BEA Tuxedo

Programming a BEA Tuxedo Application Using TxRPC

BEA Tuxedo Release 8.0
Document Edition 8.0
June 2001

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Programming a BEA Tuxedo Application Using TxRPC

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

Contents

5. Running the Application

A. A Sample Application

B. A DCE-Gateway Application

About This Document

This document explains how to program and use the TxRPC feature to implement remote procedure calls (RPC) in the BEA Tuxedo environment.

This document covers the following topics:

- Chapter 1, “Introducing TxRPC,” provides an introduction to the TxRPC feature.
- Chapter 2, “Using the Interface Definition Language (IDL),” provides guidelines for using Interface Definition Language (IDL) to develop TxRPC applications.
- Chapter 3, “Writing RPC Client and Server Programs,” provides instructions on developing programs that provide procedure calls transparently between a client in one address space and a server in another address space.
- Chapter 4, “Building RPC Client and Server Programs,” provides instructions on building RPC client and server programs, using C++, and ensuring interoperation with DCE/RPC.
- Chapter 5, “Running the Application,” provides instructions on configuring, booting, and shutting down a TxRPC application. This section also describes how to dynamically advertise services.
- Appendix A, “A Sample Application,” provides a sample client-server application using TxRPC.
- Appendix B, “A DCE-Gateway Application,” provides a sample client-server application using an OSF/DCE server and a gateway so that the BEA Tuxedo ATMI client can communicate with the server using explicit binding and authenticated RPCs.

What You Need to Know

This document is intended for the following audiences:

- administrators who are interested in configuring and managing TxRPC applications in a BEA Tuxedo environment
- application developers who are interested in programming TxRPC applications in a BEA Tuxedo environment

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

e-docs Web Site

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

How to Print the Document

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

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

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

Related Information

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

- *BEA Tuxedo Command Reference*
- *BEA Tuxedo ATMI C Function Reference*

For information on external documentation relating to the Interface Definition Language (IDL), see References in Chapter 2, "Using the Interface Definition Language (IDL)."

Contact Us!

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

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

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

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

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes

- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

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

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

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*

Convention	Item
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Examples*: LPT1 SIGNON OR
`{ }`	Indicates a set of choices in a syntax line. The braces themselves should never be typed.
`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed. *Example*: `buildobjclient [-v] [-o name ] [-f file-list]... [-l file-list]...`
`|`	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.
`...`	Indicates one of the following in a command line: ■ That an argument can be repeated several times in a command line ■ That the statement omits additional optional arguments ■ That you can enter additional parameters, values, or other information The ellipsis itself should never be typed. *Example*: `buildobjclient [-v] [-o name ] [-f file-list]... [-l file-list]...`
. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

1 Introducing TxRPC

This topic includes the following section:

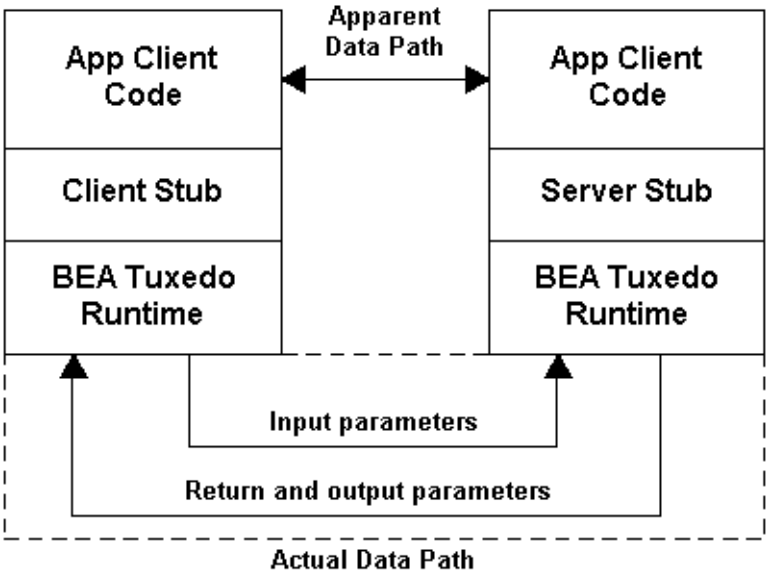
- What Is TxRPC?

What Is TxRPC?

The TxRPC feature allows programmers to use a remote procedure call (RPC) interface, such that a client process can call a remote function (that is, a remote service) in another process using a local function call. The application writer must specify the operations (that is, procedures) and data types that are used as parameters to those operations via an Interface Definition Language (IDL). *Operations* are grouped together in an *interface*. An IDL compiler is used to generate substitute procedures called *stubs* which allow the operation to be remote. An important concept to understand from the beginning is that there are two fundamental levels of naming: the interface has a name and within an interface, one or more operations are named. At run time, the interface is made available, which means that any of the operations in the interface can be called; an individual operation within an interface cannot be made available (if you need this, define the operation in its own interface).

The following illustrates how an RPC is made to look like a local procedure call.

Figure 1-1 RPC Communication



The client application code calls one of the operations (functions) defined in the IDL file. Instead of calling the actual function, which resides on the server side, the client stub is called. The client stub is generated by the IDL compiler based on the IDL input file, which defines the data types and operations. For each operation, the input parameters, return type, and output parameters are defined. The client stub takes the input parameters and converts them into a single buffer of data, sends the data to the server and waits for a response, and unpacks the buffer of data sent back from the server (the return value and output parameters). The communication between the client and server processes, whether intra-machine or inter-machine is handled by the BEA Tuxedo ATMI run time.

On the server side, the run time calls the server stub for the interface, also generated by the IDL compiler. This stub unpacks the data buffer that contains the input parameters, in some cases it allocates space needed for output parameters of the operation, calls the operation and waits for it to return, packs the return value and output parameters into a buffer and sends the response back to the client.

From the application perspective, it appears that a simple local procedure call is done. The stubs and the run time hide the calling of a remote procedure in a non-local address space (process).

The steps for building an application using remote procedure calls is very similar to building one without these calls. Most of the time will be spent writing the application code for the client and the server (where the real application work is done). The BEA Tuxedo ATMI run time frees the application programmer from worrying about communications, translation of the data from the format used on the client machine to the format used on the server machine, and so forth. TxRPC may also be used to communicate between servers.

In addition to the steps needed for building a monolithic application, it is necessary to completely define the interface between the client and server. As stated earlier, the interface contains the definition of data types and operations used for the remote procedure calls. Normally, the name of the file containing the definition has an "`.idl`" suffix; using this convention makes the file type self-documenting.

Every interface must have its own unique identifier. This Universal Unique Identifier (UUID) consists of 128-bits that uniquely identify the interface among all interfaces. The job of generating a UUID is done for the application programmer by the `uuidgen` program. By running the `uuidgen` program with the `-i` option, it generates an interface template that contains a new UUID. Refer to Appendix A, "A Sample Application," for a complete example (including code) for the development of a simple RPC application; the first step illustrates how to run the `uuidgen` command and the resulting output. More information about other options of this command are given in the `uuidgen(1)` manual page.

The UUID is used at run time to ensure that the client stub matches the server stub on the receiving side. That is, the UUID is sent from the client to the server for validation by the BEA Tuxedo ATMI run time, transparent to the application programmer.

Besides matching on the UUID, each interface also has a version number associated with it. The version consists of a major and minor number. If a version number is not specified as part of the interface definition, it defaults to 0.0. Thus, there may be multiple versions of the same interface available. The client requests a particular version of an interface by invoking the RPC in the stub generated from a particular interface version. Different versions imply that data types or operation parameters or returns have changed, or operations have been added to or deleted from the interface. Thus, the client and server UUID's and versions must match for a successful RPC. The application programmer must ensure that versions of the interface that have the same version numbers do provide the same (or a compatible) interface.

Once the template IDL is generated by `uuidgen`, the application program must provide a definition of all data types and operations in the interface. The language looks very much like the declarative parts of C or C++ (without the procedural statements). Data

types are declared via `typedef` statements, and the operations are declared via function prototypes. Additional information is provided via IDL *attributes*. Attributes appear in the language within square brackets, for example, [in]. These provide information about such things as pointer types (for example, whether or not a pointer can be NULL at run time), about parameters (for example, whether a parameter is for input, output, or both), and much more. The IDL language and the associated compiler are discussed further in "Using the Interface Definition Language (IDL)" on page 2-1.

In addition to the IDL file, an optional Attribute Configuration File (ACF) may also be provided to give additional attributes of the interface. Most important is the definition of status variables in the operations for returning the status of each RPC operation. The use of status variables will be discussed further in "Writing RPC Client and Server Programs" on page 3-1. Attributes in the ACF file do not affect the communications between the client and server (as do attributes in the IDL file), but generally have an impact on the interface between the application code and the generated stubs.

When using the BEA Tuxedo ATMI run time, the management of the binding (connection) between the client and server is done transparently. There is no information provided by the client or server application code to manage the client/server binding. (In contrast, when using the OSF DCE run time, considerable effort by the programmer must be given to binding management. BEA Tuxedo ATMI runtime does not support the OSF DCE run time functions and ignores binding attributes in IDL and ACF files.)

The IDL and optional ACF files are compiled using the IDL compiler. The compiler first generates a header file that contains all of the type definitions and function prototypes for the operations defined in the IDL file. This header file can be included in application code that makes RPC calls defined in the interface. If the input files are *`file`*`.idl` and *`file`*`.acf`, then the default header file name is *`file`*`.h`. The compiler generates stub code for both the client and server (for example, *`file`*`_cstub.c` and *`file`*`_sstub.c`). These stub files were described earlier and contain the data packaging and communications for the RPC. By default, the IDL compiler invokes the C compiler to generate client and server stub object files (for example, *`file`*`_cstub.o` and *`file`*`_sstub.o`) and the stub source files are removed. There are various IDL compiler options to request, limit generation of, and keep source and object files, and change the output filenames and directories. See the `tidl(1)` reference page for further details.

After completing the interface definition, the major portion of work is writing the application code. The client code will call the operations defined in the interface, and the server code must implement the operations (note that a server can also act as a client by calling an RPC). Further considerations regarding writing the application are discussed in "Using the Interface Definition Language (IDL)" on page 2-1.

When the application code is completed, it's time to compile and link it together with the BEA Tuxedo ATMI run time. Two programs are provided to simplify this process: `buildserver` for the server, and `buildclient` for the client. These programs compile any source files and link the object and library files with the BEA Tuxedo ATMI run time to produce the executable files. These programs allow for alternate compilers and compilation options to be specified. See the `buildserver`(1) and `buildclient(1)` reference pages for further details.

The complete process for building a server and client are shown in Figure 1-2 and Figure 1-3. More details about building client and server programs on different platforms are provided in "Building RPC Client and Server Programs" on page 4-1.

Figure 1-2 Building an RPC Server

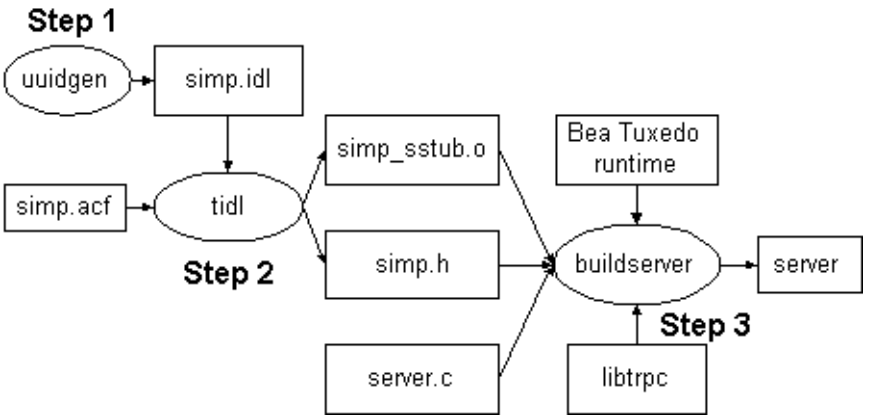
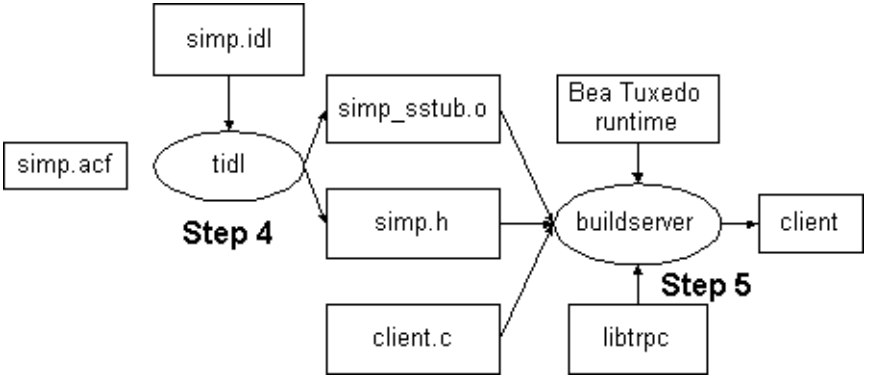


Figure 1-2 illustrates the following steps in the process for building a server:

1. Run `uuidgen` to generate a skeleton IDL file (`simp.idl`) with a `UUID`. Edit the template IDL file to define the interface between the client and server using the interface definition language.
2. Run the IDL compiler (`tidl`) using `simp.idl` and optional `simp.acf` to generate the interface header file and the server stub object file.

