BEA Tuxedo

TxRPC を使用した BEA Tuxedo アプリケーションのプログラミング

BEA Tuxedo リリース 8.0J
8.0 版
2001 年 10 月

TxRPC を使用した BEA Tuxedo アプリケーションのプログラム

Document Edition	Date	Software Version
8.0J	2001 年 10 月	BEA Tuxedo リリース 8.0J

目次

このマニュアルについて

このマニュアルでは、BEA Tuxedo 環境で TxRPC 機能をプログラミングして使用し、リモート・プロシージャ・コール (RPC: Remote Procedure Call) をインプリメントする方法について説明します。

このマニュアルでは、以下の内容について説明します。

- 「第 1 章 TxRPC の概要」では、TxRPC 機能 を紹介します。
- 「第 2 章 インターフェイス定義言語 (IDL) の使用」では、インターフェイス定義言語 (IDL: Interface Definition Language) を使用して TxRPC アプリケーションを開発するためのガイドラインを示します。
- 「第 3 章 RPC クライアント / サーバ・プログラムの作成」では、あるアドレス空間にあるクライアントと別のアドレス空間にあるサーバとの間で、透過的なプロシージャ・コールを提供するプログラムの開発手順を示します。
- 「第 4 章 RPC クライアント / サーバ・プログラムの構築」では、RPC クライアント・プログラムとサーバ・プログラムを C++ を使用して作成し、DCE/RPC との相互運用性を確立する方法を説明します。
- 「第 5 章 アプリケーションの実行」では、TxRPC アプリケーションを作成、起動、およびシャットダウンする手順を示します。この章では、サービスを動的に宣言する方法も説明します。
- 「付録 A アプリケーション例」では、TxRPC を使用したクライアント・サーバ・アプリケーションのサンプルを示します。
- 「付録 B DCE ゲートウェイ・アプリケーション」では、OSF/DCE サーバとゲートウェイを使用したクライアント / サーバ・アプリケーションのサンプルを示します。アこのサンプルでは、BEA Tuxedo ATMI クライアントは明示的バインディングと認証済み RPC を使用してサーバと通信します。

対象読者

このマニュアルは、次のような読者を対象としています。

- BEA Tuxedo 環境で TxRPC アプリケーションの作成と管理を担当する管理者
- BEA Tuxedo 環境で TxRPC アプリケーションのプログラミングを行うアプリケーション開発者

このマニュアルは、BEA Tuxedo プラットフォーム、および C 言語または COBOL 言語のプログラミングの知識があることを前提としています。

e-docs Web サイト

BEA 製品のマニュアルは BEA 社の Web サイト上で参照することができます。BEA ホーム・ページの [製品のドキュメント] をクリックするか、または http://edocs.beasys.co.jp/e-docs/index.html に直接アクセスしてください。

マニュアルの印刷方法

このマニュアルは、ご使用の Web ブラウザで一度に 1 ファイルずつ印刷できます。Web ブラウザの [ファイル] メニューにある [印刷] オプションを使用してください。

このマニュアルの PDF 版は、Web サイト上にあります。また、マニュアルの CD-ROM にも収められています。この PDF を Adobe Acrobat Reader で開くと、マニュアル全体または一部をブック形式で印刷できます。PDF 形式を利用するには、BEA Tuxedo Documents ページの [PDF 版] ボタンをクリックして、印刷するマニュアルを選択します。

Adobe Acrobat Reader をお持ちではない場合は、Adobe Web サイト (http://www.adobe.co.jp/) から無償で入手できます。

関連情報

以下の BEA Tuxedo マニュアルには、BEA Tuxedo TxRPC コンポーネントの使用方法および BEA Tuxedo 環境で TxRPC アプリケーションをインプリメントする方法についての関連情報が掲載されています。

- 『BEA Tuxedo コマンド・リファレンス』
- 『BEA Tuxedo C リファレンス』

インターフェイス定義言語 (IDL) に関する外部資料については、「第 2 章 インターフェイス定義言語 (IDL) の使用」の「参考資料」を参照してください。

サポート情報

皆様の BEA Tuxedo マニュアルに対するフィードバックをお待ちしています。ご意見やご質問がありましたら、電子メールで **docsupport-jp@bea.com** までお送りください。お寄せいただきましたご意見は、BEA Tuxedo マニュアルの作成および改訂を担当する BEA 社のスタッフが直接検討いたします。

電子メール メッセージには、BEA Tuxedo 8.0 リリースのマニュアルを使用していることを明記してください。

BEA Tuxedo に関するご質問、または BEA Tuxedo のインストールや使用に際して問題が発生した場合は、**www.bea.com** の BEA WebSUPPORT を通して BEA カスタマ・サポートにお問い合わせください。カスタマ・サポートへの問い合わせ方法は、製品パッケージに同梱されているカスタマ・サポート・カードにも記載されています。

カスタマ・サポートへお問い合わせの際には、以下の情報をご用意ください。

- お客様のお名前、電子メール・アドレス、電話番号、Fax 番号
- お客様の会社名と会社の住所
- ご使用のマシンの機種と認証コード
- ご使用の製品名とバージョン
- 問題の説明と関連するエラー・メッセージの内容

表記上の規則

このマニュアルでは、以下の表記規則が使用されています。

規則	項目
太字	用語集に定義されている用語を示します。
Ctrl + Tab	2 つ以上のキーを同時に押す操作を示します。
イタリック体	強調またはマニュアルのタイトルを示します。

等幅テキスト	コード・サンプル、コマンドとオプション、データ構造とメンバ、データ型、ディレクトリ、およびファイル名と拡張子を示します。また、キーボードから入力する文字も示します。
	例 :
	`#include <iostream.h> void main ( ) the pointer psz`
	`chmod u+w *`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

等幅太字	コード内の重要な単語を示します。 例 : `void **commit** ( )`
等幅イタリック体	コード内の変数を示します。 例 : `String *expr*`
大文字	デバイス名、環境変数、および論理演算子を示します。 例 : LPT1 SIGNON OR
{ }	構文の行で選択肢を示します。かっこは入力しません。

規則	項目
[ ]	構文の行で省略可能な項目を示します。かっこは入力しません。 例 : `buildobjclient [-v] [-o name ] [-f file-list]...[-l file-list]...`
|	構文の行で、相互に排他的な選択肢を分離します。記号は入力しません。
...	コマンド行で次のいずれかを意味します。 ■ コマンド行で同じ引数を繰り返し指定できること ■ 省略可能な引数が文で省略されていること ■ 追加のパラメータ、値、その他の情報を入力できること 省略符号は入力しません。 例 : `buildobjclient [-v] [-o name ] [-f file-list]...[-l file-list]...`
. . .	コード例または構文の行で、項目が省略されていることを示します。省略符号は入力しません。

1 TxRPC の概要

ここでは、次の内容について説明します。

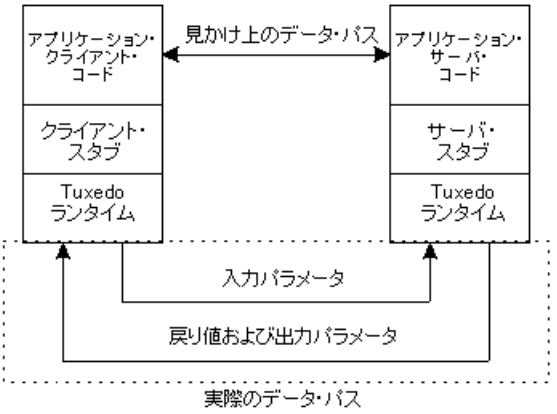
- TxRPC とは

TxRPC とは

TxRPC 機能により、プログラマはリモート・プロシージャ・コール (RPC) インターフェイスを使用できます。つまり、クライアント・プロセスは、ローカル関数呼び出しを使用しているほかのプロセス内で、リモート関数 (リモート・サービス) を呼び出すことができます。アプリケーションの作成者は、オペレーション（プロシージャ）と、インターフェイス定義言語 (IDL: Interface Definition Language) を使用してこれらのオペレーションに対してパラメータとして渡されるデータ型を指定する必要があります。オペレーション群は 1 つのインターフェイス内でグループにまとめられます。IDL コンパイラを使用して、スタブと呼ばれる代替プロシージャを生成します。このスタブを使用して、そのオペレーションをリモートに実行できます。まず理解する必要のある重要な概念は、2 つの基本的なネーミングのレベルがあることです。まず、インターフェイスが名前を持ち、1 つのインターフェイス内で 1 つ以上のオペレーションに名前がつけられます。インターフェイスは、実行時に利用可能になります。つまり、インターフェイス内のすべてのオペレーションが呼び出し可能になります。1 つのインターフェイス内の各オペレーションが利用可能になるわけではありません。個別のオペレーションを利用可能にする必要がある場合は、固有のインターフェイス内でオペレーションを定義します。

次の図は、RPC がローカル・プロシージャ・コールのように見える方法を示しています。

図 1-1 RPC の通信



クライアント・アプリケーション・コードは、IDL ファイル内で定義されたオペレーション (関数) 群の 1 つを呼び出します。サーバ側にある実際の関数を呼び出す代わりに、クライアント・スタブが呼び出されます。データ型とオペレーションを定義する IDL 入力ファイルに基づいて、IDL コンパイラがクライアント・スタブを作成します。入力パラメータ、戻り値の型、出力パラメータは、オペレーションごとに定義されます。クライアント・スタブは入力パラメータを受け取り、それらを単一のデータ・バッファに変換し、そのデータをサーバに送って応答を待ちます。その後サーバが戻り値および出力パラメータとして返したバッファ形式のデータをアンパックします。クライアント・プロセスとサーバ・プロセス間の通信は、マシン内、マシン間のいずれでも BEA Tuxedo ATMI のランタイムで処理されます。

サーバ側では、ランタイムがインターフェイスのサーバ・スタブを呼び出します。サーバ・スタブも IDL コンパイラが生成します。サーバ・スタブは入力パラメータを含むデータ・バッファをアンパックします。場合によってはサーバ・スタブは、オペレーションの出力パラメータに必要な領域を割り当て、オペレーションを呼び出して復帰を待ち、戻り値と出力パラメータを 1 つのバッファ内にパックしてクライアントに応答を返します。

アプリケーションからは、単純なローカル・プロシージャ・コールが行われているように見えます。スタブとランタイムは、ローカルなアドレス空間 (プロセス) 以外のリモート・プロシージャの呼び出しを隠ぺいします。

リモート・プロシージャ・コールを使用するアプリケーションを作成する手順は、これらのコールを用いない場合のアプリケーションの作成手順とほぼ同じです。大部分の時間はクライアントとサーバ用のアプリケーション、つまり実際にアプリケーションが行う作業のコーディングに費やされるでしょう。BEA Tuxedo ATMI のランタイムにより、アプリケーション・プログラマは、通信やクライアント・マシンで使用しているデータ形式からサーバ・マシンで使用しているデータ形式への変換などを配慮しなくて済みます。サーバ間の通信に TxRPC を使うこともできます。

単体型アプリケーションを作成する手順以外に、クライアントとサーバ間のインターフェイスを完全に定義する手順も必要になります。前述のように、インターフェイスにはリモート・プロシージャ・コールに用いるデータ型とオペレーションの定義が含まれます。通常、インターフェイスの定義を含むファイル名には拡張子 `.idl` をつけます。この規則に従えば、ファイルの種類がわかります。

すべてのインターフェイスはそれぞれ固有の識別子を持つ必要があります。この汎用一意識別子 （UUID:Universal Unique Identifier）は、128 ビットで構成され、すべてのインターフェイス間でそのインターフェイスを一意に識別します。アプリケーション・プログラマは、`uuidgen` プログラムを実行して UUID を生成します。`uuidgen` プログラムに `-i` オプションを指定して実行すると、新たな UUID を含むインターフェイス・テンプレートが作成されます。付録 A の 1 ページ「アプリケーション例」に簡単な RPC アプリケーションのコードを含む開発例を示しています。この例では、最初の手順として `uuidgen` コマンドの実行方法とその出力結果を示しています。`uuidgen` コマンドのその他のオプションについては、`uuidgen`(1) のマニュアル・ページを参照してください。

実行時にこの UUID を使用して、クライアント・スタブが受信側のサーバ・スタブと一致することを確認します。つまり、UUID は、BEA Tuxedo ATMI のランタイムが検証するためにクライアントからサーバに送られるもので、アプリケーション・プログラマに対しては透過的です。

UUID の検証以外にも、各インターフェイスはインターフェイスに関連付けられたバージョン番号も持っています。このバージョン番号はメジャー番号とマイナー番号から構成されます。バージョン番号がインターフェイス定義の一部として指定されない場合、デフォルトで 0.0 に設定されます。そのため、同じインターフェイスに対して利用できるバージョンが複数存在する可能性があります。クライアントは、特定のバージョンのインターフェイスから作成されたスタブ内の RPC を呼び出すことにより、特定のバージョンのインターフェイスを要求します。バージョンが異なっていることは、データ型、オペレーション・パラメータ、または戻り値が変更されているか、あるいはそのインターフェイスにオペレーションが追加、削除されたことを意味します。このため、RPC を正常に実行するには、クライアントとサーバの UUID およびバージョンが一致している必要があります。アプリケーション・プログラマは、同じバージョン番号を持つインターフェイスには、同じインターフェイスまたは互換性のあるインターフェイスを提供する必要があります。

uuidgen によってテンプレート IDL が作成された後、アプリケーション・プログラムは、そのインターフェイス内のすべてのデータ型およびオペレーションを定義する必要があります。IDL 言語は、プロシージャ・ステートメントを除く C や C++ の宣言部によく似ています。typedef ステートメントでデータ型を宣言し、関数プロトタイプでオペレーションを宣言します。さらに情報を追加する場合は IDL 属性を用います。IDL 言語では、属性指定は [in] のように角かっこで囲みます。属性を使用して、実行時にポインタを NULL にできるかどうかなどのポインタの形式に関する情報や、パラメータが入力、出力、またはその両方のいずれかなどのパラメータに関係する情報を提供できます。IDL 言語および関連するコンパイラについての詳細は、第 2 章の 1 ページ「インターフェイス定義言語 (IDL) の使用」の章で説明します。

IDL ファイルのほかに、インターフェイスの追加属性を指定するために、オプションの ACF (Attribute Configuration File) も用意されています。各 RPC オペレーションの状態を返すために、状態変数をオペレーション内で定義することが最も重要です。状態変数の使用法の詳細は、第 3 章の 1 ページ「RPC クライアント / サーバ・プログラムの作成」の章で説明します。ACF ファイル内の属性は、IDL ファイル内の属性とは異なり、クライアントとサーバ間の通信には影響を与えませんが、アプリケーション・コードと、生成されたスタブとの間のインターフェイスに影響を及ぼします。

BEA Tuxedo ATMI のランタイム使用時には、クライアントとサーバのバインディング (接続) の管理は透過的に行われます。クライアントやサーバのアプリケーション・コードからは、クライアントとサーバのバイディングを管理するための情報は得られません。対称的に、OSF DCE ランタイム使用時には、バインディング管理のためにプログラマにかかる負担は大変なものがあります。BEA Tuxedo ATMI ランタイムは OSF DCE ランタイム関数をサポートしていません。IDL や ACF ファイル内のバインディング属性は無視します。

