Sun Cluster 3.1 10/03 データサービス開発ガイド

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 817–4330–10
2003 年 10 月, Revision A

原典: *Sun Cluster 3.1 10/03 Data Services Developer's Guide*

Part No: 817-0520-10

Revision A

目次

はじめに

このマニュアルでは、RMAPI (Resource Management (リソース管理) API) を使用して Sun Cluster データサービスを開発する方法について説明します。

対象読者

このマニュアルは、Sun のソフトウェアとハードウェアについて豊富な知識を持っている経験のある開発者を対象にしています。このマニュアルの情報は、Solaris™ オペレーティング環境の知識があることを前提としています。

内容の紹介

このマニュアルは、次の章と付録で構成されています。

- 第 1 章では、データサービスを開発するのに必要な概念について説明します。
- 第 2 章では、データサービスの開発に関する詳細な情報を説明します。
- 第 4 章では、Resource Management API (RMAPI) を構成するアクセス関数とコールバックメソッドに関する情報を説明します。
- 第 5 章では、`in.named()` アプリケーション用の Sun Cluster データサービスの例を示します。
- 第 6 章では、Data Services Development Library (DSDL) を形成するアプリケーションプログラミングインタフェースの概要を説明します。
- 第 7 章では、リソースタイプの設計と実装におけるDSDL の代表的な使用例について説明します。

- 第 8 章DSDL により実装されるリソースタイプの例を説明します。
- 第 9 章では、SunPlex Agent Builder について説明します。
- 第 10 章では、一般的なデータサービスの作成方法について説明します。
- 第 11 章では、DSDL API 関数について説明します。
- 付録 A では、標準リソースタイプ、リソースグループ、およびリソースプロパティについて説明します。
- 付録 B では、データサービスの例について、それぞれのメソッドの完全なコードを示します。
- 付録 C では、`SUNW.xfnts()` リソースタイプにおける各メソッドの完全なコードを示します。
- 付録 D では、Resource Group Manager (RGM) の名前と値についての文字の要件を説明します。
- 付録 E では、クラスタに対応していない、通常のアプリケーションを高可用性に適用させる要件を説明します。

関連マニュアル

アプリケーション	タイトル	パート番号
概念	『*Sun Cluster 3.1 10/03* の概念』	817-4329
ソフトウェアのインストール	『*Sun Cluster 3.1 10/03* ソフトウェアのインストール』	817-4328
管理	『*Sun Cluster 3.1 10/03* のシステム管理』	817-4327
API 開発	『*Sun Cluster 3.1 10/03* データサービス開発ガイド』	817-4330
エラーメッセージ	『*Sun Cluster 3.1 10/03 Error Messages Guide*』	817-0521
ハードウェア	『*Sun Cluster 3.x Hardware Administration Manual*』	817-0168
	『*Sun Cluster 3.x Hardware Administration Collection*』 `http://docs.sun.com/db/coll/1024.1` より参照可能	
データサービス	『*Sun Cluster 3.1* データサービスの計画と管理』	817-4317
	`http://docs.sun.com/` の『*Sun Cluster 3.1 Data Services 10/03 Collection*』より参照可能	
マニュアルページ	『*Sun Cluster 3.1 10/03 Reference Manual*』	817–0522

アプリケーション	タイトル	パート番号
リリースノート	『*Sun Cluster 3.1 10/03* ご使用にあたって』	817–4522
	『*Sun Cluster 3.x Release Notes Supplement*』	816-3381

問い合わせについて

Sun Cluster のインストールまたは使用で問題が発生した場合は、ご購入先に連絡し、次の情報をお伝えください。

- 名前と電子メールアドレス (利用している場合)
- 会社名、住所、および電話番号
- システムのモデルとシリアル番号
- オペレーティング環境のバージョン番号 (例: Solaris 10)
- Sun Cluster のバージョン番号 (例: Sun Cluster 3.1)

ご購入先に知らせる、システム上の各ノードについての情報を収集するには、次のコマンドを使用します。

コマンド	機能
`prtconf -v`	システムメモリのサイズと周辺デバイス情報を表示する
`psrinfo -v`	プロセッサの情報を表示する
`showrev -p`	インストールされているパッチを報告する
`prtdiag -v`	システム診断情報を表示する
`/usr/cluster/bin/scinstall -pv`	Sun Cluster のリリースとパッケージバージョン情報を表示する

上記の情報にあわせて、`/var/adm/messages` ファイルの内容もご購入先にお知らせください。

Sun のオンラインマニュアル

`docs.sun.com` では、Sun が提供しているオンラインマニュアルを参照することができます。マニュアルのタイトルや特定の主題などをキーワードとして、検索を行うこともできます。URL は、`http://docs.sun.com` です。

表記上の規則

このマニュアルでは、次のような字体や記号を特別な意味を持つものとして使用します。

表 **P–1** 表記上の規則

字体または記号	意味	例
AaBbCc123	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コード例を示します。	.login ファイルを編集します。 ls -a を使用してすべてのファイルを表示します。 system%
AaBbCc123	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して示します。	system% **su** password:
AaBbCc123	変数を示します。実際に使用する特定の名前または値で置き換えます。	ファイルを削除するには、rm *filename* と入力します。
『 』	参照する書名を示します。	『コードマネージャ・ユーザーズガイド』を参照してください。
「 」	参照する章、節、ボタンやメニュー名、強調する単語を示します。	第 5 章「衝突の回避」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅を超える場合に、継続を示します。	sun% **grep '^#define ** **XV_VERSION_STRING'**

コード例は次のように表示されます。

- C シェル

  machine_name% **command y|n** [*filename*]

- C シェルのスーパーユーザー

  machine_name# **command y|n** [*filename*]

- Bourne シェルおよび Korn シェル

  $ **command y|n** [*filename*]

- Bourne シェルおよび Korn シェルのスーパーユーザー

  # **command y|n** [*filename*]

[ ] は省略可能な項目を示します。上記の例は、*filename* は省略してもよいことを示しています。

| は区切り文字 (セパレータ) です。この文字で分割されている引数のうち 1 つだけを指定します。

キーボードのキー名は英文で、頭文字を大文字で示します (例: `Shift` キーを押します)。ただし、キーボードによっては `Enter` キーが `Return` キーの動作をします。

ダッシュ (-) は 2 つのキーを同時に押すことを示します。たとえば、`Ctrl-D` は `Control` キーを押したまま `D` キーを押すことを意味します。

第 1 章

リソース管理の概要

このマニュアルでは、Oracle、Sun™ One Web Server、DNS などのソフトウェアアプリケーション用のリソースタイプを作成するためのガイドラインを説明します。したがって、このマニュアルはリソースタイプの開発者を対象としています。

この章では、データサービスを開発するために理解しておく必要がある概念について説明します。この章の内容は、次のとおりです。

注 – このマニュアルでは、「リソースタイプ」と「データサービス」という用語を同じ意味で使用しています。また、このマニュアルではほとんど使用されることはありませんが、「エージェント」という用語も「リソースタイプ」や「データサービス」と同じ意味で使用されます。

Sun Cluster アプリケーション環境

Sun Cluster システムを使用すると、アプリケーションを高度な可用性とスケーラビリティを備えたリソースとして実行および管理できます。RGM (Resource Group Manager) というクラスタ機能は、高可用性とスケーラビリティを実現するための機構を提供します。この機能を利用するためのプログラミングインタフェースを形成する要素は、次のとおりです。

- ユーザーが作成するコールバックメソッド。RGM は、このコールバックメソッドを利用してクラスタ上のアプリケーションを制御します。

- Resource Management API (RMAPI)。コールバックメソッドの作成に使用する低レベルの API コマンドおよび API 関数です。これらの API は、`libscha.so` ライブラリに実装されます。
- クラスタ上のプロセスを監視および再起動するプロセス管理機能。
- データサービス開発ライブラリ (Data Service Development Library: DSDL)。低レベル API およびプロセス管理機能をより高レベルでカプセル化し、コールバックメソッドの作成を支援するいくつかの機能を追加するライブラリ関数です。これらの関数は、`libdsdev.so` ライブラリに実装されます。

次の図は、これらの要素の相互関係を示しています。

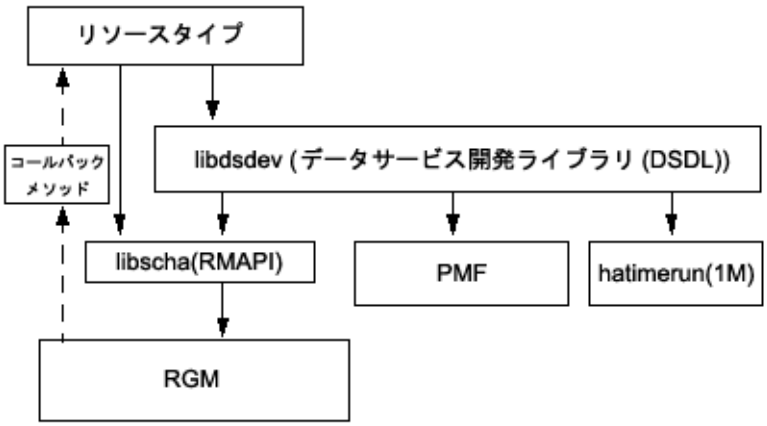


図 **1–1** プログラミングアーキテクチャ

Sun Cluster パッケージには、データサービスの作成プロセスを自動化する SunPlex Agent Builder ™ というツールが含まれます (第 9 章 を参照)。 Agent Builder はデータサービスのコードを C 言語または Korn シェル (`ksh`) のどちらでも生成できます。前者の場合は DSDL 関数、後者の場合は低レベルの API コマンドを使ってコールバックメソッドを作成します。

RGM は各クラスタ上でデーモンとして動作して、事前構成したポリシーに従って、選択したノード上のリソースを自動的に起動および停止します。リソースの高可用性を実現するために、RGM は、ノードが異常終了または再起動すると、影響を受けるノード上でリソースを停止し、別のノード上でリソースを起動します。また、リソースに固有のモニター (監視機能) を起動および停止することによって、障害のあるリソースを検出し、別のノードに再配置したり、さまざまな視点からリソース性能を監視します。

RGM はフェイルオーバーリソースとスケーラブルリソースの両方をサポートします。フェイルオーバーリソースとは、同時に 1 つのノード上だけでオンラインになることができるリソースのことです。スケーラブルリソースとは、同時に複数のノード上でオンラインになることができるリソースのことです。

RGM モデル

ここでは、基本的な用語をいくつか紹介し、RGM とそれに関連するインタフェースについて詳細に説明します。

RGM は、「リソースタイプ」、「リソース」、「リソースグループ」という 3 種類の相互に関連するオブジェクトを処理します。これらのオブジェクトを紹介するために、次のような例を使用します。

開発者は、既存の Oracle DBMS アプリケーションを高可用性にするためのリソースタイプ `ha-oracle` を実装します。エンドユーザーは、マーケティング、エンジニアリング、および財務ごとに異なるデータベースを定義し、それぞれのリソースタイプを `ha-oracle` にします。クラスタ管理者は、上記リソースを異なるリソースグループに分類することによって、異なるノード上で実行したり、個別にフェイルオーバーできるようにします。開発者は、もう 1 つのリソースタイプ ha-calendar を作成し、Oracle データベースを必要とする高可用性のカレンダサーバーを実装します。クラスタ管理者は、財務カレンダリソースと財務データベースリソースを同じリソースグループに分類することによって、両方のリソースを同じノード上で実行したり、一緒にフェイルオーバーできるようにします。

リソースタイプ

リソースタイプは、クラスタ上で実行されるソフトウェアアプリケーション、アプリケーションをクラスタリソースとして管理するために RGM がコールバックメソッドとして使用する制御プログラムおよびクラスタの静的な構成の一部を形成するプロパティセットからなります。RGM は、リソースタイププロパティを使って特定のタイプのリソースを管理します。

注 – リソースタイプは、ソフトウェアアプリケーションだけでなく、その他のシステムリソース (ネットワークアドレスなど) も表します。

リソースタイプの開発者は、リソースタイププロパティを指定し、その値をリソースタイプ登録 (RTR) ファイルに設定します。RTR ファイルの形式は、34 ページの「リソースとリソースタイププロパティの設定」と `rt_reg(4)` のマニュアルページの記述に従い、明確に定義されています。リソースタイプ登録ファイルの例については、88 ページの「リソースタイプ登録ファイルの定義」を参照してください。

表 A–1 に、リソースタイププロパティのリストを示します。

クラスタ管理者は、リソースタイプの実装と実際のアプリケーションをクラスタにインストールし、登録します。さらに登録手順で、リソースタイプ登録ファイルの情報をクラスタ構成に入力します。データサービスの登録手順については、『*Sun Cluster 3.1 Data Service Planning and Administration Guide*』を参照してください。

リソース

リソースは、そのリソースタイプからプロパティと値を継承します。さらに、開発者は、リソースタイプ登録ファイルでリソースプロパティを宣言できます。リソースプロパティのリストについては、表 A–2 を参照してください。

クラスタ管理者は、リソースタイプ登録 (RTR) ファイルにプロパティを指定することによって、特定のプロパティの値を変更できます。たとえば、プロパティ定義に値の許容範囲を指定しておきます。これにより、プロパティが調節可能なときに、作成時、常時、不可などを指定できます。このような許容範囲内であれば、クラスタ管理者は管理コマンドでプロパティを変更できます。

クラスタ管理者は、同じタイプのリソースを多数作成して、各リソースに独自の名前とプロパティ値のセットを持たせることができます。これによって、実際のアプリケーションの複数のインスタンスをクラスタ上で実行できます。このとき、各インスタンスにはクラスタ内で一意の名前が必要です。

リソースグループ

各リソースはリソースグループに構成する必要があります。RGM は、同じグループのすべてのリソースを同じノード上でオンラインかオフラインにします。このとき、グループ内の個々のリソースに対してコールバックメソッドを呼び出します。

リソースグループがオンラインになっているノードを主ノードと呼びます。リソースグループは、その主ノードによってマスター *(*制御*)* されます。各リソースグループは、クラスタ管理者が設定した独自の `Nodelist` プロパティを持っており、これによってリソースグループの潜在的な主ノードを識別します。

リソースグループはプロパティセットも持っています。このようなプロパティには、クラスタ管理者が設定できる構成プロパティや、RGM が設定してリソースグループのアクティブな状態を反映する動的プロパティがあります。

RGM は、2 種類のリソースグループ、フェイルオーバー (failover) とスケーラブル (scalable) を定義します。 フェイルオーバーリソースグループは、同時に 1 つのノード上だけでオンラインになることができます。一方、スケーラブルリソースグループは、同時に複数のノード上でオンラインになることができます。RGM は、各種類のリソースグループを作成するためのプロパティセットを提供します。このようなプロパティの詳細については、33 ページの「データサービスをクラスタに転送する方法」と 42 ページの「コールバックメソッドの実装」を参照してください。

リソースグループのプロパティのリストについては、表 A–3 を参照してください。

Resource Group Manager

Resource Group Manager (RGM) は `rgmd` デーモンとして実装され、クラスタの各メンバー (ノード) 上で動作します。`rgmd` プロセスはすべて互いに通信し、単一のクラスタ規模の機能として動作します。

RGM は、次の機能をサポートします。

- ノードが起動またはクラッシュしたとき、管理対象のすべてのリソースグループを適切なマスター上で自動的にオンラインにし、その可用性を維持します。
- 特定のリソースが異常終了した場合、そのモニタープログラムはリソースグループを同じマスター上で再起動するか、新しいマスターに切り替えるかを要求できます。
- クラスタ管理者は管理コマンドを発行して、次のいずれかのアクションを要求できます。
  - リソースグループをマスターする権利の変更
  - リソースグループ内の特定のリソースの有効化または無効化
  - リソース、リソースグループ、リソースタイプの作成、削除、変更

RGM は、構成を変更するとき、そのアクションをクラスタのすべてのメンバー (ノード) 間で調整します。このような動作を「再構成」と呼びます。RGM は、個々のリソースの状態を変更をするため、各リソース上でリソースタイプに固有のコールバックメソッドを呼び出します。

コールバックメソッド

Sun Cluster フレームワークは、コールバックメソッドを使用して、データサービスと RGM 間の通信を実現します。また、コールバックメソッド(引数と戻り値を含む) のセットと、RGM が個々のメソッドを呼び出す環境を定義します。

データサービスを作成するには、個々のコールバックメソッドのセットをコーディングし、個々のメソッドを RGM から呼び出し可能な制御プログラムとして実装します。つまり、データサービスは、単一の実行可能コードではなく、多数の実行可能なスクリプト (`ksh`) またはバイナリ (C 言語) から構成されており、それぞれを RGM から直接呼び出すことができます。

コールバックメソッドを RGM に登録するには、リソースタイプ登録 (RTR) ファイルを使用します。RTR ファイルには、データサービスとして実装した各メソッドのプログラムを指定します。システム管理者がデータサービスをクラスタに登録すると、RGM は RTR ファイルにあるさまざまな情報の中からコールバックプログラムの識別情報を読み取ります。

リソースタイプの必須コールバックメソッドは、起動メソッド (`Start` または `Prenet_start`) と停止メソッド (`Stop` または `Postnet_stop`) だけです。

コールバックメソッドは、次のようなカテゴリに分類できます。

- 制御および初期化メソッド
  - `Start` と `Stop` は、オンラインまたはオフラインにするグループ内のリソースを起動または停止します。
  - `Init`、`Fini`、 `Boot` は、リソース上で初期化と終了コードを実行します。
- 管理サポートメソッド
  - `Validate` は、管理アクションによって設定されるプロパティを確認します。
  - `Update` は、オンラインリソースのプロパティ設定を更新します。
- ネットワーク関連メソッド
  - `Prenet_start` と `Postnet_stop` は、同じリソースグループ内のネットワークアドレスが「起動」に構成される前、または「停止」に構成された後に、特別な起動アクションまたは停止アクションを実行します。
- モニター制御メソッド
  - `Monitor_start` と `Monitor_stop` は、リソースのモニターを起動または停止します。
  - `Monitor_check` は、リソースグループがノードに移動される前に、ノードの信頼性を査定します。

コールバックメソッドの詳細については第 4 章と `rt_callbacks(1HA)` のマニュアルページを参照してください。コールバックメソッドの使用例については、第 5 章および第 8 章を参照してください。

プログラミングインタフェース

リソース管理アーキテクチャは、データサービス用のコードを作成するため、低レベルのベース API、ベース API 上のものより高いレベルのライブラリを提供します。さらに、いくつかの基本的な入力情報をもとにデータサービスを自動的に生成するツール、SunPlex Agent Builder を提供します。

RMAPI

データサービスは、RMAPI (Resource Management API) の低レベルルーチンを使って、システム内のリソース、リソースタイプ、リソースグループの情報にアクセスします。ローカルの再起動やフェイルオーバーの要求、リソースの状態の設定も行います。これらの関数にアクセスするには、`libscha.so` ライブラリを使用します。RMAPI は、これらのコールバックメソッドを、シェルコマンドまたは C 関数の形で提供できます。RMAPI ルーチンの詳細については、`scha_calls(3HA)` のマニュアルページと第 4 章を参照してください。データサービスコールバックメソッドにおける RMAPI の使用例については、第 5 章を参照してください。

データサービス開発ライブラリ (DSDL)

データサービス開発ライブラリ (Data Service Development Library: DSDL) は、RMAPI 上に構築されており、RGM のメソッドコールバックモデルを基盤にして上位レベルの統合フレームワークを提供します。DSDL は、次のようなさまざまなデータサービス開発向けの機能を提供します。

- `libscha.so`—低レベルのリソース管理 API
- PMF—プロセスとその子孫の監視、停止したプロセスの再起動などを実行するプロセス管理機能 (`pmfadm`(1M) および `rpc.pmfd`(1M) のマニュアルページを参照)
- `hatimerun`—タイムアウトを適用してプログラムを実行するための機能 (`hatimerun(1M)` のマニュアルページを参照)

DSDL は、大多数のアプリケーションに対して、データサービスの構築に必要なほとんどまたはすべての機能を提供します。DSDL は、低レベルの API の代わりになるものではなく、低レベルの API をカプセル化および拡張するためのものであることに注意してください。実際、多くの DSDL 関数は `libscha.so` 関数を呼び出します。`libscha.so` 関数を直接呼び出すこともできますが、DSDL を使用することにより、データサービスの大部分を作成できます。DSDL 関数は `libdsdev.so` ライブラリとして実装されています。

DSDL の詳細については、第 6 章と `scha_calls(3HA)` のマニュアルページを参照してください。

SunPlex Agent Builder

Agent Builder は、データサービスの作成を自動化するツールです。ユーザーがターゲットアプリケーションの基本情報を入力すると、ソースコードと実行コード (C または Korn シェル)、カスタマイズされた RTR ファイル、Solaris パッケージを利用して、ユーザーの代わりにデータサービスを作成します。

大多数のアプリケーションでは、Agent Builder を使用することにより、わずかなコードを手作業で変更するだけで完全なデータサービスを生成できます。追加プロパティの妥当性検査を必要とするような、より要件の厳しいアプリケーションには、

Agent Builder では対応できないこともあります。しかし、このような場合でもコードの大部分を生成できるので、手作業によるコーディングは残りの部分だけで済みます。Agent Builder を使用すれば、少なくとも独自の Solaris パッケージを生成することができます。

Resource Group Manager の管理インタフェース

Sun Cluster はクラスタを管理するために、グラフィカルユーザーインタフェースとコマンドセットの両方を提供します。

SunPlex Manager

SunPlex Manager は、次の作業を実行できる Web ベースのツールです。

- クラスタのインストール
- クラスタの管理
- リソースおよびリソースグループの作成と構成
- Sun Cluster ソフトウェアを使ったデータサービスの構成

SunPlex Manager のインストール方法、SunPlex Manager によるクラスタソフトウェアのインストール方法については、『*Sun Cluster 3.1 10/03* ソフトウェアのインストール』を参照してください。管理作業については、SunPlex Manager のオンラインヘルプを参照してください。

管理コマンド

RGM オブジェクトの管理用 Sun Cluster コマンドは、`scrgadm(1M)`、`scswitch(1M)`、および `scstat(1M) -g` の 3 つです。

`scrgadm` コマンドでは、RGM が使用するリソースタイプ、リソースグループ、リソースオブジェクトの表示、作成、構成が可能です。このコマンドはクラスタの管理インタフェースの一部であり、この章の残りで説明しているアプリケーションインタフェースとは異なったプログラミングコンテキストで使用されます。しかし、このコマンドを使って、API が動作するクラスタ構成を構築することもできます。管理インタフェースを理解すると、アプリケーションインタフェースも理解しやすくなります。`scrgadm` コマンドで実行できる管理作業の詳細については、`scrgadm(1M)` のマニュアルページを参照してください。

scswitch コマンドでは、指定のノード上のリソースグループのオンラインとオフラインの切り替えや、リソースまたはそのモニターの有効と無効の切り替えが可能です。scswitch コマンドで実行できる管理作業の詳細については、scswitch(1M) のマニュアルページを参照してください。

scstat -g コマンドでは、すべてのリソースグループおよびリソースの現在の動的な状態を表示できます。

第 2 章

データサービスの開発

この章では、データサービスを開発するための詳細な方法について説明します。

この章の内容は次のとおりです。

アプリケーションの適合性の分析

データサービスを作成するための最初の手順では、ターゲットアプリケーションが高可用性またはスケーラビリティを備えるための要件を満たしているかどうかを判定します。すべての要件を満たしていない場合は、要件を満たすようにアプリケーションのソースコードを変更します。

次に、アプリケーションが高可用性またはスケーラビリティを備えるための要件を要約します。要件に関する詳細情報を確認したい場合や、アプリケーションのソースコードを変更する必要がある場合は、付録 B を参照してください。

注 – スケーラブルサービスを実現するためには、次に示す高可用性の要件をすべて満たした上で、いくつかの追加要件も満たしている必要があります。

- Sun Cluster 環境では、ネットワーク対応 (クライアントサーバーモデル) とネットワーク非対応 (クライアントレス) のアプリケーションはどちらも、高可用性またはスケーラビリティを備えることが可能です。ただし、タイムシェアリング環境では、アプリケーションは サーバー上で動作し、`telnet` または `rlogin` 経由でアクセスされるため、Sun Cluster の可用性を強化することはできません。
- アプリケーションはクラッシュに対する耐障害性 (クラッシュトレラント) を備えていなければなりません。つまり、ノードが予期せぬ停止状態になった後、アプリケーションは再起動時に必要なディスクデータを復元できなければなりません。さらに、クラッシュ後の復元時間にも制限が課せられます。ディスクを復元し、アプリケーションを再起動できる能力は、データの整合性に関わる問題であるため、クラッシュトレラントであることは、アプリケーションが高可用性を備えるための前提条件となります。データサービスは接続を復元できる必要はありません。
- アプリケーションは、自身が動作するノードの物理的なホスト名に依存してはなりません。詳細については、317 ページの「ホスト名」を参照してください。
- アプリケーションは、複数の IP アドレスが構成されている環境で正しく動作する必要があります。たとえば、ノードが複数のパブリックネットワーク上に存在する多重ホームホスト環境や、単一のハードウェアインタフェース上に複数の論理インタフェースが構成されているノードが存在する環境で正しく動作しなければなりません。
- 高可用性を備えるには、アプリケーションデータはクラスタファイルシステム内に格納されている必要があります。315 ページの「多重ホストデータ」を参照してください。