3. After writing the server application code (`server.c`), run `buildserver` to compile it and link it with the server stub, BEA Tuxedo ATMI run time, and TxRPC run time to generate an executable server.

Figure 1-3 Building an RPC Client



The preceding figure illustrates the process for building a client.

4. Using the IDL file created in Step 1, run the IDL compiler (`tidl`) to generate the interface header file and the client stub object file.

5. After writing the client application code (`client.c`), run `buildclient` to compile it and link it with the client stub, BEA Tuxedo ATMI run time, and TxRPC run time to generate an executable client.

After building the application client and server, the application can be configured and booted, and the client run. This is discussed in “Running the Application” on page 5-1.

2 Using the Interface Definition Language (IDL)

This topic includes the following sections:

- Using uuidgen to Create an IDL Template
- Changes in the Language
- Changes Based on the TxRPC Specification
- Enhancements to the Language
- Unsupported Features
- Using tidl, the IDL Compiler

References

BEA Tuxedo TxRPC supports the IDL grammar and associated functionality as described in Chapter 3 (“Interface Definition Language”) of *DCE: REMOTE PROCEDURE CALL* (Doc Code: P312 ISBN 1-872630-95-2). This book is available from the following.

X/OPEN Company Ltd (Publications)
P O Box 109
Penn
High Wycombe
Bucks HP10 8NP
United Kingdom

Tel: +44 (0) 494 813844
Fax: +44 (0) 494 814989

The X/OPEN document is the ultimate authority on the language and rules adhered to for the BEA Tuxedo product in an ATMI environment. Note that the X/OPEN TxRPC IDL-only interface is supported (parts of the document concerning the DCE binding and run time do not apply). The X/OPEN document is based on the OSF DCE AES/RPC document. There are several books containing tutorials and programmer's guides that can be used, although most will not contain the latest features. The programmer's guide available from OSF is *OSF DCE Application Development Guide*, published by Prentice-Hall (Englewood Cliffs, New Jersey, 07632).

The *X/OPEN Preliminary Specification for TxRPC Communication Application Programming Interface* is also available from X/OPEN (see above). TxRPC adds transaction support for RPCs to the original X/OPEN RPC interface.

Using uuidgen to Create an IDL Template

A Universal Unique Identifier (UUID) is used to uniquely identify an interface. The `uuidgen` command is used to generate UUIDs. The output might look something like the following:

```
$ uuidgen -i > simp.idl
$ cat simp.idl
[uuid(816A4780-A76B-110F-9B3F-930269220000)]
interface INTERFACE
{
}
```

This template is then used to create the IDL input file for the application (adding type definitions, constants, and operations).

If both the ATMI and DCE `uuidgen(1)` commands are available, the DCE command can and should be used to generate the template (the DCE version will most likely have a machine-specific approach to getting the node address, as described below).

The ATMI `uuidgen` command is similar to the DCE command with the exception that the `-s` option (which generates a UUID string as an initialized C structure), and the `-t` option (which translates an old style UUID string to the new format) are not supported. See the `uuidgen(1)` reference page for details of the interface.

The `uuidgen` command requires a 48-bit node address as described in ISO/IEC 8802-3 (ANSI/IEEE 802.3). There is no platform-independent way to determine this value, and it may not be available at all on some machines (a workstation, for example). The following approach is used for the ATMI `uuidgen` command:

- If the `NADDR` environment variable is set to a value of the form *`num.num.num.num.num.num`* where *`num`* is between 0 and 255, inclusive, it is taken to be an Internet-style address and converted to a 48-bit node address. This allows conformance with the use of the 8802-3 node address. It also allows users who do not have access to this address to use another value, most likely the Internet address (which is *not* the same as the 8802-3 address). If the Internet address is used, the last *`num.num`* should be 0.0 (because Internet addresses are only 32-bit addresses).

- If the `NADDR` environment variable is not set and if the `WSNADDR` environment variable is set to a value of the form *`0xnnnnnnnnnnnnnnnn`* it is taken to be a hexadecimal network address, as used in Workstation. Again note that this is *not* the 8802-3 address, and the last 16 bits will be treated as zeros.

- If neither the `NADDR` nor the `WSNADDR` environment variable is set (and if not Windows), the `uname` for the machine is used to look up the machine entry in `/etc/hosts` to get the Internet-style address.

- If the first three choices are not available, a warning is printed and 00.00.00.00.00.00 is used. This is not desirable because it reduces the chance of generating a unique UUID.

Changes in the Language

The IDL compiler recognizes the IDL grammar and generates stub functions based on the input. The grammar and its semantics are fully described in both the X/OPEN and OSF/DCE references listed earlier in this chapter. The grammar will be recognized in its entirety with some changes as described in the following sections.

Changes Based on the TxRPC Specification

The following are changes to the base X/OPEN RPC specification that are defined by the X/OPEN TxRPC specification:

- The most important enhancement from the TxRPC specification is the addition of the `[transaction_optional]` and `[transaction_mandatory]` attributes in the interface and operation attributes in the IDL file. `[transaction_optional]` indicates that if the RPC is done while in a transaction, the remote service is done as part of the transaction. The `[transaction_mandatory]` attribute requires that the RPC be done within a transaction. Without these attributes, the remote service is not part of any transaction of which the client may be part.
- Binding types and attributes are not required by X/OPEN TxRPC IDL-only. The binding attributes are `[handle]`, `[endpoint]`, `[auto_handle]`, `[implicit_handle]`, and `[explicit_handle]`. They are recognized by `tidl(1)` but not supported (these attributes are ignored). Also the `handle_t` type is not treated specially (it is transmitted as any other defined type is transmitted, without treatment as a handle).
- Pipes are not required by X/OPEN TxRPC IDL-only. `tidl` supports pipes only in `[local]` mode; that is, they can be specified for header file, but not stub, generation.
- The `[idempotent]`, `[maybe]`, and `[broadcast]` attributes are not required by X/OPEN TxRPC IDL-only. They are ignored by `tidl(1)`.

Enhancements to the Language

The following are enhancements to the X/OPEN RPC specification. In most cases, the language has been enhanced to more closely follow the C language, simplifying the porting of existing interfaces (converting from ANSI C to IDL prototypes).

- In the X/OPEN specification, character constants and character strings are limited to the portable set, that is `space` (0x20) through `tilde` (0x7e). Other characters in the character set (0x01 through 0xff) are allowed, as in OSF DCE RPC.

- As in C, the following operators are treated as punctuators.

  ```
  || && ? | & _ == != = << >> <= >= < > + - % ! ~
  ```

  This means that white space need not follow or precede identifiers or numbers if preceded or followed by one of these tokens. (The IDL specification requires white space, as in `a = b + 3`, instead of allowing `a=b+3`.) This also seems to be the behavior of the OSF DCE IDL compiler.

- The published X/OPEN specification restricts field and parameter names from matching type names. This restriction effectively puts all names in a single name space. This restriction does not match C, C++, or the OSF IDL compiler, and is not enforced.

- The X/OPEN specification does not allow anonymous enumerations as parameter or function results and does not allow anonymous structures or unions as the targets of pointers. Each of these is allowed by the OSF DCE IDL compiler. These restrictions are not enforced; in each case, a name, based on the interface name and version, is generated for use during marshalling.

- Enumeration values (constants) may be used in integer constant expressions (as in C). This also seems to be the behavior of the DCE IDL compiler.

- As currently defined in the X/OPEN RPC specification, the grammar does not allow for a pointer in front of an operation declaration, for example:

  ```
  long *op(void);
  ```

  nor does it allow for structure or union returns. While this could be considered correct (everything could be hidden in a defined type), the DCE IDL compiler and, of course, C compiler allow a much richer operation return. The supported grammar will be the following:

[*operation_attributes*] <*type_spec*> <*declarator*>

where <*declarator*> must contain a <*function_declarator*>. (If a <*function_declarator*> does not exist, then a variable is declared, which results in an error.) Declaring an array of operations or an operation returning an array (both allowed by this grammar) will be detected and flagged as an error.

- The <*ACS_type_declaration*> takes <*ACS_named_type*> values, just as the IDL <*type_declaration*> takes a list of declarators. This seems to be the behavior of the DCE IDL compiler.
- Fielded buffers created and manipulated with the Field Manipulation Language (FML) are an integral part of many BEA Tuxedo ATMI applications. Fielded buffers are supported as a new base type in the IDL. They are indicated by the keywords `FBFR` for 16-bit buffers and `FBFR32` for 32-bit buffers and must always be defined as a pointer (for example, `FBFR *` or `FBFR32 *`). A fielded buffer cannot be defined as the base type in a `typedef`. They can be used in structure fields and as parameters. They can be used as the base type in an array or pointer (either full or reference pointer). However, conformant and varying arrays of fielded buffers are not supported.
- There are several restrictions in the OSF IDL compiler that are not documented in the AES or X/OPEN RPC specification. These are enforced in the BEA Tuxedo IDL compiler:
  - A transmitted type used in `[transmit_as()]` cannot have the `[represent_as]` attribute.
  - A union arm may not be or contain a `[ref]` pointer.
  - If a conformant and/or varying array appears in a structure, the array size attribute variable may not be a pointer (that is, it must be a non-pointer, integer element within the structure).

Enhancements that May Limit Portability

There are four additional BEA Tuxedo ATMI enhancements to the X/OPEN RPC specification that, while making the specification more C-like, are not supported in the OSF DCE IDL compiler and thus have the effect of limiting portability of the IDL file:

- String concatenation is supported (as in ANSI C). That is:

```
const char *str = “abc” “def”;
```

is treated the same as

```
const char *str = “abcdef”;
```

- Escaped newlines are allowed in string constants. That is:

```
const char *str = “abc\
 def”;
```

is treated the same as

```
const char *str = “abcdef”;
```

- Enumeration values may also be used in union cases and are treated as integers (that is, automatic conversion is provided as in C).
- The restriction that the type of each <*union_case_label*> must be that specified by the <*switch_type_spec*> will not be enforced. Instead, the type will be coerced as is done with case statements in a C switch statement.

Unsupported Features

The following seven features are not supported in the `tidl` compiler:

- The migration attributes `[v1_struct]`, `[v1_enum]`, `[v1_string]`, and `[v1_array]` are recognized but not supported (these appear in the OSF IDL specification but not the X/OPEN specification).
- Function pointers (defined in the OSF/DCE document) are supported only in `[local]` mode (as in OSF/DCE).
- An exact match is required on interface version minor between the client and the server (the X/OPEN RPC specification allows for the server version minor to be greater than or equal to the version minor specified by the client).
- On machines with 32-bit longs, integer literal values are limited to -2**31 to 2**31. This means that unsigned long integer values in the range 2**31+1 to 2**32-1 are not supported. This also seems to be the behavior of the DCE IDL compiler.
- Context handles are supported only in `[local]` mode. Interfaces cannot be written that use context handles to maintain state across operations.

- The [out-of-line] ACS attribute is ignored. This feature is not defined in a way that will support interoperation between different implementations (e.g., with the OSF IDL compiler).
- The [heap] ACS attribute is ignored.

Using tidl, the IDL Compiler

The interface for the IDL compiler is not specified in any X/OPEN specification.

For DCE application portability, the BEA Tuxedo ATMI IDL compiler has a similar interface to the DCE IDL compiler, with the following exceptions:

- The command name is tidl instead of idl so an application can easily reference either when both appear in the same environment.
- The -bug option, which generates buggy behavior for interoperability with earlier versions of the software, has no effect. The -no_bug option also has no effect.
- The -space_opt option, which optimizes the code for space, is ignored. Space is always optimized.
- A new option, -use_const, is supported. -use_const generates ANSI C const statements instead of #define statements for constant definitions. This gets around an annoying problem where a constant defined in the IDL file collides with another name in the file using a C-preprocessor definition, but is properly in another name space when defined as a C constant. Use of this feature will limit portability of the IDL file.
- By default, /lib/cpp, /usr/ccs/lib/cpp, or /usr/lib/cpp (whichever is found first) is the command used to preprocess the input IDL and ACF files.

