BEA Tuxedo

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

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

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

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、C 言語を使用して BEA Tuxedo ATMI アプリケーションをプログラミング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 BEA Tuxedo プログラミングの概要」—BEA Tuxedo プログラミングの概要。分散アプリケーション・プログラミング、クライアント、サーバ、および BEA Tuxedo アプリケーション・トランザクション・モニタ・インターフェイス (ATMI) について説明します。
- 「第 2 章 プログラミング環境」—BEA Tuxedo プログラミング環境。BEA Tuxedo システムの設定、環境変数の設定、およびアプリケーションの起動と停止について説明します。
- 「第 3 章 型付きバッファの管理」—VIEW バッファ、FML バッファ、XML バッファなどの型付きバッファの管理および使用方法。
- 「第 4 章 クライアントのコーディング」—C 言語を使用して BEA Tuxedo クライアント・アプリケーションをコーディングおよびビルドする手順。クライアント・プロセスのサンプルが同梱されています。
- 「第 5 章 サーバのコーディング」—サービスの定義や宣言など、C 言語を使用して BEA Tuxedo サーバ・アプリケーションをコーディングおよびビルドする手順。
- 「第 6 章 クライアントおよびサーバへの要求 / 応答のコーディング」—要求 / 応答型のクライアントおよびサーバのコーディング手順。同期 / 非同期メッセージング、メッセージの優先順位の設定について説明します。
- 「第 7 章 会話型クライアントおよびサーバのコーディング」—会話型のクライアントおよびサーバのコーディング手順。アプリケーションへの参加、接続の確立、メッセージの送受信、および会話の終了について説明します。
- 「第 8 章 イベント・ベースのクライアントおよびサーバのコーディング」—イベント・ベースのクライアントおよびサーバのコーディング手順。任意通知型のメッセージおよびイベントの処理について説明します。
- 「第 9 章 グローバル・トランザクションのコーディング」—グローバル・トランザクションのコーディング手順。トランザクションの開始と終了について説明します。

- 「第 10 章 マルチスレッドおよびマルチコンテキスト・アプリケーションのプログラミング」―1 つのプロセスで同時に複数のタスクを実行するアプリケーションのコーディング手順。この章では、マルチスレッドおよびマルチコンテキストのアプリケーションをプログラミングする方法について説明します。マルチスレッド化されたアプリケーションでは、1 つのプロセスに複数の実行単位が含まれます。マルチコンテキスト化されたアプリケーションでは、1 つのプロセスがドメイン内で複数の接続を確立したり、複数のドメインに対して接続を確立できます。
- 「第 11 章 エラーの管理」―システム・エラーとアプリケーション・エラーに対する処理の方法。

対象読者

このマニュアルは、BEA Tuxedo 環境で C 言語を使用してアプリケーションをプログラミングするアプリケーション開発者を対象にしています。

このマニュアルは、BEA Tuxedo プラットフォームおよび C 言語のプログラミングについて理解してい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 BEA Tuxedo 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 環境で BEA Tuxedo /Q コンポーネントを使用する方法、およびメッセージ・キュー・アプリケーションを実装する方法についての関連情報が掲載されています。

- 『BEA Tuxedo C リファレンス』
- 『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 `compilation(5)` および `tuxenv(5)`

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`

規則	項目
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。
[ ]	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 BEA Tuxedo プログラミングの概要

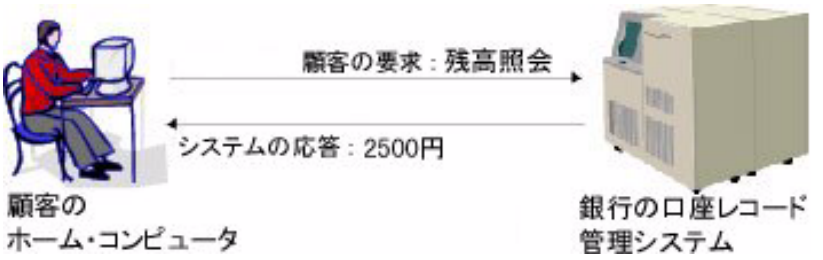
ここでは、次の内容について説明します。

- BEA Tuxedo 分散アプリケーションのプログラミング
- コミュニケーション・パラダイム
- BEA Tuxedo クライアント
- BEA Tuxedo サーバ
- BEA Tuxedo API:ATMI

BEA Tuxedo 分散アプリケーションのプログラミング

分散アプリケーションは、複数のハードウェア・システム上にあるソフトウェア・モジュールから構成され、これらのモジュールが互いに通信してアプリケーションのタスクが実行されます。たとえば、次の図に示すリモート・オンライン銀行業務システムの分散アプリケーションは、顧客のホーム・コンピュータで実行されるソフトウェア・モジュールと、すべての口座レコードを管理する銀行のコンピュータ・システムから構成されています。

図 1-1　分散アプリケーションの例 - オンライン銀行業務システム



たとえば、ログオンしてメニューからオプションを選択するだけで、残高を照会できます。この処理では、ローカル・ソフトウェア・モジュールが特別なアプリケーション・プログラミング・インターフェイス (API) 関数を使用して、リモート・ソフトウェア・モジュールと通信しています。

BEA Tuxedo 分散アプリケーション・プログラミング環境では、分散ソフトウェア・モジュール間で安全かつ信頼性の高い通信を行うために必要な API 関数が提供されています。BEA Tuxedo API は、アプリケーション・トランザクション・モニタ・インターフェイス (ATMI) と呼ばれます。

ATMI を使用すると、以下の操作を行うことができます。

- クライアントとサーバ間でのメッセージのやり取り。ネットワークを介して異機種マシン間でやり取りすることも可能です。
- クライアント命名およびセキュリティ機能の設定と使用。
- 複数の場所に格納されているデータを処理するトランザクションの定義と管理。
- データベース管理システム (DBMS) などのリソース・マネージャのオープンとクローズ。
- サービス要求のフロー管理、およびサービス要求を処理するサーバの可用性の管理。

コミュニケーション・パラダイム

次の表は、アプリケーション開発者が利用できる BEA Tuxedo のコミュニケーション・パラダイムを示しています。

表 1-1 コミュニケーション・パラダイム

パラダイム	説明
要求 / 応答型通信	要求 / 応答型通信では、あるソフトウェア・モジュールが 2 番目のソフトウェア・モジュールに要求を送り、その応答を受け取ります。要求元が応答を受け取るまで処理が行われずに待機する同期通信、または要求元が応答を待機する間も処理が継続される非同期通信の 2 種類があります。 このモードは、クライアント / サーバ相互作用とも呼ばれます。最初のソフトウェア・モジュールがクライアント、2 番目のソフトウェア・モジュールがサーバになります。 このパラダイムの詳細については、第 6 章の 1 ページ「クライアントおよびサーバへの要求 / 応答のコーディング」を参照してください。
会話型通信	会話型通信は要求 / 応答型通信に似ていますが、「会話」が終了する前に複数の要求や応答が発生します。会話型通信では、会話が切断されるまでクライアントとサーバで状態の情報が保持されます。クライアントとサーバ間でメッセージをやり取りする方法は、使用するアプリケーション・プロトコルによって決定されます。 通常、会話型通信はサーバからクライアントへの長い応答のバッファとして使用されます。 このパラダイムの詳細については、第 7 章の 1 ページ「会話型クライアントおよびサーバのコーディング」を参照してください。

パラダイム	説明
アプリケーション・キュー・ベースの通信	アプリケーション・キュー・ベースの通信では、遅延通信、つまり時間に依存しない通信が行われます。この通信では、クライアントとサーバがアプリケーション・キューを使用して通信します。BEA Tuxedo/Q 機能を使用すると、メッセージを永続的な記憶装置 (ディスク) や一時的な記憶装置 (メモリ) のキューに入れることができ、後で処理したり取り出すことができます。 アプリケーション・キュー・ベースの通信は、たとえば、メンテナンスのためにシステムをオフラインにしたときに要求をキューに登録する場合や、クライアントとサーバの処理速度が異なる場合に通信をバッファに格納する場合に使用します。 /Q 機能の詳細については、『BEA Tuxedo/Q コンポーネント』を参照してください。
イベント・ベースの通信	イベント・ベースの通信では、特別な状況 (イベント) が発生した場合に、クライアントやサーバがそれをクライアントに通知します。 イベントの通知には、次の 2 つの方法があります。 ■ 任意通知型イベント。クライアントやサーバからクライアントに直接通知される予測不能な状況です。 ■ ブローカ・イベント。予測不能な状況、または発生は予測できても発生時間を予測できない状況です。イベントは、メッセージの受信と転送を行う「無名ブローカ」プログラムによって、サーバからクライアントに間接的に通知されます。 イベント・ベースの通信は、BEA Tuxedo の イベント・ブローカ機能に基づいています。 このパラダイムの詳細については、第 8 章の 1 ページ「イベント・ベースのクライアントおよびサーバのコーディング」を参照してください。

BEA Tuxedo クライアント

BEA Tuxedo クライアントとは、ユーザの要求を収集し、その要求に対するサービスを提供するサーバにその要求を転送するソフトウェア・モジュールです。ほとんどのソフトウェア・モジュールは、BEA Tuxedo クライアントとして動作できます。その場合、ATMI クライアント初期化ルーチンを呼び出して、BEA Tuxedo アプリケーションに「参加」します。クライアントになると、メッセージ・バッファを割り当てて、サーバと情報を交換できるようになります。

クライアントは、ATMI 終了ルーチンを呼び出してアプリケーションから「分離」し、BEA Tuxedo システムにクライアントをトラッキングする必要がなくなったことを通知します。これにより、ほかの操作で BEA Tuxedo アプリケーションのリソースを使用できるようになります。

次は、基本的なクライアント・プロセスの処理を示す疑似コードです。

リスト 1-1　要求 / 応答型クライアントの疑似コード

```
main()
 {
      TPINIT バッファを割り当て
      バッファに初期クライアント ID を配置
      BEA Tuxedo アプリケーションのクライアントとして登録
      バッファを割り当て
      do while true {
            ユーザ入力データをバッファに格納
            サービス要求を送信
            応答を受信
            ユーザに応答を返信
      アプリケーションを終了
 }
```

この疑似コードに示したほとんどの処理は、ATMI 関数でインプリメントされます。ただし、ユーザ入力をバッファに格納する処理と、ユーザに応答を返す処理には、C 言語の関数を使用します。

バッファの割り当てでは、クライアント・プログラムによって型付きバッファと呼ばれるメモリ領域が BEA Tuxedo ランタイム・システムから割り当てられます。型付きバッファとは、C 構造体などの形式が指定されたメモリ・バッファです。

クライアントは、アプリケーションから分離する前に、サービス要求をいくつでも送受信できます。クライアントは、これらの要求を一連の要求 / 応答型呼び出しとして送信することができます。また、ある呼び出しから別の呼び出しに状態の情報を渡す必要がある場合は、会話型サーバに接続して要求を送ることもできます。どちらの方法でもクライアント・プログラムのロジックは同じですが、使用する ATMI 関数は異なります。

クライアントを実行するには、`buildclient` コマンドを実行してクライアントをコンパイルし、BEA Tuxedo ATMI および必要なライブラリとリンクします。`buildclient` コマンドの詳細については、第 4 章の 1 ページ「クライアントのコーディング」を参照してください。

BEA Tuxedo サーバ

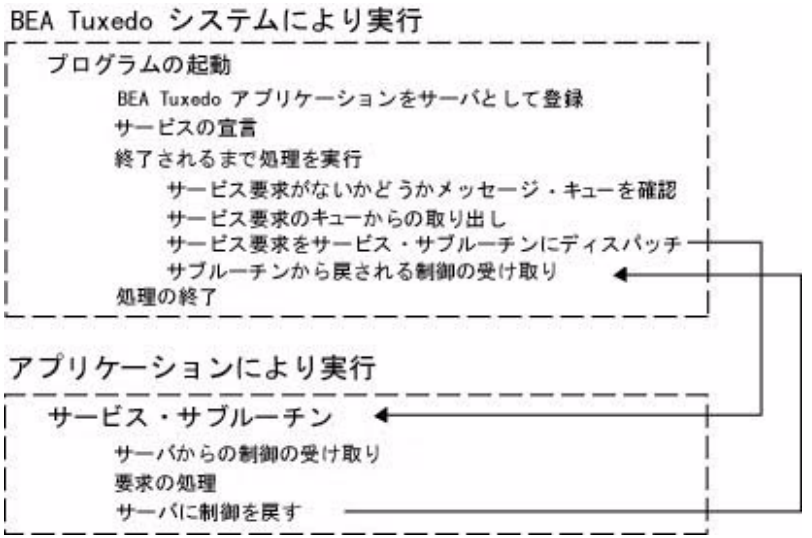
BEA Tuxedo サーバとは、クライアントに対して 1 つ以上のサービスを提供するプロセスです。サービスとは、クライアントが処理を要求する特定のビジネス・タスクです。サーバは、クライアントから要求を受け取り、それらを適切なサービス・サブルーチンにディスパッチします。

サーバの基本的な動作

サーバのビルドでは、サービス・サブルーチンと、BEA Tuxedo システムで提供される `main()` プロセスとがアプリケーションによって組み合わされます。この `main()` 関数はシステムによって提供され、定義済みの関数から構成されています。この関数は、サーバの初期化と終了を行ったり、バッファを割り当てて、要求を受信してサービス・ルーチンへのディスパッチを行ったりします。このすべての処理は、アプリケーションに透過的です。

次の図は、サーバとサービス・ルーチン間の相互作用を示す疑似コードです。

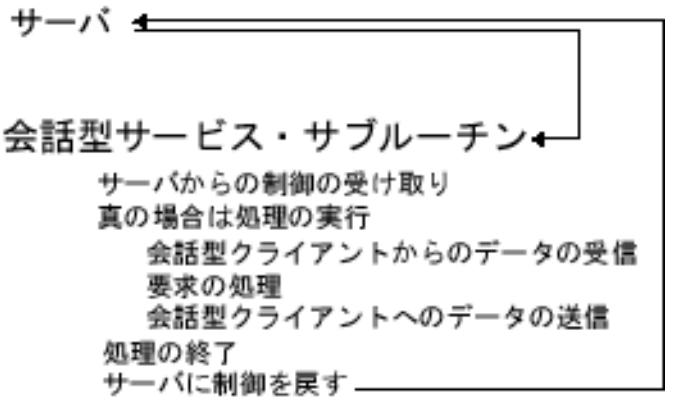
図 1-2 要求 / 応答型サーバとサービス・サブルーチンの疑似コード



初期化後、サーバはバッファの割り当てを行い、要求メッセージがメッセージ・キューに登録されるまで待機し、その要求をキューから取り出してサービス・サブルーチンに送り、要求を処理します。応答が必要な場合は、その応答は要求処理の一部と見なされます。

会話型パラダイムは、次の図の疑似コードで示すように、要求 / 応答型パラダイムとは多少異なります。

図 1-3　会話型サービス・サブルーチンの疑似コード



BEA Tuxedo システム提供の `main()` プロセスには、プロセスをサーバとして登録し、サービスを宣言し、バッファを割り当て、キューから要求メッセージを取り出すために必要なコードが記述されています。ATMI 関数は、要求を処理するサービス・サブルーチンで使用されます。サービス・サブルーチンのコンパイルとテストを行う準備ができたら、これらをサーバの `main()` とリンクして、実行可能サーバを生成します。その場合、`buildserver` コマンドを実行します。

要求元としてのサーバ

クライアントが複数のサービスや同じサービスを複数回要求する場合、サービスのサブセットを別のサーバに転送して実行することができます。その場合、サーバはクライアント、つまりサービスの要求元として動作します。つまり、クライアントとサーバの両方が要求元になることができます。ただし、クライアントは要求元にしかなれません。このようなモデルは、BEA Tuxedo の ATMI 関数を使用すると簡単にコーディングできます。

注記 要求 / 応答型サーバも、要求を別のサーバに転送できます。その場合、サーバはクライアント (要求元) としては動作しません。応答を必要としているのは、要求を転送したサーバではなく、元のクライアントだからです。

BEA Tuxedo API:ATMI

アプリケーションのロジックを記述する C 言語のほかに、アプリケーション・トランザクション・モニタ・インターフェイス (ATMI) を使用する必要があります。この ATMI は、アプリケーションと BEA Tuxedo システム間のインターフェイスになります。ATMI 関数は、オペレーティング・システム・コールに類似した C 言語の関数です。これらの関数により、必要なすべてのリソースも含め、BEA Tuxedo システムのトランザクション・モニタの下で動作するアプリケーション・モジュール間の通信がインプリメントされます。

ATMI は、リソースのオープンとクローズ、トランザクションの開始と終了、バッファの割り当てと解放、およびクライアントとサーバ間の通信のサポートなどの処理に使用されるコンパクトな関数セットです。次の表は、ATMI 関数をまとめたものです。各関数については、『BEA Tuxedo C リファレンス』を参照してください。

表 1-2 ATMI 関数

タスク	C 関数	処理内容	参照先
バッファ管理	tpalloc()	メッセージ・バッファを作成します。	第 3 章の 1 ページ「型付きバッファの管理」
	tprealloc()	メッセージ・バッファのサイズを変更します。	
	tptypes()	メッセージのタイプとサブタイプを取得します。	
	tpfree()	メッセージ・バッファを解放します。	
クライアントのメンバーシップ	tpchkauth()	認証が必要かどうかを確認します。	第 4 章の 1 ページ「クライアントのコーディング」
	tpinit()	アプリケーションに参加します。	
	tpterm()	アプリケーションから分離します。	

表 1-2 ATMI 関数

タスク	C 関数	処理内容	参照先
複数のアプリケーション・コンテキストの管理	tpgetctxt(3c)	現在のスレッドのコンテキストの識別子を取得します。	第 10 章の 1 ページ「マルチスレッドおよびマルチコンテキスト・アプリケーションのプログラミング」
	tpsetctxt(3c)	マルチコンテキスト・プロセスに現在のスレッドのコンテキストを設定します。	
サービスの登録と応答	tpsvrinit()	サーバを初期化します。	■ 第 5 章の 1 ページ「サーバのコーディング」 ■ 第 10 章の 1 ページ「マルチスレッドおよびマルチコンテキスト・アプリケーションのプログラミング」
	tpsvrdone()	サーバを終了します。	
	tpsvrthrinit()	個々のサーバ・スレッドを初期化します。	
	tpsvrthrdone()	個々のサーバ・スレッドの終了コードです。	
	tpreturn()	サービス関数を終了します。	
	tpforward()	要求を転送します。	
動的な宣言	tpadvertise()	サービス名を宣言します。	第 5 章の 1 ページ「サーバのコーディング」
	tpunadvertise()	サービス名の宣言を取り消します。	
メッセージの優先順位	tpgprio()	最後の要求の優先順位を取得します。	第 5 章の 1 ページ「サーバのコーディング」
	tpsprio()	次の要求の優先順位を設定します。	
要求 / 応答型通信	tpcall()	サービスへの同期要求 / 応答を開始します。	■ 第 5 章の 1 ページ「サーバのコーディング」 ■ 第 6 章の 1 ページ「クライアントおよびサーバへの要求 / 応答のコーディング」
	tpacall()	非同期要求を開始します。	
	tpgetrply()	非同期応答を受け取ります。	
	tpcancel()	非同期要求を取り消します。	

表 1-2 ATMI 関数

タスク	C 関数	処理内容	参照先
会話型通信	tpconnect()	サービスとの会話を開始します。	第 7 章の 1 ページ「会話型クライアントおよびサーバのコーディング」
	tpdiscon()	会話を異常終了します。	
	tpsend()	会話中にメッセージを送信します。	
	tprecv()	会話中にメッセージを受信します。	
高信頼性キュー	tpenqueue(3c)	メッセージをメッセージ・キューに登録します。	『BEA Tuxedo/Q コンポーネント』
	tpdequeue(3c)	メッセージをメッセージ・キューから取り出します。	
イベント・ベースの通信	tpnotify()	クライアントに任意通知型メッセージを送信します。	第 8 章の 1 ページ「イベント・ベースのクライアントおよびサーバのコーディング」
	tpbroadcast()	複数のクライアントにメッセージを送信します。	
	tpsetunsol()	任意通知型メッセージのコールバックを設定します。	
	tpchkunsol()	任意通知型メッセージの到着を確認します。	
	tppost()	イベント・メッセージをポストします。	
	tpsubscribe()	イベント・メッセージをサブスクライブします。	
	tpunsubscribe()	イベント・メッセージのサブスクリプションを削除します。	

表 1-2 ATMI 関数

タスク	C 関数	処理内容	参照先
トランザクション管理	tpbegin()	トランザクションを開始します。	第 9 章の 1 ページ「グローバル・トランザクションのコーディング」
	tpcommit()	現在のトランザクションをコミットします。	
	tpabort()	現在のトランザクションをロールバックします。	
	tpgetlev()	トランザクション・モードであるかどうかを確認します。	
	tpsuspend()	現在のトランザクションを一時停止します。	
	tpresume()	トランザクションを再開します。	
リソース管理	tpopen(3c)	リソース・マネージャをオープンします。	『BEA Tuxedo アプリケーションの設定』
	tpclose(3c)	リソース・マネージャをクローズします。	

表 1-2 ATMI 関数

タスク	C 関数	処理内容	参照先
セキュリティ	tpkey_open(3c)	デジタル署名の生成、メッセージの暗号化 / 暗号解読のためのキー・ハンドルをオープンします。	『BEA Tuxedo CORBA アプリケーションのセキュリティ機能』
	tpkey_getinfo(3c)	キー・ハンドルに対応付けられた情報を取得します。	
	tpkey_setinfo(3c)	キー・ハンドルに対応付けられた属性 (省略可能) を設定します。	
	tpkey_close(3c)	既にオープンされているハンドルをクローズします。	
	tpsign(3c)	デジタル署名を生成するための型付きメッセージ・バッファをマークします。	
	tpseal(3c)	暗号化エンベロープを生成するための型付きメッセージ・バッファをマークします。	
	tpenvelope(3c)	型付きメッセージ・バッファに対応付けられたデジタル署名と受信側の情報にアクセスします。	
	tpexport(3c)	型付きメッセージ・バッファをエクスポート可能でマシンに依存しない (外部化された) 文字列表現に変換します。	
	tpimport(3c)	外部化された文字列表現を型付きメッセージ・バッファに変換します。	

2　プログラミング環境

ここでは、次の内容について説明します。

- UBBCONFIG コンフィギュレーション・ファイルの更新
- 環境変数の設定
- 必要なヘッダ・ファイルのインクルード
- アプリケーションの起動と停止

UBBCONFIG コンフィギュレーション・ファイルの更新

アプリケーション管理者は、最初に UBBCONFIG コンフィギュレーション・ファイルにアプリケーションのコンフィギュレーションを定義します。プログラミング環境をカスタマイズするには、コンフィギュレーション・ファイルを作成するか更新します。

コンフィギュレーション・ファイルの作成または更新を行う場合は、次のガイドラインを参考にしてください。

- 既存のファイルをコピーして編集します。たとえば、サンプル・アプリケーション bankapp に付属するファイル ubbshm から作業を開始します。
- 複雑性を最小限にします。テストを行う場合は、共用メモリを使用する単一プロセッサ・システムとしてアプリケーションを設定します。データとしては、通常のオペレーティング・システムのファイルを使用します。
- コンフィギュレーション・ファイルの IPCKEY パラメータが、インストールされているシステムで使用されているほかのパラメータと競合しないようにします。詳細については、BEA Tuxedo アプリケーション管理者に確認し、『BEA Tuxedo アプリケーションの設定』を参照してください。
- UID および GID パラメータを設定し、定義したコンフィギュレーションが自分自身のものであることを示します。

- マニュアルを参照します。コンフィギュレーション・ファイルについては、『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 UBBCONFIG(5) で説明しています。

次の表は、プログラミング環境に影響する UBBCONFIG コンフィギュレーション・ファイルのパラメータを示しています。パラメータは機能別に分類されています。

表 2-1 プログラミング関連の UBBCONFIG パラメータ (機能別)

機能	パラメータ	セクション	説明
グローバル・リソースの制限	MAXSERVERS	RESOURCES	コンフィギュレーション内のサーバの最大数。この値を設定する場合は、すべてのサーバの MAX 値を考慮する必要があります。
	MAXSERVICES	RESOURCES	コンフィギュレーション内のサービスの最大総数。
データ依存型ルーティング	BUFTYPE	ROUTING	指定されたルーティング・エントリが有効であるデータ・バッファのタイプとサブタイプのリスト。
リンク・レベルの暗号化	MINENCRYPTBITS	NETWORK	プロセスで使用できる暗号化の最低レベル。
	MAXENCRYPTBITS	NETWORK	プロセスで使用できる暗号化の最高レベル。
ロード・バランシング	LDBAL	RESOURCES	ロード・バランシングを有効にするかどうかを示すフラグ。ロード・バランシングが有効になっている場合、BEA Tuxedo システムは要求の負荷をネットワークで分散します。
	NETLOAD	MACHINES	呼び出し元クライアントからの要求をリモートに送る場合に、サービスのロード・ファクタに追加される数値。リモート・サーバ上でローカル・サーバを選択する際の基準になります。ロード・バランシングが有効になっていること (LDBAL に Y が設定されていること) が必要です。
	LOAD	SERVICES	サービス・インスタンスに対応付けられた相対的なロード・ファクタ。デフォルトは「50」。

表 2-1 プログラミング関連の UBBCONFIG パラメータ (機能別) (続き)

機能	パラメータ	セクション	説明
セキュリティ	AUTHSVC	RESOURCES	システムに参加している各クライアントに対して、システムによって呼び出されるアプリケーション認証サービスの名前。
	SECURITY	RESOURCES	実行するアプリケーション・セキュリティの種類。
会話型通信	MAXCONV	RESOURCES	特定のマシン上で同時に関与できる会話の最大数。0 ～ 32,767 の値を指定します。SERVERS セクションに会話型サーバが定義されている場合、デフォルト値は 64 になります。それ以外の場合、デフォルト値は 1 になります。このパラメータに指定された値は、MACHINES セクションのマシンごとに上書きできます。
	CONV	SERVERS	会話型通信がサポートされているかどうかを示す値。このパラメータが N に設定されているか、値が指定されてない場合、サービスに対する tpconnect() の呼び出しは失敗します。
	MIN/MAX	SERVERS	tmboot(1) によって起動するサーバのオカレンスの最小数と最大数。指定がない場合には、MIN のデフォルト値は 1、MAX のデフォルト値は MIN となります。この 2 つのパラメータは、要求 / 応答型サーバに対しても使用できます。ただし、会話型サーバは、必要に応じて自動的に追加されます。そのため、MIN=1、MAX=10 と設定されている場合、tmboot によって最初に 1 つのサーバが起動します。そのサーバが提供するサービスに対して tpconnect() が呼び出されると、システムによって 2 番目のサーバが起動します。同じように、新しいサーバが上限の 10 まで起動します。

表 2-1 プログラミング関連の UBBCONFIG パラメータ (機能別) (続き)

機能	パラメータ	セクション	説明
トランザクション管理	AUTOTRAN	SERVICES	サービス・ルーチンでトランザクション・モードを開始するかどうかを示す値。このパラメータに Y が設定されている場合、別のプロセスから要求メッセージを受信すると、サービス・サブルーチンでトランザクションが自動的に開始します。
マルチスレッド・サーバ	MAXDISPATCHTHREADS	SERVERS	各サーバ・プロセスによってスレッドが追加された場合、同時にディスパッチされるスレッドの最大数。
	MINDISPATCHTHREADS	SERVERS	最初のサーバの起動時に開始されるサーバ・ディスパッチ・スレッドの数。

コンフィギュレーション・ファイルは、オペレーティング・システムのテキスト・ファイルです。このファイルを実際にシステムで使用する場合は、tmloadcf(1) を実行して、バイナリ・ファイルに変換する必要があります。

関連項目

- 『BEA Tuxedo アプリケーションの設定』
- 『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 UBBCONFIG(5)

環境変数の設定

アプリケーション管理者は、最初にアプリケーションの実行環境を定義する変数を設定します。これらの環境変数を設定するには、UBBCONFIG ファイルの MACHINES セクションで ENVFILE パラメータに値を指定します。詳細については、『BEA Tuxedo アプリケーションの設定』を参照してください。

アプリケーションのクライアント・ルーチンとサーバ・ルーチンに対して、既存の環境変数を更新したり、新しい変数を作成することができます。次の表は、よく使用される環境変数を示しています。変数は機能別に分類されています。

表 2-2 プログラミング関連の環境変数 (機能別)

機能	環境変数	定義内容	使用先
グローバル	TUXDIR	BEA Tuxedo システムのバイナリ・ファイルの場所。	BEA Tuxedo アプリケーション・プログラム
コンフィギュレーション	TUXCONFIG	BEA Tuxedo コンフィギュレーション・ファイルの場所。	BEA Tuxedo アプリケーション・プログラム
コンパイル	CC	C コンパイラを呼び出すコマンド。デフォルト値は cc です。	buildclient(1) コマンド、buildserver(1) コマンド
	CFLAGS	C コンパイラに渡すリンクのフラグ。このフラグは必須ではありません。	buildclient(1) コマンド、buildserver(1) コマンド
データ圧縮	TMCMPPRFM	圧縮レベル (1 〜 9)。	データ圧縮を行う BEA Tuxedo アプリケーション・プログラム
ロード・バランシング	TMNETLOAD	リモート・キューの負荷値に加算される数値。この値を指定すると、リモート・キューに実際より多くの作業負荷があるように設定できます。その結果、ロード・バランシングが有効になっていても、ローカル要求がリモート・キューよりローカル・キューに送られるようになります。	ロード・バランシングを実行する BEA Tuxedo アプリケーション・プログラム

表 2-2 プログラミング関連の環境変数 (機能別) (続き)

機能	環境変数	定義内容	使用先
バッファ管理	FIELDTBLS または FIELDTBLS32	FML および FML32 型付きバッファのフィールド・テーブル・ファイル名のカンマ区切りのリスト。FML および VIEW のみで必要です。	FML 型付きバッファ、FML32 型付きバッファ、FML VIEW
	FLDTBLDIR または FLDTBLDIR32	FML および FML32 のフィールド・テーブル・ファイルが検索されるディレクトリのコロン区切りのリスト。Windows 2000 では、セミコロンで区切られます。	FML 型付きバッファ、FML32 型付きバッファ、FML VIEW
	VIEWFILES または VIEWFILES32	VIEW および VIEW32 型付きバッファで使用できるファイル名のカンマ区切りのリスト。	VIEW 型付きバッファ、VIEW32 型付きバッファ
	VIEWDIR または VIEWDIR32	VIEW および VIEW32 ファイルが検索されるディレクトリのコロン区切りのリスト。Windows 2000 では、セミコロンで区切られます。	VIEW 型付きバッファ、VIEW32 型付きバッファ

UNIX 環境では、環境変数 PATH に $TUXDIR/bin を追加して、アプリケーションが BEA Tuxedo システムのコマンドに対する実行可能ファイルを見つけられるようにします。環境設定の詳細については、『BEA Tuxedo アプリケーションの設定』を参照してください。

関連項目

- 『BEA Tuxedo アプリケーションの設定』

必要なヘッダ・ファイルのインクルード

次の表は、#include 文を使用して、アプリケーション・プログラム内で指定する必要があるヘッダ・ファイルを示しています。これらのヘッダ・ファイルを使用すると、BEA Tuxedo システムと正しくインターフェイスをとることができるようになります

表 2-3 必要なヘッダ・ファイル

プログラム	ヘッダ・ファイル
すべての BEA Tuxedo アプリケーション・プログラム	BEA Tuxedo システムで提供される atmi.h ヘッダ・ファイル
FML 型付きバッファが使用されるアプリケーション・プログラム	■ 対応するフィールド・テーブル・ファイルから生成されるヘッダ・ファイル ■ BEA Tuxedo システムで提供される fml.h ヘッダ・ファイル
VIEW 型付きバッファが使用されるアプリケーション・プログラム	対応する VIEW 記述ファイルから生成されるヘッダ・ファイル

アプリケーションの起動と停止

アプリケーションを起動するには、tmboot(1) コマンドを実行します。このコマンドは、アプリケーションに必要な IPC 資源を確保し、管理プロセスとアプリケーション・サーバを起動します。

アプリケーションを停止するには、tmshutdown(1) コマンドを実行します。このコマンドは、サーバを停止させ、アプリケーションで使用されていた IPC 資源を解放します。ただし、データベースなどのリソース・マネージャで使用されていた資源は解放されません。

関連項目

■ 『BEA Tuxedo コマンド・リファレンス』の tmshutdown(1) と tmboot(1)

3　型付きバッファの管理

ここでは、次の内容について説明します。

- 型付きバッファの概要
- 型付きバッファの割り当て
- データのバッファへの格納
- 型付きバッファのサイズの変更
- バッファ・タイプの確認
- 型付きバッファの解放
- VIEW 型バッファ
- FML 型バッファ
- XML 型バッファ
- バッファのカスタマイズ

型付きバッファの概要

あるプロセスから別のプロセスにメッセージを送信する場合、メッセージ・データ用のバッファを割り当てておく必要があります。BEA Tuxedo システムのクライアントは、型付きバッファを使用してサーバにメッセージを送ります。型付きバッファとは、カテゴリ (タイプ) と、サブカテゴリ (サブタイプ) が定義されたメモリ領域です。サブカテゴリは必須ではありません。型付きバッファは、BEA Tuxedo システムでサポートされる分散プログラミング環境の基本要素の 1 つです。

なぜ「型付き」バッファを使用するのでしょうか。分散環境では、アプリケーションが異機種システムにインストールされ、異なるプロトコルを使用して複数のネットワーク間で通信が行われます。バッファ・タイプが異なると、初期化、メッセージの送受信、およびデータの符号化 / 復号化にそれぞれ別のルーチンが必要になります。各バッファに特定のタイプが割り当てられていると、プログラマが介在しなくても、そのタイプに対応するルーチンを自動的に呼び出すことができます。

以下に示す表は、BEA Tuxedo システムでサポートされる型付きバッファと、そのバッファが次の条件を満たしているかどうかを示しています。

- 自己記述型であるかどうか。つまり、バッファのデータ型と長さが、タイプとサブタイプ、およびそのデータからわかるかどうか。
- サブタイプが必要かどうか。
- 型付きバッファのデータ依存型ルーティングがシステムでサポートされているかどうか。
- 型付きバッファの符号化 / 復号化がシステムでサポートされているかどうか。

ルーティング関数が必要な場合は、アプリケーション・プログラマが用意します。

表 3-1 型付きバッファ

型付きバッファ	説明	自己記述型	サブタイプ	データ依存型ルーティング	符号化 / 復号化
CARRAY	未定義の文字配列。NULL 文字を含むことができます。BEA Tuxedo システムでは配列のセマンティクスは解釈されないので、この型付きバッファは曖昧なデータを処理する場合に使用します。CARRAY は自己記述型ではないので、転送時には長さを指定する必要があります。システムではバイトは解釈されないので、マシン間のメッセージ送信では符号化 / 復号化はサポートされません。	該当せず	該当せず	該当せず	該当せず
FML (フィールド操作言語)	BEA Tuxedo システム固有の自己記述型バッファ・タイプ。このバッファでは、各データ・フィールドに対応する識別子、オカレンス番号、場合によっては長さを示す値が格納されています。すべてのデータ操作は、ネイティブな C 言語の文ではなく FML 関数を使用して行われます。そのため、データからの独立性と柔軟性が確立されており、処理に多少のオーバーヘッドが生じても相殺されます。 FML バッファでは、フィールド識別子とフィールド長に 16 ビットが使用されます。 詳細については、第 3 章の 23 ページ「FML 型バッファ」を参照してください。	該当	該当せず	該当	該当

表 3-1 型付きバッファ (続き)

型付きバッファ	説明	自己記述型	サブタイプ	データ依存型ルーティング	符号化 / 復号化
FML32	FML と同じ。ただし、フィールド識別子とフィールド長に 32 ビットが使用されます。より長いフィールドを多数使用できるので、バッファ全体が大きくなります。 詳細については、第 3 章の 23 ページ「FML 型バッファ」を参照してください。	該当	該当せず	該当	該当
STRING	最後が NULL 文字で終了する文字配列。STRING バッファは、自己記述型です。そのため、異なる文字セットを使用するマシン間でデータを交換する場合は、BEA Tuxedo システムによってデータが自動的に変換されます。	該当	該当せず	該当せず	該当せず
VIEW	アプリケーションで定義される C 構造体。VIEW 型には、個々のデータ構造体を示すサブタイプが必要です。VIEW 記述ファイル (データ構造体のフィールドとタイプが定義されたファイル) は、VIEW 型バッファに定義されたデータ構造体を使用するクライアント・プロセスとサーバ・プロセスがアクセスできなければなりません。異なるタイプのマシン間でバッファがやり取りされる場合は、符号化 / 復号化が自動的に行われます。詳細については、第 3 章の 17 ページ「VIEW 型バッファ」を参照してください。	該当せず	該当	該当	該当

表 3-1 型付きバッファ (続き)

型付きバッファ	説明	自己記述型	サブタイプ	データ依存型ルーティング	符号化 / 復号化
VIEW32	VIEW と同じ。ただし、長さとカウントのフィールド長に 32 ビットが使用されます。より長いフィールドを多数使用できるので、バッファ全体が大きくなります。詳細については、第 3 章の 17 ページ「VIEW 型バッファ」を参照してください。	該当せず	該当	該当	該当
X_C_TYPE	VIEW と同じ。	該当せず	該当	該当	該当
X_COMMON	VIEW と同じ。ただし、このバッファ型は COBOL と C プログラム間の互換性を取るために使用されます。フィールド・タイプとして使用できるのは、short、long、および string だけです。	該当せず	該当	該当	該当

表 3-1 型付きバッファ (続き)

型付きバッファ	説明	自己記述型	サブタイプ	データ依存型ルーティング	符号化 / 復号化
XML	XML 文書は、次の要素から構成されます。 ■ 符号化された文字の並びで構成されるテキスト ■ 文書の論理構造の記述と、その構造に関する情報 XML 文書のルーティングは、エレメントの内容、またはエレメント・タイプと属性値に基づいて行われます。使用されている文字符号化は XML パーサによって判別されます。符号化が BEA Tuxedo のコンフィギュレーション・ファイル (UBBCONFIG(5) と DMCONFIG(5)) で使用されているネイティブな文字セット (US-ASCII または EBCDIC) と異なる場合、エレメントと属性名は US-ASCII または EBCDIC に変換されます。詳細については、第 3 章の 26 ページ「XML 型バッファ」を参照してください。	該当せず	該当せず	該当	該当せず
X_OCTET	CARRAY と同じ。	該当せず	該当せず	該当せず	該当せず

すべてのバッファ・タイプは、$TUXDIR/lib ディレクトリの tmtypesw.c ファイルに定義されています。クライアント・プログラムとサーバ・プログラムで認識されるバッファ・タイプは、tmtypesw.c に定義されているものだけです。tmtypesw.c ファイルを編集して、バッファ・タイプを追加したり削除できます。また、UBBCONFIG の BUFTYPE パラメータを使用して、特定のサービスで処理できるタイプとサブタイプを制限できます。

tmtypesw.c ファイルは、共用オブジェクトやダイナミック・リンク・ライブラリのビルドに使用されます。このオブジェクトは、BEA Tuxedo 管理サーバ、およびアプリケーション・クライアントとアプリケーション・サーバによって動的にロードされます。

関連項目

- 第 3 章の 17 ページ「VIEW 型バッファ」
- 第 3 章の 23 ページ「FML 型バッファ」
- 第 3 章の 26 ページ「XML 型バッファ」
- 『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 tuxtypes(5)
- 『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 UBBCONFIG(5)

型付きバッファの割り当て

初期状態では、バッファはクライアント・プロセスに対応付けられていません。クライアント・プロセスでメッセージを送信する場合、サポートされているタイプのバッファを割り当てて、メッセージを格納できるようにします。型付きバッファを割り当てるには、次に示すように tpalloc(3c) 関数を使用します。

```
char *
tpalloc(char *type, char *subtype, long size)
```

次の表は、tpalloc() 関数の引数を示しています。

表 3-2 tpalloc() 関数の引数

引数	説明
type	有効な型付きバッファを指すポインタ。
subtype	VIEW 記述ファイルで VIEW、VIEW32、または X_COMMON 型バッファに指定されているサブタイプの名前を指すポインタ。 *subtype* がない場合、この引数には NULL 値が使用されます。

引数	説明
size	バッファのサイズ。 BEA Tuxedo システムにより、CARRAY、X_OCTET、および XML を除くすべての型付きバッファに、自動的にデフォルトのバッファ・サイズが割り当てられます。自動的にバッファ・サイズが割り当てられないバッファ型には、バッファの終わりを識別できるようにサイズを指定する必要があります。 CARRAY、X_OCTET、および XML を除くすべての型付きバッファのサイズにゼロが指定されると、各型付きバッファに割り当てられているデフォルト値が使用されます。サイズが指定されている場合、指定されたサイズまたはそのバッファ型に割り当てられているデフォルト値のうちの大きい方が使用されます。 STRING、CARRAY、X_OCTET、および XML を除くすべての型付きバッファのデフォルト・サイズは 1024 バイトです。STRING 型バッファのデフォルト・サイズは 512 バイトです。CARRAY、X_OCTET、および XML にデフォルト値はありません。これらの型付きバッファには、ゼロより大きな値を指定する必要があります。サイズが指定されていない場合、引数にはデフォルト値の 0 が使用されます。その結果、tpalloc() 関数は NULL ポインタを返して、tperrno に TPEINVAL を設定します。

次の例で示すように、VIEW、VIEW32、X_C_TYPE、および X_COMMON 型バッファには *subtype* 引数が必要です。

リスト 3-1 VIEW 型バッファの割り当て

```
struct aud *audv;  /* VIEW 構造体 aud を指すポインタ */
 . . .
audv = (struct aud *)tpalloc("VIEW", "aud", sizeof(struct aud));
 . . .
```

次の例は、FML 型バッファの割り当て方法を示しています。*subtype* 引数に NULL 値が指定されていることに注目してください。

リスト 3-2　FML **型バッファの割り当て**

```
FBFR *fbfr;  /* FML バッファ構造体を指すポインタ */
 . . .
fbfr = (FBFR *)tpalloc("FML", NULL, Fneeded(f, v))
 . . .
```

次の例は、CARRAY 型バッファの割り当て方法を示しています。このバッファ型では、*size* 値を指定する必要があります。

リスト 3-3　CARRAY **型バッファの割り当て**

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

処理が正常に終了すると、tpalloc() 関数は char 型のポインタを返します。STRING と CARRAY 以外のタイプでは、ポインタを適切な C 構造体または FML ポインタにキャストする必要があります。

tpalloc() 関数は、エラーを検出すると NULL ポインタを返します。次は、エラー条件の例です。

- CARRAY、X_OCTET、または XML 型バッファの *size* 値が指定されていません。
- *type*、または VIEW の場合に、*subtype* が指定されていません。
- システムで認識されない値が *type* に指定されています。
- 割り当てを行う前に、アプリケーションに参加できませんでした。

全エラー・コードとその説明については『BEA Tuxedo C リファレンス』の tpalloc(3c) を参照してください。

次のコード例は、STRING 型バッファの割り当て方法を示しています。この例では、tpalloc() の *size* 引数の値として、デフォルト値が使用されています。

リスト 3-4 STRING 型バッファの割り当て

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

関連項目

- 第 3 章の 10 ページ「データのバッファへの格納」
- 第 3 章の 11 ページ「型付きバッファのサイズの変更」
- 『BEA Tuxedo C リファレンス』の tpalloc(3c)

データのバッファへの格納

バッファを割り当てると、そこにデータを格納できるようになります。

次の例では、3 つのメンバ (フィールド) を持つ VIEW 型バッファ aud を作成しています。3 つのメンバは、コマンド行がある場合はそこから取得される支店 ID の b_id、照会された残高を返す balance、およびステータス行にメッセージを返す ermsg です。audit を使用して特定の支店の残高が照会されると、b_id の値には要求の送信先の支店 ID、balance にはゼロ、ermsg には NULL 文字列がそれぞれ設定されます。

リスト 3-5 メッセージ・バッファへのデータの格納 - 例 1

```
...
audv = (struct aud *)tpalloc("VIEW", "aud", sizeof(struct aud));

/* aud 構造体を用意します。*/

audv->b_id = q_branchid;
audv->balance = 0.0;
(void)strcpy(audv->ermsg, "");
...
```

audit を使用して銀行全体の残高が照会されると、BAL サーバが呼び出されてサイトごとの残高合計が取得されます。個々のサイトで照会を行うには、代表支店 ID を指定します。代表支店 ID は、配列 sitelist[] に格納されます。したがって、aud 構造体は、次の例のように設定されます。

リスト 3-6　メッセージ・バッファへのデータの格納 - 例 2

```
...
/* aud 構造体を用意します。*/

audv->b_id = sitelist[i];/* このフィールドでルーティングを行います。*/
audv->balance = 0.0;
(void)strcpy(audv->ermsg, "");
...
```

STRING バッファにデータを格納するプロセスについては、第 3 章の 12 ページ「バッファのサイズ変更」のコード例を参照してください。

関連項目

- 第 3 章の 7 ページ「型付きバッファの割り当て」
- 第 3 章の 11 ページ「型付きバッファのサイズの変更」
- 『BEA Tuxedo C リファレンス』の tpalloc(3c)

型付きバッファのサイズの変更

tpalloc() で割り当てられたバッファのサイズを変更するには、次のように tprealloc(3c) 関数を使用します。

```
char*
tprealloc(char *ptr, long size)
```

次の表は、tprealloc() 関数の引数を示しています。

表 3-3 tprealloc() 関数の引数

引数	説明
ptr	サイズを変更するバッファを指すポインタ。このポインタは、tpalloc() の呼び出しで設定されます。それ以外の方法で設定されている場合、呼び出しは失敗し、tperrno(5) が TPEINVAL に設定されて、無効な引数がこの関数に渡されたことが示されます。
size	変更後のバッファ・サイズを指定する長精度型 (long)。

tprealloc() が返すポインタは、元のバッファと同じタイプのバッファを指します。バッファの場所が変わっている場合があるので、返されたポインタを使用してサイズが変更されたバッファを参照します。

バッファのサイズを増やすために tprealloc() 関数を呼び出すと、バッファに新しい領域が割り当てられます。バッファのサイズを減らすために tprealloc() 関数を呼び出すと、実際にバッファのサイズが変更されるのでなく、指定されたサイズを超える領域が使用不能になります。型付きバッファの内容には影響しません。未使用領域を解放するには、必要なサイズを持つ別のバッファにデータをコピーし、未使用領域を持つバッファを解放します。

tprealloc() 関数は、エラーが発生すると NULL ポインタを返し、tperrno に適切な値を設定します。エラー・コードについては、『BEA Tuxedo C リファレンス』の tpalloc(3c) を参照してください。

警告 tprealloc() 関数が NULL ポインタを返した場合、内容が変更されて有効ではなくなったバッファが渡された可能性があります。

次のコード例は、STRING バッファの領域を再度割り当てる方法を示しています。

リスト 3-7 バッファのサイズ変更

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

char instr[100];       /* 標準入力からの入力文字列を格納する文字列 */
long s1len, s2len;     /* 文字列 1 と文字列 2 の長さ */
char *s1ptr, *s2ptr;   /* 文字列 1 と文字列 2 のポインタ */

main()

{
```

```
  (void)gets(instr);              /* 標準入力から行を取得します。*/
  s1len = (long)strlen(instr)+1;  /* 長さを決定します。*/

  join application

  if ((s1ptr = tpalloc("STRING", NULL, s1len)) == NULL) {
    fprintf(stderr, "tpalloc failed for echo of:%s\n", instr);
    アプリケーションを終了
    exit(1);
  }
  (void)strcpy(s1ptr, instr);

  make communication call with buffer pointed to by s1ptr

  (void)gets(instr);       /* 標準入力から別の行を取得します。*/
  s2len = (long)strlen(instr)+1; /* その長さを決定します。*/
  if ((s2ptr = tprealloc(s1ptr, s2len)) == NULL) {
    fprintf(stderr, "tprealloc failed for echo of:%s\n", instr);
    free s1ptr's buffer
    アプリケーションを終了
    exit(1);
  }
  (void)strcpy(s2ptr, instr);

  make communication call with buffer pointed to by s2ptr
  . . .
```

次の例は前述の例を拡張したもので、発生する可能性があるエラー・コードを確認する方法を示しています。

リスト 3-8　tprealloc() **のエラーの確認**

```
. . .
if ((s2ptr=tprealloc(s1ptr, s2len)) == NULL)
  switch(tperrno) {
  case TPEINVAL:
    fprintf(stderr, "given invalid arguments\n");
    fprintf(stderr, "will do tpalloc instead\n");
      tpfree(s1ptr);
    if ((s2ptr=tpalloc("STRING", NULL, s2len)) == NULL) {
      fprintf(stderr, "tpalloc failed for echo of:%s\n", instr);
     アプリケーションを終了
    exit(1);
```

```
    }
    break;
  case TPEPROTO:
    fprintf(stderr, "tried to tprealloc before tpinit;\n");
    fprintf(stderr, "program error; contact product support\n");
     アプリケーションを終了
    exit(1);
  case TPESYSTEM:
    fprintf(stderr,
     "BEA Tuxedo error occurred; consult today's userlog file\n");
     アプリケーションを終了
    exit(1);
  case TPEOS:
    fprintf(stderr, "Operating System error %d occurred\n",Uunixerr);
    アプリケーションを終了
    exit(1);
  default:
    fprintf(stderr,
      "Error from tpalloc:%s\n", tpstrerror(tperrno));
     break;
}
```

関連項目

- 第 3 章の 7 ページ「型付きバッファの割り当て」
- 第 3 章の 10 ページ「データのバッファへの格納」
- 『BEA Tuxedo C リファレンス』の `tprealloc(3c)`

バッファ・タイプの確認

`tptypes(3c)` 関数は、バッファのタイプとサブタイプ (指定されている場合) を返します。次は `tptypes()` 関数の文法です。

```
long
tptypes(char *ptr, char *type, char *subtype)
```

次の表は、`tptypes()` 関数の引数を示しています。

表 3-4 ttptypes() 関数の引数

引数	説明
ptr	データ・バッファを指すポインタ。このポインタは、tpalloc() または tprealloc() の呼び出しで設定されます。NULL は指定できません。また、文字型にキャストする必要があります。この 2 つの条件が満たされていない場合、tptypes() 関数は引数が無効であることを示すエラーを返します。
type	データ・バッファのタイプを指すポインタ。*type* は文字型の値です。
subtype	データ・バッファのサブタイプ (指定されている場合) を指すポインタ。*subtype* は文字型の値です。VIEW、VIEW32、X_C_TYPE、および X_COMMON 以外のタイプの場合、返された *subtype* パラメータは NULL 文字列を含む文字配列を指します。

処理が正常に終了した場合、tptypes() 関数は長精度型 (long) でバッファの長さを返します。

エラーが発生した場合、tptypes() 関数は -1 を返し、tperrno(5) に対応するエラー・コードを設定します。エラー・コードについては、『BEA Tuxedo C リファレンス』の 「C 言語アプリケーション・トランザクション・モニタ・インターフェイスについて」、および tpalloc(3c) を参照してください。

処理の正常終了時に tptypes() 関数から返されたサイズ値を使用して、次の例に示すように、デフォルトのバッファ・サイズがデータを格納するのに十分な大きさかどうかを確認できます。

リスト 3-9 バッファ・サイズの取得

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

関連項目

- 第 3 章の 7 ページ「型付きバッファの割り当て」
- 『BEA Tuxedo C リファレンス』の tptypes(3c)

型付きバッファの解放

tpfree(3c) 関数は、tpalloc() で割り当てられたバッファ、または tprealloc() で再度割り当てられたバッファを解放します。次は tpfree() 関数の文法です。

```
void
tpfree(char *ptr)
```

tpfree() 関数の引数は、次の表に示す *ptr* だけです。

表 3-5 tpfree() 関数の引数

引数	説明
ptr	データ・バッファを指すポインタ。このポインタは、tpalloc() または tprealloc() の呼び出しで設定されます。NULL は指定できません。また、文字型にキャストする必要があります。この 2 つの条件が満たされていない場合、この関数は何も解放しないか、エラー条件を通知しないで制御を戻します。

tpfree() を使用して FML32 バッファを解放するとき、ルーチンは埋め込まれたバッファをすべて再帰的に解放して、メモリ・リークの発生を防ぎます。埋め込みバッファが解放されないようにするには、対応するポインタに NULL を指定してから tpfree() ルーチンを発行します。*ptr* が NULL の場合、何も処理は行われません。

次のコード例は、tpfree() 関数を使用してバッファを解放する方法を示しています。

リスト 3-10 バッファの解放

```
struct aud *audv;  /* VIEW 構造体 aud を指すポインタ */
. . .
audv = (struct aud *)tpalloc("VIEW", "aud", sizeof(struct aud));
```

```
. . .
tpfree((char *)audv);
```

関連項目

- 第 3 章の 7 ページ「型付きバッファの割り当て」
- 第 3 章の 11 ページ「型付きバッファのサイズの変更」
- 『BEA Tuxedo C リファレンス』の `tpfree(3c)`

VIEW 型バッファ

VIEW 型バッファには 2 種類あります。1 つは FML VIEW で、FML バッファから生成される C 構造体です。もう 1 つは、単なる非依存型の C 構造体です。

FML バッファを C 構造体に変換して再び元に戻す (FML VIEW 型バッファを使用する) のは、FML 型バッファでの処理にはオーバーヘッドが生じるからです。つまり、FML 型バッファにはデータ独立性と使いやすさがある代わりに、FML 関数を呼び出すためにオーバーヘッドが生じます。C 構造体は、柔軟性の点では劣りますが、バッファ・データの時間がかかる処理に適しています。大量のデータを操作する場合、フィールド化バッファのデータを C 構造体に転送し、通常の C 関数を使用してそのデータを処理し、格納やメッセージ転送を行うためにそのデータを FML 型バッファに戻すと、パフォーマンスを向上させることができます。

FML 型バッファと FML ファイルの変換の詳細については、『BEA Tuxedo FML リファレンス』を参照してください。

VIEW 型バッファを使用するには、次の手順に従います。

- 適切な環境変数を設定します。
- 各構造体を VIEW 記述ファイルに定義します。
- BEA Tuxedo VIEW コンパイラの `viewc` を使用して、VIEW 記述ファイルをコンパイルします。生成されたヘッダ・ファイルをアプリケーション・プログラムの `#include` 文に指定します。

VIEW 型バッファの環境変数の設定

アプリケーションで VIEW 型バッファを使用するには、次の環境変数を設定します。

表 3-6 VIEW 型バッファの環境変数

環境変数	説明
FIELDTBLS または FIELDTBLS32	FML または FML32 型バッファのフィールド・テーブル・ファイル名のカンマ区切りのリスト。FML VIEW 型のみで必要です。
FLDTBLDIR または FLDTBLDIR32	FML または FML32 型バッファのフィールド・テーブル・ファイルが検索されるディレクトリのコロン区切りのリスト。Microsoft Windows では、セミコロンで区切られます。FML VIEW 型のみで必要です。
VIEWFILES または VIEWFILES32	VIEW または VIEW32 記述ファイルに使用されるファイル名のカンマ区切りのリスト。
VIEWDIR または VIEWDIR32	VIEW または VIEW32 ファイルが検索されるディレクトリのコロン区切りのリスト。Microsoft Windows では、セミコロンで区切られます。

VIEW 記述ファイルの作成

VIEW 型バッファを使用するには、VIEW 記述ファイルに C 言語のレコードを定義する必要があります。VIEW 記述ファイルには、各エントリの VIEW、および C 構造体のマッピングと FML 変換パターンを記述した VIEW が定義されています。VIEW の名前は、C 言語の構造体の名前に相当します。

VIEW 記述ファイルの各構造体は、次の形式で定義します。