  アプリケーションがデータの格納先を示すのに固定されたパス名を使用している場合、アプリケーションのソースコードを変更しなくても、そのパスをクラスタファイルシステム内の場所を指すシンボリックリンクに変更できる場合があります。詳細については、316 ページの「多重ホストデータを配置するためのシンボリックリンクの使用」を参照してください。
- アプリケーションのバイナリとライブラリは、ローカルの各ノードまたはクラスタファイルシステムのどちらにも格納できます。クラスタファイルシステム上に格納する利点は、1 箇所にインストールするだけで済む点です。欠点としては、アプリケーションが RGM の制御下で動作している間はバイナリファイルが使用中になるので、ローリングアップグレードの問題が生じることが挙げられます。
- 初回の照会がタイムアウトした場合、クライアントは自動的に照会を再試行できる必要があります。アプリケーションとプロトコルがすでに単一サーバーのクラッシュと再起動に対応できている場合、関連するリソースグループのフェイルオーバーまたはスイッチオーバーにも対応できる必要があります。詳細については、319 ページの「クライアントの再試行」を参照してください。
- アプリケーションは、クラスタファイルシステム内で UNIX ドメインソケットまたは名前付きパイプを使用してはなりません。

さらに、スケーラブルサービスは、次の要件も満たしている必要があります。

- アプリケーションは、複数のインスタンスを実行でき、すべてのインスタンスがクラスタファイルシステム内の同じアプリケーションデータを処理できる必要があります。
- アプリケーションは、複数のノードからの同時アクセスに対してデータの整合性を保証する必要があります。
- アプリケーションは、クラスタファイルシステムのように、広域的に使用可能な機構を備えたロック機能を実装している必要があります。

スケーラブルサービスの場合、アプリケーションの特性により負荷均衡ポリシーが決定されます。たとえば、負荷均衡ポリシー `LB_WEIGHTED` は、任意のインスタンスがクライアントの要求に応答できるポリシーですが、クライアント接続にサーバー上のメモリー内キャッシュを使用するアプリケーションには適用されません。この場合、特定のクライアントのトラフィックをアプリケーションの 1 つのインスタンスに制限する負荷均衡ポリシーを指定する必要があります。負荷均衡ポリシー `LB_STICKY` と `LB_STICKY_WILD` は、クライアントからのすべての要求を同じアプリケーションインスタンスに繰り返して送信します。この場合、アプリケーションはメモリー内キャッシュを使用できます。異なるクライアントから複数の要求が送信された場合、RGM はサービスの複数のインスタンスに要求を分配します。スケーラブルデータサービスに対応した負荷均衡ポリシーを設定する方法については、50 ページの「フェイルオーバーリソースの実装」を参照してください。

使用するインタフェースの決定

Sun Cluster 開発者サポートパッケージ (`SUNWscdev`) は、データサービスメソッドのコーディング用に 2 種類のインタフェースセットを提供します。

- Resource Management API (RMAPI) - 低レベルのルーチンセット (`libscha.so` ライブラリとして実装されている)
- データサービス開発ライブラリ (Data Services Development Library: DSDL) - RMAPI の機能をカプセル化し、いくつかの追加機能を提供する、より高いレベルの関数セット (`libdsdev.so` ライブラリとして実装されている)

Sun Cluster 開発者サポートパッケージには、データサービスの作成を自動化するツールである SunPlex Agent Builder も含まれています。

次に、データサービスを開発する際の推奨手順を示します。

1. C 言語または Korn シェルのどちらでコーディングするかを決定します。DSDL は C 言語用のインタフェースしか提供しないため、Korn シェルでコーディングする場合は使用できません。
2. Agent Builder を実行します。必要な情報を入力するだけで、ソースコード、実行可能コード、RTR ファイル、パッケージを含むデータサービスを生成できます。

3. 生成されたデータサービスをカスタマイズする必要がある場合は、生成されたソースファイルに DSDL コードを追加できます。Agent Builder は、ソースファイル内において独自のコードを追加できる場所にコメント文を埋め込みます。
4. ターゲットアプリケーションをサポートするために、さらにコードをカスタマイズする必要がある場合は、既存のソースコードに RMAPI 関数を追加できます。

実際には、データサービスを作成する方法はいくつもあります。たとえば、Agent Builder によって生成されたコード内の特定の場所に独自のコードを追加する代わりに、生成されたメソッドの 1 つや生成された監視プログラムを DSDL や RMAPI 関数を使って最初から作成したプログラムで完全に置き換えることができます。しかし、使用する方法に関わらず、ほとんどの場合は Agent Builder を使って開発作業を開始することをお勧めします。次に、その理由を示します。

- Agent Builder が生成するコードは本質的に汎用であり、多数のデータサービスでテストされています。
- Agent Builder は、RTR ファイル、make ファイル、リソースのパッケージなど、データサービス用のサポートファイルを作成します。データサービスのコードをまったく使用しない場合でも、このようなサポートファイルを使用することによってかなりの作業を省略できます。
- 生成されたコードは変更できます。

注 – RMAPI は C 言語用の関数セットとスクリプト用のコマンドセットを提供しますが、DSDL は C 言語用の関数インタフェースしか提供しません。DSDL は `ksh` コマンドを提供しないので、Agent Builder で Korn shell (`ksh`) 出力を指定した場合、生成されるソースコードは RMAPI を呼び出します。

データサービス作成用開発環境の設定

データサービスの開発を始める前に、Sun Cluster 開発パッケージ (`SUNWscdev`) をインストールして、Sun Cluster のヘッダーファイルやライブラリファイルにアクセスできるようにする必要があります。このパッケージがすでにすべてのクラスタノード上にインストールされている場合でも、通常は、クラスタノード上にはない独立した開発マシン、すなわちクラスタノード以外の開発マシンで開発を行います。このような場合、`pkgadd` を使って、開発マシンに `SUNWscdev` パッケージをインストールする必要があります。

コードをコンパイルおよびリンクするとき、ヘッダーファイルとライブラリファイルを識別するオプションを設定する必要があります。(クラスタノード以外の) 開発マシンで開発が終了すると、完成したデータサービスをクラスタに転送して、実行および検証できます。

注 – 必ず開発バージョンの Solaris 8 以上を使用してください。

この節では、次の手順を使用します。

- Sun Cluster 開発パッケージ (SUNWscdev) をインストールして、適切なコンパイラオプションとリンカーオプションを設定します。
- データサービスをクラスタに転送します。

▼ 開発環境の設定方法

SUNWscdev パッケージをインストールして、コンパイラオプションとリンカーオプションをデータサービス開発用に設定する方法について説明します。

1. スーパーユーザーになるか、あるいは同等の役割を持つ役割を宣言し、使用したい **CD-ROM** ディレクトリに移動します。

   ```
   # cd CD-ROM_directory
   ```

2. **SUNWscdev** パッケージを現在のディレクトリにインストールします。

   ```
   # pkgadd -d . SUNWscdev
   ```

3. **Makefile** に、データサービスのコードが使用する **include** ファイルとライブラリファイルを示すコンパイラオプションとリンカーオプションを指定します。

   -I オプションは、Sun Cluster のヘッダーファイルを指定します。-L オプションは、開発システム上にあるコンパイル時ライブラリの検索パスを指定します。-R オプションはクラスタの実行時リンカーの検索パスを指定します。

   ```
   # サンプルデータサービスの Makefile
   ...

   -I /usr/cluster/include

   -L /usr/cluster/lib

   -R /usr/cluster/lib
   ...
   ```

データサービスをクラスタに転送する方法

開発マシン上でデータサービスの開発が完了した場合、クラスタに転送して検証する必要があります。この転送を行うときは、エラーが発生する可能性を減らすために、データサービスのコードと RTR ファイルを一緒にパッケージに保管して、その後、クラスタのすべてのノード上でパッケージをインストールすることを推奨します。

注 – データサービスをインストールするときは、pkgadd を使用するかどうかに関わらず、すべてのクラスタノード上にデータサービスをインストールする必要があります。Agent Builder は、RTR ファイルとデータサービスのコードを自動的にパッケージ化します。

リソースとリソースタイププロパティの設定

Sun Cluster は、データサービスの静的な構成を定義するためのリソースタイププロパティおよびリソースプロパティのセットを提供します。リソースタイププロパティでは、リソースのタイプ、そのバージョン、API のバージョンと同時に、各コールバックメソッドへのパスも指定できます。表 A–1 に、すべてのリソースタイププロパティのリストを示します。

リソースプロパティ (Failover_mode、Thorough_probe_interval など) やメソッドタイムアウトも、リソースの静的な構成を定義します。動的なリソースプロパティ (Resource_state や Status など) は、管理対象のリソースの動作状況を反映します。リソースプロパティについては、表 A–2 を参照してください。

リソースタイプおよびリソースプロパティは、データサービスの重要な要素であるリソースタイプ登録 (RTR) ファイルで宣言します。RTR ファイルは、クラスタ管理者が Sun Cluster でデータサービスを登録するとき、データサービスの初期構成を定義します。

Agent Builder が宣言するプロパティセットはどのようなデータサービスにとっても有用かつ必須です。したがって、独自のデータサービス用の RTR ファイルを生成するときは、Agent Builder を使用することを推奨します。たとえば、ある種のプロパティ (Resource_type など) が RTR ファイルで宣言されていない場合、データサービスの登録は失敗します。必須ではなくても、その他のプロパティも RTR ファイルで宣言されていなければ、システム管理者はそれらのプロパティを利用することはできません。いくつかのプロパティは RTR ファイルで宣言されていなくても使用することができますが、これは RGM がそのプロパティを定義し、そのデフォルト値を提供しているためです。このような複雑さを回避するためにも、Agent Builder を使用して、適切な RTR ファイルを生成するようにしてください。必要であれば、Agent Builder で生成した RTR ファイルを編集して、特定の値を変更することもできます。

以降では、Agent Builder で作成した RTR ファイルの例を示します。

リソースタイププロパティの宣言

クラスタ管理者は、RTR ファイルで宣言されているリソースタイププロパティを構成することはできません。このようなリソースタイププロパティは、リソースタイプの恒久的な構成の一部を形成します。

注 – `Installed_nodes` というリソースタイププロパティは、システム管理者が構成できます。事実、Installed_nodes はシステム管理者が構成できる唯一のリソースタイププロパティであり、RTR ファイルでは宣言できません。

次に、リソースタイプ宣言の構文を示します。

property_name = value;

注 – RGM はプロパティ名の大文字と小文字を区別します。Sun が提供する RTR ファイルのプロパティに対する命名規則では、名前の最初の文字が大文字で、残りが小文字です (メソッド名は例外)。メソッド名は、プロパティ属性と同様にすべて大文字です。

次に、サンプルのデータサービス (`smpl`) 用の RTR ファイルにおけるリソースタイプ宣言を示します。

```
# Sun Cluster Data Services Builder template version 1.0
# Registration information and resources for smpl
#
#NOTE: Keywords are case insensitive, i.e., you can use
#any capitalization style you prefer.
#
Resource_type = "smpl";
Vendor_id = SUNW;
RT_description = "Sample Service on Sun Cluster";

RT_version ="1.0";
API_version = 2;
Failover = TRUE;

Init_nodes = RG_PRIMARIES;

RT_basedir=/opt/SUNWsmpl/bin;

Start           =    smpl_svc_start;
Stop            =    smpl_svc_stop;

Validate        =    smpl_validate;
Update          =    smpl_update;

Monitor_start   =    smpl_monitor_start;
Monitor_stop    =    smpl_monitor_stop;
```

```
Monitor_check   =    smpl_monitor_check;
```

ヒント – RTR ファイルの最初のエントリには、Resource_type プロパティを宣言する必要があります。宣言しないと、リソースタイプの登録は失敗します。

リソースタイプ宣言の最初のセットは、次のようなリソースタイプについての基本的な情報を提供します。

Resource_type および Vendor_id	リソースタイプの名前を提供します。リソースタイプ名は Resource_type プロパティ (この例では「smpl」) 単独で指定できます。Vendor_id を接頭辞として使用し、リソースタイプ (この例では「SUNW.smpl」) との区切りにドット (.) を入力することもできます。Vendor_id を使用する場合、リソースタイプを定義する企業の略号にします。リソースタイプ名はクラスタ内で一意である必要があります。 注 – 便宜上、リソースタイプ名 (*Resource_type* と *Vendor_id*) はパッケージ名として使用されます。パッケージ名は 9 文字に制限されているので、これら 2 つのプロパティの文字数の合計も 9 文字以内に制限することをお勧めします。RGM は 9 文字の制限を適用しません。一方、Agent Builder はリソースタイプ名からパッケージ名を系統だてて生成します。つまり、Agent Builder は 9 文字の制限を適用します。
Rt_version	サンプルデータサービスのバージョンです。
API_version	API のバージョンです。たとえば、「API_version =2」は、データサービスが Sun Cluster バージョン 3.0 の管理下で動作していることを示します。
Failover = TRUE	このデータサービスが、複数のノード上で同時にオンラインにできるリソースグループ 上では実行できないサービス、すなわちフェイルオーバーデータサービスであるこ

	とを示します。詳細については、33 ページの「データサービスをクラスタに転送する方法」を参照してください。
Start、Stop、Validate など	RGM によって呼び出されるコールバックメソッドプログラムのパスを提供します。これらのパスは、 RT_basedir で指定されたディレクトリからの相対パスです。

リソースタイプ宣言の残りのセットは、次のような構成情報を提供します。

Init_nodes = RG_PRIMARIES	RGM が、データサービスをマスターできるノード上でのみ Init、Boot、Fini、Validate の各メソッドを呼び出すように指定します。RG_PRIMARIES で指定されたノードは、データサービスがインストールされているすべてのノードのサブセットです。この値に RT_INSTALLED_NODES を設定した場合、RGM は、データサービスがインストールされているすべてのノード上で上記メソッドを呼び出します。
RT_basedir	コールバックメソッドパスのような完全な相対パスとして、/opt/SUNWsample/bin をポイントします。
Start、Stop、Validate など	RGM によって呼び出されるコールバックメソッドプログラムのパスを提供します。これらのパスは、 RT_basedir で指定されたディレクトリからの相対パスです。

リソースプロパティの宣言

リソースタイププロパティと同様に、リソースプロパティも RTR ファイルで宣言します。便宜上、リソースプロパティ宣言は RTR ファイルのリソースタイププロパティ宣言の後に行います。リソース宣言の構文では、一連の属性と値のペアを記述して、全体を中括弧で囲みます。

```
{
    Attribute = Value;
    Attribute = Value;
                .
                .
                .
    Attribute = Value;
}
```

Sun Cluster が提供するリソースプロパティ (システム定義プロパティ) の場合、特定の属性は RTR ファイルで変更できます。たとえば、Sun Cluster はコールバックメソッドごとにメソッドタイムアウトプロパティを定義して、そのデフォルト値を提供します。RTR ファイルを使用すると、異なるデフォルト値を指定できます。

Sun Cluster が提供するプロパティ属性を使用することにより、RTR ファイル内に新しいリソースプロパティ (拡張プロパティ) を定義することもできます。表 A–4 に、リソースプロパティを変更および定義するための属性を示します。拡張プロパティ宣言は RTR ファイルのシステム定義プロパティ宣言の後に行います。

システム定義リソースプロパティの最初のセットでは、コールバックメソッドのタイムアウト値を指定します。

```
. . .

# リソースプロパティ宣言は中括弧で囲まれたリストであり、
#リソースタイププロパティ宣言の後で宣言する。#プロパティ名宣言は、リソースプロパティエントリの左中括弧の
# 直後にある最初の属性でなければならない。
#
# メソッドタイムアウト用の最小値とデフォルト値を設定する。
{
        PROPERTY = Start_timeout;
        MIN=60;
        DEFAULT=300;
}

{
        PROPERTY = Stop_timeout;
        MIN=60;
        DEFAULT=300;
}
{
        PROPERTY = Validate_timeout;
        MIN=60;
        DEFAULT=300;
}
{
        PROPERTY = Update_timeout;
        MIN=60;
        DEFAULT=300;
}
{
        PROPERTY = Monitor_Start_timeout;
        MIN=60;
        DEFAULT=300;
}
{
        PROPERTY = Monitor_Stop_timeout;
        MIN=60;
        DEFAULT=300;
{
        PROPERTY = Monitor_Check_timeout;
```

```
                MIN=60;
                DEFAULT=300;
        }
```

プロパティ名 (PROPERTY = *value*) は、各リソースプロパティ宣言における最初の属性でなければなりません。リソースプロパティは、RTR ファイルのプロパティ属性で定義された範囲内で構成することができます。たとえば、各メソッドタイムアウト用のデフォルト値は 300 秒です。システム管理者はこの値を変更できますが、指定できる最小値は、MIN 属性で指定されているように 60 秒です。リソースプロパティ属性の完全なリストについては、表 A–4 を参照してください。

リソースプロパティの次のセットは、データサービスにおいて特定の目的に使用されるプロパティを定義します。

```
{
        PROPERTY = Failover_mode;
        DEFAULT=SOFT;
        TUNABLE = ANYTIME;
}
{
        PROPERTY = Thorough_Probe_Interval;
        MIN=1;
        MAX=3600;
        DEFAULT=60;
        TUNABLE = ANYTIME;
}

# ある期限内に再試行する回数。この回数を超えると、
# 当該ノード上ではアプリケーションが起動できないと判断される。
{
        PROPERTY = Retry_Count;
        MAX=10;
        DEFAULT=2;
        TUNABLE = ANYTIME;
}

# Retry_Interval に60 の倍数を設定する。
# この値は秒から分に変換され、切り上げられる。
# たとえば、50 秒は 1 分に変更される。このプロパティを使用して、
# 再試行回数 (Retry_Count) を指定する。
{
        PROPERTY = Retry_Interval;
        MAX=3600;
        DEFAULT=300;
        TUNABLE = ANYTIME;
}

{
        PROPERTY = Network_resources_used;
        TUNABLE = WHEN_DISABLED;
        DEFAULT = "";
}
{
```

```
        PROPERTY = Scalable;
        DEFAULT = FALSE;
        TUNABLE = AT_CREATION;
}
{
        PROPERTY = Load_balancing_policy;
        DEFAULT = LB_WEIGHTED;
        TUNABLE = AT_CREATION;
}
{
        PROPERTY = Load_balancing_weights;
        DEFAULT = "";
        TUNABLE = ANYTIME;
}
{
        PROPERTY = Port_list;
        TUNABLE = AT_CREATION;
        DEFAULT = ;
}
```

上記のリソースプロパティ宣言では、システム管理者が値を設定し、制限を設けることができる TUNABLE 属性が追加されています。AT_CREATION は、システム管理者が値を指定できるのはリソースの作成時だけであり、後で変更できないことを示します。

上記のプロパティのほとんどは、特に理由がないかぎり、Agent Builder が生成するデフォルト値を使用しても問題ありません。こうしたプロパティのあとには、次のような情報が続きます (詳細については、246 ページの「リソースプロパティ」または r_properties(5) のマニュアルページを参照してください)。

Failover_mode
: Start または Stop メソッドの失敗時、RGM がリソースグループを再配置するか、ノードを停止するかを指定します。

Thorough_probe_interval、Retry_count、Retry_interval
: 障害モニターで使用します。障害モニターが適切に機能していない場合、システム管理者はいつでも調整できます。

Network_resources_used
: データサービスで使用される論理ホスト名または共有アドレスリソースのリスト。このプロパティは、Agent Builder によって宣言されるので、システム管理者はデータサービスを構成するとき必要に応じてリソースのリストを指定できます。

Scalable
: FALSE に設定した場合、このリソースはクラスタネットワーキング (共有アドレス) 機能を使用しません。この設定は、リソースタイプ Failover プロパティに TRUE を設定して、フェイルオーバーサービスを指定するのと同じです。このプロパティの詳しい使用方法については、33 ページの「データサービスをクラスタに転送する方法」および 42 ページの「コールバックメソッドの実装」を参照してください。

Load_balancing_policy、Load_balancing_weights
これらのプロパティは自動的に宣言されますが、フェイルオーバーリソースタイプでは使用されません。

Port_list
サーバーが待機するポートのリストです。このプロパティは Agent Builder によって宣言されるので、システム管理者はデータサービスを構成するときポートのリストを指定できます。

拡張プロパティの宣言

次に、RTR ファイルの最後の例として、拡張プロパティを示します。

```
# 拡張プロパティ
#

# クラスタ管理者は、このプロパティに値を設定して、アプリケーション
# が使用する構成ファイルが格納されているディレクトリを指定する
# 必要がある。このアプリケーション（smpl）の場合、PXFS 上に
# ある構成ファイル（通常は named.conf）のパスを指定する。
{
        PROPERTY = Confdir_list;
        EXTENSION;
        STRINGARRAY;
        TUNABLE = AT_CREATION;
        DESCRIPTION = "The Configuration Directory Path(s)";
}

# 次の 2 つのプロパティは、障害モニターの再起動を制御する。
{
        PROPERTY = Monitor_retry_count;
        EXTENSION;
        INT;
        DEFAULT = 4;
        TUNABLE = ANYTIME;
        DESCRIPTION = "Number of PMF restarts allowed for fault monitor.";
}
{
        PROPERTY = Monitor_retry_interval;
        EXTENSION;
        INT;
        DEFAULT = 2;
        TUNABLE = ANYTIME;
        DESCRIPTION = "Time window (minutes) for fault monitor restarts.";
}
# 検証用のタイムアウト値（秒）。
{
        PROPERTY = Probe_timeout;
        EXTENSION;
        INT;
        DEFAULT = 120;
        TUNABLE = ANYTIME;
```

```
        DESCRIPTION = "Time out value for the probe (seconds)";
}

# PMF 用の子プロセス監視レベル (pmfadm の -C オプション)。
# デフォルトの -1 は、pmfadm -C オプションを使用しないこと
# を示す。
# 0 より大きな値は、目的の子プロセス監視レベルを示す。
{
        PROPERTY = Child_mon_level;
        EXTENSION;
        INT;
        DEFAULT = -1;
        TUNABLE = ANYTIME;
        DESCRIPTION = "Child monitoring level for PMF";
}
# ユーザー追加コード -- BEGIN VVVVVVVVVVVV
# ユーザー追加コード -- END   ^^^^^^^^^^^^
```

次に示すように、Agent Builder は、ほとんどのデータサービスにとって有用な拡張プロパティを作成します。

`Confdir_list`
: アプリケーション構成ディレクトリへのパスを指定します。このプロパティは多くのアプリケーションにとって有用な情報です。データサービスを構成するときに、システム管理者はこのディレクトリの場所を指定できます。

`Monitor_retry_count`、`Monitor_retry_interval`、`Probe_timeout`
: サーバーデーモンではなく、障害モニター自体の再起動を制御します。

`Child_mon_level`
: PMF による監視レベルを設定します。詳細については、`pmfadm(1M)` のマニュアルページを参照してください。

「ユーザー追加コード」というコメント文で囲まれた部分に、追加の拡張プロパティを作成できます。

コールバックメソッドの実装

この節では、コールバックメソッドの実装に関する一般的な情報について説明します。

リソースとリソースグループのプロパティ情報へのアクセス

一般に、コールバックメソッドはリソースのプロパティにアクセスする必要があります。RMAPI は、リソースのシステム定義プロパティと拡張プロパティにアクセスするために、コールバックメソッドで使用できるシェルコマンドと C 関数の両方を提供します。詳細については、`scha_resource_get(1HA)` と `scha_resource_get(3HA)` のマニュアルページを参照してください。

DSDL は、システム定義プロパティにアクセスするための C 関数セット (プロパティごとに 1 つ) と、拡張プロパティにアクセスするための関数を提供します。詳細については、`scds_property_functions(3HA)` と `scds_get_ext_property(3HA)` のマニュアルページを参照してください。

`Status` と `Status_msg` の設定を除き、リソースプロパティを設定する API 関数が存在しないため、プロパティ機構を使用して、データサービスの動的な状態情報を格納することはできません。したがって、動的な状態情報は、広域ファイルに格納するようにします。

注 – クラスタ管理者は、`scrgadm` コマンド、グラフィカル管理コマンド、またはグラフィカル管理インタフェースを使って、ある種のリソースプロパティを設定することができます。ただし、`scrgadm` はクラスタの再構築時、すなわち RGM がメソッドを呼び出した時点でエラー終了するため、コールバックメソッドから呼び出さないようにします。

メソッドの呼び出し回数への非依存性

一般に、RGM は、同じリソース上で同じメソッドを (同じ引数で) 何回も連続して呼び出すことはありません。ただし、`Start` メソッドが失敗した場合には、リソースが起動していなくても RGM はそのリソース上で `Stop` メソッドを呼び出すことができます。同様に、リソースデーモンが自発的に停止している場合でも、RGM はそのリソース上で `Stop` メソッドを呼び出すことができます。`Monitor_start` メソッドと `Monitor_stop` メソッドにも、同じことが当てはまります。