By default, the IDL compiler takes an input IDL file and generates the client and server stub object files. The -keep c_source option generates only the C source files, and the -keep all option keeps both the C source and object files. The sample RPC application, listed in Appendix A, "A Sample Application," uses the -keep object option to generate the object files.

By default, at most 50 errors are printed by `tidl`. If you want to see them all (and have more than 50 errors), use the `-error all` option. The error output is printed to the `stderr`.

See `tidl(1)` in the *BEA Tuxedo Command Reference* for details on the many other options that are available.

CHAPTER

3 Writing RPC Client and Server Programs

This topic includes the following sections:

- Handling Remoteness
- Handling Status and Exception Returns
- Using Stub Support Functions
- Using RPC Header Files
- Portability of Code
- Interacting with ATMI
- Interacting with TX

Note: Sample client and server source files are provided in Appendix A, "A Sample Application."

Handling Remoteness

The goal of TxRPC is to provide procedure calls transparently between a client in one address space and a server in another address space, potentially on different machines. However, because the client and server are not in the same address space, there are some things to remember:

- Because the client and server are in different address spaces, potentially on different machines, memory is not assumed to be shared. Program state (for example, open file descriptors) and global variables are not shared between the client and server. Any state information required must be passed from the client to the server and then back to the client for subsequent calls.
- The division of labor between the client and server has some advantages, such as providing more modularity of the software and the ability to do the work near the resources required to do the work. However, it may also mean more complexity in dealing with issues related to distributed processing, such as communication problems, independent unavailability of either the client or server, and so forth. Errors resulting from the increased complexity may require different handling from those in an interface designed for local procedure calls. The handling of errors involved in communications and/or the remote process is covered in the next topic.

Handling Status and Exception Returns

In the X/OPEN RPC specification, non-application errors are returned via status parameters or a status return. A `fault_status` value is returned if there is an RPC server failure and a `comm_status` value is returned if there is a communications failure. Status returns are specified by defining an operation return value or an `[out]` parameter of type `error_status_t` in the IDL file, and declaring the same operation or parameter to have the `[fault_status]` and/or `[comm_status]` attribute in the ACF file.

For example, an operation defined in an IDL file as:

```
error_status_t op([in,out]long *parm1, [out]error_status_t *commstat);
```

with a definition in the corresponding ACF file as:

```
[fault_status]op([comm_status]commstat);
```

returns an error from the server via the operation return, and an error in communications via the second parameter. Its use in the client code could be as follows:

```
if (op(&parm1, &commstat) != 0 || commstat != 0) /* handle error */
```

The advantage of using status returns is that the error can be handled immediately at the point of failure for fine-grained error recovery.

The disadvantage of using status returns is that the remote function has additional parameters that the local version of the function does not have. Additionally, fine-grained error recovery can be tedious and error prone (for example, some cases may be missing).

DCE defines a second mechanism called exception handling. It is similar to C++ exception handling.

The application delimits a block of C or C++ code in which an exception may be raised with the `TRY`, `CATCH`, `CATCH_ALL`, and `ENDTRY` statements. `TRY` indicates the beginning of the block. `CATCH` is used to indicate an exception-handling block for a specific exception, and `CATCH_ALL` is used to handle any exceptions for which there is not a `CATCH` statement. `ENDTRY` ends the block. `TRY` blocks are nested such that if an exception cannot be handled at a lower level, the exception can be raised to a higher level block using the `RERAISE` statement. If an exception is raised out of any exception handling block, the program writes a message to the log and exits. Details of the exception handling macros and an example are described in `TRY(3c)` in the *BEA Tuxedo C Function Reference*.

In addition to exceptions generated by the communications and server for an RPC call, exceptions are also generated for lower level exceptions, specifically operating system signals. These exceptions are documented within `TRY(3c)` in the *BEA Tuxedo C Function Reference*.

Using Stub Support Functions

There are a large number of run-time support functions (over 100) defined in the X/OPEN RPC specification. These functions need not all be supported in an X/OPEN TxRPC IDL-only environment. Most of these functions relate to binding and management which are done transparently for ATMI clients and servers.

One area that affects application portability is the management of memory allocated for stub input and output parameters and return values. The Stub Memory Management routines are supported in TxRPC run time with the exception of the two routines to handle threads. The status-returning functions include:

- `rpc_sm_allocate`
- `rpc_sm_client_free`
- `rpc_sm_disable_allocate`
- `rpc_sm_enable_allocate`
- `rpc_sm_free`
- `rpc_sm_set_client_alloc_free`
- `rpc_sm_set_server_alloc_free`
- `rpc_sm_swap_client_alloc_free`

The equivalent exception-returning functions include:

- `rpc_ss_allocate`
- `rpc_ss_client_free`
- `rpc_ss_disable_allocate`
- `rpc_ss_enable_allocate`
- `rpc_ss_free`
- `rpc_ss_set_client_alloc_free`
- `rpc_ss_set_server_alloc_free`
- `rpc_ss_swap_client_alloc_free`

Refer to *BEA Tuxedo C Function Reference* for more information on these functions.

The run-time functions are contained in `libtrpc`; building RPC clients and servers is discussed in the next topic.

The following are a few tips regarding memory management:

- When an ATMI client calls a client stub, it uses `malloc` and `free` by default. All space will be freed on return from the client stub except space allocated for `[out]` pointers (including implicit `[out]` pointers in the return value of the operation). To make freeing of `[out]` pointers easier, call `rpc_ss_enable_allocate`(), and set `alloc`/ `free` to `rpc_ss_alloc`()/ `rpc_ss_free`() before calling the RPC by calling `rpc_ss_set_client_alloc_free`(). Then `rpc_ss_disable_allocate`() can

be used to free all of the allocated memory. For example, to simplify freeing space returned from a client stub the following could be used:

```
rpc_ss_set_client_alloc_free(rpc_ss_allocate, rpc_ss_free);
ptr = remote_call_returns_pointer();
/* use returned pointer here */
...
rpc_ss_disable_allocate(); /* this frees ptr */
```

- When an ATMI server stub is executed that calls an application operation, memory allocation using `rpc_ss_allocate` is always enabled in the server stub. The `[enable_allocate]` attribute in the ACF file has no effect. All memory will be freed in the server before returning the response to the client. (In DCE, memory allocation is enabled only if `[ptr]` fields or parameters exist, or the programmer explicitly specifies `[enable_allocate]`.)
- When a server stub calls an application operation which in turn calls a client stub (that is, when a server acts as a client by calling an RPC), the `rpc_ss_set_client_alloc_free`() function must be called to set up allocation such that any space allocated will be freed when the operation returns. This is done by calling:

  ```
  rpc_ss_set_client_alloc_free(rpc_ss_allocate, rpc_ss_free);
  ```

- When calling `rpc_ss_allocate`() or `rpc_sm_allocate`(), remember to cast the output to match the data type of the pointer being set. For example:

  ```
  long *ptr;
  ptr = (long *)rpc_ss_allocate(sizeof(long));
  ```

Using RPC Header Files

To ensure that stubs from both DCE/RPC and TxRPC can be compiled in the same environment, different header filenames are used in the TxRPC implementation. This should not affect the application programmer since these header files are automatically included in the interface header file generated by the IDL compiler. However, an application program may wish to view these headers to see how a type or function is defined. The new header filenames are listed here:

- dce/nbase.h, dce/nbase.idl—renamed rpc/tbase.h and rpc/tbase.idl. Contain the declarations for pre-declared types error_status_t, ISO_LATIN_1, ISO_MULTI_LINGUAL, and ISO_UCS.
- dce/idlbase.h—renamed rpc/tidlbase.h. Contains the IDL base types, as defined in the specification (for example, idl_boolean, idl_long_int), and the function prototypes for the stub functions.
- dce/pthread_exc.h—renamed rpc/texc.h. Contains the TRY/ CATCH exception handling macros.
- dce/rpcsts.h—renamed rpc/trpcsts.h. Contains the exception and status value definitions for the RPC interface.

These header files are located in $TUXDIR/include/rpc. The TxRPC IDL compiler will look in $TUXDIR/include by default as the "system IDL directory."

Portability of Code

The output from the IDL compiler is generated in a way to allow it to be compiled in a large number of environments (see the next chapter for a discussion of compilation). However, there are some constructs that don't work in various environments. The following are a few known problems:

When compiling with Classic (non-ANSI) C, "pointers to arrays" are not allowed. For example:

```
typedef long array[10][10];
func()
  {
  array t1;
  array *t2;
  t2 = &t1;   /* & ignored, invalid assignment */
  func2(&t1); /* & ignored */
  }
```

This will make it difficult to pass "pointers to arrays" to operations as parameters in a portable fashion.

When using an array of strings where the string attribute is applied to a multi-byte structure, the results will not be as desired if the compiler pads the structure. This is not the normal case (most compilers do not pad a structure that contains only character fields), but at least one occurrence is known to exist.

Constant values are, by default, implemented by generating a `#define` for each constant. This means that names used for constants should not be used for any other names in the IDL file or any imported IDL files. A TxRPC-specific option on the `tidl` compiler, `-use_const`, may be used to get around this problem in an ANSI C environment. This option will cause `const` declarations instead of `#define` definitions to be generated. The constant values will be declared in the client and server stubs, and any other source file including the header file will simply get `extern const` declarations. Note that this has the restriction that the client and server stubs may not be compiled into the same executable file (or duplicate definition errors will occur).

There are several restrictions in the C++ environment:

- Do not use the same name for a `typedef` and a structure or union tag, unless the `typedef` name matches the `struct` or `union` name.

```
struct t1 {
  long s1;
};
typedef struct t1 t1; /* ok */
typedef long t1; /* error */
```

- Do not hide a structure or union tag declaration inside another structure or union declaration and then reference it outside.

```
struct t1 {
  struct t2 {
   long s2;
} s1;
} t1;
typedef struct t3 {
struct t2 s3; /* t2 undefined error */
} t3;
```

- Some compiler warnings may be generated. These include the following:
  - Warnings that automatic variables are declared but not used.
  - Warnings that a variable is used before being set when referenced in `sizeof()` as in the following case:

```
long *ptr;
ptr = (long *)malloc(sizeof(*ptr) * 4);
```

When coding the client and server application software, you should use the data types generated by the IDL compiler, as defined in `rpc/tidlbase.h` (listed as Emitted Macro in the following table). For instance, if you use a `long` instead of `idl_long_int`, then the data type may be 32 bits on some platforms and 64 bits on others; `idl_long_int` will be 32 bits on all platforms. Table 3-1 lists the generated data types.

Table 3-1 Generated Data Types

IDL Type	Size	Emitted Macro	C Type
boolean	8 bits	`idl_boolean`	unsigned char
char	8 bits	`idl_char`	unsigned char
byte	8 bits	`idl_byte`	unsigned char
small	8 bits	`idl_small_int`	char
short	16 bits	`idl_short_int`	short
long	32 bits	`idl_long_int`	Machines with 32-bit long: `long` Machines with 64-bit long: `int`

hyper	64 bits	`idl_hyper_int`	Machines with 32-bit long:
			Big Endian
			`struct`
			`{`
			`  long high;`
			`  unsigned long low;`
			`}`
			Little Endian
			`struct`
			` {`
			`  unsigned long low;`
			`  long high;`
			` }`
			Machines with 64-bit long:
			`long`

Table 3-1 Generated Data Types

IDL Type	Size	Emitted Macro	C Type
unsigned small	8 bits	`idl_usmall_int`	unsigned char
unsigned short	16 bits	`idl_ushort_int`	short
unsigned long	32 bits	`idl_ulong_int`	Machines with 32-bit long: `long` Machines with 64-bit long: `int`

unsigned hyper	64 bits	`idl_uhyper_int`	Machines with 32-bit long:
			Big Endian
			`struct { unsigned long high; unsigned long low; }`
			Little Endian
			`struct { unsigned long low; unsigned long high; }`
			Machines with 64-bit long:
			`unsigned long`

float	32 bits	`idl_short_float`	`float`
double	64 bits	`idl_long_float`	`double`
void *	pointer	`idl_void_p_t`	`void *`
handle_t	pointer	`handle_t`	`handle_t`

As in C, there are several classes of identifiers in the IDL. Names within each class (that is, scope or name space) must be unique:

- Constant, `typedef`, operation, and enumeration member names are in one name space.
- Structure, union, and enumeration tags are in another name space.

- Structure and union member names at the same level must be unique within the structure or union in which they are defined.
- Parameter names within the operation prototype in which they are defined must be unique.

Note that an anonymous structure or union (without a tag and not defined as part of a `typedef`) cannot be used for an operation return or a parameter.