```
$ /* VIEW 構造体 */
  VIEW viewname
  type    cname     fbname   count   flag    size    null
```

次の表は、VIEW 記述ファイルに指定する必要がある各 C 構造体のフィールドを示しています。

表 3-7 VIEW 記述ファイルのフィールド

フィールド	説明
type	フィールドのデータ型。short、long、float、double、char、string、または carray を指定できます。
cname	C 構造体のフィールド名。
fbname	FML から VIEW、または VIEW から FML への変換関数を使用する場合、対応する FML 名をこのフィールドに指定する必要があります。このフィールド名は、FML フィールド・テーブル・ファイルにも必要です。FML に依存しない VIEW には必要ありません。
count	フィールドの出現回数。
flag	次のいずれかのオプション・フラグを指定します。 ■ P—NULL 値の解釈を変更します。 ■ S—フィールド化バッファから構造体に一方向のマッピングを行います。 ■ F—構造体からフィールド化バッファへの一方向マッピングを行います。 ■ N—ゼロ方向のマッピングを行います。 ■ C—連想カウント・メンバ (ACM) に追加フィールドを生成します。 ■ L—STRING および CARRAY に転送されるバイト数を保持します。
size	STRING または CARRAY 型レコードの最大長を指定します。それ以外のバッファ・タイプでは、このフィールドは無視されます。

表 3-7 VIEW 記述ファイルのフィールド (続き)

フィールド	説明
null	ユーザ定義の NULL 値、または - の場合はフィールドのデフォルト値。VIEW 型バッファで使用される NULL 値は、空の C 構造体を示します。 数値型の場合、デフォルトの NULL 値は 0 (dec_t の場合は 0.0) になります。文字型の場合、デフォルトの NULL 値は '\0' になります。STRING 型と CARRAY 型の場合、デフォルトの NULL 値は " " になります。 エスケープ文字として使用されている定数も、NULL 値の指定に使用できます。VIEW コンパイラで認識されるエスケープ定数は、\ddd (d は 8 進数)、\0、\n、\t、\v、\r、\f、\\、\'、および \" です。 STRING、CARRAY、および char 型の NULL 値は、二重引用符または一重引用符で囲みます。VIEW コンパイラでは、ユーザ定義の NULL 値でエスケープされていない引用符は使用できません。 VIEW メンバ記述の NULL フィールドにキーワード NONE を指定することもできます。このキーワードは、そのメンバの NULL 値がないことを示します。文字列メンバおよび文字配列メンバのデフォルト値の最大サイズは、2660 文字です。詳細については、『BEA Tuxedo FML リファレンス』を参照してください。

行頭に # または $ 文字を付けてコメント行を挿入できます。行頭に $ が挿入されたコード行は、.h ファイルに出力されます。

次のコード例は、FML バッファに基づく VIEW 記述サンプル・ファイルの一部です。このコード例では、*fbname* フィールドを指定する必要があり、この値は対応するフィールド・テーブル・ファイルの値と一致していなければなりません。CARRAY1 フィールドのオカレンス・カウントが 2 に設定されていること、C フラグが設定されて追加のカウント・エレメントの作成が定義されていることに注目してください。また、L フラグが設定され、アプリケーションが CARRAY1 フィールドを格納するときの文字数を示す長さエレメントが定義されています。

リスト 3-11 FML VIEW の VIEW 記述ファイル

```
$ /* VIEW 構造体 */
  VIEW MYVIEW
  #type   cname     fbname   count  flag   size     null
  float   float1    FLOAT1    1      -      -        0.0
  double  double1   DOUBLE1   1      -      -        0.0
  long    long1     LONG1     1      -      -        0
```

```
  short   short1    SHORT1   1      -      -       0
  int     int1      INT1     1      -      -       0
  dec_t   dec1      DEC1     1      -      9.16    0
  char    char1     CHAR1    1      -      -       '\0'
  string  string1   STRING1  1      -      20      '\0'
  carray  carray1   CARRAY1  2      CL     20      '\0'
  END
```

次のコード例は、同じ VIEW 記述ファイルで非依存型 VIEW のものを示しています。

リスト 3-12　非依存型 VIEW の VIEW 記述ファイル

```
$ /* VIEW データ構造体 */
  VIEW MYVIEW
  #type   cname   fbname  count   flag    size    null
  float   float1  -       1       -       -       -
  double  double1 -       1       -       -       -
  long    long1   -       1       -       -       -
  short   short1  -       1       -       -       -
  int     int1    -       1       -       -       -
  dec_t   dec1    -       1       -       9,16    -
  char    char1   -       1       -       -       -
  string  string1 -       1       -       20      -
  carray  carray1 -       2       CL      20      -
  END
```

この形式は FML 依存型 VIEW と同じです。ただし、*fbname* フィールドと *null* フィールドには意味がなく、viewc コンパイラで無視されます。これらのフィールドには、プレースホルダとしてダッシュ (-) などの値を挿入する必要があります。

VIEW コンパイラの実行

VIEW 型バッファをコンパイルするには、引数として VIEW 記述ファイルの名前を指定して viewc コマンドを実行します。非依存型 VIEW を指定するには、-n オプションを使用します。生成される出力ファイルを書き込むディレクトリを指定することもできます (省略可能)。デフォルトでは、出力ファイルはカレント・ディレクトリに書き込まれます。

たとえば、FML 依存型 VIEW をコンパイルするには、次のようにコンパイラを実行します。

```
viewc myview.v
```

注記 VIEW32 型バッファをコンパイルするには、viewc32 コマンドを実行します。

非依存型 VIEW の場合、コマンド行で次のように -n オプションを指定します。

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

viewc コマンドでは、次が出力されます。

- MYVIEW.cbl など、1 つ以上の COBOL の COPY ファイル
- アプリケーション・プログラムで使用される構造体定義が記述されたヘッダ・ファイル
- myview.V など、バイナリ・バージョンのソース記述ファイル

  注記 Microsoft Windows など、大文字と小文字が区別されないプラットフォームでは、バイナリ・バージョンのソース記述ファイル名には、拡張子 vv (myview.vv など) が使用されます。

次のコード例は、viewc によって生成されるヘッダ・ファイルを示しています。

リスト 3-13 VIEW **コンパイラによって生成されるヘッダ・ファイル**

```
struct MYVIEW {
  float   float1;
  double  double1;
  long    long1;
  short   short1;
  int     int1;
  dec_t   dec1;
  char    char1;
  char    string1[20];
  unsigned short L_carray1[2];    /* carray1 の長さを挿入する配列 */
  short   C_carray1;              /* carray1 のカウント */
  char    carray1[2][20];
};
```

FML 依存型と FML 非依存型 VIEW にも、同じヘッダ・ファイルが生成されます。

クライアント・プログラムまたはサービス・サブルーチンで VIEW 型バッファを使用するには、アプリケーションの #include 文にヘッダ・ファイルを指定する必要があります。

関連項目

- 第 3 章の 23 ページ「FML 型バッファ」
- 第 3 章の 26 ページ「XML 型バッファ」
- 『BEA Tuxedo コマンド・リファレンス』の viewc、viewc32(1)

FML 型バッファ

FML 型バッファを使用するには、次の手順に従います。

- 適切な環境変数を設定します。
- FML フィールド・テーブルに、使用する可能性があるフィールドを記述します。
- FML ヘッダ・ファイルを作成して、アプリケーションの #include 文に指定します。

FML 関数は、フィールド化バッファから C 構造体への変換、またその逆の変換など、型付きバッファを操作する場合に使用します。これらの関数を使用すると、データ構造やデータの格納状態がわからなくても、データ値にアクセスしたり更新できます。FML 関数の詳細については、『BEA Tuxedo FML リファレンス』を参照してください。

FML 型バッファの環境変数の設定

アプリケーション・プログラムで FML 型バッファを使用するには、次の環境変数を設定する必要があります。

表 3-8 FML 型バッファの環境変数

環境変数	説明
FIELDTBLS または FIELDTBLS32	FML または FML32 型バッファのフィールド・テーブル・ファイル名のカンマ区切りのリスト。
FLDTBLDIR または FLDTBLDIR32	FML または FML32 型バッファのフィールド・テーブル・ファイルが検索されるディレクトリのコロン区切りのリスト。Microsoft Windows では、セミコロンで区切られます。

フィールド・テーブル・ファイルの作成

FML 型バッファや FML 依存型 VIEW を使用する場合は、常にフィールド・テーブル・ファイルが必要です。フィールド・テーブル・ファイルは、FML 型バッファのフィールドの論理名をそのフィールドを一意に識別する文字列にマッピングします。

FML フィールド・テーブルの各フィールドは、次の形式で定義します。

```
$ /* FML 構造体 */
   *base value
   name     number      type    flags    comments
```

次の表は、FML フィールド・テーブルに指定する必要がある FML フィールドを示しています。

表 3-9 フィールド・テーブル・ファイルのフィールド

フィールド	説明
*base *value*	後続のフィールド番号をオフセットするためのベース値。関連するフィールドのセットを簡単にグループ分けし、番号を付け直すことができるようになります。*base オプションを使用すると、フィールド番号を再利用できます。16 ビットのバッファの場合、ベース値とそれに関連する番号を加算した値が、100 以上 8191 未満でなければなりません。このフィールドは省略可能です。 **注記** BEA Tuxedo システムでは、フィールド番号 1 ～ 100 と 6,000 ～ 7,000 は、内部使用のために予約されています。FML ではフィールド番号 101 ～ 8,191、FML32 ではフィールド番号 101 ～ 33、554、および 431 がアプリケーション定義のフィールド用に使用できます。
name	フィールドの識別子。この値は 30 文字以下の文字列で、英数字と下線文字だけを指定できます。
rel-number	フィールドの相対数値。現在のベース値が指定されている場合、この値は現在のベース値に加算されて、フィールド番号が計算されます。
type	フィールドのタイプ。指定できるのは、char、string、short、long、float、double、または carray です。
flag	将来使用するために予約されたフィールド。プレースホルダとしてダッシュ (-) を挿入します。
comment	コメント (省略可能)。

すべてのフィールドは省略可能です。また、複数個使用できます。

次のコード例は、FML 依存型 VIEW の例で使用されるフィールド・テーブル・ファイルを示しています。

リスト 3-14　FML VIEW のフィールド・テーブル・ファイル

```
# name        number    type      flags    comments
  FLOAT1      110       float     -        -
  DOUBLE1     111       double    -        -
  LONG1       112       long      -        -
  SHORT1      113       short     -        -
  INT1        114       long      -        -
  DEC1        115       string    -        -
  CHAR1       116       char      -        -
  STRING1     117       string    -        -
  CARRAY1     118       carray    -        -
```

FML ヘッダ・ファイルの作成

クライアント・プログラムやサービス・サブルーチンで FML 型バッファを使用するには、FML ヘッダ・ファイルを作成して、アプリケーションの #include 文にそのヘッダ・ファイルを指定する必要があります。

フィールド・テーブル・ファイルから FML ヘッダ・ファイルを作成するには、mkfldhdr(1) コマンドを使用します。たとえば、myview.flds.h というファイルを作成するには、次のコマンドを入力します。

```
mkfldhdr myview.flds
```

FML32 型バッファの場合は、mkfldhdr32 コマンドを使用します。

次のコード例は、mkfldhdr コマンドによって作成される myview.flds.h ヘッダ・ファイルを示しています。

リスト 3-15　myview.flds.h ヘッダ・ファイル

```
/*       fname    fldid            */
/*       -----    -----            */

#define  FLOAT1    ((FLDID)24686)   /* 番号 :110 タイプ :float  */
#define  DOUBLE1   ((FLDID)32879)   /* 番号 :111 タイプ :double */
```

```
#define  LONG1     ((FLDID)8304)    /* 番号:112 タイプ:long   */
#define  SHORT1    ((FLDID)113)     /* 番号:113 タイプ:short  */
#define  INT1      ((FLDID)8306)    /* 番号:114 タイプ:long   */
#define  DEC1      ((FLDID)41075)   /* 番号:115 タイプ:string */
#define  CHAR1     ((FLDID)16500)   /* 番号:116 タイプ:char   */
#define  STRING1   ((FLDID)41077)   /* 番号:117 タイプ:string */
#define  CARRAY1   ((FLDID)49270)   /* 番号:118 タイプ:carray */
```

アプリケーションの #include 文に新しいヘッダ・ファイルを指定します。ヘッダ・ファイルがインクルードされると、シンボリック名でフィールドを参照できるようになります。

関連項目

- 第 3 章の 17 ページ「VIEW 型バッファ」
- 第 3 章の 26 ページ「XML 型バッファ」
- 『BEA Tuxedo コマンド・リファレンス』の mkfldhdr、mkfldhdr32(1)

XML 型バッファ

XML 型バッファを使用すると、BEA Tuxedo アプリケーションで XML を使用して、アプリケーション内やアプリケーション間でデータを交換できるようになります。BEA Tuxedo アプリケーションでは、単純な XML 型バッファの送受信や、それらのバッファを適切なサーバにルーティングできます。解析など、XML 文書のすべての処理ロジックはアプリケーション側にあります。

XML 文書は、次の要素から構成されます。

- 文書のテキストを符号化した文字の並び
- 文書の論理構造の記述と、その構造に関する情報

XML 型バッファのプログラミング・モデルは、CARRAY 型バッファのモデルと類似しています。つまり、tpalloc() 関数を使用して、バッファの長さを指定する必要があります。最大 4 GB の XML 文書がサポートされます。

イベント処理で行われるフォーマット処理とフィルタ処理は、STRING 型バッファが使用されている場合はサポートされますが、XML 型バッファではサポートされません。そのため、XML 型バッファのバッファ・タイプ・スイッチ内の _tmfilter 関数と _tmformat 関数のポインタは、NULL に設定されます。

BEA Tuxedo システムの XML パーサは、次の操作を行います。

- 文字符号化の自動検出
- 文字コードの変換
- データ・エレメントの内容と属性値の検出
- データ型の変換

XML 型バッファでは、データ依存型ルーティングがサポートされています。XML 文書のルーティングは、エレメントの内容、またはエレメントのタイプと属性値に基づいて行われます。使用される文字符号化は XML パーサによって判別されます。符号化が BEA Tuxedo のコンフィギュレーション・ファイル (UBBCONFIG と DMCONFIG) で使用されるネイティブな文字セット (US-ASCII または EBCDIC) と異なる場合、エレメントと属性名は US-ASCII または EBCDIC に変換されます。

XML 文書には、ルーティング用に設定する属性を含めなければなりません。属性がルーティング基準として設定されていても XML 文書に含まれていない場合、ルーティング処理は失敗します。

エレメントの内容と属性値は、ルーティング・フィールド値の構文とセマンティクスに従っていることが必要です。また、ルーティング・フィールド値のタイプも指定しなければなりません。XML でサポートされるのは文字データだけです。範囲フィールドが数値の場合、そのフィールドの内容や値はルーティング処理時に数値に変換されます。

関連項目

- 第 3 章の 17 ページ「VIEW 型バッファ」
- 第 3 章の 23 ページ「FML 型バッファ」

バッファのカスタマイズ

BEA Tuxedo システムで提供されるバッファ・タイプでは、アプリケーションのニーズが満たされない場合があります。たとえば、アプリケーションがフラットではないデータ構造体、つまり SQL データベースの問い合わせのための解析ツリーなど、ほかのデータ構造体へのポインタを持つデータ構造を扱う場合があります。アプリケーション固有の要件に対応するために、BEA Tuxedo システムではカスタム・バッファがサポートされています。

バッファをカスタマイズするには、次の特性を理解しておきます。

表 3-10 カスタム・バッファ・タイプの特性

特性	説明
バッファ・タイプ	バッファ・タイプの名前。8 文字以内の文字列で指定します。
バッファ・サブタイプ	バッファ・サブタイプの名前。16 文字以内の文字列で指定します。サブタイプは、特定のタイプのバッファに必要となる処理の違いを示すために使用されます。サブタイプ値としてワイルドカード文字 (*) を指定すると、同じ汎用ルーチンを使用して、指定されたタイプのすべてのバッファが処理されます。サブタイプが定義されているバッファは、リスト内でワイルドカードより前に置く必要があります。置かないと、処理が正常に行われません。
デフォルト・サイズ	対応するバッファ・タイプが割り当てまたは再割り当てされるときの最大サイズ。バッファ・タイプにゼロより大きい適切な値が設定されている場合、バッファを割り当てや再割り当てする際に、バッファ・サイズにゼロを指定できます。

次の表は、各バッファ・タイプに指定する必要があるルーチンを示しています。特定のルーチンが必要ない場合は、NULL ポインタを指定します。必要に応じて、BEA Tuxedo システムでデフォルトの処理が行われます。

表 3-11 カスタム・バッファ・タイプのルーチン

ルーチン	説明
バッファの初期化	新しく割り当てられた型付きバッファを初期化します。
バッファの再初期化	型付きバッファを再初期化します。このルーチンは、バッファが新しいサイズで再割り当てされたときに呼び出されます。
バッファの非初期化	型付きバッファを非初期化します。このルーチンは、型付きバッファが解放される直前に呼び出されます。
バッファの送信前処理	型付きバッファを送るための前処理を行います。このルーチンは、メッセージとして型付きバッファを別のクライアントやサーバに送信する前に呼び出されます。転送されるデータの長さが返されます。
バッファの送信後処理	型付きバッファを元の状態に戻します。このルーチンは、メッセージの送信後に呼び出されます。
バッファの受信後処理	アプリケーションで受信された型付きバッファを準備します。アプリケーション・データの長さが返されます。
符号化 / 複合化	バッファ・タイプで必要なすべての符号化と復号化を行います。入力バッファと出力バッファ、およびその長さと共に、符号化や復号化の要求がルーチンに渡されます。符号化に使用される形式はアプリケーションで決定され、ほかのルーチンと同じように、バッファ・タイプによって異なる場合があります。
ルーティング	ルーティング情報を指定します。このルーチンは、型付きバッファ、バッファのデータ長、管理者が設定した論理的なルーティング名、およびターゲット・サービスを指定して呼び出されます。この情報に基づいて、アプリケーションでメッセージの送信先サーバ・グループが選択されるか、またはメッセージが不要であることが決定されます。
フィルタ処理	フィルタ情報を指定します。このルーチンは、型付きバッファに対する式の評価するときに呼び出され、合致したかどうかを返します。型付きバッファが `VIEW` または `FML` の場合、`FML` 論理式が使用されます。イベント・ブローカは、このルーチンを使用してイベントが合致しているかどうかを評価します。
フォーマット処理	型付きバッファの印字可能な文字列を指定します。

独自のバッファ・タイプの定義

領域の割り当てと解放、メッセージの送受信など、バッファを操作するコードを記述するのはアプリケーション・プログラマです。デフォルトのバッファ・タイプではアプリケーションのニーズを満たすことができない場合、ほかのバッファ・タイプを定義し、新しいルーチンを記述してバッファ・タイプ・スイッチに組み込むことができます。

ほかのバッファ・タイプを定義するには、次の手順に従います。

1. 必要なスイッチ・エレメント・ルーチンのコードを記述します。
2. `tm_typesw` に、新しいタイプとバッファ管理モジュールの名前を追加します。
3. 新しい共用オブジェクトまたは DLL をビルドします。共用オブジェクトまたは DLL には、更新されたバッファ・タイプ・スイッチとそれに対応する関数が含まれていなければなりません。
4. 新しい共用オブジェクトまたは DLL をインストールして、すべてのサーバ、クライアント、および BEA Tuxedo システムで提供される実行可能ファイルが実行時に動的にロードされるようにします。

静的ライブラリが使用されるアプリケーションで、カスタム・バッファ・タイプ・スイッチを使う場合は、カスタム・サーバをビルドして、新しいタイプ・スイッチにリンクする必要があります。詳細については、`buildwsh` (1)、`TMQUEUE` (5)、`TMQFORWARD` (5) を参照してください。

ここでは、前述の手順に従って、共用オブジェクトまたは DLL 環境で新しいバッファ・タイプを定義します。その前に、BEA Tuxedo システム・ソフトウェアに提供されているバッファ・スイッチを参照してみます。次のコード例は、システムに提供されているスイッチです。

リスト 3-16 デフォルトのバッファ・タイプ・スイッチ

```
#include <stdio.h>
#include <tmtypes.h>

/* バッファ・タイプ・スイッチの初期化 */
static struct tmtype_sw_t tm_typesw[] = {
{
"CARRAY",          /* type */
"",                /* subtype */
0                  /* dfltsize */
},
{
"STRING",          /* type */
"",                /* subtype */
```

```
512,                /* dfltsize */
NULL,               /* initbuf */
NULL,               /* reinitbuf */
NULL,               /* uninitbuf */
_strpresend,        /* presend */
NULL,               /* postsend */
NULL,               /* postrecv */
_strencdec,         /* encdec */
NULL,               /* route */
NULL,               /* filter */
NULL                /* format */
},
{
"FML",              /* type */
"",                 /* subtype */
1024,               /* dfltsize */
_finit,             /* initbuf */
_freinit,           /* reinitbuf */
_funinit,           /* uninitbuf */
_fpresend,          /* presend */
_fpostsend,         /* postsend */
_fpostrecv,         /* postrecv */
_fencdec,           /* encdec */
_froute,            /* route */
_ffilter,           /* filter */
_fformat            /* format */
},
{
"FML32",            /* type */
"",                 /* subtype */
1024,               /* dfltsize */
_finit32,           /* initbuf */
_freinit32,         /* reinitbuf */
_funinit32,         /* uninitbuf */
_fpresend32,        /* presend */
_fpostsend32,       /* postsend */
_fpostrecv32,       /* postrecv */
_fencdec32,         /* encdec */
_froute32,          /* route */
_ffilter32,         /* filter */
_fformat32          /* format */
},
{
"VIEW",             /* type */
"*",                /* subtype */
1024,               /* dfltsize */
_vinit,             /* initbuf */
_vreinit,           /* reinitbuf */
NULL,               /* uninitbuf */
```

```
_vpresend,          /* presend */
NULL,               /* postsend */
NULL,               /* postrecv */
_vencdec,           /* encdec */
_vroute,            /* route */
_vfilter,           /* filter */
_vformat            /* format */
},
{
"VIEW32",           /* type */
"*"                 /* subtype */
1024,               /* dfltsize */
_vinit32,           /* initbuf */
_vreinit32,         /* reinitbuf */
NULL,               /* uninitbuf */
_vpresend32,        /* presend */
NULL,               /* postsend */
NULL,               /* postrecv */
_vencdec32,         /* encdec */
_vroute32,          /* route */
_vfilter32,         /* filter */
_vformat32          /* format */
},
{
"X_OCTET",          /* type */
"",                 /* subtype */
0,                  /* dfltsize */
},
{
"'X','_','C','_','T','Y','P','E'",     /* type */
"*",                /* subtype */
1024,               /* dfltsize */
_vinit,             /* initbuf */
_vreinit,           /* reinitbuf */
NULL,               /* uninitbuf */
_vpresend,          /* presend */
NULL,               /* postsend */
NULL,               /* postrecv */
_vencdec,           /* encdec */
_vroute,            /* route */
_vfilter,           /* filter */
_vformat            /* format */
},
{
"'X','_','C','O','M','M','O','N'",   /* type */
"*",                /* subtype */
1024,               /* dfltsize */
_vinit,             /* initbuf */
_vreinit,           /* reinitbuf */
```

```
NULL,               /* uninitbuf */
_vpresend,          /* presend */
NULL,               /* postsend */
NULL,               /* postrecv */
_vencdec,           /* encdec */
_vroute,            /* route */
_vfilter,           /* filter */
_vformat            /* format */
},
{
"XML",              /* type */
"*",                /* subtype */
0,                  /* dfltsize */
NULL,               /* _xinit - 使用不能 */
NULL,               /* _xreinit - 使用不能 */
NULL,               /* _xuninit - 使用不能 */
NULL,               /* _xpresend - 使用不能 */
NULL,               /* _xpostsend - 使用不能 */
NULL,               /* _xpostrecv - 使用不能 */
NULL,               /* _xencdec - 使用不能 */
_xroute,            /* _xroute */
NULL,               /* filter - 使用不能 */
NULL                /* format - 使用不能 */
},
{
""
}
};
```

この例をよく理解できるように、次の例に示すバッファ・タイプ構造体の宣言を参照してください。

リスト 3-17 **バッファ・タイプ構造体**

```
/* 以下の定義は、$TUXDIR/tuxedo/include/tmtypes.h にあります。*/

#define TMTYPELEN        8
#define TMSTYPELEN      16

struct tmtype_sw_t {
  char type[TMTYPELEN];      /* バッファのタイプ */
  char subtype[TMSTYPELEN];  /* バッファのサブタイプ */
  long dfltsize;   /* バッファのデフォルト・サイズ */
  /* バッファ初期化関数へのポインタ */
```

```
  int (_TMDLLENTRY *initbuf) _((char _TM_FAR *, long));
  /* バッファ再初期化関数へのポインタ */
  int (_TMDLLENTRY *reinitbuf) _((char _TM_FAR *, long));
  /* バッファ非初期化関数へのポインタ */
  int (_TMDLLENTRY *uninitbuf) _((char _TM_FAR *, long));
  /* バッファ送信前処理関数へのポインタ */
  long (_TMDLLENTRY *presend) _((char _TM_FAR *, long, long));
  /* バッファ送信後処理関数へのポインタ */
  void (_TMDLLENTRY *postsend) _((char _TM_FAR *, long, long));
  /* バッファ受信後処理関数へのポインタ */
  long (_TMDLLENTRY *postrecv) _((char _TM_FAR *, long, long));
  /* 符号化 / 復号化関数へのポインタ */
  long (_TMDLLENTRY *encdec) _((int, char _TM_FAR *, long,
    char _TM_FAR *, long));
  /* ルーティング関数へのポインタ */
  int (_TMDLLENTRY *route) _((char _TM_FAR *, char _TM_FAR *,
    char _TM_FAR *, long, char _TM_FAR *));
  /* バッファ・フィルタ処理関数へのポインタ */
  int (_TMDLLENTRY *filter) _((char _TM_FAR *, long, char _TM_FAR *,
    long));
  /* バッファ・フォーマット処理関数へのポインタ */
  int (_TMDLLENTRY *format) _((char _TM_FAR *, long, char _TM_FAR *,
    char _TM_FAR *, long));
  /* この領域は将来の拡張のために予約されています。*/
  void (_TMDLLENTRY *reserved[10]) _((void));
};
```

前述のデフォルト・バッファ・タイプ・スイッチの例は、バッファ・タイプ・スイッチの初期化を示しています。9 つのデフォルト・バッファ・タイプの後に、サブタイプの名前を指定するフィールドがあります。VIEW (X_C_TYPE と X_COMMON) 型を除き、サブタイプは NULL です。VIEW のサブタイプは "*" として指定されています。これは、デフォルトの VIEW 型のサブタイプに制約がないことを示します。つまり、VIEW 型のすべてのサブタイプは同じ方法で処理されます。

次のフィールドには、バッファのデフォルト (最小) サイズが指定されています。CARRAY (X_OCTET) 型の場合、このフィールドに 0 が指定されています。これは、CARRAY 型バッファを使用するルーチンでは、CARRAY 型に必要な領域を tpalloc() で割り当てなければならないことを示します。

それ以外のバッファ・タイプの場合、tpalloc() が呼び出されて、dfltsize フィールドに指定されている領域が BEA Tuxedo システムによって割り当てられます。ただし、tpalloc() のサイズを示す引数にこれより大きな値が設定されていない場合に限ります。

バッファ・タイプ・スイッチのエントリで、残りの 8 つのフィールドには、スイッチ・エレメント・ルーチンの名前が指定されています。これらのルーチンの詳細については、BEA Tuxedo C リファレンス の buffer(3c) を参照してください。ルーチン名からそのルーチンの処理内容がわかります。たとえば、FML 型の _fpresend は、送信前にバッファを操作するルーチンを指すポインタです。送信前処理が必要ない場合は、NULL ポインタを指定します。NULL は、特に処理が必要ないことを示し、その結果デフォルトの処理が行われます。詳細については、buffer(3c) を参照してください。

スイッチの最後に NULL エントリがあることに注目してください。変更時には、配列の終わりに必ず NULL エントリを置いてください。

スイッチ・エレメント・ルーチンのコーディング

アプリケーションで新しいバッファ・タイプを定義するのは、特別な処理を行うためです。たとえば、アプリケーションで、次のプロセスにバッファを送信する前に、データ圧縮を何度か行うとします。このようなアプリケーションでは、送信前処理ルーチンを記述します。次のコード例は、送信前処理ルーチンの宣言を示しています。

リスト 3-18 送信前処理スイッチ・エレメントのセマンティクス

```
long
presend(ptr, dlen, mdlen)
char *ptr;

long dlen, mdlen;
```

- *ptr* は、アプリケーション・データ・バッファを指すポインタです。
- *dlen* は、ルーチンに渡すデータの長さです。
- *mdlen* は、データが格納されたバッファのサイズです。

送信前処理ルーチンで行うデータ圧縮は、アプリケーションのシステム・プログラマが行います。

ルーチンの処理が正常に終了した場合、同じバッファ内にある圧縮後の送信データの長さが返されます。処理が失敗した場合は、-1 が返されます。

プログラマが記述した送信前処理ルーチンには、C コンパイラが使用できる任意の識別子を付けることができます。たとえば、_mypresend という名前を付けます。

_mypresend 圧縮ルーチンを使用する場合、受信側にはそのデータの圧縮を解除する _mypostrecv ルーチンが必要です。BEA Tuxedo C リファレンス の buffer(3c) に示すテンプレートを使用してください。

tm_typesw への新しいバッファ・タイプの追加

新しいスイッチ・エレメント・ルーチンを記述し、そのコンパイルに成功したら、そのバッファ・タイプ・スイッチに新しいバッファ・タイプを追加する必要があります。その場合、$TUXDIR/lib/tmtypesw.c をコピーします。これはデフォルトのバッファ・タイプ・スイッチのソース・コードです。コピーしたファイル名に、mytypesw.c など、拡張子として .c を付けます。コピーしたファイルに新しいタイプを追加します。タイプ名は 8 文字以内で指定します。サブタイプには、NULL ("") または 16 文字以内の文字列を指定できます。新しいスイッチ・エレメント・ルーチンの名前は、extern 宣言も含めて適切な場所に入力します。次の例は、バッファ・スイッチに新しいタイプを追加しています。

リスト 3-19 新しいタイプのバッファ・スイッチへの追加

```
#include <stdio.h>
#include <tmtypes.h>

/* カスタマイズしたバッファ・タイプ・スイッチ */

static struct tmtype_sw_t tm_typesw[] = {
{
"SOUND",          /* type */
"",               /* subtype */
50000,            /* dfltsize */
snd_init,         /* initbuf */
snd_init,         /* reinitbuf */
NULL,             /* uninitbuf */

snd_cmprs,        /* presend */
snd_uncmprs,      /* postsend */
snd_uncmprs       /* postrecv */
},
{
"FML",            /* type */
"",               /* subtype */
1024,             /* dfltsize */
_finit,           /* initbuf */
_freinit,         /* reinitbuf */
_funinit,         /* uninitbuf */
_fpresend,        /* presend */
```

```
_fpostsend,       /* postsend */
_fpostrecv,       /* postrecv */
_fencdec,         /* encdec */
_froute,          /* route */
_ffilter,         /* filter */
_fformat          /* format */
},
{
""
}
};
```

この例では、SOUND という新しいタイプを追加しています。また、VIEW、X_OCTET、X_COMMON、および X_C_TYPE のエントリを削除して、デフォルト・スイッチで不要なエントリを削除できることを示しています。配列の最後に NULL があることに注目してください。

新しいバッファ・タイプを定義する別の方法は、既存のタイプを再定義することです。バッファ・タイプ MYTYPE に定義したデータ圧縮が文字列に対して実行されたとします。その場合、STRING 型の 2 つの _dfltblen の代わりに、新しいスイッチ・エレメント・ルーチン _mypresend と _mypostrecv を使用できます。

新しい tm_typesw のコンパイルとリンク

インストールを簡単に行うには、バッファ・タイプ・スイッチを共用オブジェクトに格納します。

注記 一部のプラットフォームでは、「共用オブジェクト」ではなく「共用ライブラリ」という言葉が使用されています。Windows 2000 プラットフォームでは、「共用オブジェクト」ではなく「ダイナミック・リンク・ライブラリ」と呼ばれています。この 3 つの用語が示す機能は同じなので、ここでは「共用オブジェクト」を使用します。

ここでは、アプリケーション内のすべての BEA Tuxedo プロセスに、変更後のバッファ・タイプ・スイッチを認識させる方法について説明します。これらのプロセスには、BEA Tuxedo システムで提供されるサーバとユーティリティのほかに、アプリケーション・サーバおよびアプリケーション・クライアントも含まれています。

1. $TUXDIR/lib/tmtypesw.c をコピーして変更します。第 3 章の 36 ページ「tm_typesw への新しいバッファ・タイプの追加」を参照してください。関数を追加する場合は、それらの関数を tmtypesw.c に記述するか、別の C ソース・ファイルに記述します。
2. 共用オブジェクトに必要なフラグを設定し、tmtypesw.c をコンパイルします。

3. すべてのオブジェクト・ファイルをリンクして、共用オブジェクトを生成します。
4. libbuft.so.71 をカレント・ディレクトリから別のディレクトリにコピーします。コピー先のディレクトリとしては、アプリケーションが libbuft.so.71 を認識でき、BEA Tuxedo システムで提供されるデフォルトの共用オブジェクトより先に libbuft.so.71 が処理されるディレクトリを選択します。$APPDIR または $TUXDIR/lib ディレクトリ、または $TUXDIR/bin (Windows 2000 の場合) のいずれかを使用することをお勧めします。

オペレーティング・システムの規則に従うために、プラットフォームが異なる場合は、バッファ・タイプ・スイッチ共用オブジェクトには異なる名前が使用されます。

表 3-12 バッファ・タイプ・スイッチ共用オブジェクトの OS 別の名前

プラットフォームの種類	バッファ・タイプ・スイッチ共用オブジェクトの名前
UNIX システム (大部分は SVR4)	libbuft.so.71
HP-UX	libbuft.sl
Sun OS	libbuft.so.71
Windows (16 ビット)	wbuft.dll
Windows (32 ビット)	wbuft32.dll
OS/2 (16 ビット)	obuft.dll
OS/2 (32 ビット)	obuft.dll

共用オブジェクト・ライブラリのビルド方法については、お使いのプラットフォームのソフトウェア開発マニュアルを参照してください。

別の方法として、すべてのクライアント・プロセスとサーバ・プロセスで新しいバッファ・タイプ・スイッチを静的にリンクすることもできます。ただし、この方法ではエラーが発生しやすくなり、効率も共用オブジェクト・ライブラリをビルドするより劣ります。

16 ビット Windows プラットフォーム用の新しい tm_typesw のコンパイルとリンク

第 3 章の 37 ページ「新しい tm_typesw のコンパイルとリンク」で説明したように、Windows プラットフォーム上で tmtypesw.c を変更した場合、次のコード例に示すコマンドを使用して、変更後のバッファ・タイプ・スイッチをアプリケーションから使用できるようにします。

リスト 3-20　Microsoft Visual C++ **のコード例**

```
CL -AL -I..\e\|sysinclu -I..\e\|include -Aw -G2swx -Zp -D_TM_WIN
-D_TMDLL -Od -c TMTYPESW.C
LINK /CO /ALIGN:16 TMTYPESW.OBJ, WBUFT.DLL, NUL, WTUXWS /SE:250 /NOD
/NOE LIBW LDLLCEW, WBUFT.DEF
RC /30 /T /K WBUFT.DLL
```

データ変換

コンフィグレーション・ファイルの MACHINES セクションにある TYPE パラメータは、データ表現と使用するコンパイラが同じであるマシンをグループ化して、TYPE が異なるマシン間でやり取りされるメッセージのデータが変換されるようにします。デフォルトのバッファ・タイプの場合、異なるマシン間でのデータ変換はユーザ、管理者、プログラマに対して透過的です。

アプリケーションで新しいバッファ・タイプを定義して、データ表現スキーマが異なるマシン間でメッセージを交換する場合、新しい符号化 / 復号化ルーチンを記述して、バッファ・タイプ・スイッチに組み込む必要があります。独自のデータ変換ルーチンを記述する場合は、次のガイドラインに従ってください。

- 『BEA Tuxedo C リファレンス』の buffer(3c) に示してある _tmencdec ルーチンのセマンティクスを使用します。つまり、同じ引数を使用し、処理が成功した場合や失敗した場合に _tmencdec と同じ値が返されるようにルーチンをコーディングします。新しいバッファ・タイプを定義する場合は、第 3 章の 30 ページ「独自のバッファ・タイプの定義」の手順に従って、新しいバッファ・タイプを使用するサービスを提供するサーバをビルドします。

符号化 / 復号化ルーチンが呼び出されるのは、データをやり取りする 2 つのマシンの TYPE が異なることが BEA Tuxedo システムによって検出された場合だけです。

4　クライアントのコーディング

ここでは、次の内容について説明します。

- アプリケーションへの参加
- TPINIT 型バッファの機能
- アプリケーションからの分離
- クライアントのビルド
- クライアント・プロセスの例

アプリケーションへの参加

クライアントがサービスを要求する場合、BEA Tuxedo アプリケーションに明示的または暗黙的に参加している必要があります。アプリケーションに参加すると、クライアントは要求を送り、その応答を受け取ることができるようになります。

クライアントが明示的にアプリケーションに参加するには、次の文法を指定して `tpinit(3c)` 関数を呼び出します。

```
int
tpinit (TPINIT *tpinfo)
```

クライアントが `tpinit()` 関数を呼び出す前にサービス要求 (または ATMI 関数) を呼び出すと、暗黙的にアプリケーションに参加したことになります。その場合、`tpinfo` 引数が NULL に設定されて、`tpinit()` 関数がクライアントではなく BEA Tuxedo システムによって呼び出されます。`tpinfo` 引数は、タイプが `TPINIT`、サブタイプが NULL の型付きバッファを指します。`TPINIT` 型バッファは `atmi.h` ヘッダ・ファイルに定義されており、次の情報が格納されています。

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

次の表は、`TPINIT` データ構造体のフィールドを示しています。

表 4-1 TPINIT データ構造体のフィールド

フィールド	説明
Usrname	クライアントを表す名前。ブロードキャスト通知と管理統計情報の検索に使用されます。クライアントは、tpinit() 関数を呼び出すときに *usrname* に値を設定します。この値には、最大文字数が MAXTIDENT で、最後が NULL で終わる文字列を指定します。最大文字数のデフォルト値は 30 ですが、管理者が変更できます。
cltname	アプリケーション定義のセマンティクスに従ったクライアント名。30 文字以内で最後が NULL で終わる文字列を指定します。ブロードキャスト通知と管理統計情報の検索に使用されます。クライアントは、tpinit() 関数を呼び出すときに *cltname* に値を設定します。この値には、最大文字数が MAXTIDENT で、最後が NULL で終わる文字列を指定します。最大文字数のデフォルト値は 30 ですが、管理者が変更できます。 **注記** sysclient の値は、*cltname* フィールド用に予約されています。
passwd	非暗号化形式のアプリケーション・パスワード。ユーザ認証に使用されます。30 文字以内の文字列を指定します。
grpname	クライアントをリソース・マネージャ・グループに対応付ける値。grpname が長さ 0 の文字列として設定されている場合、クライアントはリソース・マネージャに対応付けられず、デフォルトのクライアント・グループに属します。ワークステーション・クライアントの場合、grpname の値には NULL 文字列 (長さが 0 の文字列) を指定します。ワークステーション・クライアントの詳細については、BEA Tuxedo Workstation コンポーネントを参照してください。
flags	クライアント固有の通知メカニズム、およびシステム・アクセスのモード。この値により、マルチコンテキスト・モードとシングルコンテキスト・モードが制御されます。フラグの詳細については、第 4 章の 5 ページ「任意通知型通知の処理」、または『BEA Tuxedo C リファレンス』の tpinit() を参照してください。

フィールド	説明
datalen	アプリケーション固有データの長さ。TPINIT 型バッファのバッファ・タイプ・スイッチ・エントリは、そのバッファに渡される合計サイズに基づいてこのフィールドを設定します。アプリケーション・データのサイズは、合計サイズから TPINIT 構造体自体のサイズを差し引き、その値に構造体に定義されているデータ・プレースホルダのサイズを加算したものです。
data	アプリケーション定義の認証サービスに転送される可変長データ用のプレース・ホルダ。

クライアント・プログラムは、アプリケーションに参加する前に tpalloc() を呼び出して TPINIT バッファを割り当てる必要があります。次のコード例は、TPINIT バッファを割り当て、そのバッファを使用してアプリケーション固有の 8 バイトのデータを tpinit() 関数に渡す方法を示しています。

リスト 4-1 TPINIT 型バッファの割り当て

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

TPINIT 型バッファの詳細については、『BEA Tuxedo C リファレンス』の tpinit() を参照してください。

関連項目

- 『BEA Tuxedo C リファレンス』の tpinit(3c)

TPINIT 型バッファの機能

次の TPINIT 型バッファの機能をクライアントで利用するには、tpinit() 関数を明示的に呼び出す必要があります。

- クライアント命名
- 任意通知型通知の処理
- システム・アクセス・モード
- リソース・マネージャとの対応付け
- クライアント認証

クライアント命名

クライアントがアプリケーションに参加すると、BEA Tuxedo システムによって一意なクライアント識別子が割り当てられます。識別子は、クライアントによって呼び出される各サービスに渡されます。識別子は、任意通知型通知に使用することもできます。

一意なクライアント名とユーザ名をそれぞれ 30 文字以内で割り当てることもできます。その場合、tpinfo バッファ引数を使用して名前を tpinit() 関数に渡します。BEA Tuxedo システムでは、各プロセスに対応付けられているクライアント名とユーザ名、およびプロセスが実行されているマシンの論理マシン ID (LMID) を組み合わせることにより、そのプロセスに対して一意な識別子が使用されます。これらのフィールド値を取得する方法は選択することができます。

注記 プロセスがアプリケーションの管理ドメイン以外で実行されている場合 (つまり、管理ドメインに接続されたワークステーション上で実行されている場合)、アプリケーションにアクセスするためにワークステーション・クライアントで使用されているマシンの LMID が割り当てられます。

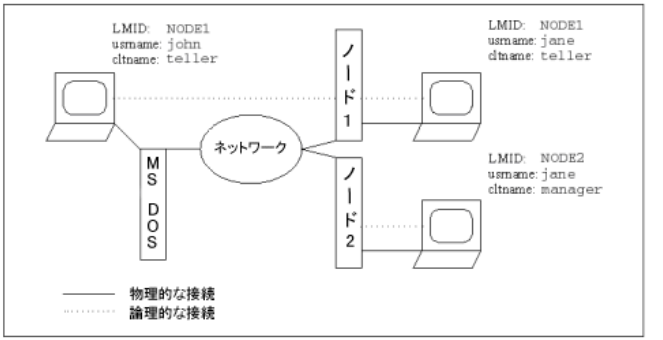
クライアント・プロセスに対して一意な識別子が作成されると、次の操作を行うことができます。

- クライアント認証をインプリメントできます。
- tpnotify() と tpbroadcast() を使用して、特定のクライアントまたはクライアントのグループに任意通知型メッセージを送信できます。
- tmadmin(1) を使用して、詳細な統計データを収集できます。

任意通知型メッセージの送受信の詳細については、第 8 章の 1 ページ「イベント・ベースのクライアントおよびサーバのコーディング」を参照してください。`tmadmin(1)` の詳細については、『BEA Tuxedo C リファレンス』を参照してください。

次の図は、アプリケーションにアクセスするクライアントに名前を割り当てる方法を示しています。この例では、ジョブ関数を示す `cltname` フィールドがアプリケーションで使用されています。

図 4-1 クライアントの名前付け



任意通知型通知の処理

任意通知型通知とは、クライアントが予期していないサービス要求に対する応答 (またはエラー・コード) を受け取る通信です。たとえば、管理者が 5 分後にシステムをシャットダウンすることを通知するメッセージをブロードキャストした場合などです。

クライアントに任意通知型メッセージを通知する方法は数多くあります。たとえば、オペレーティング・システムがクライアントにシグナルを送って、クライアントの現在の処理を中断させる方法があります。BEA Tuxedo システムでは、ATMI 関数が呼び出されるたびに任意通知型メッセージが到着していないかどうかがデフォルトで確認されます。これはディップ・インと呼ばれる方法で、次の利点があります。

- すべてのプラットフォームでサポートされています。
- 現在の処理が中断されません。

ディップ・インでは、メッセージの到着を確認するまでの間隔が長い場合があります。そのため、アプリケーションで tpchkunsol() 関数を呼び出して、既に到着している任意通知型メッセージがないかどうかを確認できます。tpchkunsol() 関数の詳細については、第 8 章の 1 ページ「イベント・ベースのクライアントおよびサーバのコーディング」を参照してください。

クライアントが tpinit() 関数を使用してアプリケーションに参加する場合、フラグを定義して任意通知型メッセージの処理方法を指定できます。クライアントへの通知では、次の表に示す値を flags に指定できます。

表 4-2 TPINIT 型バッファのクライアント通知でのフラグ

フラグ	説明
TPU_SIG	シグナルによる任意通知を選択します。このフラグは、シングル・スレッドでシングル・コンテキストのアプリケーションのみで使用します。このモードの利点は、直ちに通知できることです。このモードには、次のような不都合があります。 ■ ネイティブ・クライアントを実行している場合、呼び出し元プロセスで送信元プロセスと同じ UID を使用する必要があります。ワークステーション・クライアントには、この制約はありません。 ■ すべてのプラットフォーム上で TPU_SIG が使用できるわけではありません。特に、MS-DOS ワークステーションでは使用できません。 システムや環境の要件を満たしていない場合にこのフラグを指定すると、フラグに TPU_DIP が設定され、ログにイベントが記録されます。
TPU_DIP (デフォルト)	ディップ・インを使用した任意通知型メッセージを指定します。クライアントは tpchkunsol() 関数を使用してメッセージ処理関数の名前を指定し、tpsetunsol() 関数を使用して待機中の任意通知型メッセージを確認できます。
TPU_THREAD	別のスレッド内の THREAD 通知を指定します。このフラグは、マルチスレッドをサポートするプラットフォーム専用です。マルチスレッドがサポートされていないプラットフォームで TPU_THREAD を指定すると、無効な引数として処理されます。その結果、tperrno(5) エラーが返されて TPEINVAL が設定されます。
TPU_IGN	任意通知を無視します。

TPINIT 型バッファ・フラグの詳細については、『BEA Tuxedo C リファレンス』の tpinit(3c) を参照してください。

システム・アクセス・モード

アプリケーションは、protected または fastpath のいずれかのモードで BEA Tuxedo システムにアクセスできます。クライアントは、tpinit() 関数を使用してアプリケーションに参加するときに、モードを要求できます。モードを指定するには、TPINIT バッファの flags フィールドにいずれかのモードを指定して、その値を tpinit() 関数に渡します。

表 4-3 TPINIT 型バッファのシステム・アクセス・フラグ

モード	説明
PROTECTED	アプリケーション内で ATMI 関数を呼び出し、共用メモリを使用して BEA Tuxedo システムの内部テーブルにアクセスします。BEA Tuxedo システム・ライブラリ外のアプリケーション・コードからのアクセスに対して、共用メモリを保護します。この値は、NO_OVERRIDE が指定されている場合を除き、UBBCONFIG の値に優先します。UBBCONFIG の詳細については、『BEA Tuxedo アプリケーションの設定』を参照してください。
FASTPATH (デフォルト)	アプリケーション・コード内で ATMI 関数を呼び出し、共用メモリを使用して BEA Tuxedo システム内部にアクセスします。BEA Tuxedo システム・ライブラリ外のアプリケーション・コードからのアクセスに対して、共用メモリを保護しません。この値は、NO_OVERRIDE が指定されている場合を除き、UBBCONFIG の値に優先します。UBBCONFIG の詳細については、『BEA Tuxedo アプリケーションの設定』を参照してください。

リソース・マネージャとの対応付け

アプリケーション管理者は、リソース・マネージャに対応付けられたサーバ（トランザクションを調整するための管理プロセスを提供するサーバを含む）をグループ化できます。グループの定義については、『BEA Tuxedo アプリケーションの設定』を参照してください。

アプリケーションに参加している場合、クライアントは TPINIT 型バッファの *grpname* フィールドにグループ名を指定して、特定のグループに参加できます。

クライアント認証

BEA Tuxedo システムでは、オペレーティング・システムのセキュリティ、アプリケーション・パスワード、ユーザ認証、オプションのアクセス制御リスト、必須のアクセス制御リスト、リンク・レベルの暗号化などのセキュリティ・レベルを設定できます。セキュリティ・レベル設定の詳細については、『BEA Tuxedo アプリケーションの設定』を参照してください。

セキュリティ・レベルとしてアプリケーション・パスワードが設定されている場合、アプリケーションに参加するときに、すべてのクライアントがアプリケーション・パスワードを入力する必要があります。管理者はアプリケーション・パスワードを設定したり変更できます。また、そのパスワードを有効なユーザに提供する必要があります。

このレベルのセキュリティが設定されていると、BEA Tuxedo システムで提供される ud(1) などのクライアント・プログラムでは、アプリケーション・パスワードの入力が求められます。ud(1)、wud(1) 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。アプリケーション固有のクライアント・プログラムには、ユーザからパスワードを取得するコードが記述されていることが必要です。クライアントがアプリケーションに参加するために tpinit() を呼び出すと、非暗号化パスワードが TPINIT バッファに格納されて評価されます。

注記 パスワードは画面には表示されません。

tpchkauth(3c) 関数を使用すると、次の内容を確認できます。

- アプリケーションで認証が必要かどうか。
- アプリケーションで認証が必要な場合、次のどちらの認証が行われるか。
  - アプリケーション・パスワードに基づくシステム認証
  - アプリケーション・パスワードとユーザ固有の情報に基づくアプリケーション認証

通常、クライアントは tpinit() より先に tpchkauth() 関数を呼び出して、初期化時に指定する必要のあるセキュリティ情報を確認します。

セキュリティのプログラミング手法の詳細については、『BEA Tuxedo CORBA アプリケーションのセキュリティ機能』を参照してください。

アプリケーションからの分離

クライアントがすべてのサービスを要求して、それに対する応答を受信したら、tpterm(3c) 関数を使用してアプリケーションから分離できます。tpterm(3c) 関数には引数がなく、エラー時には -1 を返します。

クライアントのビルド

実行可能クラアイントをビルドするには、buildclient(1) コマンドを実行して、BEA Tuxedo システム・ライブラリとそのほかのすべての参照ファイルを使用してアプリケーションをコンパイルします。次は、buildclient コマンドの構文です。

```
buildclient filename.c -o filename -f filenames -l filenames
```

次の表は、buildclient コマンドのオプションを示しています。

表 4-4 buildclient のオプション

オプションまたは引数	説明
filename.c	コンパイルする C 言語のアプリケーション。
-o *filename*	実行可能な出力ファイル。出力ファイルのデフォルト名は a.out です。
-f *filenames*	BEA Tuxedo システムのライブラリより先にリンクされるファイルのリスト。-f オプションは、コマンド行で複数回指定できます。また、各 -f に複数のファイル名を指定できます。C プログラム・ファイル (*file*.c) を指定すると、リンクされる前にコンパイルが行われます。ほかのオブジェクト・ファイル (*file*.o) を個別に、またはアーカイブ・ファイル (file.a) にまとめて指定することもできます。

オプションまたは引数	説明
-l *filenames*	BEA Tuxedo システム・ライブラリの後にリンクされるファイルのリスト。-l オプションは、コマンド行で複数回指定できます。また、各 -l に複数のファイル名を指定できます。C プログラム・ファイル (*file*.c) を指定すると、リンクされる前にコンパイルが行われます。ほかのオブジェクト・ファイル (*file*.o) を個別に、またはアーカイブ・ファイル (file.a) にまとめて指定することもできます。
-r	実行可能サーバにリンクされるリソース・マネージャのアクセス・ライブラリ。アプリケーション管理者は、buildtms(1) コマンドを使用して、すべての有効なリソース・マネージャ情報を $TUXDIR/updataobj/RM ファイルに事前に定義しておく必要があります。指定できるリソース・マネージャは 1 つだけです。詳細については、『BEA Tuxedo アプリケーションの設定』を参照してください。

注記 BEA Tuxedo ライブラリは自動的にリンクされます。コマンド行に BEA Tuxedo ライブラリを指定する必要はありません。

リンクするライブラリ・ファイルの指定順序は重要です。関数を呼び出す順序と、それらの関数への参照を含むライブラリによって、この順序が決定されます。

デフォルトでは、buildclient コマンドは UNIX の cc コマンドを呼び出します。環境変数 CC を指定して別のコンパイル・コマンドを指定したり、CFLAGS を指定してコンパイル・フェーズやリンク・フェーズにフラグを設定することができます。詳細については、第 2 章の 4 ページ「環境変数の設定」を参照してください。

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

次のコマンド行の例では、C プログラム audit.c をコンパイルして、実行可能ファイル audit を生成しています。

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

関連項目

- 第 5 章の 31 ページ「サーバのビルド」
- 『BEA Tuxedo コマンド・リファレンス』の buildclient(1)

クライアント・プロセスの例

次の疑似コードは、通常のクライアント・プロセスがアプリケーションに参加してから分離するまでの処理を示しています。

リスト 4-2 クライアント・プロセスのパラダイム

```
main()
{
     セキュリティ・レベルをチェック
     tpsetunsol() を呼び出して TPU_DIP 用にハンドラを命名
     usrname, cltname を取得
     アプリケーション・パスワードの入力を要求
     TPINIT バッファを割り当て
     TPINIT バッファ構造体メンバに値を格納

     if (tpinit((TPINIT *) tpinfo) == -1){
          エラー・ルーチン ;
     }

     ユーザ入力が存在する間に
     メッセージ・バッファを割り当て {
          ユーザ入力をバッファに格納
          サービス呼び出しを行う
          応答を受信
          任意通知型メッセージを確認
     }
     バッファを開放
     . . .
     if (tpterm() == -1) {
          エラー・ルーチン ;
     }
}
```

エラーが発生すると -1 が返され、アプリケーションが外部グローバル変数 tperrno にエラーの原因を示す値を設定します。tperrno は、atmi.h ヘッダ・ファイルに定義されています。詳細については、『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 tperrno(5) を参照してください。通常、プログラマが発生したエラーの種類を示すエラー コードをこのグローバル変数に設定します。tperrno の値については 第 11 章の 1 ページ「システム・エラー」を参照してください。各 ATMI 呼び出しで返される全エラー・コードのリストについては、『BEA Tuxedo C リファレンス』の 「C 言語アプリケーション・トランザクション・モニタ・インターフェイスについて」を参照してください。

次の例は、tpinit() と tpterm() 関数の使用方法を示しています。このコード例は、BEA Tuxedo ソフトウェアに提供されている銀行業務のサンプル・アプリケーション bankapp から引用したものです。

リスト 4-3 アプリケーションへの参加と分離