IDL ファイルおよびオプションの ACF ファイルは、IDL コンパイラを使ってコンパイルされます。IDL コンパイラはまずヘッダ・ファイルを 1 つ作成します。このヘッダ・ファイルには、IDL ファイル内で定義されたオペレーションの型定義と関数プロトタイプがすべて含まれています。インターフェイス内で定義された RPC を呼び出すアプリケーション内でヘッダ・ファイルをインクルードできます。入力ファイルが *file*.idl と *file*.acf であれば、デフォルトのヘッダ・ファイルの名前は *file*.h になります。コンパイラはクライアントとサーバの両方のスタブ・コードを作成します (例 :*file*_cstub.c および *file*_sstub.c)。これらのスタブ・ファイルは前述したように、データのパッケージ処理と RPC の通信処理を含みます。特に指定しない限り、IDL コンパイラは C コンパイラを起動し、クライアントとサーバのスタブ・オブジェクト・ファイル (例 :*file*_cstub.o および *file*_sstub.o) を作成します。その後、スタブ・ソース・ファイルは削除されます。IDL コンパイラには、さまざまなオプションがあります。制限付きの生成、ソースおよびオブジェクト・ファイルの保持、および出力ファイル名やディレクトリの変更を指定できます。詳細については、tidl(1) のマニュアル・ページを参照してください。

インターフェイスの定義が完了した後の大部分の作業はアプリケーション・コードを記述することです。クライアント・コードでは、インターフェイス内で定義されたオペレーションを呼び出します。サーバ・コードではそのオペレーションを実装する必要があります。サーバは、RPC を呼び出すことにより、クライアントとしても動作できる事に注意してください。アプリケーションを作成する上での注意事項については、第 2 章の 1 ページ「インターフェイス定義言語 (IDL) の使用」の章でより詳細に解説します。

アプリケーション・コードが完成すると、いよいよそのコードをコンパイルして BEA Tuxedo ATMI のランタイムとリンクします。このプロセスを簡略化するためのプログラムが 2 つ用意されています。それは、サーバ用の `buildserver` とクライアント用の `buildclient` です。これらのプログラムを使用して任意のソース・ファイルをコンパイルし、BEA Tuxedo ATMI のランタイムを使用して、コンパイルされたオブジェクトとライブラリ・ファイルをリンクして実行形式のファイルを作成します。これらのプログラムでは代替のコンパイラやコンパイラ・オプションを指定することもできます。詳細については `buildserver`(1) と `buildclient`(1) のマニュアル・ページを参照してください。

サーバとクライアントを作成するプロセス全体を図 1-2 と図 1-3 に示します。異なるプラットフォーム上でのクライアントとサーバ・プログラムの作成については、第 4 章の 1 ページ「RPC クライアント / サーバ・プログラムの構築」を参照してください。

図 1-2　RPC サーバの作成方法

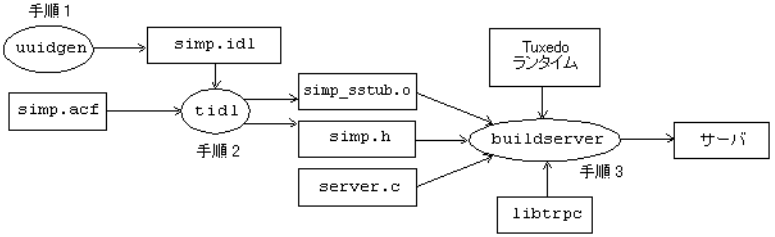
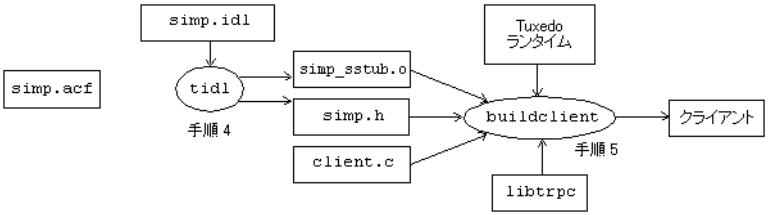


図 1-2 は、サーバを作成するプロセスを示します。

1. `uuidgen` を実行し、`UUID` を持つスケルトンの IDL ファイル (`simp.idl`) を生成します。テンプレート IDL ファイルを編集し、インターフェイス定義言語を使用してクラインアトとサーバ間のインターフェイスを定義します。
2. `simp.idl` およびオプションの `simp.acf` を使って IDL コンパイラ (`tidl`) を実行し、インターフェイス・ヘッダ・ファイルとサーバ・スタブ・オブジェクト・ファイルを生成します。

3. サーバのアプリケーション・コード (server.c) を記述した後、buildserver を実行し、そのコードをコンパイルしてサーバ・スタブ、BEA Tuxedo ATMI のランタイム、および TxRPC ランタイムとリンクし実行可能なサーバを生成します。

図 1-3 RPC クライアントの作成方法



上の図はクライアントを作成するプロセスを示します。

4. 手順 1 で生成された IDL ファイルを用いて、IDL コンパイラ (tidl) を実行し、インターフェイス・ヘッダ・ファイルとクライアント・スタブ・オブジェクト・ファイルを生成します。
5. クライアントのアプリケーション・コード (client.c) を記述した後、buildclient を実行し、そのコードをコンパイルしてクライアント・スタブ、BEA Tuxedo ATMI のランタイム、および TxRPC ランタイムとリンクし実行可能なクライアントを生成します。

アプリケーション・クライアントおよびサーバが作成されると、アプリケーションを構成および起動することができ、クライアントの実行が可能になります。詳細については、第 5 章の 1 ページ「アプリケーションの実行」で説明しています。

2 インターフェイス定義言語 (IDL) の使用

ここでは、次の内容について説明します。

- uuidgen による IDL テンプレートの作成
- 言語の変更点
- TxRPC 仕様に基づく変更
- 言語の拡張
- サポートされていない機能
- IDL コンパイラ tidl の使用

参考資料

BEA Tuxedo TxRPC は、『DCE: REMOTE PROCEDURE CALL』(Doc Code: P312 ISBN 1-872630-95-2) の第 3 章「Interface Definition Language」で説明されている IDL 文法と関連機能をサポートしています。この本は以下から入手できます。

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

電話 : +44 (0) 494 813844
ファクシミリ : +44 (0) 494 814989

X/OPEN ドキュメントは、ATMI 環境の BEA Tuxedo 製品が準拠し、その言語および規則の原典とするものです。X/OPEN TxRPC IDL-only インターフェイスがサポートされていることに注意してください。DCE バイディングとランタイムに関するドキュメントの一部は適用されません。X/OPEN ドキュメントは OSF AES/RPC ドキュメントを基本としています。チュートリアル、プログラマーズ・ガイドとして入手可能な文献は多数ありますが、ほとんどのものには最新の機能は記載されていません。OSF から入手できるプログラマーズ・ガイドとして、Prentice-Hall の『OSF DCE Application Development Guide』(Englewood Cliffs, New Jersey, 07632) があります。

『X/OPEN Preliminary Specification for TxRPC Communication Application Programming Interface』(TxRPC 通信アプリケーション・プログラミング・インターフェイスのための X/OPEN 予備仕様) もまた X/OPEN から入手できます (上記を参照)。TxRPC は、RPC 用のトランザクション・サポートをオリジナルの X/OPEN RPC インターフェイスに追加しています。

uuidgen による IDL テンプレートの作成

汎用一意識別子 (UUID) はインターフェイスを一意に識別するために使用します。`uuidgen` コマンドを実行して UUID を生成します。出力は次のようになります。

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

次にこのテンプレートを使用して、型定義、定数、およびオペレーションを追加するアプリケーション用の IDL 入力ファイルを作成します。

ATMI と DCE の両方の `uuidgen(1)` コマンドを利用できる場合は、DCE コマンドでテンプレートを生成する必要があります。DCE バージョンでは通常、以下で説明するとおり、ノード・アドレスを取得するためにマシン固有の手法を採用しています。

ATMI の `uuidgen` コマンドは、`-s` オプション (初期化された C 構造体として UUID 文字列を生成する) と、`-t` オプション (以前の形式の UUID 文字列を新規形式に変換する) をサポートしていない点を除いて、DCE コマンドと同じです。インターフェイスの詳細については `uuidgen`(1) のマニュアル・ページを参照してください。

uuidgen コマンドは、ISO/IEC 8802-3 (ANSI/IEEE 802.3) に記載されているように 48 ビット・ノード・アドレスを要求します。プラットフォームに依存せずにこの値を求める方法は存在しません。また一部のマシン (ワークステーションなど) ではまったく利用できないこともあります。ATMI の uuidgen コマンドでは、次の手法が使用されています。

- NADDR 環境変数が *num.num.num.num.num.num* の形式の値で設定されている場合、インターネット・スタイルのアドレスが採用され、48 ビットのノード・アドレスに変換されます。ここで *num* は 0 と 255 を含む、0 から 255 までの値です。これにより 8802-3 ノード・アドレスの使用法に準拠できます。また、このアドレスにアクセスできないユーザも、インターネット・アドレス (8802-3 アドレスと同じではない) 等の別の値を利用できるようになります。インターネット・アドレスを使用している場合、インターネット・アドレスは 32 ビット・アドレスなので、最後の *num.num* は 0.0 とします。
- NADDR 環境変数が設定されておらず、WSNADDR 環境変数が *0xnnnnnnnnnnnnnnnn* 形式の値に設定されている場合、ワークステーションで使用されるような、16 進数のネットワーク・アドレスが採用されることになります。これは 8802-3 アドレスではなく、また最後の 16 ビットはゼロとして処理されることに注意してください。
- NADDR 環境変数と WSNADDR 環境変数のいずれも設定されていない場合 (また Windows でもない場合)、そのマシンの uname を用いて /etc/hosts 内のマシン・エントリと照合し、インターネット・スタイルのアドレスを取得します。
- 上記のいずれにも当てはまらない場合、警告が表示され、アドレス 00.00.00.00.00.00 が採用されます。これは、一意 UUID を生成できる可能性が低くなるので望ましい方法ではありません。

言語の変更点

IDL コンパイラは IDL 文法を認識して、入力に基づいてスタブ関数を生成します。文法とその指定方法は、本章前半に記載した X/OPEN リファレンスと OSF/DCE リファレンスの両方に詳しく説明されています。次の項目で説明するいくつかの変更を加えることで文法はそのまま認識されます。

TxRPC 仕様に基づく変更

次に X/OPEN TxRPC 仕様に定義されている基本 X/OPEN RPC 仕様に対する変更点を示します。

- TxRPC 仕様からの最も重要な拡張点は、IDL ファイルのインターフェイスとオペレーション属性に [transaction_optional] と [transaction_mandatory] 属性を追加した点です。[transaction_optional] は、トランザクション内で RPC を実行する場合、トランザクションの一部としてリモート・サービスが実行されることを示します。[transaction_mandatory] 属性ではトランザクション内で RPC が実行される必要があります。これらの属性の指定がなければ、リモート・サービスはクライアントのトランザクションの一部ではありません。
- X/OPEN TxRPC IDL-only では型と属性のバインディングは必須ではありません。属性のバインディングには、[handle]、[endpoint]、[auto_handle]、[implicit_handle]、および [explicit_handle] があります。これらの属性は tidl(1) によって認識されますがサポートされていません (無視されます)。handle_t 型に対しても特別な処理は行われません。ほかの定義済みの型と同様に受け渡され、ハンドルとしては処理されません。
- X/OPEN TxRPC IDL-only ではパイプは必須ではありません。tidl は [local] モードでしかパイプをサポートしません。つまり、パイプはヘッダ・ファイル用として指定できますが、スタブ生成用としては指定できません。
- X/OPEN TxRPC IDL-only では [idempotent]、[maybe]、および [broadcast] 属性は必須ではありません。tidl(1) はこれらの属性を無視します。

言語の拡張

次に X/OPEN RPC 仕様で拡張された点を示します。多くの場合、言語は C 言語により密接に従うように拡張されており、ANSI C から IDL プロトタイプに変換することにより、既存インターフェイスの移植を簡素化しています。

- X/OPEN 仕様では、文字定数と文字列は移植可能なセット、つまり スペース (0x20) からチルダ (0x7e) までに制限されています。OSF DCE RPC では、文字セット (0x01 から 0xff まで) 内のその他の文字の使用も認められています。

- C と同様、以下の演算子は区切り文字として扱われます。

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

  つまり、これらのトークンの 1 つが先行するまたは後続する場合は、識別子または数字の前または後に空白を入れる必要はないということです IDL 仕様では `a=b+3` は認められず、`a = b + 3` のように空白を入れる必要があります。これは OSF DCE IDL コンパイラの動作によるものと思われます。

- 公開されている X/OPEN 仕様では、フィールド名とパラメータ名に型名を使用できないという制約があります。この制約により、すべての名前を単一の名前空間に置くことができます。この制約は C、C++ または OSF IDL コンパイラの仕様と整合性がないので、強制するものではありません。

- X/OPEN 仕様は、パラメータまたは関数の結果として匿名列挙法を認めておらず、またポインタの対象として匿名構造体または共用体を認めていません。これらについては OSF DCE IDL コンパイラでは認められています。これらの制約は強制されません。インターフェイス名およびバージョンに基づく名前は、マーシャリング中に使用するために生成されます。

- C と同様に、列挙値 (定数) は整数定数式で使用できます。これも DCE IDL コンパイラの動作によるものと思われます。

- 現時点の X/OPEN RPC 仕様の定義では、次のようにオペレーション宣言子の前にポインタを置くことは文法上認められません。

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

  また、構造体または共用体を返すことも認めていません。すべてを定義済みの型に隠ぺいできるという点で正しいと考えられますが、DCE IDL コンパイラと、もちろん C コンパイラはより豊富なオペレーションを返すことを認めています。サポートされている文法は以下のようになります。

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

  この場合の <*declarator*> は、<*function_declarator*> を含む必要があります。<*function_declarator*> が存在しない場合、変数は宣言されますが、結果としてエラーになります。オペレーションの配列または配列を返すオペレーションを宣言することは共にこの文法で認められていて、宣言としては認識されますが、エラーが通知されます。

- IDL <*type_declaration*> が宣言子のリストを取るように、<*ACS_type_declaration*> は <*ACS_named_type*> 値を取ります。これは DCE IDL コンパイラの動作によるものと思われます。

- フィールド操作言語 (FML:Field Manipulation Language) を使用して作成、操作されるフィールド化バッファは多くの BEA Tuxedo ATMI アプリケーションの不可欠の部分になっています。フィールド化バッファは、IDL では新しい基本型としてサポートされます。フィールド化バッファは、16 ビット・バッファについてはキーワード FBFR で、32 ビット・バッファはキーワード FBFR32 で示され、ポインタとして定義される必要があります (たとえば FBFR * または FBFR32 *)。フィールド化バッファは typedef で基本型として定義することはできません。フィールド化バッファは、構造体のフィールドで、さらにパラメータとして使用できます。フィールド化バッファは配列またはポインタ (完全ポインタまたは参照ポインタのいずれか) の基本型として使用できますが、フィールド化バッファの整合配列と可変配列はサポートされません。
- OSF IDL コンパイラでは、AES 仕様または X/OPEN RPC 仕様にドキュメント化されていない 制約がいくつか存在します。BEA Tuxedo の IDL コンパイラではこれらの制約が強制されます。
  - [transmit_as()] で使用する転送型には [represent_as] 属性を設定できません。
  - 共用体アームを [ref] ポインタにしたり、共用体アームに [ref] ポインタを含めることはできません。
  - 整合配列と可変配列の両方あるいはいずれか一方が構造体に存在する場合は、配列サイズの属性変数はポインタにはできません。つまりポインタではなく、構造体内の整数要素である必要があります。