Interacting with ATMI

The TxRPC executables use the BEA Tuxedo system to do the RPC communications. Other BEA Tuxedo interfaces and communications mechanisms can be used within the same clients and servers that are using the RPC calls. Thus, it is possible to have a single client making Request/Response calls (for example `tpcall(3c)`, `tpacall(3c)`, and `tpgetrply(3c)`), making conversational calls (`tpconnect(3c)`, `tpsend(3c)`, `tprecv(3c)`, and `tpdiscon(3c)`), and accessing the stable queue (`tpenqueue(3c)` and `tpdequeue(3c)`). When a client makes the first call to the BEA Tuxedo software, either an RPC call, any of these other communications calls, or any other ATMI call (such as a call for buffer allocation or unsolicited notification), the client automatically joins the application. However, if the application is running with security turned on or if the client must run as part of a particular resource manager group, then `tpinit(3c)` must be called explicitly to join the application. Refer to `tpinit(3c)` in the *BEA Tuxedo C Function Reference* for further details, and a list of options that can be explicitly set. When an application completes work using the BEA Tuxedo system, `tpterm(3c)` should be called explicitly to leave the application and free up any associated resources. If this is not done for native (non-Workstation) clients, the monitor detects this, prints a warning in the `userlog(3c)`, and frees up the resources. In the case of Workstation clients, the resources may not be freed up and eventually the Workstation Listener or Handler will run out of resources to accept new clients.

As with clients, servers can use any of the communication paradigms in the role of client. However, a server cannot provide (advertise) both conversational services and RPC services within the same server; as described later, an RPC server must be marked as non-conversational. Although it is possible to mix ATMI request/response and RPC services within the same server, this is not recommended. One further restriction is that RPC operations cannot call `tpreturn(3c)` or `tpforward(3c)`. Instead, RPC

operations must return as they would if called locally. Any attempt to call `tpreturn(3c)` or `tpforward(3c)` from an RPC operation will be intercepted and an error will be returned to the client (exception `rpc_x_fault_unspec` or status `rpc_s_fault_unspec`).

Two functions available to servers but not to clients are `tpsvrinit(3c)` and `tpsvrdone(3c)`, which are called when the server starts up and when it is shut down. Since the server must call `tx_open(3c)` before receiving any TxRPC operation requests, `tpsvrinit()` is a good place to call it. The default `tpsvrinit()` function already calls `tx_open()`.

Interacting with TX

The TX functions provide an interface for transaction demarcation. `tx_begin(3c)` and `tx_commit(3c)` or `tx_rollback(3c)` encapsulate any work, including communications, within a transaction. Other primitives are provided to set transaction timeout, declare the transaction as chained or unchained, and retrieve transaction information. These are discussed in detail in the *X/OPEN TX Specification*, and reviewed in the *X/OPEN TxRPC Specification*. The *X/OPEN TxRPC Specification* indicates the interactions between TX and RPC. These are summarized as follows:

- An interface or an operation can have the `[transaction_optional]` attribute which indicates that if the RPC is called within a transaction, the work done in the called operation will be part of the transaction.
- An interface or an operation can have the `[transaction_mandatory]` attribute which indicates that the RPC must be called within a transaction or the `txrpc_x_not_in_transaction` exception is returned.
- If neither of these attributes is specified, then the work in the called operation is not part of any transaction that may be active in the caller.
- If a TxRPC operation is called in the server and `tx_open(3c)` has not been called, a `txrpc_x_no_tx_open_done` exception is returned to the caller.
- TxRPC allows `tx_rollback(3c)` to be called from an operation to mark the transaction as rollback-only, such that any work performed on behalf of the transaction will be ultimately rolled back. It is recommended in this case that the

application also return an application-level error to the caller indicating that the transaction will be rolled back.

Other changes or restrictions for the IDL defined by the TxRPC specification have been described earlier in the discussion about the IDL itself.

4 Building RPC Client and Server Programs

This topic includes the following sections:

- Prerequisite Knowledge
- Building an RPC Server
- Building an RPC Client
- Building a Windows Workstation RPC Client
- Using C++
- Interoperating with DCE/RPC

Prerequisite Knowledge

The TxRPC programmer should be familiar with the C compilation system and building BEA Tuxedo ATMI clients and servers. Information on building BEA Tuxedo ATMI clients and servers is provided in the *Programming a BEA Tuxedo Application Using C*, *Programming a BEA Tuxedo Application Using COBOL*, and *Programming a BEA Tuxedo Application Using FML*. Building Workstation clients is provided in *Using the BEA Tuxedo Workstation Component*.

Building an RPC Server

RPC servers are built and configured in much that same way that ATMI Request/Response servers are. In fact, the service name space for RPC and Request/Response servers is the same. However, the names advertised for RPC services are different. For Request/Response servers, a service name is mapped to a procedure. For RPC servers, a service name is mapped to an IDL interface name. The RPC service advertised will be <*interface*>v<*major*>_<*minor*>, where <*interface*> is the interface name, and <*major*> and <*minor*> are the major and minor numbers of the version, as specified (or defaulted to 0.0) in the interface definition. Because the service name is limited to 15 characters, this limits the length of the interface name to 13 characters minus the number of digits in the major and minor version numbers. This also implies that an exact match is used on major AND minor version numbers because of the way name serving is done in the BEA Tuxedo system. Note that the interface, and not individual operations, are advertised (similar to DCE/RPC). The server stub automatically takes care of calling the correct operation within the interface.

RPC servers are built using the `buildserver(1)` command. We recommend using the `-s` option to specify the service (interface) names at compilation time. The server can then be booted using the `-A` option to get the services automatically advertised. This approach is used in the sample application, as shown in Appendix A, "A Sample Application."

The `buildserver(1)` command automatically links in the BEA Tuxedo libraries. However, the RPC run time must be linked in explicitly. This is done by specifying the `-f -ltrpc` option after any application files on the `buildserver` line. Normally, the output of the `tidl(1)` command is a server stub object file. This can be passed directly to the `buildserver` command. Note that the server stub and the application source, object, and library files implementing the operations should be specified ahead of the run-time library, also using the `-f` option. See the makefile `rpcsimp.mk`, in Appendix A, "A Sample Application," for an example.

Building an RPC Client

A native RPC client is built using the `buildclient(1)` command. This command automatically links in the BEA Tuxedo libraries. However, the RPC run time must be linked in explicitly. This is done by specifying the `-f -ltrpc` option after any application files on the `buildclient` command line. Generally, the output of the `tidl(1)` command is a client stub object file. This can be passed directly to the `buildclient` command. Note that the client stub and the application source, object, and library files executing the remote procedure calls should be specified ahead of the run-time library, also using the `-f` option. For an example, see the makefile `rpcsimp.mk` in Appendix A, "A Sample Application."

To build a UNIX Workstation client, simply add the `-w` option to the `buildclient(1)` command line so that the Workstation libraries are linked in instead of the native libraries.

Building a Windows Workstation RPC Client

Compilation of the client stub for Windows requires the `-D_TM_WIN` definition as a compilation option. This ensures that the correct function prototypes for the TxRPC and BEA Tuxedo ATMI run time functions are used. While the client stub source is the same, it must be compiled specially to handle the fact that the text and data segments for the DLL will be different from the code calling it. The header file and stub are automatically generated to allow for the declarations to be changed easily, using C preprocessor definitions. The definition `_TMF` (for "far") appears before all pointers in the header file and `_TMF` is automatically defined as "`_far`" if `_TM_WIN` is defined.

In most cases, using standard libraries, the `buildclient(1)` command can be used to link the client. The library to be used is `wtrpc.lib`.

The sample also shows how to create a Dynamic Link Library (DLL) using the client stub. This usage will be very popular when used with a visual application builder that requires DLL use (where the application code cannot be statically linked in). Windows functions are traditionally declared to have the `_pascal` calling convention. The

header file and stub are automatically generated to allow for the declarations to be changed easily, using C preprocessor definitions. _TMX (for "eXport") appears before all declared functions. By default, this definition is defined to nothing. When compiling a stub for inclusion in a DLL, _TMX should be defined to _far _pascal. Also, the files to be included in the DLL must be compiled with the large memory model. Because using _pascal automatically converts the function names to uppercase in the library, it is a good idea to run with the -port case option turned on, which does additional validation to see if two declared names differ only in case.

A complete example of building a Windows DLL is shown in Appendix A, "A Sample Application."

Note: A compilation error may occur if a TxRPC client includes windows.h, due to a duplicate uuid_t definition. It will be necessary for the application to either not include windows.h (because it is included already) or to include it within a different file in the application.

Using C++

Clients and servers can be built using C or C++, interchangeably. The header files and generated stub source files are defined in such a way that all Stub Support functions and generated operations allow for complete interoperability between C++ and C. They are declared with C linkage, that is, as extern "C," so that name mangling is turned off.

The stub object files can be built using C++ by specifying CC -c for the -cc_cmd option of tidl(1). The CC command can be used to compile and link client and server programs by setting and exporting the CC environment variable before running buildclient(1) and buildserver(1). For example:

```
tidl -cc_cmd "CC -c" -keep all t.idl
CC=CC buildserver -o server -s tv1_0 -f "-I. t_sstub.o server.c -ltrpc"
```

In the Windows environment, C++ compilation is normally accomplished via a flag on the compilation command line or a configuration option rather than a different command name. Use the appropriate options to get C++ compilation.

Interoperating with DCE/RPC

The BEA Tuxedo TxRPC compiler uses the same IDL interface as OSF/DCE but the generated stubs do not use the same protocol. Thus, a BEA Tuxedo TxRPC stub cannot directly communicate with a stub generated by the DCE IDL compiler.

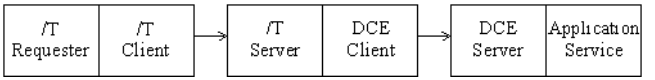
However, it is possible to have the following interoperations between DCE/RPC and BEA Tuxedo TxRPC:

- Client side stubs from both DCE and BEA Tuxedo TxRPC can be called from the same program (either client or server).
- A BEA Tuxedo ATMI server stub can call application code that calls a DCE client stub (as well as a BEA Tuxedo TxRPC client stub).
- A DCE server (manager) can call application code that calls a BEA Tuxedo TxRPC client stub.

The following sections show possible interactions between BEA Tuxedo TxRPC and OSF/DCE. In each case, the originator of the request is called the requester. This term is used instead of “client” because the requester could, in fact, be a DCE or BEA Tuxedo ATMI service making a request of another service. The terms “client” and “server” refer to the client and server stubs generated by the IDL compilers (either DCE `idl(1)` or BEA Tuxedo `tidl(1)`); these terms are used for consistency with the DCE and TxRPC terminology. Finally, the term “application service” is used for the application code that implements the procedure that is being called remotely (it is generally transparent whether the invoking software is the server stub generated by DCE or BEA Tuxedo).

BEA Tuxedo Requester to DCE Service via BEA Tuxedo Gateway

Figure 4-1 BEA Tuxedo Requester to DCE Service via BEA Tuxedo Gateway



The first approach uses a "gateway" such that the BEA Tuxedo ATMI client stub invokes a BEA Tuxedo ATMI server stub, via TxRPC, that has a DCE client stub linked in (instead of the application services) that invokes the DCE services, via DCE RPC. The advantage to this approach is that it is not necessary to have DCE on the client platform. In fact, the set of machines running BEA Tuxedo and the set of machines running DCE could be disjoint except for one machine where all such gateways are running. This also provides a migration path with the ability to move services between BEA Tuxedo and DCE. A sample application that implements this approach is described in Appendix B, "A DCE-Gateway Application."

In this configuration, the requester is built as a normal BEA Tuxedo ATMI client or server. Similarly, the server is built as a normal DCE server. The additional step is to build the gateway process which acts as a BEA Tuxedo ATMI server using a TxRPC server stub and a DCE client using a DCE/RPC client stub.

The process of running the two IDL compilers and linking the resultant files is simplified with the use of the `blds_dce(1)` command, which builds a BEA Tuxedo ATMI server with DCE linked in.

The usage for `blds_dce` is as follows:

```
blds_dce [-o output_file] [-i idl_options] [-f firstfiles] [-l lastfile] \
   [idl_file . . . ]
```

The command takes as input one or more IDL files so that the gateway can handle one or more interfaces. For each one of these files, `tidl` is run to generate a server stub and `idl` is run to generate a client stub.

This command knows about various DCE environments and provides the necessary compilation flags and DCE libraries for compilation and linking. If you are developing in a new environment, it may be necessary to modify the command to add the options and libraries for your environment.

This command compiles the source files in such a way (with `-DTMDCEGW` defined) that memory allocation is always done using `rpc_ss_allocate`(3c) and `rpc_ss_free`(3c), as described in the *BEA Tuxedo C Function Reference*. This ensures that memory is freed on return from the BEA Tuxedo ATMI server. The use of `-DTMDCEGW` also includes DCE header files instead of BEA Tuxedo TxRPC header files.

