   The nlsaddr value must be the same as that specified for the NLSADDR parameter for this machine in your configuration file. Because this value changes from one machine to another, it is important that your tlisten argument agrees with your configuration file specification.

   Note: Detection of an error in this specification is not easy. tmloadcf does not check for agreement between your configuration file and your tlisten command. If the two addresses do not match, then the application will probably fail to boot on the machine with the mismatched value of nlsaddr or on which the tlisten process has not been started.

The log file used by `tlisten` is separate from all other BEA Tuxedo system log files, but one log can be used by more than one `tlisten` process. The default filename is `TUXDIR/udataobj/tlog`.

How to Stop the Listener Process (tlisten)

`tlisten` is designed to run as a daemon process. For suggestions about incorporating it in startup scripts or running it as a cron job, see `tlisten(1)` in the *BEA Tuxedo Reference Manual*.

For `bankapp`, you may prefer simply to start it and bring it down as you need it. To bring it down, send it a `SIGTERM` signal such as the following:

kill -15 *pid*

Note: In a Windows 2003 environment, you can start and stop the listener process in two ways: using the `tlisten` on the command line or using the Control Panel.

Sample tlisten Error Messages

If no remote `tlisten` is running, the boot sequence is displayed on your screen as follows:

```
Booting admin processes
exec DBBL -A :
      on MASTER -> process id=17160Started.
exec BBL -A :
      on MASTER -> process id=17161Started.
exec BBL -A :
      on NONMAST2 -> CMDTUX_CAT:814: cannot propagate TUXCONFIG file
tmboot: WARNING: No BBL available on site NONMAST2.
      Will not attempt to boot server processes on that site.
exec BBL -A :
      on NONMAST1 -> CMDTUX_CAT:814: cannot propagate TUXCONFIG file
tmboot: WARNING: No BBL available on site NONMAST1.
      Will not attempt to boot server processes on that site.
2 processes started.
and messages such as these will be in the ULOG:
133757.mach1!DBBL.17160: LIBTUX_CAT:262: std main starting
133800.mach1!BBL.17161: LIBTUX_CAT:262: std main starting
133804.mach1!BRIDGE.17162: LIBTUX_CAT:262: std main starting
133805.mach1!tmboot.17159: LIBTUX_CAT:278: Could not contact NLS on NONMAST2
133805.mach1!tmboot.17159: LIBTUX_CAT:276: No NLS available for remote
  machine NONMAST2
133806.mach1!tmboot.17159: LIBTUX_CAT:276: No NLS available for remote
  machine NONMAST2
133806.mach1!tmboot.17159: CMDTUX_CAT:850: Error sending TUXCONFIG
  propagation request to TAGENT on NONMAST2
```

```
    133806.mach1!tmboot.17159: WARNING: No BBL available on site NONMAST2.
           Will not attempt to boot server processes on that site.
    133806.mach1!tmboot.17159: LIBTUX_CAT:278: Could not contact NLS on NONMAST1
    133806.mach1!tmboot.17159: LIBTUX_CAT:276: No NLS available for
      remote machine NONMAST1
    133806.mach1!tmboot.17159: LIBTUX_CAT:276: No NLS available for
     remote machine NONMAST1
    133806.mach1!tmboot.17159: CMDTUX_CAT:850: Error sending TUXCONFIG
      propagation request to TAGENT on NONMAST1
    133806.mach1!tmboot.17159: WARNING: No BBL available on site NONMAST1.
            Will not attempt to boot server processes on that site.
```

If tlisten is started with the wrong machine address, the following messages appear in the tlisten log.

```
    Mon Aug 26 10:51:56 1991; 14240; BEA TUXEDO System Listener Process Started
    Mon Aug 26 10:51:56 1991; 14240; Could not establish listening endpoint
    Mon Aug 26 10:51:56 1991; 14240; Terminating listener process, SIGTERM
```

See Also

- "Preparing bankapp Files and Resources" on page 3-24
- tlisten(1)
- tmadmin(1)
- tmloadcf(1)

Running bankapp

This documentation leads you through the procedures for booting bankapp, testing it by running various client programs and transactions, and shutting it down when you have finished. Click on any of the following tasks for instructions on completing that task.

Step 1: How to Prepare to Boot

1. Before booting bankapp, verify that your machine has enough IPC resources to support your application. To generate a report on IPC resources, run the `tmboot` command with the `-c` option.

   Note: Because insufficient IPC resources may lead to a boot failure, it is imperative that you ensure you have appropriate values specified for your configuration.

Listing 3-19 IPC Report

```
Ipc sizing (minimum /T values only)
                Fixed Minimums Per Processor
SHMMIN: 1
SHMALL: 1
SEMMAP: SEMMNI
                Variable Minimums Per Processor
                SEMUME,           A                              SHMMAX
                SEMMNU,           *                                *
Node            SEMMNS  SEMMSL  SEMMSL  SEMMNI  MSGMNI  MSGMAP  SHMSEG
------          ------  ------  ------  ------  ------  ------  ------
sfpup               60       1      60   A + 1      10      20     76K
sfsup               63       5      63   A + 1      11      22     76K
where 1 <= A <= 8.
```

2. Add the number of application clients used per processor to each `MSGMNI` value. `MSGMAP` should be twice `MSGMNI`.

3. Compare the minimum IPC requirements to the parameters set for your machine. The location of these parameter settings is platform-dependent:

   – On many UNIX system platforms, machine parameters are defined in `/etc/conf/cf.d/mtune`.

   – On Windows 2003 platforms, machine parameters are set and displayed through a control panel.

See Also

- “Running bankapp” on page 3-54

Step 2: How to Boot bankapp

1. Set the environment:

   ```
   . ./bankvar
   ```

2. Boot the application by entering the following:

   ```
   tmboot
   ```

   The following prompt is displayed:

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

   A running report such as the following is displayed:

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

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

If you prefer, you can boot only a portion of the configuration. For example, to boot only administrative servers, include the `-A` option. If no options are specified, the entire application is booted.

In addition to reporting on the number of servers booted, `tmboot` also sends messages to the `ULOG`.

See Also

- "Running bankapp" on page 3-54

Step 3: How to Populate the Database

The `populate.sh` script is provided to put records into the database so you can run `bankapp` and test its functionality. `populate` is a one line script that pipes records from a program called `gendata` to the system server, `ud`. The `gendata` program creates records for 10 branches, 30 tellers, and 200 accounts. A record of the files created is kept in `pop.out`, so you can use values in the database when forming your sample service requests.

To run the script, enter `populate`.

Note: The output file that was created by the populate script, `pop.out`, can be used to provide account numbers, branch IDs, and other fields you can specify, so your service requests produce some output.

See Also

- "Running bankapp" on page 3-54
- `tmboot(1)` in the *BEA Tuxedo Command Reference*
- `ud, wud(1)` in the *BEA Tuxedo Command Reference*
- `userlog(3c)` in the *BEA Tuxedo ATMI C Function Reference*
- "What Is the User Log (ULOG)?" in *Administering a BEA Tuxedo Application at Run Time*
- "How to Boot the Application" in *Administering a BEA Tuxedo Application at Run Time*
- "How to Shut Down Your Application" in *Administering a BEA Tuxedo Application at Run Time*

Step 4: How to Test bankapp Services

1. If you are logging in cold to a running system, you must set your environment for `bankapp`. To do so, enter the following command:

   ```
   . ./bankvar
   ```

2. Run the `audit` client program. To execute the `audit` client program, enter the following command:

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

   specifying either `-a` for account balances or `-t` for teller balances. If you specify a `branch_id`, the report is limited to the branch specified; if you do not, the report includes data for all branches. For sample account numbers, branch IDs, and other values that you can provide as input to audit, use values listed in `pop.out`, the output of the populate program.

3. Run `auditcon`. To start the conversational version of the audit program, enter the following command:

   ```
   auditcon
   ```

   The program displays the following message on your terminal:

```
to request a TELLER or ACCOUNT balance for a branch,
type the letter t or a, followed by the branch id,
followed by <return>
for ALL TELLER or ACCOUNT balances, type t or a <return>
q <return> quits the program
```

When you have typed your request and pressed return, the requested information is displayed on your terminal followed by the following message:

```
another balance request ??
```

4. The program continues to offer you this service until you enter a `q`.
5. Use the driver program. By default, the driver program generates 300 transactions. You can change that number with the `-n` option, as in the following example. The command

   ```
   driver -n1000
   ```

   specifies that the program should run for 1,000 loops.

   `driver` is a script that generates a series of transactions to simulate activity on the system. It is included as part of `bankapp` so you can get realistic-looking statistics by running `tmadmin` commands.