移植性に制限を加える可能性がある拡張

X/OPEN RPC 仕様に対して BEA Tuxedo ATMI の 4 つの拡張機能が追加されており、これらは仕様をより C 言語に近づける一方で、OSF DCE IDL コンパイラではサポートされないために IDL ファイルの移植性に制限を加えることになります。

- ANSI C と同様に、文字列連結がサポートされます。つまり、

  ```
  const char *str = "abc" "def";
  ```

  は、次と同じように扱われます。

  ```
  const char *str = "abcdef";
  ```

- エスケープ文字を使用した改行を文字列定数内で用いることができます。つまり、

  ```
  const char *str = "abc¥
   def";
  ```

  は、次と同じように扱われます。

  ```
  const char *str = "abcdef";
  ```

- 列挙値は共用体ケースでも使用でき、整数として扱われます。C と同様に自動的に変換されます。
- 各 <*union_case_label*> の型を <*switch_type_spec*> によって指定する必要があるという制約は設けていません。代わりに、C の switch ステートメントの case ステートメントで行われるような型の強制変換が行われます。

サポートされていない機能

以下の 7 つの機能は tidl コンパイラではサポートされていません。

- 移行属性 [v1_struct]、[v1_enum]、[v1_string]、[v1_array] は認識されますが、サポートされていません。これらの属性は OSF IDL 仕様にはありますが、X/OPEN 仕様にはありません。
- OSF/DCE と同様に [local] モードでのみ OSF/DCE ドキュメントに定義された関数ポインタがサポートされます。
- クライアントとサーバ間ではインターフェイスのマイナー・バージョンまで完全に一致する必要があります。X/OPEN RPC 仕様では、サーバのマイナー・バージョンがクライアントのマイナー・バージョンよりも大きいか等しい場合も許可しています。
- 32 ビット長のマシンでは、整数リテラル値は -2**31 から 2**31 の範囲に制限されています。2**31+1 から 2**32-1 の範囲の unsigned long 整数値はサポートされません。これは DCE IDL コンパイラの動作によるものと思われます。
- コンテキスト・ハンドルは [local] モードだけでサポートされます。状態をまたがるオペレーションを管理するためにコンテキスト・ハンドルを使用して、インターフェイスを記述することはできません。
- [out-of-line] ACS 属性は無視されます。この機能は、たとえば OSF IDL コンパイラを使用する異なるインプリメンテーション間での相互運用をサポートするという点では定義されていません。
- [heap] ACS 属性は無視されます。

IDL コンパイラ tidl の使用

IDL コンパイラ用のインターフェイスは、X/OPEN 仕様では指定されていません。

DCE アプリケーションの移植性を考慮し、BEA Tuxedo ATMI の IDL コンパイラのインターフェイスは DCE IDL コンパイラと同様ですが、次のような例外があります。

- コマンド名は `idl` の代わりに `tidl` を用います。同じ環境に両方のコンパイラが存在する場合でも、コンパイラの識別がアプリケーションから容易に行えます。
- `-bug` オプションは、旧バージョンのソフトウェアとの相互運用のために、誤動作を起こさせます。このオプションは無効です。`-no_bug` オプションも無効です。
- `-space_opt` オプションは無視されます。このオプションは、コード領域の最適化を行ないます。領域は常に最適化されます。
- 新しいオプション `-use_const` がサポートされています。`-use_const` は、定数定義用の `#define` ステートメントの代わりに ANSI C `const` ステートメントを生成します。このオプションにより、IDL ファイル内で定義された定数が C プリプロセッサ定義を使用するファイル内の別の定数名と競合するというやっかいな問題を回避できます。このオプションにより、C の定数として定義されるときに、これらの定数は別の名前空間に正しく配置されます。本機能を使うと、IDL ファイルの移植性が制限されます。
- 特に指定しない限り、`/lib/cpp`、`/usr/ccs/lib/cpp`、または `/usr/lib/cpp` のいずれかのコマンドを使用して、入力 IDL ファイルと入力 ACS ファイルのプリプロセッサ処理を行ないます。3 つのプリプロセッサのうち最初に見つかったものを使用します。

デフォルトでは、IDL コンパイラは入力 IDL ファイルを受け取り、クライアントとサーバ・スタブのオブジェクト・ファイルを生成します。`-keep c_source` オプションは C ソース・ファイルだけを生成し、`-keep all` オプションは C ソース・ファイルとオブジェクト・ファイルの両方を保持します。付録 A の 1 ページ「アプリケーション例」で示したサンプル RPC アプリケーションでは、`-keep object` オプションを使用してオブジェクト・ファイルを生成しています。

デフォルトでは、`tidl` は最大 50 個までのエラーを表示します。エラー数が 50 個を超えている場合に、すべてのエラーを表示するときは、`-error all` オプションを指定します。エラー出力は `stderr` に送られます。

その他使用可能なオプションの詳細については、『BEA Tuxedo コマンド・リファレンス』の `tidl(1)` を参照してください。

3 RPC クライアント / サーバ・プログラムの作成

ここでは、次の内容について説明します。

- リモート処理への対応
- 返されるステータスと例外の処理
- スタブ・サポート関数の使用
- RPC ヘッダ・ファイルの使用
- コードの移植性
- ATMI との対話
- TX との対話

注記 クライアントおよびサーバのソース・ファイルのサンプルについては、付録 A の 1 ページ「アプリケーション例」を参照してください。

リモート処理への対応

TxRPC の目的は、あるアドレス空間にあるクライアントと別のアドレス空間にあるサーバとの間で、透過的なプロシージャ・コールを提供することです。クライアントとサーバは、別のマシンに存在する場合もあります。ただし、クライアントとサーバのアドレス空間が異なるので、次のような点に注意する必要があります。

- クライアントとサーバが異なるアドレス空間に存在するので、また別のマシンに存在する場合もあるので、メモリの共有を想定することはできません。プログラムの状態 (例 : オープン・ファイル記述子) やグローバル変数をクライアントとサーバ間で共有できません。必要な状態情報は、クライアントからサーバに渡す必要があり、その後の呼び出しでサーバからクライアントに返す必要があります。

- 作業をクライアントとサーバ間で分割すると、ソフトウェアのモジュール性が高まり、必要なリソースの近くで作業が行えるようになります。ただし、通信の問題やクライアントとサーバのいずれかが個別に利用できなくなるといった分散処理に関する問題を処理する複雑さが増加することも意味します。分散処理に伴う複雑さに起因するエラーには、ローカル・プロシージャ・コール用インターフェイスで発生するエラーとは異なった処理が必要です。通信やリモート・プロセスに関係したエラーの処理については、次のトピックで扱います。

返されるステータスと例外の処理

X/OPEN RPC 仕様では、アプリケーション以外のエラーは状態パラメータまたは状態復帰で返されます。RPC サーバに障害があった場合には fault_status 値が返され、通信に障害があった場合には comm_status 値が返されます。状態復帰を指定するには、IDL ファイルでオペレーションの戻り値または error_status_t 型の [out] パラメータを定義します。さらに ACF ファイルで、そのオペレーションまたはパラメータが [fault_status] または [comm_status] のいずれか、あるいは両方の属性を持つことを宣言します。

たとえば、IDL ファイル内で次のようにオペレーションを定義します。

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

対応する ACF ファイル内での定義は次のようになります。

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

サーバからのエラーは、オペレーションの戻り値に、通信エラーは 2 番目のパラメータに返されます。クライアント・コードでは、以下のようにエラーを処理します。

```
if (op(&parm1, &commstat) != 0 || commstat != 0) /* エラーの処理 */
```

状態復帰を使用する利点は、エラー発生個所で対処が行え、きめ細かなエラー回復処理を実行できることです。

状態復帰の欠点は、ローカル版の関数には必要のないパラメータをリモート関数に付加することです。さらに、きめ細かなエラー回復処理は冗長になりがちで、エラーを引き起こす傾向があります (たとえば、場合分けに見落としが生じるなど)。

DCE は第 2 のメカニズムとして例外処理を定義しています。この例外処理は C++ のものと類似しています。

C や C++ のアプリケーション・コードを TRY、CATCH、CATCH_ALL、および ENDTRY ステートメントを用いて、例外が生じる可能性のあるブロックに区切ります。TRY はブロックの開始を示します。CATCH は、特定の例外用の処理ブロックを示します。CATCH_ALL は、対応する CATCH ステートメントを持たない例外を処理するために使われます。ENDTRY でブロックを終了します。例外処理が低いレベルでは行なえず、RERAISE ステートメントを用いて高いレベルのブロックで例外処理を行う場合、TRY ブロックはネストされます。例外処理ブロックの外部で例外が生じた場合は、プログラムはログにメッセージを記録して終了します。例外処理マクロの詳細と使用例については、『BEA Tuxedo C 言語リファレンス』の TRY(3c) を参照してください。

RPC コールに対して通信とサーバで生成される例外のほかに、さらに低いレベルの例外、特に、オペレーティング・システムのシグナルも生成されます。このような例外の詳細については、『BEA Tuxedo C 言語リファレンス』の TRY(3c) を参照してください。

スタブ・サポート関数の使用

X/OPEN 仕様では、100 を越える膨大な数のランタイム・サポート関数が定義されています。これらの関数を、X/OPEN TxRPC IDL-only 環境ですべてサポートする必要はありません。これらの関数の大部分は、ATMI のクライアントやサーバに対して透過的に行われるバインディングと管理に関連したものです。

アプリケーションの移植性に影響を及ぼす事柄の 1 つに、スタブ用の入出力パラメータと戻り値に割り当てられるメモリの管理があります。Stub Memory Management ルーチンは、スレッドを扱う 2 つの関数を除き TxRPC ランタイムでサポートされています。状態を返す関数には以下のものがあります。

- 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

同等の例外を返す関数には以下のものがあります。

- 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

これらの関数の詳細については、『BEA Tuxedo C 言語リファレンス』を参照してください。

ランタイム関数は、libtrpc ファイルに含まれます。RPC クライアントとサーバの作成方法については、次のトピックで解説します。

メモリ管理を効果的に行う方法を次に示します。

- デフォルトでは、ATMI クライアントはクライアント・スタブを呼び出すときに malloc と free を使います。オペレーションの戻り値での暗黙の [out] ポインタを含めて、[out] ポインタに割り当てられた領域以外の全領域はクライアント・スタブからの復帰時に解放されます。[out] ポインタの解放を容易にするために、rpc_ss_enable_allocate() を呼び出します。さらに、RPC を呼び出す前に、rpc_ss_set_client_alloc_free() を呼び出して、alloc/free をそれぞれ rpc_ss_alloc()/rpc_ss_free() に設定します。その後、関数 rpc_ss_disable_allocate() を用いて割り当てたメモリをすべて解放します。たとえば、クライアント・スタブから返される領域の解放を簡素にするには次のように使用します。

  ```
  rpc_ss_set_client_alloc_free(rpc_ss_allocate, rpc_ss_free);
  ptr = remote_call_returns_pointer();
  /* 返されたポインタをここで使用 */
  ...
  rpc_ss_disable_allocate(); /* ptr を解放 */
  ```

- アプリケーション操作を呼び出す ATMI サーバ・スタブの実行時には、サーバ・スタブ内では rpc_ss_allocate を用いたメモリ割り当てが常に有効です。ACF ファイル内の属性 [enable_allocate] は無効です。クライアントに応答を返す前に全メモリはサーバ内で解放されます。DCE では、メモリ割り当てが有効になるのは、[ptr] フィールドまたはパラメータが存在する場合と、プログラマが [enable_allocate] を明示的に指定したときだけです。

- サーバ・スタブがアプリケーション・オペレーションを呼びだし、さらにそのアプリケーション・オペレーションがクライアント・スタブを呼び出す場合、つまり RPC を呼び出すことで、サーバがクライアントとして動作するときは、関数 rpc_ss_set_client_alloc_free() を呼び出してメモリ割り当ての準備をする必要があります。その結果、オペレーションからの復帰時に割り当てられた領域がすべて開放されるようになります。次の呼び出しを行います。

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

- 関数 rpc_ss_allocate() または関数 rpc_sm_allocate() を呼び出すときには、設定しているポインタのデータ型に一致するように出力をキャストするようにしてください。次に例を示します。

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

RPC ヘッダ・ファイルの使用

DCE/RPC と Tuxedo/TxRPC 両方からのスタブが同一環境で確実にコンパイルできるようにするため、Tuxedo/TxRPC のインプリメンテーションで用いるヘッダ・ファイル名は異なるものにします。これらのヘッダ・ファイルは IDL コンパイラが生成するインターフェイス・ヘッダ・ファイルに自動的にインクルードされるので、アプリケーション・プログラマには影響を与えません。ただし、型や関数の定義方法を調べるために、アプリケーション・プログラムでこれらのヘッダ・ファイルを表示することは可能です。以下に、RPC における新しいヘッダ・ファイル名のリストを示します。

- dce/nbase.h、dce/nbase.idl―rpc/tbase.h および rpc/tbase.idl を名称変更したヘッダ・ファイル。定義済みの型、error_status_t、ISO_LATIN_1、ISO_MULTI_LINGUAL、および ISO_UCS の宣言を含みます。
- dce/idlbase.h―rpc/tidlbase.h を名称変更したヘッダ・ファイル。IDL の仕様に準拠した基本型 (例：idl_boolean、idl_long_int) を含みます。さらにスタブ関数用の関数プロトタイプも含みます。
- dce/pthread_exc.h―rpc/texc.h を名称変更したヘッダ・ファイル。TRY/CATCH 例外処理マクロを含みます。
- dce/rpcsts.h―rpc/trpcsts.h を名称変更したヘッダ・ファイル。RPC インターフェイス用の例外値と状態値の定義を含みます。

ヘッダ・ファイルは $TUXDIR/include/rpc ディレクトリにあります。デフォルトでは、Tuxedo/TxRPC IDL コンパイラは「システム IDL ディレクトリ」として $TUXDIR/include を検索します。

コードの移植性

IDL コンパイラからの出力は、多数の環境でコンパイルできるように作成されます。コンパイル作業の説明は次の章を参照してください。ただし、環境によっては動作に支障が起きる構成もあります。既知の問題点を以下に示します。

ANSI に準拠しない旧形式の C 言語を使用してコンパイルする場合、「配列へのポインタ」は許可されません。例を示します。

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

このため、移植を考慮した場合、「配列へのポインタ」をパラメータとして、オペレーションに渡すことが難しくなります。

文字列属性がマルチ・バイト構造体に適用されるところに文字列の配列を使用すると、コンパイラが構造体に文字を埋め込むときに期待する結果が得られないことがあります。これは通常は起こり得ません。ほとんどのコンパイラは文字フィールドのみを含む構造体には文字の埋め込みは行いません。しかし、この現象は少なくとも一度確認されています。