The IDL output object files are compiled, optionally with specified application files (using the `-f` and `-l` options), to generate a BEA Tuxedo ATMI server using `buildserver(1)`. The name of the executable server can be specified with the `-o` option.

When running this configuration, the DCE server would be started first in the background, then the BEA Tuxedo configuration including the DCE gateway would be booted, and then the requester would be run. Note that the DCE gateway is single-threaded so you will need to configure and boot as many gateway servers as you want concurrently executing services.

There are several optional things to consider when building this gateway.

Setting the DCE Login Context

First, as a DCE client, it is normal that the process runs as some DCE principal. There are two approaches to getting a login context. One approach is to "log in" to DCE. In some environments, this occurs simply by virtue of logging into the operating system. In many environments, it requires running `dce_login`. If the BEA Tuxedo ATMI server is booted on the local machine, then it is possible to run `dce_login`, then run `tmboot(1)` and the booted server will inherit the login context. If the server is to be booted on a remote machine which is done indirectly via `tlisten(1)`, it is necessary to run `dce_login` before starting `tlisten`. In each of these cases, all servers booted in the session will be run by the same principal. The other drawback to this approach is that the credentials will eventually expire.

The other alternative is to have the process set up and maintain its own login context. The `tpsvrinit(3c)` function provided for the server can set up the context and then start a thread that will refresh the login context before it expires. Sample code to do this is provided in `$TUXDIR/lib/dceserver.c`; it must be compiled with the `-DTPSVRINIT` option to generate a simple `tpsvrinit()` function. (It can also be used as the `main()` for a DCE server, as described in the following section.) This code is described in further detail in Appendix B, "A DCE-Gateway Application."

Using DCE Binding Handles

BEA Tuxedo TxRPC does not support binding handles. When sending an RPC from the requester's client stub to the server stub within the gateway, the BEA Tuxedo system handles all of the name resolution and choosing the server, doing load balancing between available servers. However, when going from the gateway to the DCE server, it is possible to use DCE binding. If this is done, it is recommended that two versions of the IDL file be used in the same directory or that two different directories be used to build the requester, and the gateway and server. The former approach of using two different filenames is shown in the example with the IDL file linked to a second name. In the initial IDL file, no binding handles or binding attributes

are specified. With the second IDL file, which is used to generate the gateway and DCE server, there is an associated ACF file that specifies [explicit_handle] such that a binding handle is inserted as the first parameter of the operation. From the BEA Tuxedo server stub in the gateway, a NULL handle will be generated (because handles aren't supported). That means that somewhere between the BEA Tuxedo ATMI server stub and the DCE client stub in the gateway, a valid binding handle must be generated.

This can be done by making use of the manager entry point vector. By default, the IDL compiler defines a structure with a function pointer prototype for each operation in the interface, and defines and initializes a structure variable with default function names based on the operation names. The structure is defined as:

`<INTERF>_v<major>_<minor>_epv_t<INTERF>_v<major>_<minor>_s_epv`

where `<INTERF>` is the interface name and `<major>_<minor>` is the interface version. This variable is dereferenced when calling the server stub functions. The IDL compiler option, `-no_mepv`, inhibits the definition and initialization of this variable, allowing the application to provide it in cases where there is a conflict or difference in function names and operation names. In the case where an application wants to provide explicit or implicit binding instead of automatic binding, the `-no_mepv` option can be specified, and the application can provide a structure definition that points to functions taking the same parameters as the operations but different (or static) names. The functions can then create a valid binding handle that is passed, either explicitly or implicitly, to the DCE/RPC client stub functions (using the actual operation names).

This is shown in the example in Appendix B, "A DCE-Gateway Application." The file `dcebind.c` generates the binding handle, and the entry point vector and associated functions are shown in `dceepv.c`.

Note that to specify the `-no_mepv` option when using the `blds_dce`, the `-i -no_mepv` option must be specified so that the option is passed through to the IDL compiler. This is shown in the makefile, `rpcsimp.mk`, in Appendix B, "A DCE-Gateway Application."

Authenticated RPC

Now that we have a login context and a handle, it is possible to use authenticated RPC calls. As part of setting up the binding handle, it is also possible to annotate the binding handle for authentication by calling `rpc_binding_set_auth_info`(), as described in the *BEA Tuxedo C Function Reference*. This is shown as part of generating the binding handle in `dcebind.c` in Appendix B, "A DCE-Gateway Application." This sets up the authentication (and potentially encryption) between the gateway and the DCE

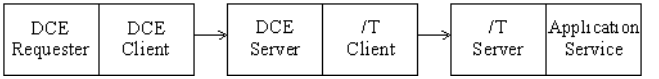
server. If the requester is a BEA Tuxedo ATMI server, then it is guaranteed to be running as the BEA Tuxedo administrator. For more information about authentication for BEA Tuxedo clients, see *Administering the BEA Tuxedo System*.

Transactions

OSF/DCE does not support transactions. That means that if the gateway is running in a group with a resource manager and the RPC comes into the BEA Tuxedo ATMI client stub in transaction mode, the transaction will not `carray` to the DCE server. There is not much you can do to solve this; just be aware of it.

DCE Requester to BEA Tuxedo Service Using BEA Tuxedo Gateway

Figure 4-2 DCE Requester to BEA Tuxedo Service Using BEA Tuxedo Gateway



In the preceding figure, the DCE requester uses a DCE client stub to invoke a DCE service which calls the BEA Tuxedo ATMI client stub (instead of the application services), which invokes the BEA Tuxedo ATMI service (via TxRPC). Note that in this configuration, the client has complete control over the DCE binding and authentication. The fact that the application programmer builds the middle server means that the application also controls the binding of the DCE server to BEA Tuxedo ATMI service. This approach would be used in the case where the DCE requester does not want to directly link in and call the BEA Tuxedo system.

The `main()` for the DCE server should be based on the code provided in `$TUXDIR/lib/dceserver.c`. If you already have your own template for the `main()` of a DCE server, there are a few things that may need to be added or modified.

First, `tpinit(3c)` should be called to join the ATMI application. If application security is configured, then additional information may be needed in the `TPINIT` buffer such as the username and application password. Prior to exiting, `tpterm(3c)` should be called to cleanly terminate participation in the ATMI application. If you look at `dceserver.c`, you will see that by compiling it with `-DTCLIENT`, code is included that calls `tpinit` and `tpterm`. The code that sets up the `TPINIT` buffer must be modified

appropriately for your application. To provide more information with respect to administration, it might be helpful to indicate that the client is a DCE client in either the user or client name (the example sets the client name to `DCECLIENT`). This information shows up when printing client information from the administration interface.

Second, since the BEA Tuxedo ATMI system software is not thread-safe, the threading level passed to `rpc_server_listen` must be set to 1. In the sample `dceserver.c`, the threading level is set to 1 if compiled with `-DTCLIENT` and to the default, `rpc_c_listen_max_calls_default`, otherwise. (For more information, refer to the *BEA Tuxedo C Function Reference*.)

In this configuration, the requester is built as a normal DCE client or server. Similarly, the server is built as a normal BEA Tuxedo ATMI server. The additional step is to build the gateway process, which acts as a BEA Tuxedo ATMI client using a TxRPC client stub, and a DCE server, using a DCE/RPC server stub.

The process of running the two IDL compilers and linking the resultant files is simplified with the use of the `bldc_dce(1)` command which builds a BEA Tuxedo ATMI client with DCE linked in.

The usage for `bldc_dce` is as follows:

```
bldc_dce [-o output_file] [-w] [-i idl_options] [-f firstfiles] \
   [-l lastfiles] [idl_file . . . ]
```

The command takes as input one or more IDL files so that the gateway can handle one or more interfaces. For each one of these files, `tidl` is run to generate a client stub and `idl` is run to generate a server stub.

This command knows about various DCE environments and provides the necessary compilation flags and DCE libraries. If you are developing in a new environment, it may be necessary to modify the command to add the options and libraries for your environment. The source is compiled in such a way (with `-DTMDCEGW` defined) that memory allocation is always done using `rpc_ss_allocate` and `rpc_ss_free` (described in the *BEA Tuxedo C Function Reference*) to ensure that memory is freed on return. The use of `-DTMDCEGW` also includes DCE header files instead of BEA Tuxedo TxRPC header files.

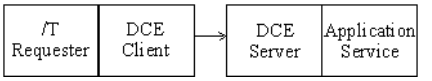
The IDL output object files are compiled, optionally with specified application files (using the `-f` and `-l` options), to generate a BEA Tuxedo ATMI client using `buildclient(1)`. Note that one of the files included should be the equivalent of the `dceserver.o`, compiled with the `-DTCLIENT` option.

The name of the executable client can be specified with the -o option.

When running this configuration, the BEA Tuxedo ATMI configuration must be booted before starting the DCE server so that it can join the BEA Tuxedo ATMI application before listening for DCE requests.

BEA Tuxedo Requester to DCE Service Using DCE-only

Figure 4-3 BEA Tuxedo Requester to DCE Service Using DCE-only



This approach assumes that the DCE environment is directly available to the client (this can be a restriction or disadvantage in some configurations). The client program has direct control over the DCE binding and authentication. Note that this is presumably a mixed environment in which the requester is either a BEA Tuxedo ATMI service that calls DCE services, or a BEA Tuxedo client (or server) that calls both BEA Tuxedo and DCE services.

When compiling BEA Tuxedo TxRPC code that will be used mixed with DCE code, the code must be compiled such that DCE header files are used instead of the TxRPC header files. This is done by defining `-DTMDCE` at compilation time, both for client and server stub files and for your application code. If you are generating object files from `tidl(1)`, you must add the `-cc_opt -DTMDCE` option to the command line. The alternative is to generate `c_source` from the IDL compiler and pass this C source (not object files) to `bldc_dce or blds_dce` as in the following examples:

```
tidl -keep c_source -server none t.idl
idl -keep c_source -server none dce.idl
bldc_dce -o output_file -f client.c -f t_cstub.c -f dce_cstub.c
```

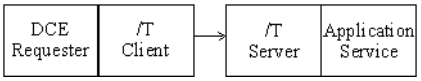
or

```
blds_dce -o output_file -s service -f server.c -f t_cstub.c -f dce_cstub.c
```

In this example, we are not building a gateway process so `.idl` files *cannot* be specified to the `build` commands. Also note that the `blds_dce` command cannot figure out the service name associated with the server so it must be supplied on the command line using the `-s` option.

DCE Requester to BEA Tuxedo Service Using BEA Tuxedo-only

Figure 4-4 DCE Requester to BEA Tuxedo Service Using BEA Tuxedo-only



In this final case, the DCE requester calls the BEA Tuxedo client stub directly.

Again, `-DTMDCE` must be used at compilation time, both for client and server stub files and for your application code. In this case the requester must be a BEA Tuxedo ATMI client:

```
tidl -keep c_source -client none t.idl
bldc_dce -o output_file -f -DTCLIENT -f dceserver.c -f t_cstub.c
```

Note that `dceserver.c` should call `tpinit(3c)` to join the application and `tpterm(3c)` to leave the application, as was discussed earlier.

Building Mixed DCE/RPC and BEA Tuxedo TxRPC Clients and Servers

This section summarizes the rules to follow if you are compiling a mixed client or server without using the `bldc_dce(1)` or `blds_dce(1)` commands:

- When compiling the generated client and server stubs, and compiling the client and server application software that includes the header file generated by `tidl(1)`, `TMDCE` must be defined (for example, `-DTMDCE=1`). This causes some DCE header files to be used instead of the BEA Tuxedo TxRPC header files. Also, some versions of DCE have a DCE compilation shell that adds the proper directories for the DCE header files and ensures the proper DCE definitions for the local environment. This shell should be used instead of directly using the C compiler. The DCE/RPC compiler and TMDCE definition can be specified using the `-cc_cmd` option on `tidl`. For example:

  ```
  tidl -cc_cmd "/opt/dce/bin/cc -c -DTMDCE=1" simp.idl
  ```

  or

```
tidl -keep c_source simp.idl
 /opt/dce/bin/cc -DTMDCE=1 -c -I. -I$TUXDIR/include simp_cstub.c
 /opt/dce/bin/cc -DTMDCE=1 -c -I. -I$TUXDIR/include client.c
```

On a system without such a compiler shell, it might look like the following:

```
cc <DCE options> -DTMDCE=1 -c -I. -I$(TUXDIR)/include \
   -I/usr/include/dce simp_cstub.c
```

Refer to the DCE/RPC documentation for your environment.