See Also

- "Running bankapp" on page 3-54

Step 5: How to Shut Down bankapp

To bring down `bankapp`, enter the `tmshutdown(1)` command with no arguments, from the `MASTER` machine, as follows:

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

Server Id = 1 Group Id = BANKB1 Machine = Site1:  shutdown succeeded.
```

Running this command (or the shutdown command of `tmadmin`) causes the following results:

- All application servers, gateway servers, TMS's, and administrative servers, are shut down.
- All associated IPC resources are removed.

See Also

- “Running bankapp” on page 3-54
- `tmadmin(1)` in the *BEA Tuxedo Command Reference*
- `tmshutdown(1)` in the *BEA Tuxedo Command Reference*

CHAPTER 4

Tutorial for CSIMPAPP, a Simple COBOL Application

This topic includes the following sections:

What Is CSIMPAPP?

`CSIMPAPP` is a basic sample ATMI application delivered with the BEA Tuxedo system. While instructions are written for the Microfocus COBOL compiler, these may vary depending on your specific compiler. To find out which COBOL platforms are supported by the BEA Tuxedo system, consult Appendix A, "BEA Tuxedo 10.0 Platform Data Sheets," in *Installing the BEA Tuxedo System*.

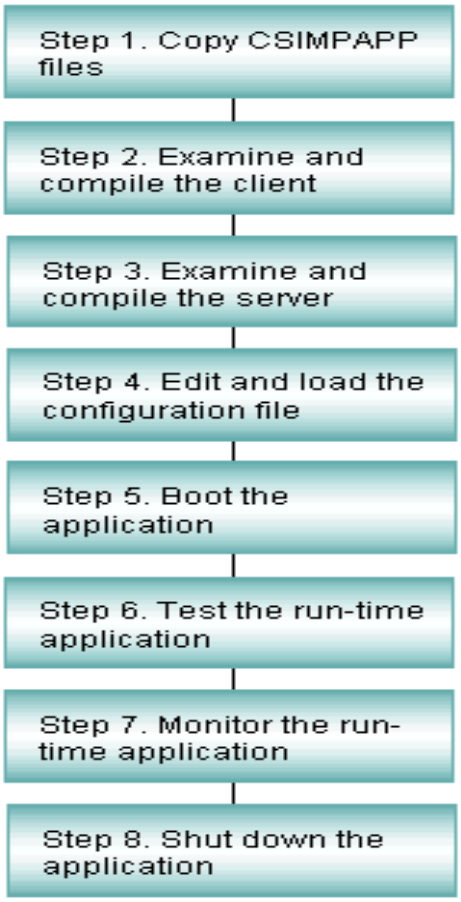
CSIMPAPP includes one client and one server. The server performs only one service: it accepts a string from the client and returns the same string in uppercase.

Preparing CSIMPAPP Files and Resources

This topic leads you through the procedures you must complete to develop CSIMPAPP. The following flow chart summarizes this procedure.

Click on each task for instructions on completing that task.

Figure 4-1 CSIMPAPP Development Process



Before You Begin

Before you can run this tutorial, the BEA Tuxedo ATMI client and server software must be installed so that the files and commands referred to are available. If you are responsible for installing the BEA Tuxedo system software, refer to *Installing the BEA Tuxedo System* for installation instructions. If the installation has already been done by someone else, you need to find out the pathname of the directory in which the software is installed (`TUXDIR`). 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 system commands.

What You Will Learn

After you complete this procedure, you will be able to understand the tasks clients and servers can perform, edit a configuration file for your own environment, and invoke `tmadmin` to check on the activity of your application. In short, you will understand the basic elements of all BEA Tuxedo applications—client processes, server processes, and a configuration file—and you will know how to use BEA Tuxedo system commands to manage your application.

Step 1: How to Copy the CSIMPAPP Files

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

   ```
   mkdir CSIMPDIR
   cd CSIMPDIR
   ```

Note: This step is suggested so you can see 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; do not use `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$TUXDIR/include"
PATH=$TUXDIR/bin:$APPDIR: $PATH
LD_LIBRARY_PATH=$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 directory structure and to execute BEA Tuxedo commands:

   – On Sun Solaris, `/usr/5bin` must be the first directory in your `PATH`.

   – On an AIX platform on the RS/6000, use `LIBPATH` instead of `LD_LIBRARY_PATH`.

   – On an HP-UX platform on the HP 9000, use `SHLIB_PATH` instead of `LD_LIBRARY_PATH`. You need to set `TUXCONFIG` to be able to load the configuration file as shown in step 4.

3. Copy the CSIMPAPP files:

   ```
   cp TUXDIR/samples/atmi/CSIMPAPP/* .
   ```

   Note: Later, you will edit some files and make them executable, so we recommend using copies 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 text form of the configuration file for the application.

   – WUBBCSIMPLE—the configuration file for the Workstation example.

   – ws—a directory with .MAK files for client programs for three workstation platforms.

Step 2: Examining and Compiling the Client

How to Examine the Client

Review the client program source code:

```
$ more CSIMPCL.cbl
```

The output is shown in the following list.

Listing 4-1 Source Code for CSIMPCL.cbl

```
1       IDENTIFICATION DIVISION.
2       PROGRAM-ID. CSIMPCL.
3       AUTHOR. Tuxedo DEVELOPMENT.
4       ENVIRONMENT DIVISION.
5       CONFIGURATION SECTION.
6       WORKING-STORAGE SECTION.
7   ******************************************************
8   * Tuxedo definitions
9   ******************************************************
10      01 TPTYPE-REC.
11      COPY TPTYPE.
12   *
13      01 TPSTATUS-REC.
14      COPY TPSTATUS.
15   *
16      01 TPSVCDEF-REC.
17      COPY TPSVCDEF.
18   *
19      01 TPINFDEF-REC VALUE LOW-VALUES.
20      COPY TPINFDEF.
21   ******************************************************
22   * Log messages definitions
23   ******************************************************
24      01 LOGMSG.
25             05 FILLER   PIC X(8) VALUE "CSIMPCL:".
26             05 LOGMSG-TEXT PIC X(50).
27      01 LOGMSG-LEN      PIC S9(9) COMP-5.
28   *
29      01 USER-DATA-REC   PIC X(75).
30      01 SEND-STRING     PIC X(100) VALUE SPACES.
31      01 RECV-STRING     PIC X(100) VALUE SPACES.
32   ******************************************************
33   * Command line arguments
34   ******************************************************
35   * Start program with command line args
36   ******************************************************
37
38   PROCEDURE
39   START-CSIMPCL.
40      MOVE LENGTH OF LOGMSG TO LOGMSG-LEN.
41      ACCEPT SEND-STRING FROM COMMAND-LINE.
42      DISPLAY “SEND-STRING:” SEND-STRING.
43
44      MOVE “Started” TO LOGMSG-TEXT.
45      PERFORM DO-TPINIT.
46      PERFORM DO-TPCALL.
```

```
47      DISPLAY “RECV-STRING:” RECV-STRING.
48      PERFORM DO-TPTERM.
49      PERFORM EXIT-PROGRAM.
50  ******************************************************
51  * Now register the client with the system.
52  ******************************************************
53  DO-TPINIT.
54     MOVE SPACES TO USRNAME.
55     MOVE SPACES TO CLTNAME.
56     MOVE SPACES TO PASSWD.
57     MOVE SPACES TO GRPNAME.
58     MOVE ZERO TO DATALEN.
59     SET TPU-DIP TO TRUE.
60
61   CALL "TPINITIALIZE" USING TPINFDEF-REC
62             USER-DATA-REC
63             TPSTATUS-REC.
64
65     IF NOT TPOK
66             MOVE "TPINITIALIZE Failed" TO LOGMSG-TEXT
67             PERFORM DO-USERLOG
68             PERFORM EXIT-PROGRAM
69     END-IF.
70
71  ******************************************************
72  * Issue a TPCALL
73  ******************************************************
74  DO-TPCALL.
75     MOVE 100 to LEN.
76     MOVE "STRING" TO REC-TYPE.
77     MOVE "CSIMPSRV" TO SERVICE-NAME.
78     SET TPBLOCK TO TRUE.
79     SET TPNOTRAN TO TRUE.
80     SET TPNOTIME TO TRUE.
81     SET TPSIGRSTRT TO TRUE.
82     SET TPCHANGE TO TRUE.
83
84     CALL "TPCALL" USING TPSVCDEF-REC
85                  TPTYPE-REC
86                  SEND-STRING
87                  TPTYPE-REC
88                  RECV-STRING
89                  TPSTATUS-REC.
90
91     IF NOT TPOK
92           MOVE "TPCALL Failed" TO LOGMSG-TEXT
93           PERFORM DO-USERLOG
94     END-IF.
95
```

```
96   ******************************************************
97   * Leave Tuxedo
98   ******************************************************
99   DO-TPTERM.
100    CALL "TPTERM" USING TPSTATUS-REC.
101    IF NOT TPOK
102           MOVE "TPTERM Failed" TO LOGMSG-TEXT
103           PERFORM DO-USERLOG
104    END-IF.
105
106  ******************************************************
107  * Log messages to the userlog
108  ******************************************************
109  DO-USERLOG.
110    CALL "USERLOG" USING LOGMSG
111           LOGMSG-LEN
112           TPSTATUS-REC.
113
114  ******************************************************
115  *Leave Application
116  ******************************************************
117  EXIT-PROGRAM.
118    MOVE "Ended" TO LOGMSG-TEXT.
119    PERFORM DO-USERLOG.
120    STOP RUN.
```

Table 4-1 Significant Lines in the CSIMPCL.cbl Source Code

Line(s)	File/Function	Purpose
11, 14, 17, 20	COPY	Command used to replicate files needed whenever BEA Tuxedo ATMI functions are used.
61	TPINITIALIZE	The ATMI function used by a client program to join an application.
84	TPCALL	The ATMI function used to send 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. An argument, LEN IN TPTYPE-REC, specifies the length of the record in USER-DATA-REC.

Table 4-1 Significant Lines in the CSIMPCL.cbl Source Code (Continued)

Line(s)	File/Function	Purpose
100	`TPTERM`	The ATMI function used to leave an application. A call to `TPTERM` is used to exit an application before performing a `STOP RUN`.
110	`USERLOG`	The function that displays the message returned from the server, the successful conclusion of `tpcall`.

How to Compile the Client

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

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

   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
   ```

   You now have an executable module called `CSIMPCL`.