このような理由のため、`Stop` メソッドと `Monitor_stop` メソッドは呼び出し回数に依存しないように組み込む必要があります。同じリソース上で、同じパラメータを指定して `Stop` メソッドまたは `Monitor_stop` メソッドを何回連続して呼び出しても、1 回だけ呼び出したときと同じ結果になります。

呼び出し回数に依存しないということは、リソースまたはモニターがすでに停止し、動作していなくても、`Stop` メソッドと `Monitor_stop` メソッドが 0 (成功) を返す必要があるということも意味します。

注 – Init、Fini、Boot、Update の各メソッドも呼び出し回数に依存しない必要があります。Start メソッドは呼び出し回数に依存してもかまいません。

汎用データサービス

汎用データサービス (GDS) は、単純なアプリケーションを Sun Cluster の Resource Group Manager フレームワークに組み込むことにより、スケーラビリティと高可用性を実現する機構です。この機構では、アプリケーションを可用性の高いものにしたり、スケーラブルなものにするために通常必要になるエージェントのコーディングは必要ありません。

GDS モデルは、コンパイル済みのリソースタイプ SUNQ.gds により、RGM フレームワークとやりとりします。

詳細については、第 10 章を参照してください。

アプリケーションの制御

RGM は、ノードがクラスタに結合されるとき、またはクラスタから切り離されるとき、コールバックメソッドを使って実際のリソース (アプリケーション) を制御できます。

リソースの起動と停止

リソースタイプを実装するには、少なくとも、Start メソッドと Stop メソッドが必要です。RGM は、リソースタイプのメソッドプログラムを適切なノード上で適切な回数だけ呼び出して、リソースグループをオフラインまたはオンラインにします。たとえば、クラスタノードのクラッシュ後、そのノードがマスターしているリソースグループを新しいノードに移動します。Start メソッドを実装して、RGM が正常に動作しているホストノード上で各リソースを再起動できるようにする必要があります。

Start メソッドは、ローカルノード上でリソースが起動し、使用可能な状態になるまで終了してはいけません。初期化に時間がかかるリソースタイプでは、その Start メソッドに、十分な長さのタイムアウト値を設定する必要があります。Start_timeout プロパティのデフォルト値と最小値は、リソースタイプ登録ファイルで設定します。

Stop メソッドは、RGM がリソースをオフラインにする状況に合わせて実装する必要があります。たとえば、リソースグループがノード 1 上でオフラインになり、ノード 2 上でもう一度オンラインになると仮定します。リソースグループをオフラインにしている間、RGM は、そのリソースグループ内のリソース上で Stop メソッドを呼び出して、ノード 1 上のすべての活動を停止しようとします。ノード 1 上ですべてのリソースの Stop メソッドが完了したら、RGM は、ノード 2 上でそのリソースグループを再度オンラインにします。

Stop メソッドは、ローカルノード上でリソースがすべての活動を完全に停止し完全にシャットダウンするまで終了してはいけません。Stop メソッドは、ローカルノード上でリソース関連のすべてのプロセスを終了することでもっとも安全に実装できます。シャットダウンに時間がかかるリソースタイプでは、その Stop メソッドに十分な長さのタイムアウト値を設定する必要があります。Stop_timeout プロパティはリソースタイプ登録ファイルで設定します。

Stop メソッドが失敗またはタイムアウトすると、リソースグループはエラー状態になり、システム管理者の介入が必要となります。この状態を回避するには、Stop および Monitor_stop メソッドがすべてのエラー状態から回復するようにする必要があります。理想的には、これらのメソッドは 0 (成功) のエラー状態で終了し、ローカルノード上でリソースとそのモニターのすべての活動を正常に停止するべきです。

Start および Stop メソッドを使用するかどうかの決定

この節では、 Start メソッドと Stop メソッドを使用するか、または、Prenet_start メソッドと Postnet_stop メソッドを使用するかを決定するときのいくつかの注意事項について説明します。どちらのメソッドを使用するのが適切かを決定するには、クライアントおよびデータサービスのクライアントサーバー型ネットワークプロトコルについて十分に理解している必要があります。

ネットワークアドレスリソースを使用するサービスでは、論理ホスト名のアドレス構成から始まる順番で、起動手順または停止手順を実行する必要があります。コールバックメソッドの Prenet_start と Postnet_stop を使用してリソースタイプを実装すると、同じリソースグループ内のネットワークアドレスが「起動」に構成される前、または「停止」に構成されたあとに、特別な起動アクションまたは停止アクションを行います。

RGM は、データサービスの Prenet_start メソッドを呼び出す前に、ネットワークアドレスを取り付ける (plumb、ただし起動には構成しない) メソッドを呼び出します。RGM は、データサービスの Postnet_stop メソッドを呼び出したあとに、ネットワークアドレスを取り外す (unplumb) メソッドを呼び出します。RGM がリソースグループをオンラインにするときは、次のような順番になります。

1. ネットワークアドレスを取り付けます。
2. データサービスの Prenet_start メソッドを呼び出します (存在する場合)。
3. ネットワークアドレスを「起動」に構成します。
4. データサービスの Start メソッドを呼び出します (存在する場合)。

RGM がリソースグループをオフラインにするときは、逆の順番になります。

1. データサービスの Stop メソッドを呼び出します (存在する場合)。
2. ネットワークアドレスを「停止」に構成します。
3. データサービスの Postnet_stop メソッドを呼び出します (存在する場合)。
4. ネットワークアドレスを取り外します。

Start、Stop、Prenet_start、Postnet_stop のうち、どのメソッドを使用するかを決定するには、まずサーバー側を考えます。データサービスアプリケーションリソースとネットワークアドレスリソースの両方を持つリソースグループをオンラインにするとき、RGM は、データサービスリソースの Start メソッドを呼び出す前に、ネットワークアドレスを「起動」に構成するメソッドを呼び出します。したがって、データサービスを起動するときにネットワークアドレスが「起動」に構成されている必要がある場合は、Start メソッドを使用してデータサービスを起動します。

同様に、データサービスアプリケーションリソースとネットワークアドレスリソースの両方を持つリソースグループをオフラインにするとき、RGM は、データサービスリソースの Stop メソッドを呼び出したあとに、ネットワークアドレスを「停止」に構成するメソッドを呼び出します。したがって、データサービスを停止するときにネットワークアドレスが「起動」に構成されている必要がある場合は、Stop メソッドを使用してデータサービスを停止します。

たとえば、データサービスを起動または停止するときに、データサービスの管理ユーティリティまたはライブラリを呼び出す必要がある場合もあります。また、クライアントサーバー型ネットワークインタフェースを使用して管理を実行するような管理ユーティリティまたはライブラリを持っているデータサービスもあります。つまり、管理ユーティリティがサーバーデーモンを呼び出すので、管理ユーティリティまたはライブラリを使用するためには、ネットワークアドレスが「起動」に構成されている必要があります。このような場合は、Start メソッドと Stop メソッドを使用します。

データサービスが起動および停止するときにネットワークアドレスが「停止」に構成されている必要がある場合は、Prenet_start メソッドと Postnet_stop メソッドを使用して データサービスを起動および停止します。クラスタ再構成 (SCHA_GIVEOVER 引数を指定した scha_control() または scswitch によるスイッチオーバー) のあとネットワークアドレスとデータサービスのどちらが最初にオンラインになるかによってクライアントソフトウェアの応答が異なるかどうかを考えます。たとえば、クライアントの実装が最小限の再試行を行うだけで、データサービスのポートが利用できないと判断すると、すぐにあきらめる場合もあります。

データサービスを起動するときにネットワークアドレスが「起動」に構成されている必要がない場合、ネットワークインタフェースが「起動」に構成される前に、データサービスを起動します。すると、ネットワークアドレスが「起動」に構成されるとすぐに、データサービスはクライアントの要求に応答できます。したがって、クライアントが再試行を停止する可能性も減ります。このような場合は、Start ではなく、Prenet_start メソッドを使用してデータサービスを起動します。

Postnet_stop メソッドを使用した場合、ネットワークアドレスが「停止」に構成されている時点では、データサービスリソースは「起動」のままです。Postnet_stop メソッドを呼び出すのは、ネットワークアドレスが「停止」に構成されたあとだけで

す。結果として、データサービスの TCP または UDP のサービスポート (つまり、その RPC プログラム番号) は、常に、ネットワーク上のクライアントから利用できます。ただし、ネットワークアドレスが応答しない場合を除きます。

`Start` メソッドと `Stop` メソッドを使用するか、`Prenet_start` メソッドと `Postnet_stop` メソッドを使用するか、または両方を使用するかを決定するには、サーバーとクライアントの要件と動作を考慮に入れる必要があります。

`Init`、`Fini`、`Boot` の各メソッド

RGM は、3 つの任意のメソッド `Init`、`Fini`、`Boot` を使用し、リソース上で初期化と終了コードを実行できます。リソースを管理下に置くとき (リソースが属しているリソースグループを管理していない状態から管理している状態に切り替えるとき、またはすでに管理されているリソースグループでリソースを作成するとき)、RGM は `Init` メソッドを呼び出して、1 回だけリソースの初期化を実行します。

リソースを管理下から外すとき (リソースが属しているリソースグループを管理していない状態に切り替えるとき、またはすでに管理されているリソースグループからリソースを削除するとき)、RGM は `Fini` を呼び出して、リソースをクリーンアップします。クリーンアップは呼び出し回数に依存しない必要があります。つまり、すでにクリーンアップが行われている場合、`Fini` は 0 (成功) で終了する必要があります。

RGM は、新たにクラスタに結合したノード、すなわち起動または再起動したノード上で、`Boot` メソッドを呼び出します。

`Boot` メソッドは、通常、`Init` と同じ初期化を実行します。この初期化は呼び出し回数に依存しない必要があります。つまり、ローカルノード上ですでにリソースが初期化されている場合、`Boot` と `Init` は 0 (成功) で終了する必要があります。

リソースの監視

通常、モニターは、リソース上で定期的に障害検証を実行し、検証したリソースが正しく動作しているかどうかを検出するように実装します。障害検証が失敗した場合、モニターはローカルで再起動するか、RMAPI 関数 `scha_control ()` または DSDL 関数 `scds_fm_action()` を呼び出して、影響を受けるリソースグループのフェイルオーバーを要求できます。

また、リソースの性能を監視して、性能を調節または報告できます。可能であれば、リソースタイプに固有な障害モニターを作成することを推奨します。このような障害モニターを作成しなくても、リソースタイプは Sun Cluster により基本的なクラスタの監視が行われます。Sun Cluster は、ホストハードウェアの障害、ホストのオペレーティングシステムの全体的な障害、およびパブリックネットワーク上で通信できるホストの障害を検出します。

RGM がリソースモニターを直接呼び出すことはありませんが、RGM は自動的にリソース用のモニターを起動する準備を整えます。リソースをオフラインにするとき、RGM は、リソース自体を停止する前に、Monitor_stop メソッドを呼び出して、ローカルノード上でリソースのモニターを停止します。リソースをオンラインにするとき、RGM は、リソース自体を起動した後に、Monitor_start メソッドを呼び出します。

RMAPI 関数 scha_control() と scha_control() を呼び出す DSDL 関数 scds_fm_action() を使用すると、リソースモニターは異なるノードへのリソースグループのフェイルオーバーを要求できます。Monitor_check が定義されている場合、scha_control は健全性検査の一環としてこの関数を呼び出して、リソースが属するリソースグループをマスターするのに要求されたノードが十分信頼できるかどうかを判断します。Monitor_check が「このノードは信頼できない」と報告した場合、あるいはメソッドがタイムアウトした場合、RGM はフェイルオーバー要求に適する別のノードを探します。すべてのノードで Monitor_check が失敗した場合、フェイルオーバーは取り消されます。

リソースモニターは、モニターから見たリソースの状態を反映するように Status と Status_msg プロパティを設定します。これらのプロパティを設定するには、RMAPI 関数 scha_resource_setstatus() または scha_resource_setstatus コマンド、あるいは DSDL 関数 scds_fm_action() を使用します。

注 – Status と Status_msg はリソースモニターに固有の使用方法ですが、これらのプロパティは任意のプログラムで設定できます。

RMAPI による障害モニターの実装例については、104 ページの「障害モニターの定義」を参照してください。DSDL による障害モニターの実装例については、151 ページの「SUNW.xfnts 障害モニター」を参照してください。Sun が提供するデータサービスに組み込まれている障害モニターについては、『*Sun Cluster 3.1* データサービスの計画と管理』を参照してください。

メッセージログのリソースへの追加

状態メッセージをほかのクラスタメッセージと同じログファイルに記録する場合は、scha_cluster_getlogfacility() 関数を使用して、クラスタメッセージを記録するために使用されている機能番号を取得します。

この機能番号を通常の Solaris syslog() 関数で使用して、状態メッセージをクラスタログに書き込みます。scha_cluster_get() 汎用インタフェースからでも、クラスタログ機能情報にアクセスできます。

プロセス管理の提供

リソースモニターとリソース制御コールバックを実装するために、プロセス管理機能が RMAPI および DSDL に提供されています。RMAPI は次の機能を定義します。これらのコマンドとプログラムの詳細については、各マニュアルページを参照してください。

プロセス監視機能 `pmfadm` および `rpc.pmfd`	プロセス監視機能 (Process Monitor Facility: PMF) は、プロセスとその子孫を監視し、プロセスが終了したときに再起動する手段を提供します。この機能は、監視するプロセスを起動および制御する `pmfadm` コマンドと、`rpc.pmfd` デーモンからなります。
`halockrun`	ファイルロックを保持しながら子プログラムを実行するためのプログラムです。このコマンドはシェルスクリプトで使用すると便利です。
`hatimerun`	タイムアウト制御下で子プログラムを実行するためのプログラムです。このコマンドはシェルスクリプトで使用すると便利です。

DSDL は、`hatimerun` 機能を実装するために `scds_hatimerun` 関数を提供します。

DSDL は、PMF 機能を実装するための関数セット (`scds_pmf_*`) を提供します。DSDL の PMF 機能の概要と、個々の関数のリストについては、206 ページの「PMF 関数」を参照してください。

リソースへの管理サポートの提供

リソース上での管理アクションには、リソースプロパティの設定と変更があります。このような管理アクションを行うために、API は `Validate` と `Update` というコールバックメソッドを定義しています。

リソースの作成時や、管理アクションによるリソースまたはリソースグループのプロパティの更新時、RGM は、オプションの `Validate` メソッドを呼び出します。RGM は、リソースとそのリソースグループのプロパティ値を `Validate` メソッドに渡します。RGM は、リソースタイプの `Init_nodes` プロパティが示す複数のクラスタノー

ド上で Validate を呼び出します。Init_nodes の詳細については、239 ページの「リソースタイププロパティ」か、rt_properties(5) のマニュアルページを参照してください。RGM は、作成または更新が行われる前に Validate を呼び出します。任意のノード上でメソッドから失敗の終了コードが戻ってくると、作成または更新は取り消されます。

RGM が Validate メソッドを呼び出すのは、管理アクションがリソースまたはグループのプロパティを変更したときだけです。RGM がプロパティを設定したときや、モニターがリソースプロパティ Status と Status_msg を設定したときではありません。

RGM は、オプションの Update メソッドを呼び出して、プロパティが変更されたことを実行中のリソースに通知します。RGM は、管理アクションがリソースまたはそのリソースグループのプロパティの設定に成功したあとに、Update を呼び出します。RGM は、リソースがオンラインであるノード上で、このメソッドを呼び出します。このメソッドは、API アクセス関数を使用して、アクティブなリソースに影響する可能性があるプロパティ値を読み取り、その値に従って、実行中のリソースを調節できます。

フェイルオーバーリソースの実装

フェイルオーバーリソースグループには、ネットワークアドレス (組み込みリソースタイプである論理ホスト名や共有アドレスなど) やフェイルオーバーリソース (フェイルオーバーデータサービス用のデータサービスアプリケーションリソースなど) があります。データサービスがフェイルオーバーするかスイッチオーバーされると、ネットワークアドレスリソースは関連するデータサービスリソースと共にクラスタノード間を移動します。RGM は、フェイルオーバーリソースの実装をサポートするプロパティをいくつか提供します。

ブール型リソースタイププロパティ Failover を TRUE に設定し、同時に複数のノード上でオンラインになることができるリソースグループだけで構成されるようにリソースを制限します。このプロパティのデフォルト値は FALSE です。したがって、フェイルオーバーリソースを実現するためには、RTR ファイルで TRUE として宣言する必要があります。

Scalable リソースプロパティは、リソースがクラスタ共有アドレス機能を使用するかどうかを決定します。フェイルオーバーリソースの場合、フェイルオーバーリソースは共有アドレスを使用しないので、Scalable には FALSE を設定します。

RG_mode リソースグループプロパティを使用すると、クラスタ管理者はリソースグループがフェイルオーバーまたはスケーラブルのどちらであるかを識別できます。RG_mode が FAILOVER の場合、RGM はリソースグループの Maximum_primaries

プロパティを 1 に設定して、リソースグループが単一のノードでマスターされるように制限します。RGM は、Failover プロパティが TRUE であるリソースを、RG_mode が SCALABLE であるリソースグループで作成することを禁止します。

Implicit_network_dependencies リソースグループプロパティは、リソースグループ内におけるネットワークアドレスリソース (論理ホスト名や共有アドレス) への非ネットワークアドレスリソースの暗黙で強力な依存関係を、RGM が強制することを指定します。これは、リソースグループ内のネットワークアドレスが「起動」に構成されるまで、リソースグループ内の非ネットワークアドレス (データサービス) リソースが、自分の Start メソッドを呼び出さないことを意味します。この Implicit_network_dependencies プロパティのデフォルト値は TRUE です。

スケーラブルリソースの実装

スケーラブルリソースは、同時に複数のノード上でオンラインになることができます。スケーラブルリソースには、Sun Cluster HA for Sun One Web Server や HA-Apache などのデータサービスがあります。

RGM は、スケーラブルリソースの実装をサポートするプロパティをいくつか提供します。

ブール型リソースタイププロパティの Failover を FALSE に設定し、一度に複数のノードでオンラインにできるリソースグループ内でリソースが構成されるようにします。

Scalable リソースプロパティは、リソースがクラスタ共有アドレス機能を使用するかどうかを決定します。スケーラブルサービスは共有アドレスリソースを使用するので (スケーラブルサービスの複数のインスタンスが単一のサービスであるかのようにクライアントに見せるため)、Scalable には TRUE を設定します。

RG_mode プロパティを使用すると、クラスタ管理者はリソースグループがフェイルオーバーまたはスケーラブルのどちらであるかを識別できます。RG_mode が SCALABLE の場合、RGM は Maximum_primaries が 1 より大きな値を持つこと、つまり同時に複数のノードがグループをマスターすることを許可します。RGM は、Failover プロパティが FALSE であるリソースが、RG_mode が SCALABLE であるリソースグループ内でインスタンス化されることを許可します。

クラスタ管理者は、スケーラブルサービスリソースが属するためのスケーラブルリソースグループを作成します。また、スケーラブルリソースが依存する共有アドレスリソースが属するためのフェイルオーバーリソースグループも別に作成します。

クラスタ管理者は、RG_dependencies リソースグループプロパティを使用して、あるノード上でリソースグループをオンラインまたはオフラインにする順番を指定します。スケーラブルリソースとそれらが依存する共有アドレスリソースは異なるリソー

スグループに属するので、この順番はスケーラブルサービスにとって重要です。スケーラブルデータサービスが起動する前に、そのネットワークアドレス (共有アドレス) リソースが構成されていることが必要です。したがって、クラスタ管理者は、スケーラブルサービスが属するリソースグループの RG_dependencies プロパティを設定して、共有アドレスリソースが属するリソースグループを組み込む必要があります。

リソースの RTR ファイルでスケーラブルプロパティを宣言した場合、RGM はそのリソースに対して、次のようなスケーラブルプロパティのセットを自動的に作成します。

Network_resources_used　このリソースによって使用される共有アドレスリソースです。このプロパティのデフォルト値は空の文字列です。したがって、クラスタ管理者は、リソースを作成するときに、スケーラブルサービスが使用する実際の共有アドレスのリストを指定する必要があります。scsetup コマンドと SunPlex Manager は、スケーラブルサービスに必要なリソースとグループを自動的に設定する機能を提供します。

Load_balancing_policy　リソースの負荷均衡ポリシーを指定します。このポリシーは RTR ファイルに明示的に設定しても、デフォルトの LB_WEIGHTED を使用してもかまいません。どちらの場合でも、クラスタ管理者はリソースを作成するときに値を変更できます (RTR ファイルで Load_balancing_policy の Tunable を NONE または FALSE に設定していない場合)。有効な値は次のとおりです。

- LB_WEIGHTED　Load_balancing_weights プロパティで設定されているウエイトに従って、さまざまなノードに負荷が分散されます。
- LB_STICKY　スケーラブルサービスの指定のクライアント (クライアントの IP アドレスで識別される) は、常に同じクラスタノードに送信されます。
- LB_STICKY_WILD　指定のクライアント (クライアントの IP アドレスで識別される) はワイルドカードスティッキーサービスの IP アドレスに接続され、送信時に使用されるポート番号とは無関係に、常に同じクラスタノードに送信されます。

	Load_balancing_policy、LB_STICKY、LB_STICKY_WILD を持つスケーラブルなサービスの場合、サービスがオンラインの状態で Load_balancing_weights を変更すると、既存のクライアントとの関連がリセットされることがあります。リセットされると、(同じクラスタ内にある) 今までサービスを行っていたノードとは別のノードが、後続のクライアント要求を処理します。 同様に、サービスの新しいインスタンスをクラスタ上で開始すると、既存のクライアントとの関連がリセットされることがあります。
Load_balancing_weights	個々のノードへ負荷を送信することを指定します。形式は *weight*@*node*,*weight*@*node* です。*weight* は、*node* に分散される負荷の相対的な割り当てを示す整数です。ノードに分散される負荷の割合は、このノードのウェイトをアクティブなインスタンスのすべてのウェイトの合計で割った値になります。たとえば、1@1,3@2 は、ノード 1 に負荷の 1/4 が割り当てられ、ノード 2 に負荷の 3/4 が割り当てられることを意味します。
Port_list	サーバーが待機するポートです。このプロパティのデフォルト値は空の文字列です。ポートのリストは RTR ファイルに指定できます。このファイルで指定しない場合、クラスタ管理者は、リソースを作成するときに、実際のポートのリストを提供する必要があります。

データサービスは、管理者がスケーラブルまたはフェイルオーバーのどちらにでも構成できるように作成できます。このためには、データサービスの RTR ファイルにおいて、Failover リソースタイププロパティと Scalable リソースプロパティの両方を FALSE に宣言します。Scalable プロパティは作成時に調整できるように指定します。

Failover プロパティが FALSE の場合、リソースはスケーラブルリソースグループに構成できます。管理者はリソースを作成するときに Scalable を TRUE に変更する、すなわちスケーラブルサービスを作成することによって、共有アドレスを有効にできます。

一方、Failover が FALSE の場合でも、管理者はリソースをフェイルオーバーリソースグループに構成して、フェイルオーバーサービスを実装できます。この場合、Scalable の値 (FALSE) は変更しません。このような偶然性に対処するために、Scalable プロパティの Validate メソッドで妥当性を検査する必要があります。Scalable が FALSE の場合、リソースがフェイルオーバーリソースグループに構成されていることを確認します。

スケーラブルリソースの詳細については、『*Sun Cluster 3.1 10/03* の概念』を参照してください。

スケーラブルサービスの妥当性検査

Scalable プロパティが TRUE であるリソースが作成または更新されるたびに、RGM は、さまざまなリソースプロパティの妥当性を検査します。プロパティが正しく構成されていない場合、RGM は作成または更新を拒否します。RGM は次の検査を行います。

- Network_resources_used プロパティは、空の文字列であってはならず、既存の共有アドレスリソースの名前を含む必要があります。スケーラブルリソースを含むリソースグループの Nodelist にあるすべてのノードは、指定した共有アドレスリソースの 1 つである NetIfList プロパティまたは AuxNodeList プロパティに存在する必要があります。
- スケーラブルリソースを含むリソースグループの RG_dependencies プロパティは、スケーラブルリソースの Network_resources_used プロパティに存在する、すべての共有アドレスリソースのリソースグループを含む必要があります。
- Port_list プロパティは、空の文字列であってはならず、ポートとプロトコル (tcp または udp) のペアのリストを含む必要があります。次に例を示します。

  ```
  Port_list=80/tcp,40/udp
  ```

データサービスの作成と検証

この節では、データサービスを作成および検証する方法について説明します。

キープアライブの使用方法

サーバー側で TCP キープアライブを有効にしておくと、サーバーはダウン時の (または、ネットワークで分割された) クライアントのリソースを浪費しません。長時間稼働するようなサーバーでこのようなリソースがクリーンアップされない場合、浪費されたリソースが無制限に大きくなり、最終的にはクライアントに障害が発生して再起動します。

クライアントサーバー通信が TCP ストリームを使用する場合、クライアントとサーバーは両方とも TCP キープアライブ機構を有効にしなければなりません。これは、非高可用性の単一サーバーの場合でも適用されます。