- If the server makes an RPC call, then `set_client_alloc_free()` should be called to set the use of `rpc_ss_allocate()` and `rpc_ss_free()`, as described earlier. (For more information, refer to the *BEA Tuxedo C Function Reference*.)
- When linking the executable, use `-ldrpc` instead of `-ltrpc` to get a version of the BEA Tuxedo TxRPC runtime that is compatible with DCE/RPC. For example:

```
buildclient -o client -f client.o -f simp_cstub.o -f dce_cstub.o \
   -f-ldrpc -f-ldce -f-lpthreads -f-lc_r
```

or

```
CC=/opt/dce/bin/cc buildclient -d " " -f client.o -f simp_cstub.o \
   -f dce_cstub.o -f -ldrpc -o client
```

Assume that `simp_cstub.o` was generated by `tidl(1)` and `dce_cstub.o` was generated by `idl`. The first example shows building the client without a DCE compiler shell; in this case, the DCE library (`-ldce`), threads library (`-lpthreads`), and re-entrant C library (`-lc_r`) must be explicitly specified. The second example shows the use of a DCE compiler shell which transparently includes the necessary libraries. In some environments, the libraries included by `buildserver` and `buildclient` for networking and XDR will conflict with the libraries included by the DCE compiler shell (there may be re-entrant versions of these libraries). In this case, the `buildserver(1)` and `buildclient(1)` libraries may be modified using the `-d` option. If a link problem occurs, trying using `-d " "` to leave out the networking and XDR libraries, as shown in the example above. If the link still fails, try running the command without the `-d` option and with the `-v` option to determine the libraries that are used by default; then use the `-d` option to specify a subset of the libraries if there is more than one. The correct combination of libraries is environment-dependent because the networking, XDR, and DCE libraries vary from one environment to another.

Note: Mixing DCE and BEA Tuxedo TxRPC stubs is not currently supported on Windows.

CHAPTER

5 Running the Application

This topic includes the following sections:

- Prerequisite Knowledge
- Configuring the Application
- Booting and Shutting Down the Application
- Administering the Application
- Using Dynamic Service Advertisement

Prerequisite Knowledge

The BEA Tuxedo ATMI system administrator modifying the configuration to add RPC servers should be familiar with creating an ASCII configuration file (the format is described in `UBBCONFIG(5)`), and loading the binary configuration using `tmloadcf(1)`. These activities are described in *Administering the BEA Tuxedo System*.

Configuring the Application

When configuring an RPC server, it is configured the same as a Request/Response server. One entry is needed in the SERVERS page for each RPC server or group of RPC servers. (MAX can be set to a value greater than one to configure multiple RPC servers with one entry.) An RQADDR can optionally be specified so that multiple instances of an RPC server share the same request queue (multiple servers, single queue configuration). The CONV parameter must be not specified or must be set to N (for example, CONV=N). See the sample configuration file in Appendix A, "A Sample Application."

If a server will be part of a transaction, then it must be in a group on a machine that has a TLOGDEVICE. The GROUPS entry must be configured with a TMSNAME and an OPENINFO string that are used to access the associated resource manager.

It is optional to specify SERVICES entries. If specified, the service name must be the name described in the previous chapter, based on the interface name and version number. This entry is needed only if you want to give a specific load, priority, or transaction time that is different than the defaults. It can also be used to turn on the AUTOTRAN feature, which ensures that a transaction is automatically started for the service if the incoming request is not in transaction mode. Do not use the service entry to specify buffer types BUFTYPE since the only buffer type handled is CARRAY. Also, do not specify ROUTING because routing is not supported for RPC requests.

The tmloadcf(1) command is used to load the ASCII configuration file into a binary TUXCONFIG file before the application is booted.

Note that entries for RPC servers can be added to a booted application using the tmconfig command, as described in tmconfig, wtmconfig(1) in the *BEA Tuxedo Command Reference*.

Booting and Shutting Down the Application

When the configuration has been modified, boot the application using tmboot(1). The application is shut down using tmshutdown(1). See the example in Appendix A, "A Sample Application."

The RPC servers are booted and shut down in the same way that Request/Response servers are. They can be booted or shut down as part of the entire configuration with the `-y` option, as part of a group with the `-g` option, as part of a logical machine with the `-l` option, or by server name with the `-s` option.

Administering the Application

RPC servers appear as Request/Response servers in the administration interfaces. As mentioned above, `tmconfig` can be used for dynamic reconfiguration of RPC servers and services, as described in `tmconfig, wtmconfig(1)` in the *BEA Tuxedo Command Reference*. The `tmadmin(1)` command can be used to monitor RPC servers. The RPC server name and associated run-time information (for example, services or operations run, load, and so forth) can be printed using the `tmadmin printserver` command. The RPC services (interfaces) that are available can be printed using `printservice`. For samples of the output, see the example in Appendix A, "A Sample Application."

Using Dynamic Service Advertisement

RPC services can be dynamically controlled in the same way that Request/Response services can be controlled. Remember that the service name is not the operation name, but the interface name and version number, as described earlier. Generally, the service name is specified at the time that `buildserver(1)` is run using the `-s` option and automatically advertised when the server is booted with the `-A` option. Service (interface) names can be dynamically advertised either from `tmadmin` using the `adv` command or from within the server using the `tpadvertise(3c)` function. Service (interface) names can be dynamically unadvertised either from `tmadmin` using the `unadv` command or from within the server using the `tpunadvertise(3c)` function. Service names can also be temporarily suspended and unsuspended (resumed) from `tmadmin(1)`. Note that unadvertising or suspending a service name makes all operations defined in the associated interface unavailable.

CHAPTER

A A Sample Application

This topic includes the following sections:

- Appendix Contents
- Prerequisites
- Building the rpcsimp Application

Appendix Contents

This appendix contains a description of a one-client, one-server application called `rpcsimp` that uses TxRPC. The source files for this interactive application are distributed with the BEA Tuxedo ATMI software, except they are not included in the RTK binary delivery.

Prerequisites

Before you can run this sample application, the BEA Tuxedo software must be installed so that the files and commands referred to in this chapter are available.

Building the rpcsimp Application

rpcsimp is a very basic BEA Tuxedo ATMI application that uses TxRPC. It has one application client and one server. The client calls the remote procedure calls (operations) to_upper() and to_lower(), which are implemented in the server. The operation to_upper() converts a string from lowercase to uppercase and returns it to the client, while to_lower() converts a string from uppercase to lowercase and returns it to the client. When each procedure call returns, the client displays the string output on the user's screen.

What follows is a procedure to build and run the example.

Step 1: Create an Application Directory

Make a directory for rpcsimp and cd to it:

```
mkdir rpcsimpdir
cd rpcsimpdir
```

Note: This is suggested so you will be able to see clearly the rpcsimp 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 the C shell (csh).

Step 2: Set Environment Variables

Set and export the necessary environment variables:

```
TUXDIR=<pathname of the BEA Tuxedo System root directory>
TUXCONFIG=<pathname of your present working directory>/TUXCONFIG
PATH=$PATH:$TUXDIR/bin
# SVR4, Unixware
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$TUXDIR/lib
# HPUX
SHLIB_PATH=$LD_LIBRARY_PATH:$TUXDIR/lib
# RS6000
LIBPATH=$LD_LIBRARY_PATH:$TUXDIR/lib
export TUXDIR TUXCONFIG PATH LD_LIBRARY_PATH SHLIB_PATH LIBPATH
```

Note: You need `TUXDIR` and `PATH` to be able to access files in the BEA Tuxedo ATMI directory structure and to execute BEA Tuxedo ATMI commands. You need to set `TUXCONFIG` to be able to load the configuration file. It may also be necessary to set an environment variable (for example, `LD_LIBRARY_PATH`) if shared objects are being used.

Step 3: Copy files

Copy the `rpcsimp` files to the application directory:

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

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

Step 4: List the Files

List the files:

```
$ ls
client.c
rpcsimp.mk
server.c
simp.idl
ubbconfig
wclient.def
wsimpdll.def
$
```

Note: This list does not include files that are used in the DCE-Gateway example described in Appendix B, "A DCE-Gateway Application."

The files that make up the application are described in the following sections.

IDL Input File—simp.idl

Listing A-1 simp.idl

```
[uuid(C996A680-9FC2-110F-9AEF-930269370000), version(1.0) ]

interface changecase
{
/* change a string to upper case */
void to_upper([in, out, string] char *str);

/* change a string to lower case */
void to_lower([in, out, string] char *str);
}
```

This file defines a single interface, `changecase` version 1.0, with two operations, `to_upper` and `to_lower`. Each of the operations takes a NULL-terminated character string, that is both an input and output parameter. Because no ACF file is provided, status variables are not used and the client program must be able to handle exceptions. Each operation has a void return indicating that no return value is generated. `simp.idl` is used to generate the stub functions (see below).

The Client Source Code—client.c

Listing A-2 client.c

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

main(argc, argv)
int argc;
char **argv;
{
  idl_char str[100];
  unsigned char error_text[100];
  int status;

  if (argc > 1) {/* use command line argument if it exists */
    (void) strncpy(str, argv[1], 100);
```

```
    str[99] = '\0';
  }
  else
    (void) strcpy(str, "Hello, world");

  TRY
  to_upper(str);
  (void) fprintf(stdout, "to_upper returns: %s\n", str);
  to_lower(str);
  (void) fprintf(stdout, "to_lower returns: %s\n", str);
  /* control flow continues after ENDTRY */
  CATCH_ALL
    exc_report(THIS_CATCH); /* print to stderr */
    (void) tpterm();
    exit(1);
  ENDTRY

  (void) tpterm();
  exit(0);
}
```

The header, `simp.h`, which is generated by the IDL compiler based on `simp.idl`, has the function prototypes for the two operations. The `simp.h` header also includes the header files for the RPC run-time functions (none appear in this example) and exception handling. The `atmi.h` header file is included because `tpterm(3c)` is called. If an argument is provided on the command line, then it is used for the conversion to uppercase and lowercase (the default being "`hello world`"). Exception handling is used to catch any errors. For example, exceptions are generated for unavailable servers, memory allocation failures, communication failures, and so forth. The `TRY` block encapsulates the two remote procedure calls. If an error occurs, the execution will jump to the `CATCH_ALL` block which converts the exception (`THIS_CATCH`) into a string, prints it to the standard error output using `exc_report`, and exits. Note that in both the abnormal and normal execution, `tidl(1)` is called to leave the application gracefully. If this is not done, a warning is printed in the `userlog(3c)` for non-Workstation clients, and resources are tied up (until the connection times out, for Workstation clients).

The Server Source Code—server.c

Listing A-3 server.c

```
#include <stdio.h>
#include <ctype.h>
#include "tx.h"
#include "simp.h"

int
tpsvrinit(argc, argv)
int argc;
char **argv;
{
  if (tx_open() != TX_OK) {
    (void) userlog("tx_open failed");
    return(-1);
  }
  (void) userlog("tpsvrinit() succeeds.");
  return(1);
}
void
to_upper(str)
idl_char *str;
{
  idl_char *p;
  for (p=str; *p != '\0'; p++)
    *p = toupper((int)*p);
  return;
}

void
to_lower(str)
idl_char *str;
{
  idl_char *p;
  for (p=str; *p != '\0'; p++)
    *p = tolower((int)*p);
  return;
}
```

As with `client.c`, this file includes `simp.h`.

It also includes `tx.h` because `tx_open(3c)` is called (as required by the X/OPEN TxRPC specification, even if no resource manager is accessed). A `tpsvrinit(3c)` function is provided to ensure that `tx_open()` is called once at boot time. On failure, `-1` is returned and the server fails to boot. This is done automatically, so you may not need to supply it.

The two operation functions are provided to do the application work, in this case, converting to upper and lower case.

Makefile—rpcsimp.mk

Listing A-4 rpcsimp.mk

```
CC=cc
CFLAGS=
TIDL=$(TUXDIR)/bin/tidl
LIBTRPC=-ltrpc
all:  client server

# Tuxedo client
client: simp.h simp_cstub.o
        CC=$(CC) CFLAGS=$(CFLAGS) $(TUXDIR)/bin/buildclient \
            -oclient -fclient.c -fsimp_cstub.o -f$(LIBTRPC)

# Tuxedo server
server: simp.h simp_sstub.o
        CC=$(CC) CFLAGS=$(CFLAGS) $(TUXDIR)/bin/buildserver \
                  -oserver -s changecasev1_0 -fserver.c -fsimp_sstub.o \
                  -f$(LIBTRPC)

simp_cstub.o simp_sstub.o simp.h:     simp.idl
        $(TIDL) -cc_cmd "$(CC) $(CFLAGS) -c" simp.idl
#
# THIS PART OF THE FILE DEALING WITH THE DCE GATEWAY IS OMMITTED
#