See Also

- `buildclient(1)` in the *BEA Tuxedo Command Reference*
- `TPINITIALIZE(3cbl)` in the *BEA Tuxedo ATMI COBOL Function Reference*
- `TPTERM(3cbl)` in the *BEA Tuxedo ATMI COBOL Function Reference*
- `TPCALL(3cbl)` in the *BEA Tuxedo ATMI COBOL Function Reference*
- `USERLOG(3cbl)` in the *BEA Tuxedo ATMI COBOL Function Reference*

Step 3: Examining and Compiling the Server

How to Examine the Server

1. Review the source code from the `CSIMPSRV` ATMI server program:

   ```
   $ more CSIMPSRV.cbl
   ```

Listing 4-2 Source Code for CSIMPSRV.cbl

```
1       IDENTIFICATION DIVISION.
2       PROGRAM-ID. CSIMPSRV.
3       AUTHOR. BEA Tuxedo DEVELOPMENT.
4       ENVIRONMENT DIVISION.
5       CONFIGURATION SECTION.
6       WORKING-STORAGE SECTION.
7   ******************************************************
8   * Tuxedo definitions
9   ******************************************************
10      01 TPSVCRET-REC.
11      COPY TPSVCRET.
12   *
13      01 TPTYPE-REC.
14      COPY TPTYPE.
15   *
16      01 TPSTATUS-REC.
17      COPY TPSTATUS.
18   *
19      01 TPSVCDEF-REC.
20      COPY TPSVCDEF.
21   ******************************************************
22   * Log message definitions
23   ******************************************************
24      01 LOGMSG.
25               05 FILLER        PIC X(10) VALUE
26                     "CSIMPSRV :".
27               05 LOGMSG-TEXT   PIC X(50).
28      01 LOGMSG-LEN             PIC S9(9) COMP-5.
29   ******************************************************
31   * User defined data records
32   ******************************************************
33      01 RECV-STRING            PIC X(100).
34      01 SEND-STRING            PIC X(100).
35   *
36      LINKAGE SECTION.
37   *
38      PROCEDURE DIVISION.
39   *
40    START-FUNDUPSR.
41      MOVE LENGTH OF LOGMSG TO LOGMSG-LEN.
42      MOVE "Started" TO LOGMSG-TEXT.
43      PERFORM DO-USERLOG.
44
45   ******************************************************
46   * Get the data that was sent by the client
47   ******************************************************
```

```
48     MOVE LENGTH OF RECV-STRING TO LEN.
49     CALL "TPSVCSTART" USING TPSVCDEF-REC
50                     TPTYPE-REC
51                     RECV-STRING
52                     TPSTATUS-REC.
53
54     IF NOT TPOK
55            MOVE "TPSVCSTART Failed" TO LOGMSG-TEXT
56            PERFORM DO-USERLOG
57            PERFORM EXIT-PROGRAM
58     END-IF.
59
60     IF TPTRUNCATE
61            MOVE "Data was truncated" TO LOGMSG-TEXT
62            PERFORM DO-USERLOG
63            PERFORM EXIT-PROGRAM
64     END-IF.
65
66     INSPECT RECV-STRING CONVERTING
67     "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.
```

Table 4-2 Significant Lines in the CSIMPSRV.cbl Source Code

Line(s)	Routine	Purpose
49	TPSVCSTART	Routine used to start this service and to receive the service's parameters and data. After a successful call, the RECV-STRING contains the data sent by the client.
66-68	INSPECT statement	Statement that converts the input to uppercase (Microfocus-specific).
72	COPY TPRETURN	Command line that returns the converted string to the client with TPSUCCESS set.
79	USERLOG	Routine that logs messages used by the BEA Tuxedo system and applications.

2. During server initialization (that is, before the server starts processing service requests), the BEA Tuxedo system calls the TPSVRINIT subroutine. To familiarize yourself with TPSVRINIT, page through the source code for it.

   ```
   $ more TPSVRINIT.cbl
   ```

Listing 4-3 Source Code for TPSVRINIT.cbl

```
1   IDENTIFICATION DIVISION.
2   PROGRAM-ID. TPSVRINIT.
3   ENVIRONMENT DIVISION.
4   CONFIGURATION SECTION.
5  *
6   DATA DIVISION.
7   WORKING-STORAGE SECTION.
8  *
9   01 LOGMSG.
10      05 FILLER               PIC X(11) VALUE "TPSVRINIT :".
11      05 LOGMSG-TEXT          PIC X(50).
12  01 LOGMSG-LEN               PIC S9(9) COMP-5.
13  *
14   01 TPSTATUS-REC.
15   COPY TPSTATUS.
16  ********************************************************
17   LINKAGE SECTION.
18   01 CMD-LINE.
19    05 ARGC PIC 9(4) COMP-5.
```

```
20    05 ARG.
21      10 ARGS PIC X OCCURS 0 TO 9999 DEPENDING ON ARGC.
22   *
23    01 SERVER-INIT-STATUS.
24    COPY TPSTATUS.
25   ************************************************************
26    PROCEDURE DIVISION USING CMD-LINE SERVER-INIT-STATUS.
27    A-000.
28   MOVE LENGTH OF LOGMSG TO LOGMSG-LEN.
29   ************************************************************
30   * There are no command line parameters in this TPSVRINIT
31   ************************************************************
32      IF ARG NOT EQUAL TO SPACES
33             MOVE "TPSVRINIT failed" TO LOGMSG-TEXT
34             CALL "USERLOG" USING LOGMSG
35                    LOGMSG-LEN
36                    TPSTATUS-REC
37      ELSE
38             MOVE "Welcome to the simple service" TO LOGMSG-TEXT
39             CALL "USERLOG" USING LOGMSG
40                    LOGMSG-LEN
41                    TPSTATUS-REC
42      END-IF.
43   *
44    SET TPOK IN SERVER-INIT-STATUS TO TRUE.
45   *
46    EXIT PROGRAM.
```

A default is provided by the BEA Tuxedo system that writes a message to `USERLOG` indicating that the server has been booted.

How to Compile the Server

1. Run `buildserver` as follows to compile the ATMI server program.

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

   The executable file to be created is named `CSIMPSRV` and `CSIMPSRV.cbl` and `TPSVRINIT.cbl` are the input source files. The service being offered by the server `CSIMPSRV` is indicated by `-s CSIMPSRV`.

2. Check the results by displaying a list of the files in your current directory.

```
$ 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
```

You now have an executable module called CSIMPSRV.

See Also

- buildserver(1) in *BEA Tuxedo Command Reference*
- TPSVCSTART(3cbl) in the *BEA Tuxedo ATMI COBOL Function Reference*
- TPSVRINIT(3cbl) in the *BEA Tuxedo ATMI COBOL Function Reference*
- TPRETURN(3cbl) in the *BEA Tuxedo ATMI COBOL Function Reference*
- USERLOG(3cbl) in the *BEA Tuxedo ATMI COBOL Function Reference*

Step 4: Editing and Loading the Configuration File

How to Edit the Configuration File

1. In a text editor, familiarize yourself with the configuration file for CSIMPAPP.

Listing 4-4 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

DOMAINID                 UBBCSIMPLE
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"
#                   ENVFILE="/home/me/simpapp/envfile"
<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. For each *string* (that is, for each string shown in italic between angle brackets), substitute an appropriate value:
   - IPCKEY—use a value that will not conflict with any other users.
   - TUXCONFIG—provide the full pathname of the binary TUXCONFIG file.
   - TUXDIR—the full pathname of your BEA Tuxedo system root directory.
   - APPDIR—the full pathname of the directory in which you intend to boot the application; in this case, the current directory.
   - ENVFILE—the full pathname for the environment file to be used by mc, viewc, tmloadcf, and so on.
   - *machine-name*—the machine name as returned by the uname -n command on a UNIX platform.