デフォルトでは、定数値のインプリメントは各定数に対して `#define` を作成することで行われます。このため、定数に用いる名前は、IDL ファイル内やインポートされたすべての IDL ファイル内のほかのすべての名前と同じ名前にしてはいけません。ANSI C 環境でこの問題を回避するには、`tidl` コンパイラの TxRPC 固有のオプション `_use_const` を使用します。このオプションを使うと、`#define` 定義の代わりに、`const` 宣言が作成されます。定数値はクライアント・スタブとサーバ・スタブ内で宣言されます。ヘッダ・ファイルをインクルードするほかのすべてのソース・ファイルでは、単に `extern const` 宣言が使用されます。こうしないと、クライアント・スタブとサーバ・スタブを同一の実行ファイルにコンパイルできないという制限を生じるか、二重定義エラーが発生することに注意してください。

C++ 環境では、次のような制限があります。

- `typedef` の名前と、構造体や共用体のタグに用いる名前を同一にしないでください。`typedef` の名前が `struct` 名や `union` 名と一致している場合は、この制限はありません。

  ```
  struct t1 {
    long s1;
  };
  ```

```
typedef struct t1 t1; /* 正常 */
typedef long t1; /* エラー */
```

- 構造体や共用体タグの宣言をほかの構造体や共用体の内部で行い、それを外部で参照することはできません。

```
struct t1 {
  struct t2 {
   long s2;
} s1;
} t1;
typedef struct t3 {
struct t2 s3; /*t2 は未定義エラー */
} t3;
```

- 一部のコンパイラでは警告メッセージが生成される場合があります。例を次に示します。
  - 宣言された自動変数が使われないことを示す警告
  - 以下の場合のように変数が sizeof() 関数で参照されているときなど、変数が設定される前に使用されていることを示す警告。

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

クライアントとサーバのアプリケーション・ソフトウェアをコーディングしているときは、IDL コンパイラが作成したデータ型、すなわち rpc/tidlbase.h 内で定義されるデータ型を使う必要があります。(rpc/tidlbase.h 内での定義とは、次の表に「発行されるマクロ」として記載されているものです。) たとえば、idl_long_int の代わりに long を使うと、データ型はプラットフォームによっては 32 ビットであったり 64 ビットであったりします。一方、idl_long_int は全プラットフォームで 32 ビットです。作成されるデータ型の一覧を表 3-1 に示します。

表 3-1 生成されるデータ型

IDL の型	サイズ	発行されるマクロ	C の型
boolean	8 ビット	idl_boolean	unsigned char
char	8 ビット	idl_char	unsigned char
byte	8 ビット	idl_byte	unsigned char
small	8 ビット	idl_small_int	char
short	16 ビット	idl_short_int	short
long	32 ビット	idl_long_int	32 ビット長のマシンの場合 : long 64 ビット長のマシンの場合 :int

IDL の型	サイズ	発行されるマクロ	C の型

hyper	64 ビット	idl_hyper_int	32 ビット長のマシンの場合 :
			ビッグ・エンディアン
			struct
			{
			long high;
			unsigned long low;
			}
			リトル・エンディアン
			struct
			{
			unsigned long low;
			long high;
			}
			64 ビット長のマシンの場合 :
			long

unsigned small	8 ビット	idl_usmall_int	unsigned char
unsigned short	16 ビット	idl_ushort_int	short
unsigned long	32 ビット	idl_ulong_int	32 ビット長のマシンの場合 : long 64 ビット長のマシンの場合 : int

unsigned hyper	64 ビット	idl_uhyper_int	32 ビット長のマシンの場合
			ビッグ・エンディアン
			struct
			{
			unsigned long high;
			unsigned long low;
			}
			リトル・エンディアン
			struct
			{
			unsigned long low;
			unsigned long high;
			}
			64 ビット長のマシンの場合 :
			unsigned long

IDL の型	サイズ	発行されるマクロ	C の型
float	32 ビット	idl_short_float	float
double	64 ビット	idl_long_float	double
void *	ポインタ	idl_void_p_t	void *
handle_t	ポインタ	handle_t	handle_t

C の場合と同様、IDL でも複数のクラスの識別子があります。スコープや名前空間などのクラス内では、名前を一意にする必要があります。

- 定数、typedef、オペレーション、列挙用メンバー名は、1 つの名前空間内にあります。
- 構造体，共用体，および列挙用のタグは、別の名前空間内におかれます。
- 構造体や共用体のメンバー名で同じレベルのものは、定義が行われた構造体や共用体の中で一意にする必要があります。
- パラメータ名は、定義が行われたオペレーション・プロトタイプ内では一意にする必要があります。

タグを持たず、typedef の一部として定義されていない匿名の構造体または共用体は、オペレーションの戻り値やパラメータとして使用できないことに注意してください。

ATMI との対話

TxRPC 実行プログラムは BEA Tuxedo システムを使って RPC 通信を行います。その他の BEA Tuxedo のインターフェイスや通信メカニズムは、RPC 呼び出しを使用している同一のクライアントおよびサーバ内で使用できます。このため、単一のクライアントが要求 / 応答呼び出し (例 :tpcall(3c)、tpacall(3c)、tpgetrply(3c))、会話型呼び出し (tpconnect(3c)、tpsend(3c)、tprecv(3c)、tpdiscon(3c)) を行い、安定キューにアクセス (tpenqueue(3c) および tpdequeue(3c)) することが可能です。クライアントが、BEA Tuxedo ソフトウェアを最初に呼び出すときに、自動的にアプリケーションに参加します。クライアントが最初に行う呼び出しは、RPC 呼び出し、その他の通信呼び出し、ATMI 呼び出し (例 : バッファ割り当て、任意通知など) です。ただし、アプリケーションがセキュリティをオンにした状態で実行していたり、あるいはクライアントが特定のリソース・マネージャ・グループの一部として稼働する必要がある場合は、関数 tpinit(3c) を明示的に呼び出してアプリケーションに参加しなければなりません。明示的に設定できるオプションの詳細と一覧については、『BEA Tuxedo C 言語リファレンス』の tpinit(3c) を参照してください。アプリケーションが BEA Tuxedo システムを使用した作業を完了する場合には、関数 tpterm(3c) を明示的に呼び出してアプリケーションを終了し、関連のあるリソースをすべて解放する必要があります。ワークステーション以外のネイティブなクライアントに対してこの明示的な呼び出しが行われなかった場合、モニタがこれを検知し、userlog(3c) に警告を出力し、リソースを解放します。クライアントがワークステーションの場合は、リソースは解放されず、結果的にワークステーションのリスナまたはハンドラがリソースを使い果たし、新たなクライアントを受け付けることができなくなります。

クライアントと同様に、サーバはクライアントのロール内の任意の通信パラダイムを使用できます。ただし、同一サーバ内では会話型サービスと RPC サービスの両方を提供 (宣言) することはできません。後で説明するように、RPC サーバは非会話型として指定する必要があります。ATMI 要求 / 応答と RPC サービスを同一サーバ内で混用することは可能ですが、お勧めできません。その他の制限として、RPC オペレーションは関数 tpreturn(3c) や関数 tpforward(3c) を呼び出すことはできません。RPC オペレーションは、ローカルに呼び出された場合と同様の復帰動作で復帰する必要があります。RPC オペレーションから関数 tpreturn(3c) や関数 tpforward(3c) の呼び出しを試みると、その呼び出しはインターセプトされ、クライアントにエラーが返されます (例外 rpc_x_fault_unspec または状態 rpc_s_fault_unspec)。

サーバでは利用できますが、クライアントでは利用できない関数が 2 つあります。関数 tpsvrinit(3c) と関数 tpsvrdone(3c) がこれにあたります。それぞれサーバの起動時とシャットダウン時に呼び出されます。サーバは、TxRPC オペレーション要求を受け取る前に、tx_open(3c) を呼び出す必要があるので、tpsvrinit() で tx_open を呼び出すことをお勧めします。デフォルトの tpsvrinit() 関数は関数 tx_open() を呼び出しています。

TX との対話

TX 関数は、トランザクションの境界用のインターフェイスを提供します。関数 tx_begin(3c) と、tx_commit(3c) または tx_rollback(3c) は、通信を含むトランザクション内のすべての作業をカプセル化します。その他のプリミティブとして、トランザクションのタイムアウト設定、連鎖または非連鎖としてのトランザクションの宣言、トランザクション情報の取得も提供されています。以上についての詳細は、X/OPEN TX 仕様に記載され、また X/OPEN TxRPC 仕様にも解説されています。X/OPEN TxRPC 仕様には、TX と RPC の相互動作についての解説があります。その要点を次に述べます。

- インターフェイスまたはオペレーションは、属性 [transaction_optional] を持つことができます。この属性は、あるトランザクション内で RPC を呼び出した場合、呼び出されるオペレーション内で行われる作業は、そのトランザクションの一部になることを示します。
- インターフェイスまたはオペレーションは、属性 [transaction_mandatory] を持つことができます。この属性は、RPC 呼び出しはトランザクション内で行う必要があることを示します。そうでなければ例外 txrpc_x_not_in_transaction が返されます。
- 上記 2 つの属性がいずれも指定されない場合には、呼び出されるオペレーション内での作業は、呼び出し側でアクティブになるトランザクションの一部にはなりません。
- サーバ内で TxRPC オペレーションが呼び出され、関数 tx_open(3c) が呼び出されていなければ、例外 txrpc_x_no_tx_open_done が呼び出し側に返されます。
- TxRPC では、オペレーションが関数 tx_rollback(3c) を呼び出すことを認めています。同関数を呼び出すことで、トランザクションがロールバックのみとしてマークされ、トランザクションとして行われた作業は、すべてロールバックされます。この場合、アプリケーションも呼び出し側にアプリケーション・レベルのエラーを返し、トランザクションがロールバックされることを示すことをお勧めします。

TxRPC 仕様で定義された IDL に対するその他の変更と制限については、IDL に関する項で既に説明しました。

4 RPC クライアント / サーバ・プログラムの構築

ここでは、次の内容について説明します。

- 予備知識
- RPC サーバの構築方法
- RPC クライアントの構築方法
- Windows ワークステーション RPC クライアントの構築方法
- C++ の使い方
- DCE/RPC との相互運用

予備知識

TxRPC のプログラマは、C コンパイレーション・システムと、BEA Tuxedo ATMI でのクライアントとサーバの構築方法に知識があることを前提としています。BEA Tuxedo ATMI クライアントおよびサーバの構築については、『C 言語を使用した BEA Tuxedo アプリケーションのプログラミング』、『COBOL を使用した BEA Tuxedo アプリケーションのプログラミング』、および『FML を使用した BEA Tuxedo アプリケーションのプログラミング』を参照してください。ワークステーション・クライアントの構築については、『BEA Tuxedo Workstation コンポーネント』を参照してください。

RPC サーバの構築方法

RPC サーバは、ATMI 要求 / 応答サーバの場合とほぼ同じように構築され構成されます。実際、RPC と要求 / 応答サーバのサービス名前空間は同一です。しかし、RPC サービス用に宣言される名前は異なります。要求 / 応答サーバ用では、サービス名はプロシージャにマップされます。RPC サーバでは、サービス名は IDL インターフェイス名にマップされます。宣言された RPC サービスは *<interface>*v*<major>*_*<minor>* となります。ここで、*<interface>* はインターフェイス名、*<major>* と *<minor>* はそれぞれメジャー・バージョン番号とマイナー・バージョン番号です。バージョン番号は、インターフェイス定義で指定されたものか、あるいはデフォルトでの設定値 0.0 のいずれかをとります。サービス名の長さ制限は 15 文字です。このため、インターフェイス名の長さは、13 からメジャーとマイナーのバージョン番号の桁数を引いたものに制限されます。これは、BEA Tuxedo システムで名前が処理される方法によるもので、メジャー・バージョン番号とマイナー・バージョン番号が正確に一致する必要があることを意味しています。宣言されるのはインターフェイスであって、個別のオペレーションではないことに注意してください (DCE/RPC と同様)。インターフェイス内の適切なオペレーションが呼び出されるようにサーバ・スタブが自動的に調整を行います。

RPC サーバは、`buildserver(1)` コマンドを使って構築されます。コンパイル時には、`-s` オプションを用いてサービス (インターフェイス) 名を指定することをお勧めします。その後、`-A` オプションを用いてサーバを起動すると、指定したサービスを自動的に宣言することができます。この方法は、付録 A の 1 ページ「アプリケーション例」のアプリケーション例で用いられています。

`buildserver(1)` コマンドは、BEA Tuxedo ライブラリを自動的にリンクしますが、RPC ランタイムのリンクは明示的に行う必要があります。このリンクは、`buildserver` 行のアプリケーション・ファイルの後に、`-f -ltrpc` オプションを指定することで行われます。通常は、`tidl(1)` コマンドの出力はサーバ・スタブ・オブジェクト・ファイルです。このファイルは `buildserver` コマンドに直接受け渡されます。サーバ・スタブと、オペレーションを実装するアプリケーション・ソース、オブジェクト、ライブラリ・ファイルは、ランタイム・ライブラリよりも前に、`-f` オプションを使用して指定する必要があることに注意してください。付録 A の 1 ページ「アプリケーション例」の makefile 例 `rpcsimp.mk` を参照してください。

RPC クライアントの構築方法

ネイティブ RPC クライアントは、buildclient(1) コマンドを使って構築されます。このコマンドは、BEA Tuxedo ライブラリを自動的にリンクします。RPC ランタイムは明示的にリンクする必要があります。このリンクは、buildclient コマンド行のアプリケーション・ファイルの後に、-f -ltrpc オプションを指定することで行われます。通常は、tidl(1) コマンドの出力はクライアント・スタブ・オブジェクト・ファイルです。このファイルは buildclient コマンドに直接受け渡されます。クライアント・スタブとリモート・プロシージャ・コールを実行するアプリケーション・ソース、オブジェクト、ライブラリ・ファイルは、ランタイム・ライブラリよりも前に、-f オプションを使用して指定する必要があることに注意してください。詳細については付録 A の 1 ページ「アプリケーション例」の rpcsimp.mk を参照してください。

UNIX ワークステーション・クライアントを構築するには、ネイティブ・ライブラリの代わりに ワークステーション・ライブラリをリンクするように単純に -w オプションを buildclient(1) コマンド行に追加します。

Windows ワークステーション RPC クライアントの構築方法

Windows 用のクライアント・スタブのコンパイルには、コンパイル・オプションとして -D_TM_WIN 定義が必要になります。このオプションにより、該当する TxRPC 用の関数プロトタイプと BEA Tuxedo ATMI のランタイム関数が使用されます。DLL のテキスト・セグメントとデータ・セグメントが、DLL を呼び出しているコードとは異なるように、特別にコンパイルする必要があります。これは、クライアント・スタブ・ソースでは同じになります。ヘッダ・ファイルやスタブの作成は C プリプロセッサの定義を用いて自動的に行われ、宣言が容易に変更できるよう配慮されています。_TMF ("far" 用) の定義は、ヘッダ・ファイル内のすべてのポインタ定義の前で行われ、_TM_WIN が定義されていれば、_TMF は自動的に "_far" 型として定義されます。

ほとんどの場合、標準ライブラリを使えば buildclient(1) コマンドでクライアントをリンクできます。使用されるライブラリは wtrpc.lib です。

付録 A に示した例は、クライアント・スタブを使ってダイナミック・リンク・ライブラリ (DLL) を作成する方法を示したものです。この手法は、DLL (アプリケーション・コードが静的にリンクされることが無い) を使用する必要のあるビジュアル・アプリケーション・ビルダを使用する場合には一般的です。Windows 関数は、_pascal 呼び出し規約に従って宣言されることが従来からの方法でした。ヘッダ・ファイルやスタブの作成は C プリプロセッサの定義を用いて自動的に行われ、宣言が容易に変更できるよう配慮されています。_TMX ("eXport" 用) は、宣言されたすべての関数の前におかれます。デフォルトでは、この宣言は何も定義しません。DLL に含まれるスタブをコンパイルするときは、_TMX を _far _pascal として定義する必要があります。さらに、DLL に含まれるファイルは、ラージ・メモリ・モデルを用いてコンパイルする必要があります。_pascal を使用すると、関数名は自動的にライブラリ内で大文字に変換されます。このため、-port case オプションをオン状態にしておくと、2 つの名前の違いは大文字 / 小文字の区別だけなのかを調べる追加の検証を行え便利です。