```
#include <stdio.h>          /* UNIX */
#include <string.h>         /* UNIX */
#include <fml.h>            /* BEA Tuxedo システム */
#include <atmi.h>           /* BEA Tuxedo システム */
#include <Uunix.h>          /* BEA Tuxedo システム */
#include <userlog.h>        /* BEA Tuxedo システム */
#include "bank.h"           /* 銀行業務アプリケーションのマクロ定義 */
#include "aud.h"            /* 銀行業務アプリケーションの VIEW 定義 */

...

main(argc, argv)
int argc;
char *argv[];

{
     ...
     if (strrchr(argv[0],'/') != NULL)
      proc_name = strrchr(argv[0],'/')+1;
     else
      proc_name = argv[0];
     ...
     /* アプリケーションへの参加 */
     if (tpinit((TPINIT *) NULL) == -1) {
     (void)userlog("%s:failed to join application\n", proc_name);
       exit(1);
     }
     ...
```

```
    /* アプリケーションからの分離 */
    if (tpterm() == -1) {
    (void)userlog("%s:failed to leave application\n", proc_name);
     exit(1);
    }
}
```

この例は、tpinit() を呼び出すことによって、アプリケーションに参加しているクライアント・プロセスを示しています。エラー (戻り値が -1) が発生すると、このプロセスは C プログラム文の printf() と似た引数を取る userlog() を呼び出して、中央イベント・ログにメッセージを書き込みます。詳細については、『BEA Tuxedo C リファレンス』の userlog(3c) を参照してください。

同様に、tpterm() を呼び出し、エラーが発生すると、このプロセスは中央イベント・ログにメッセージを書き込みます。

5　サーバのコーディング

ここでは、次の内容について説明します。

- BEA Tuxedo システムの main( )
- システムで提供されるサーバおよびサービス
- サーバのコーディングのためのガイドライン
- サービスの定義
- 例 : バッファ・タイプの確認
- 例 : サービス要求の優先順位の確認
- サービス・ルーチンの終了
- サービスの宣言と宣言の取り消し
- サーバのビルド C++ コンパイラ

BEA Tuxedo システムの main( )

BEA Tuxedo システムには、サーバを簡単に開発できるように、サーバのロード・モジュール用に定義済み `main()` ルーチンが提供されています。`buildserver` コマンドを実行すると、`main()` ルーチンが自動的にサーバの一部として組み込まれます。

注記 `main()` はシステムで提供されたルーチンなので、変更することはできません。

定義済み `main()` ルーチンは、アプリケーションへの参加と終了のほかに、サーバに代わって次の操作を行います。

- ハングアップを無視してプロセスを実行します。つまり、`SIGHUP` シグナルを無視します。
- 標準オペレーティング・システム・ソフトウェアの終了シグナル (`SIGTERM`) を受信すると、終了処理を開始します。サーバはシャットダウンされ、必要な場合は再起動します。
- 掲示板サービスが参照できるように共用メモリを割り当てます。
- プロセスに対してメッセージ・キューを作成します。

- サーバによって提供される初期サービスを宣言します。初期サービスは、定義済み main() 関数とリンクされたすべてのサービス、または BEA Tuxedo システムの管理者がコンフィギュレーション・ファイルに指定したサブセットです。
- コマンド行に入力された 2 つのダッシュ (--) までの引数を処理します。2 つのダッシュは、システムで認識される引数の終わりを示します。
- tpsvrinit() 関数を呼び出して、コマンド行で 2 つのダッシュ (--) の後に入力された引数を処理したり、リソース・マネージャをオープンします (省略可能)。このようなコマンド行の引数は、アプリケーション固有の初期化に使用されます。
- 中止が要求されるまで、要求キューにサービス要求メッセージがあるかどうかを確認します。
- サービス要求メッセージが要求キューに到着すると、中止が要求されるまで、main() は次の処理を行います。
  - -r オプションが指定されている場合、サービス要求の開始時間を記録します。
  - 掲示板を更新して、サーバが BUSY であることを示します。
  - 要求メッセージ用にバッファを割り当て、サービスにディスパッチします。つまり、サービス・サブルーチンを呼び出します。
- サービスが入力に対する処理を終了して制御が戻ると、中止が要求されるまで、main() は次の処理を行います。
  - -r オプションが指定されている場合、サービス要求の終了時間を記録します。
  - 統計を更新します。
  - 掲示板を更新して、サーバが IDLE 状態であること、つまりサーバの準備ができたことを示します。
  - キューに次のサービス要求があるかどうかを確認します。
- サーバの中止が要求されると、tpsvrdone() を呼び出して必要なシャットダウン操作を行います。

以上からわかるように、main() ルーチンは、アプリケーションへの参加と終了、バッファやトランザクションの管理、および通信に関する詳細を扱っています。

注記 システムで提供される main() は、アプリケーションへの参加と終了を行います。そのため、tpinit() または tpterm() 関数への呼び出しをコードに記述しないでください。これらの関数を呼び出すとエラーが発生して、tperrno に TPEPROTO が返されます。tpinit() または tpterm() 関数の詳細については、第 4 章の 1 ページ「クライアントのコーディング」を参照してください。

システムで提供されるサーバおよびサービス

main() ルーチンは、システムで提供される 1 つのサーバ AUTHSVR、および 2 つのサブルーチン tpsvrinit() と tpsvrdone() を提供します。これらの 3 つのデフォルト・バージョンについては、以降の節で説明します。これらのサーバとサブルーチンは、ご使用のアプリケーションに合わせて変更することができます。

注記 tpsvrinit() と tpsvrdone() を独自にコーディングする場合、この 2 つのルーチンのデフォルト・バージョンが、それぞれ tx_open() と tx_close() を呼び出すことに注意してください。tx_open() ではなく tpopen() を呼び出す tpsvrinit() の新しいバージョンをコーディングする場合は、tpclose() を呼び出す tpsvrdone() もコーディングする必要があります。つまり、この 2 つの関数が呼び出すオープンとクローズの関数は対になっていなければなりません。

ここで説明するサブルーチンのほかに、tpsvrthrinit(3c) と tpsvrthrdone(3c) の 2 つのサブルーチンがシステムで提供されています。詳細については、第 10 章の 1 ページ「マルチスレッドおよびマルチコンテキスト・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。

システムで提供されるサーバ : AUTHSVR( )

AUTHSVR(5) を使用すると、アプリケーションで各クライアントの認証を行うことができます。このサーバは、アプリケーションのセキュリティ・レベルが TPAPPAUTH に設定されている場合に、tpinit() によって呼び出されます。

AUTHSVR のサービスは、TPINIT 型バッファの data フィールドでユーザ・パスワードを確認します。このユーザ・パスワードは、TPINIT 型バッファの passwd フィールドにあるアプリケーション・パスワードとは異なります。デフォルトでは、システムによって data から文字列が取得され、それと合致する文字列が /etc/passwd ファイルで検索されます。

tpinit() がネイティブ・サイトのクライアントから呼び出された場合、受信した data フィールドはそのまま転送されます。そのため、アプリケーションでパスワードの暗号化が必要な場合、それに応じてクライアント・プログラムをコーディングする必要があります。

tpinit() がワークステーション・クライアントから呼び出された場合、データは暗号化されてからネットワークに送信されます。

システムで提供されるサービス :tpsvrinit( ) 関数

サーバの起動時に、BEA Tuxedo システムの main( ) はその初期化時、つまりサービス要求の処理を開始する前に、tpsvrinit(3c) を呼び出します。

アプリケーションがこの関数のカスタム・バージョンをサーバに提供していない場合、main( ) で提供されるデフォルトの関数が呼び出されます。この関数は、リソース・マネージャをオープンし、エントリを中央イベント・ログに記録してサーバの起動が成功したことを示します。中央ユーザ・ログは自動的に生成されるファイルで、userlog(3c) を呼び出して、プロセスがこのファイルにメッセージを書き込みます。中央イベント・ログの詳細については、第 11 章の 1 ページ「エラーの管理」を参照してください。

tpsvrinit( ) 関数を使用すると、アプリケーションで要求される次のような初期化プロセスを行うことができます。

- コマンド行オプションの取得
- データベースのオープン

以降の節では、tpsvrinit( ) を呼び出すことによって、これらの初期化プロセスがどのように行われるのかをコード例で示します。以下のコード例では示していませんが、このルーチン内ではメッセージ交換を行うこともできます。ただし、非同期応答が未処理のままで tpsvrinit( ) 関数が制御を戻すと、この関数は失敗します。その場合、BEA Tuxedo システムでは応答は無視されて、サーバが正常に終了します。

また、tpsvrinit( ) 関数で、トランザクションを開始したり終了することができます。第 11 章の 1 ページ「エラーの管理」を参照してください。

tpsvrinit( ) 関数の呼び出しには、次の文法を使用します。

```
int
tpsvrinit(int argc, char **argv)
```

コマンド行オプションの取得

サーバは、起動時に最初の処理として、コンフィグレーション・ファイルに指定されているサーバ・オプションを EOF まで読み取ります。その場合、サーバは UNIX 関数 getopt(3) を呼び出します。コマンド行に 2 つのダッシュ (--) が出現した時点で、getopt( ) 関数は EOF を返します。getopt 関数は、次に処理する引数の argv インデックスを外部変数 optind に設定します。その後、定義済み main( ) 関数が tpsvrinit( ) を呼び出します。

次のコード例は、tpsvrinit( ) 関数でコマンド行オプションを取得する方法を示しています。

リスト 5-1 tpsvrinit() **を使用したコマンド行オプションの取得**

```
tpsvrinit(argc, argv)
int argc;
char **argv;
{
       int c;
       extern char *optarg;
       extern int optind;
       .
       .
       .
       while((c = getopt(argc, argv, "f:x:"))!= EOF)
          switch(c){
          .
          .
          .
          }
       .
       .
       .
}
```

main() は tpsvrinit() を呼び出すときに、コマンド行の 2 つのダッシュ (--) の後にある引数を使用します。前述の例では、f と x オプションは、コロンで区切られていることからわかるようにそれぞれ引数を取ります。optarg は、オプション引数の先頭を指しています。switch 文のロジックは省略してあります。

リソース・マネージャのオープン

tpsvrinit() 関数のもう 1 つの使用法として、リソース・マネージャをオープンすることができます。BEA Tuxedo システムには、リソース・マネージャをオープンする関数として、tpopen(3c) と tx_open(3c) があります。また、これらと対の関数として、tpclose(3c) と tx_close(3c) があります。これらの関数を呼び出してリソース・マネージャをオープンしたりクローズするアプリケーションは、その意味では移植性があります。これらのアプリケーションは、コンフィギュレーション・ファイルに設定されたリソース・マネージャのインスタンス固有の情報にアクセスすることによって動作します。

注記 マルチスレッド・サーバのコーディングでは、tpsvrthrinit() 関数を使用して、リソース・マネージャをオープンする必要があります。第 10 章の 1 ページ「マルチスレッドおよびマルチコンテキスト・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。

これらの関数呼び出しは省略可能です。また、リソース・マネージャがデータベースの場合、リソース・マネージャ固有の呼び出しがデータ操作言語 (DML) の一部であるときは、その呼び出しの代わりに使用できます。userlog(3c) 関数を使用して、中央イベント・ログに書き込んでいることに注目してください。

注記 コマンド行オプションを受け取り、データベースをオープンする初期化関数を作成するには、次のコード例と前述のコード例を組み合わせます。

リスト 5-2 tpsvrinit() を使用したリソース・マネージャのオープン

```
tpsvrinit()
{

    /* データベースのオープン */

    if (tpopen() == -1) {
         (void)userlog("tpsvrinit:failed to open database: ");
         switch (tperrno) {
           case TPESYSTEM:
              (void)userlog("System error\n");
               break;
           case TPEOS:
              (void)userlog("Unix error %d\n",Uunixerr);
               break;
           case TPEPROTO:
              (void)userlog("Called in improper context\n");
               break;
           case TPERMERR:
              (void)userlog("RM failure\n");
               break;
      }
      return(-1);     /* サーバの終了 */
  }
  return(0);
}
```

初期化時にエラーが発生しないように、サーバを終了してからサービス要求の処理を開始するように tpsvrinit() 関数をコーディングできます。

システムで提供されるサービス :tpsvrdone( ) 関数

tpsvrinit( ) が tpopen( ) を呼び出してリソース・マネージャをオープンするのと同じように、tpsvrdone( ) 関数は tpclose( ) を呼び出してリソース・マネージャをクローズします。

注記 マルチスレッド・サーバのコーディングでは、tpsvrthrdone( ) コマンドを使用して、リソース・マネージャをクローズする必要があります。第 10 章の 1 ページ「マルチスレッドおよびマルチコンテキスト・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。

tpsvrdone( ) 関数の呼び出しには、次の文法を使用します。

```
void
tpsvrdone()  /* サーバ終了ルーチン */
```

tpsvrdone( ) 関数に引数は必要ありません。

アプリケーションに tpsvrdone( ) 関数のクローズ用ルーチンが定義されていない場合、main( ) で提供されるデフォルトのルーチンが BEA Tuxedo システムによって呼び出されます。このルーチンは、tx_close( ) と userlog( ) を呼び出してリソース・マネージャをクローズし、中央イベント・ログへに書き込みます。ログには、サーバが間もなく終了することを伝えるメッセージが書き込まれます。

tpsvrdone( ) 関数は、サーバがサービス要求の処理を終了した後、サーバが終了する前に呼び出されます。その場合、サーバはまだシステムの一部なので、特定の規則に従っている限り、ルーチン内でさらに通信およびトランザクションが行われる場合があります。これらの規則については、第 11 章の 1 ページ「エラーの管理」を参照してください。

次のコード例は、tpsvrdone( ) 関数を使用して、リソース・マネージャをクローズして終了する方法を示しています。

リスト 5-3 tpsvrdone() **を使用したリソース・マネージャのクローズ**

```
void
tpsvrdone()
{

     /* データベースのクローズ */
     if(tpclose() == -1)
           (void)userlog("tpsvrdone:failed to close database: ");
           switch (tperrno) {
                  case TPESYSTEM:
                         (void)userlog("BEA TUXEDO error\n");
```

```
                        break;
                case TPEOS:
                        (void)userlog("Unix error %d\n",Uunixerr);
                        break;
                case TPEPROTO:
                       (void)userlog("Called in improper context\n");
                        break;
                case TPERMERR:
                        (void)userlog("RM failure\n");
                        break;
        }
        return;
    }
    return;
}
```

サーバのコーディングのためのガイドライン

通信の詳細は BEA Tuxedo システムの `main()` ルーチンによって処理されるので、プログラマは通信のインプリメントよりもアプリケーション・サービスのロジックに集中できます。ただし、システムで提供される `main()` と互換性を保つために、アプリケーション・サービスが特定の規約に従っている必要があります。これらの規約は、サービス・ルーチンをコーディングするためのサービス・テンプレートと言えます。以下に、これらの規約についてまとめます。これらの規約の詳細については、『BEA Tuxedo C リファレンス』の `tpservice(3c)` リファレンス・ページを参照してください。

- 要求 / 応答サービスが一度に受信できる要求の数は 1 つだけであり、送信できる応答も 1 つだけです。
- 要求 / 応答サービスが一度に処理する要求は 1 つだけです。別の要求を受け取ることができるのは、要求元に応答を送信した後、または別の処理を行うために別のサービスに要求を転送した後だけです。
- サービス・ルーチンを終了するには、`tpreturn()` または `tpforward()` 関数のいずれかを呼び出す必要があります。これらの関数は、C 言語の `return` 文と同じように機能します。ただし、処理が終了すると、呼び出し元の関数ではなく BEA Tuxedo システムの `main()` 関数に制御が戻ります。
- `tpacall()` を使用して別のサーバと通信する場合は、サービスの開始元は未処理の応答がすべて処理されるまで待機するか、または `tpcancel()` を使用して未処理の応答をすべて無効にしてから、`tpreturn()` または `tpforward()` を呼び出す必要があります。

- サービス・ルーチンは、1 つの引数、つまり *svcinfo* を使用して呼び出します。この引数は、サービス情報が定義された構造体 (TPSVCINFO) を指すポインタです。

サービスの定義

すべてのサービス・ルーチンは、1 つの引数、つまり TPSVCINFO 構造体を指すポインタを受け取る関数として定義する必要があります。TPSVCINFO 構造体は、atmi.h ヘッダ・ファイルに定義され、次の情報が含まれています。

```
char      name[32];
long      flags;
char      *data;
long      len;
int       cd;
int       appkey;
CLIENTID  cltid;
```

次の表は、TPSVCINFO データ構造体を示しています。

表 5-1 TPSVCINFO データ構造体

フィールド	説明
name	要求元プロセスがサービスの呼び出しに使用した名前。この名前は、サービス・ルーチンに対して指定されます。

フィールド	説明
flags	サービスがトランザクション・モードであるかどうか、または呼び出し元が応答を要求しているかどうかをサービスに通知するための値。サービスをトランザクション・モードにする方法については、第 9 章の 1 ページ「グローバル・トランザクションのコーディング」を参照してください。 TPTRAN フラグは、サービスがトランザクション・モードであることを示します。tpcall() または tpacall() の flags パラメータに TPNOTRAN を設定してサービスを呼び出した場合、サービスは現在のトランザクションに参加できません。ただし、トランザクション・モードでサービスを実行することはできます。つまり、呼び出し元によって TPNOTRAN 通信フラグが設定されていても、TPTRAN を svcinfo->flags に設定できます。このような状況の例については、第 9 章の 1 ページ「グローバル・トランザクションのコーディング」を参照してください。 tpacall() で TPNOREPLY 通信フラグが設定されてサービスが呼び出された場合、*flags* メンバは TPNOREPLY に設定されます。ただし、呼び出されたサービスが、呼び出し元プロセスと同じトランザクションに含まれる場合、呼び出し元に応答を返す必要があります。
data	main() 内で tpalloc() を使用して既に割り当てられているバッファを指すポインタ。このバッファは、要求メッセージの受信に使用されます。応答メッセージの返信や要求メッセージの転送にも、このバッファを使用することをお勧めします。
len	*data* フィールドによって参照されるバッファ内の要求データの長さ。
cd	会話型通信での接続記述子。
appkey	アプリケーションで使用するために予約された値。アプリケーション固有の認証が設計に含まれている場合、認証サーバによって認証の成功または失敗を示す値、およびクライアント認証キーが返される必要があります。認証サーバは、クライアントがアプリケーションに参加するときに呼び出されます。BEA Tuxedo システムは、*appkey* をクライアントのために保持し、このフィールドに格納して以降のサービス要求に情報を渡します。*appkey* がサービスに渡されたときは、クライアントの認証は終了しています。ただし、サービス内で *appkey* フィールドを使用して、サービスを呼び出したユーザ、またはそのユーザに関するそのほかのパラメータを識別できます。 このフィールドが使用されていない場合、デフォルト値の -1 が割り当てられます。

フィールド	説明
cltid	クライアントの識別子が保持された CLIENTID 型の構造体。この構造体は変更できません。

プロセスが TPSVCINFO 構造体の *data* フィールドにアクセスする場合、次のバッファ・タイプが合致する必要があります。

- 呼び出し元プロセスによって渡される要求バッファのタイプ
- 呼び出されたサービスに定義された対応するバッファ・コードのタイプ
- 呼び出されたサービスに対してコンフィグレーション・ファイルに定義された対応するバッファのタイプ

次のコード例は、一般的なサービス定義を示しています。このコードは、BEA Tuxedo ソフトウェアで提供される銀行業務アプリケーションの ABAL (残高照会) サービス・ルーチンから引用したものです。ABAL サービスは BAL サーバの一部です。

リスト 5-4 一般的なサービス定義

```
#include <stdio.h>      /* UNIX */
#include <atmi.h>       /* BEA Tuxedo システム */
#include <sqlcode.h>    /* BEA Tuxedo システム */
#include "bank.flds.h"  /* bankdb フィールド */
#include "aud.h"        /* 銀行業務アプリケーションの VIEW 定義 */

EXEC SQL begin declare section;
static long branch_id;  /* 支店番号 */
static float bal;       /* 残高 */
EXEC SQL end declare section;

/*
 * あるサイトの口座残高の合計を算出するサービス
 */

void
#ifdef __STDC__
ABAL(TPSVCINFO *transb)

#else

ABAL(transb)
TPSVCINFO *transb;
```

```
#endif

{
     struct aud *transv;          /* 復号化されたメッセージの VIEW */

     /* tpsvcinfo データ・バッファへのポインタの設定 */

     transv = (struct aud *)transb->data;

     set the consistency level of the transaction

     /* メッセージから支店番号を受け取り、照会を行います。*/

     EXEC SQL declare acur cursor for
      select SUM(BALANCE) from ACCOUNT;
     EXEC SQL open acur;          /* オープン */
     EXEC SQL fetch acur into :bal;   /* フェッチ */
     if (SQLCODE != SQL_OK) {      /* 何もありません。*/
     (void)strcpy (transv->ermsg,"abal failed in sql aggregation");
      EXEC SQL close acur;
      tpreturn(TPFAIL, 0, transb->data, sizeof(struct aud), 0);
     }
     EXEC SQL close acur;
     transv->balance = bal;
     tpreturn (TPSUCCESS, 0, transb->data, sizeof(struct aud), 0);
}
```

この例では、アプリケーションがクライアント側に要求バッファを割り当てるために、tpalloc() の呼び出しで *type* パラメータに VIEW、*subtype* に aud が設定されています。ABAL サービスは、VIEW 型バッファをサポートするサービスとして定義されています。ABAL には BUFTYPE パラメータは指定されてなく、デフォルト値の ALL が使用されます。ABAL サービスは VIEW 型バッファを割り当て、このバッファへのポインタに ABAL サブルーチンが渡された TPSVCINFO 構造体の data メンバを割り当てます。ABAL サーバは、前述のコード例に示してあるように、対応する *data* メンバにアクセスして適切なデータ・バッファを取得します。

注記 このバッファが取得された後、データベースへの最初のアクセスを行う前に、サービスでトランザクションの整合性レベルを指定する必要があります。トランザクションの整合性レベルの詳細については、第 9 章の 1 ページ「グローバル・トランザクションのコーディング」を参照してください。

例 : バッファ・タイプの確認

ここで示すコード例は、サービスが TPSVCINFO 構造体に定義されているデータ・バッファにアクセスし、tptypes() 関数を使ってそのタイプを確認する方法を示しています。この処理については、第 3 章の 14 ページ「バッファ・タイプの確認」を参照してください。また、このサービスはバッファの最大サイズを確認して、そのバッファに領域を再割り当てするかどうかを判定します。

このコード例は、BEA Tuxedo ソフトウェアで提供される銀行業務アプリケーションの ABAL サービスから引用したものです。このコードでは、aud VIEW または FML のいずれかのバッファとして要求を受け入れるサービスをコーディングする方法を示しています。メッセージ・タイプの確認が失敗した場合、エラー・メッセージの文字列と戻りコードが返されます。成功した場合、バッファ・タイプに合致したコードが実行されます。tpreturn() 関数の詳細については、第 5 章の 16 ページ「サービス・ルーチンの終了」を参照してください。

リスト 5-5 バッファ・タイプの確認

```
#define TMTYPERR 1 /* tptypes が失敗したことを示す戻りコード */
#define INVALMTY 2 /* 無効なメッセージ・タイプであること示す戻りコード */

void
ABAL(transb)

TPSVCINFO *transb;

{
   struct aud *transv; /* VIEW メッセージ */
   FBFR *transf;  /* フィールド化バッファ・メッセージ */
   int repc;  /* tpgetrply の戻りコード */
   char typ[TMTYPELEN+1], subtyp[TMSTYPELEN+1]; /* メッセージのタイプとサブタイプ */
   char *retstr;  /* tptypes が失敗した場合に返される文字列 */

/* 送信されたバッファ・タイプを確認します。*/
   if (tptypes((char *)transb->data, typ, subtyp) == -1) {
      retstr=tpalloc("STRING", NULL, 100);
      (void)sprintf(retstr,
      "Message garbled; tptypes cannot tell what type message\n");
      tpreturn(TPFAIL, TMTYPERR, retstr, 100, 0);
   }
/* タイプに基づいてサービス要求を処理する方法を決定します。*/
   if (strcmp(typ, "FML") == 0) {
      transf = (FBFR *)transb->data;
```

```
... フィールド化バッファのための abal サービスを実行するコード ...
 tpreturn が成功し、応答として FML バッファを送信します。
   }
   else if (strcmp(typ, "VIEW") == 0 && strcmp(subtyp, "aud") == 0) {
      transv = (struct aud *)transb->data;
... aud 構造体のための abal サービスを実行するコード ...
 tpreturn の実行が成功し、応答として aud VIEW バッファを送信します。
   }
   else {
      retstr=tpalloc("STRING", NULL, 100);
     (void)sprintf(retstr,
     "Message garbled; is neither FML buffer nor aud view\n");
     tpreturn(TPFAIL, INVALMTY, retstr, 100, 0);
   }
}
```

例 : サービス要求の優先順位の確認

注記 優先順位を取得する tpgprio()、および優先順位を設定する tpsprio() 関数の詳細については、第 6 章の 16 ページ「メッセージの優先順位の設定および取得」を参照してください。

ここで示すコード例は、PRINTER サービスが tpgprio() ルーチンを使用して、受信したばかりの要求の優先順位を確認する方法を示しています。その後、その優先順位に基づいて印刷ジョブが適切なプリンタに送られ、そのプリンタに pbuf->data の内容がパイプされています。

アプリケーションは pbuf->flags を照会し、応答が必要かどうかを判定します。応答が必要な場合は、プリンタ名をクライアントに返します。tpreturn() 関数の詳細については、第 5 章の 16 ページ「サービス・ルーチンの終了」を参照してください。

リスト 5-6 受信した要求の優先順位の確認

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

char *roundrobin();

PRINTER(pbuf)

TPSVCINFO *pbuf;      /* 印刷バッファ */
```

```
{
char prname[20], ocmd[30];      /* プリンタ名、出力コマンド */
long rlen;                      /* 応答バッファの長さ */
int prio;                       /* 要求の優先順位 */
FILE *lp_pipe;                  /* パイプ・ファイルへのポインタ */

prio=tpgprio();
if (prio <= 20)
  (void)strcpy(prname,"bigjobs"); /* 優先順位の低い ( 冗長的 ) ジョブを
                                     大型コンピュータの中央
                                     レーザ・プリンタに送信します。そこで
                                     出力をソートし、
                                     bin に保存します。 */
else if (prio <= 60)
  (void)strcpy(prname,roundrobin()); /* ローカルの小型レーザ・プリンタの 1 つに
                                        プリンタを順に割り当てます。
                                        そのプリンタで直ちに出力されます。
                                        プリンタの
                                        リストに従って、roundrobin() が
                                        実行されます。 */
else
   (void)strcpy(prname,"hispeed");
                                   /* ジョブを高速レーザ・
                                      プリンタに割り当てます。
                                      冗長出力を毎日行う人のために
                                      予約されています。 */

(void)sprintf(ocmd, "lp -d%s", prname);   /* lp(1) 出力コマンド */
lp_pipe = popen(ocmd, "w");               /* コマンドへのパイプの作成 */
(void)fprintf(lp_pipe, "%s", pbuf->data); /* 出力の印刷 */
(void)pclose(lp_pipe);                    /* パイプのクローズ */

if ((pbuf->flags & TPNOREPLY))
   tpreturn(TPSUCCESS, 0, NULL, 0, 0);
rlen = strlen(prname) + 1;
pbuf->data = tprealloc(pbuf->data, rlen); /* 名前のために十分な領域を確保します。 */
(void)strcpy(pbuf->data, prname);
tpreturn(TPSUCCESS, 0, pbuf->data, rlen, 0);

char *
roundrobin()

{
static char *printers[] = {"printer1", "printer2", "printer3", "printer4"};
static int p = 0;
```

```
if (p > 3)
    p=0;
return(printers[p++]);
}
```

サービス・ルーチンの終了

tpreturn(3c)、tpcancel(3c)、および tpforward(3c) 関数は、サービス ルーチンが完了したことをそれぞれ次の方法で通知します。

- tpreturn() は、呼び出し元クライアントに応答を送信します。
- tpcancel() は、現在の要求を取り消します。
- tpforward() は、別の処理を行うために別のサービスにサービス要求を転送します。

応答の送信

tpreturn(3c) 関数はサービス・ルーチンの終了を示し、要求元にメッセージを送ります。tpreturn() 関数の呼び出しには、次の文法を使用します。

```
void
tpreturn(int rval, int rcode, char *data, long len, long flags)
```

次の表は、tpreturn() 関数の引数を示しています。

表 5-2 tpreturn( ) **関数の引数**

引数	説明
rval	サービスが正常に終了したかどうかをアプリケーション・レベルで示す値。この値は、シンボリック名で表される整数値です。有効な設定は、次のとおりです。 ■ TPSUCCESS—関数呼び出しが成功したことを示します。関数は、応答メッセージを呼び出し元のバッファに格納します。つまり、応答メッセージがある場合は、呼び出し元のバッファ内にあります。 ■ TPFAIL (デフォルト)—サービスが失敗したことを示します。関数は、応答を待つクライアント・プロセスにエラー・メッセージを通知します。その場合、クライアントが呼び出した tpcall( ) または tpgetrply( ) 関数が失敗し、変数 tperrno(5) にアプリケーション定義の失敗を示す TPESVCFAIL が設定されます。応答メッセージが要求されている場合、呼び出し元のバッファから取得できます。 ■ TPEXIT—サービスが失敗したことを示します。関数は、応答を待つクライアント・プロセスにエラー・メッセージを通知して終了します。 この引数の値がグローバル・トランザクションに与える影響については、第 9 章の 1 ページ「グローバル・トランザクションのコーディング」を参照してください。
rcode	アプリケーション定義の戻りコードを呼び出し元に返します。クライアントは、グローバル変数 tpurcode(5) を照会して、rcode に返された値にアクセスできます。成功または失敗に関係なく、このコードは関数から返されます。

引数	説明
data	クライアント・プロセスに返される応答メッセージを指すポインタ。メッセージ・バッファは、tpalloc() を使用して既に割り当てられていることが必要です。 SVCINFO 構造体のサービスに渡されたバッファと同じものを使用する場合は、バッファの割り当ておよび解放について意識する必要はありません。この 2 つの処理は、システムで提供される main() 関数で行われます。このバッファは、tpfree() コマンドを使用しても解放できません。このコマンドを使ってバッファの解放を試みると失敗します。tprealloc() 関数を使用すると、バッファのサイズを変更できます。 サービス・ルーチンに渡されたバッファとは別のバッファを使ってメッセージを返す場合、自分でそのバッファを割り当てる必要があります。アプリケーション側で tpreturn() 関数が呼び出されると、バッファは自動的に解放されます。 応答メッセージを返す必要がない場合は、この引数に NULL ポインタを設定します。 **注記** クライアントに応答を返す必要がない場合、つまり TPNOREPLY が設定されている場合、tpreturn() 関数は *data* と *len* 引数を無視して main() に制御を戻します。
len	応答バッファの長さ。アプリケーションはこの引数の値に、tpcall() 関数の olen パラメータ、または tpgetrply() 関数の len パラメータを使用してアクセスできます。 クライアントとして動作するプロセスは、この戻り値を使用して、応答バッファのサイズが大きくなったかどうか調べることができます。 クライアントが応答を要求していて応答バッファにデータが存在していない場合、つまり *data* 引数に NULL ポインタが設定されている場合、クライアントのバッファを変更せずに長さがゼロの応答が返されます。 *data* 引数が指定されていない場合、この引数の値は無視されます。
flag	現在使用されていません。

サービス・ルーチンの主なタスクは、要求を処理してクライアント・プロセスに応答を返すことです。ただし、要求されたタスクを行うために必要なすべての処理を 1 つのサービスで行う必要はありません。サービスは要求元として動作し、クライアントが元の要求を行ったときと同じように、tpcall() または tpacall() を呼び出して要求を別のサービスに渡すことができます。

注記 tpcall() と tpacall() 関数の詳細については、第 6 章の 1 ページ「クライアントおよびサーバへの要求 / 応答のコーディング」を参照してください。

tpreturn() が呼び出された場合、常に main() 関数に制御が戻ります。非同期応答でサービスが要求を送信している場合、main() に制御を戻す前にすべての応答を受信するか、または tpcancel() を使用して既に送信した要求を無効にする必要があります。それ以外の場合、未処理の応答は BEA Tuxedo システムの main() で受信されると自動的に破棄され、呼び出し元にエラーが返されます。

クライアントが tpcall() を使用してサービスを呼び出した場合、tpreturn() の呼び出しが成功すると、応答メッセージが *odata ポインタで示されるバッファ内に格納されます。tpacall() を使用して要求を送信し、tpreturn() から正常に制御が戻されると、応答メッセージは *data ポインタが指す tpgetrply() のバッファ内に格納されます。

応答が必要な場合に、tpreturn() の引数の処理時にエラーが発生すると、呼び出し元プロセスに失敗を示すメッセージが送信されます。呼び出し元は、tperrno に格納されている値を調べてエラーを検出します。失敗を示すメッセージが送信された場合、tperrno に TPESVCERR が設定されます。この値は、グローバル変数 tpurcode の値よりも優先されます。このようなエラーが発生した場合、応答データは返されず、呼び出し元の出力バッファの内容と長さは変更されません。

tpreturn() が不明なタイプのバッファにメッセージを返すか、または呼び出し元で使用できないバッファにメッセージを返した場合、つまり *flags* に TPNOCHANGE が設定されて呼び出しが行われた場合、tperrno(5) に TPEOTYPE が返されます。その場合、アプリケーションの成功また失敗は判定されず、呼び出し元の出力バッファの内容と長さは変更されません。

tpreturn() が呼び出され、呼び出し元が応答を待っている間にタイムアウトが発生した場合、グローバル変数 tpurcode(5) に返される値は意味を持ちません。この状況は、tperrno(5) に値が返されるどの状況よりも優先します。その場合、tperrno(5) に TPETIME が設定され、応答データは送信されず、呼び出し元の応答バッファの内容と長さは変更されません。BEA Tuxedo システムでは、ブロッキング・タイムアウトとトランザクション・タイムアウトの 2 種類のタイムアウトが発生します。詳細については、第 9 章の 1 ページ「グローバル・トランザクションのコーディング」を参照してください。

ここで示すコード例は、XFER サーバの一部である TRANSFER サービスを示しています。基本的に、TRANSFER サービスは WITHDRAWAL および DEPOSIT サービスへの同期呼び出しを行います。このサービスでは、WITHDRAWAL と DEPOSIT の両サービスの呼び出しに同じ要求バッファを使用する必要があるので、応答メッセージ用に別のバッファが割り当てられます。WITHDRAWAL の呼び出しが失敗した場合、フォーム上のステータス行に「cannot withdraw」というメッセージが出力され、応答バッファが解放され、tpreturn() 関数の *rval* 引数に TPFAIL が設定されます。呼び出しが成功した場合、振替元口座の残高が応答バッファから取得されます。

注記 次のコード例では、フィールド化バッファ transf 内の ACCOUNT_ID フィールドのゼロ番目のオカレンスに、アプリケーションが cr_id 変数から取得した「振替先口座」の識別子を移動しています。このような移動が必要なのは、FML バッファ内のフィールドのこのオカレンスが、データ依存型ルーティングに使用されるからです。詳細については、『BEA Tuxedo アプリケーションの設定』を参照してください。

withdrawal サービス呼び出しのシナリオは、DEPOSIT サービスへの呼び出しにも適用できます。呼び出しが成功すると、このサービスによってサービス・ルーチンに割り当てられていた応答バッファが解放され、*rval* 引数に TPSUCCESS が設定されて、適切な残高情報がステータス行に返されます。

リスト 5-7 tpreturn( ) **関数**

```
#include <stdio.h>      /* UNIX */
#include <string.h>     /* UNIX */
#include "fml.h"        /* BEA Tuxedo システム */
#include "atmi.h"       /* BEA Tuxedo システム */
#include "Usysflds.h"   /* BEA Tuxedo システム */
#include "userlog.h"    /* BEA Tuxedo システム */
#include "bank.h"       /* 銀行業務アプリケーションのマクロ定義 */
#include "bank.flds.h"  /* bankdb フィールド */


/*
 * 振替元口座から振替先口座に
 * 振り替えるサービス
 */

void
#ifdef __STDC__
TRANSFER(TPSVCINFO *transb)

#else

TRANSFER(transb)
TPSVCINFO *transb;
#endif

{
   FBFR *transf;          /* 復号化されたメッセージのフィールド化バッファ */
   long db_id, cr_id;     /* 振替元口座と振替先口座の番号               */
   float db_bal, cr_bal;  /* 振替元口座と振替先口座の残高          */
   float tamt;            /* 振替額             */
```

```
   FBFR *reqfb;           /* 要求メッセージ用のフィールド化バッファ */
   int reqlen;            /* フィールド化バッファの長さ           */
   char t_amts[BALSTR];   /* 振替額を表す文字列         */
   char db_amts[BALSTR];  /* 振替元の口座残高を表す文字列  */
   char cr_amts[BALSTR];  /* 振替先の口座残高を示す文字列 */

/* tpsvcinfo データ・バッファを指すポインタを設定します。*/
transf = (FBFR *)transb->data;

/* 振替元と振替先の口座番号を取得します。*/

/* 振替元の口座番号が有効かどうかを確認します。*/
if (((db_id = Fvall(transf, ACCOUNT_ID, 0)) < MINACCT) || (db_id > MAXACCT)) {
   (void)Fchg(transf, STATLIN, 0,"Invalid debit account number",(FLDLEN)0);
   tpreturn(TPFAIL, 0, transb->data, 0L, 0);
}
/* 振替先の口座番号が妥当かどうかを確認します。*/
if ((cr_id = Fvall(transf, ACCOUNT_ID, 1)) < MINACCT || cr_id > MAXACCT) {
   (void)Fchg(transf,STATLIN, 0,"Invalid credit account number",(FLDLEN)0);
   tpreturn(TPFAIL, 0, transb->data, 0L, 0);
}

/* 振替金額 ( 払い戻し金額 ) を取得します。*/
if (Fget(transf, SAMOUNT, 0, t_amts, <  0) 0 || strcmp(t_amts,"") == 0) {
   (void)Fchg(transf, STATLIN, 0, "Invalid amount",(FLDLEN)0);
   tpreturn(TPFAIL, 0, transb->data, 0L, 0);
}
(void)sscanf(t_amts,"%f",tamt);

/* 振替金額が妥当かどうかを確認します。*/
if (tamt = 0.0) {
   (void)Fchg(transf, STATLIN, 0,
      "Transfer amount must be greater than $0.00",(FLDLEN)0);
   tpreturn(TPFAIL, 0, transb->data, 0L, 0);
}

/* 引出要求バッファを作成します。*/
if ((reqfb = (FBFR *)tpalloc("FML",NULL,transb->len)) == (FBFR *)NULL) {
   (void)userlog("tpalloc failed in transfer\n");
   (void)Fchg(transf, STATLIN, 0,
      "unable to allocate request buffer", (FLDLEN)0);
   tpreturn(TPFAIL, 0, transb->data, 0L, 0);
}
reqlen = Fsizeof(reqfb);

/* 要求バッファに口座番号を格納します。*/
(void)Fchg(reqfb,ACCOUNT_ID,0,(char *)&db_id, (FLDLEN)0);
```

```
/* 要求バッファに振替金額を格納します。*/
(void)Fchg(reqfb,SAMOUNT,0,t_amts, (FLDLEN)0);

/* WITHDRAWAL サービス呼び出しの優先順位を上げます。*/
if (tpsprio(PRIORITY, 0L) == -1)
   (void)userlog("Unable to increase priority of withdraw\n");

if (tpcall("WITHDRAWAL", (char *)reqfb,0, (char **)&reqfb,
      (long *)&reqlen,TPSIGRSTRT) == -1) {
   (void)Fchg(transf, STATLIN, 0,
         "Cannot withdraw from debit account", (FLDLEN)0);
   tpfree((char *)reqfb);
   tpreturn(TPFAIL, 0, transb->data, 0L, 0);
}

/* 応答バッファから振替元口座の残高を取得します。*/

(void)strcpy(db_amts, Fvals((FBFR *)reqfb,SBALANCE,0));
void)sscanf(db_amts,"%f",db_bal);
if ((db_amts == NULL) || (db_bal < 0.0)) {
   (void)Fchg(transf, STATLIN, 0,
         "illegal debit account balance", (FLDLEN)0);
   tpfree((char *)reqfb);
   tpreturn(TPFAIL, 0, transb->data, 0L, 0);
}

/* 要求バッファに振替先の口座番号を格納します。*/
(void)Fchg(reqfb,ACCOUNT_ID,0,(char *)&cr_id, (FLDLEN)0);

/* 要求バッファに振替金額を格納します。*/
(void)Fchg(reqfb,SAMOUNT,0,t_amts, (FLDLEN)0);

/* 振替 ( 預入 ) の関数の呼び出しの優先順位を上げます。*/
if (tpsprio(PRIORITY, 0L) == -1)
     (void)userlog("Unable to increase priority of deposit\n");

/* tpcall を実行して、振替先口座に振替金額を移動します。*/
if (tpcall("DEPOSIT", (char *)reqfb, 0, (char **)&reqfb,
        (long *)&reqlen, TPSIGRSTRT) == -1) {
   (void)Fchg(transf, STATLIN, 0,
           "Cannot deposit into credit account", (FLDLEN)0);
   tpfree((char *)reqfb);
   tpreturn(TPFAIL, 0, transb->data, 0L, 0);
}

/* 応答バッファから振替先口座の残高を取得します。*/
```

```
(void)strcpy(cr_amts, Fvals((FBFR *)reqfb,SBALANCE,0));
(void)sscanf(cr_amts,"%f",&cr_bal);
if ((cr_amts == NULL) || (cr_bal 0.0)) {
   (void)Fchg(transf, STATLIN, 0,
         "Illegal credit account balance", (FLDLEN)0);
   tpreturn(TPFAIL, 0, transb->data, 0L, 0);
}

/* 振替の成功を示す値を格納する応答バッファを設定します。*/
(void)Fchg(transf, FORMNAM, 0, "CTRANSFER", (FLDLEN)0);
(void)Fchg(transf, SAMOUNT, 0, Fvals(reqfb,SAMOUNT,0), (FLDLEN)0);
(void)Fchg(transf, STATLIN, 0, "", (FLDLEN)0);
(void)Fchg(transf, SBALANCE, 0, db_amts, (FLDLEN)0);
(void)Fchg(transf, SBALANCE, 1, cr_amts, (FLDLEN)0);
tpfree((char *)reqfb);
tpreturn(TPSUCCESS, 0,transb->data, 0L, 0);
}
```

記述子の無効化

tpgetrply() を呼び出したサービス (第 6 章の 1 ページ「クライアントおよびサーバへの要求 / 応答のコーディング」を参照) が TPETIME で失敗して要求を取り消す場合、tpcancel(3c) を呼び出して記述子を無効にできます。以降、応答が届いても自動的に破棄されます。

tpcancel() 関数の呼び出しには、次の文法を使用します。

```
void
tpcancel(int cd)
```

cd (呼び出し記述子) 引数には、取り消すプロセスを指定します。

tpcancel() はトランザクション応答、つまり TPNOTRAN フラグが設定されていない状態で呼び出された要求への応答には使用できません。トランザクション内では、tpabort(3c) がトランザクションの呼び出し記述子を無効にします。

次のコード例は、タイムアウト後の応答を無効にする方法を示しています。

リスト 5-8 タイムアウト後の応答の無効化

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

要求の転送

tpforward(3c) 関数を使用すると、サービス要求をほかのサービスに転送して、別の処理を行うことができます。

tpforward() 関数の呼び出しには、次の文法を使用します。

```
void
tpforward(char *svc, char *data, long len, long flags)
```

次の表は、tpreturn() 関数の引数を示しています。

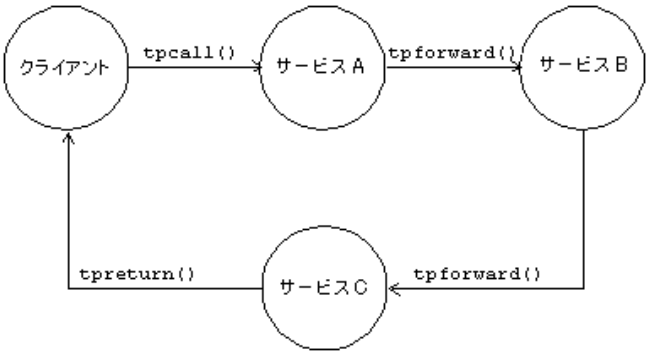
表 5-3 tpreturn( ) **関数の引数**

引数	説明
svc	要求が転送されるサービス名を指す文字型のポインタ。
data	クライアント・プロセスに返される応答メッセージを指すポインタ。メッセージ・バッファは、tpalloc( ) を使用して既に割り当てられていることが必要です。 SVCINFO 構造体のサービスに渡されたバッファと同じものを使用する場合は、バッファの割り当ておよび解放について意識する必要はありません。この 2 つの処理は、システムで提供される main( ) 関数で行われます。このバッファは、tpfree( ) コマンドを使用しても解放できません。このコマンドを使ってバッファの解放を試みると失敗します。tprealloc( ) 関数を使用すると、バッファのサイズを変更できます。 サービス・ルーチンに渡されたバッファとは別のバッファを使ってメッセージを返す場合、自分でそのバッファを割り当てる必要があります。アプリケーション側で tpreturn( ) 関数が呼び出されると、バッファは自動的に解放されます。 応答メッセージを返す必要がない場合は、この引数に NULL ポインタを設定します。 **注記** クライアントに応答を返す必要がない場合、つまり TPNOREPLY が設定されている場合、tpreturn( ) 関数は *data* と *len* 引数を無視して main( ) に制御を戻します。
len	応答バッファの長さ。アプリケーションはこの引数の値に、tpcall( ) 関数の olen パラメータ、または tpgetrply( ) 関数の len パラメータを使用してアクセスできます。 クライアントとして動作するプロセスは、この戻り値を使用して、応答バッファのサイズが大きくなったかどうか調べることができます。 クライアントが応答を要求していて応答バッファにデータが存在していない場合、つまり *data* 引数に NULL ポインタが設定されている場合、クライアントのバッファを変更せずに長さがゼロの応答が返されます。 *data* 引数が指定されていない場合、この引数の値は無視されます。
flag	現在使用されていません。

tpforward() は、サービス呼び出しとは異なります。つまり、要求の転送元サービスでは、応答は要求されていません。応答を返すのは、要求の転送先サービスです。このサービスを転送されたサービスが、要求の発信元プロセスに応答を返します。転送が連鎖的に行われる場合、最後のサーバが tpreturn() を呼び出して、要求の発信元であるクライアントに応答を返します。

次の図は、あるサーバから別のサーバに要求を転送したときのイベントの流れを示しています。この例では、クライアントは tpcall() 関数を使用して要求を開始し、連鎖の最後のサービス (サービス C) が tpreturn() 関数を使用して応答を返しています。

図 5-1 要求の転送



サービス・ルーチンは tpsprio() 関数を使用して、クライアント・プロセスが要求を送るのと同じように、指定された優先順位に従って要求を転送できます。

プロセスが tpforward() を呼び出すと、システムで提供された main() に制御が戻り、サーバ・プロセスは別の要求を処理できるようになります。

注記 クライアントとして動作するサーバ・プロセスが応答を要求する場合、このサーバが自分自身からサービスを要求することはできません。つまり、必要なサービスの唯一のインスタンスが要求を行っているサーバ・プロセスからのみ提供される場合、その呼び出しは失敗して再帰呼び出しができないことが示されます。ただし、TPNOREPLY 通信フラグが設定された状態でサービス・ルーチンが自分宛てに要求を送信または転送した場合、サービスは自分からの応答を待機しないので、呼び出しは失敗しません。

tpforward() 呼び出しを使用して、その呼び出しを行った時点まで要求の処理が成功していたことを示すことができます。アプリケーション・エラーが検出されなかった場合、tpforward() を呼び出します。エラーが検出された場合、*rval* に TPFAIL を設定して tpreturn() を呼び出します。

次のコード例は、ACCT サーバの一部である OPEN_ACCT サービス・ルーチンから引用したものです。この例は、tpforward() を呼び出して、サービスがそのデータ・バッファを DEPOSIT サービスに送る方法を示しています。このコードは、SQLCODE をテストして、口座の挿入が成功したかどうかを調べています。新規口座の追加が成功した場合、支店レコードが更新されてその口座が反映され、データ・バッファが DEPOSIT サービスに転送されます。失敗した場合、*rval* に TPFAIL が設定されて tpreturn() が呼び出され、失敗を示すメッセージがフォーム上のステータス行に出力されます。

リスト 5-9　tpforward( ) **関数**

```
 ...
/* tpsvcinfo データ・バッファへのポインタを設定します。*/
transf = (FBFR *)transb->data;
...
/* account に新規口座レコードを挿入します。*/
account_id = ++last_acct;     /* 新規口座番号を取得します。*/
tlr_bal = 0.0;                /* 一時的に残高をゼロにします。*/
EXEC SQL insert into ACCOUNT (ACCOUNT_ID, BRANCH_ID, BALANCE,
ACCT_TYPE, LAST_NAME, FIRST_NAME, MID_INIT, ADDRESS, PHONE) values
(:account_id, :branch_id, :tlr_bal, :acct_type, :last_name,
      :first_name, :mid_init, :address, :phone);
if (SQLCODE != SQL_OK) {    /* レコードの挿入が失敗しました。*/
      (void)Fchg(transf, STATLIN, 0,
          "Cannot update ACCOUNT", (FLDLEN)0);
      tpreturn(TPFAIL, 0, transb->data, 0L, 0);
}

/* 新規 last_acct で支店レコードを更新します。*/

EXEC SQL update BRANCH set LAST_ACCT = :last_acct where BRANCH_ID = :branch_id;
if (SQLCODE != SQL_OK) {      /* レコードの更新が失敗しました。*/
      (void)Fchg(transf, STATLIN, 0,
           "Cannot update BRANCH", (FLDLEN)0);
      tpreturn(TPFAIL, 0, transb->data, 0L, 0);
}
/* 振替 ( 預金 ) の関数の呼び出しの優先順位を上げます。*/
if (tpsprio(PRIORITY, 0L) == -1)
     (void)userlog("Unable to increase priority of deposit\n");

/* tpforward を使用して同じバッファを預金サービスに転送し、最初の残高を設定します。*/
tpforward("DEPOSIT", transb->data, 0L, 0);
```

サービスの宣言と宣言の取り消し

サーバは起動時に、コンフィギュレーション・ファイルの CLOPT パラメータに指定された値に基づいて、提供するサービスを宣言します。

注記 サーバが宣言するサービスは、buildserver コマンドの実行時に最初に定義されます。-s オプションを使用すると、複数のサービスをカンマ区切りで指定できます。また、宣言されたサービスと異なる名前の関数を呼び出して、サービス要求を処理できます。詳細については、『BEA Tuxedo コマンド・リファレンス』の buildserver(1) を参照してください。

デフォルトでは、サーバに組み込まれたすべてのサービスをそのサーバが宣言します。詳細については、『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 UBBCONFIG(5) または servopts(5) リファレンス・ページを参照してください。

宣言されたサービスでは掲示板のサービス・テーブル・エントリが使用されるので、リソースが消費される場合があります。そのため、サーバの起動時には、提供されるサービスのサブセットだけを利用できるようにします。アプリケーションで利用できるサービスを制限するには、コンフィギュレーション・ファイルの SERVERS セクションで該当するエントリに CLOPT パラメータを定義し、-s オプションの後に必要なサービスをカンマで区切って指定します。また、-s オプションを使用すると、サービス要求を処理するために呼び出される宣言済みのサービスと異なる名前の関数を呼び出すこともできます。詳細については、『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 servopts(5) リファレンス・ページを参照してください。

BEA Tuxedo アプリケーションの管理者は、tmadmin(1) の advertise および unadvertise コマンドを使用して、サーバで提供されるサービスを管理できます。tpadvertise() および tpunadvertise() 関数を使用すると、要求 / 応答型サーバまたは会話型サーバでのサービスの宣言を動的に制御できます。ただし、宣言されるサービス、または宣言を取り消すサービスは、要求を行うサービスと同じサーバ内になければなりません。

サービスの宣言

tpadvertise(3c) 関数の呼び出しには、次の文法を使用します。

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

次の表は、tpadvertise( ) 関数の引数を示しています。

表 5-4 tpadvertise( ) 関数の引数

引数	説明
svcname	宣言するサービス名を指すポインタ。サービス名は 15 文字以下の文字列で指定します。15 文字を超える名前は切り捨てられます。NULL 文字列は指定できません。NULL 文字列が指定されると、エラー (TPEINVAL) になります。
func	サービスを実行するために呼び出される BEA Tuxedo システム関数のアドレスを指すポインタ。通常、この名前とサービス名は同じです。NULL 文字列は指定できません。NULL 文字列が指定されると、エラーになります。

サービス宣言の取り消し

tpunadvertise(3c) 関数は、掲示板のサービス・テーブルからサービス名を削除します。サービス名が削除されたサービスは、宣言されていない状態になります。

tpunadvertise( ) 関数の呼び出しには、次の文法を使用します。

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

tpunadvertise( ) 関数の引数は、次の表で説明する *svcname* だけです。

表 5-5 tpunadvertise( ) 関数の引数

引数	説明
svcname	宣言するサービス名を指すポインタ。サービス名は 15 文字以下の文字列で指定します。15 文字を超える名前は切り捨てられます。NULL 文字列は指定できません。NULL 文字列が指定されると、エラー (TPEINVAL) になります。

例 : サービスの動的な宣言と宣言の取り消し

次のコード例は、tpadvertise( ) 関数の使用方法を示しています。このコードでは、サーバ TLR が起動時に TLR_INIT サービスだけを提供するようにコーディングされています。初期化後、TLR_INIT は DEPOSIT と WITHDRAW という 2 つのサービスを宣言します。両サービスとも tlr_funcs 関数によって実行され、サーバ TLR に組み込まれています。

DEPOSIT と WITHDRAW を宣言した後、TLR_INIT は自分自身で宣言を取り消します。

リスト 5-10 動的な宣言と宣言の取り消し

```
extern void tlr_funcs()
 .
 .
 .
 if (tpadvertise("DEPOSIT", (tlr_funcs)(TPSVCINFO *)) == -1)
             check for errors;
 if (tpadvertise("WITHDRAW", (tlr_funcs)(TPSVCINFO *)) == -1)
       check for errors;
 if (tpunadvertise("TLR_INIT") == -1)
       check for errors;
 tpreturn(TPSUCCESS, 0, transb->data,0L, 0);
```

サーバのビルド

実行可能なサーバをビルドするには、buildserver(1) コマンドを使用して、BEA Tuxedo System サーバ・アダプタなどすべての参照ファイルと共にアプリケーション・サービス・サブルーチンをコンパイルします。