Note: The pathnames for TUXCONFIG and TUXDIR must be identical to those you set and exported earlier. You must specify actual pathnames; references to pathnames through

environment variables (such as TUXCONFIG) are not acceptable. Do not forget to remove the angle brackets.

How to 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 by displaying a list of the files in your current directory:

```
$ 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 now have a file called TUXCONFIG (a new file system under the control of the BEA Tuxedo system).

See Also

- tmloadcf(1) in the *BEA Tuxedo Command Reference*
- UBBCONFIG(5) in the *File Formats, Data Descriptions, MIBs, and System Processes Reference*

Step 5: How to 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.
$
```

The Bulletin Board Liaison (BBL) is the administrative process that monitors the shared memory structures in the application. CSIMPSRV is the CSIMPAPP server that runs continuously, awaiting requests.

See Also

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

Step 6: How to Test the Run-time Application

To test CSIMPAPP, have the client submit a request:

```
$ CSIMPCL “hello world”
HELLO WORLD
```

Step 7: How to Monitor the Run-time Application

As the administrator, you can use the tmadmin command interpreter to check an application and make dynamic changes. To run tmadmin, you must set the TUXCONFIG variable

tmadmin can interpret and run over 50 commands. For a complete list, see tmadmin(1) in the *BEA Tuxedo Command Reference*. The following demonstrates two of the many tmadmin commands:

1. Enter the following command:

   ```
   tmadmin
   ```

   The following lines are displayed:

   ```
   tmadmin - Copyright (c) 1999 BEA Systems Inc.; 1991 USL. All rights
   reserved.
   >
   ```

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

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

```
> psc
Service Name  Routine Name  a.out Name  Grp Name  ID  Machine #  Done  Status
------------  ------------  ----------  --------  --  -------    ----  -------
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`.)

See Also

- `tmadmin(1)` in the *BEA Tuxedo Command Reference*

Step 8: How to Shut Down the Application

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

```
$ 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 is significant. First look at the format of a ULOG line:

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

Now look at an actual line.

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

See Also

- tmshutdown(1) in the *BEA Tuxedo Command Reference*
- USERLOG(3cbl) in the *BEA Tuxedo ATMI COBOL Function Reference*

CHAPTER 5

Tutorial for STOCKAPP, a Full COBOL Application

This topic includes the following sections:

What Is STOCKAPP?

STOCKAPP is a sample ATMI stocks application that is provided with the BEA Tuxedo system software. The application performs the following stock brokering functions: validates and updates a customer's account information, and executes buy and sell orders for stocks and/or funds.

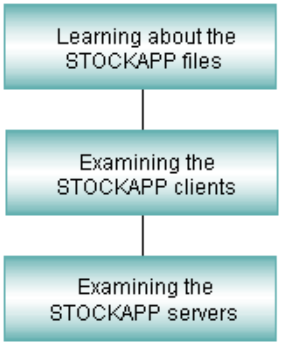
This documentation leads you, step-by-step, through the procedure you must perform to develop the STOCKAPP application. Once you have "developed" STOCKAPP through this tutorial, you will be ready to start developing applications of your own.

The STOCKAPP tutorial is presented in three sections:

Note: This information is focused on system users with some experience in application development, administration, or programming. We assume some familiarity with the BEA Tuxedo system software. A development license is required to build BEA Tuxedo applications.

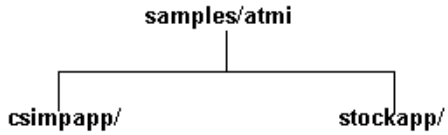
Familiarizing Yourself with STOCKAPP

This documentation provides a tour of the files, client, and services that make up the STOCKAPP application. Click on any of the following activities for more information about that part of the tour.



Learning About the STOCKAPP Files

The files that make up the STOCKAPP application are delivered in a directory called STOCKAPP, which is positioned as follows:



Exploring the Stock Application Files

The STOCKAPP directory contains the following files:

- Eight .cbl files
- Four clients: BUY.cbl, SELL.cbl, FUNDPR.cbl and FUNDUP.cbl
- One conversational server: FUNDUPSR.cbl
- Three files that are servers or are associated with servers
- Two servers to generate data or transactions for the application
- Files provided to facilitate the use of STOCKAPP as an example

The following table lists the files that make up STOCKAPP. The table lists the source files delivered with the BEA Tuxedo system software, files that are generated when the stock application is built, and a summary of the contents of each file.

Table 5-1 Purpose of the Stock Application Files

Source File	Generated File	Contents
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

Table 5-1 Purpose of the Stock Application Files (Continued)

Source File	Generated File	Contents
FUNDUP.cbl	FUNDUP.o FUNDUP	Client
FUNDUPSR.cbl	FUNDUPSR.o FUNDUPSR	Contains FUND UPDATE service
README		Online 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 the 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 the structure passed between the FUNDPR and FUNDUP clients and all the servers

See Also

- "Familiarizing Yourself with STOCKAPP" on page 5-2

Examining the STOCKAPP Clients

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

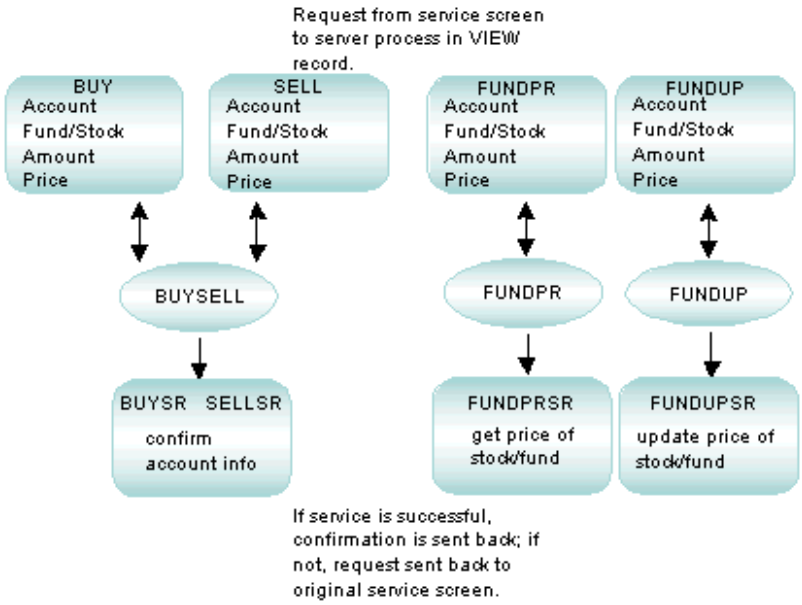
- Request/response mode, which is characterized by the sending of a single request for a service to be performed by the server and getting back a single response.

- 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

The following figure shows the hierarchy for `STOCKAPP`. The user selects one of the four service requests. The oval shapes in the illustration represent application services.

Figure 5-1 STOCKAPP Requests



Typed Buffers

Typed buffers are an essential part of the BEA Tuxedo system. In the BEA Tuxedo system, a typed buffer is designed to hold a specific data type. Six types are defined: `VIEW`, `STRING`, `CARRAY`, `X_OCTET`, `X_COMMON`, and `XML`. Applications have the ability to define additional types.

A Request/Response Client: BUY.cbl

BUY 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

Review the following sections of the BUY.cbl 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 functions are used. Similar to csimpl.cbl, BUY.cbl needs to call TPINITIALIZE to join the application; call TPCALL to make an RPC request to a service; and call TPTERM to leave an application. 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 Clients

View description files, of which cust is an example, are processed by the view compiler, viewc(1). Run view(c) to compile the view:

```
viewc-C-n

  cust.v
```

where 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.

See Also

- "What You Can Do Using the ATMI" in *Introducing BEA Tuxedo ATMI*

- "What Are Typed Buffers?" in *Introducing BEA Tuxedo ATMI*
- ATMI commands and functions in *BEA Tuxedo Command Reference* and *BEA Tuxedo ATMI C Function Reference*
- "Familiarizing Yourself with STOCKAPP" on page 5-2

Examining the STOCKAPP Servers

This topic provides the following information:

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

ATMI 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 `STOCKAPP` services use functions provided in the Application-to-Transaction Monitor Interface (ATMI) for performing the following tasks:

- Communicating synchronously or asynchronously with other services
- Defining global transactions
- Sending replies back to clients

STOCKAPP Services

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

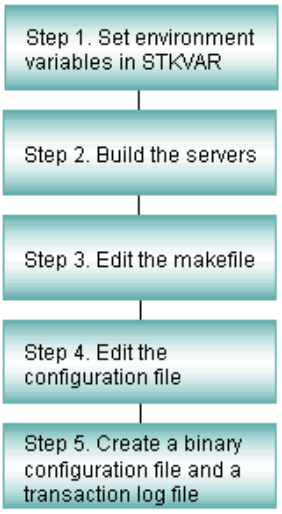
FUNDPRSR
: Price quote; offered by the PRICEQUOTE server; accepts a VIEW record as input

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

Preparing STOCKAPP Files and Resources

This documentation leads you through the procedures you must complete to create the files and other resources you need to run STOCKAPP.