Windows DLL を構築する例が、付録 A の 1 ページ「アプリケーション例」に示してあります。

注記 TxRPC クライアントが windows.h をインクルードしている場合、uuid_t の定義が重複するためにコンパイル・エラーが起きることがあります。windows.h は、すでにインクルードされているのでそれをインクルードしないようにするか、アプリケーションの別のファイルでそれをインクルードする必要があります。

C++ の使い方

クライアントとサーバは、C と C++ のいずれを用いても構築できます。ヘッダ・ファイルと作成されたスタブ・ソース・ファイルは、すべてのスタブ・サポート関数と作成したオペレーションが C++ と C 間で完全に相互運用できる方法で定義されています。これらの関数やオペレーションは C リンケージを使用して、つまり extern "C" として宣言されているので、名前のアングリングはオフになります。

スタブ・オブジェクト・ファイルを C++ を用いて作成するには、tidl(1) の -cc_cmd オプションに CC -c を指定します。CC コマンドを用いると、クライアントとサーバのプログラムのコンパイルとリンクができます。そのためには、buildclient(1) と buildserver(1) を実行する前に、CC 環境変数を設定してエクスポートします。次に例を示します。

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

Windows 環境では C++ によるコンパイルは通常、異なるコマンド名を用いるのではなく、コンパイル・コマンド・ラインのフラグまたは設定オプションで行えます。C++ でコンパイルするための適切なオプションを使用してください。

DCE/RPC との相互運用

BEA Tuxedo の TxRPC コンパイラは OSF/DCE と同一の IDL インターフェイスを使用しますが、生成されたスタブでは同一のプロトコルは使用されません。したがって、BEA Tuxedo の TxRPC スタブは、DCE IDL コンパイラが生成するスタブとは直接通信できません。

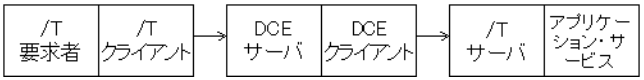
ただし、DCE/RPC と BEA Tuxedo TxRPC 間では以下の相互動作を行うことができます。

- DCE と BEA Tuxedo TxRPC からのクライアント側スタブは、両方とも同一プログラム (クライアントとサーバのいずれでも) から呼び出すことができます。
- BEA Tuxedo ATMI サーバ・スタブは、BEA Tuxedo TxRPC クライアント・スタブと同様に DCE クライアント・スタブを呼び出すアプリケーション・コードを呼び出せます。
- DCE サーバ (マネージャ) は、BEA Tuxedo TxRPC クライアント・スタブを呼び出すアプリケーション・コードを呼び出すことができます。

以下のセクションでは、BEA Tuxedo TxRPC と OSF/DCE との間で行うことのできる相互作用について説明します。各場合に、要求の発行元を要求者と呼びます。要求者の発行元は、実際は別のサービスの要求を行う DCE または BEA Tuxedo ATMI サービスである場合もあるので、要求者という用語が " クライアント " の代わりとして使用されます。「クライアント」と「サーバ」という用語は、IDL コンパイラによって生成されるクライアントおよびサーバ・スタブを指しています。IDL コンパイラは、DCE `idl`(1) または BEA Tuxedo `tidl(1)` のいずれかを指します。つまりこれらの用語は、DCE と TxRPC 間での用語の統一を計るために使用します。結局、「アプリケーション・サービス」という用語は、リモートに呼び出されるプロシージャを実装するアプリケーション・コードに使用されます。一般的には、起動ソフトウェアが DCE または BEA Tuxedo のいずれによって生成されたサーバ・スタブであっても無関係です。

BEA Tuxedo ゲートウェイを使用した DCE サービスに対する BEA Tuxedo 要求者

図 4-1 BEA Tuxedo ゲートウェイを使用した DCE サービスに対する BEA Tuxedo 要求者



最初の方法では、BEA Tuxedo ATMI クライアント・スタブが TxRPC を使用して BEA Tuxedo ATMI サーバ・スタブを呼び出す「ゲートウェイ」を使用します。サーバ・スタブは、（アプリケーション・サービスの代わりに）リンクされた DCE クライアントを持ち、この DCE クライアントが DCE RPC を使用して DCE サービスを呼び出します。この方法の長所は、クライアント・プラットフォームで DCE を必要としない点です。実際に、すべてのゲートウェイを実行している 1 台のマシンを除いて BEA Tuxedo を実行するマシンのセットおよび DCE を実行するマシンのセットを切り離すことができます。この結果、BEA Tuxedo と DCE 間でサービスを移動する機能を使用して、移行パスを提供します。この方法を実施するアプリケーション例については、付録 B の 1 ページ「DCE ゲートウェイ・アプリケーション」で説明します。

この構成では、要求者は通常の BEA Tuxedo ATMI のクライアントまたはサーバとして構築されます。同様に、サーバも通常の DCE サーバとして構築されます。もう一つの手順は、TxRPC サーバ・スタブを使う BEA Tuxedo ATMI サーバおよび DCE/RPC クライアント・スタブを使う DCE クライアントとして機能するゲートウェイ・プロセスを構築することです。

2 つの IDL コンパイラを実行し、得られたファイルをリンクするプロセスは、リンクされている DCE を使って BEA Tuxedo ATMI サーバを構築する `blds_dce(1)` のコマンドの利用によって簡素化されています。

`blds_dce` の使用法は次のとおりです。

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

このコマンドは入力として 1 つ以上の IDL ファイルを入力するため、ゲートウェイは 1 つ以上のインターフェイスを処理できます。これらのファイルごとにサーバ・スタブを生成する場合は `tidl` を実行し、クライアント・スタブを生成する場合は `idl` を実行します。

このコマンドは各種の DCE 環境を認識し、コンパイルとリンクに必要なコンパイル・フラグと DCE ライブラリを提供します。新たな環境で開発するのであれば、コマンドを変更して現在の環境に合ったオプションとライブラリの追加が必要になります。

このコマンドは、常に rpc_ss_allocate(3c) と rpc_ss_free(3c) を使用してメモリ割り当てを行う方法 (-DTMDCEGW を定義した状態) でソース・ファイルをコンパイルします。詳細については、『BEA Tuxedo C 言語リファレンス』を参照してください。これにより BEA Tuxedo ATMI サーバから復帰する際には必ずメモリが解放されるようになります。-DTMDCEGW を使用すると、BEA Tuxedo TxRPC ヘッダ・ファイルの代わりに DCE ヘッダ・ファイルが組み込まれます。

buildserver(1) を用いて BEA Tuxedo ATMI サーバを作成するには、アプリケーション・ファイルを指定して (-f と -l オプションを使用します)、IDL 出力オブジェクト・ファイルをコンパイルします。実行可能なサーバ名は -o オプションで指定できます。

この構成を実行する場合、まず DCE サーバがバックグラウンドで開始され、次に DCE ゲートウェイを含む BEA Tuxedo のコンフィギュレーションが起動し、要求者が実行されます。DCE ゲートウェイはシングル・スレッドで実行されるので、同時に実行したいサービスの数と同数のゲートウェイ・サービスを構成して起動する必要があることに注意してください。

このゲートウェイを構築する際に考慮すべき選択事項がいくつかあります。

DCE ログイン・コンテキストの設定

まず DCE クライアントでは通常、プロセスが DCE プリンシパルとして実行します。ログイン・コンテキストを取得する方法は 2 つあります。1 つの方法としては DCE に“ログイン”します。ある種の環境では、単にオペレーティング・システムにログインするだけでログイン・コンテキストを取得できます。多くの環境では、dce_login の実行が必要です。ローカル・マシンで BEA Tuxedo ATMI サーバを起動すると、初めに dce_login の実行が、そして次に tmboot(1) の実行が可能となり、起動したサーバはログイン・コンテキストを継承することになります。tlisten(1) を使用して間接的に実行されるリモート・マシンでサーバを起動する場合は、tlisten を開始する前に dce_login を実行する必要があります。これらの各事例では、セッション内で起動したすべてのサーバは同一プリンシパルによって実行されます。この方法の欠点は、クリデンシャルに有効期限がある点です。

ほかの方法としては、プロセスをセットアップしそのプロセスが独自のログイン・コンテキストを管理するようにします。サーバに提供される tpsvrinit(3c) 関数は、ログイン・コンテキストをセットアップし、そのコンテキストの有効期限が切れる前にコンテキストのリフレッシュを行うスレッドを起動できます。セットアップと起動を行うためのコード列が $TUXDIR/lib/dceserver.c に用意されています。このコードは、-DTPSVRINIT オプションでコンパイルし、tpsvrinit() 関数を作成します。(次のセクションで説明するような DCE サーバ用の main() としても使用できます)。このコードの詳細は付録 B の 1 ページ「DCE ゲートウェイ・アプリケーション」で説明しています。

DCE バインディング・ハンドルの使用法

BEA Tuxedo TxRPC はバインディング・ハンドルをサポートしていません。要求者のクライアント・スタブからゲートウェイ内のサーバ・スタブに RPC を送信する場合、BEA Tuxedo システムは名前解決のすべてとサーバの選択を処理し、利用可能なサーバ間のロード・バランシングを行います。ただし、ゲートウェイから DCE サーバへの送信時には DCE バインディングを使用できます。DCE バインディングを行った場合は、同じディレクトリ内で IDL ファイルを 2 つ使用するか、あるいは要求者および、ゲートウェイとサーバを構築するために 2 つのディレクトリを使用することをお勧めします。2 つの異なるファイル名を用いる前者の方法が例で示されていますが、この場合の IDL ファイルは第 2 の名前にリンクされます。最初の IDL ファイルでは、バインディング・ハンドルまたはバイディング属性は指定されていません。第 2 の IDL ファイルはゲートウェイと DCE サーバの生成に使用されますが、このファイルには、バインディング・ハンドルがオペレーションの第 1 パラメータとして挿入されるよう [explicit_handle] を指定する関連付けられた ACF ファイルが存在しています。ゲートウェイではハンドルがサポートされていないので、BEA Tuxedo サーバ・スタブから NULL ハンドルが生成されます。つまり、ゲートウェイの BEA Tuxedo ATMI サーバ・スタブと DCE クライアント・スタブのどこかの場所で、有効なバインディング・ハンドルを生成する必要があるということです。

バインディング・ハンドルの生成は、マネージャ・エントリ・ポイント・ベクトルを使用することで可能になります。デフォルトでは IDL コンパイラは、インターフェイス内のオペレーションごとに関数ポインタ・プロトタイプをもつ構造体を定義します。さらに、オペレーション名を元にしたデフォルト関数名をもつ構造体変数を定義し、初期化します。この構造体は以下のように定義されます。

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

ここで *<INTERF>* はインターフェイス名で、*<major>.<minor>* はインターフェイス・バージョンです。この変数は、サーバ・スタブ関数を呼び出すときに、逆参照されます。関数名とオペレーション名に競合や相違がある場合に、アプリケーションからこの変数の定義と初期化が行えるように、IDL コンパイラ・オプション、-no_mepv を使用してこの変数の定義と初期化を無効にできます。アプリケーションで自動バインディングの代わりに明示的または暗黙的なバインディングを行う場合には、-no_mepv オプションを指定できます。またアプリケーションは、オペレーションと同じで、異なる名前または静的な名前をパラメータとして使用する関数を指す構造体定義を指定できます。その後、この関数は、実際のオペレーション名を用いて明示的または暗示的に DCE/RPC クライアント・スタブ関数に渡される有効なバインディング・ハンドルを生成できます。

これは付録 B の 1 ページ「DCE ゲートウェイ・アプリケーション」の例に示されています。dcebind.c ファイルはバインディング・ハンドルを生成し、またエントリ・ポイント・ベクトルと関連する関数は dceepv.c に示されています。

blds_dce を使用する場合に -no_mepv オプションを指定するには、オプションが IDL コンパイラを経由して渡されるように "-i -no_mepv" オプションを指定する必要があることに注意してください。これは付録 B の 1 ページ「DCE ゲートウェイ・アプリケーション」の makefile 、rpcsimp.mk に示されています。

認証済み RPC

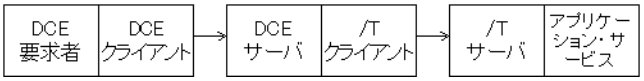
この時点でログイン・コンテキストとハンドルが有効なので、認証済み RPC コールを使用できます。バインディング・ハンドルをセットアップする一部として、rpc_binding_set_auth_info() を呼び出すことによって認証のためにバインディング・ハンドルに注釈を付けることができます。詳細については、『BEA Tuxedo C 言語リファレンス』を参照してください。これは、付録 B の 1 ページ「DCE ゲートウェイ・アプリケーション」の dcebind.c でバインディング・ハンドルを生成する一部として示されています。これによりゲートウェイと DCE サーバ間に認証（および潜在的な暗号化）をセットアップします。要求者が BEA Tuxedo ATMI サーバであれば、要求者は BEA Tuxedo 管理者として実行されることが保証されています。BEA Tuxedo クライアントの認証の詳細については、『BEA Tuxedo アプリケーション実行時の管理』を参照してください。

トランザクション

OSF/DCE はトランザクションをサポートしていません。つまり、ゲートウェイがリソース・マネージャを使用する 1 つのグループ内で実行されていて、RPC がトランザクション・モードで BEA Tuxedo ATMI クライアント・スタブに渡される場合、そのトランザクションは DCE サーバには転送されません。これに対する解決策は注意することだけです。

BEA Tuxedo ゲートウェイを用いた BEA Tuxedo サービスに対する DCE 要求者

図 4-2 BEA Tuxedo ゲートウェイを用いた BEA Tuxedo サービスに対する DCE 要求者



この図では、DCE 要求者は DCE クライアント・スタブを使って DCE サービスを起動し、次いでこの DCE サービスがアプリケーション・サービスの代わりに BEA Tuxedo ATMI クライアント・スタブを呼び出し、さらにこの BEA Tuxedo ATMI クライアント・スタブが TxRPC を使用して BEA Tuxedo ATMI サービスを起動します。この構成では、クライアントが DCE バインディングと認証を完全に制御していることに注意してください。アプリケーション・プログラマがミドル・サーバを構築するということは、アプリケーションも BEA Tuxedo ATMI サービスに対する DCE サーバのバインディングを制御するということを意味します。この方法は、DCE 要求者を BEA Tuxedo システムに直接リンクし、呼び出すことを希望しない場合に使用されます。

DCE サーバ用の `main()` は、`$TUXDIR/lib/dceserver.c` で提供されているコードを元にする必要があります。DCE サーバの `main()` 用のテンプレートがすでに用意されている場合、追加し、修正する項目がわずかですが必要になります。