ほかにも、キープアライブ機構を持っている接続指向のプロトコルは存在します。

クライアント側で TCP キープアライブを有効にしておくと、ある物理ホストから別の物理ホストに論理ホストがフェイルオーバーまたはスイッチオーバーしたとき、接続の切断がクライアントに通知されます。このようなネットワークアドレスリソースの転送 (フェイルオーバーやスイッチオーバー) が発生すると、TCP 接続が切断されます。しかし、クライアント側で TCP キープアライブを有効にしておかなければ、接続が休止したとき、必ずしも接続の切断はクライアントに通知されません。

たとえば、クライアントが、実行に時間がかかる要求に対するサーバーからの応答を待っており、また、クライアントの要求メッセージがすでにサーバーに到着しており、TCP 層で認識されているものと想定します。この状況では、クライアントの TCP モジュールは要求を再転送し続ける必要はないので、クライアントアプリケーションはブロックされて、要求に対する応答を待ちます。

TCP キープアライブ機構は必ずしもあらゆる限界状況に対応できるわけではないので、クライアントアプリケーションは、可能であれば、TCP キープアライブ機構に加えて、独自の定期的なキープアライブをアプリケーションレベルで実行する必要があります。アプリケーションレベルのキープアライブ機構を使用するには、通常、クライアントサーバー型プロトコルが NULL 操作、または、少なくとも効率的な読み取り専用操作 (状態操作など) をサポートする必要があります。

HA データサービスの検証

この節では、高可用性環境におけるデータサービスの実装を検証する方法について説明します。この検証は一例であり、完全ではないことに注意してください。実際に稼働させるマシンに影響を与えないように、検証時は、検証用の Sun Cluster 構成にアクセスする必要があります。

リソースグループが物理ホスト間で移動するような場合を想定して、HA データサービスが適切に動作するかどうかを検証します。たとえば、システムがクラッシュした場合や、`scswitch` コマンドを使用した場合です。また、このような場合にクライアントマシンがサービスを受け続けられるかどうかも検証します。

メソッドの呼び出し回数への非依存性を検証します。たとえば、各メソッドを一時的に、元のメソッドを 2 回以上呼び出す短いシェルスクリプトに変更します。

リソース間の依存関係の調節

あるクライアントサーバーのデータサービスが、クライアントからの要求を満たすために、別のクライアントサーバーのデータサービスに要求を行うことがあります。このように、データサービス A が自分のサービスを提供するために、データサービス B にそのサービスを提供してもらう場合、データサービス A はデータサービス B に依存

していると言います。この要件を満たすために、Sun Cluster では、リソースグループ内でリソースの依存関係を構築できます。依存関係は、Sun Cluster がデータサービスを起動および停止する順番に影響します。詳細については、`scrgadm(1M)` のマニュアルページを参照してください。

あるリソースタイプのリソースが別のリソースタイプのリソースに依存する場合、データサービス開発者は、リソースとリソースグループを適切に構成するようにユーザーに指示するか、これらを正しく構成するスクリプトまたはツールを提供する必要があります。依存するリソースを依存されるリソースと同じノード上で実行する必要がある場合、両方のリソースを同じリソースグループ内で構成する必要があります。

明示的なリソースの依存関係を使用するか、このような依存関係を省略して、HA データサービス独自のコードで別のデータサービスの可用性をポーリングするかを決定します。依存するリソースと依存されるリソースが異なるノード上で動作できる場合は、これらのリソースを異なるリソースグループ内で構成します。この場合、グループ間にはリソースの依存関係を構築できないため、ポーリングが必要です。

データサービスによっては、データを自分自身で直接格納せず、別のバックエンドデータサービスに依頼して、すべてのデータを格納してもらうものもあります。このようなデータサービスは、すべての読み取り要求と更新要求をバックエンドデータサービスへの呼び出しに変換します。たとえば、すべてのデータを SQL データベース (Oracle など) に格納するようなクライアントサーバー型のアポイントメントカレンダサービスの場合、このサービスは独自のクライアントサーバー型ネットワークプロトコルを持っています。たとえば、RPC 仕様言語 (ONC RPC など) を使用するプロトコルを定義している場合があります。

Sun Cluster 環境では、HA-ORACLE を使用してバックエンド Oracle データベースを高可用性にできます。つまり、アポイントメントカレンダデーモンを起動および停止する簡単なメソッドを作成できます。エンドユーザーは Sun Cluster でアポイントメントカレンダのリソースタイプを登録できます。

アポイントメントカレンダアプリケーションが Oracle データベースと同じノード上で動作する必要がある場合、エンドユーザーは、HA-ORACLE リソースと同じリソースグループ内でアポイントメントカレンダリソースを構築して、アポイントメントカレンダリソースを HA-ORACLE リソースに依存するようにします。この依存関係を指定するには、`scrgadm` の `Resource_dependencies` プロパティを使用します。

アポイントメントカレンダリソースが HA-ORACLE リソースとは別のノード上で動作できる場合、エンドユーザーはこれらのリソースを 2 つの異なるリソースグループ内で構成します。カレンダリソースグループのリソースグループ依存関係を、Oracle リソースグループ上で構築することもできます。しかし、リソースグループ依存関係が有効になるのは、両方のリソースグループが同時に同じノード上で起動または停止されたときだけです。したがって、カレンダデータサービスデーモンは、起動後、Oracle データベースが利用可能になるまで、ポーリングして待機します。この場合、通常、カレンダリソースタイプの `Start` メソッドは単に成功を戻すだけです。これは、`Start` メソッドが無限にブロックされると、そのリソースグループがビジー状態になり、それ以降、リソースグループで状態の変化 (編集、フェイルオーバー、ス

イッチオーバーなど) が行われなくなるためです。しかし、カレンダリソースの `Start` メソッドがタイムアウトまたは非ゼロで終了すると、Oracle データベースが利用できない間、リソースグループが複数のノード間でやりとりを無限に繰り返す可能性があります。

第 3 章

リソースタイプの更新

この章では、リソースタイプの開発者がリソースタイプの更新や移行を行うために必要な情報を提供します。

概要

システム管理者は、既存のリソースタイプの新しいバージョンをインストールおよび登録できなければなりません。これは、リソースを削除したり作成し直したりすることなく、複数のバージョンのリソースタイプを登録したり、既存のリソースを新しいバージョンのリソースタイプに移行したりする必要があるからです。リソース開発者は、リソースタイプのアップグレードや移行の要件を把握しておく必要があります。

アップグレードを念頭に置いたリソースタイプの開発を「アップグレード対応」と呼びます。

新しいバージョンのリソースタイプは、次の点で前のバージョンとは異なっている可能性があります。

- リソースタイププロパティの属性

- 標準プロパティ、拡張プロパティを含む宣言済みリソースプロパティ
- リソースプロパティの属性 (default、min、max、arraymin、arraymax) または tunable 属性
- 宣言済みメソッド
- メソッドやモニターの実装

リソースタイプ開発者は、既存のリソースを新しいバージョンへ移行するタイミングを次の Tunable 属性のオプションによって特定します。制約の小さいものから順に、次のようなオプションがあります。

- 任意の時点 (Anytime)
- リソースが監視されていないとき (When_unmonitored)
- リソースがオフラインのとき (When_offline)
- リソースが無効のとき (When_disabled)
- リソースグループが管理されていないとき (When_unmanaged)
- 作成時 (At_creation)

各オプションについては、62 ページの「Type_version リソースプロパティ」を参照してください。

注 – この章では、アップグレード手順の記述で常に scrgadm コマンドを使用します。 これは、管理者が scrgadm コマンドしか使用できないということではありません。GUI、scsetup コマンドなども使用できます。

リソースタイプ登録ファイル

リソースタイプ名

リソースタイプ名は、RTR ファイルに指定されたプロパティ、*Vendor_id*、*Resource_type*、*RT_version* の 3 つで構成されます。scrgadm コマンドは、ピリオドとコロンを区切り文字として使用し、リソースタイプ名を作成します。

```
vendor_id.resource_type:rt_version
```

Vendor_id 接頭辞では、同一名でベンダーの異なる 2 つの登録ファイルを識別できます。*RT_version* では、同じベースリソースタイプの複数のバージョン (アップグレード) を識別できます。重複を防ぐため、*Vendor_id* には、リソース型の作成元の会社のストックシンボルを使用することをお勧めします。

RT_version 文字列には、空白文字、タブ、スラッシュ (/)、バックスラッシュ(\)、アスタリスク (*)、疑問符 (?)、コンマ (,)、セミコロン (;)、左角括弧 ([)、右角括弧 (]) は使用できません。

`RT_Version` プロパティは、Sun Cluster 3.0 まではオプションでしたが、Sun Cluster 3.1 以降では必須です。

完全名は、次のコマンドで取得できます。

```
scha_resource_get -O Type -R resource_name -G resource_group_name
```

Sun Cluster 3.1 以前に登録されたリソースタイプ名は次の形式をとります。

```
vendor_id.resource_type
```

ディレクティブ

アップグレード対応リソースタイプの RTR ファイルには、次の形式の `#$upgrade` ディレクティブが必要です。ほかにディレクティブがある場合は、このディレクティブの後ろに続きます。

```
#$upgrade_from version tunability
```

`upgrade_from` ディレクティブは、文字列 `#$upgrade_from`、 `RT_Version`、リソースの Tunable 属性の制約で構成されます。アップグレードを行う前のリソースタイプにバージョンがない場合、`RT_Version` は、以下の例の最後の行のように空文字列として指定されます。

```
#$upgrade_from   "1.1"   when_offline
#$upgrade_from   "1.2"   when_offline
#$upgrade_from   "1.3"   when_offline
#$upgrade_from   "2.0"   when_unmonitored
#$upgrade_from   "2.1"   anytime
#$upgrade_from   ""      when_unmanaged
```

システム管理者がリソース `Type_version` を変更しようとすると、RGM によってこれらの制約が課されます。現在のリソースタイプのバージョンがリストに表示されない場合、Tunable 属性は `When_unmanaged` になります。

これらのディレクティブは、RTR ファイル内のリソースタイププロパティ宣言とリソース宣言セクションの間に指定されます。`rt_reg`(4) を参照してください。

RTR ファイル内の `RT_Version` の変更

RTR ファイルの内容が変更されたときは、必ず RTR ファイル内の `RT_Version` 文字列を変更します。このプロパティの値は、どちらのリソースタイプが新しく、どちらが古いかをはっきりと示す必要があります。RTR ファイルに変更が加えられていなければ、`RT_Version` 文字列を変更する必要はありません。

以前のバージョンの Sun Cluster のリソースタイプ名

Sun Cluster 3.0 のリソースタイプ名には、バージョン接尾辞がありません。

```
vendor_id.resource_name
```

元々 Sun Cluster 3.0 で登録されたリソースタイプは、クラスタリングソフトウェアを Sun Cluster 3.1 にアップグレードしたあとも、この形式の名前を持ちます。RTR ファイルを Sun Cluster 3.1 ソフトウェアで登録した場合でも、RTR ファイル内に `#$upgrade` ディレクティブの指定がないリソースタイプは、バージョン接尾辞のない Sun Cluster 3.0 の形式の名前を付与されます。

Sun Cluster 3.0 では、`#$upgrade` ディレクティブや `#$upgrade_from` ディレクティブを使った RTR ファイルの登録は可能ですが、既存のリソースの新しいリソースタイプへの移行はサポートされません。

Type_version リソースプロパティ

標準リソースプロパティ `Type_version` は、リソースタイプの `RT_Version` プロパティを格納します。このプロパティは、RTR ファイル内には指定されません。システム管理者は、次のコマンドを使って、Type_version プロパティを編集します。

```
scrgadm -c -j resource -y Type_version=new_version
```

このプロパティの Tunable 属性は、次の項目によって決まります。

- 現在のリソースタイプのバージョン
- RTR ファイル内の `#$upgrade_from` ディレクティブ

`#$upgrade_from` ディレクティブの値は次のとおりです。

`Anytime`
: リソースをいつアップグレードしてもよい場合。リソースを完全にオンラインにできます。

`When_unmonitored`
: 新しいリソースタイプのバージョンの `Update`、 `Stop`、`Monitor_check`、および `Postnet_stop` メソッドが古いリソースタイプのバージョンの起動メソッド (`Prenet_stop` および `Start`) と互換することがわかっている場合と、新しいリソースタイプのバージョンの `Fini` メッセージが古いバージョンの `Init` メソッドと互換することがわかっている場合。アップグレード前にリソース監視プログラムを停止するだけですみます。

`When_offline`
: 新しいリソースタイプのバージョンの `Update`、`Stop`、`Monitor_check`、または `Postnet_stop` メソッドと古いリソースタイプのバージョンの起動メソッド

(Prenet_stop および Start) に互換性がないが、これらのメソッドと古いリソースタイプのバージョンの Init メソッドには互換性があることがわかっている場合、タイプのアップグレード時にリソースはオフラインにする必要があります。

When_disabled
: When_offline と同様です。ただし、より厳しい条件で、リソースが無効化されている場合になります。

When_unmanaged
: 新しいリソースタイプのバージョンの Fini メソッドが古いバージョンの Init メソッドと互換しないことがわかっている場合。リソースのアップグレード前に既存のリソースグループを管理されていない状態にする必要があります。

At_creation
: 新しいリソースタイプのバージョンにアップグレードできないリソースの場合。新しいバージョンの新しいリソースしか作成できません。

  Tunable 属性が At_creation の場合、リソースタイプ開発者は既存のリソースを新しいタイプに移行することを禁止できます。この場合、システム管理者は、リソースを削除して作成し直す必要があります。これは、リソースのバージョンが作成時にだけ設定されるという宣言と同じことです。

リソースを別のバージョンへ移行

システム管理者がリソースの Type_version プロパティを編集すると、既存のリソースは新しいリソースタイプバージョンを取得します。これは、その他のリソースプロパティの編集時と同じ規則に従います。ただし、一部の情報が現在のバージョンではなく新しいリソースタイプから取得されるという点を除きます。

- min、max、arraymin、arraymax、デフォルト、Tunable 属性など、すべてのプロパティのリソースプロパティ属性は、新しいリソースタイプのバージョンから取得されます。
- Type_version プロパティに適用されるTunable 属性は、RTR ファイル内の #$upgrade_from ディレクティブと、既存のリソースのリソースタイプの RT_version プロパティから取得されます。Tunable 属性は、property_attributes(5) のマニュアルページの説明とは異なります。
- 新しいリソースタイプのバージョンの Validate メソッドが適用されます。このため、プロパティ属性は、新しいリソースタイプで有効です。既存のリソースプロパティ属性が新しいリソースタイプのバージョンの妥当性検査の条件を満たしていない場合、システム管理者は、scrgadm コマンドに、こうしたプロパティの有効な値を指定する必要があります。この手続きは、新しいリソースタイプのバージョンが、以前のバージョンでは宣言されていなかったプロパティを使用し、デフォルト値がない場合に行われます。既存のリソースが、新しいリソースタイプのバージョンでは無効なプロパティ値を持っている場合にも、この手続きが行われます。

- 古いバージョンのリソースタイプで宣言されたリソースプロパティが新しいバージョンでは宣言されないことがあります。こうしたリソースを新しいバージョンに移行すると、リソースからプロパティが削除されます。

注 – `Validate` メソッドは、リソースの新しい `Type_version` (`Validate` コマンド行に渡される) だけでなく現在の `Type_version` も照会できます (`scha_resource_get` を使用)。このため、 サポートされないバージョンのアップグレードを `Validate` によって禁止できます。

リソースタイプのアップグレードとダウングレード

リソースタイプのアップグレードおよび移行の詳細については、『*Sun Cluster 3.1* データサービスの計画と管理』の「リソースタイプのアップグレード」を参照してください。

▼ リソースタイプをアップグレードする方法

1. 新しいリソースタイプのアップグレード手順の説明を読み、リソースタイプの変更とリソースの **Tunable** 属性の制約を確認します。
2. すべてのクラスタノードに、リソースタイプアップグレードパッケージをインストールします。

   新しいリソースタイプパッケージのインストールは、段階的に行うことをお勧めします。ノードが非クラスタモードで起動している間に `pkgadd` を実行します。

   クラスタモードのノードに新しいリソースタイプパッケージをインストールする場合は、条件に合わせて異なった方法を使用します。

   - リソースタイプパッケージのインストールによってメソッドコードが変更されず、モニターだけが更新される場合は、インストール中にそのタイプのすべてのリソース上で監視を停止する必要があります。
   - リソースタイプパッケージのインストールによってメソッドとモニターの両方のコードが変更されない場合は、ディスク上に新しい RTR ファイルが 追加されるだけなので、インストール中にリソース上での監視を停止する必要はありません。
3. アップグレードの **RTR** ファイルを参照し、**`scrgadm`** または同等のコマンドを使って新しいリソースタイプのバージョンを登録します。

   RGM により、次の形式の新しいリソースタイプが作成されます。

```
vendor_id.resource_type:version
```

4. リソースタイプのアップグレードがノードのサブセットにだけインストールされる場合は、新しいリソースタイプの **Installed_nodes** プロパティに実際のインストール先ノードを設定する必要があります。

   リソースが新しいタイプ (新しく作成された、または更新された) を取得するとき、RGM は、リソースグループ nodelist がリソースタイプの Installed_nodes リストのサブセットであることを要求します。

   ```
   scrgadm -c -t resource_type -h installed_node_list
   ```

5. 以前にタイプがアップグレードされ、これからそのタイプに移行する各リソースに対して、**scswitch** を呼び出し、そのリソース自体またはそのリソースグループを、アップグレード手順に示されている適切な状態に変更します。

6. 以前にタイプがアップグレードされ、これからそのタイプに移行する各リソースを編集し、**Type_version** プロパティに新しいバージョンを設定します。

   ```
   scrgadm -c -j resource -y Type_version=new_version
   ```

   必要に応じて、同じコマンドを使って、同じリソースのその他のプロパティに適切な値を設定します。

7. 手順 **5** で呼び出したコマンドを逆方向に実行すると、リソースまたはリソースグループの以前の状態を回復できます。

▼ 古いバージョンのリソースタイプにダウングレードする方法

リソースをダウングレードして古いバージョンのリソースタイプにすることができます。古いバージョンのリソースタイプにダウングレードする場合は、新しいバージョンのリソースタイプにアップグレードする場合よりも条件が厳しくなります。まず、リソースグループの管理を解除する必要があります。アップグレード可能なバージョンのリソースタイプにしかダウングレードできないということにも注意してください。アップグレード可能なバージョンは scrgadm -p コマンドを使用して確認できます。アップグレード可能なバージョンの場合、接尾辞 *version* が表示されます。

1. ダウングレードするリソースを含んでいるリソースグループをオフラインに切り替えます。

   ```
   scswitch -F -g resource_group
   ```

2. ダウングレードするリソースと、そのリソースグループ内のすべてのリソースを無効にします。

   ```
   scswitch -n -j resource_to_downgrade
   scswitch -n -j resource1
   scswitch -n -j resource2
   scswitch -n -j resource3
   ...
   ```

注 – リソースの無効化は、依存性の高いもの (アプリケーションリソース) から開始し、もっとも依存性の低いもの (ネットワークアドレスリソース) で終了するように行なってください。

3. リソースグループを管理されていない状態にします。

   scswitch -u -g *resource_group*

4. ダウングレード後のリソースタイプバージョンとする古いリソースバージョンがクラスタ内にまだ登録されているかどうか確認します。
   - 登録されている場合は、次の手順に進みます。
   - 登録されていない場合は、希望する旧バージョンを登録し直します。

     scrgadm -a -t *resource_type_name*

5. 希望する旧バージョンを **Type_version** に指定し、リソースをダウングレードします。

   scrgadm -c -j *resource_to_downgrade* **-y Type_version=***old_version*

   必要に応じて、同じコマンドを使って、同じリソースのその他のプロパティに適切な値を設定します。

6. ダウングレードしたリソースを含んでいるリソースグループを管理状態にし、すべてのリソースを有効にしたあと、このグループをオンラインに切り替えます。

   scswitch -Z -g *resource_group*

デフォルトのプロパティ値

RGM は、システム管理者によって明示的に設定されなかったためリソースの CCR (クラスタ構成リポジトリ) に格納されていないデフォルトのプロパティなど、すべてのリソースを格納します。RGM は、CCR からリソースが読み込まれるとき、欠落したリソースプロパティのデフォルト値をそのリソースタイプから取得します。リソースタイプにデフォルト値が定義されていない場合は、システム定義のデフォルト値を使用します。アップグレードされたリソースタイプは、この方法により、新しいプロパティや既存のプロパティの新しいデフォルト値を定義することができます。

リソースプロパティが変更された場合、RGM は編集コマンドで指定されたプロパティを CCR に格納します。

リソースタイプのアップグレードされたバージョンが、デフォルトのプロパティの新しいデフォルト値を宣言する場合、新しいデフォルト値は既存のリソースから継承されます。この手続きは、プロパティが作成時または無効化されている場合だ

け「Tunable」と宣言されている場合でも変わりません。新しいデフォルトのアプリケーションで、Stop、 Postnet_stop、Fini などのメソッドが失敗する場合、リソースタイプの実装者は、アップグレード時のリソースの状態を適切に制限する必要があります。具体的には、Type_version プロパティの Tunable 属性を制限します。

新しいリソースタイプのバージョンの Validate メソッドでは、既存のプロパティ属性が適切であるかどうかを検査できます。既存のプロパティ属性が適切でない場合、システム管理者は、Type_version プロパティを編集するコマンドを使って、新しいリソースタイプのバージョンにリソースをアップグレードできるように、プロパティの属性値を変更できます。

注 – Sun Cluster 3.0 で作成されたリソースは、新しいバージョンに移行しても、リソースタイプからデフォルトのプロパティ属性を継承しません。これは、これらのリソースのデフォルトのプロパティ値が CCR に格納されているからです。

リソースタイプ開発者の文書

リソースタイプ開発者は、新しいリソースに関する文書を提供する必要があります。必須記載事項は次のとおりです。

- プロパティの追加、変更、削除の説明
- プロパティを新しい要件に準拠させた方法の説明
- リソースの Tunable 属性の制約
- 新しいデフォルトプロパティ属性
- Type_version プロパティの編集に使用するコマンドを使って、既存のリソースプロパティを適切な値に変更し、リソースを新しいリソースタイプのバージョンにアップグレードできることの通知 (システム管理者向けの情報)

リソースタイプ名とリソースタイプモニターの実装

Sun Cluster 3.0 でもアップグレード対応のリソースタイプを登録することはできますが、これらの名前はバージョン接尾辞なしで CCR に記録されています。Sun Cluster 3.0 と3.1 を両方とも正常に実行するには、このリソースタイプのモニターが両方の命名規則を処理できなければなりません。

```
vendor_id.resource_name:version
vendor_id.resource_name
```

モニターコードは、次のコマンドと同等のコマンドを実行することにより、適切な名前を判断できます。

```
scha_resourcetype_get -O RT_VERSION -T VEND.myrt
scha_resourcetype_get -O RT_VERSION -T VEND.myrt:vers
```

次に、`vers` と出力値を比較します。同じリソースタイプの同じバージョンを別の名前で 2 回以上登録することはできません。したがって、`vers` の特定の値に対して成功するのは、上記のコマンドのいずれか一方のみとなります。

アプリケーションのアップグレード

アプリケーションコードのアップグレードは、いくつかの共通点はあるものの、エージェントコードのアップグレードとは大きく異なっています。アプリケーションのアップグレードでは、リソースタイプのアップグレードが必要な場合とそうでない場合があります。

リソースタイプのアップグレード例

以下では、リソースタイプのインストールとアップグレードの例をいくつか紹介します。Tunable 属性とパッケージ情報は、リソースタイプ実装の変更に基づいて選択されています。Tunable 属性は、リソースを新しいリソースタイプに移行するときに適用されます。

すべての例は、次の条件に従うものとします。

- リソースタイプは Solaris 形式のパッケージで配布されています。`pkgadd(1M)` および `pkgrm(1M)` のマニュアルページを参照してください。
- リソースタイプの以前のバージョンは 1 つしかないので、新しい RTR ファイル内には `#$upgrade_from` ディレクティブが 1 つしかありません。
- RGM がディスクから削除されたメソッドを呼び出すことができる場合、インストールによってメソッドが削除または上書きされることはありません。
- 特に注記がない場合、新しいメソッドと古いメソッドには互換性があります。
- リソースおよびリソースグループは、インストールまたは移行前に正しい `scswitch(1M)` コマンド (または同等のコマンド) により適切な状態に設定されます。次に示すのは、リソースグループを管理対象外に設定する例です。

```
scswitch -M -n -j resource
scswitch -n -j resource
scswitch -F -g resource_group
scswitch -u -g resource_group
```

- リソースタイプの登録は次のコマンドで行います。

```
scrgadm -a -t resource_type -f path_to_RTR_file
```

- リソースの移行は次のコマンドで行います。

```
scrgadm -c -j resource -y Type_version=version \
           -y property=value \
           -x property=value ...
```

- リソースおよびリソースグループは、移行後に適切な `scswitch` (1M) コマンド (または同等のコマンド) により以前の状態に戻されます。

```
scswitch -M -e -j resource
scswitch -e -j resource
scswitch -o -g resource_group
scswitch -Z -g resource_group
```