Click on each task for instructions on completing that task.



Step 1: How to Set Environment Variables

Environment variables required for STOCKAPP are defined in the STKVAR file. The file is large (approximately 100 lines) because it includes extensive comments.

1. In a text editor, familiarize yourself with the STKVAR file. Line 9 ensures that TUXDIR is set. If it is not set, execution of the file fails with the following message:

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

2. Set TUXDIR to the root directory of your BEA Tuxedo system directory structure, and export it.

3. Another line in STKVAR sets APPDIR to the directory {TUXDIR}/samples/atmi/STOCKAPP which is the directory where STOCKAPP source files are located: APPDIR is a directory where the 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 enter the directory there. It does not have to be under TUXDIR.

   Note: Other variables specified in STKVAR play various roles in the sample application; you need to be aware of them when you are developing your own application. By including them in STKVAR, we provide you with a template that you may want to adapt at a later time for use with a real application.

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

   ```
   . ./STKVAR
   ```

Listing 5-1 STKVAR: Environment Variables for STOCKAPP

```
#ident    "@(#)samples/atmi: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
```

Additional Requirements

- On AIX, set `LIBPATH` instead of `LD_LIBRARY_PATH`.

- On HP-UX, set SHLIB_PATH instead of LD_LIBRARY_PATH.
- If your operating system is Sun Solaris, you need to: put /usr/5bin at the beginning of your PATH. The following command can be used:

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

  Use /bin/sh rather than csh for your shell.

See Also

- "Preparing STOCKAPP Files and Resources" on page 5-8

Step 2: Building Servers in STOCKAPP

buildserver is used to put together an executable ATMI 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.

The buildserver command is used in STOCKAPP.mk to compile and build each server in the stock application. (Refer to buildserver(1) in the *BEA Tuxedo Command Reference* for complete details.)

How to Build the BUYSELL Server

The BUYSELL ATMI 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.

1. Create the BUYSELL.o file (performed 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 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 -o option is used to assign a name to the executable output file. If no name is provided, the file is named SERVER.

– 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) in the *BEA Tuxedo Command Reference* 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, refer to *Administering a BEA Tuxedo Application at Run Time* and *Setting Up a BEA Tuxedo Application*.

– The -f option specifies the files that are used in the link-edit phase. Also refer to the -l option on the buildserver reference page. For more detail information on both of these options, refer to the "Building Servers" in *Programming BEA Tuxedo ATMI Applications Using COBOL*. 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 filename 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 topics on creating the STOCKAPP servers are important to your understanding of how the buildserver command is specified. However, in actual practice you are apt to incorporate the build into a makefile; that is the way it is done in STOCKAPP.

See Also

- “Familiarizing Yourself with STOCKAPP” on page 5-2
- `buildserver(1)`

Step 3: Editing the STOCKAPP.mk File

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

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

How to Edit the TUXDIR Parameter

Go to the following comment in `STOCKAPP.mk` 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=../..
```

You should set the `TUXDIR` parameter to the absolute pathname of the root directory of your BEA Tuxedo system installation.

How to Edit the APPDIR Parameter

You may want to give some thought to the setting of the `APPDIR` parameter. As `STOCKAPP` is delivered, `APPDIR` is set to the directory in which the `STOCKAPP` files are located, relative to `TUXDIR`. The following section of `STOCKAPP.mk` defines and describes the setting of `APPDIR`.

```
#
# 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)/samples/atmi/STOCKAPP
#
```

If you have copied the files to another directory, as suggested in the README file, you should set APPDIR 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.

How to Run the STOCKAPP.mk File

1. 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 &
   ```

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

See Also

- "Preparing STOCKAPP Files and Resources" on page 5-8

Step 4: How to Edit the Configuration File

The STOCKAPP configuration file defines how an application runs on a set of machines. STOCKAPP is delivered with a configuration file in text format described in UBBCONFIG(5). UBBCBSHM, defines an application on a single computer.

1. In a text editor, familiarize yourself with the configuration file for STOCKAPP.

Listing 5-2 UBBCBSHM 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                  5226164
      DOMAINID            STOCKAPP
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
```

2. To enable the application password feature, add the following line to the RESOURCES section of UBBCBSHM:

   ```
   SECURITY   APP_PW
   ```

3. You may notice that the values of some parameters are enclosed in angle brackets (<>). Values shown in angle brackets are generic; you need to replace them with values that pertain to your installation. All of these fields occur within the RESOURCES, MACHINES, and GROUPS sections in the file. The following table describes the values with which you must replace the angle-bracketed strings. For each string, substitute an appropriate value.

Table 5-2 Explanation of Values

Line	String to Be Replaced	Purpose
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 BEA Tuxedo 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. The BEA Tuxedo system 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.

See Also

- "Preparing STOCKAPP Files and Resources" on page 5-8
- `UBBCONFIG(5)` in *File Formats, Data Descriptions, MIBs, and System Processes Reference*

Step 5: Creating a Binary Configuration File

Before Creating the Binary Configuration File

Before creating the binary configuration file, you need to be in the directory in which your STOCKAPP files are located and you must set the environment variables. Complete the following tasks.

1. Go to the directory in which your STOCKAPP files are located.
2. Set the environment variables by entering:

   ```
   . ./STKVAR
   ```

How to Load the Configuration File

Once you have finished editing the configuration file, you must load it into a binary file on your MASTER machine. The name of the binary configuration file is TUXCONFIG; its path name is defined in the TUXCONFIG environment variable. 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 this requirement is not met, you may have permission problems in running STOCKAPP.

1. To create TUXCONFIG, enter the following command:

   ```
   tmloadcf UBBCBSHM
   ```

   While the configuration file is being loaded, you are prompted several times to confirm that you want to install this configuration, even if doing so means an existing configuration file must be overwritten. If you want to suppress such prompts, include the -y option on the command line.

2. If you want the amount of IPC resources needed by your application to be calculated by the BEA Tuxedo system, include the -c option on the command line.

   TUXCONFIG can be installed only on the MASTER machine; it is propagated to other machines by tmboot when the application is booted.

   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 are required to supply the password.

   tmloadcf parses the text configuration file (UBBCONFIG) for syntax errors before it loads it, so if there are errors in the file, the job fails.

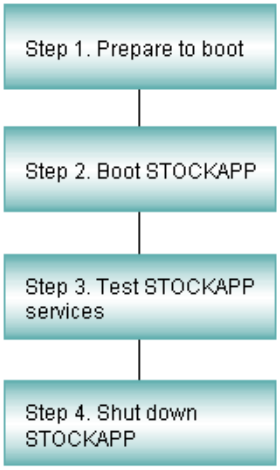
See Also

- "Preparing STOCKAPP Files and Resources" on page 5-8
- `tmloadcf(1)` in *BEA Tuxedo Command Reference*

Running STOCKAPP

This documentation leads you through the procedures for booting `STOCKAPP`, testing it by running various client programs and transactions, and shutting it down when you have finished.

Click on each task for instructions on completing that task.



Step 1: How to Prepare to Boot

1. Before booting `STOCKAPP`, verify that your machine has enough IPC resources to support your application. To generate a report on IPC resources, run the `tmboot` command with the `-c` option.

Listing 5-3 IPC Report

```
Ipc sizing (minimum /T values only)
                Fixed Minimums Per Processor
SHMMIN: 1
```

```
SHMALL: 1
SEMMAP: SEMMNI
                Variable Minimums Per Processor
                SEMUME,           A                              SHMMAX
                SEMMNU,           *                                *
Node            SEMMNS  SEMMSL  SEMMSL  SEMMNI  MSGMNI  MSGMAP  SHMSEG
------          ------  ------  ------  ------  ------  ------  ------
machine 1               60       1      60   A + 1      10      20     76K
machine 2               63       5      63   A + 1      11      22     76K
where 1 <= A <= 8.
```

2. You should add the number of application client used per processor to each `MSGMNI` value. `MSGMAP` should be twice `MSGMNI`.
3. Compare the minimum IPC requirements to the parameters set for your machine. The location of these parameter settings is platform-dependent:
   - On many UNIX system platforms, machine parameters are defined in `/etc/conf/cf.d/mtune`.
   - On Windows 2003 platforms, machine parameters are set and displayed through a control panel.

See Also

- "Running STOCKAPP" on page 5-19

Step 2: How to Boot STOCKAPP

1. Set the environment:

   ```
   ../STKVAR
   ```

2. Boot the application by entering the following:

   ```
   tmboot
   ```

   The following prompt is displayed:

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

   When you enter `y` after the prompt, a running report, such as the following, is displayed on the screen:

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

The report continues until all servers in the configuration have been started. It ends by reporting the total number of servers started.

If you prefer, you can boot only a portion of the configuration. For example, to boot only administrative servers, include the `-A` option. If no options are specified, the entire application is booted.

In addition to reporting on the number of servers booted, `tmboot` also sends messages to the `ULOG`.

See Also

- "Running STOCKAPP" on page 5-19
- `tmboot(1)` in the *BEA Tuxedo Command Reference*
- `USERLOG(3cbl)` in the *BEA Tuxedo ATMI COBOL Function Reference*

Step 3: How to Test STOCKAPP Services

1. If you are logging in cold to a running system, you must set your environment for `STOCKAPP`. To do so, enter the following command:

   ```
   ../STKVAR
   ```

2. Run the `BUY` client program. To execute the `BUY` client program, enter the following command:

   ```
   BUY
   ```

3. Monitor `STOCKAPP`. While `STOCKAPP` is running, run the `tmadmin` subcommands and try various commands with it to see the kind of status information you can produce.

See Also

- "Running STOCKAPP" on page 5-19

Step 4: How to Shut Down STOCKAPP

To bring down `STOCKAPP`, enter the `tmshutdown(1)` command with no arguments, from the `MASTER` machine, as follows.