# Cleanup
clean::
         rm -f *.o server $(ALL2) ULOG.* TUXCONFIG
         rm -f stderr stdout *stub.c *.h simpdce.idl gwinit.c
clobber: clean
```

The `makefile` builds the executable client and server programs.

The part of the `makefile` dealing with the DCE Gateway (described in Appendix B, "A DCE-Gateway Application," is omitted from the figure.

The client is dependent on the `simp.h` header file and the client stub object file. `buildclient` is executed to create the output client executable, using the `client.c` source file, the client stub object file, and the `-ltrpc` RPC run-time library.

The server is dependent on the `simp.h` header file and the server stub object file. `buildserver` is an output server executable, using the `server.c` source file, the server stub object file, and the `-ltrpc` RPC run-time library.

The client and server stub object files and the `simp.h` header file are all created by running the `tidl` compiler on the IDL input file.

The `clean` target removes any files that are created while building or running the application.

The Configuration File—ubbconfig

The following is a sample ASCII configuration file. The machine name, `TUXCONFIG`, `TUXDIR`, and `APPDIR` must be set based on your configuration.

Listing A-5 ubbconfig

```
*RESOURCES
IPCKEY     187345
MODEL      SHM
MASTER     SITE1
PERM       0660
*MACHINES
<UNAME>   LMID=SITE1
          TUXCONFIG="<TUXCONFIG>"
          TUXDIR="<TUXDIR>"
          APPDIR="<APPDIR>"
#         MAXWSCLIENTS=10
*GROUPS
GROUP1    LMID=SITE1      GRPNO=1
*SERVERS
server SRVGRP=GROUP1 SRVID=1
#WSL SRVGRP=GROUP1 SRVID=2 RESTART=Y GRACE=0
#         CLOPT="-A -- -n <address> -x 10 -m 1 -M 10 -d <device>"
#
```

```
# Tuxedo-to-DCE Gateway
#simpgw SRVGRP=GROUP1 SRVID=2
*SERVICES
*ROUTING
```

The lines for MAXWSCLIENTS and WSL would be uncommented and are used for a Workstation configuration. The literal *netaddr* for the Workstation listener must be set as described in WSL(5) in the *BEA Tuxedo File Formats and Data Descriptions Reference*.

Step 5: Modify the Configuration

Edit the ASCII ubbconfig configuration file to provide location-specific information (for example, your own directory pathnames and machine name), as described in the next step. The text to be replaced is enclosed in angle brackets. You need to substitute the full pathname for TUXDIR, TUXCONFIG, and APPDIR, and the name of the machine on which you are running. The following is a summary of the required values.

TUXDIR
: The full pathname of the root directory of the BEA Tuxedo software, as set above.

TUXCONFIG
: The full pathname of the binary configuration file, as set above.

APPDIR
: The full pathname of the directory in which your application will run.

UNAME
: The machine name of the machine on which your application will run; this is the output of the UNIX command uname -n.

For a Workstation configuration, the MAXWSCLIENTS and WSL lines must be uncommented and the <*address*> must be set for the Workstation Listener. (See WSL(5) for further details.)

Step 6: Build the Application

Build the client and server programs by running the following:

```
make -f rpcsimp.mk TUXDIR=$TUXDIR
```

Step 7: Load the Configuration

Load the binary TUXCONFIG configuration file by running the following:

```
tmloadcf -y ubbconfig
```

Step 8: Boot the Configuration

Boot the application by running the following:

```
tmboot -y
```

Step 9: Run the Client

1. The native client program can be run by optionally specifying a string to be converted first to uppercase, and then to lowercase, as shown in the following:

   ```
   $ client HeLlO
   to_upper returns: HELLO
   to_lower returns: hello
   $
   ```

2. When running on a Workstation, set the WSNADDR environment variable to match the address specified for the WSL program. The Windows client can be run by executing:

   ```
   >win wclient
   ```

   Note: The dynamic link library may be used in a separately developed application such as a visual builder.

Step 10: Monitor the RPC Server

You can monitor the RPC server using `tmadmin`(1). In the following example, `psr` and `psc` are used to view the information for the `server` program. Note that the length of the RPC service name causes it to be truncated in terse mode (indicated by the "+"); verbose mode can be used to get the full name.

Listing A-6 tmadmin psr and psc Output

```
$ tmadmin
> psr
a.out Name Queue Name  Grp Name ID RqDone Load Done Current Service
---------- ----------  -------- -- ------ --------- ---------------
BBL        587345      SITE1     0   0            0 ( IDLE )
server     00001.00001 GROUP1    1   2          100 ( IDLE )

> psc
Service Name Routine Name a.out Name Grp Name ID  Machine # Done Status
------------ ------------ ---------- -------- --  ------- ------ ------
ADJUNCTBB    ADJUNCTBB    BBL        SITE1     0    SITE1      0 AVAIL
ADJUNCTADMIN ADJUNCTADMIN BBL        SITE1     0    SITE1      0 AVAIL
changecasev+ changecasev+ server     GROUP1    1    SITE1      2 AVAIL

> verbose
Verbose now on.

> psc -g GROUP1
    Service Name: changecasev1_0
    Service Type: USER
    Routine Name: changecasev1_0
      a.out Name: /home/sdf/trpc/rpcsimp/server
      Queue Name: 00001.00001
      Process ID: 8602, Machine ID: SITE1
        Group ID: GROUP1, Server ID: 1
    Current Load: 50
Current Priority: 50
Current Trantime: 30
   Requests Done: 2
  Current status: AVAILABLE
> quit
```

Step 11: Shut Down the Configuration

Shut down the application by running the following:

```
tmshutdown -y
```

Step 12: Clean Up the Created Files

Clean up the created files by running the following:

```
make -f rpcsimp.mk clean
```

CHAPTER

B A DCE-Gateway Application

This topic includes the following sections:

- Appendix Contents
- Prerequisites
- What Is the DCE-Gateway Application?
- Installing, Configuring, and Running the rpcsimp Application

Appendix Contents

This appendix builds on the `rcpsimp` application described in Appendix A, “A Sample Application.” The server is changed to be an OSF/DCE server and a gateway is used so that the BEA Tuxedo ATMI client can communicate with the server using explicit binding and authenticated RPCs. The source files for this interactive application are distributed with the BEA Tuxedo ATMI software development kit.

Prerequisites

This topic requires knowledge about DCE, and a DCE tutorial is beyond the scope of this document. For further reading, try *Guide to Writing DCE Applications* by John Shirley, et. al., published by O'Reilly and Associates, Inc.

What Is the DCE-Gateway Application?

This application is an extension to the `rpcsimp` application. As before, the client calls the remote procedure calls (operations) `to_upper`() and `to_lower`().

In this case, the RPC goes from the BEA Tuxedo ATMI client to the DCE Gateway process that forwards the request to a DCE server. To make this example more realistic, the communications from the Gateway process to the DCE server use explicit binding instead of automatic binding and an authenticated RPC.

What follows is a procedure to build and run the example. The client can run on any platform described in Appendix A, "A Sample Application." There is no difference in building or running the client and it will not be described further in this chapter. The gateway and DCE server must run on a POSIX platform that also has DCE software installed on it. This chapter will not discuss installation or compilation of the clients on the Workstation platforms.

The sample programs work on platforms that conform to OSF/DCE software standards.

Installing, Configuring, and Running the rpcsimp Application

The following steps provide you with the instructions for installing, configuring, and running the sample application.

Step 1: Create an Application Directory

Make a directory for `rpcsimp` and `cd` to it:

```
mkdir rpcsampdir
cd rpcsampdir
```

Note: This is suggested so you will be able to see clearly the `rpcsimp` 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 the C shell (`csh`).

Step 2: Set Your Environment

Set and export the necessary environment variables:

```
TUXDIR=<pathname of the BEA Tuxedo root directory>
TUXCONFIG=<pathname of your present working directory>/tuxconfig
PATH=$PATH:$TUXDIR/bin
# SVR4, Unixware
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$TUXDIR/lib
# HPUX
SHLIB_PATH=$LD_LIBRARY_PATH:$TUXDIR/lib
# RS6000
LIBPATH=$LD_LIBRARY_PATH:$TUXDIR/lib
export TUXDIR TUXCONFIG PATH LD_LIBRARY_PATH SHLIB_PATH LIBPATH
```

You need TUXDIR and PATH to be able to access files in the BEA Tuxedo ATMI directory structure and to execute BEA Tuxedo ATMI commands. You need to set TUXCONFIG to be able to load the configuration file. It may also be necessary to set an environment variable (for example, LD_LIBRARY_PATH) if shared objects are being used.

Step 3: Copy the Files

Copy the rpcsimp files to the application directory:

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

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

Step 4: List the Files

List the files:

```
$ ls
client.c
dcebind.c
dceepv.c
dcemgr.c
dceserver.c
rpcsimp.mk
simp.idl
simpdce.acf
ubbconfig
$
```

(Some files that are not referenced in this section are omitted.)

The files that make up the application are described in the following sections. The client.c, simp.idl, and ubbconfig files described in Appendix A, “A Sample Application,” are not discussed further.

IDL ACF File—simpdce.acf

Listing B-1 simpdce.acf

```
[explicit_handle]interface changecase
{
}
```

The `simp.idl` file used in the earlier example will be used to build the gateway and the DCE server. However, since it is being compiled by both the DCE and BEA Tuxedo IDL compilers, two different versions of the `simp.h` header file are being generated with the same name. Additionally, we wish to use an ACF file in this example so that we can specify explicit binding for the server, but not for the client. The recommended approach is to link the IDL file to a second filename within the same directory, using one for TxRPC without binding and one for DCE/RPC with an explicit handle. In this case, `simp.idl` is renamed `simpdce.idl` and the associated ACF file is `simpdce.acf`. The makefile creates `simpdce.idl` and when the IDL compiler is executed, it also will find `simpdce.acf`. Note that the ACF file is used simply to indicate that all operations in the interface will use explicit handles. Because the operations are defined in the IDL file without [handle] parameters as the first parameter, one will be added automatically to the function prototype and to the stub function calls.

Binding Function—dcebind.c

In the interest of space, the source code for `dcebind.c` is not included here but can be found in `$TUXDIR/apps/rpcsimp`.

This file has a function, `dobind`(), that does the following three things:

- It gets a binding handle for the DCE server with the desired interface specification and gets the associated endpoint for a fully resolved handle.
- It does some authentication of the server by getting the principal name for the server and checking the Security registry to see if the principal is a member of a specified group.
- It also annotates the binding handle so that an authenticated RPC is done. The protection level is packet level integrity (mutual authentication on every call with a packet `checksum`) using DCE private key authentication and DCE PAC-based authorization.

The following things need to be modified in `dcebind.c`:

- <*`HOST`*> needs to be changed to the name of the host machine where the DCE server will be run. This is part of the service name that is put into the directory and follows the convention that the service name ends with `_host`. You may choose to get rid of the suffix entirely (if you do, the same change needs to be made in `dceserver.c`).
- <*`SERVER_PRINCIPAL_GROUP`*> must be changed to the group associated with the DCE principal running the server. It is used as part of the mutual authentication.
- The server principal group must be created by running `rgy_edit` as `cell_admin`, the server principal must be created, an account must be added for the principal with the group, and a key table must be created for the server. You must also create a principal and account for yourself to run the client. An example script to create these DCE entities is shown in Step 8: Configuring DCE.

Entry Point Vector—dceepv.c

Listing B-2 dceepv.c

```
#include <simpdce.h>   /* header generated by IDL compiler */
#include <dce/rpcexc.h> /* RAISE macro */

static void myto_upper(rpc_binding_handle_t hdl, idl_char *str);
static void myto_lower(rpc_binding_handle_t hdl, idl_char *str);

/*
 * A manager entry point vector is defined so that we can generate
 * a valid DCE binding handle to go to the DCE server.
 * Note that the input handle to entry point functions will always
 * be NULL since Tuxedo TxRPC doesn't support handles.
 */

 /* Manager entry point vector with two operations */
changecase_v1_0_epv_t changecase_v1_0_s_epv = {
  myto_upper,
  myto_lower
};

int dobind(rpc_binding_handle_t *hdl);

void
myto_upper(rpc_binding_handle_t hdl, idl_char *str)
{
  rpc_binding_handle_t handle;
  if (dobind(&handle) 0) {  /* get binding handle for server */
   userlog("binding failed");
   RAISE(rpc_x_invalid_binding);
  }
  to_upper(handle, str);   /* call DCE client stub */
}

void
myto_lower(rpc_binding_handle_t hdl, idl_char *str)
{
  rpc_binding_handle_t handle;
  if (dobind(&handle) 0) { /* get binding handle for server */
   userlog("binding failed");
   RAISE(rpc_x_invalid_binding);
  }
  to_lower(handle, str);   /* call DCE client stub */
}
```

dceepv.c contains the manager entry point vector used in the gateway. It is called by the BEA Tuxedo ATMI server stub and calls the DCE client stub. The data type for the structure is defined in simpdce.h, which is included in dceepv.c, and it is initialized with the local functions myto_upper() and myto_lower(). Each of these functions simply calls dobind() to get the binding handle that has been annotated for authenticated RPC and calls the associated client stub function.

DCE Manager—dcemgr.c

Listing B-3 dcemgr.c