場合によっては、リソースタイプ開発者は、例で使用されているものより制限の厳しい Tunable 属性の値を指定する必要があります。Tunable 属性の値は、リソースタイプ実装の変更内容によって決定します。リソースタイプ開発者は、例で使用されている Solaris 形式のパッケージ以外のパッケージスキーマを選択することもできます。

表 3–1 リソースタイプのアップグレード例

変更の種類	**Tunable** 属性	パッケージ	手続き
プロパティの変更は、RTR ファイルのみに加えられました。	`Anytime`	新しい RTR ファイルを配布するだけです。	新しい RTR ファイルの `pkgadd` をすべてのノード上で実行します。 新しいリソースタイプを登録します。 リソースを移行します。
メソッドが更新されました。	`Anytime`	更新されたメソッドを古いメソッドとは異なったパスに配置します。	更新されたメソッドの `pkgadd` をすべてのノード上で実行します。 新しいリソースタイプを登録します。 リソースを移行します。

表 **3–1** リソースタイプのアップグレード例　(続き)

変更の種類	**Tunable** 属性	パッケージ	手続き
新しいモニタープログラムです。	`When_unmonitored`	以前のバージョンのモニターを上書きするだけです。	監視を無効化します。 新しい監視プログラムの `pkgadd` をすべてのノード上で実行します。 新しいリソースタイプを登録します。 リソースを移行します。 監視を有効化します。
メソッドが更新されました。新しい `Update`/`Stop` メソッドと古い `Start` メソッドには互換性がありません。	`When_offline`	更新されたメソッドを古いメソッドとは異なったパスに配置します。	更新されたメソッドの `pkgadd` をすべてのノード上で実行します。 新しいリソースタイプを登録します。 リソースをオフラインにします。 リソースを移行します。 リソースをオンラインにします。
メソッドが更新され、RTR ファイルに新しいプロパティが追加されました。新しいメソッドには新しいプロパティが必要です。リソースの所属リソースグループをオンラインのまま保持しながらリソースがオンラインになることを防ぐ必要があります。このためには、リソースグループをノード上でオフラインの状態からオンラインの状態に変更します。	`When_disabled`	以前のバージョンのメソッドを上書きします。	リソースを無効にします。 各ノード上で次の処理を行います。 ■ クラスタからノードを削除 ■ 更新されたメソッドの `pkgrm`/`pkgadd` の実行 ■ クラスタにノードを戻す 新しいリソースタイプを登録します。 リソースを移行します。 リソースを有効にします。

表 3–1 リソースタイプのアップグレード例　(続き)

変更の種類	Tunable 属性	パッケージ	手続き
メソッドが更新され、RTR ファイルに新しいプロパティが追加されました。新しいメソッドは新しいプロパティを必要としません。	Anytime	以前のバージョンのメソッドを上書きします。	各ノード上で次の処理を行います。 ■ クラスタからノードを削除 ■ 更新されたメソッドの pkgrm/pkgadd の実行 ■ クラスタにノードを戻す この手続きの間に、RGM は新しいメソッドを呼び出します。これは、まだ移行が完了しておらず、新しいプロパティが構成されていない場合でも変わりません。新しいメソッドは、新しいプロパティなしで正常に機能しなければなりません。 新しいリソースタイプを登録します。 リソースを移行します。
メソッドが更新されました。新しい Fini メソッドと古い Init メソッドには互換性がありません。	When_unmanaged	更新されたメソッドを古いメソッドとは異なったパスに配置します。	リソースの所属リソースグループを管理対象外にします。 更新されたメソッドの pkgadd をすべてのノード上で実行します。 リソースタイプを登録します。 リソースを移行します。 リソースの所属リソースグループを管理対象にします。
メソッドが更新されました。RTR ファイルは変更されていません。	該当しません。RTR ファイルは変更されていません。	以前のバージョンのメソッドを上書きします。	各ノード上で次の処理を行います。 ■ クラスタからノードを削除 ■ 更新されたメソッドの pkgadd を実行 ■ クラスタにノードを戻す RTR ファイルが変更されなかったので、リソースを登録または移行する必要はありません。

リソースタイプパッケージのインストール要件

新しいリソースタイプをインストールするときは、次の2 つの要件が満たされていなければなりません。

- リソースタイプが登録されている場合、ディスク上の RTR ファイルにアクセスできなければなりません。
- 新しいタイプのリソースを作成した場合、新しいタイプの宣言済みメソッドのパス名および監視プログラムがディスク上に存在し、実行できなければなりません。リソースが使用されている間は、以前のメソッドおよび監視プログラムを定位置に確保しておく必要があります。

  最適のパッケージを決定するため、リソースタイプの実装者は、次のことを考慮する必要があります。

  - RTR ファイルが変更されたか
  - プロパティのデフォルト値または tunable 属性が変更されたか
  - プロパティの `min` または `max` 値が変更されたか
  - アップグレードによってプロパティが追加されたか、または削除されたか
  - メソッドコードが変更されたか
  - モニターコードが変更されたか
  - 新しいメソッドまたはモニターコードが以前のバージョンのものと互換するか

RTR ファイルの変更前に認識しておくべき事項

リソースタイプをアップグレードしても、新しいメソッドまたはモニターコードが呼び出されない場合がありますたとえば、リソースプロパティのデフォルト値または tunable 属性の値だけが変更される場合があります。メソッドコードは変更されないので、読み取り可能な RTR ファイルの有効なパス名を指定するだけでアップグレードをインストールできます。

古いリソースタイプを登録し直す必要がない場合、新しい RTR ファイルによって以前のバージョンのものが上書きされます。それ以外の場合、新しい RTR ファイルは新しいパス名で配置できます。

アップグレードによってプロパティのデフォルト値または tunable 属性が変更される場合、移行時に新しいバージョンの `Validate` メソッドで既存のプロパティ属性が新しいリソースタイプでも有効かどうかを検査できます。アップグレードによってプロパティの `min`、`max` または `type` 属性が変更される場合、移行時に `scrgadm` コマンドでこれらの制約が自動的に検査されます。

アップグレード文書には、新しいデフォルトのプロパティ値をすべて記載する必要があります。この文書では、システム管理者に、Type_version コマンドを編集するコマンドを使って、新しいリソースタイプのバージョンにプロパティをアップグレードできるように、既存のリソースプロパティの値を変更できることを通知します。

アップグレードによってプロパティが追加または削除される場合、コールバックメソッドまたはモニターコードの一部を変更しなければならないことがあります。

モニターコードの変更

更新後のリソースタイプでモニターコードだけが変更される場合、パッケージのインストールによってモニターのバイナリが上書きされます。文書には、システム管理者向けの情報として、新しいパッケージをインストールする前に監視を一時停止する指示を記述します。

メソッドコードの変更

更新後のリソースタイプでメソッドコードだけが変更される場合、新しいメソッドコードが以前のバージョンのものと互換するかどうかを確認する必要があります。これにより、新しいメソッドコードを新しいパス名で格納するか、古いメソッドを上書きするかが決定します。

新しい Stop、Postnet_stop および Fini メソッドが宣言されていて、古いバージョンの Start 、Prenet_stop または Init メソッドによって初期化または起動されたリソースに適用できる場合、新しいメソッドで古いメソッドを上書きできます。

新しいメソッドコードが以前のバージョンのものと互換しない場合、アップグレードされたリソースタイプに移行する前に、古いバージョンのメソッドを使ってリソースを停止または構成を解除する必要があります。新しいメソッドが古いメソッドを上書きする場合、リソースタイプをアップグレードする前に、そのタイプのすべてのリソースを停止 (場合によっては、さらに管理対象外に設定) しなければならないことがあります。新しいメソッドが古いものと別の場所に格納されていて、両方に同時にアクセスできる場合は、後方互換性がなくても、新しいリソースタイプのバージョンをインストールし、リソースを 1 つずつアップグレードすることができます。

新しいメソッドに後方互換性がある場合でも、その他のリソースが古いメソッドを使用し続けている間は、リソースを 1 つずつアップグレードして、新しいメソッドを使用できるようにする必要があります。また、新しいメソッドは、古いメソッドを上書きすることがないように別のディレクトリに格納する必要があります。

個々のリソースタイプのバージョンのメソッドを別々のディレクトリに格納すると、新しいバージョンで問題が発生したとき、元のリソースタイプのバージョンに切り替える手続きが簡単であるという点で有利です。

引き続きサポートされている以前のバージョンをすべてパッケージに含めるという方法もあります。この方法では、古いメソッドのパスを上書きまたは削除することなく、新しいパッケージのバージョンで古いバージョンを置き換えることができます。サポートされる以前のバージョンの数は、リソースタイプ開発者が決定します。

注 – メソッド、または現在クラスタ内にあるノード上の `pkgrm/ pkgadd` メソッドの上書きはお勧めしません。RGM がディスク上のアクセス不能なメソッドを呼び出そうとすると、予測できない結果を招くことがあります。実行中のメソッドのバイナリの削除または置き換えでも、予測できない結果を招く可能性があります。

第 4 章

Resource Management API リファレンス

この章では、Resource Management API (RMAPI) を構成するアクセス関数やコールバックメソッドに関する情報を提供します。ここでは、各関数やメソッドについて簡単に説明します。詳細については、RMAPI のそれぞれのマニュアルページを参照してください。

この章の内容は、次のとおりです。

- 76 ページの「RMAPI アクセスメソッド」 – シェルスクリプトコマンドと C 関数
  - `scha_resource_get`(1HA)、 `scha_resource_close`(3HA)、`scha_resource_get(3HA)`、`scha_resource_open`(3HA)
  - `scha_resource_setstatus`(1HA)、 `scha_resource_setstatus`(3HA)
  - `scha_resourcetype_get`(1HA)、 `scha_resourcetype_close`(3HA)、`scha_resourcetype_get(3HA)`、`scha_resourcetype_open`(3HA)
  - `scha_resourcegroup_get`(1HA)、 `scha_resourcegroup_get`(3HA)、`scha_resourcegroup_close`(3HA)、`scha_resourcegroup_open`(3HA)
  - `scha_control` (1HA)、`scha_control` (3HA)
  - `scha_cluster_get`(1HA)、 `scha_cluster_close`(3HA)、`scha_cluster_get`(3HA)、`scha_cluster_open`(3HA)
  - `scha_cluster_getlogfacility`(3HA)
  - `scha_cluster_getnodename`(3HA)
  - `scha_strerror` (3HA)
- 81 ページの「RMAPI コールバックメソッド」 – `rt_callbacks(1HA)` のマニュアルページで説明されている内容
  - `Start`
  - `Stop`
  - `Init`
  - `Fini`
  - `Boot`
  - `Prenet_start`

- Postnet_stop
- Monitor_start
- Monitor_stop
- Monitor_check
- Update
- ValidateS

RMAPI アクセスメソッド

API は、リソース、リソースタイプ、リソースグループのプロパティ、および他のクラスタ情報にアクセスするための関数を提供します。これらの関数はシェルコマンドと C 関数の両方の形で提供されるため、リソースタイプの開発者はシェルスクリプトまたは C プログラムのどちらでも制御プログラムを実装できます。

RMAPI シェルコマンド

シェルコマンドは、クラスタの RGM によって制御されるサービスを表すリソースタイプのコールバックメソッドを、シェルスクリプトで実装するときに使用します。このコマンドを使用すると、次のことを行えます。

- リソース、リソースタイプ、リソースグループ、クラスタについての情報にアクセスする。
- モニターと併用し、リソースの Status プロパティと Status_msg プロパティを設定する。
- リソースグループの再起動と再配置を要求する。

注 – この節では、シェルコマンドについて簡単に説明します。詳細については、個々のマニュアルページのセクション 1HA を参照してください。特に注記しないかぎり、各コマンドと同じ名前のマニュアルページがあります。

RMAPI リソースコマンド

以下のコマンドを使用すると、リソースについての情報にアクセスしたり、リソースの Status プロパティや Status_msg プロパティを設定できます。

scha_resource_get
RGM の制御下のリソースまたはリソースタイプに関する情報にアクセスできます。このコマンドは、scha_resource_get() 関数と同じ情報を提供します。

`scha_resource_setstatus`
RGM の制御下のリソースの `Status` および `Status_msg` プロパティを設定します。このコマンドはリソースのモニターによって使用され、モニターから見たリソースの状態を反映します。このコマンドは、C 関数 `scha_resource_setstatus()` と同じ機能を提供します。

注 – `scha_resource_setstatus()` はリソースモニター専用の関数ですが、任意のプログラムから呼び出すことができます。

リソースタイプコマンド

このコマンドは、RGM に登録されているリソースタイプについての情報にアクセスします。

`scha_resourcetype_get`
このコマンドは、C 関数 `scha_resourcegroup_get()` と同じ機能を提供します。

リソースグループコマンド

以下のコマンドを使用すると、リソースグループについての情報にアクセスしたり、リソースグループを再起動できます。

`scha_resourcegroup_get`
RGM の制御下のリソースグループに関する情報にアクセスできます。このコマンドは、C 関数 `scha_resourcegroup_get()` と同じ機能を提供します。

`scha_control`
RGM の制御下のリソースグループの再起動、または別のノードへの再配置を要求します。このコマンドは、C 関数 `scha_control()` と同じ機能を提供します。

クラスタコマンド

このコマンドは、クラスタについての情報 (ノード名、ノード ID、ノードの状態、クラスタ名、リソースグループなど) にアクセスします。

`scha_cluster_get`
このコマンドは、C 関数 `scha_cluster_get()` と同じ情報を提供します。

C 関数

C 関数は、クラスタの RGM によって制御されるサービスを表すリソースタイプのコールバックメソッドを、C プログラムで実装するときに使用します。この関数を使用すると、次のことを行えます。

- リソース、リソースタイプ、リソースグループ、クラスタについての情報にアクセスする。
- モニターと併用し、リソースの Status プロパティと Status_msg プロパティを設定する。
- リソースグループの再起動と再配置を要求する。
- エラーコードを適切なエラーメッセージに変換する。

注 – この節では、C 関数について簡単に説明します。詳細については、各関数の（3HA）マニュアルページを参照してください。特に注記しないかぎり、各関数と同じ名前のマニュアルページがあります。C 関数の出力関数および戻りコードについては、scha_calls (3HA) のマニュアルページを参照してください。

リソース関数

以下の関数は、RGM に管理されているリソースについての情報にアクセスします。モニターから見たリソースの状態を表します。

scha_resource_open()、scha_resource_get ()、scha_resource_close()
これらの関数は、RGM によって管理されるリソースの情報にアクセスします。scha_resource_open() 関数は、リソースへのアクセスを初期化し、scha_resource_get() のハンドルを戻します。scha_resource_get() 関数は、リソースの情報にアクセスします。scha_resource_close() 関数は、ハンドルを無効にし、scha_resource_get() の戻り値に割り当てられているメモリーを解放します。

scha_resource_open() 関数がリソースのハンドルを戻したあとに、クラスタの再構成や管理アクションによって、リソースが変更されることがあります。この場合、scha_resource_get() 関数がハンドルを通じて獲得した情報は正しくない可能性があります。リソース上でクラスタの再構成や管理アクションが行われた場合、RGM は scha_err_seqid() エラーコードを scha_resource_get() 関数に戻し、リソースが変更されたことを示します。これは、重大ではないエラーメッセージです。関数は正常に終了します。したがって、このメッセージは無視してもかまいません。また、現在のハンドルを閉じて新しいハンドルを開き、リソースの情報にアクセスし直してもかまいません。

これら 3 つの関数は 1 つのマニュアルページ内で説明しています。このマニュアルページには、個々の関数名 scha_resource_open(3HA)、scha_resource_get（3HA）、scha_resource_close(3HA) でアクセスできます。

scha_resource_setstatus()
RGM の制御下のリソースの Status および Status_msg プロパティを設定します。この関数はリソースのモニターによって使用され、モニターから見たリソースの状態を反映します。

注 – scha_resource_setstatus() はリソースモニター専用の関数ですが、任意のプログラムから呼び出すことができます。

リソースタイプ関数

これらの関数は、RGM に登録されているリソースタイプについての情報にアクセスします。

scha_resourcetype_open()、 scha_resourcetype_get()、scha_resourcetype_close()

scha_resourcetype_open() 関数は、リソースへのアクセスを初期化し、scha_resourcetype_get() のハンドルを戻します。scha_resourcetype_get() 関数はリソースタイプの情報にアクセスします。scha_resourcetype_close() 関数は、ハンドルを無効にし、scha_resourcetype_get() の戻り値に割り当てられているメモリーを解放します。

scha_resourcetype_open() 関数がリソースタイプのハンドルを戻したあとに、クラスタの再構成や管理アクションによって、リソースタイプが変更されることがあります。この場合、scha_resourcetype_get() 関数がハンドルを通じて獲得した情報は正しくない可能性があります。リソースタイプ上でクラスタの再構成や管理アクションが行われた場合、RGM は scha_err_seqid エラーコードを scha_resourcetype_get() 関数に戻し、リソースタイプが変更されたことを示します。これは、重大ではないエラーメッセージです。関数は正常に終了します。したがって、このメッセージは無視してもかまいません。また、現在のハンドルを閉じて新しいハンドルを開き、リソースタイプの情報にアクセスし直してもかまいません。

これら 3 つの関数は 1 つのマニュアルページ内で説明しています。このマニュアルページには、個々の関数名 scha_resourcetype_open(3HA)、scha_resourcetype_get3HA、scha_resourcetype_close(3HA) からアクセスできます。

リソースグループ関数

以下の関数を使用すると、リソースグループについての情報にアクセスしたり、リソースグループを再起動できます。

scha_resourcegroup_open(3HA)、scha_resourcegroup_get(3HA)、scha_resourcegroup_close(3HA)

これらの関数は、RGM によって管理されるリソースグループの情報にアクセスします。scha_resourcegroup_open() 関数は、リソースグループへのアクセスを初期化し、scha_resourcegroup_get() のハンドルを戻します。scha_resourcegroup_get() 関数は、リソースグループの情報にアクセスしま

す。scha_resourcegroup_close() 関数は、ハンドルを無効にし、scha_resourcegroup_get() の戻り値に割り当てられているメモリーを解放します。

scha_resourcegroup_open() 関数がリソースグループのハンドルを戻したあとに、クラスタの再構成や管理アクションによって、リソースグループが変更されることがあります。この場合、scha_resourcegroup_get() 関数がハンドルを通じて獲得した情報は正しくない可能性があります。リソースグループ上でクラスタの再構成や管理アクションが行われた場合、RGM は scha_err_seqid エラーコードを scha_resourcegroup_get() 関数に戻し、リソースグループが変更されたことを示します。これは、重大ではないエラーメッセージです。関数は正常に終了します。したがって、このメッセージは無視してもかまいません。また、現在のハンドルを閉じて新しいハンドルを開き、リソースグループの情報にアクセスし直してもかまいません。

scha_control(3HA)
RGM の制御下のリソースグループの再起動、または別のノードへの再配置を要求します。

クラスタ関数

以下の関数は、クラスタについての情報にアクセスし、その情報を戻します。

scha_cluster_open(3HA)、scha_cluster_get(3HA)、scha_cluster_close(3HA)
これらの関数は、ノード名、ID、状態、クラスタ名、リソースグループなど、クラスタに関する情報にアクセスします。

scha_cluster_open() 関数がクラスタのハンドルを戻したあとに、再構成や管理アクションによって、クラスタが変更されることがあります。この場合、scha_cluster_get() 関数がハンドルを通じて獲得した情報は正しくない可能性があります。クラスタ上でクラスタの再構成や管理アクションが行われた場合、RGM は scha_err_seqid エラーコードを scha_cluster_get() 関数に戻し、クラスタが変更されたことを示します。これは、重大ではないエラーメッセージです。関数は正常に終了します。したがって、このメッセージは無視してもかまいません。また、現在のハンドルを閉じて新しいハンドルを開き、クラスタの情報にアクセスし直してもかまいません。

scha_cluster_getlogfacility(3HA)
クラスタログとして使用されるシステムログ機能の数を戻します。戻された番号を Solaris の syslog() 関数で使用すると、イベントと状態メッセージをクラスタログに記録できます。

scha_cluster_getnodename(3HA)
関数が呼び出されたクラスタノードの名前を戻します。

ユーティリティ関数

この関数は、エラーコードをエラーメッセージに変換します。

`scha_strerror` (3HA)
: `scha_` 関数によって戻されるエラーコードを適切なエラーメッセージに変換します。この関数を `logger` と共に使用すると、メッセージをシステムログ (`syslog`) に記録できます。

RMAPI コールバックメソッド

コールバックメソッドは、リソースタイプを実装するための API が提供する重要な要素です。コールバックメソッドを使用すると、RGM は、クラスタのメンバーシップが変更されたとき (ノードが起動またはクラッシュしたとき) にクラスタ内のリソースを制御できます。

注 – クライアントプログラムがクラスタシステム上の HA サービスを制御するため、コールバックメソッドはルートのアクセス権を持つ RGM によって実行されます。したがって、このようなコールバックメソッドをインストールおよび管理するときは、ファイルの所有権とアクセス権を制限します。特に、このようなコールバックメソッドには、特権付き所有者 (`bin` や `root` など) を割り当てます。

さらに、このようなコールバックメソッドは、書き込み可能にしてはなりません。この節では、コールバックメソッドの引数と終了コードについて説明し、次のカテゴリのコールバックメソッドについて説明します。

- 制御および初期化メソッド
- 管理サポートメソッド
- ネットワーク関連メソッド
- モニター制御メソッド

注 – この節では、メソッドが呼び出されるタイミングやよそうされるリソースへの影響など、コールバックメソッドについて簡単に説明します。詳細については `rt_callbacks` (1HA) のマニュアルページを参照してください。

メソッドの引数

RGM はコールバックメソッドを呼び出すとき、次のような引数を使用します。

method `-R` *resource-name* `-T` *type-name* `-G` *group-name*

method は、`Start` や `Stop` などのコールバックメソッドとして登録されているプログラムのパス名です。リソースタイプのコールバックメソッドは、それらの登録ファイルで宣言します。

コールバックメソッドの引数はすべて、フラグ付きの値として渡されます。-R はリソースインスタンスの名前を示し、-T はリソースのタイプを示し、-G はリソースが構成されているグループを示します。このような引数をアクセス関数で使用すると、リソースについての情報を取得できます。

Validate メソッドを呼び出すときは、追加の引数 (リソースのプロパティ値と呼び出されるリソースグループ) を使用します。

詳細については、scha_calls(3HA) のマニュアルページを参照してください。

終了コード

終了コードは、すべてのコールバックメソッドで共通で、メソッドの呼び出しによるリソースの状態への影響を示すように定義されています。これらの終了コードについては、scha_calls (3HA) のマニュアルページを参照してください。終了コードには、以下のものがあります。

- 0 (ゼロ) – メソッドは成功しました。
- ゼロ以外の任意の値 – メソッドは失敗しました。

RGM は、コールバックメソッドの実行の異常終了 (タイムアウトやコアダンプ) も処理します。

メソッドは、各ノード上で syslog を使用して障害情報を出力するように実装する必要があります。stdout や stderr に書き込まれる出力は、ローカルノードのコンソール上には表示されますが、それをユーザーが確認するかどうかは保証できないためです。

制御および初期化コールバックメソッド

制御および初期化コールバックメソッドは、主に、リソースを起動および停止します。その他にも、リソース上で初期化と終了コードを実行します。

Start
: この必須メソッドは、リソースを含むリソースグループがクラスタノード上でオンラインになったとき、そのクラスタノード上で呼び出されます。このメソッドは、そのノード上でリソースを起動します。

  Start メソッドは、ローカルノード上でリソースが起動し、使用可能な状態になるまで終了してはなりません。したがって、Start メソッドは終了する前にリソースをポーリングし、リソースが起動しているかどうかを判断する必要があります。さらに、このメソッドには、十分な長さのタイムアウト値を設定する必要があります。たとえば、あるリソース (データベースデーモンなど) が起動するのに時間がかかる場合、そのメソッドには十分な長さのタイムアウト値を設定する必要があります。

  RGM が Start メソッドの失敗に応答する方法は、Failover_mode プロパティの設定によって異なります。

リソースの Start メソッドのタイムアウト値を設定するには、リソースタイプ登録ファイルの START_TIMEOUT プロパティを使用します。

Stop
この必須メソッドは、リソースを含むリソースグループがクラスタノード上でオフラインになったとき、そのクラスタノード上で呼び出されます。このメソッドは、リソースを (アクティブであれば) 停止します。

Stop メソッドは、ローカルノード上でリソースがすべての活動を完全に停止し、すべてのファイル記述子を終了するまで終了してはなりません。そうしないと、RGM が (実際にはアクティブであるのに) リソースが停止したと判断するため、データが破壊されることがあります。データの破壊を防ぐために最も安全な方法は、ローカルノード上でリソースに関連するすべてのプロセスを停止することです。

Stop メソッドは終了する前にリソースをポーリングし、リソースが停止しているかどうかを判断する必要があります。さらに、このメソッドには、十分な長さのタイムアウト値を設定する必要があります。たとえば、あるリソース (データベースデーモンなど) が停止するのに時間がかかる場合、そのメソッドには十分長めのタイムアウト値を設定する必要があります。