```
tmshutdown
```

Running this command (or the shutdown command of `tmadmin`) causes the following results:

- All application servers, gateway servers, TMS servers, and administrative servers are shut down.
- All associated IPC resources are removed.

See Also

- "Running STOCKAPP" on page 5-19
- `tmadmin(1)` in the *BEA Tuxedo Command Reference*
- `tmshutdown(1)` in the *BEA Tuxedo Command Reference*

CHAPTER 6

Tutorial for XMLSTOCKAPP: a C and C++ XML Parser Application

This topic includes the following sections:

What Is XMLSTOCKAPP?

`XMLSTOCKAPP` is a sample ATMI stock application that is provided with the BEA Tuxedo system software. The application runs two servers on a single machine and illustrates invoking the parser from a C and a C++ Tuxedo server and routing of XML buffers. One server is a Tuxedo server written in C++ (`stockxml`) and the other server is written in C (`stockxml_c`). The two servers

offer the same STOCKQUOTE service. The client calls the service and returns the stock price and the client then prints the XML buffer.

This documentation leads you, step-by-step, through the procedure you must perform to develop the XMLSTOCKAPP application. Once you have "developed" XMLSTOCKAPP through this tutorial, you will be ready to start developing applications of your own.

The XMLSTOCKAPP tutorial is presented in three sections:

- "Familiarizing Yourself with XMLSTOCKAPP" on page 6-2
- "Preparing XMLSTOCKAPP Files and Resources" on page 6-4
- "Running XMLSTOCKAPP" on page 6-10

Note: This information is focused on system users with some experience in application development, administration, or programming. We assume some familiarity with the BEA Tuxedo system software. A development license is required to build BEA Tuxedo applications.

Familiarizing Yourself with XMLSTOCKAPP

This documentation provides a tour of the files, client, and services that make up the XMLSTOCKAPP application. The following activities for more information about that part of the tour.

- Learning About the XMLSTOCKAPP Files
- Examining the XMLSTOCKAPP Clients
- Examining the XMLSTOCKAPP Servers

Learning About the XMLSTOCKAPP Files

The files that make up the XMLSTOCKAPP application are delivered in the samples/atmi/xmlstockapp directory. The files that are delivered with this sample are:

The XMLSTOCKAPP directory contains the following files:

- Two .xml input files to the client: stock_quote_beas.xml and stock_quote_msft.xml
- One client: Client.cpp
- Two files that are servers: stockxml and stockxml_c

- Files provided to facilitate the use of STOCKAPP as an example:
  - SAXPrint.cpp
  - SAXPrintHandler.cpp
  - DOMTreeErrorReporter.cpp
  - xmlWrapper.cpp

Examining the XMLSTOCKAPP Clients

In the ATMI 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.

The XMLSTOCKAPP implements the request/response mode and uses the STOCKQUOTE service to request a stock price.

1. A request for a stock price for BEAS or MSFT.
2. The client, which is run with a single argument in an XML file, calls the STOCKQUOTE service.
3. The service updates the XML buffer with the stock price.
4. The client prints the XML buffer.

A Request/Response Client: stock_quote_beas.xml

Client.cpp is a client program that uses input from one of the XML files, stock_quote_beas.xml or stock_quote_msft.xml. It makes an inquiry that calls on the service STOCKQUOTE and returns the stock price for BEAS or MSFT. As an executable, it is invoked as follows:

```
Client stock_quote_beas.xml
```

or

```
Client stock_quote_msft.xml
```

See Also

- "What You Can Do Using the ATMI" in *Introducing BEA Tuxedo ATMI*
- "What Are Typed Buffers?" in *Introducing BEA Tuxedo ATMI*
- ATMI commands and functions in *BEA Tuxedo Command Reference* and *BEA Tuxedo ATMI C Function Reference*

Examining the XMLSTOCKAPP Servers

ATMI 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. 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).

The `STOCKQUOTE` service in the `XMLSTOCKAPP` program uses functions provided in the Application-to-Transaction Monitor Interface (ATMI) to return a stock price to the client as an XML buffer.

Preparing XMLSTOCKAPP Files and Resources

This documentation leads you through the procedures you must complete to create the files and other resources you need to run `XMLSTOCKAPP`.

- Step1: Copy the XMLSTOCKAPP Files to a New Directory
- Step 2: Set Environment Variables
- Step 3: Building Clients
- Step 4: Building Servers in XMLSTOCKAPP
- Step 5: How to Edit the Configuration File
- Step 6: Creating a Binary Configuration File

Step1: Copy the XMLSTOCKAPP Files to a New Directory

It is recommended that you copy the XMLSTOCKAPP files to your own directory prior to editing any of the files or running the sample.

Step 2: Set Environment Variables

You will need to edit the environment variables file.

1. Ensure that TUXDIR is set. If it is not set, execution of the file fails with the following message:

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

2. Set TUXDIR to the root directory of your BEA Tuxedo system directory structure, and export it.

3. Set APPDIR to the directory {TUXDIR}/samples/atmi/XMLSTOCKAPP which is the directory where XMLSTOCKAPP source files are located: APPDIR is a directory where the BEA Tuxedo system looks for your application-specific files. If you copied the XMLSTOCKAPP files to a different directory to safeguard the original source files, then enter the directory there. It does not have to be under TUXDIR.

4. When you have made all necessary changes to the environment variables file, execute it as follows:

   ```
   . ./<VARFILE>
   ```

   where <VARFILE> is the name of your environment variable file.

Additional Requirements

LD_LIBRARY_PATH must include $TUXDIR/lib on systems that use shared libraries, with the exception of HP-UX and AIX.

- On AIX, set LIBPATH instead of LD_LIBRARY_PATH.
- On HP-UX, set SHLIB_PATH instead of LD_LIBRARY_PATH.
- If your operating system is Sun Solaris, you need to: put /usr/5bin at the beginning of your PATH. The following command can be used:

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

  Use /bin/sh rather than csh for your shell.

Step 3: Building Clients

To build the client:

```
export CFLAGS=-I
```

Use the following commands for the specified opearting system:

- For Solaris:

  ```
  export CC=CC
  ```

- For HP-UX:

  ```
  export CC=aCC
  ```

- For Digital Unix:

  ```
  export CC=cxx
  ```

- For AIX:

  ```
  export CC=xlC_r
  ```

- For Linux:

  ```
  export CC=g++
  ```

The following command builds the client:

```
buildclient -o Client -f Client.cpp -f SAXPrint.cpp -f SAXPrintHandlers.cpp
-f -ltxml
```

Step 4: Building Servers in XMLSTOCKAPP

In the XMLSTOCKAPP sample, two servers are provided for you. However, if you want to build the servers for this example, you will need to follow the directions inn the README file.

`buildserver` is used to put together an executable ATMI 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.

The key `buildserver` command-line options are illustrated in the examples that follow.

The `buildserver` command is used in a `.mk` file to compile and build each server in the stock application. (Refer to `buildserver(1)` in the *BEA Tuxedo Command Reference* for complete details.)

How to Build the stockxml and stockxml_c Servers

The `buildserver` command that was used to build the `stockxml` server and the `stockxml_c` server follows:

```
buildserver -s STOCKQUOTE -o stockxml -f stockxml.cpp -f
DOMTreeErrorReporter.cpp -f -ltxml
buildserver -s STOCKQUOTE -f stockxml_c.c -o stockxml_c -f xmlWrapper.cpp -f
DOMTreeErrorReporter.cpp -f -ltxml
```

The explanation of the command-line options follows:

– 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 `-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 `XMLSTOCKAPP`, 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)` in the *BEA Tuxedo Command Reference* 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, refer to *Administering a BEA Tuxedo Application at Run Time* and *Setting Up a BEA Tuxedo Application*.

– The `-f` option specifies the files that are used in the link-edit phase. Also refer to the `-l` option on the `buildserver` reference page. 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. Object files can be either separate `.o` files or groups of files in archive (`.a`) files. If more than a single filename 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 `STOCKQUOTE` service to be the services that comprise the `stockxml` and `stockxml_c` servers. The `-o` option assigns the name `stockxml` and `stockxml_c` to the executable output file and the `-f` option specifies that the `stockxml.cpp`, `DOMTreeErrorReporter.cpp`, and the `xmlWrapper.cpp` files are to be used in the link edit phase of the build.

See Also

- “Familiarizing Yourself with XMLSTOCKAPP” on page 6-2
- `buildserver(1)`

Step 5: How to Edit the Configuration File

The sample configuration file, ubbsimple, must be edited to replace the bracketed items with values appropriate to your installation. Your TUXDIR and TUXCONFIG environment variables must match the values in the configuration file.

Listing 6-1 The ubbsimple Configuration File

```
1$
2
3 #Skeleton UBBCONFIG file for the BEA Tuxedo Simple Application.
4 #Replace the <bracketed> items with the appropriate values.
5 RESOURCES
6 IPCKEY            <Replace with valid IPC Key greater than 32,768>
7
8  #Example:
9
10 #IPCKEY         62345
11
12 MASTER          simple
13 MAXACCESSERS    5
14 MAXSERVERS      5
15 MAXSERVICES     10
16 MODEL           SHM
17 LDBAL           N
18
19 *MACHINES
20
21 DEFAULT:
22
23                          APPDIR="<Replace with the current pathname>"
24              TUXCONFIG="<Replace with TUXCONFIG Pathname>"
25         TUXDIR="<Root directory of Tuxedo (not /)>"
26 #Example:
27 #       APPDIR="/usr/me/simpdir"
28 #       TUXCONFIG="/usr/me/simpdir/tuxconfig"
29 #       TUXDIR="/usr/tuxedo"
30
31 <Machine-name>  LMID=simple
32 #Example:
33 #tuxmach        LMID=simple
34 *GROUPS
35 GROUP1
36              LMID=simple   GRPNO=1  OPENINFO=NONE
37
38 *SERVERS
```

```
39  DEFAULT:
40              CLOPT="-A"
41  stockxml    SRVGRP=GROUP1 SRVID=1
42  stockxml_c  SRVGRP=GROUP1 SRVID=1
43 *SERVICES
44  STOCKQUOTE
```

5. For each <string> (that is, for each string shown between angle brackets), substitute an appropriate value.

See Also

- "Preparing XMLSTOCKAPP Files and Resources" on page 6-4
- UBBCONFIG(5) in *File Formats, Data Descriptions, MIBs, and System Processes Reference*

Step 6: Creating a Binary Configuration File

Before creating the binary configuration file, you need to be in the directory in which your XMLSTOCKAPP files are located and you must set the environment variables. Complete the following tasks.

1. Go to the directory in which your XMLSTOCKAPP files are located.
2. Set the environment variables by entering:

   ```
   . ./<variable_file>
   ```

   where <variable_file> is the name of your variables file.

How to Load the Configuration File

Once you have finished editing the configuration file, you must load it into a binary file on your MASTER machine. The name of the binary configuration file is TUXCONFIG; its path name is defined in the TUXCONFIG environment variable. 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 this requirement is not met, you may have permission problems in running XMLSTOCKAPP.

1. To create TUXCONFIG, enter the following command:

   ```
   tmloadcf ubbsimple
   ```

While the configuration file is being loaded, you are prompted several times to confirm that you want to install this configuration, even if doing so means an existing configuration file must be overwritten. If you want to suppress such prompts, include the `-y` option on the command line.

2. If you want the amount of IPC resources needed by your application to be calculated by the BEA Tuxedo system, include the `-c` option on the command line.

   `TUXCONFIG` can be installed only on the `MASTER` machine; it is propagated to other machines by `tmboot` when the application is booted.

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

See Also

- "Preparing XMLSTOCKAPP Files and Resources" on page 6-4
- `tmloadcf(1)` in *BEA Tuxedo Command Reference*

Running XMLSTOCKAPP

This documentation leads you through the procedures for booting `XMLSTOCKAPP`, testing it by running various client programs and transactions, and shutting it down when you have finished.

Step 1: How to Prepare to Boot

Before booting `XMLSTOCKAPP`, verify that your machine has enough IPC resources to support your application. To generate a report on IPC resources, run the `tmboot` command with the `-c` option.

Step 2: How to Boot XMLSTOCKAPP

1. Set the environment:

   ```
   ../<variable_file>
   ```

2. Boot the application by entering the following:

   ```
   tmboot -y
   ```

   When you enter -y, a running report, such as the following, is displayed on the screen:

   ```
   Booting all admin and server processes in /usr/me/appdir/tuxconfig
   Booting all admin processes
   ```

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

The report continues until all servers in the configuration have been started. It ends by reporting the total number of servers started.

If you prefer, you can boot only a portion of the configuration. For example, to boot only administrative servers, include the `-A` option. If no options are specified, the entire application is booted.

In addition to reporting on the number of servers booted, `tmboot` also sends messages to the `ULOG`.

See Also

- `tmboot(1)` in the *BEA Tuxedo Command Reference*
- `USERLOG(3cbl)` in the *BEA Tuxedo ATMI COBOL Function Reference*

Step 3: How to Test XMLSTOCKAPP Services

1. If you are logging in cold to a running system, you must set your environment for `XMLSTOCKAPP`. To do so, enter the following command:

   ```
   ../<variable_file>
   ```

2. Run the client program. To execute the client program, enter the following command:

   ```
   Client stock_quote_beas.xml
   ```

Step 4: How to Shut Down XMLSTOCKAPP

To bring down `XMLSTOCKAPP`, enter the `tmshutdown(1)` command with no arguments, from the `MASTER` machine, as follows.