注記 BEA Tuxedo サーバ・アダプタは、メッセージの受信、処理のディスパッチ、トランザクションが有効な場合はトランザクションの管理を行います。

buildserver コマンドには、次の構文を使用します。

```
buildserver -C -o filename -f filenames -l filenames -r rmname -s -v
```

次の表は、buildserver コマンド行オプションを示しています。

表 5-6 buildserver コマンド行オプション

オプション	説明
-o *filename*	実行可能な出力ファイル名。デフォルト値は a.out です。
-f *filenames*	BEA Tuxedo システム・ライブラリより先にリンクされるファイルのリスト。-f オプションは複数回指定できます。また、各 -f に複数のファイル名を指定できます。C プログラム・ファイル (*file*.c) を指定すると、リンクされる前にコンパイルが行われます。ほかのオブジェクト・ファイル (*file*.o) を個別に、またはアーカイブ・ファイル (*file*.a) にまとめて指定することもできます。
-l *filenames*	Tuxedo システム・ライブラリの後でリンクされるファイルのリスト。-l オプションは複数回指定できます。また、各 -l に複数のファイル名を指定できます。C プログラム・ファイル (*file*.c) を指定すると、リンクされる前にコンパイルが行われます。ほかのオブジェクト・ファイル (*file*.o) を個別に、またはアーカイブ・ファイル (*file*.a) にまとめて指定することもできます。

表 5-6 buildserver **コマンド行オプション**

オプション	説明
-r *rmname*	実行可能サーバにリンクされるリソース・マネージャのアクセス・ライブラリのリスト。アプリケーション管理者は、buildtms(1) コマンドを使用して、すべての有効なリソース・マネージャ情報を $TUXDIR/updataobj/RM ファイルに事前に定義しておく必要があります。リソース・マネージャは 1 つしか指定できません。詳細については、『BEA Tuxedo アプリケーションの設定』を参照してください。
-s[*service*:]*function*	サーバに提供されるサービス名、および各サービスを実行する関数名。-s オプションは複数回指定できます。また、各 -s に複数のサービスを指定できます。サーバは指定されたサービス名を使用して、クライアントにサービスを宣言します。 通常、サービスとそのサービスを実行する関数には同じ名前を割り当てます。ただし、別の名前を指定することもできます。名前の割り当てには、*service*:*function* という構文を使用します。
-t	サーバがスレッドセーフな方法でコーディングされており、コンフィギュレーション・ファイルでマルチスレッドとして指定されている場合は、マルチスレッドとして起動することを示す値。

注記 BEA Tuxedo ライブラリは自動的にリンクされます。コマンド行に BEA Tuxedo ライブラリ名を指定する必要はありません。

リンクするライブラリ・ファイルの指定順序は重要です。関数を呼び出す順序と、これらの関数への参照を含むライブラリによって、この順序が決定されます。

デフォルトでは、buildserver コマンドは UNIX の cc コマンドを呼び出します。環境変数 CC を指定して別のコンパイル・コマンドを指定したり、CFLAGS を指定してコンパイル・フェーズやリンク・フェーズに独自のフラグを指定することができます。詳細については、第 2 章の 4 ページ「環境変数の設定」を参照してください。

次のコマンドは、acct.o アプリケーション・ファイルを処理して、NEW_ACCT と CLOSE_ACCT という 2 つのサービスを含む ACCT サーバを作成しています。NEW_ACCT は OPEN_ACCT 関数を呼び出し、CLOSE_ACCT は 同じ名前の関数を呼び出します。

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

関連項目

- 第 4 章の 9 ページ「クライアントのビルド」
- 『BEA Tuxedo コマンド・リファレンス』の buildclient(1)

C++ コンパイラ

アプリケーション・サーバの開発時に C++ コンパイラを使用する場合と、C コンパイラを使用する場合では、基本的に次の 2 点が異なります。

- サービス関数の宣言方法
- コンストラクタとデストラクタの使用方法

サービス関数の宣言

C++ コンパイラでサービス関数を宣言する場合、extern "C" を使用して、"C" リンクを持つサービス関数を宣言しなければなりません。次のように、関数のプロトタイプを指定します。

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

C リンクを持つサービス名を宣言すると、C++ コンパイラによって名前が変更されなくなります。多くの C++ コンパイラでは、パラメータおよび関数の戻り値の型情報を含むように関数名が変更されます。

このように宣言すると、次の操作が可能になります。

- C と C++ の両方のサービス・ルーチンを 1 つのサーバにリンクできます。その場合、各ルーチンのタイプを示す必要はありません。
- 動的なサービス宣言を行うことができます。動的なサービス宣言では、実行可能ファイルのシンボル・テーブルにアクセスして、関数名を取得する必要があります。

コンストラクタとデストラクタ

C++ のコンストラクタは、クラス・オブジェクトを作成するときに呼び出されて、クラス・オブジェクトを初期化します。デストラクタは、クラス・オブジェクトを破棄するときに呼び出されます。コンストラクタとデストラクタを持つ自動 (ローカルで非静的な) 変数では、変数がスコープに入るときにコンストラクタが呼び出され、変数がスコープから出るときにデストラクタが呼び出されます。ただし、tpreturn() または tpforward() 関数 が呼び出されると、longjmp(3) を使用して、非ローカルの goto がコンパイラによって実行され、自動変数のデストラクタは呼び出されません。この問題を防ぐには、tpreturn() や tpforward() の呼び出しが、サービス・ルーチンから呼び出される関数からではなく、サービス・ルーチンから直接行われるようにアプリケーションをコーディングします。また、次のいずれかの条件を満たす必要があります。

- サービス・ルーチンがデストラクタ付きの自動変数を持たないようにします。自動変数は、サービス・ルーチンによって呼び出される関数内で宣言して使用します。
- 自動変数は、tpreturn() または tpforward() 関数を呼び出す前にスコープが終了するように、ネストされた ({} で囲まれた) スコープ内で宣言して使用します。

つまり、tpreturn() または tpforward() 関数を呼び出すときに、現在の関数のスコープ内またはスタック上にデストラクタ付を持つ自動変数が存在しないように、アプリケーションを定義します。

コンストラクタとデストラクタを持つグローバル変数および静的変数を正しく処理するために、多くの C++ コンパイラでは main() のコンパイルに C++ コンパイラを使用する必要があります。

注記 main() ルーチンには、プログラム開始時にコンストラクタ、プログラム終了時にデストラクタを確実に実行するための特別な処理が定義されています。

main() は BEA Tuxedo システムで提供される関数なので、直接コンパイルされることはありません。ファイルが C++ を使用してコンパイルされるには、buildserver コマンドで C++ コンパイラを使用する必要があります。デフォルトでは、buildserver コマンドは UNIX の cc コマンドを呼び出します。buildserver コマンドから C++ コンパイラが呼び出されるようにするには、CC 環境変数に C++ コンパイラの絶対パス名を設定します。また、C++ のコマンド行で指定するオプションにフラグを設定するには、CFLAGS 環境変数を設定します。詳細については、第 2 章の 4 ページ「環境変数の設定」を参照してください。

6 クライアントおよびサーバへの要求 / 応答のコーディング

ここでは、次の内容について説明します。

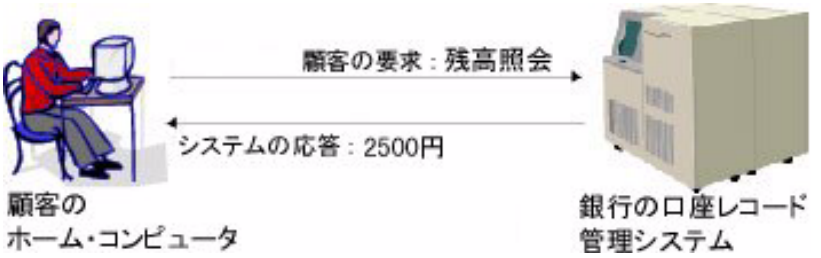
- 要求 / 応答通信の概要
- 同期メッセージの送信
- 非同期メッセージの送信
- メッセージの優先順位の設定および取得

要求 / 応答通信の概要

要求 / 応答通信モードでは、あるソフトウェア・モジュールが別のソフトウェア・モジュールに要求を送り、応答を待ちます。最初のソフトウェア・モジュールがクライアント、2 番目のソフトウェア・モジュールがサーバとして動作するので、このモードはクライアント / サーバ相互作用とも呼ばれます。オンラインの銀行業務の多くは、要求 / 応答モードでプログラミングされます。たとえば、残高照会の要求は、次のように行われます。

1. 顧客 (クライアント) は、口座レコード管理システム (サーバ) に、残高照会の要求を送信します。
2. 口座レコード管理システム (サーバ) は、指定された口座の残高を応答として顧客 (クライアント) に送ります。

図 6-1 オンライン銀行業務での応答 / 要求通信



クライアント・プロセスがアプリケーションに参加し、バッファを割り当て、入力データ要求をバッファに格納すると、そのプロセスは要求メッセージをサービス・サブルーチンに送信したり、応答メッセージを受信することができるようになります。

同期メッセージの送信

`tpcall(3c)` 関数は、サービス・サブルーチンに要求を送信し、同期的に応答を待ちます。`tpcall()` 関数の呼び出しには、次の文法を使用します。

```
int
tpcall(char *svc, char *idata, long ilen, char **odata, long *olen, long
flags)
```

次の表は、`tpcall()` 関数の引数を示しています。

表 6-1 tpcall( ) 関数の引数

引数	説明
svc	アプリケーションで提供されるサービス名を指すポインタ。

引数	説明
idata	要求のデータ部分のアドレスを含むポインタ。このポインタは、tpalloc() の以前の呼び出しで割り当てられた型付きバッファを参照しなければなりません。*idata* の type (および指定されている場合は subtype) は、サービス・ルーチンで使用できる type (および指定されている場合は subtype) と合致する必要があります。タイプが合致しない場合、tperrno に TPEITYPE が設定され、関数の呼び出しが失敗します。 データが要求されていない場合、*idata* に NULL ポインタを設定します。NULL ポインタが設定されていると、パラメータは無視されます。要求でデータが送信されない場合、*idata* のバッファを割り当てる必要はありません。
ilen	*idata* で参照されるバッファ内の要求データの長さ。バッファが自己記述型の場合、つまり FML、FML32、VIEW、VIEW32、X_COMMON、X_C_TYPE、または STRING 型バッファの場合、この引数にゼロを設定すると引数が無視されます。
**odata*	応答を受信する出力バッファを指すポインタのアドレス。tpalloc() 関数を使用して、出力バッファを割り当てておく必要があります。tpcall() から正常に制御が戻ったときに、応答メッセージにデータが含まれていないと、*olen にゼロが設定されます。ポインタと出力バッファの内容は変更されません。 要求メッセージと応答メッセージの両方に同じバッファを使用できます。その場合、**odata* には、入力バッファを割り当てたときに返されたポインタのアドレスを設定します。このパラメータが NULL を指している場合は、エラーになります。
olen	応答データの長さを指すポインタ。このパラメータが NULL を指している場合は、エラーになります。
flags	フラグのオプション。論理演算子 OR を使用すると、複数のフラグをリストできます。この値にゼロを設定すると、デフォルトの方法で通信が行われます。有効なフラグとデフォルト値については、『BEA Tuxedo C リファレンス』の tpcall(3c) を参照してください。

tpcall() は、応答を待ちます。

注記 tpcall() 関数の呼び出しは、tpacall() を呼び出した直後に tpgetrply() を呼び出すことと論理的には同じです。第 6 章の 11 ページ「非同期メッセージの送信」を参照してください。

要求は、*svc* で指定されたサービスの優先順位で送信されます。ただし、tpsprio() の呼び出しで明示的に異なる優先順位が設定されている場合は除きます。詳細については、第 6 章の 16 ページ「メッセージの優先順位の設定および取得」を参照してください。

tpcall() は整数を返します。失敗した場合、この整数値は -1 になり、tperrno(5) 発生したエラー条件が返されます。有効なエラー・コードの詳細については、『BEA Tuxedo C リファレンス』の tpcall(3c) を参照してください。

注記 通信呼び出しはいろいろな原因で失敗します。そのほとんどは、アプリケーション・レベルで修正することができます。失敗の原因としては、アプリケーション定義のエラー (TPESVCFAIL)、戻り値の処理エラー (TPESVCERR)、型付きバッファのエラー (TPEITYPE、TPEOTYPE)、タイムアウト・エラー (TPETIME)、プロトコル・エラー (TPEPROTO) などがあります。エラーの詳細については、第 11 章の 1 ページ「エラーの管理」を参照してください。発生する可能性があるエラーについては、『BEA Tuxedo C リファレンス』の tpcall(3c) を参照してください。

BEA Tuxedo システムでは、割り当てられているバッファより大きなメッセージを受信した場合、メッセージ受信用バッファのサイズが自動的に変更されます。そのため、応答バッファのサイズが変更されたかどうかを確認する必要があります。

バッファの新しいサイズにアクセスするには、**olen* パラメータに返されたアドレスを使用します。応答バッファのサイズが変更されたかどうかを確認するには、tpcall() を呼び出す前の応答バッファのサイズと、返された応答バッファの *olen の値とを比較します。*olen が元の値より大きい場合、バッファのサイズは大きくなっています。それ以外の場合、バッファのサイズは変更されていません。

呼び出しの後に *odata* に戻された値で、出力バッファを参照します。サイズの増加以外に、出力バッファが変更されている場合があるからです。要求バッファのサイズを確認する必要はありません。要求データは一度割り当てられると、調整されないからです。

注記 要求メッセージと応答メッセージに同じバッファを使用している場合に、バッファのサイズが変更されたために応答バッファを指すポインタが変更されたときは、入力バッファへのポインタは有効なアドレスを参照しなくなります。

例 : 要求メッセージと応答メッセージに同じバッファを使用する

以下のコード例は、クライアント・プログラム audit.c で、要求メッセージと応答メッセージに同じバッファを使用して、同期呼び出しを行う方法を示しています。この例では、*audv メッセージ・バッファは要求情報と応答情報の両方を格納するように設定されているので、同じバッファを使用することができます。このコードでは、次の処理が行われます。

1. サービスは b_id フィールドを照会します。ただし、このフィールドを上書きしません。
2. アプリケーションは、bal フィールドをゼロ、ermsg フィールドを NULL 文字列に初期化して、サービスから返される値を受け取る準備をします。
3. svc_name 変数は要求されたサービス名、hdr_type 変数は要求された残高のタイプを示しします。この例では、口座残高と窓口残高が示されます。

リスト 6-1 要求メッセージと応答メッセージに同じバッファを使用する

```
. . .
/* バッファを作成し、データ・ポインタを設定します。*/

audv = (struct aud *)tpalloc("VIEW", "aud", sizeof(struct aud));

       /* aud 構造体を用意します。*/

audv->b_id = q_branchid;
audv->balance = 0.0;
(void)strcpy(audv->ermsg, "");

        /* tpcall の実行 */

if (tpcall(svc_name,(char *)audv,sizeof(struct aud),
        (char **)&audv,(long *)&audrl,0)== -1){
        (void)fprintf (stderr, "%s service failed\n %s:%s\n",
        svc_name, svc_name, audv->ermsg);
        retc = -1;
}
else
       (void)printf ("Branch %ld %s balance is $%.2f\n",
           audv->b_id, hdr_type, audv->balance);
. . .
```

例 : 応答バッファのサイズ変更の確認

次のコード例は、tpcall() を呼び出した後、アプリケーションでバッファのサイズが変更されたかどうかを確認する方法を示しています。この例では、入力バッファと出力バッファは同じサイズでなければなりません。

リスト 6-2 応答バッファのサイズ変更の確認

```
char *svc, *idata, *odata;
long ilen, olen, bef_len, aft_len;
. . .
if (idata = tpalloc("STRING", NULL, 0) == NULL)
   error

if (odata = tpalloc("STRING", NULL, 0) == NULL)
   error

place string value into idata buffer

ilen = olen = strlen(idata)+1;
. . .
bef_len = olen;
if (tpcall(svc, idata, ilen, &odata, &olen, flags) == -1)
   error

aft_len = olen;

if (aft_len > bef_len){  /* メッセージ・バッファのサイズが大きくなっている場合 */

   if (idata = tprealloc(idata, olen) == NULL)
     error
}
```

例 :TPSIGRSTRT フラグを設定した同期メッセージの送信

以下のコード例は、bankapp の XFER サーバ・プロセスの一部である TRANSFER サービスに基づいています。bankapp は、BEA Tuxedo システムに提供されている銀行業務のサンプル・アプリケーションです。TRANSFER サービスは、クライアントとして WITHDRAWAL および DEPOSIT サービスを呼び出します。アプリケーションはこの 2 つのサービスを呼び出すときに通信フラグを TPSIGRSTRT に設定して、トランザクションをコミットしやすいようにします。TPSIGRSTRT フラグは、シグナルの割り込みがあった場合に行う処理を指定します。通信フラグの詳細については『BEA Tuxedo C リファレンス』の tpcall(3c) を参照してください。

リスト 6-3　TPSIGRSTRT **フラグを設定した同期メッセージの送信**

```
     /* tpcall を実行して、振替元の口座から引き出します。*/

if (tpcall("WITHDRAWAL", (char *)reqfb,0, (char **)&reqfb,
     (long *)&reqlen,TPSIGRSTRT) == -1) {
     (void)Fchg(transf, STATLIN, 0,
     "Cannot withdraw from debit account", (FLDLEN)0);
     tpfree((char *)reqfb);
}
...
     /* tpcall を実行して、振替先口座に振替金額を移動します。*/

if (tpcall("DEPOSIT", (char *)reqfb, 0, (char **)&reqfb,
     (long *)&reqlen, TPSIGRSTRT) == -1) {
     (void)Fchg(transf, STATLIN, 0,
     "Cannot deposit into credit account", (FLDLEN)0);
     tpfree((char *)reqfb);
}
```

例 :TPNOTRAN フラグを設定した同期メッセージの送信

以下のコード例は、トランザクション・モードではない通信呼び出しを示しています。この呼び出しは、リソース・マネージャに関連していないサービスに対して行われます。サービスがトランザクションに参加するとエラーになります。アプリケーションは、データベース accounts から取得した情報に基づいて生成された売掛金勘定レポート accrcv を出力します。

サービス・ルーチン REPORT は指定されたパラメータを解釈し、完了したレポートのバイト・ストリームを応答として送信します。クライアントは、tpcall() を使用して PRINTER サービスにバイト・ストリームを送信します。PRINTER は、クライアントに近いプリンタにバイト・ストリームを送信します。そして、応答が印刷されます。最後に、PRINTER サービスはハードコピーの印刷が終了したことをクライアントに通知します。

注記 第 6 章の 13 ページ「TPNOTRAN と TPNOREPLY フラグを設定した非同期メッセージの送信」では、同じ例を使用して非同期メッセージの呼び出しを行っています。

リスト 6-4 TPNOTRAN フラグを設定した同期メッセージの送信

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

main()

{
char *rbuf;                 /* レポート用バッファ */
long r1len, r2len, r3len;  /* レポート用の送信バッファ、最初の応答バッファ、
                              および 2 番目の応答バッファの長さ */
join application

if (rbuf = tpalloc("STRING", NULL, 0) == NULL) /* レポート用の領域の割り当て
*/
   leave application and exit program
(void)strcpy(rbuf,
   "REPORT=accrcv DBNAME=accounts"); /* レポートのパラメータの送信 */
r1len = strlen(rbuf)+1;             /* 要求の長さ */

start transaction
```

```
if (tpcall("REPORT", rbuf, r1len, &rbuf,
   &r2len, 0) == -1)                /* レポート印刷ストリームの取得 */
   error routine
if (tpcall("PRINTER", rbuf, r2len, &rbuf,
   &r3len, TPNOTRAN) == -1)         /* プリンタへのレポートの送信 */
   error routine
(void)printf("Report sent to %s printer\n",
   rbuf);                           /* プリンタの識別 */

terminate transaction
free buffer
アプリケーションを終了
}
```

注記 この例の error routine は、エラー・メッセージの出力、トランザクションの中止、割り当てられたバッファの解放、クライアントのアプリケーションからの分離、およびプログラムの終了が行われることを示しています。

例 :TRNOCHANGE フラグを設定した同期メッセージの送信

以下のコード例では、最初に割り当てられたバッファ・タイプと同じタイプで応答メッセージを返す必要があることを示して、TPNOCHANGE 通信フラグを使用して厳密なタイプ・チェックを行う方法を示しています。この例では、REPORT というサービス・ルーチンを参照します。REPORT サービスは、第 6 章の 8 ページ「例 :TPNOTRAN フラグを設定した同期メッセージの送信」でも使用されています。

このコード例では、クライアントが応答を VIEW 型バッファ rview1 で受信し、printf() 文でその内容を出力しています。厳密なタイプ・チェックのフラグ TPNOCHANGE が設定されているので、タイプ VIEW およびサブタイプ rview1 のバッファに応答が返されます。

厳密なタイプ・チェックを行うのは、REPORT サービス・サブルーチンでエラーが発生して、不適切なタイプの応答バッファが使用されることを防ぐためです。もう 1 つの理由は、依存関係にあるすべてのエリアで一貫していない変更が行われることを防ぐためです。たとえば、あるプログラマが REPORT サービスを変更してすべての応答を別の VIEW 形式で標準化したが、それを反映するためにクライアント・プロセスを変更しなかった場合などがあります。

リスト 6-5 TRNOCHANGE **フラグを設定した同期メッセージの送信**

```
#include <stdio.h>
#include "atmi.h"
#include "rview1.h"

main(argc, argv)
int argc;
char *argv[];

{
char *rbuf;             /* レポート用バッファ */
struct rview1 *rrbuf;   /* 応答バッファの通知 */
long rlen, rrlen;       /* レポート用の送信バッファと
                           受信バッファの長さ */
if (tpinit((TPINIT *) tpinfo) == -1)
   fprintf(stderr, "%s:failed to join application\n", argv[0]);

if (rbuf = tpalloc("STRING", NULL, 0) == NULL) { /* レポート用の領域の割り当て
*/
   tpterm();
   exit(1);
}
                        /* 応答バッファ用の領域の割り当て */
if (rrbuf = (struct rview1 *)tpalloc("VIEW", "rview1",
sizeof(struct rview1)) \ == NULL{
   tpfree(rbuf);
   tpterm();
   exit(1);
}
(void)strcpy(rbuf, "REPORT=accrcv DBNAME=accounts FORMAT=rview1");
rlen = strlen(rbuf)+1;    /* 要求の長さ */
                          /* rview1 構造体でのレポートの取得 */
if (tpcall("REPORT", rbuf, rlen, (char **)&rrbuf, &rrlen, TPNOCHANGE) ==
-1) {
   fprintf(stderr, "accounts receivable report failed in service
call\n");
   if (tperrno == TPEOTYPE)
      fprintf(stderr, "report returned has wrong view type\n");
   tpfree(rbuf);
   tpfree(rrbuf);
   tpterm();
   exit(1);
}
(void)printf("Total accounts receivable %6d\n", rrbuf->total);
(void)printf("Largest three outstanding %-20s %6d\n", rrbuf->name1,
rrbuf->amt1);
```

```
(void)printf("%-20s %6d\n", rrbuf->name2, rrbuf->amt2);
(void)printf("%-20s %6d\n", rrbuf->name3, rrbuf->amt3);
tpfree(rbuf);
tpfree(rrbuf);
 tpterm();
}
```

非同期メッセージの送信

この節では、次の操作を行う方法について説明します。

- tpacall() 関数を使用した非同期要求の送信
- tpgetrply() 関数を使用した非同期応答の取得

ここで説明する非同期の処理は、ファンアウト並列処理と呼ばれます。クライアントの要求が複数のサービスに同時に分散 (つまり「ファンアウト」) されて処理が行われるからです。

このほかに、BEA Tuxedo システムでは、非同期処理としてパイプライン並列処理もサポートされています。この処理では、tpforward() 関数を使用して 1 つのサービスから別のサービスに処理が渡されます (転送されます)。tpforward() 関数については、第 5 章の 1 ページ「サーバのコーディング」を参照してください。

非同期要求の送信

tpacall(3c) 関数は、サービス要求を送信し、直ちに制御を戻します。tpacall() 関数の呼び出しには、次の文法を使用します。

```
int
tpacall(char *svc, char *data, long len, long flags)
```

次の表は、tpacall() 関数の引数を示しています。

表 6-2 tpacall( ) 関数の引数

引数	説明
svc	アプリケーションで提供されるサービス名を指すポインタ。
data	要求のデータ部分のアドレスを含むポインタ。このポインタは、tpalloc() の以前の呼び出しで割り当てられた型付きバッファを参照しなければなりません。*idata* の type (および指定されている場合は subtype) は、サービス・ルーチンで使用できる type (および指定されている場合は subtype) と合致する必要があります。タイプが合致しない場合、tperrno に TPEITYPE が設定され、関数の呼び出しが失敗します。 データが要求されていない場合、*data* に NULL ポインタを設定します。NULL ポインタが設定されていると、パラメータは無視されます。要求でデータが信されていない場合は、*data* のバッファを割り当てる必要はありません。
len	*data* で参照されるバッファ内の要求データの長さ。バッファが自己記述型の場合、つまり FML、FML32、VIEW、VIEW32、X_COMMON、X_C_TYPE、または STRING 型バッファの場合、この引数にゼロを設定すると引数が無視されます。
flags	フラグのオプション。論理演算子 OR を使用すると、複数のフラグをリストできます。この値にゼロを設定すると、デフォルトの方法で通信が行われます。有効なフラグとデフォルト値については、『BEA Tuxedo C リファレンス』の tpacall(3c) を参照してください。

tpacall( ) 関数は、*svc* パラメータに指定されたサービスに要求メッセージを送信し、直ちに制御を戻します。呼び出しが正常に終了すると、tpacall( ) 関数は整数値を返します。この値は、関連する要求に対する正しい応答にアクセスするための記述子として使用されます。tpacall( ) がトランザクション・モードで実行されている場合 (第 9 章の 1 ページ「グローバル・トランザクションのコーディング」を参照)、トランザクションのコミット時に未処理の応答が存在することはありません。つまり、あるトランザクションの範囲内では、要求ごとにその応答が返されるので、最終的には対応する応答を必ず受信することになります。

flags 引数に TPNOREPLY が設定されると、応答が必要ないことが tpacall() に通知されます。このフラグが設定されている場合、tpacall() の処理が正常に終了すると、応答記述子として 0 が返されます。以降の処理で、この値が tpgetrply() 関数に渡されると、この値は無効になります。プロセスがトランザクション・モードのときにこのフラグを正しく使用するためのガイドラインについては、第 9 章の 1 ページ「グローバル・トランザクションのコーディング」を参照してください。

エラーが発生した場合、tpacall() は -1 を返し、tperrno(5) 発生したエラー条件が返されます。tpacall() が返すエラー・コードの多くは、tpcall() が返すエラー・コードと同じです。この 2 つの関数のエラー・コードは、一方が同期呼び出し、もう一方が非同期呼び出しに基づいているという点が異なります。これらのエラーについては、第 11 章の 1 ページ「エラーの管理」を参照してください

例 : TPNOTRAN と TPNOREPLY フラグを設定した非同期メッセージの送信

以下のコード例は、tpacall() で TPNOTRAN と TPNOREPLY フラグを使用する方法を示しています。このコードは、第 6 章の 8 ページ「例 :TPNOTRAN フラグを設定した同期メッセージの送信」のコードと同じです。ただし、ここで示すコードでは、PRINTER サービスからの応答は要求されていません。TPNOREPLY フラグはクライアントが応答を要求していないこと、TPNOTRAN フラグは PRINTER サービスが現在のトランザクションに参加しないことを示します。詳細については、第 11 章の 1 ページ「エラーの管理」を参照してください。

リスト 6-6　TPNOTRAN と TPNOREPLY フラグを設定した非同期メッセージの送信

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

main()

{
char *rbuf;          /* レポート用バッファ */
long rlen, rrlen;    /* レポート用の送信バッファと受信バッファの長さ */

  join application

if (rbuf = tpalloc("STRING", NULL, 0) == NULL) /* レポート用の領域の割り当て
*/
 error
(void)strcpy(rbuf, "REPORT=accrcv DBNAME=accounts");/* レポートのパラメータ
の送信 */
```

```
rlen = strlen(rbuf)+1;  /* 要求の長さ */

start transaction

if (tpcall("REPORT", rbuf, rlen, &rbuf, &rrlen, 0)
   == -1) /* レポート印刷ストリームの取得 */
   error
if (tpacall("PRINTER", rbuf, rrlen, TPNOTRAN|TPNOREPLY)
   == -1) /* プリンタへのレポートの送信 */
   error

. . .
commit transaction
free buffer
アプリケーションを終了
}
```

例 : 非同期要求の送信

以下のコード例は、銀行の総残高照会を行う非同期呼び出しを示しています。銀行業務アプリケーションのデータは、複数のデータベース・サイトに分散されているので、各サイトに対して SQL クエリを実行する必要があります。その場合、データベース・サイトごとに支店番号が 1 つ選択され、そのサイトに対して ABAL または TBAL サービスが呼び出されます。支店番号は実際の SQL クエリでは使用されませんが、この番号によって BEA Tuxedo システムが適切なサイトに要求を送ることができるようになります。次のコードでは、サイトごとに for ループが tpacall() を一度呼び出しています。

リスト 6-7 非同期要求の送信

```
audv->balance = 0.0;
(void)strcpy(audv->ermsg, "");

for (i=0; i<NSITE; i++) {

  /* aud 構造体を用意します。*/

  audv->b_id = sitelist[i];/* このフィールドでルーティングを行います。*/

  /* tpacall の実行 */

  if ((cd[i]=tpacall(sname, (char *)audv, sizeof(struct aud), 0))
     == -1) {
```

```
    (void)fprintf (stderr,
       "%s:%s service request failed for site rep %ld\n",
       pgmname, sname, sitelist[i]);
       tpfree((char *)audv);
       return(-1);
  }
}
```

非同期応答の受信

サービス呼び出しに対する応答は、tpgetrply(3c) 関数を呼び出すと非同期的に受信できます。tpgetrply() 関数は、tpacall() が以前に送信した要求に対する応答をキューから取り出します。

tpgetrply() 関数の呼び出しには、次の文法を使用します。

```
int
tpgetrply(int *cd, char **data, long *len, long flags)
```

次の表は、tpgetrply() 関数の引数を示しています。

表 6-3 tpgetrply( ) 関数の引数

引数	説明
cd	tpacall() が返す呼び出し記述子を指すポインタ。
**data*	応答を受信する出力バッファを指すポインタのアドレス。tpalloc() 関数を使用して、出力バッファを割り当てておく必要があります。tpcall() から正常に制御が戻ったときに、応答メッセージにデータが含まれていないと、*data にゼロが設定されます。ポインタと出力バッファの内容は変更されません。 要求メッセージと応答メッセージの両方に同じバッファを使用できます。その場合、*odata* には、入力バッファを割り当てたときに返されたポインタのアドレスを設定します。このパラメータが NULL を指している場合は、エラーになります。
len	応答データの長さを指すポインタ。このパラメータが NULL を指している場合は、エラーになります。

引数	説明
flags	フラグのオプション。論理演算子 OR を使用すると、複数のフラグをリストできます。この値にゼロを設定すると、デフォルトの方法で通信が行われます。有効なフラグとデフォルト値については、『BEA Tuxedo C リファレンス』の tpcall(3c) を参照してください。

デフォルトでは、この関数は、cd パラメータが参照する値に対応した応答を待ちます。応答を待っている間に、ブロッキング・タイムアウトが発生する場合があります。タイムアウトが発生するのは、tpgetrply() が失敗し、tperrno(5) に TPETIME が設定された場合です。ただし、*flags* パラメータに TPNOTIME が設定されている場合は除きます。

メッセージの優先順位の設定および取得

tpsprio(3c) と tpgprio(3c) の 2 つの ATMI 関数を使用して、メッセージ要求の優先順位を決定したり設定できます。この優先順位に従って、サーバがキューから要求を取り出します。つまり、最も優先順位の高い要求が最初に取り出されます。

この節では、次の内容について説明します。

- メッセージの優先順位の設定
- メッセージの優先順位の取得

メッセージの優先順位の設定

tpsprio(3c) 関数を使用すると、メッセージ要求の優先順位を設定できます。

tpsprio() ルーチンで優先順位を設定できるのは、1 つの要求だけです。つまり、tpcall() または tpacall() によって次に送信される要求、またはサービス・サブルーチンによって次に転送される要求だけです。

tpsprio() 関数の呼び出しには、次の文法を使用します。

```
int
tpsprio(int prio, long flags);
```

次の表は、tpsprio() 関数の引数を示しています。

表 6-4 tpsprio( ) 関数の引数

引数	説明
prio	新しい優先順位を示す整数値。この引数の持つ意味は、*flags* パラメータによって異なります。*flags* にゼロが設定されている場合、*prio* は相対値を示し、値の符号は現在の優先順位を上げることまたは下げることを示します。ほかの値が設定されている場合、指定された値は絶対値を示し、*prio* には 0 ～ 100 の範囲の値を設定する必要があります。この範囲外の値を指定すると、値が自動的に 50 に設定されます。
flags	*prio* の値を相対値 (0) または絶対値 (TPABSOLUTE) のどちらの値として処理するのかを示すフラグ。デフォルトは相対値です。

以下のコード例は、TRANSFER サービスから引用したものです。このコードでは、TRANSFER サービスはクライアントとして動作し、tpcall() を使用して WITHDRAWAL サービスに同期要求を送信しています。TRANSFER は tpsprio() を呼び出して WITHDRAWAL に対する要求メッセージの優先順位を上げます。また、TRANSFER のキューを待機した後で、WITHDRAWAL サービス (その後 DEPOSIT サービス) に対する要求がキューに格納されないようにします。

リスト 6-8 要求メッセージの優先順位の設定

```
/* WITHDRAWAL サービス呼び出しの優先順位を上げます。*/
if (tpsprio(PRIORITY, 0L) == -1)
   (void)userlog("Unable to increase priority of withdraw\n");

if (tpcall("WITHDRAWAL", (char *)reqfb,0, (char **)&reqfb, (long *)
\
       &reqlen,TPSIGRSTRT) == -1) {
   (void)Fchg(transf, STATLIN, 0, "Cannot withdraw from debit account", \
       (FLDLEN)0);
   tpfree((char *)reqfb);
   tpreturn(TPFAIL, 0, transb->data, 0L, 0);
}
```

メッセージの優先順位の取得

tpgprio(3c) を使用すると、メッセージ要求の優先順位を取得できます。

tpgprio() 関数の呼び出しには、次の文法を使用します。

```
int
tpgprio();
```

要求元は、tpcall() または tpacall() 関数を呼び出した後に tpgprio() 関数を呼び出して、要求メッセージの優先順位を取得できます。要求元が関数を呼び出したが要求が送信されていない場合、関数は失敗して -1 が返され、tperrno(5) に TPENOENT が設定されます。tpgprio() の処理が成功すると、1 ～ 100 の範囲内の整数値が返されます。100 が最も高い優先順位です。

tpsprio() 関数を使用して優先順位が明示的に設定されていない場合、メッセージの優先順として、要求を処理するサービス・ルーチンの優先順位が設定されます。アプリケーション内では、要求を処理するサービスの優先順位にデフォルト値の 50 が設定されます。ただし、システム管理者が別の値を指定している場合は除きます。

次のコード例は、非同期呼び出しによって送信されたメッセージの優先順位を確認する方法を示しています。

リスト 6-9 送信後の要求の優先順位の確認

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

main ()
{
int cd1, cd2;              /* 呼び出し記述子 */
int pr1, pr2;              /* 2 つの呼び出しに対する優先順位 */
char *buf1, *buf2;         /* バッファ */
long buf1len, buf2len;     /* バッファの長さ */

join application

if (buf1=tpalloc("FML", NULL, 0) == NULL)
   error
if (buf2=tpalloc("FML", NULL, 0) == NULL)
   error

populate FML buffers with send request

if ((cd1 = tpacall("service1", buf1, 0, 0)) == -1)
   error
```

```
if ((pr1 = tpgprio()) == -1)
   error
if ((cd2 = tpacall("service2", buf2, 0, 0)) == -1)
   error
if ((pr2 = tpgprio()) == -1)\
   error

if (pr1 >= pr2) {   /* 呼び出しの優先順位に基づいて、tpgetreply の順序を設定し
ます。*/
        if (tpgetrply(&cd1, &buf1, &buf1len, 0) == -1)
           error
        if (tpgetrply(&cd2, &buf2, &buf2len, 0) == -1)
           error
}
else {
        if (tpgetrply(&cd2, &buf2, &buf2len, 0) == -1)
           error
        if (tpgetrply(&cd1, &buf1, &buf1len, 0) == -1)
           error
}
. . .
}
```

7　会話型クライアントおよびサーバのコーディング

ここでは、次の内容について説明します。

- 会話型通信の概要
- アプリケーションへの参加
- 接続の確立
- メッセージの送受信
- 会話の終了
- 会話型のクライアントおよびサーバのビルド
- 会話型通信イベント

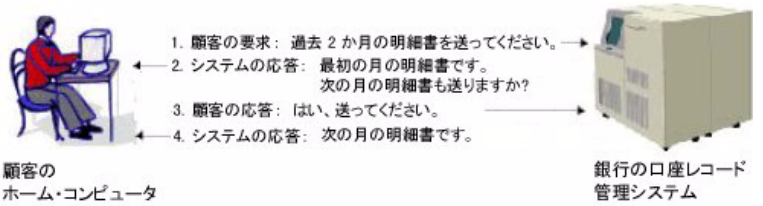
会話型通信の概要

会話型通信は BEA Tuxedo システムのメッセージ交換のパラダイムで、人の会話に似た通信がクライアントとサーバ間でインプリメントされています。この通信方法では、クライアント (イニシエータ) とサーバ (下位サーバ) 間で仮想接続が行われて、双方で会話の状態に関する情報が保持されます。この接続は、接続を終了するイベントが発生するまで継続します。

会話型通信では、クライアントとサーバ間に半二重接続が確立されます。半二重接続では、メッセージが 1 方向だけに送信されます。接続に関する制御は、イニシエータから下位サーバへ、またはその逆に移ります。制御を持つプロセスがメッセージを送信でき、持たないプロセスは受信しかできません。

以下に銀行業務のオンライン・アプリケーションを例に、BEA Tuxedo アプリケーションで行われる会話型通信について説明します。この例では、銀行の顧客が過去 2 か月間の当座預金の明細書を要求しています。

図 7-1 銀行業務オンライン・アプリケーションでの会話型通信



1. 顧客が過去 2 か月間の当座預金口座の明細書を要求します。
2. 口座レコード管理システムは、この要求に対する応答として、当座預金口座の最初の月の明細書を送信します。次に、次の月の明細書を送信するかどうかを `More` プロンプトで確認します。
3. 顧客は `More` プロンプトを選択して、次の月の明細書を要求します。

   注記 口座レコード管理システムでは、状態情報を保持して、顧客が `More` プロンプトを選択した場合にどの明細書を送るのか認識できるようにする必要があります。
4. 口座レコード管理システムは、次の月の明細書を送信します。

要求 / 応答型通信の場合と同じように、BEA Tuxedo システムでは型付きバッファを使用してデータが渡されます。アプリケーションがバッファ・タイプを認識できることが必要です。バッファ・タイプの詳細については、第 3 章の 2 ページ「型付きバッファの概要」を参照してください。

会話型のクライアントおよびサーバには、次の特徴があります。

- 会話型のクライアントとサーバ間の論理接続は、接続が終了するまで継続します。
- 会話型のクライアントとサーバ間の接続で転送できるメッセージの数には制限はありません。
- クライアントとサーバとの会話では、データの送受信に `tpsend()` および `tprecv()` ルーチンが使用されます。

会話型通信は、次の点で要求 / 応答型通信と異なります。

- 会話型クライアントは、サービスを要求するときに `tpcall()` または `tpacall()` ではなく、`tpconnect()` を使用します。
- 会話型クライアントは、会話型サーバにサービス要求を送信します。
- 会話型サービスを定義するために、コンフィギュレーション・ファイルに会話型サーバの一部が予約されています。
- 会話型サーバは、`tpforward()` を使用して呼び出しを行うことはできません。

アプリケーションへの参加

会話型クライアントは、サービスへの接続を確立する前に、tpinit() を呼び出してアプリケーションに参加する必要があります。詳細については、第 4 章の 1 ページ「クライアントのコーディング」を参照してください。

接続の確立

tpconnect(3c) 関数は、会話を行うための接続を確立します。

tpconnect() 関数の呼び出しには、次の文法を使用します。

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

次の表は、tpconnect() 関数の引数を示しています。

表 7-1 tpconnect( ) 関数の引数

引数	説明
name	会話型サービス名を指す文字型のポインタ。*name* に会話型サービスを指すポインタが指定されていない場合、呼び出しが失敗して -1 が返され、tperrno にエラー・コード TPENOENT が設定されます。
data	データ・バッファを指すポインタ。*data* 引数に、tpalloc() を使用して既に割り当てられているバッファを指すポインタを設定すると、接続の確立と同時にデータを送信することができます。バッファのタイプとサブタイプは、呼び出されたサービスで認識できなければなりません。*data* に NULL を設定して、送信データがないことを示すことがきます。 呼び出された会話型サービスは、TPSVCINFO 構造体を介して *data* および *len* ポインタを受け取ります。この構造体は、サービスが呼び出されたときに main() 関数によって渡されます。要求 / 応答型サービスも同じ方法で、*data* および *len* ポインタを受け取ります。TPSVCINFO 構造体の詳細については、第 5 章の 9 ページ「サービスの定義」を参照してください。
len	データ・バッファの長さ。バッファが自己記述型 (FML バッファなど) の場合、*len* に 0 を設定できます。

引数	説明
flag	フラグの設定値。有効なフラグの設定値については『BEA Tuxedo C リファレンス』の tpconnect(3c) を参照してください。 システムは、TPSVCINFO 構造体のフラグ・メンバを使用して呼び出されたサービスに通知します。

tpconnect() によって接続が確立されると、BEA Tuxedo システムから接続記述子 (*cd*) が返されます。この *cd* は、特定の会話で以降に送られるメッセージを識別するために使用されます。クライアントまたは会話型サービスは、複数の会話に同時に参加できます。最大 64 個の会話を同時に行うことができます。

tpconnect() 関数の呼び出しが失敗すると -1 が返され、対応するエラー・コードが tperrno に設定されます。エラー・コードについては、『BEA Tuxedo C リファレンス』の tpconnect(3c) を参照してください。

次のコード例は、tpconnect() 関数の使用方法を示しています。

リスト 7-1 会話型接続の確立

```
#include atmi.h
#define    FAIL    -1
int cd1;        /* 接続記述子 */
main()
{
 if ((cd = tpconnect("AUDITC",NULL,0,TPSENDONLY)) == -1) {
   error routine
 }
}
```

メッセージの送受信

BEA Tuxedo システムで会話型接続が確立されると、イニシエータと下位サーバ間の通信は送信呼び出しと受信呼び出しによって行われます。接続の制御を持つプロセスは、tpsend(3c) 関数を使用してメッセージを送信できます。制御がないプロセスは、tprecv(3c) 関数を使用してメッセージを受信できます。

注記 発信元 (クライアント) は、最初に tpconnect( ) 呼び出しの TPSENDONLY または TPRECVONLY フラグを使用して、どのプロセスが制御を持っているのかを判別します。TPSENDONLY は、発信元が制御を持つことを示します。TPRECVONLY は、呼び出されたサービスに制御が渡されたことを示します。

メッセージの送信

メッセージを送信するには tpsend(3c) 関数を使用します。この関数には、次の文法を使用します。

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

次の表は、tpsend( ) 関数の引数を示しています。

表 7-2 tpsend( ) 関数の引数

引数	説明
cd	tpconnect( ) 関数で返される接続記述子。データが送信される接続が識別されます。
data	データ・バッファを指すポインタ。*data* 引数に、tpalloc( ) を使用して既に割り当てられているバッファを指すポインタを設定すると、接続の確立と同時にデータを送信することができます。バッファのタイプとサブタイプは、呼び出されたサービスで認識できなければなりません。*data* に NULL を設定して、送信データがないことを示すことがきます。 呼び出された会話型サービスは、TPSVCINFO 構造体を介して *data* および *len* ポインタを受け取ります。この構造体は、サービスが呼び出されたときに main( ) 関数によって渡されます。要求 / 応答型サービスも同じ方法で、*data* および *len* ポインタを受け取ります。 TPSVCINFO 構造体の詳細については、第 5 章の 9 ページ「サービスの定義」を参照してください。

引数	説明
len	データ・バッファの長さ。バッファが自己記述型 (FML バッファなど) の場合、*len* に 0 を設定できます。*data* に値が設定されていない場合、この引数は無視されます。
revent	エラーの発生時 (つまり、tperrno(5) に TPEEVENT が設定された場合) に、設定されるイベント値を指すポインタ。有効なイベント値については、『BEA Tuxedo C リファレンス』の tpsend(3c) を参照してください。
flag	フラグの設定値。有効なフラグ設定値については、『BEA Tuxedo C リファレンス』の tpsend(3c) を参照してください。

tpsend() 関数の呼び出しが失敗すると -1 が返され、対応するエラー・コードが tperrno(5) に設定されます。エラー・コードについては、『BEA Tuxedo C リファレンス』の tpsend(3c) を参照してください。

tpsend() 関数を呼び出すたびに、制御を渡す必要はありません。一部のアプリケーションでは、tpsend() の呼び出しを認められているプロセスが、制御をほかのプロセスに渡すまで、現在のタスクで必要な回数だけ呼び出しを実行できます。ただし、プログラムのロジックによっては、会話が継続する間は常に 1 つのプロセスが接続の制御を持たなければならないアプリケーションもあります。

次のコード例は、tpsend() 関数の呼び出し方法を示しています。

リスト 7-2 会話モードでのデータ送信

```
if (tpsend(cd,line,0,TPRECVONLY,revent) == -1) {
              (void)userlog("%s:tpsend failed tperrno %d",
                     argv[0],tperrno);
              (void)tpabort(0);
              (void)tpterm();
              exit(1);
       }
```

メッセージの受信

オープン接続を介してデータを受信するには、tprecv(3c) 関数を使用します。この関数には、次の文法を使用します。

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

次の表は、tprecv() 関数の引数を示しています。

引数	説明
cd	接続記述子。下位プログラムで呼び出しを行う場合は、*cd* 引数に、そのプログラムの TPSVCINFO 構造体に指定されている値を設定します。発信元のプログラムで呼び出しを行う場合は、*cd* 引数に、tpconnect() 関数で返される値を設定します。
data	データ・バッファを指すポインタ。*data* 引数は、tpalloc() を使用して既に割り当てられているバッファを指す必要があります。バッファのタイプとサブタイプは、呼び出されたサービスで認識できなければなりません。NULL は指定できません。NULL を指定すると呼び出しは失敗し、tperrno(5) に TPEINVAL が設定されます。 呼び出された会話型サービスは、TPSVCINFO 構造体を介して *data* および *len* ポインタを受け取ります。この構造体は、サービスが呼び出されたときに main() 関数によって渡されます。要求 / 応答型サービスも同じ方法で、*data* および *len* ポインタを受け取ります。TPSVCINFO 構造体の詳細については、第 5 章の 9 ページ「サービスの定義」を参照してください。
len	データ・バッファの長さ。バッファが自己記述型 (FML バッファなど) の場合、*len* に 0 を設定できます。NULL は指定できません。NULL を指定すると呼び出しは失敗し、tperrno(5) に TPEINVAL が設定されます。
revent	エラーの発生時 (つまり、tperrno に TPEEVENT が設定された場合) に、設定されるイベント値を指すポインタ。有効なイベント値については、『BEA Tuxedo C リファレンス』の tprecv(3c) を参照してください。
flag	フラグの設定値。有効なフラグについては、『BEA Tuxedo C リファレンス』の tprecv(3c) を参照してください。

処理が成功すると、*data が受信データを指し、*len* にバッファのサイズが格納されます。*len* が tprecv() を呼び出す前のバッファの合計サイズより大きい場合、バッファのサイズは変更されていて、*len* はバッファの新しいサイズを示しています。*len* 引数が 0 の場合、受信データがないことを示します。

次のコード例は、tprecv() 関数の使用方法を示しています。

リスト 7-3 会話型でのデータ受信