```
#include <stdio.h>
#include <ctype.h>
#include "simpdce.h"    /* header generated by IDL compiler */
#include <dce/rpcexc.h>  /* RAISE macro */

#include <dce/dce_error.h> /* required to call dce_error_inq_text */
#include <dce/binding.h>  /* binding to registry */
#include <dce/pgo.h>    /* registry i/f */
#include <dce/secidmap.h> /* translate global name -> princ name */

void
checkauth(rpc_binding_handle_t handle)
{
  int error_stat;
  static unsigned char error_string[dce_c_error_string_len];
  sec_id_pac_t *pac;           /* client pac */
  unsigned_char_t *server_principal_name; /* requested server principal */
  unsigned32  protection_level;     /* protection level */
  unsigned32  authn_svc;         /* authentication service */
  unsigned32  authz_svc;         /* authorization service */
  sec_rgy_handle_t rgy_handle;
  error_status_t status;
  /*
   * Check the authentication parameters that the client
   * selected for this call.
   */
   rpc_binding_inq_auth_client(
     handle,           /* input handle */
     (rpc_authz_handle_t *)&pac,  /* returned client pac */
     &server_principal_name,    /* returned requested server princ */
     &protection_level,      /* returned protection level */
     &authn_svc,          /* returned authentication service */
     &authz_svc,    /* returned authorization service */
     &status);
  if (status != rpc_s_ok) {
```

```
    dce_error_inq_text(status, error_string, &error_stat);
    fprintf(stderr, "%s %s\n", "inq_auth_client failed",
      error_string);
    RAISE(rpc_x_invalid_binding);
    return;
  }
  /*
   * Make sure that the caller has specified the required
   * level of protection, authentication, and authorization.
   */
  if (protection_level != rpc_c_protect_level_pkt_integ ||
    authn_svc != rpc_c_authn_dce_secret ||
    authz_svc != rpc_c_authz_dce) {
      fprintf(stderr, "not authorized");
      RAISE(rpc_x_invalid_binding);
      return;
    }
    return;
}

void
to_upper(rpc_binding_handle_t handle, idl_char *str)
{
  idl_char *p;

  checkauth(handle);

  /* Any ACL or reference monitor checking could be done here */

  /* Convert to upper case */
  for (p=str; *p != '\0 '; p++)
    *p = toupper((int)*p);
  return;
}

void
to_lower(rpc_binding_handle_t handle, idl_char *str)
{
  idl_char *p;

  checkauth(handle);

  /* Any ACL or reference monitor checking could be done here */

  /* Convert to lower case */
  for (p=str; *p != '\0 '; p++)
    *p = tolower((int)*p);
  return;
}
```

dcemgr.c has the manager code for the DCE server. The checkauth() function is a utility function to check the authentication of the client (level of protection, authentication, and authorization). Each of the operations, to_upper and to_lower, calls this function to validate the client and then does the operation itself. In an application using access control lists, the ACL checking would be done after the authentication checking and before the work of the operation.

DCE Server - dceserver.c

In the interest of space, the source code for dceserver.c is not included here. There are several modifications needed for this file based on your environment:

- <*HOST*> needs to be changed to the name of the host machine where the DCE server will be run. This is part of the service name that is put into the directory and follows the convention that the service names ends with *_host*. You may choose to get rid of the suffix entirely (if you do, the same change needs to be made in dcebind.c).
- <*DIRECTORY*> needs to be set to the full pathname of the directory where you will create the server key table. The key table is created by executing the following:

```
rgy_edit
ktadd -p SERVER_PRINCIPAL -pw PASSWORD -f SERVER_KEYTAB
q
```

where *SERVER_PRINCIPAL* is the DCE principal under which the server will be run, *PASSWORD* is the password associated with the principal, and *SERVER_KEYTAB* is the name of the server key table.

<*PRINCIPAL*> must be changed to the name of the DCE principal under which the server will be run.

The *ANNOTATION* can be changed to an annotation to be stored in the directory entry for the server.

dceserver.c is actually used twice in the application: once as the main() for the DCE server and again (linked to gwinit.c and compiled with -DTPSVRINIT in the makefile) as the tpsvrinit() for the DCE gateway.

When compiled without extra macro definitions, this file generates a `main()` (with `argc` and `argv` command-line options) for a DCE server that does the following:

- Registers its interfaces
- Creates its server binding information and endpoints
- Establishes its DCE login context for the server principal using information in the server key table
- Registers its authentication information
- Gets its bindings and registers the information in the endpoint map
- Exports the binding information to the directory name space
- Optionally, adds its name to a group in the name space
- Listens for requests
- Cleans up after `rpc_server_listen` returns

The program could be modified to look at and use its command_line options.

When compiled with `-DTCLIENT`, this file generates a `main()` as above but calls `tpinit`() to join the BEA Tuxedo ATMI application as a client, and calls `tpterm`() before exiting. This would be used for a DCE gateway for calls coming from DCE to BEA Tuxedo (such that the process is a DCE server and a BEA Tuxedo ATMI client).

When compiled with `-DTPSVRINIT`, this file generates a `tpsvrinit`() (with `argc` and `argv` server command-line options) for a BEA Tuxedo server that does the following:

- Establishes its DCE login for the principal using the information in the server key table
- Registers its authentication information
- Calls `tx_open` to open any resource managers associated with the server

The program could be modified to look at and use its command-line options.

In each of these cases, the login context is established by calling `establish_identity`, which gets the network identity for the server, uses the server's secret key from the key table file to unseal the identity, and sets the login context for the process. Two threads are started: one to refresh the login context just before it expires, and a second thread to periodically change the server's secret key.

Makefile—rpcsimp.mk

Listing B-4 rpcsimp.mk

```
CC=cc
CFLAGS=
TIDL=$(TUXDIR)/bin/tidl
LIBTRPC=-ltrpc
all: client server
# Tuxedo client
client: simp.h simp_cstub.o
  CC=$(CC) CFLAGS=$(CFLAGS) $(TUXDIR)/bin/buildclient -oclient \
    -fclient.c -fsimp_cstub.o -f$(LIBTRPC)
#
# OMIT Tuxedo server
#

# Tuxedo Gateway example
# Uses Tuxedo client above plus a gateway server and a DCE server
#
#
# Alpha FLAGS/LIBS
#DCECFLAGS=-D_SHARED_LIBRARIES -Dalpha -D_REENTRANT -w -I. \
  -I/usr/include/dce -I$(TUXDIR)/include
#DCELIBS=-ldce -lpthreads -lc_r -lmach -lm
#
#
# HPUX FLAGS/LIBS
#DCECFLAGS=-Aa -D_HPUX_SOURCE -D_REENTRANT -I. \
  -I/usr/include/reentrant -I${TUXDIR}/include
#DCELIBS=-Wl,-Bimmediate -Wl,-Bnonfatal -ldce -lc_r -lm
#
IDL=idl

ALL2=client simpgw dceserver
all2: $(ALL2)

# TUXEDO-to-DCE Gateway
simpdce.idl: simp.idl
  rm -f simpdce.idl
  ln simp.idl simpdce.idl

gwinit.c: dceserver.c
  rm -f gwinit.c
  ln dceserver.c gwinit.c
```

```
gwinit.o: gwinit.c
  $(CC) -c $(DCECFLAGS) -DTPSVRINIT gwinit.c

dceepv.o: dceepv.c simpdce.h
   $(CC) -c $(DCECFLAGS) dceepv.c

dcebind.o: dcebind.c simpdce.h
  $(CC) -c $(DCECFLAGS) dcebind.c

simpgw: simpdce.idl gwinit.o dcebind.o dceepv.o
  blds_dce -i -no_mepv -o simpgw -f -g -f gwinit.o -f \
    dcebind.o -f dceepv.o simpdce.idl

# DCE server
simpdce_sstub.o simpdce.h: simpdce.idl
  $(IDL) -client none -keep object simpdce.idl

dceserver.o: dceserver.c simpdce.h
  $(CC) -c $(DCECFLAGS) dceserver.c

dcemgr.o: dcemgr.c simpdce.h
  $(CC) -c $(DCECFLAGS) dcemgr.c

dceserver: simpdce_sstub.o dceserver.o dcemgr.o
  $(CC) dceserver.o simpdce_sstub.o dcemgr.o -o dceserver \
    $(DCELIBS)

# Cleanup
clean::
  rm -f *.o server $(ALL2) ULOG.* TUXCONFIG
  rm -f stderr stdout *stub.c *.h simpdce.idl gwinit.c

clobber: clean
```

The `makefile` builds the executable client, gateway, and DCE server programs.

Before building the software, `rpcsimp.mk` must be modified to set the correct options and libraries for building the DCE server. As sent out, the makefile contains the proper settings for several platforms. Based on the platform that you are using, uncomment (delete the pound sign) in front of the correct pair of `DCECFLAGS` and `DCELIBS` variables, or add your own definitions for a different platform.

Briefly reviewing the makefile, the client is built in the same fashion as in Appendix A, “A Sample Application.” The DCE gateway is built by passing `simpdce.idl` to `blds_dce`, which builds a BEA Tuxedo ATMI server that acts as a gateway to DCE. Also included are `gwinit.o` (a version of `dceserver.c` compiled with `-DTPSVRINIT`), `dobind.o` (to get the binding handle for the DCE server), and

dceepv.o (the manager entry point vector). Note that -i -no_mepv is specified so that the IDL compiler does not generate its own manager entry point vector. The DCE server is built compiling simpdce.idl with the DCE IDL compiler, and including dceserver.o and dcemgr.o.

Step 5: Modify the Configuration

1. Modify the ASCII ubbconfig configuration file as described in Appendix A, "A Sample Application." (This step is mandatory.)
2. In the SERVERS section, comment out the *server* line by putting a pound sign (#) at the beginning of the line. (Do not comment out the *dceserver* line.)

Step 6: Build the Application

1. Before building the software, you must modify rpcsimp.mk to set the correct options and libraries for building the DCE server, as described above.
2. Build the client and server programs by running the following:

   ```
   make -f rpcsimp.mk TUXDIR=$TUXDIR all2
   ```

Step 7: Load the Configuration

Load the binary TUXCONFIG configuration file by running the following:

```
tmloadcf -y ubbconfig
```

Step 8: Configuring DCE

To set up DCE entities for running this example, as described earlier, you must customize (for your environment) identifiers in all capital letters.

- If you already have a DCE principal for yourself, you do not need to create `MYGROUP`, `MYPRINCIPAL`, or the associated account.
- This example assumes that the `cell_admin` password is the default `-dce`. (You can change this password as necessary.)
- The *`SERVER_PRINCIPAL`* must be the same as the BEA Tuxedo administrator identifier, because the server must be booted as the BEA Tuxedo administrator and the server must be able to read the server key table.

Listing B-5 DCE Configuration

```
$ dce_login cell_admin -dce-
$ rgy_edit
> domain group
> add SERVER_PRINCIPAL_GROUP
> add MYGROUP
> domain principal
> add SERVER_PRINCIPAL
> add MYPRINCIPAL
> domain account
> add SERVER_PRINCIPAL -g SERVER_PRINCIPAL_GROUP -o none -pw \
    SERVERPASSWORD -mp -dce-
> add MYPRINCIPAL -g MYGROUP -o none -pw MYPASSWORD -mp -dce-
> ktadd -p SERVER_PRINCIPAL -pw SERVERPASSWORD -f SERVER_KEYTAB
> q
$ chown SERVER_PRINCIPAL SERVER_KEYTAB
$ chmod 0600 SERVER_KEYTAB
```

Step 9: Boot the Configuration

1. Log in as *SERVER_PRINCIPAL* (the owner of the server key table).
2. Start the DCE server by running the following:

   ```
   dceserver &
   ```

   The message `Server ready` is displayed just before the DCE server starts listening for requests.
3. Boot the BEA Tuxedo ATMI application by running the following:

   ```
   tmboot -y
   ```

Step 10: Run the Client

The client program can be run by optionally specifying a string to be converted, first to uppercase, and then to lowercase:

```
$ client HeLlO
to_upper returns: HELLO
to_lower returns: hello
$
```

Step 11: Shut Down the Configuration

1. Shut down the application by running the following:

   ```
   tmshutdown -y
   ```
2. Stop the DCE server.

Step 12: Clean Up the Created Files

Clean up the created files by running the following:

```
make -f rpcsimp.mk clean
```