まず、tpinit(3c) を呼び出して ATMI アプリケーションに参加する必要があります。アプリケーション・セキュリティが設定されている場合は、TPINIT バッファにユーザ名やアプリケーション・パスワードなど追加情報が必要になることもあります。終了する前に、tpterm(3c) を呼び出して ATMI アプリケーションへの参加を終了させる必要があります。-DTCLIENT でコンパイルしてみると、dceserver.c には tpinit と tpterm を呼び出すコードが含まれていることが分かるはずです。TPINIT バッファをセットアップするコードは、アプリケーションに応じた修正が必要です。管理に関するより多くの情報を提供するには、クライアントが DCE クライアントであることを、ユーザ名またはクライアント名で示すのが効果的です（例ではクライアント名を DCECLIENT に設定しています）。この情報は管理インターフェイスからクライアント情報を出力すると表示されます。

次の手順としては、BEA Tuxedo ATMI システム・ソフトウェアはスレッドセーフではないので、rpc_server_listen に渡すスレッディング・レベルは 1 に設定する必要があります。dceserver.c では、-DTCLIENT でコンパイルする場合にはスレッディング・レベルは 1 に設定され、それ以外の場合スレッディング・レベルはデフォルトの rpc_c_listen_max_calls_default に設定されます。詳細については、『BEA Tuxedo C 言語リファレンス』を参照してください。

この構成では、要求者は通常の DCE クライアントまたは DCE サーバとして構築されます。同様に、サーバは通常の BEA Tuxedo ATMI サーバとして構築されます。追加の作業として、TxRPC クライアント・スタブを使用する BEA Tuxedo ATMI クライアントとして機能し、さらに DCE/RPC サーバ・スタブを使用する DCE サーバとしても機能するゲートウェイ・プロセスを構築します。

2 つの IDL コンパイラを実行し、得られたファイルをリンクするプロセスは、DCE がリンクされた BEA Tuxedo ATMI クライアントを構築する bldc_dce(1) コマンドを使用することによって簡略化できます。

bldc_dce の使用法は次のとおりです。

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

このコマンドは入力として 1 つ以上の IDL ファイルを使用するので、ゲートウェイは 1 つ以上のインターフェイスを処理できます。これらのファイルごとに、クライアント・スタブを生成する場合は tidl を実行し、サーバ・スタブを生成する場合は idl を実行します。

このコマンドは各種の DCE 環境を認識し、コンパイルとリンクに必要なコンパイル・フラグと DCE ライブラリを提供します。新たな環境で開発している場合は、コマンドを変更して現在の環境に合ったオプションとライブラリの追加が必要になります。『BEA Tuxedo C 言語リファレンス』で説明しているように、復帰時メモリを確実に解放するために、常に rpc_ss_allocate(3) と rpc_ss_free(3) をメモリ割り当てに使用することを定義する -DTMDCEGW を指定してソース・ファイルがコンパイルされます。-DTMDCEGW を使用する場合には、BEA Tuxedo TxRPC ヘッダ・ファイルの代わりに DCE ヘッダ・ファイルが組み込まれます。

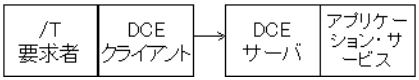
buildclient(1) を用いて BEA Tuxedo ATMI クライアントを作成するには、アプリケーション・ファイルを指定して (-f と -l オプションを使用します)、IDL 出力オブジェクト・ファイルをコンパイルします。-DTCLIENT オプションを指定してコンパイルされた dceserver.o と等価なファイルが 1 つ含まれる必要があることに注意してください。

実行可能なクライアントの名前は -o オプションで指定できます。

この構成を実行する場合、DCE サーバを開始する前に BEA Tuxedo ATMI のコンフィギュレーションを起動し、DCE 要求を受信する前に BEA Tuxedo のコンフィギュレーションが BEA Tuxedo ATMI アプリケーションに参加できるようにする必要があります。

DCE だけを用いた DCE サービスに対する BEA Tuxedo 要求者

図 4-3 DCE だけを用いた DCE サービスに対する BEA Tuxedo 要求者



この方法では、DCE 環境をクライアントが直接利用できるものとみなしています。構成によっては、これが制限または短所になる場合もあります。クライアント・プログラムは、DCE バインディングと認証を直接制御します。これは要求者が、DCE サービスを呼び出す BEA Tuxedo ATMI サービスであるか、または BEA Tuxedo サービスと DCE サービスの両方を呼び出す BEA Tuxedo クライアント (またはサーバ) である複合環境になっている可能性があることに注意してください。

DCE コードと混在して使用する BEA Tuxedo の TxRPC コードをコンパイルするときは、TxRPC ヘッダ・ファイルの代わりに、DCE ヘッダ・ファイルを使用してコードをコンパイルする必要があります。このことは、クライアントやサーバのスタブ・ファイルとアプリケーション・コードの両方に対してコンパイル時に -DTMDCE を定義することで行えます。tidl(1) からオブジェクト・ファイルを生成している場合は、"-cc_opt -DTMDCE" オプションをコマンド行に追加する必要があります。別の方法としては、次の例のように、IDL コンパイラから c_source を生成し、オブジェクト・ファイルではないこの C ソースを（オブジェクト・ファイルではなく）bldc_dce または blds_dce に渡します。

```
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
```

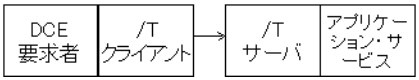
または

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

この例では、ゲートウェイ・プロセスを構築しているのではないので、build コマンドに .idl ファイルを指定することはできません。また、blds_dce コマンドはサーバに関連したサービス名を認識できないので、-s オプションを使ってサービス名を指定する必要があることに注意してください。

BEA Tuxedo-only を用いた BEA Tuxedo サービスに対する DCE 要求者

図 4-4　BEA Tuxedo-only を用いた BEA Tuxedo サービスに対する DCE 要求者



この最後の事例では、DCE 要求者が直接 BEA Tuxedo クライアント・スタブを呼び出します。

ここでも、クライアントおよびサーバのスタブ・ファイルとアプリケーション・コードの両方に対して、-DTMDCE をコンパイル時に使用する必要があります。この場合、要求者は BEA Tuxedo ATMI クライアントである必要があります。

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

前述のように、dceserver.c はアプリケーションを終了するために tpterm(3c) を、アプリケーションに参加するために tpinit(3c) を呼び出す必要があることに注意してください。

DCE/RPC と BEA Tuxedo TxRPC が混在したクライアントとサーバの構築

このセクションでは、bldc_dce(1) または blds_dce(1) コマンドを使用せずに、複合クライアントまたはサーバをコンパイルする場合に従う必要のある規則を要約します。

- 生成したクライアントとサーバ・スタブをコンパイルし、tidl(1) が生成したヘッダ・ファイルをインクルードしているクライアントとサーバ・アプリケーション・ソフトウェアをコンパイルするときは、TMDCE を定義する必要があります。(たとえば、-DTMDCE=1)。これにより、BEA Tuxedo TxRPC ヘッダ・ファイルの代わりに DCE ヘッダ・ファイルが使用されます。また、DCE のバージョンに中には、DCE ヘッダ・ファイル用に適切なディレクトリを追加し、ローカル環境用に適切な DCE 定義を確保する DCE コンパイル・シェルが用意されているものもあります。直接 C コンパイラを実行する代わりに、このシェルを使用します。
  DCE/RPC コンパイラと TMDCE 定義は、tidl の -cc_cmd オプションを使って指定できます。たとえば、

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

  または

  ```
  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
  ```

  コンパイル・シェルをもたないシステムの場合は、次のようになります。

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

  環境については、DCE/RPC のマニュアルを参照してください。
- サーバが RPC コールを行う場合は、前記のとおり set_client_alloc_free() を呼び出して rpc_ss_allocate() および rpc_ss_free() を使用できるようにします。詳細については、『BEA Tuxedo C 言語リファレンス』を参照してください。

- 実行可能ファイルをリンクする場合は、-ltrpc の代わりに -ldrpc を使用して、DCE/RPC と互換性のある BEA Tuxedo TxRPC ランタイムを取得します。たとえば、

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

または

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

simp_cstub.o は tidl(1) によって生成され、また dce_cstub.o は idl によって生成されたと仮定します。最初の例では、DCE コンパイラ・シェルを使用しないクライアントの構築を示しています。この場合、DCE ライブラリ （-ldce)、スレッド・ライブラリ（-lpthreads)、リエントラント C ライブラリ (-lc_r) は明示的に指定しなければなりません。2 番目の例では、必要なライブラリを透過的にインクルードする DCE コンパイラ・シェルの使用方法を示します。一部の環境では、ネットワーキングと XDR 用の buildserver および buildclient によってインクルードされたライブラリは、DCE コンパイラ・シェルによってインクルードされるライブラリと競合する場合があります（これらのライブラリのリエントラント・バージョンが存在することもあります)。この場合は、buildserver(1) と buildclient(1) ライブラリは -d オプションを使って変更できます。リンク上の問題が発生した場合は、上記の例で示すとおり、-d " " を使用してネットワーキングと XDR ライブラリを切り離してみてください。それでもリンクに問題がある場合は、-d オプションを指定しないで、-v オプションを指定してコマンドを実行し、デフォルトで使用されるライブラリを調べてみてください。その後で、ライブラリが複数存在していれば、-d オプションを使ってライブラリのサブセットを指定します。ネットワーキング、XDR、DCE ライブラリは環境に応じて異なるため、ライブラリの正しい組み合せは環境に依存します。

注記 Windows では現在、DCE スタブと BEA Tuxedo TxRPC スタブを混在させることはできません。

5　アプリケーションの実行

ここでは、次の内容について説明します。

- 予備知識
- アプリケーションの構成方法
- アプリケーションの起動とシャットダウン
- アプリケーションの管理
- 動的サービス宣言の使用

予備知識

RPC サーバを追加するための構成の変更を行う BEA Tuxedo ATMI システムの管理者は、ASCII コンフィギュレーション・ファイル (形式については UBBCONFIG(5) を参照) の作成方法と、tmloadcf(1) を用いたバイナリ・コンフィギュレーション・ファイルの読み込みについての知識を持っている必要があります。これらの作業については、『BEA Tuxedo アプリケーション実行時の管理』で説明されています。

アプリケーションの構成方法

RPC サーバの構成は、要求 / 応答サーバの場合と同様に行います。RPC サーバや RPC サーバのグループごとに、SERVERS ページ内にエントリが 1 つ必要です。(MAX を 1 よりも大きな値にセットすると、1 つのエントリで複数の RPC サーバが設定できます)。必要に応じて RQADDR を指定すると、RPC サーバの複数のインスタンスが同一の要求キューを共有します (複数サーバ、単一キュー設定)。CONV パラメータは指定しないか、あるいは N に設定 (例 : CONV=N) する必要があります。付録 A の 1 ページ「アプリケーション例」のコンフィギュレーション・ファイル例を参照してください。

サーバがトランザクションの一部になる場合には、TLOGDEVICE をもつマシン上のグループに存在する必要があります。GROUPS エントリは、関連するリソース・マネージャへのアクセスに用いる TMSNAME と OPENINFO 文字列を使用して構成される必要があります。

SERVICES エントリの指定はオプションです。これを指定した場合、サービス名は前章で説明したとおり、インターフェイス名とバージョン番号に基づく必要があります。SERVICES エントリが必要になるのは、デフォルトの値とは異なる特別な負荷、優先順位、トランザクション時間を指定する場合です。また、このエントリは、AUTOTRAN 機能をオンにするためにも使用されます。この機能をオンにすると、入力要求がトランザクション・モードではない場合に、サービスに対してトランザクションが自動的に起動されるようにします。処理されるバッファの種類は CARRAY だけなので、SERVICES エントリを使用してバッファの種類 BUFTYPE を指定しないでください。また、ルーティングは RPC 要求に対してはサポートされていないので、ROUTING も指定しないでください。

アプリケーションを起動する前に、tmloadcf(1) コマンドを用いて ASCII コンフィギュレーション・ファイルをバイナリ TUXCONFIG ファイルにロードします。

tmconfig コマンドを使用すると、起動したアプリケーションに RPC サーバ用エントリを追加できることに注意してください。追加方法については『BEA Tuxedo コマンド・リファレンス』の tmconfig、wtmconfig(1) を参照してください。

アプリケーションの起動とシャットダウン

構成を変更したときには、tmboot(1) を用いてアプリケーションを起動してください。tmshutdown(1) を用いて、アプリケーションをシャットダウンします。付録 A の 1 ページ「アプリケーション例」の例を参照してください。

RPC サーバの起動とシャットダウン方法は、要求 / 応答サーバのものと同じです。RPC サーバの起動やシャットダウンは、全体構成の一部として行うときには -y オプション、グループの一部として行うときには -g オプション、論理マシンの一部として行うときには -l オプション、サーバ名で行うときには -s オプションを使用します。

アプリケーションの管理

管理インターフェイス内では、RPC サーバは要求 / 応答サーバとして表現されます。前に説明したとおり、tmconfig を使用すれば RPC サーバとサービスの動的再コンフィギュレーションが可能です。詳細については、『BEA Tuxedo コマンド・リファレンス』の tmconfig、wtmconfig(1) を参照してください。tmadmin(1) コマンドにより RPC サーバを監視できます。tmadmin printserver コマンドを用いて、RPC サーバ名および関連するランタイム情報 (たとえば、サービスやオペレーションの実行、ロード等) を出力できます。また printservice コマンドを使用して、利用可能な RPC サービス (インターフェイス) を出力できます。出力例については、付録 A の 1 ページ「アプリケーション例」を参照してください。

動的サービス宣言の使用

RPC サービスは、要求 / 応答サービスの制御と同じ方法で動的に制御可能です。既に説明したとおり、サービス名はオペレーションの名前ではなく、インターフェイス名とバージョン番号でした。一般的に、サービス名は、-s オプションを使用して、buildserver(1) が実行されるときに指定され、サーバが -A オプションを使用して起動されるときに自動的に宣言されます。サービス (インターフェイス) 名を動的に宣言する方法としては、adv コマンドを用いて tmadmin から宣言するか、あるいはサーバ内から関数 tpadvertise(3c) を用いて宣言します。サービス (インターフェイス) 名の宣言を動的に解除する方法としては、unadv コマンドを用いて tmadmin から解除するか、サーバ内から関数 tpunadvertise(3c) を用いて解除します。サービス名は、tmadmin(1) から一時的に使用停止したり、停止解除 (再開) したりできます。サービス名の宣言解除、または使用停止を行うと、関連のあるインターフェイス内で定義されたオペレーションはすべて利用できなくなることに注意してください。

A　アプリケーション例

ここでは、次の内容について説明します。

- この付録の内容
- 前提条件
- rpcsimp アプリケーションの構築

この付録の内容

この付録では、TxRPC を使用した `rpcsimp` と呼ばれる単一クライアント、単一サーバで構成するアプリケーションについて解説します。この対話型アプリケーションのソース・ファイルは、BEA Tuxedo ATMI ソフトウェアに同梱されていますが、RTK 製品には含まれません。

前提条件

このアプリケーション例を実行する前に、BEA Tuxedo ソフトウェアをインストールし、この付録で参照するファイルやコマンドが使用できるようにしておきます。

rpcsimp アプリケーションの構築

rpcsimp は、TxRPC を利用した非常に基本的な BEA Tuxedo ATMI アプリケーションです。このアプリケーションは、1 つのクライアントと 1 つのサーバで構成します。クライアントはリモート・プロシージャ・コール (オペレーション) to_upper() と to_lower() を呼び出します。これらのプロシージャ・コールは、いずれもサーバ内に実装されるものです。オペレーション to_upper() は、文字列を小文字から大文字へ変換してクライアントに返します。一方 to_lower() は、文字列を大文字から小文字へ変換してクライアントに返します。各プロシージャ・コールが復帰すると、クライアントは文字列出力をユーザの画面に表示します。