```
if (tprecv(cd,line,len,TPNOCHANGE,revent) != -1) {
            (void)userlog("%s:tprecv failed tperrno %d revent %ld",
                 argv[0],tperrno,revent);
             (void)tpabort(0);
             (void)tpterm();
             exit(1);
}
```

会話の終了

次の場合、接続が切断されて会話が正常に終了します。

- 単純な会話で、tpreturn() の呼び出しが成功した場合
- 接続が階層構造になった複雑な会話で、一連の tpreturn() の呼び出しが成功した場合
- グローバル・トランザクションの場合 (第 9 章の 1 ページ「グローバル・トランザクションのコーディング」を参照)

注記 tpreturn() 関数については、第 6 章の 1 ページ「クライアントおよびサーバへの要求 / 応答のコーディング」を参照してください。

以下の節では、会話を正常に終了する方法について、2 つの例を挙げて説明します。これらの会話には、tpreturn() 関数を使用するグローバル・トランザクションは含まれていません。

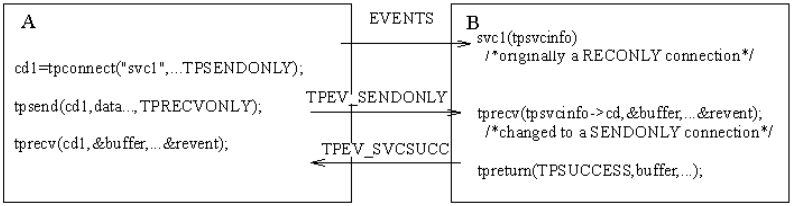
最初の例では、2 つのコンポーネント間の単純な会話を終了する方法を示します。2 番目の例では、会話が階層構造になっている複雑な会話を終了する方法を示します。

接続がオープンになっているときに会話を終了すると、エラーが返されます。その場合、tpcommit() または tpreturn() は失敗します。

例：単純な会話の終了

次の図は、正常に終了する A と B 間の単純な会話を示しています。

図 7-2　正常に終了する単純な会話



次の順序で処理が行われます。

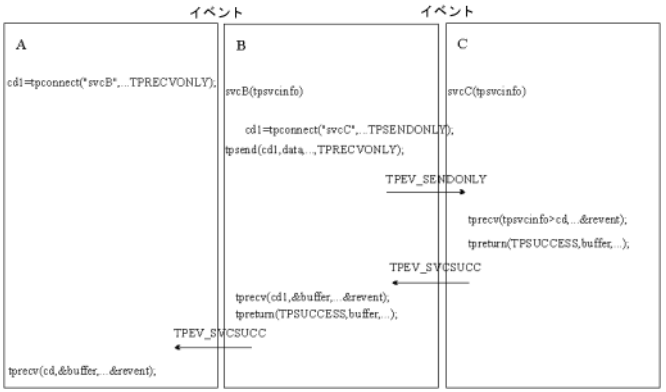
1. A は、TPSENDONLY フラグで tpconnect() を呼び出して接続を設定します。このフラグは、B が会話の受信側であることを示します。
2. A は TPRECVONLY フラグで tpsend() を呼び出して、接続の制御を B に移します。その結果、TPEV_SENDONLY イベントが生成されます。
3. B が次に tprecv() を呼び出すと -1 が返され、tperrno(5) に TPEEVENT が設定されてます。*revent* 引数に TPEV_SENDONLY が返されて、制御が B に移ったことが示されます。
4. B は、*rval* に TPSUCCESS を設定して tpreturn() を呼び出します。この呼び出しにより、A に対して TPEV_SVCSUCC イベントが生成され、接続が正常に切断されます。
5. A は、tprecv() を呼び出して、イベントが発生したことと会話が終了したことを認識します。イベント TPEV_SVCFAIL が発生した場合でも、この tprecv() への呼び出しでデータを受信できます。

注記　この例では、A はクライアントまたはサーバのどちらでもかまいませんが、B はサーバでなければなりません。

例：階層構造の会話の終了

次の図は、正常に終了する階層構造の会話を示しています。

図 7-3　接続の階層構造



この例では、サービス B は会話のメンバで、2 番目のサービス C との接続を開始しています。つまり、A - B 間 と B - C 間という 2 つのアクティブな接続が存在しています。B がこの両方の接続を制御している場合に `tpreturn()` の呼び出しを行うと、呼び出しは失敗し、すべてのオープン接続に TPEV_SVCERR イベントが通知され、接続が切断されます。

両方の接続を正常に終了するには、次の処理を順に行います。

1. B は、C との接続に TPRECVONLY フラグを設定して `tpsend()` を呼び出し、B - C 間の接続の制御を C に渡します。
2. C は、状況に応じて *rval* に TPSUCCESS、TPFAIL、または TPEXIT を設定して、`tpreturn()` を呼び出します。
3. B は、`tpreturn()` を呼び出し、A にイベント (TPEV_SVCSUCC または TPEV_SVCFAIL) を通知します。

注記　会話型サービスは、別のサービスと通信するために要求 / 応答型の呼び出しを行うことができます。そのため、前述の例では、B から C への呼び出しに `tpconnect()` ではなく `tpcall()` または `tpacall()` を使用することもできます。ただし、会話型サービスが `tpforward()` を呼び出すことはできません。

会話の切断

エラーの発生により接続を終了する唯一の方法は、tpdiscon(3c) 関数を呼び出すことです。これは、「プラグを抜くこと」と同じです。この関数を呼び出すことができるのは、会話のイニシエータ (クライアント) だけです。

注記 この方法で会話を終了することはお勧めしません。アプリケーションを正常に終了するには、下位サーバで tpreturn() 関数を呼び出します。

tpdiscon() 関数の呼び出しには、次の文法を使用します。

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

cd 引数は、接続が確立したときに tpconnect() 関数によって返される接続記述子を指定します。

tpdiscon() 関数は、接続の相手側のサービスに対して TPEV_DISCONIMM イベントを生成し、*cd* を無効にします。トランザクションが実行中の場合、そのトランザクションはアボートし、データは失われます。

cd で接続の開始側と識別されていないサービスから tpdiscon() が呼び出されると、その呼び出しは失敗し、エラー・コード TPEBADDESC が生成されます。

イベントおよびエラー・コードの全リストとその説明については、『BEA Tuxedo C リファレンス』の tpdiscon(3c) を参照してください。

会話型のクライアントおよびサーバのビルド

次のコマンドを使用して、会話型のクライアントおよびサーバをビルドします。

- buildclient() (第 4 章の 9 ページ「クライアントのビルド」を参照)
- buildserver() (第 5 章の 31 ページ「サーバのビルド」を参照)

会話型サービスと要求 / 応答型サービスでは、次の操作を行うことはできません。

- 同じサーバにクライアントとサーバを作成すること
- クライアントとサーバに同じ名前を付けること

会話型通信イベント

BEA Tuxedo システムの会話型通信では、5 つのイベントが認識されます。これらのイベントはすべて tprecv() に通知でき、そのうちの 3 つは tpsend() にも通知できます。

次の表は、イベント、そのイベントを受け取る関数、および各イベントの簡単な説明を示しています。

表 7-3 会話型通信のイベント

イベント	イベントを受け取る関数	説明
TPEV_SENDONLY	tprecv()	接続の制御が渡されました。この時点で、このプロセスは tpsend() を呼び出すことができます。
TPEV_DISCONIMM	tpsend(), tprecv(), tpreturn()	接続は既に切断され、通信を継続することはできません。tpdiscon() 関数はこのイベントを接続の開始側に通知します。下位サービスとの接続がオープンしたままになっている場合に、tpreturn() が呼び出されたときは、このイベントをすべてのオープン接続に送信します。接続はエラーが原因で切断されます。トランザクションが存在している場合は、アボートします。
TPEV_SVCERR	tpsend()	接続の開始側が受信します。通常は、下位プログラムが接続の制御を持たない場合に、tpreturn() を呼び出したことを示します。
	tprecv()	接続の開始側が受信します。下位プログラムが TPSUCCESS または TPFAIL、および妥当なデータ・バッファを使用して tpreturn() を呼び出したが、エラーが発生して呼び出しが完了しなかったことを示します。

表 7-3 会話型通信のイベント

イベント	イベントを受け取る関数	説明
TPEV_SVCFAIL	tpsend()	接続の開始側が受信します。下位プログラムが接続の制御を持たない場合に tpreturn() を呼び出し、TPFAIL または TPEXIT、およびデータなしで tpreturn() が呼び出されたことを示します。
	tprecv()	接続の開始側が受信します。下位サービスの処理が正常に終了しなかったこと (つまり、tpreturn() が TPFAIL または TPEXIT で呼び出されたこと) を示します。
TPEV_SVCSUCC	tprecv()	接続の開始側が受信します。下位サービスの処理が正常に終了したこと (つまり、tpreturn() が TPSUCCESS で呼び出されたこと) を示します。

8 イベント・ベースのクライアントおよびサーバのコーディング

ここでは、次の内容について説明します。

- イベントの概要
- 任意通知型メッセージ・ハンドラの定義
- 任意通知型メッセージの送信
- 任意通知型メッセージの確認
- イベントのサブスクライブ
- イベントに対するサブスクリプションの削除
- イベントのポスト
- イベントのサブスクリプションの例

イベントの概要

イベント・ベースの通信では、特定の状況 (イベント) が発生すると、BEA Tuxedo システムのプロセスに通知されます。

BEA Tuxedo システムには、次の 2 種類のイベント・ベースの通信があります。

- 任意通知型イベント
- ブローカ・イベント

任意通知型イベント

任意通知型イベントは、メッセージを待たないクライアント・プログラム、またはメッセージを要求しないクライアント・プログラムとの通信に使用されるメッセージです。

ブローカ・イベント

ブローカ・イベントを使用すると、メッセージの受信と配信を行う「無名」ブローカを介して、クライアントとサーバが透過的に通信できるようになります。このブローカを使用した通信は、BEA Tuxedo システムの基本要素であるクライアント / サーバ通信パラダイムの 1 つです。

イベント・ブローカは、イベント・ポスト・メッセージを受信してフィルタ処理し、それらのメッセージをサブスクライバに配布する BEA Tuxedo のサブシステムです。ポスト元とは、特定のイベントが発生したときにそれをイベント・ブローカに報告 (ポスト) する BEA Tuxedo システムのプロセスです。サブスクライバとは、特定のイベントがポストされたときに常に通知する必要がある BEA Tuxedo システムのプロセスです。

BEA Tuxedo システムでは、サービスの要求側と提供側の数の比率が一定である必要はなく、任意の数のポスト元が任意の数のサブスクライバに対してメッセージ・バッファをポストできます。ポスト元は単にイベントをポストするだけで、情報を受信するプロセスや、情報の処理方法については関知しません。サブスクライバには指定されたイベントが通知されますが、その情報のポスト元は通知されません。このように、イベント・ブローカでは位置透過性が実現されます。

通常、イベント・ブローカ・アプリケーションは、例外イベントを処理します。アプリケーションの設計者は、アプリケーション内でどのイベントを例外イベントとして定義して監視する必要があるのかを決定しなければなりません。たとえば、銀行業務アプリケーションでは、高額な引き出しがあったときにイベントがポストされるように設定し、すべての引き出しに対してイベントがポストされる必要はありません。また、すべてのユーザがそのイベントをサブスクライブする必要はありません。支店長だけに通知すれば十分です。

通知処理

イベントがポストされると、イベント・ブローカはイベントをサブスクライブしているクライアントまたはサーバに対して、1 つ以上の通知処理を行います。次の表は、イベント・ブローカが行う通知処理の種類を示しています。

表 8-1 イベント・ブローカの通知処理

通知処理	説明
任意通知型通知メッセージ	クライアントは、`tpnotify()` で送信された場合と同じように、クライアントの任意通知型メッセージ処理ルーチンでイベント通知メッセージを受信します。

通知処理	説明
サービス呼び出し	サーバは、tpacall() によって送信された場合と同じように、サービス・ルーチンに対する入力としてイベント通知メッセージを受け取ります。
信頼性の高いキュー	イベント通知メッセージは、tpenqueue(3c) 関数を使用して、BEA Tuxedo システムの信頼性の高いキューに格納されます。イベント通知バッファは、バッファ内容が要求されるまで格納されます。BEA Tuxedo システムのクライアントまたはサーバ・プロセスで tpdequeue(3c) 関数を呼び出して、この通知バッファを取り出すことができます。または、TMQFORWARD(5) を設定して、通知バッファを取り出す BEA Tuxedo システムのサービス・ルーチンを自動的にディスパッチすることもできます。/Q の詳細については、『BEA Tuxedo/Q コンポーネント』を参照してください。

アプリケーション管理者は、BEA Tuxedo の管理用 API を使用して、次の通知処理を行う EVENT_MIB(5) エントリを作成できます。

- システム・コマンドの呼び出し
- ディスク上のシステムのログ・ファイルへのメッセージの書き込み

注記 EVENT_MIB(5) エントリを作成できるのは、BEA Tuxedo アプリケーション管理者だけです。

EVENT_MIB(5) の詳細については、『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』を参照してください。

イベント・ブローカ・サーバ

TMUSREVT は BEA Tuxedo システムで提供されるサーバで、ユーザ定義のイベントに対するイベント・ブローカとして動作します。TMUSREVT はイベント・レポート用のメッセージ・バッファを処理し、それらのバッファをフィルタ処理して配信します。イベントのブローカ処理を行うには、BEA Tuxedo アプリケーション管理者がこれらのサーバを 1 つ以上起動する必要があります。

TMSYSEVT は BEA Tuxedo システムで提供されるサーバで、システム定義のイベントに対するイベント・ブローカとして動作します。TMSYSEVT と TMUSREVT は似ています。ただし、別個のサーバが提供されているので、アプリケーション管理者はこの 2 種類のイベント通知に対して異なる処理方法を取り入れることができます。詳細については、『BEA Tuxedo アプリケーションの設定』を参照してください。

システム定義のイベント

BEA Tuxedo システムでは、システムの警告と障害に関連する定義済みの特定のイベントが検出されてポストされます。これらの処理はイベント・ブローカによって行われます。たとえば、システム定義のイベントには、設定の変更、状態の変更、接続の障害、マシンの分断などがあります。イベント・ブローカによって検出されるシステム定義のイベントの全リストについては、『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 EVENTS(5) を参照してください。

システム定義のイベントは、BEA Tuxedo システムで定義されているので、ポストする必要はありません。システム定義のイベント名は、アプリケーション定義のイベント名とは異なり、必ずドット (".") で始まります。アプリケーション定義のイベント名をドットで始めることはできません。

クライアントとサーバは、システム定義のイベントをサブスクライブできます。ただし、システム定義のイベントは、アプリケーション内のすべてのクライアントが使用するのではなく、主にアプリケーション管理者が使用します。

イベント・ブローカをアプリケーションに組み込む場合、イベント・ブローカが多数のサブスクライバに大量の配信を行うためのシステムではないことを考慮してください。発生するすべての動作に対してイベントをポストしないでください。また、すべてのクライアントとサーバがイベントをサブスクライブする必要はありません。イベント・ブローカに負荷がかかると、システムのパフォーマンスに影響し、通知が行われなくなる場合があります。負荷を最小限にするには、『Tuxedo システムのインストール』で説明するように、アプリケーション管理者がオペレーティング・システムの IPC 資源を慎重に調整する必要があります。

イベント・ブローカ・プログラミング・インターフェイス

イベント・ブローカ・プログラミング・インターフェイスは、ワークステーションを含むすべての BEA Tuxedo システムのサーバ・プロセスおよびクライアント・プロセスに対して C 言語および COBOL 言語で使用できます。

プログラマは、次の処理をコーディングします。

1. クライアントまたはサーバは、アプリケーション定義のイベント名にバッファをポストします。
2. ポストされたバッファは、そのイベントをサブスクライブしている任意の数のプロセスに転送されます。

サブスクライバへの通知方法にはいろいろな方法 (「通知処理」を参照) があり、イベントはフィルタ処理されます。通知処理とフィルタ処理は、プログラミング・インターフェイスと BEA Tuxedo システムの管理用 API を使用して設定します。

任意通知型メッセージ・ハンドラの定義

任意通知型メッセージ・ハンドラ関数を定義するには、次の文法を使用して tpsetunsol(3c) 関数を呼び出します。

```
int
tpsetunsol(*myfunc)
```

tpsetunsol() 関数の引数は、次の表で説明する *myfunc* だけです。

表 8-2 tpsetunsol( ) 関数の引数

引数	説明
myfunc	コールバック関数のプロトタイプに準拠する関数へのポインタ。このプロトタイプに準拠するには、関数で次の 3 つのパラメータを使用できることが必要です。 ■ *data*—任意通知型メッセージを含む型付きバッファを指します。 ■ *len*—バッファの長さ。 ■ *flags*—現在使用されていません。

クライアントが任意通知型メッセージを受信すると、そのメッセージと共にコールバック関数がディスパッチされます。できる限りタスクが中断されないようにするには、任意通知型メッセージのハンドラ関数で最小限の処理しか行われないようにコーディングします。このようにすると、ハンドラ関数が待機中のプロセスにすぐに戻ることができます。

任意通知型メッセージの送信

BEA Tuxedo システムでは、要求 / 応答型呼び出しまたは会話型通信の処理を妨げずに、クライアント・プロセスに任意通知型メッセージを送信できます。

任意通知型メッセージは、名前、または以前に処理されたメッセージと共に受信した識別子を使用して、クライアント・プロセスに送信できます。名前による送信には tpbroadcast(3c)、識別子による送信には tpnotify(3c) を使用します。tpbroadcast() で送信されるメッセージの発信元は、サービスまたは別のクライアントです。tpnotify() で送信されるメッセージの発信元は、サービスだけです。

名前によるメッセージのブロードキャスト

tpbroadcast(3c) を使用すると、アプリケーションの登録されたクライアントにメッセージが送信されます。tpbroadcast() は、サービスまたは別のクライアントから呼び出すことができます。登録されたクライアントとは、tpinit() を呼び出したが、まだ tpterm() を呼び出してはいないクライアントのことです。

tpbroadcast() 関数の呼び出しには、次の文法を使用します。

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

次の表は、tpbroadcast() 関数の引数を示しています。

表 8-3 tpbroadcast( ) 関数の引数

引数	説明
lmid	クライアントの論理マシン識別子を指すポインタ。NULL 値をワイルドカードとして使用できるので、複数のクライアントにメッセージを送信できます。
usrname	クライアント・プロセスのユーザ名が存在する場合、そのユーザ名を指すポインタ。NULL 値をワイルドカードとして使用できるので、複数のクライアントにメッセージを送信できます。
cltname	クライアント・プロセスのクライアント名が存在する場合、そのクライアント名を指すポインタ。NULL 値をワイルドカードとして使用できるので、複数のクライアントにメッセージを送信できます。
data	メッセージの内容を指すポインタ。
len	メッセージ・バッファのサイズ。*data* が自己記述型のバッファ・タイプ (FML など) を指す場合、*len* に 0 を設定できます。
flags	フラグのオプション。使用できるフラグについては、『BEA Tuxedo C リファレンス』の tpbroadcast(3c) を参照してください。

次のコード例は、すべてのクライアントを送信先として tpbroadcast() を呼び出す方法を示しています。送信メッセージは、STRING 型バッファ内にあります。

リスト 8-1 tpbroadcast( ) **の使用**

```
char *strbuf;

 if ((strbuf = tpalloc("STRING", NULL, 0)) == NULL) {
          error routine
          }

 (void) strcpy(strbuf, "hello, world");

 if (tpbroadcast(NULL, NULL, NULL, strbuf, 0, TPSIGRSTRT) == -1)
              error routine
```

識別子によるメッセージのブロードキャスト

tpnotify(3c) 関数を使用すると、以前に処理されたメッセージと共に受信した識別子を使用してメッセージがブロードキャストされます。tpnotify() 関数は、サービスからのみ呼び出すことができます。

tpnotify() 関数の呼び出しには、次の文法を使用します。

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

次の表は、tpnotify() 関数の引数を示しています。

表 8-4 tpnotify( ) **関数の引数**

引数	説明
clientid	CLIENTID 構造体を指すポインタ。この構造体は、このサービスへの要求を持つ TPSVCINFO 構造体から取得されたものです。
data	メッセージの内容を指すポインタ。
len	メッセージ・バッファのサイズ。*data* が自己記述型のバッファ・タイプ (FML など) を指す場合、*len* に 0 を設定できます。
flags	フラグのオプション。使用できるフラグについては、『BEA Tuxedo C リファレンス』の tpnotify(3c) を参照してください。

任意通知型メッセージの確認

クライアントが「ディップ・イン」通知モードで実行されている場合に任意通知型メッセージがあるかどうかを確認するには、次の文法を使用して tpchkunsol(3c) 関数を呼び出します。

```
int
tpchkunsol()
```

この関数に引数はありません。

未処理のメッセージがある場合、tpsetunsol() で指定された任意通知型メッセージ処理関数が呼び出されます。処理が終了すると、この関数は処理した任意通知型メッセージの数、またはエラーを示す -1 を返します。

クライアントがシグナル・ベースまたはスレッド・ベースの通知モードで実行されている場合、またはクライアントが任意通知型メッセージを無視している場合にこの関数を呼び出すと、関数は何も処理を行わずにすぐに制御を戻します。

イベントのサブスクライブ

tpsubscribe(3c) 関数を使用すると、BEA Tuxedo システムのクライアントまたはサーバがイベントをサブスクライブできるようになります。

サブスクライバは、任意通知型通知メッセージ、サービス呼び出し、高い信頼性のキュー、またはアプリケーション管理者が設定する別の通知方法によって、通知を受け取ります。別の通知方法の設定については、『BEA Tuxedo アプリケーションの設定』を参照してください。

tpsubscribe() 関数の呼び出しには、次の文法を使用します。

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

次の表は、tpsubscribe() 関数の引数を示しています。

表 8-5 tpsubscribe( ) **関数の引数**

引数	説明
eventexpr	プロセスがサブスクライブする 1 つ以上のイベントを指すポインタ。正規表現を含み、NULL 文字で終了する最大 255 文字の文字列を指定します。正規表現は、`tpsubscribe(3c)` で指定された形式です (『BEA Tuxedo C リファレンス』を参照)。たとえば、次のように eventexpr を設定します。 ■ `"\\..*"`—呼び出し元は、—すべてのシステム定義のイベントをサブスクライブします。 ■ `"\\.SysServer.*"`—呼び出し元は、サーバに関連するすべてのシステム定義のイベントをサブスクライブします。 ■ `"[A-Z].*"`—呼び出し元は、A ～ Z の大文字で始まるすべてのユーザ定義のイベントをサブスクライブします。 ■ `".*(ERR|err).*"`—呼び出し元は、`account_error` または `ERROR_STATE` など、イベント名に `err` または `ERR` を含むすべてのユーザ定義のイベントをサブスクライブします。
filter	ブール型のフィルタ規則を含む文字列を指すポインタ。イベント・ブローカがイベントをポストする前に、この規則を評価する必要があります。ポストするイベントを受け取ると、イベント・ブローカはそのイベントのデータにフィルタ規則 (存在する場合) を適用します。データが正しく評価された場合、イベント・ブローカは指定された通知方法を呼び出します。正しく評価されなかった場合、イベント・ブローカは指定された通知方法を無視します。呼び出し元は、異なるフィルタ・ルールを利用して同じイベントを何度でもサブスクライブすることができます。 イベント・フィルタ機能を使用すると、サブスクライバは通知されるイベントを限定できます。たとえば、100 万円を超える額の引き出しがあった場合に、イベントがポストされるとします。その場合、サブスクライバが 100 万円より高い額 (たとえば 500 万円) の通知だけを必要とすることがあります。または、特定の顧客による高額の引き出しの通知だけを必要とする場合があります。 フィルタ・ルールは、それが適用される型付きバッファに固有なものです。フィルタ規則の詳細については、『BEA Tuxedo C リファレンス』の `tpsubscribe(3c)` を参照してください。

引数	説明
ctl	サブスクライバへのイベント通知の方法を制御するフラグを指すポインタ。次の値を指定できます。 ■ NULL—任意通知型メッセージを送信します。詳細については、第 8 章の 10 ページ「任意通知型メッセージを使用した通知」を参照してください。 ■ 有効な TPEVCTL 構造体へのポインター—TPEVCTL 構造体に基づく情報を送信します。詳細については、第 8 章の 11 ページ「サービス呼び出しまたは信頼できるキューを使用した通知」を参照してください。
flags	フラグのオプション。使用できるフラグについては、『BEA Tuxedo C リファレンス』の tpsubscribe(3c) を参照してください。

システム定義のイベントとアプリケーション定義のイベントは、tpsubscribe() 関数を使用してサブスクライブできます。

サブスクリプション、および MIB を更新するために BEA Tuxedo システムのサーバ・プロセスで実行されるサービス・ルーチンは、信頼されたコードと見なされます。

任意通知型メッセージを使用した通知

サブスクライバが BEA Tuxedo システムのクライアント・プロセスであり、*ctl* に NULL が設定されている場合、クライアントがサブスクライブしているイベントがポストされると、イベント・ブローカは次のようにサブスクライバに任意通知型メッセージを送ります。*eventexpr* に対して正常に評価されたイベント名がポストされると、イベント・ブローカは対応付けられたフィルタ規則でポストされたデータを確認します。データがフィルタ規則で正しく評価された場合 (またはイベントにフィルタ規則がない場合)、サブスクライバはイベントと共にポストされたデータと任意通知型メッセージを受信します。

任意通知型メッセージを受け取るには、クライアントは tpsetunsol() 関数を使用して、任意通知型メッセージ処理ルーチンを登録しておく必要があります。

クライアントが任意通知型メッセージによってイベント通知を受け取った場合、終了する前にイベント・ブローカのアクティブなサブスクリプションのリストからそのサブスクリプションを削除する必要があります。サブスクリプションの削除には tpunsubscribe() 関数を使用します。

サービス呼び出しまたは信頼できるキューを使用した通知

サービス呼び出しを使用したイベント通知では、アプリケーションの特定の条件に対して自動的に応答するようにプログラミングできます。高い信頼性のキューを使用したイベント通知では、イベント・データが損失しないことが保証されます。また、サブスクライバがいつでもイベント・データを取り出せるという柔軟性があります。

サブスクライバ (クライアント・プロセスまたはサーバ・プロセス) がイベント通知をサービス・ルーチンまたは安定記憶領域のキューに送信する場合、tpsubscribe() の *ctl* パラメータが有効な TPEVCTL 構造体を指していることが必要です。

TPEVCTL 構造体には、次の情報が格納されています。

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

次の表は、TPEVCTL 型バッファのデータ構造を示しています。

表 8-6 TPEVCTL 型バッファの形式

フィールド	説明
flags	フラグのオプション。フラグの詳細については、『BEA Tuxedo C リファレンス』の tpsubscribe(3c) を参照してください。
name1	32 文字以下の文字列。
name2	32 文字以下の文字列。
qctl	TPQCTL 構造体。詳細については、『BEA Tuxedo C リファレンス』の tpsubscribe(3c) を参照してください。

イベントに対するサブスクリプションの削除

tpunsubscribe(3c) 関数を使用すると、BEA Tuxedo システムのクライアントまたはサーバがイベントに対するサブスクリプションを削除できます。

tpunsubscribe() 関数の呼び出しには、次の文法を使用します。

```
int
tpunsubscribe (long subscription, long flags)
```

次の表は、tpunsubscribe() 関数の引数を示しています。

表 8-7 tpunsubscribe( ) 関数の引数

引数	説明
subscription	tpsubscribe() への呼び出しで返されるサブスクリプション・ハンドル。
flags	フラグのオプション。使用できるフラグについては、『BEA Tuxedo C リファレンス』の tpunsubscribe(3c) を参照してください。

イベントのポスト

tppost(3c) 関数を使用すると、BEA Tuxedo のクライアントまたはサーバがイベントをポストできます。

tppost() 関数の呼び出しには、次の文法を使用します。

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

次の表は、tppost() 関数の引数を示しています。

表 8-8 tppost( ) 関数の引数

引数	説明
eventname	31 文字以下の文字列と最後に NULL 文字を含むイベント名を指すポインタ。名前をドット (".") で開始することはできません。ドットは、BEA Tuxedo のシステム定義のイベントの先頭文字として予約されています。イベントのサブスクライブでは、イベント名にワイルド・カードを使用して、1 回の関数呼び出しで複数のイベントをサブスクライブすることができます。そのため関連するイベント名のカテゴリに同じ接頭語を使用すると便利です。
data	tpalloc() 関数で既に割り当てられているバッファを指すポインタ。
len	イベントと共にポストするデータ・バッファのサイズ。*data* が長さを指定する必要がないバッファ・タイプ (FML フィールド化バッファなど) を指している場合、または NULL が設定されている場合、len は無視され、データなしでイベントがポストされます。
flags	フラグのオプション。使用できるフラグについては、『BEA Tuxedo C リファレンス』の tppost(3c) を参照してください。

次のコード例は、BEA Tuxedo システムの銀行業務のサンプル・アプリケーション bankapp から引用したイベント・ポストを示しています。この例は、WITHDRAWAL サービスの一部です。WITHDRAWAL サービスは、10,000 ドルを超える引き出しかどうかを確認し、また BANK_TLR_WITHDRAWAL イベントをポストします。

リスト 8-2 tppost() を使用したイベントのポスト

```
.
.
.
/* 関連するイベント・ロジック */
static float evt_thresh = 10000.00 ; /* デフォルトのイベントしきい値 */
static char  emsg[200] ;  /* イベント・ポスト・ロジックで使用 */
.
.
.
/* BANK_TLR_WITHDRAWAL イベントをポストするかどうかの判定 */
if (amt < evt_thresh) {
     /* ポストするイベントはありません。*/
     tpreturn(TPSUCCESS, 0,transb->data, 0L, 0);
```

```
}
/* イベントをポストする準備 */
if ((Fchg (transf, EVENT_NAME, 0, "BANK_TLR_WITHDRAWAL", (FLDLEN)0) == -1) ||
(Fchg (transf, EVENT_TIME, 0, gettime(), (FLDLEN)0) == -1) ||
(Fchg (transf, AMOUNT, 0, (char *)&amt, (FLDLEN)0) == -1)) {
     (void)sprintf (emsg, "Fchg failed for event fields:%s",
     Fstrerror(Ferror)) ;
}
/* イベントのポスト */
else if (tppost ("BANK_TLR_WITHDRAWAL", /* event name */
(char *)transf, /* data */
0L,   /* len */
TPNOTRAN | TPSIGRSTRT) == -1) {
/* イベント・ブローカを使用できない場合、エラーを無視 */
     if (tperrno != TPENOENT)
     (void)sprintf (emsg, "tppost failed:%s", tpstrerror (tperrno));
}
```

このコード例では、アプリケーションで目立った状況が発生したことを示すために、イベントをイベント・ブローカにポストしているだけです。特定のイベントに関心を持つクライアント (必要に応じて処理を行うクライアント) のイベントへのサブスクリプションは、別にコーディングします。

イベントのサブスクリプションの例

次のコード例は、bankapp アプリケーション・サーバの一部であり、このサーバは BANK_TLR_.* イベントをサブスクライブしています。この例には、前述の例の BANK_TLR_WITHDRAWAL イベントと、BANK_TLR_ で始まるそのほかのイベント名が含まれています。合致するイベントがポストされると、WATCHDOG サービスへの呼び出しを使用して、アプリケーションからサブスクライバにイベントが通知されます。

リスト 8-3 tpsubscribe() を使用したイベントのサブスクライブ

```
.
.
.
/* イベント・サブスクリプション・ハンドル */
static long sub_ev_largeamt = 0L ;
.
.
```

```
.
/* 'w' (WATCHDOG のしきい値 ) にデフォルト値を設定 */
(void)strcpy (amt_expr, "AMOUNT > 1000000.00") ;
.
.
.
/*
 * 高額な引き出しが行われた場合に
 * 生成されるイベントをサブスクライブ
 */
evctl.flags = TPEVSERVICE ;
(void)strcpy (evctl.name1, "WATCHDOG") ;
/* サブスクライブ */
sub_ev_largeamt = tpsubscribe ("BANK_TLR_.*",amt_expr,&evctl,TPSIGRSTRT) ;
if (sub_ev_largeamt == -1L) {
     (void)userlog ("ERROR:tpsubscribe for event BANK_TLR_.* failed:%s",
     tpstrerror(tperrno)) ;
     return -1 ;
}
.
.
.
{
/* イベントへのサブスクリプションを削除 */
if (tpunsubscribe (sub_ev_largeamt, TPSIGRSTRT) == -1)
     (void)userlog ("ERROR:tpunsubscribe to event BANK_TLR_.* failed:%s",
     tpstrerror(tperrno)) ;
     return;
}
/*
* BANK_TLR_.* イベントがポストされたときに呼び出されるサービス
*/
void
#if defined(__STDC__) || defined(__cplusplus)
WATCHDOG(TPSVCINFO *transb)
#else
WATCHDOG(transb)
TPSVCINFO *transb;
#endif
{
FBFR *transf;          /* 復号化されたメッセージのフィールド化バッファ */
/* TPSVCINFO データ・バッファを指すポインタを設定します。*/
transf = (FBFR *)transb->data;
/* ログ・エントリを標準出力に出力 */
(void)fprintf (stdout, "%20s|%28s|%8ld|%10.2f\n",
Fvals (transf, EVENT_NAME, 0),
Fvals (transf, EVENT_TIME, 0),
Fvall (transf, ACCOUNT_ID, 0),
```

```
*( (float *)CFfind (transf, AMOUNT, 0, NULL, FLD_FLOAT)) );
/* イベントのサブスクライバのサービス・ルーチンから返されるデータはありません。*/
tpreturn(TPSUCCESS, 0,NULL, 0L, 0);
}
```

9 グローバル・トランザクションのコーディング

ここでは、次の内容について説明します。

- グローバル・トランザクションとは
- トランザクションの開始
- トランザクションの中断と再開
- トランザクションの終了
- グローバル・トランザクションの暗黙的な定義
- XA 準拠のサーバ・グループに対するグローバル・トランザクションの定義
- トランザクションが開始されたことの確認

グローバル・トランザクションとは

グローバル・トランザクションとは、複数のリソース・マネージャを使用し、複数のサーバ上で行われる複数の操作を 1 つの論理単位として処理できるようにするメカニズムです。

プロセスがトランザクション・モードになると、サーバに要求されたサービスが現在のトランザクションに代わって処理されます。呼び出されてトランザクションに参加したサービスは、「トランザクションのパーティシパント」と呼ばれます。パーティシパントから返される値によって、トランザクションの結果が変わる場合があります。

グローバル・トランザクションは複数のローカル・トランザクションから構成され、各トランザクションは同じリソース・マネージャにアクセスします。リソース・マネージャは、同時実行制御とデータ更新の原子性を実現します。ローカル・トランザクションでは、アクセスが正常に終了するか、または全体が失敗します。つまり、一部だけが成功することはありません。

1 つのトランザクションに参加可能なサーバ・グループは最大 16 個です。

BEA Tuxedo システムでは、グローバル・トランザクションが参加しているリソース・マネージャと共に管理され、原子性、一貫性、独立性、および持続性という特徴を持つ特定シーケンスの操作として処理されます。つまり、グローバル・トランザクションは、以下のような特徴を持つ論理的な作業単位と言えます。

- すべての部分が成功するか、または何も効果が発生しません。
- 操作が実行されて、リソースがある一貫した状態から別の状態に正しく移行します。
- ほかのトランザクションから中間の結果にアクセスすることはできません。ただし、トランザクションに含まれるプロセスには、別のプロセスに対応付けられたデータにアクセスできるものもあります。
- シーケンスが完了すると、その結果はどのような失敗による影響も受けません。

BEA Tuxedo システムでは、個々のグローバル・トランザクションの状態がトラッキングされ、そのトランザクションをコミットするかロールバックするかが決定されます。

注記 トランザクションで、*flags* 引数に明示的に TPNOTRAN を設定して tpcall()、tpacall() または tpconnect() を呼び出した場合、呼び出されたサービスによって実行される操作はトランザクションに含まれません。その場合、呼び出し元プロセスは、呼び出されたサービスを現在のトランザクションのパーティシパントとは見なしません。その結果、呼び出されたプロセスによって実行されるサービスは、現在のトランザクションの結果の影響を受けません。XA 準拠のサーバ・グループのサービスに対する呼び出しに TPNOTRAN が設定されている場合、その呼び出しはトランザクション・モード外、または別のトランザクションで実行されます。どちらで実行されるかは、サービスの設定方法とコーディング方法によって決まります。詳細については、第 9 章の 16 ページ「グローバル・トランザクションの暗黙的な定義」を参照してください。

トランザクションの開始

グローバル・トランザクションを開始するには、次の文法を使用して tpbegin(3c) 関数を呼び出します。

```
int
tpbegin(unsigned long timeout, long flags)
```

次の表は、tpbegin( ) 関数の引数を示しています。

表 9-1 tpbegin( ) 関数の引数

フィールド	説明
timeout	トランザクションがタイムアウトになるまでの時間 (秒単位)。この引数に 0 を指定すると、システムで可能な最長時間 (秒単位) に設定されます。つまり、*timeout* には、システムで定義された符号なし long 型の最大値が設定されます。 *timeout* に 0 または非現実的な大きな値を指定すると、システムによるエラー検出と報告が遅れる原因となります。*timeout* パラメータを使用すると、サービス要求に対する応答が妥当な時間内に確実に返されるようになります。また、ネットワーク障害などの問題が発生した場合に、コミットされる前にトランザクションを終了できます。 応答を人が待っている場合、このパラメータには小さな値 (可能な場合は 30 秒未満) を設定します。 生産システムの場合、*timeout* に大きな値を設定して、システムの負荷やデータベースの競合に起因する遅延に対応できるようにします。予測される平均応答時間を 2、3 倍した時間が最適です。 **注記** *timeout* 引数に設定する値は、BEA Tuxedo のアプリケーション管理者がコンフィギュレーション・ファイルに設定した SCANUNIT パラメータの値と一致していなければなりません。SCANUNIT パラメータには、タイムアウトになったトランザクションとサービス要求でブロックされた呼び出しがないかどうかを確認、つまりスキャンする頻度を指定します。このパラメータの値は、定期的なスキャンの間隔 (走査を行う間隔) を表します。 *timeout* パラメータには、走査を行う間隔より大きな値を設定します。*timeout* パラメータに設定された値が走査を行う間隔より小さいと、トランザクションがタイムアウトになる時間と、そのタイムアウトが検出される時間にずれが生じます。SCANUNIT のデフォルト値は 10 秒です。*timeout* パラメータの設定値についてはアプリケーション管理者と検討し、*timeout* に設定した値がシステム・パラメータの値と矛盾しないようにします。
flags	現在は定義されていません。0 を設定します。

tpbegin() は、どのプロセスからも呼び出すことができます。ただし、既にトランザクション・モードになっているプロセス、または未処理の応答を待っているプロセスからは呼び出すことができません。トランザクション・モードで tpbegin() が呼び出されると、プロトコル・エラーになって呼び出しが失敗し、tperrno(5) に TPEPROTO が設定されます。プロセスがトランザクション・モードの場合でも、この失敗はトランザクションには影響しません。

次のコード例は、グローバル・トランザクションの定義方法を簡単に示しています。

リスト 9-1 グローバル・トランザクションの定義 - 簡単な例

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

次のコード例は、トランザクションの定義方法をより詳細に示しています。このコードは、BEA Tuxedo システムで提供される銀行業務のサンプル・アプリケーション bankapp の audit.c クライアント・プログラムから引用したものです。

リスト 9-2 グローバル・トランザクションの定義 - 詳細な例

```
#include <stdio.h>          /* UNIX */
#include <string.h>         /* UNIX */
#include <atmi.h>           /* BEA Tuxedo システム */
#include <Uunix.h>          /* BEA Tuxedo システム */
#include <userlog.h>        /* BEA Tuxedo システム */
#include "bank.h"           /* 銀行業務アプリケーションのマクロ定義 */
#include "aud.h"            /* 銀行業務アプリケーションの VIEW 定義 */

#define INVI 0              /* 残高照会 */
#define ACCT 1              /* 残高照会 */
#define TELL 2              /* 窓口の残高照会 */

static int sum_bal _((char *, char *));
static long sitelist[NSITE] = SITEREP;    /* 監査用マシンのリスト */
static char pgmname[STATLEN];         /* プログラム名 = argv[0] */
static char result_str[STATLEN]; /* 照会の結果を入れる配列 */

main(argc, argv)
```

```
int argc;
char *argv[];
{
        int aud_type=INVI;          /* 監査の種類―指定されていない場合は無効 */
        int clarg;                  /* optind からのコマンド行引数のインデックス */
        int c;                      /* オプション文字 */
        int cflgs=0;                /* コミットのフラグ。現在使用されていません。*/
        int aflgs=0;                /* アボートのフラグ。現在使用されていません。*/
        int nbl=0;                  /* 支店リストのエントリ数 */
        char svc_name[NAMELEN];     /* サービス名 */
        char hdr_type[NAMELEN];     /* 出力の表題 */
        int retc;                   /* sum_bal() の戻り値 */
        struct aud *audv;           /* audit バッファ構造体へのポインタ */
        int audrl=0;                /* audit の戻り値の長さ */
        long q_branchid;            /* 照会する支店番号 */

. . .       /* コマンド行オプションの取得と変数の設定 */

/* アプリケーションへの参加 */

if (tpinit((TPINIT *) NULL) == -1) {
       (void)userlog("%s:failed to join application\n", pgmname);
       exit(1);
}

/* グローバル・トランザクションの開始 */

if (tpbegin(30, 0) == -1) {
       (void)userlog("%s:failed to begin transaction\n", pgmname);
       (void)tpterm();
       exit(1);
}

if (nbl == 0) {    /* 支店番号が指定されていないので、総勘定を計算 */
  retc = sum_bal(svc_name, hdr_type);    /* sum_bal ルーチンは省略 */

} else {

        /* バッファを作成し、データ・ポインタを設定します。*/

        if ((audv = (struct aud *)tpalloc("VIEW", "aud", sizeof(struct aud)))
             == (struct aud *)NULL) {
             (void)userlog("audit:unable to allocate space for VIEW\n");
             exit(1);
        }

        /* aud 構造体を用意します。*/
```

```
        audv->b_id = q_branchid;
        audv->balance = 0.0;
        audv->ermsg[0] = '\0';

        /* tpcall の実行 */

        if (tpcall(svc_name,(char *)audv,sizeof(struct aud),
           (char **)audv,(long *)audrl,0) == -1){
                    (void)fprintf (stderr,"%s service failed\n%s:%s\n",
                    svc_name, svc_name, audv->ermsg);
                    retc = -1;

        }else {

                    (void)sprintf(result_str,"Branch %ld %s balance is $%.2f\n",
                    audv->b_id, hdr_type, audv->balance);
        }
tpfree((char *)audv);
}

/* グローバル・トランザクションのコミット */

if (retc < 0)        /* sum_bal の実行が失敗したので処理をアボート */
       (void) tpabort(aflgs);
else {
         if (tpcommit(cflgs) == -1) {
                (void)userlog("%s:failed to commit transaction\n", pgmname);
                (void)tpterm();
                exit(1);
       }
        /* トランザクションのコミットが成功した場合のみ、結果を出力 */
        (void)printf("%s",result_str);
}

/* アプリケーションからの分離 */

if (tpterm() == -1) {
       (void)userlog("%s:failed to leave application\n", pgmname);
       exit(1);
}
```

トランザクションがタイムアウトになった場合、tpcommit() を呼び出すとトランザクションがアボートします。その結果、tpcommit() が失敗し、tperrno(5) に TPEABORT が設定されます。

次のコード例は、トランザクションのタイムアウトを確認する方法を示しています。*timeout* の値が 30 秒に設定されていることに注目してください。

リスト 9-3 **トランザクションのタイムアウトの確認**

```
if (tpbegin(30, 0) == -1) {
  (void)userlog("%s:failed to begin transaction\n", argv[0]);
  tpterm();
  exit(1);
}
. . .
communication calls
. . .
if (tperrno == TPETIME){
    if (tpabort(0) == -1) {
      check for errors;
}
else if (tpcommit(0) == -1){
      check for errors;
}
. . .
```

注記 トランザクション・モードのプロセスで、*flags* 引数に TPNOTRAN を設定して通信呼び出しを行うと、呼び出されたサービスは現在のトランザクションに参加できません。サービス要求の成功や失敗は、トランザクションの結果に影響しません。トランザクションは、サービスから応答が返されるのを待つ間にタイムアウトになる場合もあります。これは、そのサービスがトランザションに参加しているかどうかには関係ありません。TPNOTRAN フラグの影響については、第 11 章の 1 ページ「エラーの管理」を参照してください。

トランザクションの中断と再開

場合によっては、実行中のトランザクションから一時的にプロセスを削除し、tpbegin() または tpresume() を呼び出して、そのプロセスで別のトランザクションを開始した方がよい場合もあります。たとえば、サーバがデータベースの中央イベント・ログに要求を記録する場合、トランザクションがアボートしてもログ処理をロールバックしたくない場合などです。

BEA Tuxedo システムでは、このような場合にクライアントまたはサーバでトランザクションを中断して再開する tpsuspend(3c) と tpresume(3c) という 2 つの関数が提供されています。この 2 つの関数を使用すると、次の処理を行うことができます。

1. tpsuspend() を呼び出して、現在のトランザクションを一時的に中断します。
2. 別のトランザクションを開始します。前述の例では、サーバのイベント・ログへのエントリの書き込みが開始されます。
3. 手順 2 で開始されたトランザクションをコミットします。
4. tpresume() を呼び出して、元のトランザクションを再開します。

トランザクションの中断

tpsuspend(3c) 関数を使用すると、現在のトランザクションを中断できます。tpsuspend() 関数の呼び出しには、次の文法を使用します。

```
int
tpsuspend(TPTRANID *t_id,long flags)
```

次の表は、tpsuspend() 関数の引数を示しています。

表 9-2 tpsuspend( ) 関数の引数

フィールド	説明
*t_id	トランザクション識別子を指すポインタ。
flags	現在使用されていません。将来使用するために予約されたフィールド。

未処理の非同期イベントを持つトランザクションを中断することはできません。トランザクションが中断すると、トランザクションがコミットまたはアボートされるまで、あるいはタイムアウトになるまで、中断前に行った変更はすべて保留状態のまま維持されます。

トランザクションの再開

現在のトランザクションを再開するには、次の文法を使用して tpresume(3c) 関数を呼び出します。

```
int
tpresume(TPTRANID *t_id,long flags)
```

次の表は、tpresume() 関数の引数を示しています。

表 9-3 tpresume( ) 関数の引数

フィールド	説明
**t_id*	トランザクション識別子を指すポインタ。
flags	現在使用されていません。将来使用するために予約されたフィールド。

トランザクションを中断したプロセス以外のプロセスからトランザクションを再開できますが、制限があります。この制限については、『BEA Tuxedo C リファレンス』の tpsuspend(3c) と tpresume(3c) を参照してください。

例 : トランザクションの中断と再開

次のコード例は、あるトランザクションを中断し、別のトランザクションを開始してコミットし、最初のトランザシクションを再開する方法を示しています。コードを簡単にするために、エラー・チェックは省略してあります。

リスト 9-4 トランザクションの中断と再開

```
DEBIT(SVCINFO *s)
{
      TPTRANID t;
      tpsuspend(&t,TPNOFLAGS);  /* 実行中のトランザクションを中断 */

      tpbegin(30,TPNOFLAGS); /* 別のトランザクションを開始 */
      別のトランザクションで処理を実行
      tpcommit(TPNOFLAGS);   /* トランザクションをコミット */

      tpresume(&t,TPNOFLAGS);   /* 最初のトランザクションを再開 */

      .
      .
      .
      tpreturn(. . . );
}
```

トランザクションの終了

グローバル・トランザクションを終了するには、tpcommit(3c) を呼び出して現在のトランザクションをコミットするか、または tpabort(3c) を呼び出して処理をアボートして、すべての操作をロールバックします。

注記 tpcall()、tpacall()、または tpconnect() を呼び出すときに *flags* 引数に明示的に TPNOTRAN が設定されている場合、呼び出されたサービスによって実行される操作は、トランザクションに含まれません。つまり、このようなサービスによって実行される操作は、tpabort() 関数を呼び出したときにロールバックされません。

現在のトランザクションのコミット

tpcommit(3c) 関数は、現在のトランザクションをコミットします。tpcommit() 関数から正常に制御が戻ると、現在のトランザクションの結果としてリソースに加えられた変更は永続的なものとなります。

tpcommit() 関数の呼び出しには、次の文法を使用します。

```
int
tpcommit(long flags)
```

flags 引数は現在使用されていませんが、今後のリリースとの互換性を保つためにゼロに設定してください。

トランザクションをコミットするための条件

tpcommit() を正常に実行するには、次の 2 つの条件を満たしていることが必要です。

- 呼び出し元プロセスは、tpbegin() を呼び出してトランザクションを開始したプロセスと同じでなければなりません。
- TPNOTRAN フラグを設定しないで呼び出した場合、呼び出し元プロセスに未処理のトランザクション応答が存在することはできません。
- トランザクションの状態が「ロールバックのみ」ではなく、またタイムアウトになっていないことが必要です。

最初の条件を満たしていない場合、呼び出しは失敗し、プロトコル・エラーを示す TPEPROTO が tperrno(5) に設定されます。2 番目または 3 番目の条件を満たしていない場合、呼び出しは失敗し、トランザクションがロールバックされたことを示す TPEABORT が tperrno() に設定されます。トランザクションに未処理の応答があるときに tpcommit() がイニシエータによって呼び出されると、トランザクションはアボートされ、トランザクションに関連する応答記述子が無効になります。パーティシパントが tpcommit() または tpabort() を呼び出しても、トランザクションには影響しません。

サービス呼び出しで TPFAIL が返されるか、またはサービス・エラーが発生すると、トランザクションは「ロールバックのみ」の状態になります。「ロールバックのみ」のトランザクションに対して tpcommit() が呼び出されると、この関数はトランザクションを取り消し、-1 を返して tperrno(5) に TPEABORT を設定します。既にタイムアウトになっているトランザクションに対して tpcommit() を呼び出した場合も同じ結果になり、tpcommit() は -1 を返して tperrno() に TPEABORT が設定されます。トランザクション・エラーの詳細については、第 11 章の 1 ページ「エラーの管理」を参照してください。

2 フェーズ・コミット・プロトコル

tpcommit() 関数が呼び出されると、2 フェーズ・コミット・プロトコルによる通信が開始されます。このプロトコルは、その名前が示すように、次の 2 段階の処理に分かれています。

1. 参加する各リソース・マネージャがコミットの準備ができたことを示します。
2. トランザクションのイニシエータが、参加する各リソース・マネージャにコミット許可を与えます。

トランザクションのイニシエータが tpcommit() 関数を呼び出すと、コミット・シーケンスが開始されます。指定されたコーディネータ・グループ内の BEA Tuxedo TMS サーバ・プロセスは、コミット・プロトコルの最初のフェーズを実行する各パーティシパント・グループの TMS と通信を行います。次に、各グループの TMS は、そのグループのリソース・マネージャ (RM) に、トランザクション・マネージャと RM 間の通信用に定義されている XA プロトコルを使用してコミットするように指示します。RM は、安定記憶域にコミット・シーケンスの前後のトランザクションの状態を書き込み、TMS に成功か失敗かを通知します。その後、TMS はトランザクション・コーディネータの TMS に応答を渡します。

トランザクション・コーディネータの TMS は、すべてのグループから成功の通知を受け取ると、トランザクションのコミット中であることをログに記録し、第 2 フェーズのコミット通知をすべてのパーティシパント・グループに送信します。その後、各グループの RM はトランザクションの更新を完了します。

トランザクション・コーディネータの TMS が、グループから第 1 フェーズのコミットの失敗の通知を受けた場合、またはグループからの応答の受信に失敗した場合、各 RM にロールバック通知を送信し、RM はすべてのトランザクション更新を以前の状態に戻します。これにより、tpcommit() は失敗し、tperrno(5) に TPEABORT が設定されます。

コミットの成功条件の選択

1 つのトランザクションに複数のグループが関係している場合、tpcommit() が正常に制御を戻すための条件として、次のいずれかを指定できます。

- すべてのパーティシパントからコミットの準備が完了したことが通知された場合。つまり、すべてのパーティシパントが 2 フェーズ・コミットの第 1 フェーズが完了したことを報告し、トランザクション・コーディネータの TMS が安定記憶域にコミットの決定を書き込んだ場合です。
- すべてのパーティシパントで 2 フェーズ・コミットの第 2 フェーズが完了した場合。

この 2 つの条件のいずれかを指定するには、コンフィギュレーション・ファイルの RESOURCES の CMTRET パラメータに、次のいずれかの値を設定します。

- LOGGED—第 1 フェーズの完了が必須であることを示します。
- COMPLETE—第 2 フェーズの完了が必須であることを示します。

デフォルトでは、CMTRET は COMPLETE に設定されます。

後でコンフィグレーション・ファイルの設定値を変更する場合は、*flags* 引数に TP_CMT_LOGGED または TP_CMT_COMPLETE を設定して、tpscmt() を呼び出します。

コミット条件での妥協点

ほとんどの場合、グローバル・トランザクションのすべてのパーティシパントが第 1 フェーズの正常終了を記録した場合、第 2 フェーズも正常終了します。CMTRET に LOGGED を設定すると、tpcommit() の呼び出しから制御が多少早く戻るようになります。ただし、パーティシパントが、コミットの決定と矛盾する方法で、トランザクションの担当部分をヒューリスティックに終了する危険性があります。

このようなリスクを負うべきかどうかの選択は、アプリケーションの性質に左右されます。たとえば、財務アプリケーションなど正確さが要求されるアプリケーションでは、すべてのパーティシパントが 2 フェーズ・コミットを完了するまでは、制御を戻さないようにします。時間的な条件を重視するアプリケーションでは、正確さを犠牲にしても実行速度を上げます。

現在のトランザクションのアボート

tpabort(3c) 関数を使用すると、異常な状態を通知して、明示的にトランザクションをアボートできます。この関数は、トランザクションの応答に未処理のものがあると、その呼び出し記述子を無効にします。その場合、トランザクションで行われた変更はリソースには適用されません。tpabort() 関数の呼び出しには、次の文法を使用します。

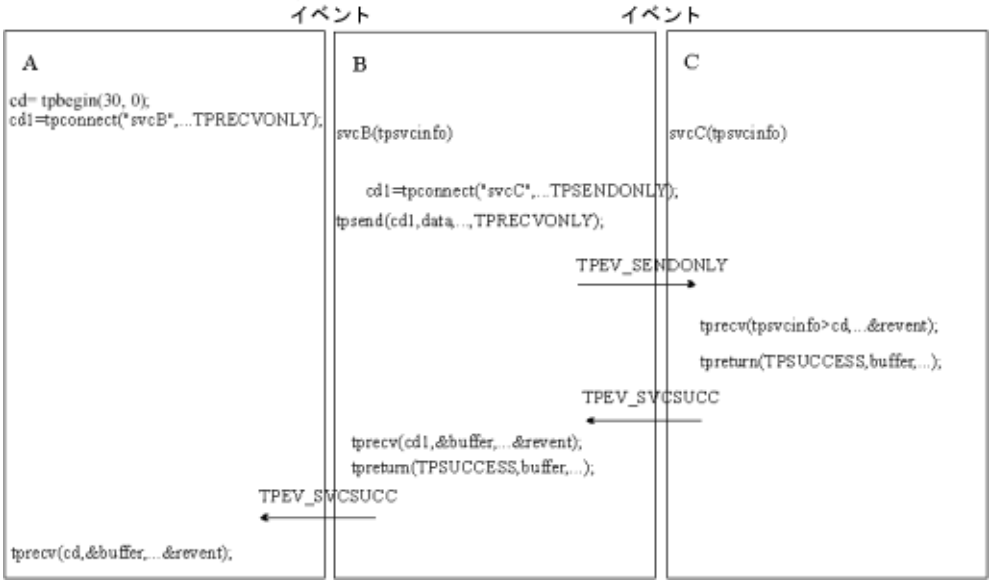
```
int
tpabort(long flags)
```

flags 引数は現在使用されていませんが、今後のリリースとの互換性を保つためにゼロに設定してください。

例 : 会話モードによるトランザクションのコミット

次の図は、グローバル・トランザクションを行う階層構造の会話型接続を示しています。

図 9-1 トランザクション・モードにおける接続階層構造



接続階層構造は、次の処理が行われることで構築されます。

1. クライアント (プロセス A) は、`tpbegin()` と `tpconnect()` を呼び出して、トランザクション・モードで接続を開始します。
2. クライアントは、実行する下位サービスを呼び出します。
3. 各下位サービスは、処理が完了すると、処理が成功したか失敗したか (`TPEV_SVCSUCC` または `TPEV_SVCFAIL`) を示す応答を階層構造を通じてトランザクションを開始したプロセスに送信します。この例では、トランザクションを開始したプロセスはクライアント (プロセス A) です。下位サービスは、応答の送信が終了すると、つまり未処理の応答がなくなると、`tpreturn()` を呼び出します。

4. クライアント (プロセス A) は、すべての下位サービスが正常終了したかどうかを確認します。
   - すべての下位サービスが正常終了した場合、クライアントは tpcommit() を呼び出して、それらのサービスが実行した変更をコミットし、トランザクションを終了します。
   - 正常終了していない下位サービスがある場合、tpcommit() は成功しないので、クライアントは tpabort() を呼び出します。

例 : パーティシパントのエラーの確認

次のコード例では、クライアントは REPORT サービスへの同期呼び出し (18 行目) を行います。次に、通信呼び出しで返される可能性があるエラーを調べて (19 ～ 34 行目)、パーティシパントの失敗を確認します。

リスト 9-5　パーティシパントの成功 / 失敗の確認

```
001    #include <stdio.h>
002    #include "atmi.h"
003
004    main()
005    {
006    char *sbuf, *rbuf;
007    long slen, rlen;
008    if (tpinit((TPINIT *) NULL) == -1)
009        error message, exit program;
010    if (tpbegin(30, 0) == -1)
011        error message, tpterm, exit program;
012    if ((sbuf=tpalloc("STRING", NULL, 100)) == NULL)
013        error message, tpabort, tpterm, exit program;
014    if ((rbuf=tpalloc("STRING", NULL, 2000)) == NULL)
015        error message, tpfree sbuf, tpabort, tpterm, exit program;
016    (void)strcpy(sbuf, "REPORT=accrcv DBNAME=accounts");
017    slen=strlen(sbuf);
018    if (tpcall("REPORT", sbuf, slen, &rbuf, &rlen, 0) == -1) {
019        switch(tperrno) {
020        case TPESVCERR:
021            fprintf(stderr,
022                "REPORT service's tpreturn encountered problems\n");
023            break;
024        case TPESVCFAIL:
025            fprintf(stderr,
026                "REPORT service TPFAILED with return code of %d\n", tpurcode);
```

```
027             break;
028         case TPEOTYPE:
029             fprintf(stderr,
030                 "REPORT service's reply is not of any known data type\n");
031             break;
032         default:
033             fprintf(stderr,
034                 "REPORT service failed with error %d\n", tperrno);
035             break;
036         }
037         if (tpabort(0) == -1){
038             check for errors;
039         }
040     }
041     else
042         if (tpcommit(0) == -1)
043             fprintf(stderr, "Transaction failed at commit time\n");
044     tpfree(rbuf);
045     tpfree(sbuf);
046     tpterm();
047     exit(0);
048     }
```

グローバル・トランザクションの暗黙的な定義

アプリケーションでは、次のいずれかの方法でグローバル・トランザクションを開始できます。

- ATMI 関数を明示的に呼び出します。第 9 章の 3 ページ「トランザクションの開始」を参照してください。
- サービス・ルーチン内から暗黙的に開始します。

この節では、2 番目の方法について説明します。

サービス・ルーチンのトランザクションの暗黙的な定義

コンフィギュレーション・ファイルのシステム・パラメータ AUTOTRAN を設定すると、サービス・ルーチンがトランザクション・モードになります。AUTOTRAN に Y を設定すると、別のプロセスから要求を受信したときに、サービス・サブルーチン内でトランザクションが自動的に開始されます。

暗黙的にトランザクションを定義する場合は、以下の規則に従います。

- 呼び出し元プロセスがトランザクション・モードになっていない場合に、システム・パラメータ AUTOTRAN がトランザクションを開始するように設定されていると、プロセスが別のプロセスのサービスを要求したときにトランザクションが開始されます。
- 既にトランザクション・モードになっているプロセスが別のプロセスのサービスを要求した場合、システムは呼び出し元の *flags* 引数が TPNOTRAN に設定されているかどうかをまず確認します。

  flags 引数に TPNOTRAN が設定されていない場合、呼び出されたプロセスは「伝達の規則」によってトランザクション・モードになります。システムによって AUTOTRAN パラメータは確認されません。

  flags 引数に TPNOTRAN が設定されている場合、呼び出されたプロセスによって実行されるサービスは、現在のトランザクションに含まれません。つまり、「伝達の規則」は適用されません。システムによって AUTOTRAN パラメータが確認されます。

  - AUTOTRAN に N が設定されている場合 (つまり AUTOTRAN が有効になっていない場合)、呼び出されたプロセスはトランザクション・モードになりません。
  - AUTOTRAN に Y が設定されている場合、呼び出されたプロセスはトランザクション・モードになります。ただし、新しいトランザクションとして処理されます。

注記 サービスは自動的にトランザクション・モードにできるので、TPNOTRAN フラグが設定されたサービスから、AUTOTRAN パラメータが設定されたサービスを呼び出すことができます。そのようなサービスが別のサービスを要求した場合、サービスのデータ構造体の *flags* メンバは照会を行ったときに TPTRAN を返します。たとえば、*flags* に TPNOTRAN | TPNOREPLY を設定して呼び出しを行い、サービスが呼び出されたときに (そのサービスによって) トランザクションが自動的に開始された場合、データ構造体の *flags* メンバは、TPTRAN | TPNOREPLY に設定されます。

XA 準拠のサーバ・グループに対するグローバル・トランザクションの定義

アプリケーション・プログラマが XA 準拠のサーバ・グループのサービスをコーディングする場合、グループのリソース・マネージャを介して操作を行うようにするのが一般的です。通常、サービスは 1 つのトランザクション内ですべての操作を行います。それに対して、*flags* に TPNOTRAN を設定してサービスを呼び出すと、データベース操作の実行時に予期しない結果を受け取る場合があります。

予測不能な動作を防ぐには、XA 準拠のリソース・マネージャに対応付けられているグループのサービスが、常にトランザクション・モードで呼び出されるようにアプリケーションを設計します。または、コンフィギュレーション・ファイルの AUTOTRAN に Y を設定します。また、サービス・コードの早い段階で、トランザクション・レベルを確認します。

トランザクションが開始されたことの確認

トランザクション・モードのプロセスが別のプロセスのサービスを要求した場合、後者のプロセスは不参加の指示が特にない限り、そのトランザクションのパーティシパントになります。

特定のエラー条件を回避して正しく解釈するには、プロセスがトランザクション・モードかどうかを確認することが大切です。たとえば、既にトランザクション・モードになっているプロセスが `tpbegin()` を呼び出すとエラーになります。そのようなプロセスが `tpbegin()` を呼び出すと、呼び出しは失敗し、`tperrno(5)` に TPEPROTO が設定されて、呼び出し元が既にトランザクションに参加しているにもかかわらず呼び出されたことが示されます。トランザクションに影響はありません。

サービス・サブルーチンがトランザクション・モードかどうかを確認した後で、tpbegin() を呼び出すようにアプリケーションを設計できます。次のいずれかの方法で、トランザクション・レベルを確認できます。

- サービス・サブルーチンに渡されるサービスのデータ構造体の *flags* フィールドを照会します。TPTRAN に設定されていると、サービスはトランザクション・モードになっています。
- tpgetlev(3c) 関数を呼び出します。

tpgetlev() 関数の呼び出しには、次の文法を使用します。