RGM が Stop メソッドの失敗に応答する方法は、Failover_mode プロパティの設定によって異なります (表 A–2 を参照)。

リソースの Stop メソッドのタイムアウト値を設定するには、リソースタイプ登録ファイルの STOP_TIMEOUT プロパティを使用します。

Init
このオプションメソッドは、リソースを管理下に置くとき (リソースが属しているリソースグループを管理していない状態から管理している状態に切り替えるとき、またはすでに管理されているリソースグループでリソースを作成するとき)、1 回だけ呼び出され、リソースの初期化を実行します。このメソッドは、Init_nodes リソースプロパティが示すノード上で呼び出されます。

Fini
このオプションメソッドは、リソースを管理下から外すとき (リソースが属しているリソースグループを管理していない状態に切り替えるとき、またはすでに管理されているリソースグループからリソースを削除するとき) に呼び出され、リソースをクリーンアップします。このメソッドは、Init_nodes リソースプロパティが示すノード上で呼び出されます。

Boot
このオプションメソッドは、Init と同様に、リソースの所属リソースグループが RGM の管理下に置かれたあと、クラスタを結合するノード上のリソースを初期化します。このメソッドは、Init_nodes リソースプロパティが示すノード上で呼び出されます。Boot メソッドは、起動または再起動の結果とし、ノードがクラスタに結合または再結合したときに呼び出されます。

注 – Init、Fini、Boot メソッドが失敗すると、syslog() 関数がエラーメッセージを生成しますが、それ以外は RGM のリソース管理に影響しません。

管理サポートメソッド

リソース上での管理アクションには、リソースプロパティの設定と変更があります。Validate と Update コールバックメソッドを使用してリソースタイプを実装すると、このような管理アクションを行うことができます。

Validate
このオプションメソッドは、リソースの作成時と、管理アクションによってリソースまたはそのリソースグループのプロパティが更新されたときに呼び出されます。このメソッドは、リソースタイプの Init_nodes プロパティが示す複数のクラスタノード上で呼び出されます。Validate は、作成または更新が行われる前に呼び出されます。任意のノード上でメソッドから失敗の終了コードが戻ると、作成または更新は取り消されます。

Validate が呼び出されるのは、リソースまたはリソースグループのプロパティが管理アクションを通じて変更されたときだけです。RGM がプロパティを設定したときや、モニターがリソースプロパティ Status や Status_msg を設定したときではありません。

Update
このオプションメソッドを呼び出して、プロパティが変更されたことを実行中のリソースに通知することができます。Update は、管理アクションがリソースまたはリソースグループのプロパティの設定に成功したあとに呼び出されます。このメソッドは、リソースがオンラインであるノード上で呼び出されます。このメソッドは、API アクセス関数を使用し、アクティブなリソースに影響する可能性があるプロパティ値を読み取り、その値に従って実行中のリソースを調節します。

Update メソッドが失敗すると、syslog() 関数がエラーメッセージを生成しますが、それ以外は RGM のリソース管理に影響しません。

ネットワーク関連コールバックメソッド

ネットワークアドレスリソースを使用するサービスでは、ネットワークアドレス構成に相対的な順番で、起動手順または停止手順を行う必要があります。任意コールバックメソッドの Prenet_start と Postnet_stop を使用してリソースタイプを実装すると、関連するネットワークアドレスが「起動」に構成される前、または、「停止」に構成されたあとに、特別な起動アクションまたはシャットダウンアクションを行うことができます。

Prenet_start
このオプションメソッドを呼び出して、同じリソースグループ内のネットワークアドレスが構成される前に特殊な起動アクションを実行することができます。

Postnet_stop
: このオプションメソッドを呼び出して、同じリソースグループ内のネットワークアドレスの構成後に特殊な終了アクションを実行することができます。

モニター制御コールバックメソッド

リソースタイプは、オプションとして、リソースの性能を監視したり、その状態を報告したり、リソースの障害に対処するようなプログラムを含むようにも実装できます。Monitor_start、Monitor_stop、Monitor_check メソッドは、リソースタイプへのリソースモニターの実装をサポートします。

Monitor_start
: このオプションメソッドを呼び出して、リソースの起動後にリソースの監視を開始することができます。

Monitor_stop
: このオプションメソッドを呼び出して、リソースの停止前にリソースの監視を停止することができます。

Monitor_check
: このオプションメソッドを呼び出して、リソースグループを新しいノードに配置する前に、そのノードの信頼性を査定することができます。Monitor_check メソッドは、並行して実行中のその他のメソッドと競合しない方法で実装する必要があります。

第 5 章

サンプルデータサービス

この章では、in.named アプリケーションを Sun Cluster データサービスとして稼働する HA-DNS について説明します。in.named デーモンは Solaris におけるドメインネームサービス (DNS) の実装です。サンプルのデータサービスでは、Resource Management API を使用して、アプリケーションの高可用性を実現する方法を示します。

RMAPI は、シェルスクリプトと C プログラムの両方のインタフェースをサポートします。この章のサンプルアプリケーションはシェルスクリプトインタフェースで作成されています。

この章の内容は、次のとおりです。

サンプルデータサービスの概要

サンプルのデータサービスはクラスタのイベント(管理アクション、アプリケーションの異常終了、ノードの異常終了など) に応じて、DNS アプリケーションの起動、停止、再起動や、クラスタノード間での DNS アプリケーションの切り替えを行います。

アプリケーションの再起動は、プロセス監視機能 (PMF) によって管理されます。アプリケーションの障害が再試行最大期間または再試行最大回数を超えると、障害モニターはアプリケーションリソースを含むリソースグループを別のノードにフェイルオーバーします。

サンプルのデータサービスは、PROBE メソッドという形で障害監視機能を提供します。PROBE メソッドは、nslookup コマンドを使用し、アプリケーションが正常な状態であることを保証します。DNS サービスのハングを検出すると、PROBE メソッドは、DNS アプリケーションをローカルで再起動することによって、この状況を修正しようとします。この方法で状況が改善されず、サービスの問題が繰り返し検出される場合、PROBE メソッドは、サービスをクラスタ内の別のノードにフェイルオーバーしようとします。

サンプルのアプリケーションには、具体的に、次のような機能が含まれています。

- リソースタイプ登録ファイル - データサービスの静的なプロパティを定義します。
- Start コールバックメソッド - HA-DNS データサービスを含むリソースグループがオンラインになるときに RGM によって呼び出され、in.named デーモンを起動します。
- Stop コールバックメソッド - HA-DNS データサービスを含むリソースグループがオフラインになるときに RGM によって呼び出され、in.named デーモンを停止します。
- 障害モニター - DNS サーバーが動作しているかどうかを確認することによって、サービスの信頼性を検査します。障害モニターはユーザー定義の PROBE メソッドによって実装され、Monitor_start と Monitor_stop コールバックメソッドによって起動および停止されます。
- Validate コールバックメソッド - RGM によって呼び出され、サービスの構成ディレクトリがアクセス可能であるかどうかを検査します。
- Update コールバックメソッド - システム管理者がリソースプロパティの値を変更したときに RGM によって呼び出され、障害モニターを再起動します。

リソースタイプ登録ファイルの定義

この例で使用するサンプルのリソースタイプ登録 (RTR) ファイルは、DNS リソースタイプの静的な構成を定義します。このタイプのリソースは、RTR ファイルで定義されているプロパティを継承します。

RTR ファイル内の情報は、クラスタ管理者が HA-DNS データサービスを登録したときに RGM によって読み取られます。

RTR ファイルの概要

RTR ファイルの形式は明確に定義されています。リソースタイププログラム、システム定義リソースプロパティ、拡張プロパティという順番で並んでいます。詳細は、rt_reg(4) のマニュアルページと34 ページの「リソースとリソースタイププロパティの設定」を参照してください。

この節では、サンプルの RTR ファイルの特定のプロパティについて説明します。この節で扱うリストは、サンプルの RTR ファイルの一部だけです。サンプルの RTR ファイルの完全なリストについては、261 ページの「リソースタイプ登録ファイルのリスト」を参照してください。

サンプル RTR ファイルのリソースタイププロパティ

次のリストに示すように、サンプルのRTR ファイルはコメントから始まり、そのあとに、HA-DNS 構成を定義するリソースタイププロパティが続きます。

```
#
# Copyright (c) 1998-2003 by Sun Microsystems, Inc.
# All rights reserved.
#
# Registration information for Domain Name Service (DNS)
#

#pragma ident   "@(#)SUNW.sample   1.1   00/05/24 SMI"

RESOURCE_TYPE = "sample";
VENDOR_ID = SUNW;
RT_DESCRIPTION = "Domain Name Service on Sun Cluster";

RT_VERSION ="1.0";
API_VERSION = 2;
FAILOVER = TRUE;

RT_BASEDIR=/opt/SUNWsample/bin;
PKGLIST = SUNWsample;

START         =   dns_svc_start;
STOP          =   dns_svc_stop;

VALIDATE      =   dns_validate;
UPDATE        =   dns_update;

MONITOR_START =   dns_monitor_start;
MONITOR_STOP  =   dns_monitor_stop;
MONITOR_CHECK =   dns_monitor_check;
```

ヒント – RTR ファイルの最初のエントリには、`Resource_type` プロパティを宣言する必要があります。宣言しないと、リソースタイプの登録は失敗します。

注 – RGM はプロパティ名の大文字と小文字を区別します。Sun が提供する RTR ファイルのプロパティに対する命名規則では、名前の最初の文字が大文字で、残りが小文字です (メソッド名は例外)。メソッド名は、プロパティ属性と同様にすべて大文字です。

次に、これらのプロパティについての情報を説明します。

- リソースタイプ名は、`Resource_type` プロパティだけで指定できます (例 `:sample`)。あるいは、 `Vendor_id+"."+Resource_type` という形式でも指定できます (例 `:SUNW.sample`)。

  `Vendor_id` を使用する場合、リソースタイプを定義する企業の略号にします。リソースタイプ名はクラスタ内で一意である必要があります。
- `Rt_version` プロパティは、ベンダーによって指定されたサンプルのデータサービスのバージョンを識別します。
- `API_version` プロパティは Sun Cluster のバージョンを識別します。たとえば、「`API_version =2`」は、データサービスが Sun Cluster バージョン 3.0 の管理下で動作していることを示します。
- `Failover = TRUE` の場合、同時に複数のノード上でオンラインになることができるリソースグループでは、データサービスが動作できないことを示します。
- `RT_basedir` は相対パス (コールバックメソッドのパスなど) を補完するためのディレクトリパスで、`/opt/SUNWsample/bin` を指します。
- `Start`、`Stop`、`Validate` などは、RGM によって呼び出される個々のコールバックメソッドプログラムへのパスを提供します。これらのパスは、`RT_basedir` で指定されたディレクトリからの相対パスです。
- `Pkglist` は、`SUNWsample` をサンプルのデータサービスのインストールを含むパッケージとして識別します。

この RTR ファイルに指定されていないリソースタイププロパティ (`Single_instance`、`Init_nodes`、`Installed_nodes` など) は、デフォルト値を取得します。リソースタイププロパティの完全なリストとそのデフォルト値については、表 A–1を参照してください。

クラスタ管理者は、RTR ファイルのリソースタイププロパティに指定されている値を変更できません。

サンプル RTR ファイルのリソースプロパティ

慣習上、RTR ファイルでは、リソースプロパティをリソースタイププロパティのあとに宣言します。リソースプロパティには、Sun Cluster が提供するシステム定義プロパティと、データサービス開発者が定義する拡張プロパティが含まれます。どちらのタイプの場合でも、Sun Cluster が提供するプロパティ属性の数 (最小、最大、デフォルト値など) を指定できます。

RTR ファイルのシステム定義プロパティ

次のリストは、サンプル RTR ファイルのシステム定義プロパティを示しています。

```
# リソースタイプ宣言のあとに、中括弧に囲まれたリソースプロパティ宣言の
# リストが続く。プロパティ名宣言は、各エントリの左中括弧の直後にある最初
# の属性である必要がある。

# <method>_timeout プロパティの値は、RGM がメソッドの呼び出しが
# 失敗したと結論するまでの時間 (秒) を設定する。

# すべてのメソッドタイムアウトの MIN 値は 60 秒に設定されている。
# これは、管理者が短すぎる時間を設定するのを防ぐためである。短すぎる
# 時間を設定すると、スイッチオーバーやフェイルオーバーの性能が上がらず、
# さらには予期せぬ RGM アクションが発生する可能性がある (誤った
# フェイルオーバー、ノードの再起動、リソースグループの
# ERROR_STOP_FAILED 状態への移行など、管理者の介入を必要とする
# RGM アクション)。
# メソッドタイムアウトに短すぎる時間を設定すると、データサービス全体の
# 可用性が低下する。
{
   PROPERTY = Start_timeout;
   MIN=60;
   DEFAULT=300;
}

{
   PROPERTY = Stop_timeout;
   MIN=60;
   DEFAULT=300;
}
{
        PROPERTY = Validate_timeout;
        MIN=60;
        DEFAULT=300;
}
{
        PROPERTY = Update_timeout;
        MIN=60;
        DEFAULT=300;
}
{
        PROPERTY = Monitor_Start_timeout;
        MIN=60;
        DEFAULT=300;
}
{
        PROPERTY = Monitor_Stop_timeout;
        MIN=60;
        DEFAULT=300;
}
{
        PROPERTY = Thorough_Probe_Interval;
        MIN=1;
        MAX=3600;
```

```
        DEFAULT=60;
        TUNABLE = ANYTIME;
}

#  当該ノード上でアプリケーションを正常に起動できないと結論するまでに
# 指定された時間内 (Retry_Interval) に行う再試行回数。
{
        PROPERTY = Retry_Count;
        MIN=0;
        MAX=10;
        DEFAULT=2;
        TUNABLE = ANYTIME;
}

# Retry_Interval には 60 の倍数を指定する。これは、この値は秒から分に変換され、
# 端数が切り上げられるためである。
# たとえば、50 (秒) は 1 分に変換される。このプロパティ値は再試行回数
# (Retry_Count ) のタイミングを指定する。
{
        PROPERTY = Retry_Interval;
        MIN=60;
        MAX=3600;
        DEFAULT=300;
        TUNABLE = ANYTIME;
}

{
        PROPERTY = Network_resources_used;
        TUNABLE = AT_CREATION;
        DEFAULT = "";
}
```

Sun Cluster はシステム定義プロパティを提供しますが、リソースプロパティ属性を使用すると、異なるデフォルト値を設定できます。リソースプロパティに適用するために利用できる属性の完全なリストについては、259 ページの「リソースプロパティの属性」 を参照してください。

サンプルの RTR ファイル内のシステム定義リソースプロパティについては、次の点に注意してください。

- Sun Cluster は、すべてのタイムアウトに最小値 (1 秒) とデフォルト値 (3600 秒) を提供します。サンプルの RTR ファイルは、最小値をそのまま (60 秒) にし、デフォルト値を 300 秒に変更しています。クラスタ管理者は、このデフォルト値を使用することも、タイムアウト値を変更することもできます (たとえば、60秒以上)。Sun Cluster は正当な最大値を持っていません。
- `Thorough_Probe_Interval`、`Retry_count`、`Retry_interval` プロパティの `TUNABLE` 属性は `ANYTIME` に設定されています。この設定は、データサービスが動作中でも、クラスタ管理者がこれらのプロパティの値を変更できることを意味します。上記のプロパティは、サンプルのデータサービスによって実装される障害モニターによって使用されます。サンプルのデータサービスは、管理アクションによってさまざまなリソースが変更されたときに障害モニターを停止および再起動するように、`Update` を実装します。詳細は、118 ページの「`Update` メ

ソッド」を参照してください。

- リソースプロパティは次のように分類されます。
  - 必須—クラスタ管理者はリソースを作成するときに必ず値を指定する必要があります。
  - 任意—クラスタ管理者が値を指定しない場合、システムがデフォルト値を提供します。
  - 条件付き—RTR ファイルで宣言されている場合だけ、RGM はプロパティを作成します。

  サンプルのデータサービスの障害モニターは、Thorough_probe_interval、Retry_count、Retry_interval、Network_resources_used という条件付きプロパティを使用しているため、開発者はこれらのプロパティを RTR ファイルで宣言する必要があります。プロパティの分類方法については、r_properties(5) のマニュアルページまたは246 ページの「リソースプロパティ」を参照してください。

RTR ファイルの拡張プロパティ

次に、RTR ファイルの最後の例として、拡張プロパティを示します。

```
# 拡張プロパティ
# クラスタ管理者は、このプロパティの値として、アプリケーションによって
# 使用される構成ファイルが格納されているディレクトリのパスを指定する。
# このアプリケーション (DNS) は、PXFS 上の DNS 構成ファイルのパス(通常
#  named.conf) のパスを指定する。
{
   PROPERTY = Confdir;
   EXTENSION;
   STRING;
   TUNABLE = AT_CREATION;
   DESCRIPTION = "The Configuration Directory Path";
}

# 検証の失敗が宣言されるまでのタイムアウト値 (秒)。
{
        PROPERTY = Probe_timeout;
        EXTENSION;
        INT;
        DEFAULT = 120;
        TUNABLE = ANYTIME;
        DESCRIPTION = "Time out value for the probe (seconds)";
}
```

サンプルの RTR ファイルは 2 つの拡張プロパティ、Confdir と Probe_timeout を定義します。Confdir は、DNS 構成ディレクトリへのパスを指定します。このディレクトリには、DNS が正常に動作するために必要な in.named ファイルが格納されています。サンプルのデータサービスの Start と Validate メソッドはこのプロパティを使用し、DNS を起動する前に、構成ディレクトリと in.named ファイルがアクセス可能であるかどうかを確認します。

データサービスが構成されるとき、Validate メソッドは、新しいディレクトリがアクセス可能であるかどうかを確認します。

サンプルのデータサービスの PROBE メソッドは、Sun Cluster コールバックメソッドではなく、ユーザー定義メソッドです。したがって、Sun Cluster は この Probe_timeout プロパティを提供しません。開発者はこの拡張プロパティを RTR ファイルに定義し、クラスタ管理者が Probe_timeout の値を構成できるようにする必要があります。

すべてのメソッドに共通な機能の提供

この節では、サンプルのデータサービスのすべてのコールバックメソッドで使用される次のような機能について説明します。

- 94 ページの「コマンドインタプリタの指定およびパスのエクスポート」
- 95 ページの「PMF_TAG と SYSLOG_TAG 変数の宣言」
- 96 ページの「関数の引数の構文解析」
- 97 ページの「エラーメッセージの生成」
- 98 ページの「プロパティ情報の取得」

コマンドインタプリタの指定およびパスのエクスポート

シェルスクリプトの最初の行は、コマンドインタプリタを指定します。サンプルのデータサービスの各メソッドスクリプトは、次に示すように、コマンドインタプリタを指定します。

```
#!/bin/ksh
```

サンプルアプリケーション内のすべてのメソッドスクリプトは、Sun Cluster のバイナリとライブラリへのパスをエクスポートします。ユーザーの PATH 設定には依存しません。

```
##############################################################################
# MAIN
##############################################################################

export PATH=/bin:/usr/bin:/usr/cluster/bin:/usr/sbin:/usr/proc/bin:$PATH
```

PMF_TAG と SYSLOG_TAG 変数の宣言

すべてのメソッドスクリプト (Validate を除く) は、リソース名を渡し、pmfadm を使用してデータサービスまたはモニターのいずれかを起動 (または停止) します。各スクリプトは変数 PMF_TAG を定義し、pmfadm に渡すことによって、データサービスまたはモニターを識別できます。

同様に、各メソッドスクリプトは、logger コマンドを使用してメッセージをシステムログに記録します。各スクリプトは変数 SYSLOG_TAG を定義し、-t オプションで logger に渡すことによって、メッセージが記録されるリソースのリソースタイプ、リソースグループ、リソース名を識別できます。

すべてのメソッドは、次に示す例と同じ方法で SYSLOG_TAG を定義します。dns_probe、dns_svc_start、dns_svc_stop、dns_monitor_check の各メソッドは、次のように PMF_TAG を定義します。なお、 pmfadm と logger は dns_svc_stop メソッドのものを使用しています。

```
##############################################################################
# MAIN
##############################################################################

PMF_TAG=$RESOURCE_NAME.named

SYSLOG_TAG=$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME

   # データサービスに SIGTERM シグナルを送り、タイムアウト値の 80%
   # が経過するまで待機する。
   pmfadm -s $PMF_TAG.named -w $SMOOTH_TIMEOUT TERM
   if [ $? -ne 0 ]; then
      logger -p ${SYSLOG_FACILITY}.info \
          -t [$SYSLOG_TAG] \
          "${ARGV0} Failed to stop HA-DNS with SIGTERM; Retry with \
           SIGKILL"
```

dns_monitor_stop、dns_monitor_stop、dns_update の各メソッドは、次のように PMF_TAG を定義します (なお、pmfadm は dns_monitor_stop メソッドのものを使用しています)。

```
##############################################################################
# MAIN
##############################################################################

PMF_TAG=$RESOURCE_NAME.monitor
SYSLOG_TAG=$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME
...

# in.named が実行中であるかどうかを確認し、実行中であれば強制終了する。
if pmfadm -q $PMF_TAG.monitor; then
   pmfadm -s $PMF_TAG.monitor KILL
```

関数の引数の構文解析

RGM は、次に示すように、すべてのコールバックメソッド (Validate を除く) を呼び出します。

method_name -R *resource_name* -T *resource_type_name* -G *resource_group_name*

method_name は、コールバックメソッドを実装するプログラムのパス名です。データサービスは、各メソッドのパス名を RTR ファイルに指定します。このようなパス名は、RTR ファイルの Rt_basedir プロパティに指定されたディレクトリからのパスになります。たとえば、サンプルのデータサービスの RTR ファイルでは、ベースディレクトリとメソッド名は次のように指定されます。

```
RT_BASEDIR=/opt/SUNWsample/bin;
START = dns_svc_start;
STOP =  dns_svc_stop;
...
```

コールバックメソッドの引数はすべて、フラグ付きの値として渡されます。-R はリソースインスタンスの名前を示し、-T はリソースのタイプを示し、-G はリソースが構成されているグループを示します。コールバックメソッドの詳細については、rt_callbacks (1HA) のマニュアルページを参照してください。

注 – Validate メソッドを呼び出すときは、追加の引数 (リソースのプロパティ値と呼び出されるリソースグループ) を使用します。詳細は、113 ページの「プロパティ更新の処理」を参照してください。

各コールバック、メソッドには、渡された引数を構文解析する関数が必要です。すべてのコールバックメソッドには同じ引数が渡されるので、データサービスは、アプリケーション内のすべてのコールバックメソッドで使用される単一の構文解析関数を提供します。

次に、サンプルのアプリケーションのコールバックメソッドで使用される parse_args() 関数を示します。

```
#############################################################################
# プログラム引数の解析。
#
function parse_args # [args ...]
{
      typeset opt

      while getopts 'R:G:T:' opt
      do
              case "$opt" in
              R)
                   # DNS リソース名。
                   RESOURCE_NAME=$OPTARG
                   ;;
              G)
```

```
                # リソースが構成されたリソースグループ名。
                RESOURCEGROUP_NAME=$OPTARG
                ;;
            T)
                # リソースタイプ名。
                RESOURCETYPE_NAME=$OPTARG
                ;;
            *)
                logger -p ${SYSLOG_FACILITY}.err \
                -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME] \
                "ERROR: Option $OPTARG unknown"
                exit 1
                    ;;
            esac
    done
}
```

注 – サンプルのアプリケーションの PROBE メソッドはユーザー定義メソッドですが、Sun Cluster コールバックメソッドと同じ引数で呼び出されます。したがって、このメソッドには、他のコールバックメソッドと同じ構文解析関数が含まれています。

構文解析関数は、次に示すように、MAIN の中で呼び出されます。

```
parse_args "$@"
```

エラーメッセージの生成

エンドユーザーに対してエラーメッセージを出力するには、syslog 機能をメソッドに使用することを推奨します。サンプルのデータサービスのすべてのコールバックメソッドは、次に示すように、scha_cluster_get() 関数を使用し、クラスタログ用に使用されている syslog 機能番号を取得します。

```
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`
```

この値はシェル変数 SYSLOG_FACILITY に格納されます。logger コマンドの機能として使用すると、エラーメッセージをクラスタログに記録できます。たとえば、サンプルのデータサービスの Start メソッドは、次に示すように、SYSLOG_FACILITY を取得し、データサービスが起動したことを示すメッセージを記録します。

```
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`
...

if [ $? -eq 0 ]; then
   logger -p ${SYSLOG_FACILITY}.err \
         -t [$SYSLOG_TAG] \
         "${ARGV0} HA-DNS successfully started"
fi
```

詳細については、scha_cluster_get (1HA) のマニュアルページを参照してください。

プロパティ情報の取得

ほとんどのコールバックメソッドは、データサービスのリソースとリソースタイプのプロパティについての情報を取得する必要があります。 このために、API は scha_resource_get() 関数を提供しています。