以下に本プログラム例を構成、実行する手順を解説します。

ステップ 1: アプリケーション・ディレクトリの作成

rpcsimp 用のディレクトリを作成し、そのディレクトリに移動します。

```
mkdir rpcsimpdir
cd rpcsimpdir
```

注記 この様にしておくと、作業開始時にあった rpcsimp 関連のファイルと、作業中に作成した新たなファイルを容易に把握できます。標準シェル (/bin/sh) または Korn シェルを使用してください。C シェル (csh) は使わないでください。

ステップ 2: 環境変数の設定

必要な環境変数を設定し、エクスポートします。

```
TUXDIR=<BEA Tuxedo システム・ルート・ディレクトリのパス名 >
TUXCONFIG=< 現在のワーキング・ディレクトリのパス名 >/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
```

注記 TUXDIR と PATH は、BEA Tuxedo ATMI のディレクトリ構造内のファイルにアクセスし、BEA Tuxedo ATMI コマンドを実行するのに必要です。コンフィギュレーション・ファイルがロードできるようにするには、TUXCONFIG を設定する必要があります。共有オブジェクトを使用する場合、LD_LIBRARY_PATH 環境変数も設定します。

ステップ 3: ファイルのコピー

rpcsimp 用のファイルをアプリケーション・ディレクトリにコピーします。

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

一部のファイルを編集してから、実行形式ファイルを作成することがあるので、本ソフトウェアで提供されたオリジナルのファイルではなく、コピーしたファイルを用いて作業することをお勧めします。

ステップ 4: ファイルの一覧表示

ファイルを一覧表示します。

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

注記 この一覧には、付録 B の 1 ページ「DCE ゲートウェイ・アプリケーション」で説明する DCE ゲートウェイ例で使用するファイルは含まれていません。

アプリケーションを構成するファイルは次のセクションで説明します。

IDL 入力ファイル—simp.idl

リスト A-1 simp.idl

```
[uuid(C996A680-9FC2-110F-9AEF-930269370000), version(1.0) ]
interface changecase
{
/* 文字列を大文字に変換 */
void to_upper([in, out, string] char *str);
/* 文字列を小文字に変換 */
void to_lower([in, out, string] char *str);
}
```

このファイルは 2 つのオペレーション (to_upper と to_lower) を持った単一インターフェイス (changecase version1.0) を定義します。各オペレーションは、NULL (文字) で終わる文字列を入出力パラメータとして使用します。ACF フィルは提供されないので、状態変数は使用されず、クライアント・プログラムは例外処理を行う必要があります。各オペレーションには戻り値が生成されないことを示すために void で宣言します。simp.idl はスタブ関数の生成に使用されます（以下を参照）。

クライアント・ソース・コード—client.c

リスト 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) {/* コマンド行引数がある場合はそれを使用 */
    (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);
  /* 制御の流れは ENDTRY 後に続く */
  CATCH_ALL
    exc_report(THIS_CATCH); /* stderr に出力 */
    (void) tpterm();
    exit(1);
  ENDTRY
  (void) tpterm();
  exit(0);
}
```

ヘッダ・ファイル simp.h は、simp.idl を元に IDL コンパイラによって生成され、2 つのオペレーション用の関数プロトタイプを持っています。simp.h ヘッダには、RPC ランタイム関数 (本例には示されていません) 用と例外処理用のヘッダ・ファイルも含まれます。関数 tpterm(3c) が呼び出されるので、ヘッダ・ファイル atmi.h がインクルードされます。コマンド行に引数を指定すると、その引数は大文字や小文字への変換に使用されます (デフォルトでは "hello world" になります)。例外処理はエラーの捕捉に使用されます。たとえば、サーバが利用できない、メモリを割り当てられない、通信できないなどの場合に例外が生成されます。TRY ブロックは、2 つのリモート・プロシージャをカプセル化しています。エラーが発生した場合、実行の制御は CATCH_ALL ブロックに移り、例外（THIS_CATCH）を文字列に変換し、この文字列を exc_report を使用して標準エラー出力に表示して、終了します。アプリケーションを正しく終了するには、アプリケーションの実行が正常に行われた場合も、異常が発生した場合も共に、tpterm(3c) が呼び出されていることに注意してください。tpterm(3c) を呼び出さないと、ワークステーション以外のクライアントでは、userlog(3c) に警告が出力され、リソースが関連付けられたままになります。ワークステーション・クライアントでは、接続がタイムアウトするまでリソースが関連付けられたままになります。

サーバ・ソース・コード—server.c

リスト 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;
}
```

client.c の場合と同様、このファイルでも simp.h をインクルードします。

さらに、関数 tx_open(3c) を呼び出すので、このファイルは tx.h もインクルードします。(リソース・マネージャにアクセスしない場合でも tx.h をインクルードします。これは X/OPEN TxRPC 仕様 で定められているとおりです)。関数 tpsvrinit(3c) は、ブート時に必ず関数 tx_open() が 1 度呼び出されるようにするために用意されています。障害が起こると、-1 が返され、サーバのブートは失敗します。これは自動的に行われるので、この機能を提供する必要はありません。

これら 2 つのオペレーション関数は、アプリケーション作業、つまり今回の例では大文字や小文字への変換を行うために用意されています。

Makefile—rpcsimp.mk

リスト A-4　rpcsimp.mk

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

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

# Tuxedo サーバ
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
#
# DCE ゲートウェイの処理に関する部分は省略
#

# クリーンアップ
clean::
         rm -f *.o server $(ALL2) ULOG.* TUXCONFIG
         rm -f stderr stdout *stub.c *.h simpdce.idl gwinit.c
clobber:clean
```

makefile により、クライアントやサーバの実行プログラムが構築されます。

付録 B の 1 ページ「DCE ゲートウェイ・アプリケーション」で説明する、DCE ゲートウェイの処理に関する makefile の部分はリストでは省略されています。

クライアントはヘッダ・ファイル simp.h とクライアント・スタブ・オブジェクト・ファイルに依存して構築されます。buildclient を実行すると、ソース・ファイルとして client.c を、リンクするファイルとしてクライアント・スタブ・オブジェクト・ファイルを使用し、-ltrpc RPC ランタイム・ライブラリがリンクされ、実行形式のファイルが作成、出力されます。

サーバは、ヘッダ・ファイル simp.h とサーバ・スタブ・オブジェクト・ファイルに依存して構築されます。buildserver を実行すると、ソース・ファイルとして server.c を、リンクするファイルとしてサーバ・スタブ・オブジェクトファイルを使用し、-ltrpc により、RPC ランタイム・ライブラリがリンクされ、実行形式のファイルが作成、出力されます。

クライアントやサーバのスタブ・オブジェクト・ファイルと、ヘッダ・ファイル simp.h は、いずれも IDL 入力ファイルで tidl コンパイラを実行することで作成されます。

clean ターゲットは、アプリケーションの構築中および実行中に作成されたファイルをすべて削除します。

コンフィギュレーション・ファイル—ubbconfig

ASCII コンフィギュレーション・ファイルの例を次に示します。ユーザが使用している構成に応じて、マシン名、TUXCONFIG、TUXDIR、APPDIR を設定する必要があります。

リスト 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 から DCE へのゲートウェイ
#simpgw SRVGRP=GROUP1 SRVID=2
*SERVICES
*ROUTING
```

MAXWSCLIENTS と WSL 行は、ワークステーション構成の場合にコメントを解除します。ワークステーション・リスナ用のリテラル *netaddr* は、『ファイル形式、データ記述方法、MIB、およびシステム・プロセスのリファレンス』の WSL(5) の説明に従って設定してください。

ステップ 5: 構成の変更

ASCII コンフィギュレーション・ファイル ubbconfig を編集して、位置に依存した情報 (例 : ユーザ独自のディレクトリ・パス名とマシン名) を指定します。置き換えるテキストは山かっこで囲みます。TUXDIR、TUXCONFIG、APPDIR、および実行するマシン名は、フル・パス名に置き換えます。必要な値を以下に要約します。

TUXDIR
: BEA Tuxedo ソフトウェアのルートディレクトリのフル・パス名です。上記のように設定します。

TUXCONFIG
: バイナリ・コンフィギュレーション・ファイルのフル・パス名です。上記のように設定します。

APPDIR
: アプリケーションを実行するディレクトリのフル・パス名です。

UNAME
: アプリケーションを実行するマシンの名前です。この名前は、UNIX の uname -n コマンドが出力するものです。

ワークステーションのコンフィギュレーション・ファイルでは、MAXWSCLIENTS と WSL 行のコメントを解除します。<*address*> は、ワークステーション・リスナに対して設定します。(詳細については WSL(5) を参照。)

ステップ 6: アプリケーションの構築

次のコマンドを実行してクライアントとサーバ・プログラムを構築します。

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

ステップ 7: コンフィギュレーション・ファイルのロード

次のコマンドを実行してバイナリの TUXCONFIG コンフィギュレーション・ファイルをロードします。

```
tmloadcf -y ubbconfig
```

ステップ 8: 構成のブート

次のコマンドを実行してアプリケーションをブートします。

```
tmboot -y
```

ステップ 9: クライアントの実行

1. 必要に応じて大文字に変換した後で小文字に変換する文字列を指定して、ネイティブ・クライアント・プログラムを実行します。

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

2. ワークステーション上で実行する場合には、環境変数 WSNADDR を WSL プログラム用に指定したアドレスと一致するように設定します。Windows クライアントは、次のコマンドで実行できます。

   ```
   >win wclient
   ```

   注記 ダイナミック・リンク・ライブラリは、別に開発されたアプリケーション (例 : ビジュアル・ビルダ) で使用することもできます。

ステップ 10: RPC サーバの監視

tmadmin(1) を使うと、RPC サーバを監視できます。次の例では、psr と psc を使ってサーバ・プログラムの情報を表示します。簡潔モード ("+" で表示) では、RPC サービス名の長さは切りつめられることに注意してください。一方、冗長モードを使えば、フル・ネームを表示できます。

リスト A-6　tmadmin psr および psc の出力

```
$ 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
```

ステップ 11: 構成のシャットダウン

次のコマンドを実行して、アプリケーションをシャットダウンします。

```
tmshutdown -y
```

ステップ 12: 作成ファイルのクリーンアップ

次のコマンドを入力して、作成したファイルを削除します。

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

B DCE ゲートウェイ・アプリケーション

ここでは、次の内容について説明します。

- この付録の内容
- 前提条件
- DCE ゲートウェイ・アプリケーションとは
- rpcsimp アプリケーションのインストール、構成、実行

この付録の内容

この付録は、付録 A の 1 ページ「アプリケーション例」で解説したアプリケーション例 `rcpsimp` を元に構築されています。サーバは OSF/DCE サーバに変更され、またゲートウェイが使用されますので、BEA Tuxedo ATMI クライアントは明示的バインディングと認証済み RPC を使ってサーバと通信できます。この対話型アプリケーションのソース・ファイルは、BEA Tuxedo ATMI ソフトウェア開発キットに付属しています。

前提条件

このトピックでは DCE に関する知識が必要ですが、このマニュアルには DCE チュートリアルは含まれていません。参考文献として『Guide to Writing DCE Applications』(John Shirley、O'Reilly and Associates, Inc.) を参照してください。

DCE ゲートウェイ・アプリケーションとは

このアプリケーションは、アプリケーション例 rpcsimp の拡張版です。「付録 A」の場合と同様に、クライアントはリモート・プロシージャ・コール (オペレーション) to_upper() と to_lower() を呼び出します。

この例では、RPC は BEA Tuxedo ATMI クライアントから DCE ゲートウェイ・プロセスに転送され、DCE ゲートウェイ・プロセスが要求を DCE サーバに送ります。この例をもう少し現実的なものとするため、ゲートウェイ・プロセスから DCE サーバへの通信では、自動バインディングと認証済み RPC の代わりに明示的バインディグを使用します。

以下に本プログラム例を構築、実行する手順を解説します。クライアントは付録 A の 1 ページ「アプリケーション例」で説明した、どのプラットフォーム上でも実行できます。クライアントの構築や実行方法は同じですから、この章ではこれ以上の説明は行いません。ゲートウェイと DCE サーバは、DCE ソフトウェアがインストールされた POSIX プラットフォームで実行する必要があります。この付録では、ワークステーション・プラットフォーム上でのクライアントのインストールやコンパイルについては説明しません。

このサンプル・プログラムは、OSF/DCE 準拠のソフトウェアを組み込んだほかのプラットフォームでも動作します。

rpcsimp アプリケーションのインストール、構成、実行

以下のステップは、サンプル・アプリケーションのインストール、構成、実行を行う方法を示しています。

ステップ 1: アプリケーション・ディレクトリの作成

rpcsimp 用のディレクトリを作成し、そのディレクトリに移動します。

```
mkdir rpcsampdir
cd rpcsampdir
```

注記 この様にしておくと、作業開始時にあった rpcsimp 関連のファイルと、作業中に作成した新たなファイルを容易に把握できます。標準のシェル (/bin/sh) または Korn シェルを使用してください。C シェル (csh) は使わないでください。

ステップ 2: 環境の設定

必要な環境変数を設定しエクスポートします。

```
TUXDIR=<BEA Tuxedo ルート・ディレクトリのパス名 >
TUXCONFIG=< ユーザが指定する作業ディレクトリのパス名 >/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
```

BEA Tuxedo ATMI ディレクトリ内のファイルにアクセスできるように、さらに BEA Tuxedo ATMI のコマンドを実行できるように、TUXDIR と PATH を指定する必要があります。TUXCONFIG を設定し、コンフィギュレーション・ファイルがロードできるようにします。共有オブジェクトを使用する場合は、環境変数も設定する必要があります。(たとえば、LD_LIBRARY_PATH)。

ステップ 3: ファイルのコピー

rpcsimp 用のファイルをアプリケーション・ディレクトリにコピーします。

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

ここでは一部のファイルを編集し、実行形式にするので、ソフトウェアに同梱されたオリジナルのファイルではなく、コピーしたファイルを使用して作業を開始することをお勧めします。

ステップ 4: ファイルの一覧表示

ファイルを一覧表示します。

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

(このセクションで参照しない一部のファイルは省略しています。)

アプリケーションを構成するファイルは次のセクションで説明します。付録 A の 1 ページ「アプリケーション例」で説明したファイル、client.c、simp.idl、ubbconfig については、ここでは解説しません。

IDL ACF ファイル—simpdce.acf

リスト B-1　simpdce.acf

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

「付録 A」の例で用いた simp.idl ファイルは、ゲートウェイと DCE サーバの構築に使用されます。しかし、simp.idl ファイルは DCE コンパイラと BEA Tuxedo の IDL コンパイラの両方でコンパイルされるために、2 つの simp.h ヘッダ・ファイルが同じ名前で作成されます。また、クライアントにではなく、サーバに明示的バインディングを指定できるように、この例では ACF ファイルを使用しています。バインディングを指定しない TxRPC 用のものと、明示的ハンドルをもつ DCE/RPC 用のものを使用して、同一ディレクトリ内で第 2 のファイル名に IDL ファイルをリンクすることをお勧めします。この場合、simp.idl は simpdce.idl に名前が変更され、また関連する ACF ファイルは simpdce.acf になります。Makefile は simpdce.idl を作成し、さらに IDL コンパイラを実行するときに simpdce.acf も検索します。インターフェイスのすべてのオペレーションが明示的ハンドルを使用することを示すために、ACF ファイルが使用されている点に注意してください。第 1 パラメータとして［handle］パラメータを指定せずにオペレーションが IDL ファイル内で定義されているので、1 つのオペレーションが自動的に関数プロトタイプとスタブ関数呼び出しに追加されます。