```
int
tpgetlev()  /*  現在のトランザクション・レベルを取得  */
```

tpgetlev() 関数に引数は必要ありません。この関数は、呼び出し元がトランザクション・モードになっていない場合は 0 を返し、トランザクション・モードになっている場合は 1 を返します。

次のコード例は、OPEN_ACCT サービスの 1 つであり、tpgetlev() 関数 (12 行目) を使用してトランザクション・レベルを確認する方法を示しています。プロセスがトランザクション・モードになっていない場合、アプリケーションでトランザクションを開始します (14 行目)。tpbegin() が失敗した場合、メッセージがステータス行に返され (16 行目)、tpreturn() の *rcode* 引数にグローバル変数 tpurcode(5) で取得できるコードが設定されます (1 行目と 17 行目)。

リスト 9-6　トランザクション・レベルの確認

```
001 #define BEGFAIL    3    /* tpbegin が失敗した場合に返す tpurcode の値を設定 */

002 void
003 OPEN_ACCT(transb)

004 TPSVCINFO *transb;

005 {
  ... other declarations ...
006 FBFR *transf;    /* 復号化されたメッセージのフィールド化バッファ */
007 int dotran;      /* tpbegin、tpcommit、tpaborts のどれであるかを確認 */

008 /* tpsvcinfo データ・バッファへのポインタを設定 */

009 transf = (FBFR *)transb->data;

010 /* トランザクションがあるかどうかを確認し、ない場合は開始、ある場合は確認 */
```

```
011 dotran = 0;
012 if (tpgetlev() == 0) {
013     dotran = 1;
014      if (tpbegin(30, 0) == -1) {
015         Fchg(transf, STATLIN, 0,
016             "Attempt to tpbegin within service routine failed\n");
017         tpreturn(TPFAIL, BEGFAIL, transb->data, 0, 0);
018     }
019 }
    . . .
```

AUTOTRAN に Y が設定されている場合、トランザクション関数の tpbegin()、tpcommit()、tpabort() を明示的に呼び出す必要はありません。その結果、トランザクション・レベルを確認するオーバーヘッドを減らすことができます。また、TRANTIME パラメータを設定して、タイムアウト間隔を指定することもできます。タイムアウト間隔は、サービスに対するトランザクションが開始されてからの経過時間です。また、トランザクションが完了しなかった場合は、トランザクションがロールバックされるまでの時間です。

たとえば、前述のコードの OPEN_ACCT サービスを変更するとします。現在のコードでは、OPEN_ACCT にトランザクションが明示的に定義され、そのトランザクションの有無を確認しています (7 行目、10 ～ 19 行目)。これらの処理のオーバーヘッドを減らすには、そのコードを削除します。その場合、OPEN_ACCT は常にトランザクション・モードで呼び出す必要があります。この要件を指定するには、コンフィギュレーション・ファイルの AUTOTRAN と TRANTIME システム・パラメータを有効にします。

関連項目

- 『BEA Tuxedo アプリケーションの設定』の第 9 章の 16 ページ「グローバル・トランザクションの暗黙的な定義」の AUTOTRAN コンフィギュレーション・パラメータ
- 『BEA Tuxedo アプリケーションの設定』の TRANTIME コンフィギュレーション・パラメータ

10 マルチスレッドおよびマルチコンテキスト・アプリケーションのプログラミング

ここでは、次の内容について説明します。

- マルチスレッドおよびマルチコンテキスト・アプリケーションのプログラミングに対するサポート
- マルチスレッドおよびマルチコンテキスト・アプリケーションの計画と設計
- マルチスレッドおよびマルチコンテキスト・アプリケーションのインプリメント
- マルチスレッドおよびマルチコンテキスト・アプリケーションのテスト

マルチスレッドおよびマルチコンテキスト・アプリケーションのプログラミングに対するサポート

BEA Tuxedo システムでは、次のアプリケーションがサポートされています。

- カーネルレベルのスレッド・パッケージ（ユーザレベルのスレッド・パッケージはサポートされていません）。
- C 言語で記述されたマルチスレッド・アプリケーション。COBOL でのマルチスレッド・アプリケーションはサポートされていません。
- C 言語または COBOL 言語で記述されたマルチコンテキスト・アプリケーション。

お使いのオペレーティング・システムで POSIX スレッド関数と共にほかのスレッド関数がサポートされている場合は、POSIX スレッド関数を使用することをお勧めします。この関数を使用すると、後でコードをほかのプラットフォームに簡単に移植できます。

お使いのプラットフォームでカーネルレベルのスレッド・パッケージ、C 言語の関数、または POSIX 関数がサポートされているかどうかを確認するには、『BEA Tuxedo システムのインストール』の付録 A の「BEA Tuxedo 8.0 プラットフォーム・データ・シート」で、使用しているオペレーティング・システムのデータ・シートを参照してください。

マルチスレッドおよびマルチコンテキスト・アプリケーションに関するプラットフォーム固有の検討事項

多くのプラットフォームには、マルチスレッドおよびマルチコンテキスト・アプリケーション固有の要件があります。プラットフォーム固有の要件については、『BEA Tuxedo システムのインストール』の付録 A の「BEA Tuxedo 8.0 プラットフォーム・データ・シート」に説明があります。お使いのプラットフォームの要件については、該当するデータシートを参照してください。

関連項目

- 第 10 章の 3 ページ「マルチスレッドおよびマルチコンテキストとは」
- 第 10 章の 7 ページ「マルチスレッドおよびマルチコンテキスト・アプリケーションの利点と問題点」
- 第 10 章の 9 ページ「クライアントでのマルチスレッドとマルチコンテキストの動作」
- 第 10 章の 15 ページ「サーバでのマルチスレッドとマルチコンテキストの動作」

マルチスレッドおよびマルチコンテキスト・アプリケーションの計画と設計

ここでは、次の内容について説明します。

- マルチスレッドおよびマルチコンテキストとは
- マルチスレッドおよびマルチコンテキスト・アプリケーションの利点と問題点
- クライアントでのマルチスレッドとマルチコンテキストの動作
- サーバでのマルチスレッドとマルチコンテキストの動作
- マルチスレッドおよびマルチコンテキスト・アプリケーションの設計上の検討事項

マルチスレッドおよびマルチコンテキストとは

BEA Tuxedo システムでは、単一のプロセスで複数のタスクを同時に実行できます。このようなプロセスをインプリメントするプログラミング手法はマルチスレッドおよびマルチコンテキストと呼ばれます。この節では、これらの手法に関する基本事項について説明します。

- マルチスレッドとは
- マルチコンテキストとは

マルチスレッドとは

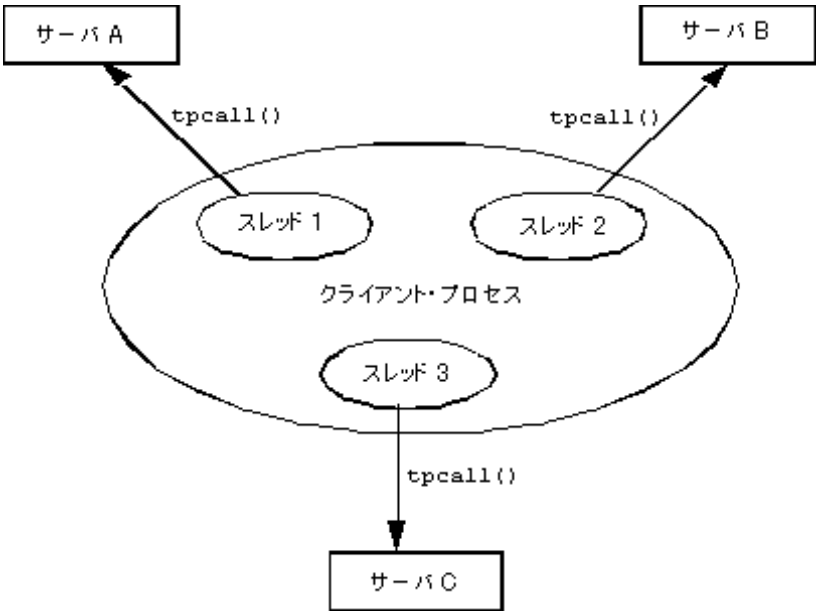
マルチスレッドとは、1 つのプロセスに複数の実行単位が含まれている処理方法です。マルチスレッド・アプリケーションでは、同じプロセスから同時に複数の呼び出しを行うことができます。たとえば、個々のプロセスが 1 つの未終了の `tpcall()` に制限されることはありません。

サーバのマルチスレッドでは、アプリケーション生成のスレッドがシングルコンテキスト・サーバで使用される場合を除き、マルチコンテキストが必要です。マルチスレッドのシングルコンテキスト・アプリケーションを作成する唯一の方法は、アプリケーション生成のスレッドを使用することです。

BEA Tuxedo システムでは、C 言語で記述されたマルチスレッド・アプリケーションがサポートされています。COBOL 言語のマルチスレッド・アプリケーションはサポートされていません。

次の図は、マルチスレッド・クライアントが 3 つのサーバに対して同時に呼び出しを行う方法を示しています。

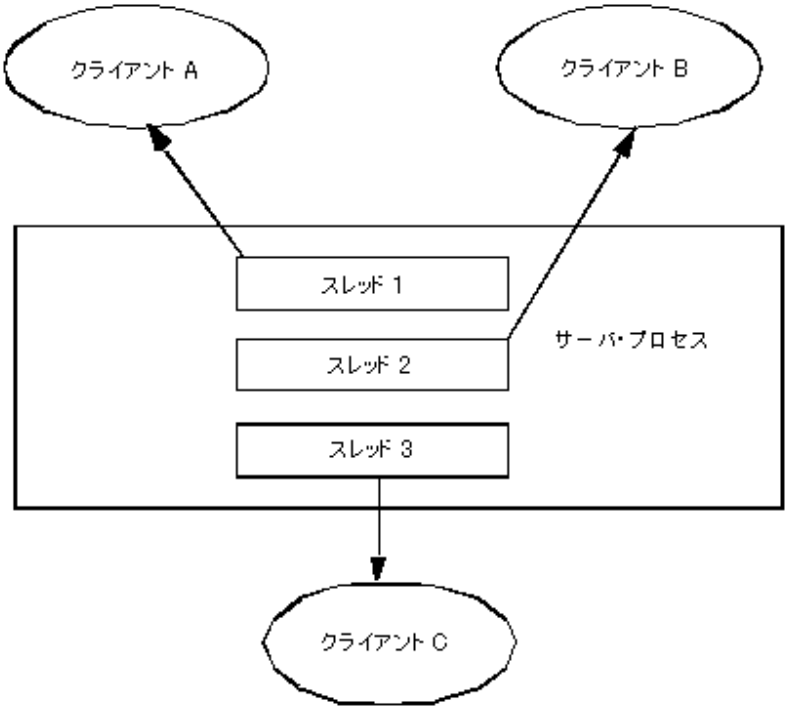
図 10-1 マルチスレッド・プロセスの例



マルチスレッド・アプリケーションでは、同じサーバで複数のサービス・ディスパッチ・スレッドを使用できるので、アプリケーションに対して起動するサーバ数が少なくて済みます。

次の図は、異なるクライアントに対して、サーバ・プロセスが同時に複数のスレッドをディスパッチする方法を示しています。

図 10-2　1 **つのサーバ・プロセスによる複数のサービス・スレッドのディスパッチ**



マルチコンテキストとは

コンテキストはドメインへの対応付けです。マルチコンテキストを使用すると、1 つのプロセスで次のいずれかが可能になります。

- ドメイン内での複数接続
- 複数ドメインへの接続

マルチコンテキストは、クライアントとサーバの両方で使用できます。サーバでマルチコンテキストを使用すると、マルチスレッドも使用することになります。

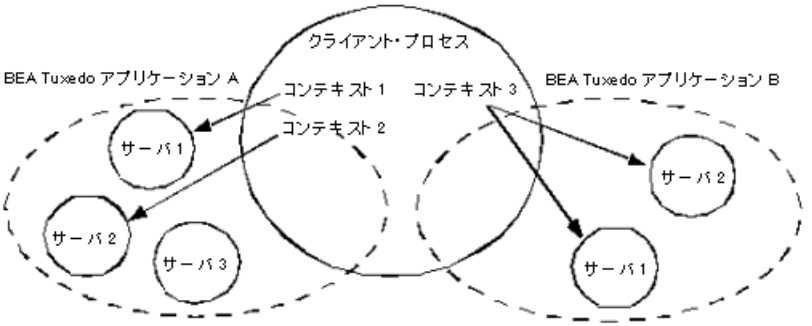
コンテキストの特徴の詳細については、次のいずれかの節で「コンテキストの属性」を参照してください。

- 第 10 章の 27 ページ「クライアントでマルチコンテキストを使用するためのコーディング」
- 第 10 章の 35 ページ「サーバでマルチコンテキストとマルチスレッドを使用するためのコーディング」

BEA Tuxedo システムでは、C 言語または COBOL 言語のいずれかで記述されたマルチコンテキスト・アプリケーションがサポートされています。ただし、サポートされるマルチスレッド・アプリケーションは、C 言語で記述されたものだけです。

次の図は、ドメイン内でのマルチコンテキスト・クライアント・プロセスの動作を示しています。矢印はサーバへの未終了の呼び出しを表します。

図 10-3 2 つのドメイン内でのマルチコンテキスト・プロセス



マルチスレッド・アプリケーションまたはマルチコンテキスト・アプリケーションのライセンス

ライセンスの関係で、各コンテキストは 1 人のユーザとしてカウントされます。1 つのコンテキストで複数のスレッドを使用するために、ライセンスを追加する必要はありません。次に例を示します。

- アプリケーション A に対応する 2 つのコンテキストと、アプリケーション B に対応する 1 つのコンテキストを持つプロセスの場合、アプリケーション A に 2 人、アプリケーション B に 1 人の合計 3 人のユーザとしてカウントされます。
- 1 つのアプリケーションにアクセスする複数のスレッドが同じコンテキスト内にあるプロセスの場合、ユーザは 1 人としてカウントされます。

関連項目

- 第 10 章の 7 ページ「マルチスレッドおよびマルチコンテキスト・アプリケーションの利点と問題点」
- 第 10 章の 9 ページ「クライアントでのマルチスレッドとマルチコンテキストの動作」
- 第 10 章の 15 ページ「サーバでのマルチスレッドとマルチコンテキストの動作」

マルチスレッドおよびマルチコンテキスト・アプリケーションの利点と問題点

マルチスレッドとマルチコンテキストを適切な状況で使用すると、BEA Tuxedo アプリケーションのパフォーマンスを向上できます。ただし、これらの手法を取り入れる前に、潜在的な利点と問題点について理解しておくことが大切です。

マルチスレッドおよびマルチコンテキスト・アプリケーションの利点

マルチスレッドおよびマルチコンテキスト・アプリケーションには、以下の利点があります。

- パフォーマンスと並列性の向上

  マルチスレッドとマルチコンテキストを併用すると、一部のアプリケーションではパフォーマンスと並列性が向上します。また、一部のアプリケーションでは、パフォーマンスが変わらないか、逆に低下する場合があります。パフォーマンスへの影響は、使用しているアプリケーションによって異なります。

- リモート・プロシージャ・コールと会話のコーディングの単純化

  アプリケーションによっては、異なるリモート・プロシージャ・コールと会話を別々のスレッドでコーディングした方が、同じスレッドで管理するより簡単な場合があります。

- 複数アプリケーションへの同時アクセス

  BEA Tuxedo クライアントを同時に複数のアプリケーションに接続できます。

- 必要最低限の数のサーバの使用

  1 つのサーバで複数のサービス・スレッドをディスパッチできるので、アプリケーションに対して起動するサーバの数を減らすことができます。このように複数のスレッドをディスパッチできる機能は、特に会話型サーバの場合に有用です。会話型サーバにこの機能がない場合、会話が終了するまで 1 つのクライアントしか使用できなくなります。

アプリケーションで、クライアント・スレッドが Microsoft Internet Information Server API または Netscape Enterprise Server インターフェイス (NSAPI) によって生成される場合、これらのツールの機能を最大限に利用するにはマルチスレッドが不可欠です。ほかのツールについても同じことが言えます。

マルチスレッドおよびマルチコンテキスト・アプリケーションの問題点

マルチスレッドおよびマルチコンテキスト・アプリケーションには、以下の問題点があります。

- コーディングの難しさ

  マルチスレッドおよびマルチコンテキスト・アプリケーションのコーディングは簡単ではありません。このようなアプリケーションのコーディングは、十分な経験を持つプログラマだけが行うことができます。

- デバッグの難しさ

  マルチスレッド・アプリケーションまたはマルチコンテキスト・アプリケーションで発生したエラーを再現することは、シングルスレッドおよびシングルコンテキスト・アプリケーションで再現するより難しい作業です。そのため、エラー発生時にその根本的な原因を特定して検証することがさらに難しくなります。

- 並列性の管理の難しさ

  スレッド間の並列性の管理は難しい作業であり、アプリケーションで新たに問題を引き起こす可能性があります。

- テストの難しさ

  マルチスレッド・アプリケーションのテストは、シングルスレッド・アプリケーションのテストより難しい作業です。問題がタイミングに関連していることが多く、再現が困難だからです。

- 既存コードの移植の難しさ

  既存のコードでマルチスレッドとマルチコンテキストを使用するには、大部分のコードを再構築しなければなりません。プログラマは、以下の作業を行う必要があります。

- 静的変数を削除します。
- スレッド・セーフではないすべての関数呼び出しを置き換えます。
- スレッド・セーフではないその他のコードを置き換えます。

移植が完了したら何度もテストを繰り返す必要があり、マルチスレッドおよびマルチコンテキスト・アプリケーションの移植には多くの作業が必要になります。

関連項目

- 第 10 章の 3 ページ「マルチスレッドおよびマルチコンテキストとは」
- 第 10 章の 9 ページ「クライアントでのマルチスレッドとマルチコンテキストの動作」
- 第 10 章の 15 ページ「サーバでのマルチスレッドとマルチコンテキストの動作」
- 第 10 章の 19 ページ「マルチスレッドおよびマルチコンテキスト・アプリケーションの設計上の検討事項」

クライアントでのマルチスレッドとマルチコンテキストの動作

マルチスレッドおよびマルチコンテキスト・アプリケーションがアクティブの場合、クライアントのライフ・サイクルは次の 3 つのフェーズから構成されます。

- 起動フェーズ
- 作業フェーズ
- 完了フェーズ

起動フェーズ

起動フェーズでは、次の操作が行われます。

- 一部のクライアント・スレッドは `tpinit()` を呼び出して、1 つ以上の BEA Tuxedo アプリケーションに参加します。
- その他のクライアント・スレッドは `tpsetctxt(3c)` を呼び出して、上記のスレッドによって生成されるコンテキストを共有します。

- 一部のクライアント・スレッドは複数のコンテキストに参加します。
- 一部のクライアント・スレッドは既存のコンテキストに切り替えます。

注記 BEA Tuxedo システムから独立して動作するスレッドが存在する場合もあります。ここでは、そのようなスレッドについては説明していません。

クライアント・スレッドの複数コンテキストへの参加

BEA Tuxedo マルチコンテキスト・アプリケーションのクライアントは、次の規則に従う限り、複数のアプリケーションに対応付けることができます。

- すべての対応付けは、BEA Tuxedo システムの同じインストールに対して行う必要があります。
- すべてのアプリケーションとの対応付けは、同じタイプのクライアントから行う必要があります。つまり、次のいずれの条件を満たしている必要があります。
  - すべてのアプリケーションとの対応付けは、ネイティブ・クライアントから行う必要があります。
  - すべてのアプリケーションとの対応付けは、ワークステーション・クライアントから行う必要があります。

クライアントが複数のコンテキストに参加するには、`TPINFO` データ型の *`flags`* に `TPMULTICONTEXTS` フラグを設定して、`tpinit()` 関数を呼び出します。

`TPMULTICONTEXTS` フラグを設定して `tpinit()` 関数が呼び出されると、アプリケーションとの新しい対応付けが生成され、スレッドに対するカレントの対応付けが指定されます。新しい対応付けが生成される BEA Tuxedo ドメインは、`TUXCONFIG` または `WSENVFILE/WSNADDR` 環境変数の値で決定されます。

クライアント・スレッドの既存のコンテキストへの切り替わり

多くの ATMI 関数はコンテキスト単位で動作します。このような ATMI 関数のリストについては、第 10 章の 48 ページ「マルチスレッド・クライアントでのコンテキスト単位の関数とデータ構造体」を参照してください。コンテキスト単位で動作するには、ターゲット・コンテキストが現在のコンテキストであることが必要です。クライアントは複数のコンテキストに参加できますが、状況とスレッドにかかわらず現在のコンテキストになることができるコンテキストは 1 つだけです。

アプリケーション内でタスクの優先順位が移り、ほかの BEA Tuxedo ドメインと通信する必要が発生した場合、あるコンテキストから別のコンテキストにスレッドを再割り当てした方がよい場合があります。

そのような場合、あるクライアント・スレッドが tpgetctxt(3c) を呼び出し、返された現在のコンテキストを値として持つハンドルを別のクライアント・スレッドに渡します。2 番目のスレッドは tpsetctxt(3c) を呼び出し、最初のスレッドで tpgetctxt(3c) から受け取ったハンドルを指定して、現在のコンテキストとの対応付けを確立します。

目的のコンテキストとの対応付けが確立されると、2 番目のスレッドはコンテキスト単位で動作する ATMI 関数を使用してタスクを実行できるようになります。詳細については、第 10 章の 48 ページ「マルチスレッド・クライアントでのコンテキスト単位の関数とデータ構造体」を参照してください。

作業フェーズ

このフェーズでは、各スレッドによって処理が行われます。次は、行われる処理の例です。

- サービスを要求します。
- サービス要求に対する応答を受け取ります。
- 会話を開始して会話に参加します。
- トランザクションの開始、コミット、またはロールバックを行います。

サービス要求

スレッドは、同期要求の場合は tpcall()、非同期要求の場合は tpacall() を呼び出して、サーバに要求を送ります。tpcall() で要求を送った場合、以降操作を行わなくても応答を受け取ることができます。

サービス要求に対する応答

tpcall() でサービスの非同期要求を送った場合、同じコンテキスト内のスレッドは tpgetrply() を呼び出して応答を受け取ります。このスレッドは、要求を送ったスレッドと同じスレッドではない場合もあります。

トランザクション

あるスレッドがトランザクションを開始すると、そのスレッドのコンテキストを共有するすべてのスレッドでそのトランザクションが共有されます。

コンテキスト内の多くのスレッドでトランザクションに関する処理が行われますが、トランザクションをコミットまたはアボートできるのは 1 つのスレッドだけです。トランザクションをコミットまたはアボートするスレッドは、トランザクションを開始したスレッドである必要はなく、トランザクションを処理しているどのスレッドでもかまいません。スレッド・アプリケーションでは、通常のトランザクション規則に従うために、適切に同期を行う必要があります。たとえば、未終了の RPC 呼び出しや会話がある場合に、トランザクションをコミットすることはできません。また、トランザクションがコミットまたはアボートされた後で、そのトランザクションに対する呼び出しを行うことはできません。プロセスは、アプリケーションの各対応付けに対して、1 つのトランザクションの一部にだけなることができます。

アプリケーションの 1 つのスレッドが `tpcommit()` を呼び出し、それと同時に別のスレッドが RPC 呼び出しまたは会話型呼び出しを行うと、これらの呼び出しは特定の順序で呼び出されたものとして処理されます。アプリケーション・コンテキストは、シングルスレッド・プログラムとシングルコンテキスト・プログラムに対する制約と同じ制約に従って、`tpsuspend()` を呼び出してトランザクションを一時的に中断し、別のトランザクションを開始します。

任意通知型メッセージ

マルチスレッド・アプリケーションまたはマルチコンテキスト・アプリケーションの各コンテキストでは、任意通知型メッセージを次の 3 種類のいずれかの方法で処理できます。

処理方法	設定
任意通知型メッセージの無視	`TPU_IGN`
ディップ・イン通知	`TPU_DIP`
専用のスレッド通知 (C 言語のアプリケーションのみで利用可能です)	`TPU_THREAD`

以下の事柄に注意してください。

- シグナル・ベースの通知は、マルチスレッド・プロセスまたはマルチコンテキスト・プロセスでは使用できません。

- アプリケーションを実行しているプラットフォームで、マルチコンテキストがサポートされていてもマルチスレッドがサポートされていない場合、TPU_THREAD を使用した任意通知型通知の処理を行うことはできません。そのため、イベントの即時通知を受け取ることはできません。

  イベントの即時通知を受け取る必要がある場合は、そのプラットフォームでマルチコンテキストを使用するかどうかを慎重に検討します。
- 専用のスレッド通知は、次のものに対してだけ使用できます。
  - C 言語で記述されたアプリケーション
  - BEA Tuxedo システムでサポートされているマルチスレッド・プラットフォーム

専用のスレッド通知の場合、任意通知型メッセージの受信と、任意通知型メッセージ・ハンドラのディスパッチに別々のスレッドが使用されます。あるコンテキストで一度に実行できる任意通知型メッセージ・ハンドラは 1 つだけです。

スレッドがサポートされていない BEA Tuxedo システム用プラットフォームで tpinit() が呼び出された場合に、スレッドがサポートされていないプラットフォーム上で TPU_THREAD 通知が要求されたことを示すパラメータが指定されていると、tpinit() は -1 を返して tperrno に TPEINVAL を設定します。UBBCONFIG(5) のデフォルトの NOTIFY オプションが THREAD に設定されている場合に、特定のマシンでスレッドを利用できないと、そのマシンのデフォルトの機能は DIPIN になります。このような動作の相違があるので、スレッドがサポートされているマシンとサポートされていないマシンが混在する環境では、管理者はすべてのマシンにデフォルトを指定できます。ただし、そのマシンで利用できない機能をクライアントが明示的に要求することはできません。

tpsetunsol() がコンテキストに対応付けされていないスレッドから呼び出されると、新しく生成されるすべての tpinit() コンテキストに対して、プロセス単位のデフォルトの任意通知型メッセージ・ハンドラが作成されます。特定のコンテキストは、コンテキストがアクティブのときに tpsetunsol() を再度呼び出して、そのコンテキストの任意通知型メッセージ・ハンドラを変更することができます。プロセス単位のデフォルトの任意通知型メッセージ・ハンドラは、コンテキストに現在対応付けされていないスレッドで tpsetunsol() を再度呼び出すと、変更できます。

プロセスが同じアプリケーションと複数の対応付けを持つ場合、各対応付けに異なる CLIENTID を割り当てられ、任意通知型メッセージを特定のアプリケーションとの対応付けに送信できるようになります。プロセスが同じアプリケーションと複数の対応付けを持つ場合、ブロードキャスト基準を満たすアプリケーションの各対応付けに任意の tpbroadcast() が別々に送信されます。任意通知型メッセージを受信する場合のディップ・イン・チェックでは、カレントのアプリケーションとの対応付けに送信されるメッセージだけが対象となります。

任意通知型メッセージ・ハンドラでは ATMI 関数を利用できるほか、任意通知型メッセージ・ハンドラ内で tpgetctxt(3c) を呼び出すことができます。そのため、任意通知型メッセージ・ハンドラは別のスレッドを生成して、同じコンテキスト内で必要となる実質的な ATMI 作業を行うことができるようになります。

ユーザ・ログで保持されるスレッド固有の情報

userlog(3c) を使用すると、各アプリケーション内の各スレッドに対して次の識別情報が記録されます。

process_ID.thread_ID.context_ID

スレッドがサポートされていないプラットフォームやシングルコンテキスト・アプリケーションに対しては、*thread_ID* フィールドと *context_ID* フィールドにプレースホルダが出力されます。

TM_MIB(5) では、この機能は T_ULOG クラスの TA_THREADID フィールドと TA_CONTEXTID フィールドでサポートされています。

完了フェーズ

このフェーズでは、クライアント・プロセスの終了時に、カレントのコンテキストおよび対応付けられたすべてのスレッドに代わって 1 つのスレッドが tpterm() を呼び出してそのアプリケーションとの対応付けを終了します。ほかの ATMI 関数と同じように、tpterm() は現在のコンテキストに対して処理を行います。tpterm() は、終了するコンテキストに対応付けされたすべてのスレッドに影響し、これらのスレッドで共有されるすべてのコンテキストを終了します。

良く設計されたアプリケーションでは、特定のコンテキスト内のすべての処理が完了してから tpterm() が呼び出されます。tpterm() が呼び出される前に、すべてのスレッドが同期していなければなりません。

関連項目

- 第 10 章の 3 ページ「マルチスレッドおよびマルチコンテキストとは」
- 第 10 章の 19 ページ「マルチスレッドおよびマルチコンテキスト・アプリケーションの設計上の検討事項」
- 第 10 章の 27 ページ「クライアントでマルチコンテキストを使用するためのコーディング」

- 第 10 章の 41 ページ「マルチスレッド・クライアントのコーディング」
- 第 10 章の 30 ページ「クライアント終了前のスレッドの同期」

サーバでのマルチスレッドとマルチコンテキストの動作

マルチスレッドおよびマルチコンテキスト・アプリケーションがアクティブの場合、サーバで行われる処理は次の 3 つのフェーズに分類できます。

- 起動フェーズ
- 作業フェーズ
- 完了フェーズ

起動フェーズ

起動フェーズで行われる処理は、コンフィギュレーション・ファイルの MINDISPATCHTHREADS と MAXDISPATCHTHREADS パラメータの値によって異なります。

MINDISPATCHTHREADS の値	MAXDISPATCHTHREADS の値	処理内容
0	> 1	1. BEA Tuxedo システムがスレッド・ディスパッチャを生成します。 2. ディスパッチャが tpsvrinit() を呼び出して、アプリケーションに参加します。
> 0	> 1	1. BEA Tuxedo システムがスレッド・ディスパッチャを生成します。 2. ディスパッチャが tpsvrinit() を呼び出して、アプリケーションに参加します。 3. BEA Tuxedo システムがサービス要求を処理する新しいスレッドと、そのスレッドに対するコンテキストを生成します。 4. システムで生成された新しいスレッドがそれぞれ tpsvrthrinit(3c) を呼び出して、アプリケーションに参加します。

作業フェーズ

このフェーズでは、次の処理が行われます。

- 1 つのサーバに対する複数のクライアント要求が、複数のコンテキストで同時に処理されます。要求ごとに別のスレッドが割り当てられます。
- 必要に応じて、MAXDISPATCHTHREADS で指定された値まで新しいスレッドが生成されます。
- サーバ・スレッドの統計がシステムで保持されます。

サーバ・ディスパッチ・スレッド

クライアントのサービス要求への応答として、サーバ・ディスパッチャは設定可能な最大数まで複数のスレッドを 1 つのサーバに生成します。このサーバは、各種のクライアント要求に同時に割り当てることができます。サーバが tpinit() を呼び出してクライアントになることはできません。

各ディスパッチ・スレッドは、別々のコンテキストと対応付けられています。この機能は会話型サーバと RPC サーバの両方で有用です。特に、会話型サーバではこの機能を利用できないと、ほかの会話接続がサービスを待っている間、クライアント側の会話をアイドル状態で待つことになります。

この機能は、UBBCONFIG(5) ファイルの SERVERS セクションと TM_MIB(5) の次のパラメータで制御されます。

UBBCONFIG パラメータ	MIB パラメータ	デフォルト値
MINDISPATCHTHREADS	TA_MINDISPATCHTHREADS	0
MAXDISPATCHTHREADS	TA_MAXDISPATCHTHREADS	1
THREADSTACKSIZE	TA_THREADSTACKSIZE	0 （オペレーティング・システムのデフォルト値）

- 各ディスパッチ・スレッドは、THREADSTACKSIZE または TA_THREADSTACKSIZE で指定されるスタック・サイズで生成されます。このパラメータが指定されていない場合または 0 の場合、オペレーティング・システムのデフォルト値が使用されます。オペレーティング・システムのデフォルト値が小さすぎて BEA Tuxedo システムで使用できない場合、その値より大きなデフォルト値が使用されます。

- このパラメータが指定されていない場合または 0 の場合、あるいはオペレーティング・システムで THREADSTACKSIZE 設定がサポートされていない場合は、オペレーティング・システムのデフォルト値が使用されます。
- MINDISPATCHTHREADS または TA_MINDISPATCHTHREADS は、MAXDISPATCHTHREADS または TA_MAXDISPATCHTHREADS 以下でなければなりません。
- MAXDISPATCHTHREADS または TA_MAXDISPATCHTHREADS が 1 の場合、ディスパッチャ・スレッドとサービス関数スレッドは同じスレッドです。
- MAXDISPATCHTHREADS または TA_MAXDISPATCHTHREADS が 1 より大きい場合、ほかのスレッドのディスパッチに使用されるスレッドは、ディスパッチ・スレッドとしてカウントされません。
- システムは最初に MINDISPATCHTHREADS または TA_MINDISPATCHTHREADS のサーバ・スレッドを起動します。
- システムが MAXDISPATCHTHREADS または TA_MAXDISPATCHTHREADS を超えるサーバ・スレッドを起動することはありません。

アプリケーション生成のスレッド

オペレーティング・システム関数を使用して、アプリケーション・サーバ内に新しいスレッドを追加できます。アプリケーション生成のスレッドは、次のように動作します。

- BEA Tuxedo システムから独立して動作します。
- 既存のサーバ・ディスパッチ・スレッドと同じコンテキストで動作します。
- サーバ・ディスパッチ・コンテキストに代わって作業を行います。

アプリケーション内にスレッドを生成する場合、次の制約があります。

- サーバは tpinit() を呼び出してクライアントになることはできません。
- 最初、アプリケーション生成サーバ・スレッドは、どのサーバ・ディスパッチ・コンテキストにも関連していません。アプリケーション生成のサーバ・スレッドは tpsetctxt(3c) を呼び出し、サーバ・ディスパッチ・スレッド内の以前の tpgetctxt(3c) 呼び出しによって返される値を渡して、サーバ・ディスパッチ・コンテキストとの対応付けを確立します。
- アプリケーション生成のサーバ・スレッドは、tpreturn() または tpforward() を呼び出すことはできません。アプリケーション生成のサーバ・スレッドは処理が終了したら、元のディスパッチ・スレッドが tpreturn() を呼び出す前に、TPNULLCONTEXT に設定されたコンテキストで tpsetctxt(3c) を呼び出す必要があります。

BBL によるシステム・プロセスの正常性チェック

BBL は定期的にサーバを検証します。特定のサービス要求の実行に時間がかかりすぎている場合、BBL はそのサーバを強制終了します。そして、指定されている場合は、そのサーバを再起動します。BBL がマルチコンテキスト・サーバを強制終了した場合、プロセスを強制終了した結果として、実行中のそのほかのサービス呼び出しも終了します。

また、BBL はタイムアウト値を超えてメッセージの受信を待機しているプロセスまたはスレッドにメッセージを送信します。ブロッキング・メッセージ受信への呼び出しがタイムアウトを示すエラーを返します。

システムで保持されるサーバ・スレッドの統計

BEA Tuxedo システムでは、各サーバに対して次の統計情報が保持されます。

- 使用できるサーバ・ディスパッチ・スレッドの最大数
- 現在使用中のサーバ・ディスパッチ・スレッドの数 (TA_CURDISPATCHTHREADS)
- サーバ起動後の同時実行サーバ・ディスパッチ・スレッドの最大数 (TA_HWDISPATCHTHREADS)
- これまでに起動したサーバ・ディスパッチ・スレッドの数 (TA_NUMDISPATCHTHREADS)

ユーザ・ログで保持されるスレッド固有の情報

userlog(3c) を使用すると、各アプリケーション内の各スレッドに対して次の識別情報が記録されます。

process_ID.thread_ID.context_ID

スレッドがサポートされていないプラットフォームやシングルコンテキスト・アプリケーションに対しては、*thread_ID* フィールドと *context_ID* フィールドにプレースホルダが出力されます。

TM_MIB(5) では、この機能は T_ULOG クラスの TA_THREADID フィールドと TA_CONTEXTID フィールドでサポートされています。

完了フェーズ

アプリケーションをシャットダウンすると、tpsvrthrdone(3c) と tpsvrdone(3c) が呼び出されて、リソース・マネージャのクローズなど、必要な終了処理が行われます。

関連項目

- 第 10 章の 3 ページ「マルチスレッドおよびマルチコンテキストとは」
- 第 10 章の 19 ページ「マルチスレッドおよびマルチコンテキスト・アプリケーションの設計上の検討事項」
- 第 10 章の 35 ページ「サーバでマルチコンテキストとマルチスレッドを使用するためのコーディング」
- 第 10 章の 55 ページ「マルチスレッド・サーバのコーディング」

マルチスレッドおよびマルチコンテキスト・アプリケーションの設計上の検討事項

マルチスレッドおよびマルチコンテキスト・アプリケーションは、一部の BEA Tuxedo ドメインでは正しく動作しますが、すべてのドメインで正しく動作するとは限りません。そのようなアプリケーションを作成するかどうかは、次の基本事項を検討してから決定します。

- 開発環境と実行時環境
- アプリケーションの設計上の要件
- 使用するスレッド・モデルのタイプ
- ワークステーション・クライアントに対する相互運用性での制約

環境の要件

マルチスレッド・アプリケーションまたはマルチコンテキスト・アプリケーションの開発では、開発環境と実行時環境に関して次の内容を検討します。

- 十分な経験を持つプログラマ・チームがあり、並列性と同期を正しく管理するマルチスレッドおよびマルチコンテキスト・プログラムのコーディングとデバッグを行うことができるかどうか。
- アプリケーションを開発するプラットフォームで、BEA Tuxedo システムのマルチスレッド機能がサポートされているかどうか。マルチスレッド機能は、オペレーティング・システムで提供されるスレッド・パッケージがインストールされ、適切なレベルの機能性が提供されたプラットフォームだけでサポートされます。
- サーバで使用されるリソース・マネージャ (RM) でマルチスレッドがサポートされているかどうか。サポートされている場合は、次の内容も検討します。
  - サーバでマルチスレッド・アクセスを使用するために RM に必要なパラメータを設定する必要があるかどうか。たとえば、マルチスレッド・アプリケーションで Oracle データベースを使用する場合、Oracle に渡される `OPENINFO` 文字列の一部として `THREADS=true` パラメータを設定する必要があります。この設定により、各スレッドが Oracle との別々の対応付けとして動作するようになります。
  - RM で処理の混在モードがサポートされているかどうか。処理の混在モードは、あるプロセスの複数のスレッドが、1 つの RM との対応付けにマップでき、また同じプロセスのほかのスレッドが別の RM との対応付けにマップできるアクセス方法です。たとえば、1 つのプロセス内でスレッド A と B が RM との対応付け X にマップし、同時にスレッド C が RM との対応付け Y にマップできます。

    すべての RM で混在モードがサポートされているわけではありません。プロセス内のすべてのスレッドが同じ RM との対応付けにマップされなければならない場合もあります。アプリケーション生成のスレッド内でトランザクションに関与する RM アクセスを使用するアプリケーションを設計する場合は、RM で混在モードがサポートされていることを確認します。

設計の要件

マルチスレッド・アプリケーションやマルチコンテキスト・アプリケーションの設計では、次の内容を検討します。

- アプリケーションによって実行されるタスクが、マルチスレッドやマルチコンテキストに適しているかどうか。
- 複数の BEA Tuxedo アプリケーションに接続するかどうか。各ターゲット・アプリケーションに必要な接続数はいくつか。
- アプリケーションで考慮すべき同期に関する検討事項は何か。
- 初期アプリケーションの完成後、アプリケーションを別のプラットフォームに移植する必要があるかどうか。

マルチスレッドやマルチコンテキストに適するアプリケーションのタスク

次の表は、アプリケーションをマルチスレッドまたはマルチコンテキストにすべきかどうかを判断するための参考となる検討事項を示しています。この表だけでは十分ではないので、個々の要件に基づいてほかの事項も検討してください。

そのほかの検討事項については、マルチスレッド・アプリケーションやマルチコンテキスト・アプリケーションのプログラミングに関する書籍を参照してください。

検討事項	検討事項に該当する場合に使用する機能
ドメイン機能を使用せずに、クライアントが複数のアプリケーションに接続する必要があるか。	マルチコンテキスト。
アプリケーション内でクライアントが多重化の機能を果たすか。たとえば、アプリケーション内の 1 つのマシンがそのほかの 100 台のマシンの「代理」に指定されているか。	マルチコンテキスト。
クライアントでマルチコンテキストが使用されるか。	マルチスレッド。各コンテキストにスレッドを 1 つずつ割り当てると、コードを簡略化できます。
クライアントが 2 つ以上のタスクを長時間個別に実行することで、並列処理によるパフォーマンスがスレッド同期のコストと複雑さを上回るか。	マルチスレッド。

検討事項	検討事項に該当する場合に使用する機能
1 つのサーバで複数の要求を同時に処理するか。	マルチスレッド。MAXDISPATCHTHREADS に 1 より大きな値を割り当てます。このように設定すると、1 つのサーバで複数のクライアントをそのクライアントのスレッドで処理できるようになります。
クライアントまたはサーバに複数のスレッドがある場合、各スレッドでわずかな処理を行った後でもスレッドを同期する必要があるか。	マルチスレッドを使用しない。

必要なアプリケーションと接続の数

アクセスするアプリケーションの数と、確立する接続の数を決定します。

- 複数のアプリケーションに接続する場合は、次のいずれかを使用します。
  - シングルスレッドおよびマルチコンテキスト・アプリケーション
  - マルチスレッドおよびマルチコンテキスト・アプリケーション
- 1 つのアプリケーションに複数の接続を確立する場合は、マルチスレッドおよびマルチコンテキスト・アプリケーションを使用します。
- 1 つのアプリケーションに 1 つだけ接続を確立する場合は、次のいずれかを使用します。
  - マルチスレッドおよびシングルコンテキスト・クライアント
  - シングルスレッドおよびシングルコンテキスト・クライアント

  いずれの場合も、マルチスレッドおよびマルチコンテキスト・サーバを使用できます。

同期に関する検討事項

これは設計段階での重要な検討事項です。このマニュアルでは、この内容について取り上げていません。マルチスレッド・アプリケーションやマルチコンテキスト・アプリケーションのプログラミングに関する書籍を参照してください。

アプリケーションの移植

後でアプリケーションを移植する必要がある場合、オペレーティング・システムに応じて異なる関数が使用されていることに注目してください。あるプラットフォームで作成した初期バージョンのアプリケーションを後で移植する場合、異なる関数でコードを書き直すためにどれだけのリソース時間が必要なのかを考慮する必要があります。

最適なスレッド・モデル

現在使用されているマルチスレッド・プログラムには、次のようなモデルがあります。

- 従属 / 被従属モデル
- 兄弟モデル
- ワークフロー・モデル

スレッド・モデルについては、このマニュアルでは取り上げていません。アプリケーションのプログラミング・モデルを選択する場合は、利用できるすべてのモデルを調べて、設計の要件を慎重に検討してください。

ワークステーション・クライアントの相互運用性に関する制約

リリース 7.1 の Workstation クライアントと 7.1 以前の BEA Tuxedo システムに基づくアプリケーションとの相互運用性は、次のどの場合でもサポートされています。

- マルチスレッド化またはマルチコンテキスト化されたクライアントではない。
- マルチコンテキスト化されたクライアントである。
- クライアントがマルチスレッド化されており、各スレッドが異なるコンテキストにある。

BEA Tuxedo リリース 7.1 のワークステーション・クライアントで、1 つのコンテキストに複数のスレッドがある場合は、リリース 7.1 より前の BEA Tuxedo システムとは相互運用しません。

関連項目

- 第 10 章の 7 ページ「マルチスレッドおよびマルチコンテキスト・アプリケーションの利点と問題点」
- 第 10 章の 25 ページ「マルチスレッドおよびマルチコンテキスト・アプリケーションのプログラミング開始前のガイドライン」

マルチスレッドおよびマルチコンテキスト・アプリケーションのインプリメント

- 第 10 章の 25 ページ「マルチスレッドおよびマルチコンテキスト・アプリケーションのプログラミング開始前のガイドライン」
- 第 10 章の 27 ページ「クライアントでマルチコンテキストを使用するためのコーディング」
- 第 10 章の 35 ページ「サーバでマルチコンテキストとマルチスレッドを使用するためのコーディング」
- 第 10 章の 41 ページ「マルチスレッド・クライアントのコーディング」
- 第 10 章の 55 ページ「マルチスレッド・サーバのコーディング」
- 第 10 章の 55 ページ「マルチスレッドおよびマルチコンテキスト・アプリケーションのコードのコンパイル」

マルチスレッドおよびマルチコンテキスト・アプリケーションのプログラミング開始前のガイドライン

コーディングを開始する前に、次の内容や条件を満たしていることを確認してください。

- 第 10 章の 25 ページ「マルチスレッド・アプリケーションに必要な条件」
- 第 10 章の 26 ページ「マルチスレッド・アプリケーションのプログラミングでの一般的な検討事項」
- 第 10 章の 26 ページ「並列性に関する検討事項」

マルチスレッド・アプリケーションに必要な条件

開発プロジェクトを開始する前に、開発環境が次の条件を満たしていることを確認します。

- BEA Tuxedo システムでサポートされる適切なスレッド・パッケージが、オペレーティング・システムで提供されていることが必要です。

  BEA Tuxedo システムには、スレッド生成用のツールは提供されていません。ただし、ほかのオペレーティング・システムで提供される各種のスレッド・パッケージがサポートされています。スレッドを生成して同期するには、オペレーティング・システム固有の関数を使用する必要があります。オペレーティング・システムでサポートされているスレッド・パッケージを確認するには、『BEA Tuxedo システムのインストール』の付録 A の「BEA Tuxedo 8.0 プラットフォーム・データ・シート」を参照してください。
- マルチスレッド・サーバを使用している場合は、これらのサーバで使用されるリソース・マネージャでスレッドがサポートされていることが必要です。

マルチスレッド・アプリケーションのプログラミングでの一般的な検討事項

マルチスレッド・プログラムは、十分に経験を持つプログラマがコーディングします。特に、次のようなマルチスレッド固有の設計に関する基本的な知識があることが必要です。

- 複数スレッド間の同時実行制御の必要性
- ほどんどのインスタンスで静的変数の使用を避ける必要性
- マルチスレッド・プログラムでシグナルを使用することにより発生する可能性のある問題

これらは検討事項の一部にすぎず、ここに記せないほど多くの検討事項があります。マルチスレッド・プログラムをコーディングするプログラマは、それらの検討事項を熟知していることが前提となります。これらの検討事項については、マルチスレッド・アプリケーションのプログラミングに関する書籍を参照してください。

並列性に関する検討事項

マルチスレッドを使用すると、1 つのアプリケーションの異なるスレッドが同じ会話で並列処理を行うことができるようになります。この方法はお勧めしませんが、BEA Tuxedo システムで禁止されているわけではありません。異なるスレッドによって同じ会話で並列処理が行われると、システムは同時呼び出しが任意の順序で行われたように動作します。

複数のスレッドを使ってプログラミングする場合、ミューテックスなどの同時実行制御関数を使用して、スレッド間の並列処理を管理する必要があります。以下は、同時実行制御が必要になる 3 つの例です。

- マルチスレッドのスレッドが同じコンテキストで動作する場合、プログラマは関数が必要な順序で実行されるようにします。たとえば、すべての RPC 呼び出しと会話が完了した後でのみ、`tpcommit()` を呼び出すようにします。これらすべての RPC 呼び出しまたは会話型呼び出しが行われるスレッドとは別のスレッドから `tpcommit()` が呼び出される場合、アプリケーションでなんらかの同時実行制御が必要になります。
- 同じように、`tpacall()` と `tpgetrply()` の呼び出しを別々のスレッドで行うことはできますが、アプリケーションが次のいずれかの条件を満たしている必要があります。
  - `tpacall()` が呼び出された後で `tpgetrply()` が呼び出されます。

  - tpacall() を呼び出す前に tpgetrply() が呼び出される場合、結果が管理されます。
- 複数のスレッドが同じ会話で処理を行うことは可能です。ただし、異なるスレッドが tpsend() をほとんど同時に呼び出した場合、システムはそれらの tpsend() 呼び出しが任意の順序で行われたように動作します。アプリケーション・プログラマは、それを認識しておかなければなりません。

  ほとんどのアプリケーションで最良の方法は、1 つの会話のすべての処理を 1 つのスレッドにまとめてコーディングすることです。また、同時実行制御を使用して、これらの処理を連続して行う方法もあります。

関連項目

- 第 10 章の 19 ページ「マルチスレッドおよびマルチコンテキスト・アプリケーションの設計上の検討事項」
- 第 10 章の 27 ページ「クライアントでマルチコンテキストを使用するためのコーディング」
- 第 10 章の 35 ページ「サーバでマルチコンテキストとマルチスレッドを使用するためのコーディング」
- 第 10 章の 41 ページ「マルチスレッド・クライアントのコーディング」
- 第 10 章の 55 ページ「マルチスレッド・サーバのコーディング」

クライアントでマルチコンテキストを使用するためのコーディング

クライアントでマルチコンテキストを使用するには、次の内容をコーディングします。

- 初期化時にマルチコンテキストを設定します。
- セキュリティをインプリメントします。
- マルチスレッドも使用する場合は、スレッドを同期します。
- コンテキストを切り替えます。
- 各コンテキストの任意通知型メッセージを処理します。

アプリケーションでトランザクションを使用する場合、トランザクションのマルチコンテキストの結果についても注目します。詳細については、第 10 章の 34 ページ「マルチスレッドおよびマルチコンテキスト・アプリケーションにおけるトランザクションのコーディング規則」を参照してください。

注記 この節で示す手順とコード例は、BEA Tuxedo システムで提供される C 言語のライブラリ関数を参照します。それらに相当する COBOL ライブラリ関数も利用できます。詳細については、『BEA Tuxedo COBOL リファレンス』を参照してください。

コンテキストの属性

コンテキストを使用する場合、コーディングで以下の事柄に注意してください。

- 元のディスパッチ・スレッドが終了する前に、アプリケーション生成のサーバ・スレッドがコンテキストを変更せずに終了すると、`tpreturn()` または `tpforward()` が失敗します。スレッドを終了しても、自動的に `tpsetctxt(3c)` が呼び出されてコンテキストが `TPNULLCONTEXT` に変更されることはありません。
- プロセス内のすべてのコンテキストに、同じバッファ・タイプ・スイッチを使用します。
- ほかのタイプのデータ構造体と同じように、マルチスレッド・アプリケーションでは BEA Tuxedo バッファを適切に使用する必要があります。つまり、次のいずれかが該当する場合、バッファを 2 つの呼び出しで同時に使用することはできません。
  - 両方の呼び出しでバッファが使用される場合
  - 両方の呼び出しでバッファを解放する場合
  - 1 つの呼び出しでバッファを使用し、もう 1 つの呼び出しでバッファを解放する場合
- `tpinit()` を複数回呼び出して、複数アプリケーションに参加するか、または単一アプリケーションに複数の接続を行う場合、`tpinit()` を呼び出すたびに、確立されているセキュリティ・メカニズムを調整する必要があります。

初期化時のマルチコンテキストの設定

クライアントがアプリケーションに参加する準備ができたら、次のコード例に示すように、TPMULTICONTEXTS フラグを設定して tpinit() を指定します。

リスト 10-1 クライアントのマルチコンテキスト・アプリケーションへの参加

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

TPINIT   *   tpinitbuf;

main()
{
      tpinitbuf = tpalloc(TPINIT, NULL, TPINITNEED(0));

      tpinitbuf->flags = TPMULTICONTEXTS;
            .
            .
            .
      if (tpinit (tpinitbuf) == -1) {
            ERROR_PROCESSING_CODE
      }
                  .
                  .
                  .
}
```

新しいアプリケーションとの対応付けが生成され、TUXCONFIG または WSENVFILE/WSNADDR 環境変数で指定された BEA Tuxedo ドメインに割り当てられます。

注記 1 つのプロセスでは、tpinit() へのすべての呼び出しに TPMULTICONTEXTS フラグを含めるか、または tpinit() へのすべての呼び出しにこのフラグを含めません。この規則には、例外が 1 つあります。つまり、tpterm() への呼び出しが正常に終了して、クライアントのすべてのアプリケーション対応付けが終了した場合、次に tpinit() を呼び出すときに必ずしも TPMULTICONTEXTS フラグを含む必要のない状態にプロセスが復元されます。

マルチコンテキスト・クライアントのセキュリティのインプリメント

同じプロセス内の各アプリケーションとの対応付けには、別個にセキュリティ検査を行う必要があります。検査の内容は、アプリケーションで使用されているセキュリティ・メカニズムのタイプによって異なります。たとえば、BEA Tuxedo アプリケーションでは、システム・レベルのパスワードまたはアプリケーション・パスワードを使用します。

マルチコンテキスト・アプリケーションのプログラマは、アプリケーションで使用するセキュリティのタイプを決定し、そのセキュリティをプロセス内の各アプリケーションとの対応付けにインプリメントします。

クライアント終了前のスレッドの同期

クライアントをアプリケーションから切断する準備ができたら、`tpterm()` を呼び出します。ただし、マルチコンテキスト・アプリケーションでは、`tpterm()` を呼び出すと現在のコンテキストが破棄されることに注目してください。その場合、現在のコンテキストで動作しているすべてのスレッドが影響を受けます。アプリケーション・プログラマは、`tpterm()` が不意に呼び出されることがないように、複数のスレッドを使用する場合は注意してください。

まだ動作中のスレッドがあるコンテキストでは、`tpterm()` を呼び出さないようにします。そのような状況で `tpterm()` を呼び出すと、そのコンテキストと対応付けられていたほかのスレッドが特別な無効コンテキスト状態になります。無効コンテキスト状態では、大部分の ATMI 関数を使用できなくなります。無効コンテキスト状態からスレッドを解放するには、`tpsetctxt(3c)` または `tpterm()` を呼び出します。良く設計されたアプリケーションでは、無効コンテキスト状態が生じることはありません。