```
tmshutdown -y
```

Running this command (or the shutdown command of `tmadmin`) causes the following results:

- All application servers, gateway servers, TMS servers, and administrative servers are shut down.
- All associated IPC resources are removed.

See Also

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

CHAPTER 7

Tutorial for xmlfmlapp: A Full C XML/FML32 Conversion Application

This topic includes the following sections:

What Is xmlfmlapp?

`xmlfmlapp` is a sample ATMI stock application that demonstrates how to query, buy and sell stocks via client request. The application runs three services, "`QUERY`", "`BUY`" and "`SELL`" on a single server. These three services are written using C language and accept FML32 buffers for input and output.

This documentation leads you, step-by-step, through the procedures you must perform to develop the `xmlfmlapp` application. Once you have "developed" `xmlfmlapp` with this tutorial, you will be ready to start developing applications of your own.

The `xmlfmlapp` tutorial is presented in three sections:

- "Familiarizing Yourself with xmlfmlapp" on page 7-2
- "Preparing xmlfmlapp Files and Resources" on page 7-5
- "Running xmlfmlapp" on page 7-9

Note: This information is geared towards system users with some experience in application development, administration, or programming. We assume some familiarity with the BEA Tuxedo system software. A development license is required to build BEA Tuxedo applications.

Familiarizing Yourself with xmlfmlapp

This sample demonstrates how to use XML to FML32 automatic and on-demand conversion functions to operate XML data instead of using Xerces parser APIs. To use Xerces parser APIs in a Tuxedo client/server application written in C, a dynamic library needs to be written using CPP and wrapped for use with a C program (for more information, see What Is XMLSTOCKAPP?). Using XML to FML32 on-demand and automatic conversion functionality, provides the developer with the freedom to manipulate FML32 buffer data as desired. For more information on the XML to FML/FML32 on-demand and automatic conversion functionality, see Converting XML Data To and From FML/FML32 Buffers in *Programming BEA Tuxedo ATMI Applications Using C*.

In this sample, the client will send requests (query, buy or sell) to corresponding services. The client sends and receives XML buffers. To communicate with server, all three services use the "BUFTYPECONV=XML2FML32" parameter, which converts the input XML buffers to FML32 before sending the request to the corresponding service. Before returning information back to the client, this parameter then converts FML32 buffers to XML buffers. The server handles FML32 data directly.

The requesting XML buffer uses a schema to validate the XML document, thus ensuring that the request sends valid data.

The server in this sample reads stock information from an XML document and converts it to an FML32 buffer directly using the `tpxmltofml32(3c)` function. After that, it can get information from this buffer based on requested FML32 data, and returns the required FML32 data.

This documentation provides a tour of the files, client, and services that make up the `xmlfmlapp` application. The following activities for more information about that part of the tour.

- Learning About the xmlfmlapp Files
- TExamining the xmlfmlapp Client
- Examining the xmlfmlapp Server

Learning About the xmlfmlapp Files

The files that make up the `xmlfmlapp` application are delivered in the `samples/atmi/xmlfmlapp` directory. The `xmlfmlapp` directory contains the following files:

- One client: stock`client.c`
- One server: `stockserver.c`
- One. xml file to store stock information used by the service: `stock.xml`
- One FML32 field definition file: `stockflds`
- Four .xml client input files:
  - `stock_query_bea.xml`
  - `stock_query_msft.xml`
  - `stock_buy_bea.xml`
  - `stock_sell_msft.xml`
- One .xml schema file to validate the XML input files: stock_operate.xsd

TExamining the xmlfmlapp Client

In the ATMI 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.

The `xmlfmlapp` implements the request/response mode using the following three services:

- `QUERY` - to query a stock price
- `BUY` - to buy stock
- `SELL` - to sell stock.

Request/Response Client

`stockclient.c` is a client program that uses input from the specified XML files. It calls the `QUERY`, `BUY`, and `SELL` services and returns the executed results. As an executable, it is invoked as follows:

- `stockclient stock_query_bea.xml`
- `stockclient stock_buy_bea.xml`
- `stockclient stock_query_msft.xml`
- `stockclient stock_sell_msft.xml`

See Also

- "What You Can Do Using the ATMI" in *Introducing BEA Tuxedo ATMI*
- "What Are Typed Buffers?" in *Introducing BEA Tuxedo ATMI*
- ATMI commands and functions in *BEA Tuxedo Command Reference* and *BEA Tuxedo ATMI C Function Reference*

Examining the xmlfmlapp Server

ATMI 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. 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).

The QUERY service in the xmlfmlapp program accepts FML32 buffers. It uses functions provided in the Application-to-Transaction Monitor Interface (ATMI) to query stock information and then returns the results to the client using FML32 buffers.

The BUY service in the xmlfmlapp program accepts FML32 buffers. It uses functions provided in the Application-to-Transaction Monitor Interface (ATMI) to buy stock.

The SELL service in the xmlfmlapp program accepts FML32 buffers. It uses functions provided in the Application-to-Transaction Monitor Interface (ATMI) to sell stock.

Preparing xmlfmlapp Files and Resources

This section leads you through the procedures you must complete to create the files and other resources you need to run xmlfmlapp.

Step 1: Copy the xmlfmlapp Files to a New Directory

It is recommended that you copy the xmlfmlapp files to your own directory prior to editing any of the files or running the sample.

Step 2: Set Environment Variables

You will need to edit the environment variables file.

1. Ensure that TUXDIR is set. If it is not set, execution of the file fails with the following message:

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

2. Set TUXDIR to the root directory of your BEA Tuxedo system directory structure, and export it.

3. Set APPDIR to the directory {TUXDIR}/samples/atmi/xmlfmlapp which is the directory where xmlfmlapp source files are located. APPDIR is a directory where the BEA Tuxedo system looks for your application-specific files. If you copied the xmlfmlapp files to a different directory to safeguard the original source files, then enter the directory there. It does not have to be under TUXDIR.

4. When you have made all necessary changes to the environment variables file, execute it as follows:

   ```
   . ./setenv.cmd
   ```

   where setenv.cmd is the executable for Windows. Use setenv.sh on Unix systems.

Additional Requirements

LD_LIBRARY_PATH must include $TUXDIR/lib on systems that use shared libraries, with the exception of HP-UX and AIX.

- On AIX, set LIBPATH instead of LD_LIBRARY_PATH.
- On HP-UX, set SHLIB_PATH instead of LD_LIBRARY_PATH.
- If your operating system is Sun Solaris, you need to: put /usr/5bin at the beginning of your PATH. The following command can be used:

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

  Use /bin/sh rather than csh for your shell.

Step 3: Create FML32 Field Table

To create the FML32 field table, use the following:

```
mkfldhdr32 stockflds
```

Step 4: Build the xmlfmlapp Binaries

The following command builds the xmlfmlapp binary files:

On windows:

```
nmake -f make.nt
```

On UNIX:

```
make -f make.mk
```

Step 5: Edit the Configuration File

The sample configuration file, ubbsimple, must be edited to replace the bracketed items with values appropriate to your installation. Your TUXDIR and TUXCONFIG environment variables must match the values in the configuration file.

Listing 7-1 The ubbsimple Configuration File

```
#(c) 2005 BEA Systems, Inc. All Rights Reserved.
#ident"@(#) samples/atmi/xmlfmlapp/ubbsimple$Revision: 1.3 $"

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

*RESOURCES
IPCKEY<Replace with a valid IPC Key>

#Example:
#IPCKEY123456

DOMAINIDsimpapp
MASTERsimple
MAXACCESSERS10
MAXSERVERS5
MAXSERVICES10
MODELSHM
LDBALN

*MACHINES
DEFAULT:
APPDIR="<Replace with the current directory pathname>"
TUXCONFIG="<Replace with your TUXCONFIG Pathname>"
TUXDIR="<Directory where TUXEDO is installed>"
#Example:
#APPDIR="/home/me/simpapp"
#TUXCONFIG="/home/me/simpapp/tuxconfig"
#TUXDIR="/usr/tuxedo"

<Machine-name>LMID=simple
#Example:
#beatuxLMID=simple

*GROUPS
```

```
GROUP1
LMID=simpleGRPNO=1OPENINFO=NONE

*SERVERS
DEFAULT:
CLOPT="-A"

stockserverSRVGRP=GROUP1 SRVID=1

*SERVICES
QUERYBUFTYPECONV=XML2FML32
BUYBUFTYPECONV=XML2FML32
SELLBUFTYPECONV=XML2FML32
```

Note: For each `<string>` (that is, for each string shown between angle brackets), substitute an appropriate value.

See Also

- "Preparing xmlfmlapp Files and Resources" on page 7-5
- `UBBCONFIG(5)` in *File Formats, Data Descriptions, MIBs, and System Processes Reference*

Step 6: Create the Binary Configuration File

Before creating the binary configuration file, you need to be in the directory in which your `xmlfmlapp` files are located and you must set the environment variables. Complete the following tasks.

1. Go to the directory in which your `xmlfmlapp` files are located.
2. Set the environment variables by entering:

   ```
   . ./setenv.cmd
   ```

   where `setenv.cmd` is the executable for Windows. Use `setenv.sh` on Unix systems.

Loading the Configuration File

Once you have finished editing the configuration file, you must load it into a binary file on your `MASTER` machine. The name of the binary configuration file is `TUXCONFIG`; its path name is defined in the `TUXCONFIG` environment variable. 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 this requirement is not met, you may have permission problems in running xmlfmlapp.

1. To create TUXCONFIG, enter the following command:

   ```
   tmloadcf ubbsimple
   ```

   While the configuration file is being loaded, you are prompted several times to confirm that you want to install this configuration, even if doing so means an existing configuration file must be overwritten. If you want to suppress such prompts, include the -y option on the command line.

2. If you want the amount of IPC resources needed by your application to be calculated by the BEA Tuxedo system, include the -c option on the command line.

   TUXCONFIG can be installed only on the MASTER machine; it is propagated to other machines by tmboot when the application is booted.

   tmloadcf parses the text configuration file (UBBCONFIG) for syntax errors before it loads it, so if there are errors in the file, the job fails.

See Also

- "Preparing xmlfmlapp Files and Resources" on page 7-5
- tmloadcf(1) in *BEA Tuxedo Command Reference*

Running xmlfmlapp

This section leads you through the procedures for booting xmlfmlapp, testing it by running the client program with several arguments, and shutting it down when you have finished.

Step 1: xmlfmlapp Boot Preparation

Before booting xmlfmlapp, verify that your machine has enough IPC resources to support your application. To generate a report on IPC resources, run the tmboot command with the -c option.

Step 2: Boot xmlfmlapp

1. Set the environment:

   ```
   ../setenv.cmd
   ```

2. Boot the application by entering the following:

   ```
   tmboot -y
   ```

If you prefer, you can boot only a portion of the configuration. For example, to boot only administrative servers, include the `-A` option. If no options are specified, the entire application is booted.

See Also

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

Step 3: Test xmlfmlapp Services

1. Each time you log-in to the system, you must set your environment for `xmlfmlapp`. To do so, enter the following command:

   ```
   ../setenv.cmd
   ```

2. Run the client program. To execute the client program, enter the following command:

   ```
   stockclient stock_query_bea.xml
   stockclient stock_query_msft.xml
   stockclient stock_buy_bea.xml
   stockclient stock_sell_msft.xml
   ```

Step 4: Shut Down xmlfmlapp

To bring down `xmlfmlapp`, enter the `tmshutdown(1)` command with no arguments, from the `MASTER` machine, as follows.

```
tmshutdown -y
```

Running this command (or the shutdown command of `tmadmin`) causes the following results:

- All application servers, gateway servers, TMS servers, and administrative servers are shut down.
- All associated IPC resources are removed.

See Also

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