関数のバインディング—dcebind.c

ここではページ数の制限から dcebind.c 用のソース・コードは示していませんが、$TUXDIR/apps/rpcsimp にあります。

このファイルには次の 3 種類の作業を行う関数、dobind() が含まれます。

- 目的のインターフェイス仕様で DCE サーバ用のバインディング・ハンドルを取得し、さらに完全に解決されたハンドルの関連する端点を取得します。
- サーバのプリンシパル名を取得し、セキュリティ・レジストリをチェックしてプリンシパルが指定されたグループのメンバーであるかどうか判別することによって、サーバの認証をある程度行います。

- 認証済み RPC が実行されるようバインディング・ハンドルに注釈を付けます。保護レベルは、秘密鍵による認証と DCE PAC を基本とした認証を使用するパケット・レベルの整合性 (パケット・チェックサム checksum による呼び出しごとの相互認証) です。

dcebind.c では次の事項を修正する必要があります。

- <*HOST*> は、DCE サーバが実行されるホスト・マシンの名前に変更する必要があります。これはディレクトリに格納されるサービス名の一部であって、"_host" で終わるサービス名の規約に従っています。完全にサフィックスを取り除くこともできます (削除する場合は、dceserver.c でも同じ変更を行う必要があります)。
- <*SERVER_PRINCIPAL_GROUP*> は、サーバを実行する DCE プリンシパルに関連付けられたグループに変更する必要があります。これは相互認証の一部として使用されます。
- cell_admin として rgy_edit を実行してサーバ・プリンシパル・グループを作成します。サーバ・プリンシパルを作成し、グループを持つプリンシパルにアカウントを追加して、サーバにキー・テーブルを作成します。クライアントを実行するためには、ユーザ自身のプリンシパルとアカウントも作成します。これらの DCE エンティティを生成するスクリプト例をステップ 8 の「DCE の構成」の項に示します。

エントリ・ポイント・ベクトル—dceepv.c

リスト B-2 dceepv.c

```
#include <simpdce.h>   /* IDL コンパイラによって生成されるヘッダ */
#include <dce/rpcexc.h> /* RAISE マクロ */

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

/*
 * マネージャのエントリ・ポイント・ベクトルは、
 * 有効な DCE バインディング・ハンドルを生成して DCE サーバに送れるように定義されます。
 * Tuxedo はハンドルをサポートしないので、エントリ・ポイント関数への入力ハンドルは、
 * 常に NULL になることに注意してください。
 */

 /* 2 つのオペレーションを持つマネージャ・エントリ・ポイント・ベクトル */
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) {  /* サーバ用のバインディング・ハンドルを取得 */
   userlog("binding failed");
   RAISE(rpc_x_invalid_binding);
  }
  to_upper(handle, str);   /* DCE クライアント・スタブを呼び出す */
}

void
myto_lower(rpc_binding_handle_t hdl, idl_char *str)
{
  rpc_binding_handle_t handle;
  if (dobind(&handle) 0) { /* サーバ用のバインディング・ハンドルを取得 */
   userlog("binding failed");
   RAISE(rpc_x_invalid_binding);
  }
  to_lower(handle, str);   /* DCE クライアント・スタブを呼び出す */
}
```

dceepv.c にはゲートウェイで使用されるマネージャ・エントリ・ポイント・ベクトルが含まれます。マネージャ・エントリ・ポイント・ベクトルは BEA Tuxedo ATMI サーバ・スタブによって呼び出され、DCE クライアント・スタブを呼び出します。構造体のデータ型は simpdce.h で定義され、dceepv.c にインクルードされます。またデータ型は、ローカル関数 myto_upper() と myto_lower() によって初期化されます。これらの関数はそれぞれ dobind() を呼び出して、認証済み RPC のために注釈が付けられたバインディング・ハンドルを取得し、関連するクライアント・スタブ関数を呼び出します。

DCE マネージャ—dcemgr.c

リスト B-3 dcemgr.c

```
#include <stdio.h>
#include <ctype.h>
#include "simpdce.h"    /* IDL コンパイラによって生成されるヘッダ */
#include <dce/rpcexc.h>  /* RAISE マクロ */

#include <dce/dce_error.h> /* dce_error_inq_text を呼び出すために必要 */
#include <dce/binding.h>  /* レジストリにバインディング */
#include <dce/pgo.h>    /* レジストリ・インターフェイス */
#include <dce/secidmap.h> /* グローバル名をプリンシパル名に変換 */

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;           /* クライアント・パッケージ */
  unsigned_char_t *server_principal_name; /* 要求されるサーバ・プリンシパル
*/
  unsigned32  protection_level;     /* 保護レベル */
  unsigned32  authn_svc;         /* 認証サービス */
  unsigned32  authz_svc;         /* 権限サービス */
  sec_rgy_handle_t rgy_handle;
  error_status_t status;
  /*
   * クライアントがこの呼び出しのために選択した認証パラメータを
   * チェックします。
   */
   rpc_binding_inq_auth_client(
     handle,           /* 入力ハンドル */
     (rpc_authz_handle_t *)&pac,  /* 返されるクライアント・パッケージ */
     &server_principal_name,    /* 返される要求サーバのプリンシパル */
     &protection_level,      /* 返される保護レベル */
     &authn_svc,          /* 返される認証サービス */
     &authz_svc,    /* 返される権限サービス */
     &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;
  }
```

```
  /*
   * 呼び出し側が呼び出し側で必要な
   * 保護レベル、認証レベルおよび権限レベルを指定していることを確認します。
   */
  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);

  /* ACL または参照モニタ・チェックはここで実行できます */

  /* 大文字に変換 */
  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);

  /* ACL または参照モニタ・チェックはここで実行できます */

  /* 小文字に変換 */
  for (p=str; *p != '\0 '; p++)
    *p = tolower((int)*p);
  return;
}
```

dcemgr.c には DCE サーバ用のマネージャ・コードがあります。checkauth() 関数は、クライアントの認証 (保護、認証、権限のレベル) を検査するユーティリティ関数です。各オペレーション、to_upper および to_lower はこの関数を呼び出してクライアントの妥当性を検査し、次にオペレーション自体を実行します。アクセス制御リストを使用するアプリケーションでは、ACL 検査は認証検査の後、オペレーションの実行の前に行われます。

DCE サーバ—dceserver.c

ここではページ数の制限から dceserver.c 用のソース・コードは示しません。使用する環境に応じて、このファイルにはいくつか修正を加える必要があります。

- *<HOST>* は、DCE サーバが実行されるホスト・マシンの名前に変更する必要があります。これはディレクトリに格納されるサービス名の一部であって、*_host.* で終わるサービス名の規約に従っています。完全にサフィックスを取り除くこともできます (削除する場合は、dcebind.c でも同じ変更を行う必要があります)。
- *<DIRECTORY>* は、サーバ・キー・テーブルを作成するディレクトリのフルパス名を設定します。次のコマンドを実行すると、キー・テーブルが作成されます。

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

ここで、*SERVER_PRINCIPAL* は、サーバが実行される DCE プリンシパル、*PASSWORD* はプリンシパルに関連付けられたパスワード、*SERVER_KEYTAB* はサーバ・キー・テーブルの名前です。

<PRINCIPAL> は、サーバが実行される DCE プリンシパルの名前に変更する必要があります。

"*ANNOTATION* " は、注釈がサーバ用のディレクトリ・エントリに保存されるように変更できます。

dceserver.c は、実際にはアプリケーションで 2 回使用されます。1 回は DCE サーバ用の main() として使用され、もう 1 回は DCE ゲートウェイ用の tpsvrinit() (makefile で gwinit.c にリンクされ、DTPSVRINIT を使用してコンパイルされます) として使用されます。

特別なマクロ定義を指定しないでコンパイルすると、このファイルは、次の作業を行う DCE サーバのための main() を (argc と argv コマンド行オプションを持ちます) を生成します。

- インターフェイスの登録。
- サーバ・バインディング情報と端点の作成。
- サーバ・テーブル内の情報を使用し、サーバ・プリンシパル用の DCE ログイン・コンテキストの確立。
- 認証情報の登録。
- バインディングの取得と、端点・マップ内への情報の登録。
- バインディング情報のディレクトリ名前空間へのエクスポート。
- 名前空間内のグループへの名前の追加 (オプション)。
- 要求の受信。

- rpc_server_listen が復帰した後のクリーンアップ。

このプログラムは、コマンド行オプションを調べ、それらを使用するように修正できます。

-DTCLIENT でコンパイルすると、このファイルは上記のように main() を生成しますが、tpinit() を呼び出してクライアントとして BEA Tuxedo ATMI アプリケーションに参加し、終了する前に tpterm() を呼び出します。これは、プロセスが DCE サーバおよび BEA Tuxedo ATMI クライアントになるような、DCE から BEA Tuxedo に渡される呼び出し用の DCE ゲートウェイとして使用されます。

-DTPSVRINIT でコンパイルすると、このファイルは、次の作業行う BEA Tuxedo サーバのために tpsvrinit() (argc と argv サーバ・コマンド行オプションを持ちます) を生成します。

- サーバ・キー・テーブル内の情報を使用して、プリンシパル用の DCE ログインの確立。
- 認証情報の登録。
- サーバに関連付けられたリソース・マネージャを開くための tx_open の呼び出し。

このプログラムは、コマンド行オプションを調べ、それらを使用するように修正できます。

これらの各事例で、ログイン・コンテキストは establish_identity を呼び出すことで確立されますが、この establish_identity は、サーバのネットワーク識別子を取得し、キー・テーブル・ファイル内のサーバのシークレット・キーを使用して識別子を解除し、プロセスのログイン・コンテキストを設定します。2 つのスレッドが開始されます。最初のスレッドは有効期限が切れる前にログイン・コンテキストを最新状態に更新し、もう一つのスレッドはサーバのシークレット・キーを定期的に変更します。

Makefile—rpcsimp.mk

リスト B-4　rpcsimp.mk

```
CC=cc
CFLAGS=
TIDL=$(TUXDIR)/bin/tidl
LIBTRPC=-ltrpc
all:client server
# Tuxedo クライアント
client:simp.h simp_cstub.o
  CC=$(CC) CFLAGS=$(CFLAGS) $(TUXDIR)/bin/buildclient -oclient \
```

```
    -fclient.c -fsimp_cstub.o -f$(LIBTRPC)
#
# Tuxedo サーバ省略
#

# Tuxedo ゲートウェイ例
# 上記の Tuxedo クライアントとゲートウェイ・サーバおよび DCE サーバを使用
#
#
# 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 から DCE へのゲートウェイ
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 サーバ
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)

# クリーンアップ
clean::
  rm -f *.o server $(ALL2) ULOG.* TUXCONFIG
  rm -f stderr stdout *stub.c *.h simpdce.idl gwinit.c

clobber:clean
```

makefile は実行可能クライアント、ゲートウェイ、DCE サーバ・プログラムを構築します。

このソフトウェアを構築する前に、rpcsimp.mk を修正して DCE サーバを構築するための正しいオプションとライブラリを設定する必要があります。上記で示したように、makefile には複数のプラットフォームに対応した設定が含まれています。使用しているプラットフォームに応じて、該当する DCECFLAGS と DCELIBS 変数ペアのコメント文指定を解除します (シャープ記号を削除します)。異なるプラットフォームを使用しているのであれば、独自の定義を追加します。

makefile を少し書き直せば、クライアントは付録 A の 1 ページ「アプリケーション例」の場合と同じ方法で構築できます。simpdce.idl を blds_dce に渡すことで DCE ゲートウェイが構築され、blds_dce は DCE に対してゲートウェイとして機能する BEA Tuxedo ATMI サーバを構築します。gwinit.o (-DTPSVRINIT でコンパイルされた dceserver.c)、dobind.o (DCE サーバ用のバインディング・ハンドルを取得)、dceepv.o (マネージャ・エントリ・ポイント・ベクトル) も含まれています。IDL コンパイラが独自のマネージャ・エントリ・ポイント・ベクトルを生成しないよう、-i -no_mepv が指定されることに注意してください。DCE サーバは、DCE IDL コンパイラで simpdce.idl をコンパイルし、dceserver.o と dcemgr.o をインクルードすることで構築されます。

ステップ 5: 構成の変更

1. ASCII の ubbconfig コンフィギュレーション・ファイルは 付録 A の 1 ページ「アプリケーション例」の説明どおりに変更する必要があります。このステップは必須です。
2. また *SERVERS セクションでは、シャープ記号 (#) を行の先頭に付け "*server* " 行をコメント文にし、*dceserver* 行ではコメント文の指定を解除します。

ステップ 6: アプリケーションの構築

1. ソフトウェアを構築する前に、上記で説明した様に rpcsimp.mk を修正して DCE サーバを構築するために正しいオプションとライブラリを設定する必要があります。
2. 次のコマンドを実行してクライアントとサーバ・プログラムを構築します。

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

ステップ 7: コンフィギュレーション・ファイルのロード

次のコマンドを実行してバイナリの TUXCONFIG コンフィギュレーション・ファイルをロードします。

```
tmloadcf -y ubbconfig
```

ステップ 8: DCE の構成

上記で説明したアプリケーション例を実行するための DCE エンティティの設定手順を次に示します。すべての文字が大文字で表される識別子は、使用する環境に応じてカスタマイズ化します。

- すでに独自の DCE プリンシパルを持っている場合は、MYGROUP や MYPRINCIPAL、または関連するアカウントを作成する必要はありません。
- この例では、cell_admin パスワードはデフォルトの -dce と想定していますが、このパスワードは必要に応じて変更できます。
- サーバは BEA Tuxedo 管理者として起動し、サーバ・キー・テーブルの読み取りが行える必要があるので、*SERVER_PRINCIPAL* は BEA Tuxedo 管理者の識別子と同じにします。

リスト B-5　DCE **構成**

```
$ 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
```

ステップ 9: コンフィギュレーションの起動

1. *SERVER_PRINCIPAL* (サーバ・キー・テーブルの所有者) としてログインします。
2. 次のコマンドを実行して DCE サーバを起動します。

   ```
   dceserver &
   ```

   DCE サーバは、要求を受け付ける前に "`Server ready`" というメッセージを表示します。
3. 次のコマンドを実行して BEA Tuxedo ATMI アプリケーションを起動します。

   ```
   tmboot -y
   ```

ステップ 10: クライアントの実行

変換する文字列を指定してクライアント・プログラムを実行できます。変換する文字列は省略できます。クライアント・プログラムは、まず文字列を大文字に変換し、この後小文字に変換します。

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

ステップ 11: 構成のシャットダウン

1. 次のコマンドを実行すると、アプリケーションをシャットダウンします。
   ```
   tmshutdown -y
   ```
2. DCE サーバを停止します。

ステップ 12: 作成ファイルのクリーンアップ

次のコマンドを入力し、作成したファイルを削除します。

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