注記 BEA Tuxedo システムでは、COBOL アプリケーションのマルチスレッドはサポートされていません。

コンテキストの切り替え

次は、2 つのコンテキストからサービスを呼び出すクライアントで行われる処理の手順をまとめたものです。

1. TUXCONFIG 環境変数に firstapp で必要な値を設定します。
2. TPMULTICONTEXTS フラグを設定して tpinit() を呼び出し、最初のアプリケーションに参加します。
3. tpgetctxt(3c) を呼び出して、現在のコンテキストへのハンドルを取得します。
4. tuxputenv() を呼び出して、TUXCONFIG 環境変数の値を secondapp コンテキストに必要な値に切り替えます。
5. TPMULTICONTEXTS フラグを設定して tpinit() を呼び出して、2 番目のアプリケーションに参加します。
6. tpgetctxt(3c) を呼び出して、現在のコンテキストへのハンドルを取得します。
7. tpsetctxt(3c) を呼び出して、firstapp コンテキストからコンテキストの切り替えを開始します。
8. firstapp サービスを呼び出します。
9. tpsetctxt(3c) を呼び出してクライアントを secondapp コンテキストに切り替え、secondapp サービスを呼び出します。
10. tpsetctxt(3c) を呼び出してクライアントを firstapp コンテキストに切り替え、firstapp サービスを呼び出します。
11. tpterm() を呼び出して、firstapp コンテキストを終了します。
12. tpsetctxt(3c) を呼び出してクライアントを secondapp コンテキストに切り替え、secondapp サービスを呼び出します。
13. tpterm() を呼び出して、secondapp コンテキストを終了します。

次のコード例は、この手順を示しています。

注記 コードを簡単にするために、エラー・チェックは省略してあります。

リスト 10-2 クライアントでのコンテキストの切り替え

```
#include <stdio.h>
#include "atmi.h"/* BEA Tuxedo ヘッダ・ファイル */

#if defined(__STDC__) || defined(__cplusplus)
main(int argc, char *argv[])
#else
main(argc, argv)
```

```
int argc;
char *argv[];
#endif
{

   TPINIT   *   tpinitbuf;
   TPCONTEXT_T firstapp_contextID, secondapp_contextID;
   /* TUXCONFIG が /home/firstapp/TUXCONFIG に設定されていることを前提とします。*/
   /*
    * BEA Tuxedo システムにマルチコンテキスト・モードで接続します。
    */
   tpinitbuf=tpalloc(TPINIT, NULL, TPINITNEED(0));
   tpinitbuf->flags = TPMULTICONTEXTS;

   if (tpinit((TPINIT *) tpinitbuf) == -1) {
      (void) fprintf(stderr, "Tpinit failed\n");
      exit(1);
}

   /*
    * 現在のコンテキストへのハンドルを取得します。
    */
   tpgetctxt(&firstapp_contextID, 0);

   /*
    * tuxputenv を使用して TUXCONFIG の値を変更し、
    * 別のアプリケーションに参加 (tpinit) します。
    */
   tuxputenv("TUXCONFIG=/home/second_app/TUXCONFIG");

   /*
    * secondapp に参加 (tpinit) します。
    */
   if (tpinit((TPINIT *) tpinitbuf) == -1) {
      (void) fprintf(stderr, "Tpinit failed\n");
      exit(1);
}

   /*
    * secondapp のコンテキストへのハンドルを取得します。
    */
   tpgetctxt(&secondapp_contextID, 0);

   /*
    * tpgetctxt から取得したハンドルと tpsetctxt を使用して、
```

```
     * 2 つのコンテキスト間で切り替えができます。firstapp から開始します。
     */

    tpsetctxt(firstapp_contextID, 0);

    /*
     * firstapp で提供されるサービスを呼び出し、次に secondapp に切り替えます。
     */

    tpsetctxt(secondapp_contextID, 0);

    /*
     * secondapp で提供されるサービスを呼び出します。
     * 次に firstapp に戻ります。
     */

    tpsetctxt(firstapp_contextID, 0);

    /*
     * firstapp で提供されるサービスを呼び出します。操作が終了したら
     * firstapp のコンテキストを終了します。
     */

    tpterm();

    /*
     * secondapp に戻ります。
     */

    tpsetctxt(secondapp_contextID, 0);
    /*
     * secondapp で提供されるサービスを呼び出します。操作が終了したら
     * secondapp のコンテキストを終了し、プログラムを終了します。
     */

    tpterm();

    return(0);
}
```

任意通知型メッセージの処理

任意通知型メッセージを処理する各コンテキストでは、任意通知型メッセージ・ハンドラを設定するか、またはプロセス・ハンドラのデフォルトが設定されている場合はそれを使用する必要があります。

tpsetunsol() がコンテキストに対応付けされていないスレッドから呼び出されると、新しく生成されるすべての tpinit() コンテキストに対して、プロセス単位のデフォルトの任意通知型メッセージ・ハンドラが作成されます。特定のコンテキストは、コンテキストがアクティブのときに tpsetunsol() を再度呼び出して、そのコンテキストの任意通知型メッセージ・ハンドラを変更することができます。プロセス単位のデフォルトの任意通知型メッセージ・ハンドラは、コンテキストに現在対応付けされていないスレッドで tpsetunsol() を再度呼び出すと、変更できます。

ハンドラの設定は、シングルスレッド・アプリケーションまたはシングルコンテキスト・アプリケーションのハンドラを設定する場合と同じように行います。詳細については、tpsetunsol() を参照してください。

現在処理を行っているコンテキストを識別するには、任意通知型メッセージ・ハンドラ内で tpgetctxt(3c) を使用します。

マルチスレッドおよびマルチコンテキスト・アプリケーションにおけるトランザクションのコーディング規則

トランザクションを使用する場合、コーディングで以下の事柄に注意してください。

- 1 つのコンテキストで所有できるトランザクションは 1 つだけです。
- 各コンテキストで異なるトランザクションを使用できます。
- ある時点で任意のコンテキストに対応付けされているすべてのスレッドは、そのコンテキストの同じトランザクション状態を共有します。
- スレッドを同期して、すべての会話と RPC 呼び出しが完了してから tpcommit() が呼び出されるようにします。
- tpcommit() の呼び出しは、特定のトランザクションの 1 つのスレッドからのみ行うことができます。

関連項目

- 第 10 章の 9 ページ「クライアントでのマルチスレッドとマルチコンテキストの動作」
- 第 10 章の 41 ページ「マルチスレッド・クライアントのコーディング」

サーバでマルチコンテキストとマルチスレッドを使用するためのコーディング

ここでは、次の内容について説明します。

- マルチコンテキスト・サーバのコーディング規則
- サーバおよびサーバ・スレッドの初期化と終了
- スレッドを生成するためのサーバのプログラミング
- マルチコンテキスト・サーバでアプリケーション・スレッドを生成するためのコード例

注記 この節で示す手順とコード例は、BEA Tuxedo システムで提供される C 言語のライブラリ関数を参照します。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。COBOL アプリケーションではマルチコンテキスト・サーバの生成に必要なマルチスレッドがサポートされていないので、C 言語の関数に相当する COBOL ルーチンは利用できません。

コンテキストの属性

コンテキストを使用する場合、コーディングで以下の事柄に注意してください。

- 元のディスパッチ・スレッドが終了する前に、アプリケーション生成のサーバ・スレッドがコンテキストを変更せずに終了すると、`tpreturn()` または `tpforward()` が失敗します。スレッドを終了しても、自動的に `tpsetctxt(3c)` が呼び出されてコンテキストが `TPNULLCONTEXT` に変更されることはありません。
- プロセス内のすべてのコンテキストに、同じバッファ・タイプ・スイッチを使用します。

- ほかのタイプのデータ構造体と同じように、マルチスレッド・アプリケーションでは BEA Tuxedo バッファを適切に使用する必要があります。つまり、次のいずれかが該当する場合、バッファを 2 つの呼び出しで同時に使用することはできません。
  - 両方の呼び出しでバッファを使用する場合
  - 両方の呼び出しでバッファを解放する場合
  - 1 つの呼び出しでバッファを使用し、もう 1 つの呼び出しでバッファを解放する場合

マルチコンテキスト・サーバのコーディング規則

マルチコンテキスト・サーバ使用する場合、コーディングで以下の規則に注意してください。

- サーバ上の BEA Tuxedo ディスパッチャは、同じサービスまたは異なるサービスを複数回ディスパッチでき、ディスパッチされるサービスごとに異なるディスパッチ・コンテキストを生成します。
- サーバは、`tpinit()` を呼び出したり、クライアントとして動作することはできません。サーバ・プロセスで `tpinit()` が呼び出されると、`tpinit()` は `-1` を返して `tperrno(5)` に `TPEPROTO` を設定します。アプリケーション生成のサーバ・スレッドは、`tpsetctxt(3c)` を呼び出す前に ATMI を呼び出すことはできません。
- サーバ・ディスパッチ・スレッドだけが `tpreturn()` または `tpforward()` を呼び出すことができます。
- アプリケーション生成のスレッドが任意のアプリケーションのコンテキストに対応付けされている場合、サーバは `tpreturn()` または `tpforward()` を実行できません。そのため、サーバ・ディスパッチ・スレッドが `tpreturn()` を呼び出す前に、そのコンテキストに対応するアプリケーション生成の各スレッドがコンテキストを `TPNULLCONTEXT` または別の有効なコンテキストに設定して `tpsetctxt(3c)` を呼び出す必要があります。

  この規則に違反すると、`tpreturn()` または `tpforward()` はユーザ・ログにメッセージを書き込み、呼び出し元に `TPESVCERR` を示して、メイン・サーバのディスパッチ・ループに制御を戻します。無効 `tpreturn()` が実行されたコンテキスト内のスレッドは、無効コンテキスト状態になります。
- `tpreturn()` または `tpforward()` が呼び出された時点で未終了の ATMI 呼び出し、RPC 呼び出し、または会話があると、`tpreturn()` または `tpforward()` はユーザ・ログにメッセージを書き込み、呼び出し元に `TPESVCERR` を示して、メイン・サーバのディスパッチ・ループに制御を戻します。

- サーバ・デスパッチ・スレッドで tpsetctxt(3c) を呼び出すことはできません。
- シングルコンテキストのサーバとは異なり、マルチコンテキスト・サーバ・スレッドでは、同じサーバ・プロセスだけで提供されるサービスを呼び出すことができます。

サーバおよびサーバ・スレッドの初期化と終了

サーバとサーバ・スレッドの初期化と終了には、BEA Tuxedo システムで提供されるデフォルトの関数や独自の関数を使用できます。

表 10-1 初期化と終了を行うデフォルトの関数

目的	使用するデフォルトの関数
サーバの初期化	tpsvrinit(3c)
サーバ・スレッドの初期化	tpsvrthrinit(3c)
サーバの終了	tpsvrdone(3c)
サーバ・スレッドの終了	tpsvrthrdone(3c)

スレッドを生成するためのサーバのプログラミング

マルチコンテキスト・サーバを使用するほとんどのアプリケーションでは、システム生成のディスパッチ・サーバ・スレッドだけが使用されます。ただし、アプリケーション・サーバに新しいスレッドを生成することもできます。この節では、その方法について説明します。

スレッドの生成

オペレーティング・システムのスレッド関数を使用して、アプリケーション・サーバに新しいスレッドを生成できます。これらの新しいスレッドは、BEA Tuxedo システムから独立して動作できます。また、いずれかのサーバ・ディスパッチ・スレッドと同じコンテキストで動作することもできます。

コンテキストへのスレッドの対応付け

アプリケーション生成のサーバ・スレッドは、当初どのサーバ・ディスパッチ・コンテキストにも対応付けられていません。ただし、初期化される前に呼び出された場合、ほとんどの ATMI 関数は暗黙的に tpinit() を実行します。サーバで tpinit() を呼び出すことは禁止されているので、そのような呼び出しを行うと問題が発生します。サーバ・プロセスが tpinit() を呼び出すと、tpinit() は -1 を返して tperrno(5) に TPEPROTO を設定します。

そのため、アプリケーション生成のサーバ・スレッドは、既存のコンテキストと対応付けを行ってから ATMI 関数を呼び出す必要があります。アプリケーション生成のサーバ・スレッドを既存のコンテキストに対応付けるには、以下の手順をコーディングします。

1. サーバ・ディスパッチ・スレッド A は、tpgetctxt(3c) を呼び出して現在のコンテキストへのハンドルを取得します。
2. サーバ・ディスパッチ・スレッド A は、tpgetctxt(3c) が返すハンドルをアプリケーション・スレッド B に渡します。
3. アプリケーション・スレッド B は、tpsetctxt(3c) を呼び出してサーバ・ディスパッチ・スレッド A から受け取ったハンドルを指定して、現在のコンテキストとの対応付けを作成します。
4. アプリケーション生成のサーバ・スレッドは、tpreturn() または tpforward() を呼び出すことはできません。元のディスパッチ・スレッドが tpreturn() または tpforward() を呼び出す前に、そのコンテキストにあったすべてのアプリケーション生成のサーバ・スレッドは TPNULLCONTEXT または別の有効なコンテキストに切り替える必要があります。

   この規則に違反すると、tpforward() または tpreturn() が失敗し、呼び出し元にサービス・エラーが示されます。

マルチコンテキスト・サーバでアプリケーション・スレッドを生成するためのコード例

次のコード例は、サービスが別のスレッドを生成してそのサービスの作業を行うマルチコンテキスト・サーバを示しています。このコードは、サーバでアプリケーション・スレッドを生成する必要があるアプリケーションで使用します。オペレーティング・システムのスレッド関数は、オペレーティング・システムによって異なります。このコード例では、POSIX 関数と ATMI 関数が使用されています。

注記 コードを簡単にするために、エラー・チェックは省略してあります。また、BEA Tuxed システムによってディスパッチされたスレッドだけを使用するマルチコンテキスト・サーバも省略してあります。そのようなサーバのコーディングは、スレッド・セーフのプログラミング方法が使用されている場合、シングルコンテキスト・サーバのコーディングとまったく同じです。

リスト 10-3 マルチコンテキスト・サーバでのスレッドの生成

```
#include <pthread.h>
#include <atmi.h>

void *withdrawalthread(void *);

struct sdata {
             TPCONTEXT_T  ctxt;
             TPSVCINFO     *svcinfoptr;
};

void
TRANSFER(TPSVCINFO *svcinfo)
{
    struct sdata     transferdata;
    pthread_t        withdrawalthreadid;

    tpgetctxt(&transferdata.ctxt, 0);
    transferdata.svcinfoptr = svcinfo;
    pthread_create(&withdrawalthreadid, NULL, withdrawalthread, &transferdata);
    tpcall("DEPOSIT", ...);
    pthread_join(withdrawalthreadid, NULL);
    tpreturn(TPSUCCESS, ...);
}


void *
withdrawalthread(void *arg)
```

```
{
    tpsetctxt(arg->ctxt, 0);
    tpopen();
    tpcall("WITHDRAWAL", ...);
    tpclose();
    return(NULL);
}
```

このコードでは、元のディスパッチ・スレッドで DEPOSIT サービスを呼び出し、アプリケーション生成のスレッドで WITHDRAWAL を呼び出して、口座振り替えを行っています。この例では、リソース・マネージャで混在モデルがサポートされていることを前提としています。つまり、サーバのすべてのスレッドが特定のインスタンスと対応付けされていなくても、そのサーバの複数のスレッドが特定のデータベース接続と対応付けられます。ただし、そのようなモデルがサポートされたリソース・マネージャはほとんどありません。

アプリケーション生成のスレッドを使用しないようにすると、このコードはさらに簡単になります。このコード例で tpcall() を 2 回呼び出して行っている並列処理を実現するには、サーバ・ディスパッチ・スレッド内で tpacall() と tpgetrply() をそれぞれ 2 回呼び出します。

関連項目

- 第 10 章の 15 ページ「サーバでのマルチスレッドとマルチコンテキストの動作」

マルチスレッド・クライアントのコーディング

ここでは、次の内容について説明します。

- マルチスレッド・クライアントのコーディング規則
- クライアントの複数のコンテキストへの初期化
- マルチスレッド環境での応答の取得
- マルチスレッド・マルチコンテキスト環境の環境変数
- マルチスレッド・クライアントでのコンテキスト単位の関数とデータ構造体
- マルチスレッド・クライアントでのプロセス単位の関数とデータ構造体
- マルチスレッド・クライアントでのスレッド単位の関数とデータ構造体
- マルチスレッド・クライアントのコード例

注記 BEA Tuxedo システムでは、COBOL アプリケーションのマルチスレッドはサポートされていません。

マルチスレッド・クライアントのコーディング規則

マルチスレッド・クライアントを使用する場合、コーディングで以下の規則に注意してください。

- 会話が開始されると、同じプロセス内のすべてのスレッドでその会話を操作できます。ハンドルと呼び出し記述子は、同じプロセスの同じコンテキスト内で移植できます。ただし、ほかのコンテキストやプロセスには移植できません。ハンドルと呼び出し記述子は、それらが元々割り当てられていたアプリケーション・コンテキストだけで使用できます。
- 同じプロセスの同じコンテキストで動作するスレッドの場合、そのスレッドが `tpacall()` の呼び出し元かどうかに関係なく、`tpgetrply()` を呼び出して以前の `tpacall()` 呼び出しの応答を受け取ることができます。
- トランザクションをコミットまたはアボートできるのは、1 つのスレッドだけです。トランザクションを開始したスレッドである必要はありません。
- すべての RPC 呼び出しとすべての会話は、トランザクションをコミットする前に完了していなければなりません。未終了の RPC 呼び出しや会話があるときに `tpcommit()` が呼び出されると、`tpcommit()` はトランザクションをアボートし、`-1` を返して `tperrno(5)` に `TPEABORT` を設定します。

- トランザクションがまだコミットまたはアボートしていないことを確認できない場合に、tpcall()、tpacall()、tpgetrply()、tpconnect()、tpsend()、tprecv()、tpdiscon() などの関数をトランザクション・モードで呼び出すことはできません。
- 同じコンテキストに 2 つの tpbegin() 呼び出しを同時に行うことはできません。
- 既にトランザクション・モードにあるコンテキストに tpbegin() を呼び出すことはできません。
- クライアントを使用している場合に複数のドメインに接続するときは、TUXCONFIG または WSNADDR の値を手動で変更してから tpinit() を呼び出します。このような処理が複数のスレッドで行われる場合、環境変数の設定と tpinit() 呼び出しを同期する必要があります。クライアントのすべてのアプリケーション対応付けは、以下の規則に従う必要があります。
  - すべての対応付けは、同じリリースの BEA Tuxedo システムに生成します。
  - 特定のクライアントのすべてのアプリケーション対応付けはネイティブ・クライアントとして生成するか、またはワークステーション・クライアントとして生成します。
- アプリケーションに参加するには、マルチスレッドのワークステーション・クライアントが、シングルコンテキスト・モードで実行している場合でも、常に TPMULTICONTEXTS フラグを設定して tpinit() 関数を呼び出す必要があります。

クライアントの複数のコンテキストへの初期化

クライアントを複数のコンテキストに参加させるには、TPINIT データ構造体の *flags* に TPMULTICONTEXTS フラグを設定して tpinit() 関数を呼び出します。

1 つのプロセスでは、tpinit() へのすべての呼び出しに TPMULTICONTEXTS フラグを含めます。または、tpinit() へのすべての呼び出しにこのフラグを含めません。この規則には、例外が 1 つあります。つまり、tpterm() への呼び出しが正常に終了して、クライアントのすべてのアプリケーション対応付けが終了した場合、次に tpinit() を呼び出すときに必ずしも TPMULTICONTEXTS フラグを含む必要のない状態にプロセスが復元されます。

TPMULTICONTEXTS フラグを設定して tpinit() 関数が呼び出されると、アプリケーションとの新しい対応付けが生成され、スレッドに対するカレントの対応付けが指定されます。新しい対応付けが生成される BEA Tuxedo ドメインは、TUXCONFIG または WSENVFILE/WSNADDR 環境変数の値で決定されます。

クライアント・スレッドが TPMULTICONTEXTS フラグを設定せずに tpinit() を正常に実行した場合、クライアントのすべてのスレッドがシングルコンテキスト状態 (TPSINGLECONTEXT) になります。

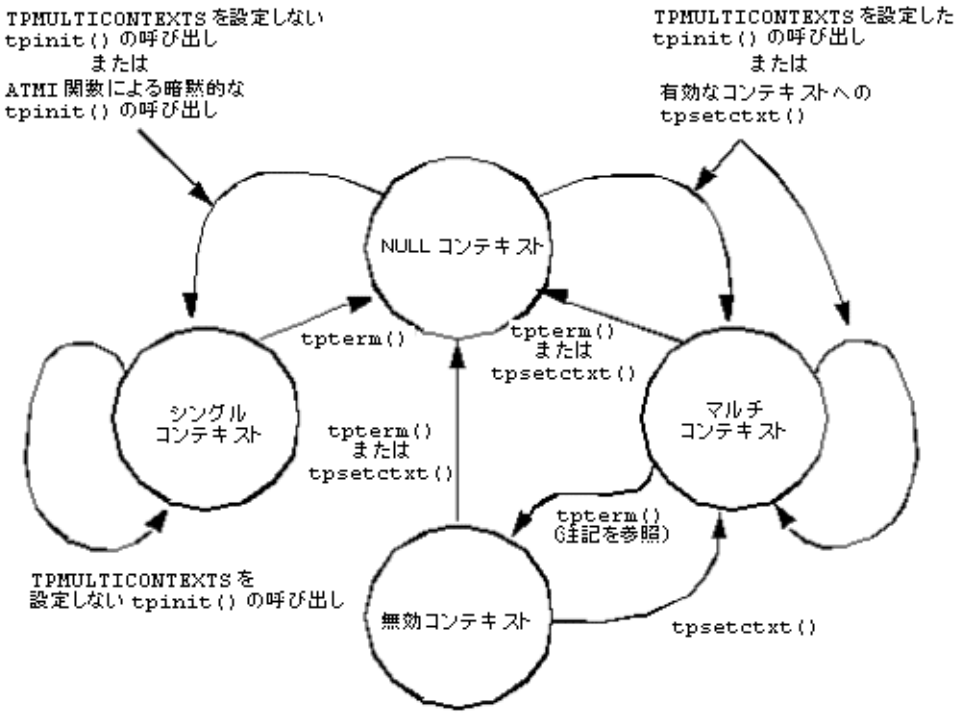
tpinit() が失敗した場合、呼び出し元スレッドは元のコンテキスト、つまり tpinit() 呼び出しの前に操作していたコンテキスト状態のままになります。

まだ動作中のスレッドがあるコンテキストから tpterm() を呼び出すことはできません。このような状況やそれ以外の状況で tpterm() を呼び出した結果生じるコンテキスト状態については、第 10 章の 44 ページ「マルチコンテキスト状態の遷移」を参照してください。

クライアント・スレッドのコンテキスト状態の変化

マルチコンテキストのアプリケーションでは、いろいろな関数を呼び出すと、呼び出し元スレッド、および呼び出し元プロセスと同じコンテキストでアクティブなその他のスレッドのコンテキスト状態が変化します。次の図は、tpinit()、tpsetctxt(3c)、および tpterm() を呼び出した結果、変化したコンテキスト状態を示しています。tpgetctxt(3c) 関数を呼び出しても、コンテキスト状態は変化しません。

図 10-4　マルチコンテキスト状態の遷移



注記　tpterm() がマルチコンテキスト状態 (TPMULTICONTEXTS) で実行しているスレッドによって呼び出されると、呼び出し元スレッドはヌル・コンテキスト状態 (TPNULLCONTEXT) になります。終了するコンテキストに関連するその他すべてのスレッドは、無効コンテキスト状態 (TPINVALIDCONTEXT) に切り替わります。

次の表は、tpinit()、tpsetctxt(3c)、および tpterm() を呼び出した場合のコンテキスト状態の変化を示しています。

表 10-2 クライアント・スレッドのコンテキスト状態の変化

	実行後のスレッドのコンテキスト状態			
実行する関数	NULL **コンテキスト**	**シングルコンテキスト**	**マルチコンテキスト**	**無効コンテキスト**
TPMULTICONTEXTS が設定されていない tpinit()	シングルコンテキスト	シングルコンテキスト	エラー	エラー
TPMULTICONTEXTS が設定された tpinit()	マルチコンテキスト	エラー	マルチコンテキスト	エラー
TPNULLCONTEXT への tpsetctxt(3c)	NULL	エラー	NULL	NULL
コンテキスト 0 への tpsetctxt(3c)	エラー	シングルコンテキスト	エラー	エラー
コンテキスト > 0 への tpsetctxt(3c)	マルチコンテキスト	エラー	マルチコンテキスト	マルチコンテキスト
暗黙的な tpinit()	シングルコンテキスト	該当せず	該当せず	エラー
このスレッドでの tpterm()	NULL	NULL	NULL	NULL
このコンテキストの異なるスレッドでの tpterm()	該当せず	NULL	無効	該当せず

マルチスレッド環境での応答の取得

tpgetrply() は、tpacall() からの要求に対する応答だけを受け取ります。tpcall() からの要求は、マルチスレッドまたはマルチコンテキストのレベルに関係なく、tpgetrply() で取得することはできません。

tpgetrply() は、1 つのコンテキスト、つまり tpgetrply() の呼び出し元コンテキストだけで動作します。そのため、TPGETANY フラグを設定して tpgetrply() を呼び出すと、同じコンテキストで生成されたハンドルだけが考慮されます。同じように、あるコンテキストで生成されたハンドルを別のコンテキストで使用することはできません。ただし、同じコンテキストで動作するスレッドには、そのハンドルを使用できます。

tpgetrply() をマルチスレッド環境で呼び出す場合、以下の制約があります。

- 1 つのスレッドが既に tpgetrply() で特定のハンドルを待機しているときに、同じコンテキストの別のスレッドが tpgetrply() を呼び出して同じハンドルの取得を試みると、tpgetrply() は -1 を返し、tperrno に TPEPROTO を設定します。
- 1 つのスレッドが既に TPGETANY フラグを設定して tpgetrply() の応答を待機しているときに、同じコンテキストの別のスレッドが tpgetrply() を呼び出して特定のハンドルの取得を試みると、tpgetrply() は -1 を返し、tperrno(5) に TPEPROTO を設定します。

  これは、1 つのスレッドが既に tpgetrply() で特定のハンドルを待機しているときに、同じコンテキストの別のスレッドが TPGETANY フラグを設定して tpgetrply() を呼び出した場合も同じです。これらの制約により、特定のハンドルを待機しているスレッドがある場合、その応答が別のスレッドに渡されることがなくなります。
- ある時点で、TPGETANY フラグを設定して tpgetrply() の応答を待機できるのは、特定のコンテキスト内で 1 つのスレッドだけです。TPGETANY フラグを設定して呼び出した tpgetrply() がまだ処理されていない場合に、同じコンテキストの別のスレッドが同じ呼び出しを行うと、この 2 番目の呼び出しは -1 を返し、tperrno(5) に TPEPROTO を設定します。

マルチスレッド・マルチコンテキスト環境の環境変数

BEA Tuxedo アプリケーションをマルチコンテキスト・マルチスレッド環境で実行する場合、環境変数に関して以下の事柄に注意してください。

- プロセスは、初期状態ではその環境をオペレーティング・システム環境から継承します。環境変数がサポートされているプラットフォームでは、そのような変数はプロセス単位で動作するエンティティを構成します。そのため、コンテキスト単位の環境設定に依存するアプリケーションでは、オペレーティング・システム関数ではなく `tuxgetenv(3c)` 関数を使用する必要があります。

  注記 オペレーティング・システム環境が認識されないオペレーティング・システムの場合、初期状態では空の環境になっています。

- 多くの環境変数は、BEA Tuxedo システムでプロセスごとに 1 回、またはコンテキストごとに 1 回だけ読み取られ、BEA Tuxedo システム内にキャッシュされます。プロセスに一度キャッシュされた変数を変更しても影響はありません。

キャッシュの実行単位	環境変数
コンテキスト単位	TUXCONFIG
	FIELDTBLS と FIELDTBLS32
	FLDTBLDIR と FLDTBLDIR32
	ULOGPFX
	VIEWDIR と VIEWDIR32
	VIEWFILES と VIEWFILES32
	WSNADDR
	WSDEVICE
	WSENV
プロセス単位	TMTRACE
	TUXDIR
	ULOGDEBUG

- tuxputenv(3c) 関数はプロセス全体の環境に影響します。
- tuxreadenv(3c) 関数を呼び出すと、環境変数を含むファイルが読み取られ、それらの変数がプロセス全体の環境に追加されます。
- tuxgetenv(3c) 関数は、現在のコンテキストで要求された環境変数の現在の値を返します。初期設定では、すべてのコンテキストが同じ環境になります。ただし、特定のコンテキストに固有の環境ファイルを使用すると、コンテキストごとに異なる環境設定を持つことができます。
- クライアントが複数のドメインに初期化する場合、tpinit() を呼び出す前に、毎回 TUXCONFIG、WSNADDR、または WSENVFILE 環境変数の値を適切な値に変更する必要があります。そのようなアプリケーションがマルチスレッドの場合、以下の処理を確実に行うために、ミューテックスなどのアプリケーション定義の同時実行制御が必要になります。
  - 適切な環境変数が再設定されること
  - ほかのスレッドによって環境変数が再設定されずに tpinit() が呼び出されること
- クライアントがシステムに初期化する場合、WSENVFILE やマシン環境ファイルが読み取られ、そのコンテキストの環境だけが影響を受けます。環境ファイルで上書きされないコンテキスト部分には、プロセス全体に対する以前の環境が適用されます。

マルチスレッド・クライアントでのコンテキスト単位の関数とデータ構造体

以下に示す ATMI 関数は、呼び出し元のアプリケーション・コンテキストだけに影響します。

- tpabort()
- tpacall()
- tpadmcall(3c)
- tpbegin()
- tpbroadcast()
- tpcall()
- tpcancel()
- tpchkauth()
- tpchkunsol()

- tpclose(3c)
- tpcommit()
- tpconnect()
- tpdequeue(3c)
- tpdiscon()
- tpenqueue(3c)
- tpforward()
- tpgetlev()
- tpgetrply()
- tpinit()
- tpnotify()
- tpopen(3c)
- tppost()
- tprecv()
- tpresume()
- tpreturn()
- tpscmt(3c)
- tpsend()
- tpservice(3c)
- tpsetunsol()
- tpsubscribe()
- tpsuspend()
- tpterm()
- tpunsubscribe()
- tx_begin(3c)
- tx_close(3c)
- tx_commit(3c)
- tx_info(3c)
- tx_open(3c)

- tx_rollback(3c)
- tx_set_commit_return(3c)
- tx_set_transaction_control(3c)
- tx_set_transaction_timeout(3c)
- userlog(3c)

注記 tpbroadcast() の場合、ブロードキャスト・メッセージは特定のアプリケーションとの対応付けから送られたものとして識別されます。tpnotify(3c) の場合、通知は特定のアプリケーションとの対応付けから送られたものとして識別されます。tpinit() の注記については、「マルチスレッド・クライアントでのプロセス単位の関数とデータ構造体」を参照してください。

tpsetunsol() がコンテキストに対応付けされていないスレッドから呼び出されると、新しく生成されるすべての tpinit() コンテキストに対して、プロセス単位のデフォルトの任意通知型メッセージ・ハンドラが作成されます。特定のコンテキストは、コンテキストがアクティブのときに tpsetunsol() を再度呼び出して、そのコンテキストの任意通知型メッセージ・ハンドラを変更することができます。プロセス単位のデフォルトの任意通知型メッセージ・ハンドラは、コンテキストに現在対応付けされていないスレッドで tpsetunsol() を再度呼び出すと、変更できます。

- CLIENTID、クライアント名、ユーザ名、トランザクション ID、および TPSVCINFO データ構造体の内容は、同じプロセス内のコンテキストによって異なる場合があります。
- 非同期呼び出しハンドルと接続記述子は、その生成元コンテキスト内で有効です。任意通知のタイプは、コンテキストごとに固有です。シグナル・ベースの通知はマルチコンテキストでは使用できませんが、各コンテキストでは次のいずれかのオプションを使用できます。
  - 任意通知型メッセージの無視
  - ディップ・イン通知
  - 専用のスレッド通知

マルチスレッド・クライアントでのプロセス単位の関数とデータ構造体

以下に示す BEA Tuxedo 関数は、呼び出し元のプロセス全体に影響します。

- tpadvertise()
- tpalloc()
- tpconvert(3c)— 要求された構造体が変換されます。ただし、プロセスのサブセットだけに対応します。
- tpfree()
- tpinit()—プロセス単位の TPMULTICONTEXTS モードまたはシングルコンテキスト・モードに応じて適用されます。第 10 章の 48 ページ「マルチスレッド・クライアントでのコンテキスト単位の関数とデータ構造体」も参照してください。
- tprealloc()
- tpsvrdone()
- tpsvrinit()
- tptypes()
- tpunadvertise()
- tuxgetenv(3c)— オペレーティング・システム環境がプロセス単位の場合
- tuxputenv(3c)— オペレーティング・システム環境がプロセス単位の場合
- tuxreadenv(3c)— オペレーティング・システム環境がプロセス単位の場合
- Usignal(3c)

シングルコンテキスト・モード、マルチコンテキスト・モード、または非初期化モードのどれを使用するかは、プロセス全体に影響します。また、バッファ・タイプ・スイッチ、ビュー・キャッシュ、および環境変数の値も、プロセス単位の関数です。

マルチスレッド・クライアントでのスレッド単位の関数とデータ構造体

以下に示す関数は、呼び出し元のスレッドだけに影響します。

- CATCH
- tperrordetail(3c)
- tpgetctxt(3c)
- tpgprio()
- tpsetctxt(3c)
- tpsprio()
- tpstrerror(3c)
- tpstrerrordetail(3c)
- TRY(3c)
- Uunix_err(3c)

Ferror、Ferror32(5)、tperrno(5)、tpurcode(5)、および Uunix_err 変数は、各スレッドに固有です。

現在のコンテキストの ID は各スレッドに固有です。

マルチスレッド・クライアントのコード例

次のコード例は、ATMI 呼び出しを使用するマルチスレッド・クライアントを示しています。スレッド関数は、オペレーティング・システムによって異なります。この例では、POSIX 関数が使用されています。

注記 コードを簡単にするために、エラー・チェックは省略してあります。

リスト 10-4　マルチスレッド・クライアントのコード例

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

TPINIT   *   tpinitbuf;
int          timeout=60;
```

```
pthread_t    withdrawalthreadid, stockthreadid;
TPCONTEXT_T  ctxt;
void * stackthread(void *);
void * withdrawalthread(void *);

main()
{
tpinitbuf = tpalloc(TPINIT, NULL, TPINITNEED(0));
/*
 * このコードでは、withdrawal スレッドと deposit スレッドという別のスレッドを
 * 使用して振り込みを行っています。また、BEA 株の現在の価格を別のアプリケーションから取得し、
 * 送金した金額で購入できる株数を計算しています。
 */

tpinitbuf->flags = TPMULTICONTEXTS;

/* 残りの tpinitbuf を設定します。 */
tpinit(tpinitbuf);

tpgetctxt(&ctxt, 0);
tpbegin(timeout, 0);
pthread_create(&withdrawalthreadid, NULL, withdrawalthread, NULL);
tpcall("DEPOSIT", ...);

/* withdrawal スレッドの完了を待機します。*/
pthread_join(withdrawalthreadid, NULL);

tpcommit(0);
tpterm();

/* stock スレッドの完了を待機します。*/
pthread_join(stockthreadid, NULL);

/* 結果を出力します。*/
printf("$%9.2f has been transferred \
from your savings account to your checking account.\n", ...);

printf("At the current BEA stock price of $%8.3f, \
you could purchase %d shares.\n", ...);

exit(0);
}



void *
stockthread(void *arg)
{
```

```
     /* ほかのスレッドが tpinit() を呼び出しているので、
      * TUXCONFIG を再設定してもスレッドに影響しません。
      */

     tuxputenv("TUXCONFIG=/home/users/xyz/stockconf");
     tpinitbuf->flags = TPMULTICONTEXTS;
     /* 残りの tpinitbuf を設定します。*/
     tpinit(tpinitbuf);
     tpcall("GETSTOCKPRICE", ...);
     /* main() でもアクセスできる変数に株価を格納します。*/
     tpterm();
     return(NULL);
}



void *
withdrawalthread(void *arg)
{
/* 別のアプリケーションから株価を取得するために新しいスレッドを生成します。*/



     pthread_create(&stockthreadid, NULL, stockthread, NULL);
     tpsetctxt(ctxt, 0);
     tpcall("WITHDRAWAL", ...);
     return(NULL);
}
```

関連項目

- 第 10 章の 9 ページ「クライアントでのマルチスレッドとマルチコンテキストの動作」
- 第 10 章の 25 ページ「マルチスレッドおよびマルチコンテキスト・アプリケーションのプログラミング開始前のガイドライン」
- 第 10 章の 27 ページ「クライアントでマルチコンテキストを使用するためのコーディング」

マルチスレッド・サーバのコーディング

ほとんどの場合、マルチスレッド・サーバはマルチコンテキストでもあります。マルチスレッド・サーバのコーディングについては、第 10 章の 35 ページ「サーバでマルチコンテキストとマルチスレッドを使用するためのコーディング」を参照してください。

マルチスレッドおよびマルチコンテキスト・アプリケーションのコードのコンパイル

buildserver(1) や buildclient(1) など、コンパイルまたはビルドの実行可能ファイル用に BEA Tuxedo システムで提供されるプログラムには、必要なコンパイラ・フラグが自動的に設定されます。これらのツールを使用すると、コンパイル時にフラグを設定する必要がありません。

ただし、最終的なコンパイルの前に .c ファイルを .o ファイルにコンパイルする場合は、プラットフォーム固有のコンパイラ・フラグを設定する必要があります。そのようなフラグは、単一のプロセスにリンクするすべてのコードに一貫して設定しなければなりません。

マルチスレッド・サーバを生成する場合、-t オプションを指定して buildserver(1) コマンドを実行する必要があります。これはマルチスレッド・サーバの場合に必須のオプションです。ビルド時にこのオプションが指定されず、その後、MAXDISPATCHTHREADS の値が 1 を超えるコンフィギュレーション・ファイルを使用して新しいサーバを起動すると、警告メッセージがユーザ・ログに記録され、サーバはシングルスレッドの動作に戻ります。

マルチスレッド環境で .c ファイルを .o ファイルにコンパイルする場合に必要なオペレーティング・システム固有のコンパイラ・パラメータを識別するには、-v オプションを指定して buildclient(1) または buildserver(1) をテスト・ファイルで実行します。

関連項目

- 第 10 章の 27 ページ「クライアントでマルチコンテキストを使用するためのコーディング」
- 第 10 章の 35 ページ「サーバでマルチコンテキストとマルチスレッドを使用するためのコーディング」
- 第 10 章の 41 ページ「マルチスレッド・クライアントのコーディング」

マルチスレッドおよびマルチコンテキスト・アプリケーションのテスト

ここでは、次の内容について説明します。

- マルチスレッドおよびマルチコンテキスト・アプリケーションのテスト時の推奨事項
- マルチスレッドおよびマルチコンテキスト・アプリケーションのトラブル・シューティング
- マルチスレッドおよびマルチコンテキスト・アプリケーションのエラー処理

マルチスレッドおよびマルチコンテキスト・アプリケーションのテスト時の推奨事項

マルチスレッドやマルチコンテキストのコードをテストする場合、以下を行うことをお勧めします。

- マルチプロセッサの使用
- マルチスレッド・デバッガの使用 (オペレーティング・システムのベンダから提供されている場合)
- いろいろなタイミング条件でのストレス・テスト

マルチスレッドおよびマルチコンテキスト・アプリケーションのトラブル・シューティング

エラーの原因を調べる場合、まず TPMULTICONTEXTS フラグが設定されているかどうか、またその設定内容を確認します。このフラグが設定されていないこと、または正しく設定されていないことが原因でよくエラーが起こります。

tpinit( ) の TPMULTICONTEXTS フラグの間違った使用

TPMULTICONTEXTS フラグを使用できない場合にこのフラグがプロセスに設定されているとき、または TPMULTICONTEXTS を設定する必要がある場合に設定されていないとき、tpinit( ) は -1 を返し、tperrno に TPEPROTO を設定します。

TPMULTICONTEXTS が設定されていない場合の tpinit() の呼び出し

TPMULTICONTEXTS が設定されていない場合に tpinit( ) が呼び出されると、この関数はシングルコンテキスト・アプリケーションで呼び出されたときと同じように動作します。tpinit( ) が既に 1 回呼び出されている場合、それ以降の tpinit( ) 呼び出しで TPMULTICONTEXTS フラグが設定されていなくても、この関数は正常に終了します。これは、アプリケーション内の TUXCONFIG または WSNADDR 環境変数の値が変更されている場合にも当てはまります。TPMULTICONTEXTS フラグを設定せずに tpinit( ) を呼び出すことは、マルチコンテキスト・モードではできません。

クライアントがアプリケーションに参加していない場合に、tpinit( ) を呼び出す別の関数の呼び出しの結果として、暗黙的に tpinit( ) が呼び出されると、BEA Tuxedo システムでは TPMULTICONTEXTS フラグが設定されずに tpinit( ) が呼び出されたと解釈されます。これは、以降の tpinit( ) 呼び出しでどのフラグが使用されるかを判断するためです。

ほとんどの ATMI 関数は、既にマルチコンテキスト・モードで動作しているプロセスのコンテキストに対応付けされていないスレッドに呼び出された場合、tperrno(5)=TPEPROTO が設定されて失敗します。

スレッドのスタック・サイズの不足

一部のオペレーティング・システムでは、オペレーティング・システムのデフォルトのスレッド・スタック・サイズが BEA Tuxedo システムで使用するには十分ではありません。Compaq Tru64 UNIX と UnixWare の 2 つのオペレーティング・システムは、サイズが小さいことが認識されています。デフォルトのスレッド・スタック・サイズのパラメータが使用されている場合に、スタックを多用する関数がメイン・スレッド以外のスレッドで呼び出されると、これらのプラットフォーム上のアプリケーションはコア・ダンプします。通常、生成されるコア・ファイルから、スタック・サイズの不足が問題の原因であることはわかりません。

サーバ・ディスパッチ・スレッドやクライアントの任意通知型メッセージ・スレッドなど、BEA Tuxedo システムで独自のスレッドが生成される場合、これらのプラットフォームのデフォルトのスタック・サイズのパラメータを適切な値に調整できます。ただし、アプリケーションで独自のスレッドが生成される場合、アプリケーションで十分なスタック・サイズを指定する必要があります。BEA Tuxedo システムにアクセスするスレッドには、最低 128 K を指定してください。

Compaq Tru64 UNIX および Posix スレッドが使用されるそのほかのシステムでは、スレッドのスタック・サイズは、`pthread_create()` を呼び出す前に `pthread_attr_setstacksize()` を呼び出して指定します。UnixWare では、スレッドのスタック・サイズは `thr_create()` の引数として指定されます。この問題の詳細については、お使いのオペレーティング・システムのマニュアルを参照してください。

マルチスレッドおよびマルチコンテキスト・アプリケーションのエラー処理

エラーはユーザ・ログに記録されます。シングルコンテキスト・モードでもマルチコンテキスト・モードでも、各エラーに対して次の情報が記録されます。

`process_ID.thread_ID.context_ID`

関連項目

- 第 10 章の 9 ページ「クライアントでのマルチスレッドとマルチコンテキストの動作」
- 第 10 章の 15 ページ「サーバでのマルチスレッドとマルチコンテキストの動作」
- 第 10 章の 25 ページ「マルチスレッドおよびマルチコンテキスト・アプリケーションのプログラミング開始前のガイドライン」

11 エラーの管理

ここでは、次の内容について説明します。

- システム・エラー
- アプリケーション・エラー
- エラー処理
- トランザクションについて
- 中央イベント・ログ
- アプリケーション・プロセスのデバッグ
- 全般的なコード例

システム・エラー

BEA Tuxedo システムでは、関数が失敗すると、tperrno(5) 変数を使用してプロセスに情報が提供されます。すべての ATMI 関数は、正常に処理が行われた場合、整数またはポインタを返します。エラーが発生した場合、-1 または NULL を返し、tperrno() にエラーの原因を示す値を設定します。サービス・ルーチンを終了させるために使用する tpreturn() や tpforward() など、呼び出し元に戻らない関数の場合、成功か失敗かを確認する唯一の方法は要求元の tperrno() 変数です。

tperrordetail(3c) と tpstrerrordetail(3c) 関数を使用すると、現在のスレッドへの BEA Tuxedo システムの最新の呼び出しで発生したエラーの詳細を取得できます。tperrordetail() は、そのシンボリック名で表される整数値を返します。この整数値は、tpstrerrordetail() の引数として使用され、エラー・メッセージを表す文字列を指すポインタを取得します。このポインタは、userlog(3c) または fprintf() の引数として使用できます。返されるシンボリック名については、『BEA Tuxedo C リファレンス』の tperrordetail(3c) 参照してください。

tpurcode(5) は、ユーザ定義の条件だけを通知します。tpurcode の値は、tpreturn() の *rcode* 引数の値に設定されます。tpreturn() でエラーまたはトランザクション・タイムアウトが発生しない限り、tpreturn() の *rval* 引数の値に関係なく、tpurcode の値が設定されます。

tperrno(5) に返されるコードは、エラーの種類を示します。次の表は、そのエラーの種類を示しています。

表 11-1 tperrno で示されるエラーの種類

エラーの種類	tperrno の値
アボート	TPEABORT[2]
BEA Tuxedo システム [1]	TPESYSTEM
呼び出し記述子	TPELIMIT と TPEBADDESC
会話	TPEVENT
複製操作	TPEMATCH
一般的な通信	TPESVCFAIL、TPESVCERR、TPEBLOCK、および TPGOTSIG
ヒューリスティックな判断	TPEHAZARD[2] と TPEHEURISTIC[2]
無効な引数 [1]	TPEINVAL
MIB	TPEMIB
エントリなし	TPENOENT
オペレーティング・システム [1]	TPEOS
パーミッション	TPEPERM
プロトコル[1]	TPEPROTO
キューへの登録、取り出し	TPEDIAGNOSTIC
リリース間の互換性	TPERELEASE
リソース・マネージャ	TPERMERR
タイムアウト	TPETIME
トランザクション	TPETRAN[2]
型付きバッファの不一致	TPEITYPE と TPEOTYPE

1. tperrno(5) で返される値によって失敗が通知されるすべての ATMI 関数に適用されます。
2. このエラーの詳細については、第 11 章の 20 ページ「致命的なトランザクション・エラー」を参照してください。

脚注 1 にあるように、tperrno(5) によって通知される 4 種類のエラーは、すべての ATMI 関数で発生するエラーです。それ以外のエラーの種類は、特定の ATMI 関数だけで発生します。以下に、一部のエラーの種類について詳しく説明します。

アボート・エラー

アボートの原因となるエラーについては、第 11 章の 20 ページ「致命的なトランザクション・エラー」を参照してください。

BEA Tuxedo のシステム・エラー

BEA Tuxedo のシステム・エラーは、問題がアプリケーション・レベルではなくシステム・レベルで発生していることを示します。BEA Tuxedo のシステム・エラーが発生すると、エラーの原因を示すメッセージが中央イベント・ログに書き込まれ、tperrno(5) に TPESYSTEM が返されます。詳細については、第 11 章の 30 ページ「中央イベント・ログ」を参照してください。これらのエラーは、アプリケーションではなくシステムで発生するので、エラーの修正についてはシステム管理者に問い合わせてください。

呼び出し記述子のエラー

呼び出し記述子のエラーは、呼び出し記述子の数が上限値を超えている場合、または無効な値を参照している場合に発生します。非同期呼び出しや会話型呼び出しでは、未処理の呼び出し記述子の数が上限値を超えると、TPELIMIT が返されます。操作に対して無効な呼び出し記述子の値が指定されている場合は、TPEBADDESC が返されます。

呼び出し記述子エラーが発生するのは、非同期呼び出しまたは会話型呼び出しを行った場合だけです。同期呼び出しでは、呼び出し記述子は使用されません。非同期呼び出しでは、呼び出し記述子を使用して対応する要求に応答が対応付けられます。会話型送受信用の関数は、呼び出し記述子を使用して接続を識別します。つまり、接続を開始する呼び出しでは、呼び出し記述子を使用できることが大切です。

呼び出し記述子エラーのトラブル・シューティングでは、アプリケーション・レベルで特定のエラーを調べます。

上限値に関するエラー

システムでは、コンテキスト (または BEA Tuxedo アプリケーションへの対応付け) ごとに未処理の呼び出し記述子 (応答) を 50 個まで使用できます。この上限値はシステムで定義されているので、アプリケーションで再定義することはできません。

会話型接続を同時に行う場合の呼び出し記述子に関する制限は、応答時の制限ほど厳しくありません。上限値は、アプリケーション管理者がコンフィギュレーション・ファイルに定義します。アプリケーションが実行中ではない場合、管理者はコンフィギュレーション・ファイルの RESOURCES セクションの MAXCONV パラメータを変更できます。アプリケーションが実行中の場合も、MACHINES セクションは動的に変更できます。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。

無効な記述子によるエラー

呼び出し記述子は無効になることがあります。無効な記述子が参照されると、次の場合に、tperrno(5) にエラーが返されます。

- 呼び出し記述子を使用してメッセージを取得したが、それがエラー・メッセージの場合 (TPEBADDESC)
- 無効になった呼び出し記述子の再利用を試みた場合 (TPEBADDESC)

呼び出し記述子が無効になるのは、以下のような場合です。

- アプリケーションで tpabort() または tpcommit() を呼び出したとき、TPNOTRAN フラグを設定せずに送信されたトランザクション応答が取得されずに残っている場合。
- トランザクションがタイムアウトになった場合。tpgetrply() の呼び出しでタイムアウトが通知された場合、指定された記述子を使用してメッセージを取得することはできず、記述子は無効になります。

会話に関するエラー

会話型サービスで不明な記述子が指定されると、tpsend()、tprecv()、および tpdiscon() 関数は TPEBADDESC を返します。

会話型接続の確立後に tpsend() と tprecv() が TPEEVENT エラーで失敗した場合、イベントが発生します。tpsend() でデータを送信できるかどうかは、発生したイベントによって決まります。システムは、関数呼び出しに渡される *revent* パラメータに TPEEVENT を返します。行われる処理は、発生したイベントによって異なります。

会話型イベントの詳細については、第 7 章の 12 ページ「会話型通信イベント」を参照してください。

複製オブジェクトに関するエラー

処理の結果として複製オブジェクトが生成されるような操作が試みられると、tperrno(5) に TPEMATCH エラー・コードが返されます。次の表は、TPEMATCH エラー・コードを返す関数とその原因を示しています。

関数	原因
tpadvertise	指定された *svcname* は、既にサーバに対して宣言されています。ただし、その処理は *func* 以外の関数で行われています。この関数は失敗しても、*svcname* は現在の関数で宣言されたままになります。つまり、*func* は現在の関数名を置き換えません。
tpresume	*tranid* が、別のプロセスが既に再開したトランザクション識別子を指しています。その場合、呼び出し元のトランザクションの状態は変化しません。
tpsubscribe	指定されたサブスクリプション情報は、既にイベント・ブローカに登録されています。

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

一般的な通信呼び出しのエラー

一般的な通信呼び出しのエラーは、呼び出しが同期または非同期で行われたかどうかに関係なく、どのような通信呼び出しでも発生する可能性があります。tperrno(5) には、TPESVCFAIL、TPESVCERR、TPEBLOCK、または TPGOTSIG エラーが返されます。

TPESVCFAIL および TPESVCERR エラー

tpcall() または tpgetrply() を呼び出した結果、通信の応答部分が失敗すると、tperrno(5) に TPESVCERR または TPSEVCFAIL が返されます。tpreturn() に渡された引数でエラーが判別され、この関数で実行する処理が決定されます。

引数の処理中に tpreturn() でエラーが発生すると、システムはエラーを元の要求元に返し、tperrno(5) に TPESVCERR を設定します。受信側では、tperrno() の値を調べてエラーの発生を確認します。システムでは、tpreturn() 関数からのデータ送信は行われず、tpgetrply() でエラーが発生した場合は、呼び出し記述子が無効なものと見なされます。

tpreturn() で TPESVCERR エラーが発生していない場合、*rval* に返された値で呼び出しが成功したか失敗したかを判断できます。アプリケーションで *rval* パラメータに TPFAIL が指定されると、システムは tperrno(5) に TPESVCFAIL を返し、呼び出し元にデータ・メッセージを送信します。*rval* に TPSUCCESS が設定されると、呼び出し元に制御が正常に戻り、tperrno() は設定されず、呼び出し元がデータを受信します。

TPEBLOCK および TPGOTSIG エラー

TPEBLOCK および TPGOTSIG エラー・コードは、メッセージの要求側に返される場合も応答側に返される場合もあるので、すべての通信呼び出に対して返される可能性があります。

ブロッキング状態が発生している場合に、要求を同期または非同期に送信するプロセスでブロッキング状態を無視するように *flags* パラメータに TPNOBLOCK が設定されていると、システムは TPEBLOCK を返します。たとえば、システムのキューがすべていっぱいになっている場合、要求が送信されるとブロッキング状態になります。

tpcall() がブロッキング状態を示していない場合は、通信の送信部分だけに影響します。要求の送信に成功すると、その呼び出しが応答を待っている間にブロッキング状態が存在したとしても、TPEBLOCK は返されません。

flags に TPNOBLOCK を設定して呼び出しを行った場合、tpgetrply() が応答を待っている間にブロッキング状態が発生すると、tpgetrply() に TPEBLOCK が返されます。この状況は、メッセージがその時点で使用できない場合などに発生します。

TPGOTSIG エラーは、シグナルによってシステム・コールに割り込みが発生したことを示します。このような状況は、実際にはエラーではありません。通信用の関数で *flags* パラメータに TPSIGRSTRT が設定されていると、呼び出しは失敗せず、tperrno(5) に TPGOTSIG エラー・コードは返されません。

無効な引数によるエラー

無効な引数によるエラーは、関数に渡された引数が無効であることを示しています。引数を取る ATMI 関数では、無効な引数が渡されると関数は失敗します。呼び出し元に制御が戻る関数の場合、関数は失敗して、tperrno(5) に TPEINVAL が設定されます。tpreturn() または tpforward() の場合、要求を開始して結果を待っている tpcall() または tpgetrply() に対して、tperrno() に TPESVCERR が設定されます。

関数に有効な引数だけを渡すようにすると、無効な引数によるアプリケーション・レベルでのエラーを修正できます。

MIB エラー

管理要求が失敗すると、tpadmcall(3c) 関数は tperrno(5) に TPEMIB を返します。*outbuf* は更新され、エラーの原因を示す FML32 フィールドが設定されて呼び出し元に返されます。エラーの原因の詳細については、『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 MIB(5) と TM_MIB(5) を参照してください。

エントリがないために発生するエラー

バッファ・タイプを識別するためのデータ構造体やシステム・テーブルにエントリがないと、エラーが発生します。エントリ・タイプのエラーを示す TPENOENT の意味は、そのエラーを返す関数によって異なります。次の表は、このエラーを返す関数とエラーのさまざまな原因を示しています。

表 11-2 エントリがないために発生するエラー

関数	原因
tpalloc()	システムで認識できないバッファ・タイプが要求されています。バッファ・タイプやサブタイプを認識するには、BEA Tuxedo システム・ライブラリに定義されているタイプ・スイッチ・データ構造体にそのエントリがあることが必要です。tuxtypes(5) と typesw(5) の詳細については、『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』を参照してください。 アプリケーション・レベルでは、既知のタイプを参照しなければなりません。参照していない場合は、システム管理者に問い合わせてください。
tpinit()	エントリ用の領域が掲示板に残っていないため、呼び出し元プロセスがアプリケーションに参加できません。システム管理者に問い合わせてください。
tpcall() tpacall()	呼び出し元プロセスが参照しているサービスは、掲示板にエントリがないため、システムに認識されません。アプリケーション・レベルでサービスを正しく参照しなければなりません。正しく参照していない場合は、システム管理者に問い合わせてください。
tpconnect()	指定されたサービスに接続できません。そのようなサービス名が存在していないか、または会話型サービスではありません。
tpgprio()	要求が行われていないにもかかわらず、呼び出し元プロセスが要求の優先度を調べています。これは、アプリケーション・レベルのエラーです。
tpunadvertise()	サービス名の宣言を取り消すことができません。そのサービス名は、呼び出し元プロセスによって現在宣言されていません。

関数	原因
tpenqueue(3c) tpdequeue(3c)	対応付けられている TMQUEUE(5) サーバが使用できないため、キュー・スペースにアクセスできません。詳細については、『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』を参照してください。
tppost() tpsubscribe() tpunsubscribe()	BEA Tuxedo システムのイベント・ブローカにアクセスできません。詳細については、第 8 章の 1 ページ「イベント・ベースのクライアントおよびサーバのコーディング」を参照してください。

オペレーティング・システムのエラー

オペレーティング・システムのエラーは、オペレーティング・システム・コールが失敗したことを示します。tperrno(5) に TPEOS が返されます。UNIX システムの場合、失敗したシステム・コールを識別する整数値がグローバル変数 Uunixerr に返されます。オペレーティング・システム・エラーを修正するには、システム管理者に問い合わせてください。

パーミッション・エラー

呼び出し元プロセスに、アプリケーションに参加するために必要なパーミッションが設定されていない場合、tpinit() 呼び出しは失敗して、tperrno(5) に TPEPERM が返されます。パーミッションは、コンフィギュレーション・ファイルに設定されるもので、アプリケーションには設定されません。このエラーが発生した場合は、アプリケーション管理者に問い合わせて、必要なパーミッションがコンフィギュレーション・ファイルに設定されていることを確認してください。

プロトコル・エラー

ATMI 関数の呼び出しが間違った順序で行われた場合、または間違ったプロセスを使用して行われた場合、プロトコル・エラーが発生します。たとえば、アプリケーションに参加する前に、クライアントがサーバとの通信の開始を試みると、このエラーが発生します。また、イニシエータではなくトランザクションのパーティシパントによって `tpcommit()` が呼び出された場合も、このエラーが発生します。

ATMI 呼び出しを正しい順序で正しく使用すると、アプリケーション・レベルでプロトコル・エラーを修正できます。

プロトコル・エラーの原因を特定するには、次の事柄を確認してください。

- 正しい順序で呼び出しが行われているかどうか
- 正しいプロセスによって呼び出しが行われているかどうか

プロトコル・エラーでは、`tperrno(5)` に `TPEPROTO` 値が返されます。

詳細については、『BEA Tuxedo C リファレンス』の「C 言語アプリケーション・トランザクション・モニタ・インターフェイスについて」を参照してください。

キューに関するエラー

特定のキューへの登録またはキューからの取り出しに失敗した場合、`tpenqueue(3c)` または `tpdequeue(3c)` 関数は `tperrno(5)` に `TPEDIAGNOSTIC` を返します。処理が失敗した原因は、*`ctl`* を介して返される診断値によって判別できます。有効な *`ctl`* フラグについては、『BEA Tuxedo C リファレンス』の `tpenqueue(3c)` または `tpdequeue(3c)` を参照してください。

リリース間の互換性に関するエラー

アプリケーション・ドメインに参加する BEA Tuxedo システムのリリース間で互換性に問題がある場合、BEA Tuxedo システムは tperrno(5) に TPERELEASE を返します。

たとえば、tpnotify(3c) 関数を呼び出す際に、呼び出し元がターゲット・クライアントから承認メッセージを受け取るまでブロックすることを示す TPACK フラグが設定されている場合、ターゲット・クライアントが TPACK 承認プロトコルがサポートされていない旧バージョンの BEA Tuxedo システムを使用していると、TPERELEASE エラーが返されます。

リソース・マネージャ・エラー

リソース・マネージャ・エラーは、tpopen(3c) および tpclose(3c) を呼び出したときに発生し、tperrno(5) に TPERMERR が返されます。リソース・マネージャを正しくオープンできなかった場合、tpopen() でこのエラー・コードが返されます。同じように、リソース・マネージャを正しくクローズできなかった場合、tpclose() でこのエラー・コードが返されます。BEA Tuxedo システムでは、移植性を保つために、この種類のエラーでは詳細な情報は返されません。リソース・マネージャ・エラーの正確な内容を判断するには、リソース・マネージャに問い合わせる必要があります。

タイムアウト・エラー

BEA Tuxedo システムでは、タイムアウト・エラーがサポートされており、アプリケーションがサービス要求またはトランザクションを待つ時間に制限があります。BEA Tuxedo システムでサポートされている設定可能なタイムアウト機構は、ブロッキング・タイムアウトとトランザクション・タイムアウトの 2 種類です。

ブロッキング・タイムアウトは、アプリケーションがサービス要求に対する応答を待つ時間の上限値を指定します。アプリケーション管理者は、コンフィギュレーション・ファイルにシステムのブロッキング・タイムアウトを設定します。

トランザクション・タイムアウトは、トランザクション (サービス要求が行われる場合もあり) の有効期間を定義します。アプリケーションのトランザクション・タイムアウトを定義するには、tpbegin() に *timeout* 引数を渡します。

通信呼び出しでは、ブロッキング・タイムアウトまたはトランザクション・タイムアウトのいずれかが返され、tpcommit() ではトランザクション・タイムアウトだけが返されます。いずれの場合も、トランザクション・モードのプロセスで呼び出しが失敗して TPETIME が返された場合は、トランザクション・タイムアウトが発生しています。

デフォルトでは、プロセスがトランザクション・モードではない場合、ブロッキング・タイムアウトが実行されます。通信呼び出しの *flags* パラメータに TPNOTIME を設定すると、フラグの設定値はブロッキング・タイムアウトだけに適用されます。プロセスがトランザクション・モードの場合はブロッキング・タイムアウトは実行されず、TPNOTIME フラグが設定されていても関係ありません。

プロセスがトランザクション・モードではない場合に、非同期呼び出しでブロッキング・タイムアウトが発生すると、ブロックされている通信呼び出しは失敗します。ただし、呼び出し記述子は有効なままであり、再度呼び出しを行う場合に使用できます。ほかの通信には影響ありません。

トランザクション・タイムアウトが発生すると、非同期トランザクション応答の呼び出し記述子 (TPNOTRAN フラグが指定されていないもの) は無効になり、参照できなくなります。

呼び出しがトランザクション・モードで行われていない場合、または *flags* パラメータに TPNOBLOCK が設定されていない場合、TPETIME は通信呼び出しでブロッキング・タイムアウトが発生したことを示します。

注記 TPNOBLOCK フラグが設定されている場合、ブロッキング状態が存在すると呼び出しは直ちに戻るので、ブロッキング・タイムアウトは発生しません。

タイムアウト・エラーの処理の詳細については、第 11 章の 18 ページ「トランザクションについて」を参照してください。

トランザクション・エラー

トランザクション、および致命的なエラーと致命的ではないエラーについては、第 11 章の 18 ページ「トランザクションについて」を参照してください。

型付きバッファのエラー

プロセスに対する要求または応答が不明なタイプのバッファで送信された場合、型付きバッファのエラーが返されます。要求データ・バッファの送信先のサービスでバッファ・タイプが認識されない場合、tpcall()、tpacall()、および tpconnect() 関数は TPEITYPE を返します。

プロセスで認識されるバッファ・タイプは、コンフィギュレーション・ファイルとプロセスにリンクされている BEA Tuxedo システム・ライブラリの両方で識別されるものです。これらのライブラリは、プロセスで認識される型付きバッファを識別するデータ構造体を定義および初期化します。プロセスごとにライブラリを作成するか、またはバッファ・タイプが定義されたアプリケーション固有のファイルのコピーをアプリケーションで用意することができます。アプリケーションでは、バッファ・タイプ・スイッチと呼ばれるバッファ・タイプ・データ構造体をプロセスごとに設定できます。tuxtypes(5) と typesw(5) の詳細については、『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』を参照してください。

呼び出し元で認識されないか、または使用できないバッファで応答メッセージが送信されると、tpcall()、tpgetrply()、tpdequeue(3c)、および tprecv() 関数は TPEOTYPE を返します。呼び出し元で使用できないバッファの場合、そのバッファ・タイプはタイプ・スイッチに含まれています。ただし、返されたタイプは応答の受信用に割り当てられたタイプと一致せず、また呼び出し元は異なるバッファ・タイプを使用することはできません。呼び出し元は、*flags* に TPNOCHANGE を設定して、このような状況を示します。その場合、厳密なタイプ・チェックが行われ、違反が見つかると TPEOTYPE が返されます。デフォルトでは、緩やかなタイプ・チェックが行われます。その場合、呼び出し元で認識される限り、最初に割り当てられたタイプ以外のバッファ・タイプが返されることもあります。応答の送信では、応答バッファは呼び出し元で認識できるものでなければなりません。また、厳密なタイプ・チェックが指定されている場合は、それに従う必要があります。

アプリケーション・エラー

アプリケーション内では、tpreturn() の *rcode* 引数を使用して、呼び出し元のプログラムにユーザ定義のエラーに関する情報を渡すことができます。また、システムは、tpurcode の値に tpreturn() の *rcode* 引数の値を設定します。
tpreturn(3c) または tpurcode(5) の詳細については、『BEA Tuxedo C リファレンス』と 『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』をそれぞれ参照してください。

エラー処理

アプリケーションのロジックは、戻り値がある呼び出しのエラー条件を調べ、エラー発生時に適切な処理を行うように設計します。特に、次を確認します。

- -1 または NULL 値が返されていないかどうかを確認します。どちらの値が返されるかは、呼び出す関数によって異なります。
- tperrno(5) の値を確認し、その値に応じて適切なアプリケーション・ロジックを実行する switch 文が記述されたコードを呼び出します。

ATMI では、tpstrerrordetail(3c)、tpstrerror(3c)、および Fstrerror、Fstrerror32(3fml) の 3 種類の関数がサポートされており、BEA Tuxedo システムと FML のメッセージ・カタログからエラー・メッセージのテキストを取得します。これらの関数は、対応するエラー・メッセージを指すポインタを返します。このポインタを使用して、userlog(3c) などに、ポインタが指すテキストを送ることができます。tpstrerrordetail(3c) および tpstrerror(3c) の詳細については、『BEA Tuxedo C リファレンス』を参照してください。Fstrerror、Fstrerror32(3fml) の詳細については、『BEA Tuxedo FML リファレンス』を参照してください。

次のコード例は、エラーの一般的な処理方法を示しています。この例では、atmicall() 関数は、一般的な ATMI 呼び出しを表しています。switch 文 (21 行目) の後のコードに注意してください。この例では、アプリケーション定義の戻りコードの解釈に tpurcode を使用する方法が示されています。

リスト 11-1 エラー処理