リソースプロパティには 2 種類 (システム定義プロパティと拡張プロパティ) あります。システム定義プロパティは事前に定義されており、拡張プロパティはデータサービス開発者が RTR ファイルに定義します。

scha_resource_get() を使用してシステム定義プロパティの値を取得するときは、-O パラメータでプロパティの名前を指定します。このコマンドは、プロパティの値だけを戻します。たとえば、サンプルのデータサービスの Monitor_start メソッドは検証プログラムを特定し、起動できるようにしておく必要があります。検証プログラムはデータベースのベースディレクトリ (RT_BASEDIR プロパティが指す位置) 内に存在します。したがって、Monitor_start メソッドは、次に示すように、RT_BASEDIR の値を取得し、その値を RT_BASEDIR 変数に格納します。

```
RT_BASEDIR=`scha_resource_get -O RT_BASEDIR -R $RESOURCE_NAME -G \
$RESOURCEGROUP_NAME`
```

拡張プロパティの場合、データサービス開発者は、これが拡張プロパティであることを示す -O パラメータを指定し、最後のパラメータとしてプロパティの名前を指定する必要があります。拡張プロパティの場合、このコマンドは、プロパティのタイプと値の両方を戻します。たとえば、サンプルのデータサービスの検証プログラムは、次に示すように、probe_timeout 拡張プロパティのタイプと値を取得し、次に awk コマンドを使用して値だけを PROBE_TIMEOUT シェル変数に格納します。

```
probe_timeout_info=`scha_resource_get -O Extension -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAME Probe_timeout`
PROBE_TIMEOUT=`echo $probe_timeout_info | awk '{print $2}'`
```

データサービスの制御

データサービスは、クラスタ上でアプリケーションデーモンを起動するために Start メソッドまたは Prenet_start メソッドを提供し、クラスタ上でアプリケーションデーモンを停止するために Stop メソッドまたは Postnet_stop メソッドを提供する必要があります。サンプルのデータサービスは、Start メソッドと Stop メソッドを実装します。代わりに Prenet_start メソッドと Postnet_stop メソッドを使用する場合は、45 ページの「Start および Stop メソッドを使用するかどうかの決定」を参照してください。

Start メソッド

データサービスリソースを含むリソースグループがクラスタノード上でオンラインになるとき、あるいは、リソースグループがすでにオンラインになっていて、そのリソースが有効なとき、RGM はそのノード上で Start メソッドを呼び出します。サンプルのアプリケーションでは、Start メソッドはそのノード上で in.named (DNS) デーモンを起動します。

この節では、サンプルのアプリケーションの Start メソッドの重要な部分だけを説明します。parse_args() 関数や syslog 機能番号を取得する方法など、すべてのコールバックメソッドに共通な機能については説明しません。このような機能については、94 ページの「すべてのメソッドに共通な機能の提供」 を参照してください。

Start メソッドの完全なリストについては、264 ページの「Start メソッド」 を参照してください。

Start の概要

DNS を起動する前に、サンプルのデータサービスの Start メソッドは、構成ディレクトリと構成ファイル (named.conf) がアクセス可能で利用可能であるかどうかを確認します。DNS が正常に動作するためには、named.conf の情報が重要です。

このコールバックメソッドは、プロセス監視機能 (pmfadm) を使用し、DNS デーモン (in.named) を起動します。DNS がクラッシュしたり、起動に失敗したりすると、このメソッドは、一定の期間に一定の回数だけ DNS の起動を再試行します。再試行の回数と期間は、データサービスの RTR ファイル内のプロパティで指定されます。

構成の確認

DNS が動作するためには、構成ディレクトリ内の named.conf ファイルからの情報が必要です。したがって、Start メソッドは、DNS を起動しようとする前にいくつかの妥当性検査を実行し、ディレクトリやファイルがアクセス可能であるかどうかを確認します。

Confdir 拡張プロパティは、構成ディレクトリへのパスを提供します。プロパティ自身は RTR ファイルに定義されています。しかし、実際の位置は、クラスタ管理者がデータサービスを構成するときに指定します。

サンプルのデータサービスでは、Start メソッドは scha_resource_get() 関数を使用して構成ディレクトリの位置を取得します。

注 – Confdir は拡張プロパティであるため、scha_resource_get() はタイプと値の両方を戻します。したがって、awk コマンドで値だけを取得し、シェル変数 CONFIG_DIR に格納します。

```
# クラスタ管理者がリソースの追加時に設定した Confdir の値を検索。
config_info=`scha_resource_get -O Extension -R $RESOURCE_NAME \
```

```
-G $RESOURCEGROUP_NAME Confdir`

# scha_resource_get は、拡張プロパティの値とともにタイプを返す。
# 拡張プロパティの値だけを取得。
CONFIG_DIR=`echo $config_info | awk '{print $2}'`
```

次に、Start メソッドは CONFIG_DIR の値を使用し、ディレクトリがアクセス可能であるかどうかを確認します。アクセス可能ではない場合、Start メソッドはエラーメッセージを記録し、エラー状態で終了します。101 ページの「Start の終了状態」を参照してください。

```
# $CONFIG_DIR がアクセス可能かどうか確認。
if [ ! -d $CONFIG_DIR ]; then
   logger -p ${SYSLOG_FACILITY}.err \
         -t [$SYSLOG_TAG] \
         "${ARGV0} Directory $CONFIG_DIR is missing or not mounted"
   exit 1
fi
```

アプリケーションデーモンを起動する前に、このメソッドは最終検査を実行し、named.conf ファイルが存在するかどうかを確認します。存在しない場合、Start メソッドはエラーメッセージを記録し、エラー状態で終了します。

```
# データファイルに相対パス名が含まれている場合は $CONFIG_DIR
# ディレクトリに移動。
cd $CONFIG_DIR

# named.conf ファイルが存在することを確認。
if [ ! -s named.conf ]; then
   logger -p ${SYSLOG_FACILITY}.err \
         -t [$SYSLOG_TAG] \
         "${ARGV0} File $CONFIG_DIR/named.conf is missing or empty"
   exit 1
fi
```

アプリケーションの起動

このメソッドは、プロセス監視機能 (pmfadm) を使用してアプリケーションを起動します。pmfadm コマンドを使用すると、アプリケーションを再起動するときの期間と回数を指定できます。このため、RTR ファイルには 2 つのプロパティ Retry_count と Retry_interval があります。Retry_count は、アプリケーションを再起動する回数を指定し、Retry_interval は、アプリケーションを再起動する期間を指定します。

Start メソッドは、scha_resource_get() 関数を使用して Retry_count と Retry_interval の値を取得し、これらの値をシェル変数に格納します。次に、-n オプションと -t オプションを使用し、これらの値を pmfadm に渡します。

```
# RTR ファイルから再試行回数の値を取得する。
RETRY_CNT=`scha_resource_get -O Retry_Count -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAME`
```

```
# RTR ファイルから次の再試行までの時間 (秒数) を取得。この値は pmfadm
# に渡されるため分数に変換される。変換時に端数が切り上げられる点に注意。
# たとえば 50 秒は 1 分に変換される。
((RETRY_INTRVAL=`scha_resource_get -O Retry_Interval -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAME` / 60))

# PMF 制御下で in.named デーモンを起動する。RETRY_INTERVAL の期間、
# $RETRY_COUNT の回数だけクラッシュおよび再起動できる。
# それ以上の回数クラッシュした場合、PMF はそれ以上再試行しない。
# <$PMF_TAG> タグで登録済みのプロセスがある場合、PMF はプロセスが
# すでに実行中であるという警告メッセージを送信する。
pmfadm -c $PMF_TAAG -n $RETRY_CNT -t $RETRY_INTRVAL \
    /usr/sbin/in.named -c named.conf

# HA-DNS が起動していることを示すメッセージを記録。
if [ $? -eq 0 ]; then
   logger -p ${SYSLOG_FACILITY}.err \
         -t [$SYSLOG_TAG] \
         "${ARGV0} HA-DNS successfully started"
fi
exit 0
```

Start の終了状態

Start メソッドは、実際のアプリケーションが本当に動作して実行可能になるまで、成功状態で終了してはなりません。特に、ほかのデータサービスが依存している場合は注意する必要があります。正常に終了したかどうかを検証するための 1 つの方法は、Start メソッドが終了する前に、アプリケーションが動作しているかどうかを確認することです。複雑なアプリケーション (データベースなど) の場合、RTR ファイルの Start_timeout プロパティに十分高い値を設定することによって、アプリケーションが初期化され、クラッシュ回復を実行できる時間を提供します。

注 – サンプルのデータサービスのアプリケーションリソース DNS は直ちに起動するため、サンプルのデータサービスは、成功状態でする前に、ポーリングでアプリケーションが動作していることを確認していません。

このメソッドが DNS の起動に失敗し、失敗状態で終了すると、RGM は Failover_mode プロパティを検査し、どのように対処するかを決定します。サンプルのデータサービスは明示的に Failover_mode プロパティを設定していないため、このプロパティはデフォルト値 NONE が設定されています (ただし、クラスタ管理者がデフォルトを変更して異なる値を指定していないと仮定します)。したがって、RGM は、データサービスの状態を設定するだけで、ほかのアクションは行いません。同じノード上で再起動したり、別のノードにフェイルオーバーしたりするには、ユーザーの介入が必要です。

Stop メソッド

HA-DNS リソースを含むリソースグループがクラスタノード上でオフラインになるとき、あるいはリソースグループがオンラインでリソースが無効なとき、Stop メソッドが呼び出されます。このメソッドは、そのノード上で in.named (DNS) デーモンを停止します。

この節では、サンプルのアプリケーションの Stop メソッドの重要な部分だけを説明します。parse_args() 関数や syslog 機能番号を取得する方法など、すべてのコールバックメソッドに共通な機能については説明しません。このような機能については、94 ページの「すべてのメソッドに共通な機能の提供」 を参照してください。

Stop メソッドの完全なリストについては、267 ページの「Stop メソッド」 を参照してください。

Stop の概要

データサービスを停止するときは、考慮すべきことが 2 点あります。1 点は、停止処理を正しい順序で行うことです。これを実現する最良の方法は、pmfadm 経由で SIGTERM シグナルを送信することです。

もう 1 点は、データサービスが本当に停止していることを保証することによって、データベースが Stop_failed 状態にならないようにすることです。これを実現する最良の方法は、pmfadm 経由で SIGKILL シグナルを送信することです。

サンプルのデータサービスの Stop メソッドは、このような点を考慮しています。まず、SIGTERM シグナルを送信し、このシグナルがデータサービスの停止に失敗した場合は、SIGKILL シグナルを送信します。

DNS を停止しようとする前に、この Stop メソッドは、プロセスが実際に動作しているかどうかを確認します。プロセスが動作している場合、Stop メソッドはプロセス監視機能 (pmfadm) を使用してプロセスを停止します。

この Stop メソッドは何回か呼びだしてもその動作が変らないことが保証されます。RGM は、Start の呼び出しでデータサービスを起動せずに、Stop メソッドを 2 回呼び出すことはありません。しかし、RGM は、リソースが起動されていなくても、あるいは、リソースが自発的に停止している場合でも、Stop メソッドをそのリソース上で呼び出すことができます。つまり、DNS がすでに動作していない場合でも、この Stop メソッドは成功状態でします。

アプリケーションの停止

Stop メソッドは、データサービスを停止するために 2 段階の方法を提供します。pmfadm 経由で SIGTERM シグナルを使用する規則正しい方法と、SIGKILL シグナルを使用する強制的な方法です。Stop メソッドは、Stop メソッドが戻るまでの時間を示す Stop_timeout 値を取得します。次に、Stop メソッドはこの時間の 80% を規則正しい方法に割り当て、15% を強制的な方法に割り当てます (5% は予約済み)。次の例を参照してください。

```
STOP_TIMEOUT=`scha_resource_get -O STOP_TIMEOUT -R $RESOURCE_NAME
\

-G $RESOURCEGROUP_NAMÈ
((SMOOTH_TIMEOUT=$STOP_TIMEOUT * 80/100))

((HARD_TIMEOUT=$STOP_TIMEOUT * 15/100))
```

Stop メソッドは pmfadm -q を使用し、DNS デーモンが動作しているかどうかを確認します。動作している場合、Stop メソッドはまず pmfadm -s を使用して TERM シグナルを送信し、DNS プロセスを終了します。このシグナルを送信してからタイムアウト値の 80% が経過してもプロセスが終了しない場合、Stop メソッドは SIGKILL シグナルを送信します。このシグナルを送信してからタイムアウト値の 15% が経過してもプロセスが終了しない場合、STOP メソッドはエラーメッセージを記録し、エラー状態で終了します。

pmfadm がプロセスを終了した場合、STOP メソッドはプロセスが停止したことを示すメッセージを記録し、成功状態で終了します。

DNS プロセスが動作していない場合、STOP メソッドは DNS プロセスが動作していないことを示すメッセージを記録しますが、成功状態で終了します。次のコード例に、Stop メソッドがどのように pmfadm を使用して DNS プロセスを停止するかを示します。

```
# in.named が実行中であるかどうかを確認し、実行中であれば強制終了する。
if pmfadm -q $PMF_TAG; then
   # データサービスに SIGTERM シグナルを送り、タイムアウト値の 80%
   # が経過するまで待機する。
   pmfadm -s $RESOURCE_NAME.named -w $SMOOTH_TIMEOUT TERM
   if [ $? -ne 0 ]; then
      logger -p ${SYSLOG_FACILITY}.err \
          -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME] \
          "${ARGV0} Failed to stop HA-DNS with SIGTERM; Retry with \
           SIGKILL"

      # SIGTERM シグナルでデータサービスが停止しないので、今度は SIGKILL を
      # 使って、合計タイムアウト値の残りの 15% が経過するまで待機する。
      pmfadm -s $PMF_TAG -w $HARD_TIMEOUT KILL
      if [ $? -ne 0 ]; then
          logger -p ${SYSLOG_FACILITY}.err \
          -t [$SYSLOG_TAG]
          "${ARGV0} Failed to stop HA-DNS; Exiting UNSUCCESFUL"

          exit 1
      fi
fi
else
   # この時点でデータサービスは実行されていない。メッセージを記録して
   # 成功状態で終了する。
   logger -p ${SYSLOG_FACILITY}.err \
           -t [$SYSLOG_TAG] \
           "HA-DNS is not started"

   # HA-DNS が実行中でなくても、データサービスリソースを STOP_FAILED
```

```
   # 状態にするのを避けるため成功状態で終了する。

   exit 0

fi

# DNS の停止に成功。メッセージを記録して成功状態で終了する。
logger -p ${SYSLOG_FACILITY}.err \
    -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME]
\
    "HA-DNS successfully stopped"
exit 0
```

Stop の終了状態

Stop メソッドは、実際のアプリケーションが本当に停止するまで、成功状態で終了してはなりません。特に、ほかのデータサービスが依存している場合は注意する必要があります。そうしなければ、データが破壊される可能性があります。

複雑なアプリケーション (データベースなど) の場合、RTR ファイルの Stop_timeout プロパティに十分高い値を設定することによって、アプリケーションが停止中にクリーンアップできる時間を提供します。

このメソッドが DNS の停止に失敗し、失敗状態で終了すると、RGM は Failover_mode プロパティを検査し、どのように対処するかを決定します。サンプルのデータサービスは明示的に Failover_mode プロパティを設定していないため、このプロパティはデフォルト値 NONE が設定されています (ただし、クラスタ管理者がデフォルトを変更して異なる値を指定していないと仮定します)。したがって、RGM は、データサービスの状態を Stop_failed に設定するだけで、ほかのアクションは行いません。アプリケーションを強制的に停止し、Stop_failed 状態をクリアするには、ユーザーの介入が必要です。

障害モニターの定義

サンプルのアプリケーションは、DNS リソース (in.named) の信頼性を監視する基本的な障害モニターを実装します。障害モニターは、次の要素から構成されます。

- dns_probe - nslookup を使用し、サンプルのデータサービスの制御下にある DNS リソースが動作しているかどうかを確認するユーザー定義プログラム。DNS が動作していない場合、このメソッドは DNS をローカルで再起動しようとします。あるいは、再起動の再試行回数によっては、RGM がデータサービスを別のノードに再配置することを要求します。
- dns_monitor_start - dns_probe を起動するコールバックメソッド。監視が有効である場合、RGM は、サンプルのデータサービスがオンラインになった後、自動的に dns_monitor_start を呼び出します。

- dns_monitor_stop - dns_probe を停止するコールバックメソッド。RGM は、サンプルのデータサービスがオフラインになる前に、自動的に dns_monitor_stop を呼び出します。
- dns_monitor_check - PROBE プログラムがデータサービスを新しいノードにフェイルオーバーするとき、Validate メソッドを呼び出し、構成ディレクトリが利用可能であるかどうかを確認するコールバックメソッド。

検証プログラム

dns_probe プログラムは、サンプルのデータサービスの管理下にある DNS リソースが動作しているかどうかを確認する、連続して動作するプロセスを実行します。dns_probe は、サンプルのデータサービスがオンラインになったあと、RGM によって自動的に呼び出される dns_monitor_start メソッドによって起動されます。データサービスは、サンプルのデータサービスがオフラインになる前、RGM によって呼び出される dns_monitor_stop メソッドによって停止されます。

この節では、サンプルのアプリケーションの PROBE メソッドの重要な部分だけを説明します。parse_args() 関数や syslog 機能番号を取得する方法など、すべてのコールバックメソッドに共通な機能については説明しません。このような機能については、94 ページの「すべてのメソッドに共通な機能の提供」 を参照してください。

PROBE メソッドの完全なリストについては、270 ページの「PROBE プログラム」 を参照してください。

検証プログラムの概要

検証プログラムは無限ループで動作します。検証プログラムは、nslookup を使用し、適切な DNS リソースが動作しているかどうかを確認します。DNS が動作している場合、検証プログラムは一定の期間 (Thorough_probe_interval システム定義プロパティに設定されている期間) だけ休眠し、その後、再び検証を行います。DNS が動作していない場合、検証プログラムは DNS をローカルで再起動しようとするか、再起動の再試行回数によっては、RGM がデータサービスを別のノードに再配置することを要求します。

プロパティ値の取得

このプログラムには、次のプロパティ値が必要です。

- Thorough_probe_interval – 検証プログラムが休眠する期間を設定します。
- Probe_timeout – nslookup コマンドが検証を行う期間 (タイムアウト値) を設定します。
- Network_resources_used – DNS が動作するサーバーを設定します。
- Retry_count と Retry_interval – 再起動を行う回数と期間を設定します。

- Rt_basedir – PROBE プログラムと gettime ユーティリティーが格納されているディレクトリを設定します。

scha_resource_get() 関数は、次に示すように、上記プロパティの値を取得し、シェル変数に格納します。

```
PROBE_INTERVAL=‘scha_resource_get -O THOROUGH_PROBE_INTERVAL \
-R $RESOURCE_NAME -G $RESOURCEGROUP_NAME‘

probe_timeout_info=‘scha_resource_get -O Extension -R $RESOURCE_NAME
\
-G $RESOURCEGROUP_NAME Probe_timeout‘
PROBE_TIMEOUT=‘echo $probe_timeout_info | awk ’{print $2}’‘

DNS_HOST=‘scha_resource_get -O NETWORK_RESOURCES_USED -R $RESOURCE_NAME
\
-G $RESOURCEGROUP_NAME‘

RETRY_COUNT=‘scha_resource_get -O RETRY_COUNT -R $RESOURCE_NAME
-G\
 $RESOURCEGROUP_NAME‘

RETRY_INTERVAL=‘scha_resource_get -O RETRY_INTERVAL -R $RESOURCE_NAME
-G\
 $RESOURCEGROUP_NAME‘

RT_BASEDIR=‘scha_resource_get -O RT_BASEDIR -R $RESOURCE_NAME -G\
 $RESOURCEGROUP_NAME‘
```

注 – システム定義プロパティ (Thorough_probe_interval など) の場合、scha_resource_get() は値だけを戻します。拡張プロパティ (Probe_timeout など) の場合、scha_resource_get() はタイプと値を戻します。値だけを取得するには awk コマンドを使用します。

サービスの信頼性の検査

検証プログラム自身は、nslookup コマンドの while による無限ループです。while ループの前に、nslookup の応答を保管する一時ファイルを設定します。*probefail* 変数と *retries* 変数は 0 に初期化されます。

```
# nslookup の応答用一時ファイルを設定。
DNSPROBEFILE=/tmp/.$RESOURCE_NAME.probe
probefail=0
retries=0
```

while ループ自身は、次の作業を行います。

- 検証プログラム用の休眠期間を設定します。
- hatimerun を使用し、nslookup に Probe_timeout の値とターゲットホストを渡して起動します。

- nslookup の戻りコード (成功または失敗) に基づいて、*probefail* 変数を設定します。
- *probefail* が 1 (失敗) に設定された場合、nslookup への応答がサンプルのデータサービスから来ており、他の DNS サーバーから来ているのではないことを確認します。

次に、while ループコードを示します。

```
while :
do
   # 検証が実行される時間は THOROUGH_PROBE_INTERVAL プロパティ
   # に指定されている。したがって、THOROUGH_PROBE_INTERVAL の間
   # 検証が休眠するように設定する。
   sleep $PROBE_INTERVAL

   # DNS がサービスを提供している IP アドレス上で nslookup コマンドを実行する。
   hatimerun -t $PROBE_TIMEOUT /usr/sbin/nslookup $DNS_HOST $DNS_HOST \
> $DNSPROBEFILE 2>&1

      retcode=$?
      if [ $retcode -ne 0 ]; then
            probefail=1
      fi

   # nslookup への応答が /etc/resolv.conf ファイルに指定されているその他の
   # ネームサーバーではなく HA-DNS サーバーから返されていることを確認する。
   if [ $probefail -eq 0 ]; then
# Get the name of the server that replied to the nslookup query.
   SERVER=` awk ' $1=="Server:" { print $2 }' \
   $DNSPROBEFILE | awk -F. ' { print $1 } ' `
   if [ -z "$SERVER" ]; then
      probefail=1
      else
         if [ $SERVER != $DNS_HOST ]; then
            probefail=1
         fi
   fi
fi
```

再起動とフェイルオーバーの評価

probefail 変数が 0 (成功) 以外である場合、nslookup コマンドがタイムアウトしたか、あるいは、サンプルのサービスの DNS 以外のサーバーから応答が来ていることを示します。どちらの場合でも、DNS サーバーは期待どおりに機能していないので、障害モニターは decide_restart_or_failover() 関数を呼び出し、データサービスをローカルで起動するか、RGM がデータサービスを別のノードに再配置することを要求するかを決定します。*probefail* 変数が 0 の場合、検証が成功したことを示すメッセージが生成されます。

```
   if [ $probefail -ne 0 ]; then
         decide_restart_or_failover
   else
```

```
        logger -p ${SYSLOG_FACILITY}.err\
        -t [$SYSLOG_TAG]\
        "${ARGV0} Probe for resource HA-DNS successful"
fi
```

decide_restart_or_failover() 関数は、再試行最大期間 (Retry_interval) と再試行最大回数 (Retry_count) を使用し、DNS をローカルで再起動するか、RGM がデータサービスを別のノードに再配置することを要求するかを決定します。この関数は、次のような条件付きコードを実装します。コードリストについては、270 ページの「PROBE プログラム」にある decide_restart_or_failover() を参照してください。

- 最初の障害である場合、データサービスをローカルで再起動します。エラーメッセージを記録し、retries 変数の再試行カウンタをインクリメントします。
- 最初の障害ではなく、再試行時間が再試行最大期間を過ぎている場合、データサービスをローカルで再起動します。エラーメッセージを記録し、再試行カウンタをリセットし、再試行時間をリセットします。
- 再試行時間が再試行最大期間を過ぎておらず、再試行カウンタが再試行最大回数を超えている場合、別のノードにフェイルオーバーします。フェイルオーバーが失敗すると、エラーメッセージを記録し、検証プログラムを状態 1 (失敗) で終了します。
- 再試行時間が再試行最大期間を過ぎておらず、再試行カウンタが再試行最大回数を超えていない場合、データサービスをローカルで再起動します。エラーメッセージを記録し、retries 変数の再試行カウンタをインクリメントします。

期限 (再試行最大期間) 内に再起動の回数 (再試行カウンタ) が制限 (再試行最大回数) に到達した場合、この関数は、RGM がデータサービスを別のノードに再配置することを要求します。再起動の回数が制限に到達していない場合、あるいは、再試行最大期間を過ぎていて、再試行カウンタをリセットする場合、この関数は DNS を同じノード上で再起動しようとします。この関数については、次の点に注意してください。

- gettime ユーティリティーを使用すると、再起動間の時間を追跡できます。このユーティリティーは C プログラムで、(Rt_basedir) ディレクトリ内にあります。
- Retry_count と Retry_interval のシステム定義リソースプロパティは、再起動を行う回数と期間を決定します。RTR ファイルのデフォルト値は、Retry_count が 2 回、Retry_interval が 5 分 (300 秒) です。クラスタ管理者はこのデフォルトを変更できます。
- restart_service() 関数は、同じノード上でデータサービスの再起動を試行する場合に呼び出されます。この関数の詳細については、109 ページの「データサービスの再起動」を参照してください。
- API 関数 scha_control() は、GIVEOVER オプションを指定すると、サンプルデータサービスを含むリソースグループをオフラインにし、別のノード上でオンラインにし直します。