```
001   #include <stdio.h>
002   #include "atmi.h"
003
004   main()
005
006   {
007   int rtnval;
008
009   if (tpinit((TPINIT *) NULL) == -1)
010      error message, exit program;
011   if (tpbegin(30, 0) == -1)
012      error message, tpterm, exit program;
013
014   allocate any buffers,
015   make atmi calls
016   check return value
017
```

```
018   rtnval = atmicall();
019
020   if (rtnval == -1) {
021       switch(tperrno) {
022       case TPEINVAL:
023           fprintf(stderr, "Invalid arguments were given to atmicall\n");
024           fprintf(stderr, "e.g., service name was null or flags wrong\n");
025           break;
026       case ...:
027           fprintf(stderr, ". . .");
028           break;
029
030   Include all error cases described in the atmicall(3) reference
031   page.
032   Other return codes are not possible, so there should be no
033   default within the switch statement.
034
035   if (tpabort(0) == -1){
036      char *p;
037      fprintf(stderr, "abort was attempted but failed\n");
038      p = tpstrerror(tperrno);
039      userlog("%s", p);
040   }
041   }
042   else
043   if (tpcommit(0) == -1)
044   fprintf(stderr, "REPORT program failed at commit time\n");
045
046   The following code fragment shows how an application-specific
047   return code can be examined.
048   .
049   .
050   .
051   ret = tpcall("servicename", (char*)sendbuf, 0, (char    **)&rcvbuf, &rcvlen, \
052   (long)0);
053   .
054   .
055   .
056   (void) fprintf(stdout, "Returned tpurcode is:%d\n", tpurcode);
057
058
059   free all buffers
060   tpterm();
061   exit(0);
062   }
```

tperrno(5) の値は、各問題の詳細を示し、どのレベルで問題の解決が可能かを示しています。アプリケーションで、ある処理に特定のエラー条件が定義されている場合、tpurcode の値にも同じことが言えます。

次のコード例は、tpstrerrordetail(3c) 関数を使用して、エラー発生時の詳細情報を取得する方法を示しています。

リスト 11-2 tpstrerrordetail( ) **によるエラー処理**

```
001   #include <stdio.h>
002   #include <string.h>
003   #include <atmi.h>/* BEA Tuxedo ヘッダ・ファイル */
004   #define LOOP_ITER 100
005   #if defined(__STDC__) || defined(__cplusplus)
006   main(int argc, char *argv[])
007   #else
008   main(argc, argv)
009   int argc;
010   char *argv[];
011   #endif
012   {
013   char *sendbuf, *rcvbuf;
014   long sendlen, rcvlen;
015   int ret;
016   int i;
017   if(argc != 2) {
018          (void) fprintf(stderr, "Usage:simpcl string\n");
019          exit(1);
020   }
021   /* クライアント・プロセスとして BEA Tuxedo システムにアタッチします。*/
022   if (tpinit((TPINIT *) NULL) == -1) {
023          (void) fprintf(stderr, "Tpinit failed\n");
024          exit(1);
025   }
026   sendlen = strlen(argv[1]);
027
028   /* 要求および応答用に STRING 型バッファを割り当てます。*/
029
030   if((sendbuf = (char *) tpalloc("STRING", NULL, sendlen+1)) == NULL) {
031          (void) fprintf(stderr,"Error allocating send buffer\n");
032          tpterm();
033          exit(1);
034   }
035
036   if((rcvbuf = (char *) tpalloc("STRING", NULL, sendlen+1)) == NULL) {
037          (void) fprintf(stderr,"Error allocating receive buffer\n");
```

```
038          tpfree(sendbuf);
039          tpterm();
040          exit(1);
041    }
042
043    for( i=0; i<LOOP_ITER; i++) {
044    (void) strcpy(sendbuf, argv[1]);
045
046    /* サービス TOUPPER を要求し、応答を待機します。*/
047    ret = tpcall("TOUPPER", (char *)sendbuf, 0, (char **)&rcvbuf, &rcvlen, (long)0);
048
049    if(ret == -1) {
050           (void) fprintf(stderr, "Can't send request to service TOUPPER\n");
051          (void) fprintf(stderr, "Tperrno = %d, %s\n", tperrno, tpstrerror(tperrno));
052
053           ret = tperrordetail(0);
054           if(ret == -1) {
055                  (void) fprintf(stderr, "tperrodetail() failed!\n");
056                  (void) fprintf(stderr, "Tperrno = %d, %s\n", tperrno,
tpstrerror(tperrno));
057           }
058           else if (ret != 0) {
059                  (void) fprintf( stderr, "errordetail:%s\n",
060                               tpstrerrordetail( ret, 0));
061           }
062           tpfree(sendbuf);
063           tpfree(rcvbuf);
064           tpterm();
065           exit(1);
066    }
067    (void) fprintf(stdout, "Returned string is:%s\n", rcvbuf);
068    }
069
070    /* バッファを解放して BEA Tuxedo システムからデタッチします。*/
071    tpfree(sendbuf);
072    tpfree(rcvbuf);
073    tpterm();
074    return(0);
```

トランザクションについて

以下の節では、各種のプログラミング機能がトランザクション・モードでどのように動作するかについて説明します。まず、トランザクション・モードのコーディングで従うべき基本的な通信規則について説明します。

通信規則

トランザクション・モードで実行するコードを記述する場合は、以下の基本的な通信規則に従います。

- 同じトランザクションに参加するプロセスでは、すべての要求で応答が必要です。応答を必要としない要求を行うには、tpacall() の *flags* パラメータに TPNOTRAN または TPNOREPLY を設定します。
- サービスは、tpreturn() または tpforward() を呼び出す前に、すべての非同期トランザクション応答を取得する必要があります。この規則には、コードをトランザクション・モードで実行するかどうかに関係なく従います。
- イニシエータは、tpcommit() を呼び出す前に、すべての非同期トランザクション応答 (TPNOTRAN フラグが指定されていないもの) を取得する必要があります。
- トランザクションのパーティシパント以外からの応答を必要とする非同期呼び出しには、応答を取得する必要があります。つまり、応答が抑制されたのではなくトランザクションが抑制された tpacall() で行った要求に対する応答を取得する必要があります。
- トランザクションがタイムアウトになっていなくても、「アボートのみ」としてマークされている場合、以降の通信では TPNOTRAN フラグを設定して、トランザクションがロールバックされた後でも通信の結果が保持されるようにします。
- トランザクションがタイムアウトになると、以下のようになります。
  - タイムアウトになった呼び出しの記述子は無効になり、以降この記述子を参照すると、TPEBADDESC が返されます。
  - 未処理の記述子の tpgetrply() または tprecv() を呼び出すと、トランザクション・タイムアウトについてのグローバル状態が返され、tperrno(5) に TPETIME が設定されます。
  - 非同期呼び出しで、tpacall() の *flags* パラメータに TPNOREPLY、TPNOBLOCK、または TPNOTRAN を設定できます。

- タイムアウト以外の理由でトランザクションが一度「アボートのみ」とマークされると、tpgetrply() の呼び出しでは、呼び出しのローカル状態を表す値が返されます。つまり、ローカル条件を反映する成功コードまたはエラー・コードのいずれかが返されます。
- 応答を取得するために tpgetrply() で一度記述子を使用した場合、またはエラー条件を通知するために tpsend() または tprecv() で一度記述子を使用した場合、その記述子は無効になり、以降この記述子を参照すると TPEBADDESC が返されます。この規則には、コードをトランザクション・モードで実行するかどうかに関係なく従います。
- トランザクションが一度アボートされると、未処理のトランザクションの呼び出し記述子 (TPNOTRAN フラグが設定されていないもの) はすべて無効になり、以降この記述子を参照すると TPEBADDESC が返されます。

トランザクション・エラー

以下の節では、トランザクションに関連するエラーについて説明します。

致命的ではないトランザクション・エラー

トランザクション・エラーが発生すると、tperrno(5) に TPETRAN が返されます。ただし、このようなエラーの意味は、そのエラーを返す関数によって異なります。次の表は、トランザクション・エラーを返す関数と、考えられるエラーの原因を示しています。

表 11-3 トランザクション・エラー

関数	原因
tpbegin()	通常は、トランザクションの開始を試みたときに発生する一時的なシステム・エラーが原因で起こります。呼び出しを繰り返し行うと、問題が解決します。
tpcancel()	TPNOTRAN フラグを設定しないで要求を行った後、トランザクション応答を取得するためにこの関数が呼び出されました。

関数	原因
tpresume()	呼び出し元が、1 つ以上のリソース・マネージャとグローバル・トランザクション外の作業に関与しているため、BEA Tuxedo システムがグローバル・トランザクションを再開できません。そのような作業はすべて、グローバル・トランザクションを再開する前に完了していなければなりません。ローカル・トランザクションについての呼び出し元の状態は、変更されません。
tpconnect()、 tppost()、 tpcall()、 tpacall()	トランザクションがサポートされていないサービスに対して、トランザクション・モードで呼び出しが行われました。サービスには、データベース管理システム (DBMS) にアクセスし、その結果トランザクションがサポートされるサーバ・グループに属するものがあります。そのようなグループに属さないサービスもあります。また、トランザクションがサポートされたサービスには、トランザクションがサポートされていないソフトウェアとの相互運用を必要とするものがあります。たとえば、フォームを出力するサービスの処理が、トランザクションがサポートされていないプリンタで行われる場合があります。トランザクションがサポートされていないサービスは、トランザクションのパーティシパントとして動作できない場合があります。 サービスをサーバやサーバ・グループにグループ分けする作業は、管理タスクの 1 つです。どのサービスでトランザクションがサポートされているかを確認するには、アプリケーション管理者に問い合わせてください。 トランザクション・レベルのエラーをアプリケーション・レベルで修正するには、TPNOTRAN フラグを有効にするか、またはトランザクション外でエラーが返されたサービスにアクセスします。

致命的なトランザクション・エラー

致命的なトランザクション・エラーが発生した場合、アプリケーションでは、イニシエータで tpabort() を呼び出してトランザクションを明示的にアボートしなければなりません。そのため、トランザクションにとって致命的なエラーを認識することが大切です。次の 3 つの場合、トランザクションは失敗します。

- トランザクションのイニシエータまたはパーティシパントが、次のいずれかの理由により「アボートのみ」にマークされました。
  - tpreturn() の引数の処理でエラーが発生しました。tperrno(5) に TPESVCERR が設定されます。

- tpreturn() の *rval* 引数が TPFAIL に設定されています。tperrno(5) に TPESVCFAIL が設定されます。
- 応答バッファの *type* または *subtype* が不明であるか、または呼び出し元で使用できないので、成功したか失敗したかを判断できません。tperrno(5) に TPEOTYPE が設定されます。

- トランザクションがタイムアウトになりました。tperrno(5) に TPETIME が設定されます。
- tpcommit() が、トランザクションの開始元ではなくパーティシパントによって呼び出されています。tperrno(5) に TPEPROTO が設定されます。

トランザクションにとって致命的なプロトコル・エラーが発生するのは、トランザクションの不正なパーティシパントから tpcommit() が呼び出された場合だけです。このエラーは、アプリケーション内で開発段階に修正できます。

イニシエータまたはパーティシパントで障害が発生した後、またはトランザクションがタイムアウトになった後で、tpcommit() が呼び出されると、暗黙的なアボート・エラーになります。その場合、コミットは失敗するので、トランザクションをアボートする必要があります。

通信呼び出しで TPESVCERR、TPESVCFAIL、TPEOTYPE、または TPETIME が返された場合、tpabort() を呼び出してトランザクションを明示的にアボートしなければなりません。トランザクションを明示的にアボートする前に、未処理の呼び出し記述子を待つ必要はありません。ただし、これらの記述子は、呼び出しがアボートされた後は無効と見なされるので、トランザクション終了後にこれらの記述子へのアクセスを試みると、TPEBADDESC が返されます。

TPESVCERR、TPESVCFAIL、および TPEOTYPE の場合、トランザクションがタイムアウトにならない限り、引き続き通信呼び出しを行うことができます。これらのエラーが返された場合、トランザクションは「アボートのみ」にマークされます。これ以降の処理の結果を保持するには、*flags* パラメータに TPNOTRAN を設定して通信用の関数を呼び出します。このフラグを設定すると、「アボートのみ」にマークされたトランザクションで実行された処理は、トランザクションがアボートしてもロールバックされません。

トランザクション・タイムアウトが発生しても通信を続けることはできますが、次のような通信要求を行うことはできません。

- 応答を要求すること
- ブロックすること
- 呼び出し元のトランザクションに対して実行すること

そのため、非同期呼び出しを行うには、*flags* パラメータに TPNOREPLY、TPNOBLOCK、または TPNOTRAN を設定する必要があります。

ヒューリスティックな判断に関するエラー

tpcommit() 関数は、TP_COMMIT_CONTROL の設定に応じて、TPEHAZARD または TPEHEURISTIC を返します。

TP_COMMIT_CONTROL に TP_CMT_LOGGED を設定すると、2 フェーズ・コミットの第 2 フェーズの実行前にアプリケーションに制御が移ります。その場合、第 2 フェーズ中に発生したヒューリスティックな判断がアプリケーションで認識されないことがあります。

TPEHAZARD または TPEHEURISTIC は 1 フェーズ・コミットで返すことができます。ただし、これが可能なのは、トランザクションに関与しているリソース・マネージャが 1 つだけで、1 フェーズ・コミットでこのリソース・マネージャがヒューリスティックな判断を返すか、なんらかの障害の発生を示す場合です。

TP_COMMIT_CONTROL に TP_CMT_COMPLETE を設定すると、リソース・マネージャがヒューリスティックな判断を通知する場合は TPEHEURISTIC が返され、リソース・マネージャがなんらかの障害を通知する場合は TPEHAZARD が返されます。TPEHAZARD は、コミットの第 2 フェーズ (または 1 フェーズ・コミット) でパーティシパントになんらかの障害が発生し、トランザクションが正常終了したかどうかがわからない状況を示します。

トランザクション・タイムアウト

第 11 章の 19 ページ「トランザクション・エラー」で説明したように、BEA Tuxedo アプリケーションでは、ブロッキング・タイムアウトとトランザクション・タイムアウトの 2 種類のタイムアウトが発生します。以下の節では、各種のプログラミング機能へのトランザクション・タイムアウトの影響について説明します。タイムアウトの詳細については、第 11 章の 19 ページ「トランザクション・エラー」を参照してください。

tpcommit() 関数への影響

tpcommit() を呼び出した後でタイムアウトが発生した場合、トランザクションはどのような状態になるでしょうか。トランザクションがタイムアウトになり、そのトランザクションがアボートされたことがシステムで認識されると、システムは tperrno(5) に TPEABORT を設定して、そのような状況の発生を通知します。トランザクションの状態が不明な場合は、エラー・コードに TPETIME を設定します。

トランザクションの状態が明確ではない場合、リソース・マネージャに問い合わせる必要があります。まず、トランザクションによって行われた変更が適用されたかどうかを確認します。これにより、トランザクションがコミットされたか、またはアボートされたかを判断できます。

TPNOTRAN フラグへの影響

トランザクション・モードのプロセスで、*flags* 引数に TPNOTRAN を設定して通信呼び出しを行うと、呼び出されたサービスは現在のトランザクションに参加できません。サービス要求の成功や失敗は、トランザクションの結果に影響しません。トランザクションは、サービスから応答が返されるのを待つ間にタイムアウトになる場合もあります。これは、そのサービスがトランザションに参加しているかどうかには関係ありません。

TPNOTRAN フラグの使用方法の詳細については、第 11 章の 23 ページ「tpreturn( ) および tpforward( ) 関数」を参照してください。

tpreturn( ) および tpforward( ) 関数

トランザクション・モードで実行中にプロセスを呼び出すと、tpreturn() と tpforward() は、トランザクションのサービス部分をそのトランザクションの完了時にコミットまたはアボートできる状態にします。同じトランザクションでサービスを何度も呼び出すことができます。システムは、トランザクションのイニシエータによって tpcommit() または tpabort() が呼び出されない限り、トランザクションを完全にはコミットまたはアボートしません。

サービス内で行われた通信呼び出しのすべての未処理の記述子が取得されるまで、tpreturn() または tpforward() を呼び出すことはできません。*rval* に TPSUCCESS を設定して、未処理の記述子で tpreturn() を呼び出すと、プロトコル・エラーが発生し、tpgetrply() を待機中のプロセスに TPESVCERR が返されます。そのプロセスがトランザクション・モードになっている場合、呼び出し元は「アボートのみ」にマークされます。トランザクションのイニシエータが tpcommit() を呼び出した場合も、トランザクションが暗黙的にアボートされます。*rval* に TPFAIL を設定して、未処理の記述子で tpreturn() を呼び出すと、tpgetrply() を待機中のプロセスに TPESVCFAIL が返されます。トランザクションへの影響は同じです。

トランザクション・モードで実行中に tpreturn() を呼び出すと、tpreturn() で発生したプロセス・エラー、またはアプリケーションによって *rval* に設定された値で示されるエラーにより、トランザクションの結果に影響することがあります。

tpforward() を使用すると、ある時点までは要求が正しく処理されていることを示すことができます。アプリケーション・エラーが検出されない場合、システムは tpforward() を呼び出します。アプリケーション・エラーが検出された場合、システムは TPFAIL を設定して tpreturn() を呼び出します。tpforward() を正しく呼び出さないと、システムはその呼び出しをプロセス・エラーと見なし、エラー・メッセージを要求元に返します。

tpterm( ) 関数

tpterm() 関数は、アプリケーションからクライアント・コンテキストを削除します。

クライアント・コンテキストがトランザクション・モードになっている場合、呼び出しは失敗して、tperrno(5) に TPEPROTO が返されます。クライアント・コンテキストは、トランザクション・モードでアプリケーションの一部として残ります。

呼び出しが成功すると、現在の実行スレッドはアプリケーション内に存在しなくなるため、クライアント・コンテキストは、トランザクションと通信したりトランザクションに参加できなくなります。

リソース・マネージャ

ATMI 関数を使ってトランザクションを定義すると、BEA Tuxedo システムによって内部呼び出しが実行され、トランザクションに参加している各リソース・マネージャにグローバル・トランザクション情報が渡されます。`tpcommit()` や `tpabort()` を呼び出すと、各リソース・マネージャに対して内部呼び出しが行われ、呼び出し元のグローバル・トランザクションのために行われていた作業がコミットまたはアボートされます。

グローバル・トランザクションが開始された場合 (明示的でも暗黙的でも)、アプリケーション・コードでリソース・マネージャのトランザクション関数を明示的に呼び出すことはできません。このトランザクション規則に従わないと、不安定な結果が生じます。`tpgetlev()` 関数を使用すると、リソース・マネージャのトランザクション関数を呼び出す前に、グローバル・トランザクション内に既にプロセスがあるかどうかを確認できます。

リソース・マネージャによっては、トランザクションの整合性レベルなど、特定のパラメータをプログラマが設定できるものがあります。その場合、リソース・マネージャ間のインターフェイスで使用可能なオプションを指定します。そのようなオプションは、次の 2 つの方法で使用できるようになります。

- リソース・マネージャ固有の関数呼び出し。分散アプリケーションのプログラマは、これらの関数を使用してオプションを設定することができます。
- ハードコーディングされたオプション。リソース・マネージャのプロバイダで提供されるトランザクション・インターフェイスに組み込まれています。

詳細については、リソース・マネージャのマニュアルを参照してください。

オプションの設定方法はリソース・マネージャによって異なります。たとえば、BEA Tuxedo システムの SQL リソース・マネージャの場合、`set transaction` 文を使用して、BEA Tuxedo システムによって既に開始されているトランザクションに対する特定のオプション (整合性レベルとアクセス・モード) が決まります。

トランザクションのサンプル・シナリオ

以降の節では、次のトランザクションについて説明します。

- 呼び出し元と同じトランザクションでのサービス呼び出し
- AUTOTRAN が設定された別のトランザクションでのサービス呼び出し
- 新しい明示的なトランザクションを開始するサービスの呼び出し

呼び出し元と同じトランザクションでのサービス呼び出し

トランザクション・モードになっている呼び出し元が、現在のトランザクションに参加するために別のサービスを呼び出した場合、次のようになります。

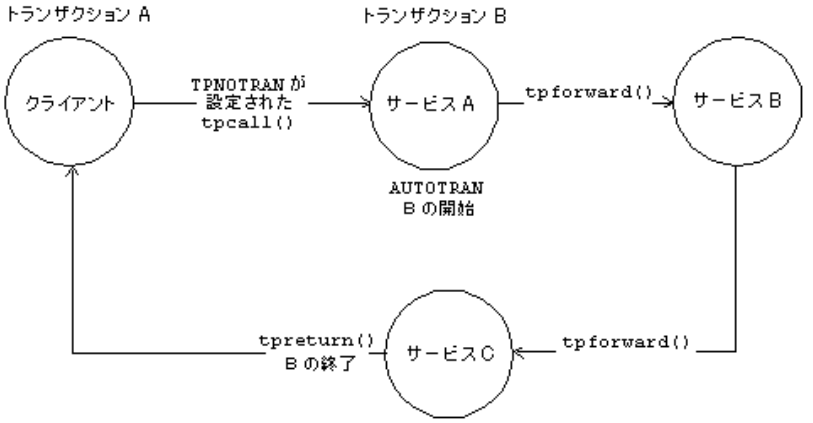
- `tpreturn()` と `tpforward()` は、トランザクションに参加しているサービスから呼び出されると、そのトランザクションのサービス部分をイニシエータによってアボートまたはコミットできる状態にします。
- 呼び出されたプロセスの成功や失敗は、現在のトランザクションに影響します。パーティシパントで致命的なトランザクション・エラーが発生すると、現在のトランザクションは「アボートのみ」にマークされます。
- 正常終了したパーティシパントによって行われた処理が適用されるかどうかは、トランザクションの結果に依存します。つまり、トランザクションがアボートされた場合は、すべてのパーティシパントの処理は適用されません。
- 現在のトランザクションに参加するために別のサービスを呼び出すときに、TPNOREPLY フラグを使用することはできません。

AUTOTRAN が設定された別のトランザクションでのサービス呼び出し

TPNOTRAN フラグを設定して通信呼び出しを行い、呼び出されたときにトランザクションが自動的に開始するようにサービスが設定されている場合、呼び出し元プロセスと呼び出されたプロセスはどちらもトランザクション・モードになります。ただし、この 2 つは別々のトランザクションを構成します。この状況では、次の処理が行われます。

- tpreturn() は、トランザクションのイニシエータの役割を果たします。つまり、トランザクションが自動的に開始されたサービスで、トランザクションを終了します。または、tpforward() によって終了するサービスでトランザクションが自動的に開始された場合、フォワード・チェーンの最後のサービスで発行された tpreturn() 呼び出しが、トランザクションのイニシエータの役割を果たします。つまり、トランザクションを終了します。例については、第 11 章の 28 ページ「AUTOTRAN が設定された tpforward( ) と tpreturn() のトランザクションでの役割」を参照してください。
- tpreturn() はトランザクション・モードなので、トランザクションのパーティシパントの障害やトランザクション・タイムアウトなどの影響を受けます。その場合、エラー・メッセージが返されます。
- 呼び出し元のトランザクションの状態は、呼び出し元に返されるエラー・メッセージやアプリケーション障害の影響を受けません。
- 呼び出し元のトランザクションは、呼び出し元が応答を待っている間にタイムアウトになることがあります。
- 応答が必要ない場合、呼び出し元のトランザクションは、通信呼び出しによる影響を受けません。

図 11-1　AUTOTRAN が設定された tpforward( ) と tpreturn() のトランザクションでの役割



新しい明示的なトランザクションを開始するサービスの呼び出し

TPNOTRAN を設定して通信呼び出しを行い、呼び出されたサービスが自動的にトランザクション・モードにならないように設定されている場合、tpbegin()、tpcommit()、および tpabort() を明示的に呼び出すと、サービスで複数のトランザクションを定義できます。その結果、tpreturn() が呼び出される前にトランザクションを完了できます。

この状況では、次の処理が行われます。

- tpreturn() はトランザクションの役割は果たしません。つまり、サービス・ルーチンにトランザクションが明示的に定義されているかどうかにかかわらず、tpreturn() の役割は常に同じです。
- トランザクションの結果に関係なく、tpreturn() は *rval* に任意の値を返します。
- 通常、システムはプロセス・エラー、バッファ・タイプ・エラー、またはアプリケーション障害を返し、TPESVCFAIL、TPEITYPE/TPEOTYPE、および TPESVCERR の一般的な規則に従います。
- 呼び出し元のトランザクションの状態は、呼び出し元に返されるエラー・メッセージやアプリケーション障害の影響を受けません。

- 呼び出し元は、応答を待っている間にトランザクションがタイムアウトになることがあります。
- 応答が必要ない場合、呼び出し元のトランザクションは、通信呼び出しによる影響を受けません。

BEA Tuxedo システムで提供されるサブルーチン

BEA Tuxedo システムで提供されるサブルーチン tpsvrinit()、tpsvrdone()、tpsvrthrinit(3c)、および tpsvrthrdone(3c) は、トランザクションで使用される場合は特定の規則に従う必要があります。

注記 tpsvrthrinit(3c) と tpsvrthrdone(3c) は、マルチスレッド・アプリケーションだけで指定できます。tpsvrinit() と tpsvrdone() は、スレッド・アプリケーションでも非スレッド・アプリケーションでも指定できます。

BEA Tuxedo システム・サーバは、初期化時に tpsvrinit() または tpsvrthrinit(3c) を呼び出します。特に、tpsvrinit() または tpsvrthrinit(3c) は、呼び出し元プロセスがサーバになった後、サービス要求の処理を開始する前に呼び出されます。tpsvrinit() または tpsvrthrinit(3c) で非同期通信を実行した場合、関数が戻る前にすべての応答が取得されなければなりません。この処理が行われなかった場合、システムは保留中の応答があっても無視して、サーバを終了します。tpsvrinit() または tpsvrthrinit(3c) でトランザクションを定義した場合、関数が戻る前にすべての非同期応答を取得して、トランザクションを終了しなければなりません。この処理が行われなかった場合、システムは未処理の応答が残っていてもトランザクションをアボートし、それらの応答をすべて無視します。その場合、サーバは正常に終了します。

BEA Tuxedo システム・サーバ用ルーチンは、サービス要求の処理が完了した後、ルーチンを終了する前に tpsvrdone() または tpsvrthrdone(3c) を呼び出します。この時点で、サーバのサービス宣言は取り消されますが、サーバ自体はアプリケーションから分離していません。tpsvrdone() または tpsvrthrdone(3c) で通信を開始した場合、これらの関数は未処理の応答をすべて取得してから戻る必要があります。この処理が行われなかった場合、システムは保留中の応答があっても無視して、サーバを終了します。tpsvrdone() または tpsvrthrdone(3c) 内でトランザクションを開始した場合、トランザクションはすべての応答を取得してから終了しなければなりません。この処理が行われなかった場合、システムは応答が残っていてもトランザクションをアボートし、応答を無視します。この場合もサーバは終了します。

中央イベント・ログ

中央イベント・ログには、BEA Tuxedo アプリケーションで発生する重要なイベントが記録されます。これらのイベントに関するメッセージは、アプリケーション・クライアントとサービスが userlog(3c) 関数を介してログに出力されます。

中央イベント・ログの分析は、アプリケーションで行う必要があります。userlog(3c) に記録するイベントに関しては、厳密なガイドラインを定義しておきます。ほとんど問題にならないようなメッセージを記録しないようにすると、アプリケーションのデバッグが簡単になります。

Windows 2000 プラットフォームの中央イベント・ログの設定の詳細については、『Windows での BEA Tuxedo の使用』を参照してください。

ログの名前

アプリケーション管理者は、コンフィギュレーション・ファイルに、各マシン上の userlog(3c) エラー・メッセージ・ファイル名の接頭辞として使用する絶対パス名を定義します。userlog(3c) 関数は、月、日、年を表す *mmddyy* の形式で日付を生成し、この日付をパス名の接頭辞に付加して中央イベント・ログの完全なファイル名を構成します。毎日、新しいファイルが作成されます。そのため、中央イベント・ログに数日間にわたってメッセージが送信された場合、メッセージはそれぞれ異なるファイルに書き込まれます。

ログ・エントリの形式

ログ・エントリは、次の要素から構成されます。

- タグ。タグは次の要素から構成されます。
  - 時刻 (*hhmmss*)。
  - マシン名。たとえば、UNIX システムでは、uname(1) コマンドから返される名前が使用されます。
  - userlog(3c) を呼び出したスレッドのスレッド ID (スレッドがサポートされていないプラットフォーム上では 0)、プロセス ID、および名前。
  - userlog(3c) を呼び出したスレッドのコンテキスト ID。

- メッセージの本文。

  各メッセージの本文の前には、そのメッセージのカタログ名と番号が付きます。
- printf(3S) 形式の省略可能な引数。

たとえば、mach1 (uname コマンドから返される名前) という UNIX マシン上で、セキュリティ・プログラムが午後 4:22:14 に次のような呼び出しを行ったとします。

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

このログ・エントリは、次のようになります。

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

この例では、security のプロセス ID は 23451、変数 usrnm の値は abc です。

前述のメッセージが、アプリケーションではなく BEA Tuxedo システムによって生成された場合は、次のようになります。

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

この例では、メッセージのカタログ名は LIBSEC_CAT 、メッセージ番号は 999 です。

プロセスがトランザクション・モードのときにメッセージが中央イベント・ログに送られると、ユーザ・ログ・エントリのタグにはそのほかの要素が付加されます。これらの要素は、リテラル文字列の gtrid と、それに続く 3 桁の long 型の 16 進数で構成されます。これらの整数はグローバル・トランザクションを一意に識別するもので、グローバル・トランザクション識別子、つまり gtrid と呼ばれます。この識別子は主に管理上の目的で使用されます。また、中央イベント・ログでメッセージの前に付加されるタグの中に挿入されます。システムがトランザクション・モードで中央イベント・ログにメッセージを書き込むと、ログ・エントリは次のようになります。

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

イベント・ログへの書き込み

イベント・ログにメッセージを書き込むには、次の手順に従います。

- ログに書き込むエラー・メッセージを char * 型の変数に割り当て、その変数名を呼び出しの引数として使用します。
- メッセージのリテラル文字列を二重引用符で囲み、次の例のように userlog(3c) 呼び出しの引数として指定します。

```
.
.
.
/* トランザクションでアクセスするデータベースをオープンします。*/
if(tpopen() == -1) {
      userlog("tpsvrinit:Cannot open database %s, tpstrerror(tperrno)");
```

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

この例では、tpopen(3c) が -1 を返した場合、メッセージが中央イベント・ログに送られます。

userlog(3c) の文法は、UNIX システムの printf(3S) 関数と同じです。どちらの関数の形式でも、リテラル文字列や変換仕様を指定して、可変個引数を使用できます。

アプリケーション・プロセスのデバッグ

userlog(3c) 文を使用して、アプリケーション・ソフトウェアをデバックできます。しかし、問題が複雑な場合は、デバッガ・コマンドを使用することもあります。

以下の節では、UNIX および Windows 2000 プラットフォーム上でアプリケーションをデバッグする方法について説明します。

UNIX プラットフォーム上でのアプリケーション・プロセスのデバッグ

UNIX システムの標準デバッグ・コマンドは、dbx(1) です。このツールの詳細については、UNIX システムのリファレンス・マニュアルで dbx(1) を参照してください。-g オプションを使用してクライアント・プロセスをコンパイルすると、dbx(1) のリファレンス・ページで説明されている手順に従ってこれらのプロセスをデバッグできます。

dbx コマンドを実行するには、次のように入力します。

```
dbx client
```

クライアント・プロセスを実行するには、次の手順に従います。

1. コード内に必要なブレークポイントを設定します。
2. dbx コマンドを入力します。
3. dbx のプロンプト (*) で、run サブコマンド (r) と、クライアント・プログラムの main() に渡すオプションを入力します。

サーバ・プログラムのデバッグは、これよりも複雑な作業です。通常、サーバは tmboot コマンドを使用して起動します。このコマンドは、サーバを正しいマシン上で正しいオプションを使って起動します。dbx を使用する場合は、tmboot コマンドからではなく、サーバを直接実行する必要があります。サーバを直接実行するには、dbx コマンドによって表示されるプロンプトの後に、r (run の省略形) を入力します。

BEA Tuxedo の tmboot(1) コマンドは、サーバであらかじめ定義されている main() に、非公開のコマンド行オプションを渡します。サーバを直接実行するには、これらのオプションを r サブコマンドに手動で渡す必要があります。指定する必要のあるオプションを確認するには、-n と -d 1 オプションを指定して tmboot を実行します。-n オプションは、tmboot が起動処理を行わないことを指定します。-d 1 は、レベル 1 のデバッグ文を表示することを指定します。デフォルトでは、-d 1 オプションは、すべてのプロセスに関する情報を返します。1 つのプロセスに関する情報だけが必要な場合は、必要に応じてオプションを追加して要求を指定できます。詳細については、『BEA Tuxedo コマンド・リファレンス』を参照してください。

tmboot -n -d 1 を実行すると、次の例に示すように、tmboot からサーバの main() に渡されるコマンド行オプションのリストが出力されます。

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

必要なコマンド行オプションを確認したら、dbx(1) の r サブコマンドでサーバ・プログラムを直接実行できます。次は、コマンド行の例です。

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

コンフィギュレーションの一部として既に動作中のサーバを実行する場合は、dbx(1) は使用しません。使用した場合、サーバは正常に終了して、サーバが複製されたことを示すメッセージが中央イベント・ログに書き込まれます。

Windows 2000 プラットフォーム上でのアプリケーション・プロセスのデバッグ

Windows 2000 プラットフォームには、Microsoft Visual C++ 環境の一部としてグラフィカル・デバッガが提供されています。このツールの詳細については、Microsoft Visual C++ のリファレンス・マニュアルを参照してください。

Microsoft Visual C++ のデバッガを起動するには、次のように start コマンドを入力します。

```
start msdev -p process_ID
```

注記 Microsoft Visual C++ 5.0 以前のデバッガを使用する場合は、次のように start コマンドを入力します。

```
start msdev -p process_id
```

デバッガを起動して自動的にプロセスに入るには、次のようにプロセス名と引数を start コマンド行に指定します。

```
start msdev filename argument
```

たとえば、デバッガを起動し、引数に ConvertThisString を指定して simpcl.exe プロセスに入るには、次のコマンドを入力します。

```
start msdev simpcl.exe ConvertThisString
```

ユーザ・モード例外が発生すると、メッセージが表示されて、デフォルトのシステム・デバッガを起動して、プログラム障害の発生場所、レジストリやスタックの状態を調べるように求められます。Windows 2000 環境では、ユーザ・モード例外の障害発生時にはデフォルトのデバッガとして「ワトソン博士」が使用され、Win32 SDK 環境ではカーネル・デバッガが使用されます。

ユーザ・モード例外の障害発生時に使用される Windows 2000 システムのデフォルト・デバッガを変更するには、次の手順に従います。

1. regedit または regedt32 を実行します。
2. HKEY_LOCAL_MACHINE サブツリー内で、\SOFTWARE\Microsoft\Windows\CurrentVersion\AeDebug に移動します。
3. Debugger キーをダブルクリックして、[文字列の編集] ダイアログ ボックスを表示します。
4. 現在表示されている文字列を変更して、使用するデバッガを指定します。

   たとえば、Microsoft Visual C++ 環境で提供されるデバッガを指定する場合は、次のコマンドを入力します。

   ```
   msdev.exe -p %ld -e %ld
   ```

注記 Microsoft Visual C++ 5.0 以前のデバッガを使用する場合は、次のようにコマンドを入力します。

```
msvc.exe -p %ld -e %ld
```

全般的なコード例

トランザクションの整合性、メッセージの通信、およびリソースへのアクセスは、オンライン・トランザクション処理 (OLTP) アプリケーションの主要な要件です。

ここでは、リソース・マネージャにアクセスする SQL 文と共に動作する通信ルーチン、バッファ管理、ATMI トランザクションを示すコード例を示します。ここで示す例は、BEA Tuxedo 銀行業務アプリケーション (bankapp) の ACCT サーバから抜粋した CLOSE_ACCT サービスです。

この例は、最初の SQL 文でデータベースにアクセスする前に、set transaction 文 (49 行目) を使用して、トランザクションの整合性レベルとアクセス・モードを設定する方法を示しています。読み取り / 書き込みの権限が指定されている場合、整合性レベルはデフォルトで高い整合性に設定されます。ACCOUNT_ID の値に基づいて口座を解約するために、SQL クエリで引き出し額を決定します (50 ～ 58 行目)。

tpalloc() は、WITHDRAWAL サービスへの要求メッセージ用のバッファを割り当て、ACCOUNT_ID と引き出し額がバッファに格納されます (62 ～ 74 行目)。次に、tpcall() を介して WITHDRAWAL サービスに要求が送信されます (79 行目)。その後、SQL の delete 文で、該当する口座を削除してデータベースを更新します (86 行目)。

すべての処理が成功すると、サービス内で割り当てられたバッファが解放され (98 行目)、サービスに送信された TPSVCINFO データ・バッファが更新されて、トランザクションが正常に終了したことが示されます (99 行目)。サービスがイニシエータであった場合、ここでトランザクションが自動的にコミットされます。tpreturn() は、口座の解約を要求したクライアント・プロセスに、TPSUCCESS と更新後のバッファを返します。最後に、要求されたサービスが正常に終了したことが、フォームのステータス行に示されます。

各関数呼び出しの後、成功したか失敗したかが確認されます。エラーが発生した場合、サービスで割り当てられたバッファが解放され、サービスで開始されたトランザクションがアボートされ、TPSVCINFO バッファが更新されてエラーの原因が示されます (80 ～ 83 行目)。最後に、tpreturn() が TPFAIL を返し、更新されたバッファ内のメッセージがフォームのステータス行に表示されます。

注記 サービス・ルーチンでグローバル・トランザクションの整合性レベルを指定する場合、同じトランザクションに参加するすべてのサービス・ルーチンで、同じようにレベルを定義する必要があります。

リスト 11-3 ACCT **サーバ**

```
001   #include <stdio.h>              /* UNIX */
002   #include <string.h>             /* UNIX */
003   #include <fml.h>                /* BEA Tuxedo システム */
004   #include <atmi.h>               /* BEA Tuxedo システム */
005   #include <Usysflds.h>           /* BEA Tuxedo システム */
006   #include <sqlcode.h>            /* BEA Tuxedo システム */
007   #include <userlog.h>            /* BEA Tuxedo システム */
008   #include "bank.h"               /* 銀行業務アプリケーションのマクロ定義 */
009   #include "bank.flds.h"          /* bankdb フィールド */
010   #include "event.flds.h"         /* イベント・フィールド */
011
012
013   EXEC SQL begin declare section;
014   static long account_id;                    /* 口座番号 */
015   static long branch_id;                     /* 支店番号 */
016   static float bal, tlr_bal;                 /* 収支 */
017   static char acct_type;                     /* 口座の種類 */
018   static char last_name[20], first_name[20]; /* 姓、名 */
019   static char mid_init;                      /* 中間名の頭文字 */
020   static char address[60];                   /* 住所 */
021   static char phone[14];                     /* 電話番号 */
022   static long last_acct;                     /* 支店で最後に開いた口座 */
023   EXEC SQL end declare section;

024   static FBFR *reqfb;             /* 要求メッセージ用のフィールド化バッファ */
025   static long reqlen;             /* 要求バッファの長さ */
026   static char amts[BALSTR];       /* 浮動小数点の文字列表現 */

027   code for OPEN_ACCT service

028   /*
029    * 口座を解約するためのサービス
030    */

031   void
032   #ifdef __STDC__
033   LOSE_ACCT(TPSVCINFO *transb)

034   #else

035   CLOSE_ACCT(transb)
036   TPSVCINFO *transb;
037   #endif
```

```
038   {
039      FBFR *transf;                /* 複合化されたメッセージのフィールド化されたバッファ */

040      /* TPSVCINFO データ・バッファへのポインタを設定 */
041         transf = (FBFR *)transb->data;

042      /* 口座番号が妥当かどうかを確認します。*/
043      if (((account_id = Fvall(transf, ACCOUNT_ID, 0)) < MINACCT) ||
044       (account_id > MAXACCT)) {
045         (void)Fchg(transf, STATLIN, 0, "Invalid account number", (FLDLEN)0);
046         tpreturn(TPFAIL, 0, transb->data, 0L, 0);
047      }

048      /* トランザクション・レベルを設定します。*/
049      EXEC SQL set transaction read write;

050      /* 削除する金額を取得します。*/
051      EXEC SQL declare ccur cursor for
052       select BALANCE from ACCOUNT where ACCOUNT_ID = :account_id;
053      EXEC SQL open ccur;
054      EXEC SQL fetch ccur into :bal;
055      if (SQLCODE != SQL_OK) {                /* 何も見つからなかった場合 */
056       (void)Fchg(transf, STATLIN, 0, getstr("account",SQLCODE), (FLDLEN)0);
057       EXEC SQL close ccur;
058       tpreturn(TPFAIL, 0, transb->data, 0L, 0);
059      }

060      /* 最終的な引き出しを実行します。*/

061      /* 引き出し要求バッファを設定します。*/
062      if ((reqfb = (FBFR *)tpalloc("FML",NULL,transb->len)) == (FBFR *)NULL) {
063       (void)userlog("tpalloc failed in close_acct\n");
064       (void)Fchg(transf, STATLIN, 0,
065           "Unable to allocate request buffer", (FLDLEN)0);
066       tpreturn(TPFAIL, 0, transb->data, 0L, 0);
067      }
068      reqlen = Fsizeof(reqfb);
069      (void)Finit(reqfb,reqlen);

070      /* 要求バッファに番号を格納します。*/
071      (void)Fchg(reqfb,ACCOUNT_ID,0,(char *)&account_id, (FLDLEN)0);

072      /* 要求バッファに金額を格納します。*/
073      (void)sprintf(amts,"%.2f",bal);
074      (void)Fchg(reqfb,SAMOUNT,0,amts, (FLDLEN)0);

075      /* この引き出しの優先順位を上げます。*/
076      if (tpsprio(PRIORITY, 0L) == -1)
```

```
077       (void)userlog("Unable to increase priority of withdraw");

078      /* withdraw サービスで残額を削除するために tpcall を呼び出します。*/
079      if (tpcall("WITHDRAWAL", (char *)reqfb, 0L, (char **)&reqfb,
080       (long *)&reqlen,TPSIGRSTRT) == -1) {
081       (void)Fchg(transf, STATLIN, 0,"Cannot make withdrawal", (FLDLEN)0);
082       tpfree((char *)reqfb);
083       tpreturn(TPFAIL, 0,transb->data, 0L, 0);
084      }

085      /* 口座レコードの削除 */

086      EXEC SQL delete from ACCOUNT where current of ccur;
087      if (SQLCODE != SQL_OK) {                /* 削除に失敗した場合 */
088       (void)Fchg(transf, STATLIN, 0,"Cannot close account", (FLDLEN)0);
089        EXEC SQL close ccur;
090        tpfree((char *)reqfb);
091        tpreturn(TPFAIL, 0, transb->data, 0L, 0);
092      }
093      EXEC SQL close ccur;

094      /* 処理が成功した場合の応答バッファを準備します。*/
095      (void)Fchg(transf, SBALANCE, 0, Fvals(reqfb,SAMOUNT,0), (FLDLEN)0);
096      (void)Fchg(transf, FORMNAM, 0, "CCLOSE", (FLDLEN)0);
097      (void)Fchg(transf, STATLIN, 0, " ", (FLDLEN)0);
098      tpfree((char *)reqfb);
099      tpreturn(TPSUCCESS, 0, transb->data, 0L, 0);
100   }
```