データサービスの再起動

`restart_service()` 関数は、`decide_restart_or_failover()` によって呼び出され、同じノード上でデータサービスの再起動を試行します。この関数は次の作業を行います。

- データサービスが PMF 下にまだ登録されているかどうかを調べます。サービスが登録されている場合、この関数は次の作業を行います。
  - データサービスの `Stop` メソッド名と `Stop_timeout` 値を取得します。
  - `hatimerun` を使用してデータサービスの `Stop` メソッドを起動し、`Stop_timeout` 値を渡します。
  - (データサービスが正常に停止した場合) データサービスの `Start` メソッド名と `Start_timeout` 値を取得します。
  - `hatimerun` を使用してデータサービスの `Start` メソッドを起動し、`Start_timeout` 値を渡します。
- データサービスが PMF 下に登録されていない場合は、データサービスが PMF 下で許可されている再試行最大回数を超えていることを示しています。したがって、`GIVEOVER` オプションを指定して `scha_control()` 関数を呼び出し、データサービスを別のノードにフェイルオーバーします。

```
function restart_service
{

        # データサービスを再起動するには、まずデータサービス自体が PMF に
        # 登録されているかどうかを確認する。
        pmfadm -q $PMF_TAG
        if [[ $? -eq 0 ]]; then
                # データサービスの TAG が PMF に登録されている場合、データ
                # サービスを停止し、再起動する。当該リソースの Stop メソッド名と
                # STOP_TIMEOUT 値を取得する。
                STOP_TIMEOUT=`scha_resource_get -O STOP_TIMEOUT \
                        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAMÈ
                STOP_METHOD=`scha_resource_get -O STOP \
                        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAMÈ
                hatimerun -t $STOP_TIMEOUT $RT_BASEDIR/$STOP_METHOD \
                        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME \
                        -T $RESOURCETYPE_NAME

                if [[ $? -ne 0 ]]; then
                        logger-p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG] \
                                "${ARGV0} Stop method failed."
                        return 1
                fi

                # 当該リソースの Start メソッド名と START_TIMEOUT 値を取得する。
                START_TIMEOUT=`scha_resource_get -O START_TIMEOUT \
                        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAMÈ
                START_METHOD=`scha_resource_get -O START \
                        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAMÈ
                hatimerun -t $START_TIMEOUT $RT_BASEDIR/$START_METHOD \
                        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME \
```

```
                -T $RESOURCETYPE_NAME

        if [[ $? -ne 0 ]]; then
                logger-p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG] \
                        "${ARGV0} Start method failed."
                return 1
        fi


else
        # データサービスの TAG が PMF に登録されていない場合、
        # データサービスが PMF で許可されている最大再試行回数
        # を超過している。したがって、データサービスを再起動しては
        # ならない。代わりに、同じクラスタ内の別のノードへの
        # フェイルオーバーを試みる。
        scha_control -O GIVEOVER -G $RESOURCEGROUP_NAME \
                -R $RESOURCE_NAME
fi

return 0
}
```

検証プログラムの終了状態

ローカルでの再起動が失敗したり、別のノードへのフェイルオーバーが失敗したりすると、サンプルのデータサービスの PROBE プログラムは失敗で終了し、Failover attempt failed (フェイルオーバーは失敗しました)" というエラーメッセージを記録します。

Monitor_start メソッド

サンプルのデータサービスがオンラインになったあと、RGM は Monitor_start メソッドを呼び出し、dns_probe メソッドを起動します。

この節では、サンプルアプリケーションの Monitor_start メソッドの重要な部分だけを説明します。parse_args() 関数や syslog 機能番号を取得する方法など、すべてのコールバックメソッドに共通な機能については説明しません。このような機能については、94 ページの「すべてのメソッドに共通な機能の提供」を参照してください。

Monitor_start メソッドの完全なリストについては、276 ページの「Monitor_start メソッド」 を参照してください。

Monitor_start の概要

このメソッドは、プロセス監視機能 (pmfadm) を使用して検証プログラムを起動します。

検証プログラムの起動

Monitor_start メソッドは、Rt_basedir プロパティの値を取得し、PROBE プログラムへの完全パス名を構築します。このメソッドは、pmfadm の無限再試行オプション (-n -1, -t -1) を使用して検証プログラムを起動します。つまり、検証プログラムの起動に失敗しても、PMF メソッドは検証プログラムを無限に再起動します。

```
# リソースの RT_BASEDIR プロパティを取得し、検証プログラムが存在する
# 場所を確認する。
RT_BASEDIR=‘scha_resource_get -O RT_BASEDIR -R $RESOURCE_NAME -G \
$RESOURCEGROUP_NAME‘

# PMF の制御下でデータサービスの検証を開始する。無限再試行オプションを使って
# 検証プログラムを起動する。リソースの名前、タイプ、グループを検証
# プログラムに渡す。
pmfadm -c $RESOURCE_NAME.monitor -n -1 -t -1 \
   $RT_BASEDIR/dns_probe -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME \
   -T $RESOURCETYPE_NAME
```

Monitor_stop メソッド

サンプルのデータサービスがオフラインになるとき、RGM は Monitor_stop メソッドを呼び出し、dns_probe の実行を停止します。

この節では、サンプルアプリケーションの Monitor_stop メソッドの重要な部分だけを説明します。parse_args() 関数や syslog 機能番号を取得する方法など、すべてのコールバックメソッドに共通な機能については説明しません。このような機能については、94 ページの「すべてのメソッドに共通な機能の提供」を参照してください。

Monitor_stop メソッドの完全なリストについては、278 ページの「Monitor_stop メソッド」 を参照してください。

Monitor_stop の概要

このメソッドは、プロセス監視機能 (pmfadm) を使用して検証プログラムが動作しているかどうかを判断し、動作している場合は検証プログラムを停止します。

検証プログラムの停止

Monitor_stop メソッドは、pmfadm -q を使用して検証プログラムが動作しているかどうかを判断し、動作している場合は pmfadm -s を使用して検証プログラムを停止します。検証プログラムがすでに停止している場合でも、このメソッドは成功状態でします。これによって、メソッドが呼び出し回数に依存しないことが保証されます。

```
# 検証プログラムが動作しているかどうかを判断し、動作している場合は停止する。
if pmfadm -q $PMF_TAG; then
   pmfadm -s $PMF_TAG KILL
```

```
    if [ $? -ne 0 ]; then
          logger -p ${SYSLOG_FACILITY}.err \
             -t [$SYSLOG_TAG] \
             "${ARGV0} Could not stop monitor for resource " \
             $RESOURCE_NAME
           exit 1
    else
          # 検証プログラムの停止に成功。メッセージを記録する。
          logger -p ${SYSLOG_FACILITY}.err \
             -t [$SYSLOG_TAG] \
             "${ARGV0} Monitor for resource " $RESOURCE_NAME \
             " successfully stopped"
    fi
fi
exit 0
```

注意 – 検証プログラムを停止するときは、必ず、pmfadm で KILL シグナルを使用するようにしてください。絶対に、マスク可能なシグナル (TERM など) は使用しないでください。そうしないと、Monitor_stop メソッドが無限にハングし、結果としてタイムアウトする可能性があります。この問題の原因は、PROBE メソッドがデータサービスを再起動またはフェイルオーバーする必要があるときに、scha_control() を呼び出すところにあります。scha_control() がデータサービスをオフラインにするプロセスの一部として Monitor_stop メソッドを呼び出したときに、Monitor_stop メソッドがマスク可能なシグナルを使用していると、Monitor_stop メソッドは scha_control() が終了するのを待ち、scha_control() は Monitor_stop メソッドが終了するのを待つため、結果として両方がハングします。

Monitor_stop の終了状態

PROBE メソッドを停止できない場合、Monitor_stop メソッドはエラーメッセージを記録します。RGM は、主ノード上でサンプルのデータサービスを MONITOR_FAILED 状態にするため、そのノードに障害が発生することがあります。

Monitor_stop メソッドは、検証プログラムが停止するまで終了してはなりません。

Monitor_check メソッド

PROBE メソッドが、データサービスを含むリソースグループを新しいノードにフェイルオーバーしようとするとき、RGM は Monitor_check メソッドを呼び出します。

この節では、サンプルアプリケーションの Monitor_check メソッドの重要な部分だけを説明します。parse_args() 関数や syslog 機能番号を取得する方法など、すべてのコールバックメソッドに共通な機能については説明しません。このような機能については、94 ページの「すべてのメソッドに共通な機能の提供」を参照してください。

Monitor_check メソッドの完全なリストについては、280 ページの「Monitor_check メソッド」を参照してください。

Monitor_check メソッドは、並行して実行中のその他のメソッドと競合しない方法で実装する必要があります。

Monitor_check メソッドは Validate メソッドを呼び出し、新しいノード上で DNS 構成ディレクトリが利用可能かどうかを確認します。Confdir 拡張プロパティが DNS 構成ディレクトリを指します。したがって、Monitor_check は Validate メソッドのパスと名前、および Confdir の値を取得します。Monitor_check は、次のように、この値を Validate に渡します。

```
# リソースタイプの RT_BASEDIR プロパティから Validate メソッドの
# 完全パスを取得する。
RT_BASEDIR=`scha_resource_get -O RT_BASEDIR -R $RESOURCE_NAME \
   -G $RESOURCEGROUP_NAMÈ

# 当該リソースの Validate メソッド名を取得する。
VALIDATE_METHOD=`scha_resource_get -O VALIDATE \
   -R $RESOURCE_NAME -G $RESOURCEGROUP_NAMÈ

# データサービスを起動するための Confdir プロパティの値を取得する。入力された
# リソース名とリソースグループを使用し、リソースを追加するときに設定した
# Confdir の値を取得する。
config_info=`scha_resource_get -O Extension -R $RESOURCE_NAME \
 -G $RESOURCEGROUP_NAME Confdir`

# scha_resource_get は、Confdir 拡張プロパティの値とともにタイプも戻す。
# awk を使用し、Confdir 拡張プロパティの値だけを取得する。
CONFIG_DIR=`echo $config_info | awk '{print $2}'`

# Validate メソッドを呼び出し、データサービスを新しいノードにフェイルオーバー
# できるかどうかを確認する。
$RT_BASEDIR/$VALIDATE_METHOD -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME \
   -T $RESOURCETYPE_NAME -x Confdir=$CONFIG_DIR
```

ノードがデータサービスのホストとして最適であるかどうかをサンプルアプリケーションが確認する方法については、114 ページの「Validate メソッド」を参照してください。

プロパティ更新の処理

サンプルのデータサービスは、クラスタ管理者によるプロパティの更新を処理するために、Validate メソッドと Update メソッドを実装します。

Validate メソッド

リソースが作成されたとき、および、リソースまたは (リソースを含む) リソースグループのプロパティが管理アクションによって更新されるとき、RGM は Validate メソッドを呼び出します。RGM は、作成または更新が行われる前に、Validate メソッドを呼び出します。任意のノード上でメソッドから失敗の終了コードが戻ると、作成または更新は取り消されます。

RGM が Validate メソッドを呼び出すのは、管理アクションがリソースまたはグループのプロパティを変更したときだけです。RGM がプロパティを設定したときや、モニターがリソースプロパティ Status と Status_msg を設定したときではありません。

注 – PROBE メソッドがデータサービスを新しいノードにフェイルオーバーする場合、Monitor_check メソッドも明示的に Validate メソッドを呼び出します。

Validate の概要

RGM は、他のメソッドに渡す追加の引数 (更新されるプロパティと値を含む) を指定して、Validate メソッドを呼び出します。したがって、サンプルのデータサービスの Validate メソッドは、追加の引数を処理する別の parse_args() 関数を実装する必要があります。

サンプルのデータサービスの Validate メソッドは、単一のプロパティである Confdir 拡張プロパティを確認します。このプロパティは、DNS が正常に動作するために重要な DNS 構成ディレクトリを指します。

注 – DNS が動作している間、構成ディレクトリは変更できないため、Confdir プロパティは RTR ファイルで TUNABLE = AT CREATION と宣言します。したがって、Validate メソッドが呼び出されるのは、更新の結果として Confdir プロパティを確認するためではなく、データサービスリソースが作成されているときだけです。

RGM が Validate メソッドに渡すプロパティの中に Confdir が存在する場合、parse_args() 関数はその値を取得および保存します。次に、Validate メソッドは、Confdir の新しい値が指すディレクトリがアクセス可能であるかどうか、および、named.conf ファイルがそのディレクトリ内に存在し、データを持っているかどうかを確認します。

parse_args() 関数が、RGM から渡されたコマンド行引数から Confdir の値を取得できない場合でも、Validate メソッドは Confdir プロパティの妥当性を検査しようとします。まず、Validate メソッドは scha_resource_get() 関数を使用し、静的な構成から Confdir の値を取得します。次に、同じ検査を実行し、構成ディレクトリがアクセス可能であるかどうか、および、空でない named.conf ファイルがそのディレクトリ内に存在するかどうかを確認します。

Validate メソッドが失敗で終了した場合、Confdir だけでなく、すべてのプロパティの更新または作成が失敗します。

Validate メソッドの構文解析関数

RGM は、ほかのコールバックメソッドとは異なるパラメータを Validate メソッドに渡します。したがって、Validate メソッドには、ほかのメソッドとは異なる引数を構文解析する関数が必要です。Validate メソッドやその他のコールバックメソッドに渡される引数の詳細については、rt_callbacks(1HA) のマニュアルページを参照してください。次に、 Validate メソッドの parse_args() 関数を示します。

```
#############################################################################
# Validate 引数の構文解析。
#
function parse_args # [args...]
{

   typeset opt
   while getopts 'cur:x:g:R:T:G:' opt
   do
         case "$opt" in
         R)
                  # DNS リソース名。
                  RESOURCE_NAME=$OPTARG
                  ;;
         G)
                  # リソースが構成されたリソースグループ名。
                  RESOURCEGROUP_NAME=$OPTARG
                  ;;
         T)
                  # リソースタイプ名。
                  RESOURCETYPE_NAME=$OPTARG
                  ;;
         r)
                  # メソッドはシステム定義プロパティ
                  # にアクセスしていない。したがって、
                  # このフラグは動作なし。
                  ;;
         g)
                  # メソッドはリソースグループプロパティに
                  # アクセスしていない。したがって、
                  # このフラグは動作なし。
                  ;;
         c)
                  # Validate メソッドがリソースの作成中に
                  # 呼び出されていることを示す。したがって、
                  # このフラグは動作なし。
                  ;;
         u)
                  # リソースがすでに存在しているときは、
                  # プロパティの更新を示す。Confdir
```

```
                # プロパティを更新する場合、Confdir
                # がコマンド行引数に現れる。現れない場合、メソッドは
                # scha_resource_get を使用して
                # Confdir を探す必要がある。
                UPDATE_PROPERTY=1
                ;;
        x)
                # 拡張プロパティのリスト。プロパティ
                # と値のペア。区切り文字は「=」
                PROPERTY=`echo $OPTARG | awk -F= '{print $1}'`
                VAL=`echo $OPTARG | awk -F= '{print $2}'`
                # Confdir 拡張プロパティがコマンド行
                # 上に存在する場合、その値を記録する。
                if [ $PROPERTY == "Confdir" ]; then
                        CONFDIR=$VAL
                        CONFDIR_FOUND=1
                fi
                ;;
        *)
                logger -p ${SYSLOG_FACILITY}.err \
                -t [$SYSLOG_TAG] \
                "ERROR: Option $OPTARG unknown"
                exit 1
                ;;
        esac
   done
}
```

ほかのメソッドの `parse_args()` 関数と同様に、この関数は、リソース名を取得するためのフラグ (`R`)、リソースグループ名を取得するためのフラグ (`G`)、RGM から渡されるリソースタイプを取得するためのフラグ (`T`) を提供します。

このメソッドはリソースが更新されるときに拡張プロパティの妥当性を検査するために呼び出されるため、`r` フラグ (システム定義プロパティを示す)、`g` フラグ (リソースグループプロパティを示す)、`c` フラグ (リソースの作成中に妥当性の検査が行われていることを示す) は無視されます。

`u` フラグは、`UPDATE_PROPERTY` シェル変数の値を `1` (`TRUE`) に設定します。`x` フラグは、更新されているプロパティの名前と値を取得します。更新されているプロパティの中に `Confdir` が存在する場合、その値が *CONFDIR* シェル変数に格納され、*CONFDIR_FOUND* 変数が `1` (`TRUE`) に設定されます。

Confdir の妥当性検査

`Validate` メソッドはまず、その `MAIN` 関数において、*CONFDIR* 変数を空の文字列に設定し、*UPDATE_PROPERTY* と *CONFDIR_FOUND* を 0 に設定します。

```
CONFDIR=""
UPDATE_PROPERTY=0
CONFDIR_FOUND=0
```

次に、`Validate` メソッドは `parse_args()` 関数を呼び出し、RGM から渡された引数を構文解析します。

```
parse_args "$@"
```

次に、Validate は、Validate メソッドがプロパティの更新の結果として呼び出されているのかどうか、および、Confdir 拡張プロパティがコマンド行上に存在するかどうかを検査します。次に、Validate メソッドは、Confdir プロパティが値を持っているかどうかを確認します。値を持っていない場合、Validate メソッドはエラーメッセージを記録し、失敗状態で終了します。

```
if ( (( $UPDATE_PROPERTY == 1 )) &&  (( CONFDIR_FOUND == 0 )) ); then
         config_info=`scha_resource_get -O Extension -R $RESOURCE_NAME \
            -G $RESOURCEGROUP_NAME Confdir`
         CONFDIR=`echo $config_info | awk '{print $2}'`
fi

# Confdir プロパティが値を持っているかどうかを確認する。持っていな
# い場合、状態1 (失敗) で終了する。
if [[ -z $CONFDIR ]]; then
         logger -p ${SYSLOG_FACILITY}.err \
            "${ARGV0} Validate method for resource "$RESOURCE_NAME " failed"
         exit 1
fi
```

注 – 上記コードにおいて、Validate メソッドが更新の結果として呼び出されているのか ($UPDATE_PROPERTY == 1)、および、プロパティがコマンド行上に存在しないのか (CONFDIR_FOUND == 0) を検査し、両者が TRUE である場合に、scha_resource_get() 関数を使用して Confdir の既存の値を取得するところに注目してください。Confdir がコマンド行上に存在する (CONFDIR_FOUND == 1) 場合、*CONFDIR* の値は、scha_resource_get() 関数からではなく、parse_args () 関数から取得されます。

次に、Validate メソッドは *CONFDIR* の値を使用し、ディレクトリがアクセス可能であるかどうかを確認します。アクセス可能ではない場合、Validate メソッドはエラーメッセージを記録し、エラー状態で終了します。

```
# $CONFDIR がアクセス可能であるかどうかを検査する。
if [ ! -d $CONFDIR ]; then
   logger -p ${SYSLOG_FACILITY}.err \
          -t [$SYSLOG_TAG] \
          "${ARGV0} Directory $CONFDIR missing or not mounted"
   exit 1
fi
```

Confdir プロパティの更新の妥当性を検査する前に、Validate メソッドは最終検査を実行し、named.conf ファイルが存在するかどうかを確認します。存在しない場合、Validate メソッドはエラーメッセージを記録し、エラー状態で終了します。

```
# named.conf ファイルがConfdir ディレクトリ内に存在するかどうかを
# 検査する。
if [ ! -s $CONFDIR/named.conf ]; then
         logger -p ${SYSLOG_FACILITY}.err \
            -t [$SYSLOG_TAG] \
```

```
            "${ARGV0} File $CONFDIR/named.conf is missing or empty"
         exit 1
fi
```

最終検査を通過した場合、Validate メソッドは、成功を示すメッセージを記録し、成功状態で終了します。

```
# Validate メソッドが成功したことを示すメッセージを記録する。
logger -p ${SYSLOG_FACILITY}.err \
   -t [$SYSLOG_TAG] \
   "${ARGV0} Validate method for resource "$RESOURCE_NAME \
   " completed successfully"

exit 0
```

Validate の終了状態

Validate メソッドが成功 (0) で終了すると、新しい値を持つ Confdir が作成されます。Validate メソッドが失敗 (1) で終了すると、Confdir を含むすべてのプロパティが作成されず、理由を示すメッセージがクラスタ管理者に送信されます。

Update メソッド

リソースのプロパティが変更されたとき、RGM は Update メソッドを呼び出し、動作中のリソースにその旨を通知します。RGM は、管理アクションがリソースまたはそのリソースグループのプロパティの設定に成功したあとに、Update を呼び出します。このメソッドは、リソースがオンラインであるノード上で呼び出されます。

Update の概要

Update メソッドはプロパティを更新しません。プロパティの更新は RGM が行います。その代わりに、動作中のプロセスに更新が発生したことを通知します。サンプルのデータサービスでは、プロパティの更新によって影響を受けるプロセスは障害モニターだけです。したがって、Update メソッドは、障害モニターを停止および再起動します。

Update メソッドは、障害モニターが動作していることを確認してから、pmfadm で障害モニターを強制終了する必要があります。UPDATE メソッドは、障害モニターを実装する検証プログラムの位置を取得し、その後、もう一度 pmfadm で障害モニターを再起動します。

Update による障害モニターの停止

Update メソッドは、pmfadm -q を使用し、障害モニターが動作していることを確認します。動作している場合、pmfadm -s TERM で障害モニターを強制終了します。障害モニターが正常に終了した場合、その影響を示すメッセージが管理ユーザーに送信されます。障害モニターが停止できない場合、Update メソッドは、エラーメッセージを管理ユーザーに送信し、失敗状態で終了します。

```
if pmfadm -q $RESOURCE_NAME.monitor; then

# すでに動作している障害モニターを強制終了する。
pmfadm -s $PMF_TAG TERM
    if [ $? -ne 0 ]; then
       logger -p ${SYSLOG_FACILITY}.err \
              -t [$SYSLOG_TAG] \
                 "${ARGV0} Could not stop the monitor"
       exit 1
    else
    # DNS の停止に成功。メッセージを記録する。
       logger -p ${SYSLOG_FACILITY}.err \
              -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME] \
                 "Monitor for HA-DNS successfully stopped"
    fi
```

障害モニターの再起動

障害モニターを再起動するために、Update メソッドは検証プログラムを実装するスクリプトの位置を見つける必要があります。検証プログラムはデータサービスのベースディレクトリ (Rt_basedir プロパティが指すディレクトリ) 内にあります。Update は、次に示すように、Rt_basedir の値を取得し、*RT_BASEDIR* 変数に格納します。

```
RT_BASEDIR=‘scha_resource_get -O RT_BASEDIR -R $RESOURCE_NAME -G \
$RESOURCEGROUP_NAME‘
```

次に、Update は、*RT_BASEDIR* の値を pmfadm で使用し、dns_probe プログラムを再起動します。検証プログラムを再起動できた場合、Update メソッドはその影響を示すメッセージを管理ユーザーに送信し、成功状態で終了します。pmfadm が検証プログラムを再起動できない場合、Update メソッドはエラーメッセージを記録し、失敗状態で終了します。

Update の終了状態

Update メソッドが失敗すると、リソースが “update failed” (更新失敗) の状態になります。この状態は RGM のリソース管理に影響しません。しかし、syslog 機能を通じて、管理ツールへの更新アクションが失敗したことを示します。

第 6 章

データサービス開発ライブラリ (DSDL)

この章では、データサービス開発ライブラリ(Data Service Development Library: DSDL) を構成するアプリケーションプログラミングインタフェースの概要について説明します。DSDL は `libdsdev.so` ライブラリとして実装されており、Sun Cluster パッケージに含まれています。

この章の内容は次のとおりです。

DSDL の概要

DSDL API は、RMAPI の最上位の階層を形成します。DSDL API は RMAPI の代わりになるものではなく、RMAPI をカプセル化および拡張するためのものであることに注意してください。DSDL は、特定の Sun Cluster 統合問題に対する事前定義されたソリューションを提供することによって、データサービスの開発を簡素化します。その結果、アプリケーションに本来求められている高可用性とスケーラビリティの実現に、より多くの開発時間を割くことが可能になります。また、アプリケーションの起動、シャットダウン、および監視機能を Sun Cluster に統合する際に、多くの時間を費やすこともありません。

構成プロパティの管理

すべてのコールバックメソッドは構成プロパティにアクセスする必要があります。DSDL は、以下により、プロパティへのアクセスを容易にします。

- 環境の初期化
- プロパティ値を簡単に取得できる関数セットの提供

`scds_initialize` 関数 (各コールバックメソッドの開始時に呼び出す必要がある) は、次の処理を行います。

- RGM がコールバックメソッドに渡すコマンド行引数 (`argc` と `argv[]`) を検査および処理します。そのため、コマンド行解析関数を作成する必要はありません。
- 他の DSDL 関数が使用できるように内部データ構造を設定します。たとえば、DSDL で提供されている関数によって RGM から取得されたプロパティ値はこのデータ構造に格納されます。同様に、コマンド行から入力された値 (RGM から取得された値よりも優先される) もこのデータ構造に格納されます。

注 – `Validate` メソッドの場合、scds_initialize はコマンド行で渡されたプロパティ値を解析します。そのため、`Validate` 用の解析関数を作成する必要はありません。

また、`scds_initialize` 関数はロギング環境を初期化して、障害モニターの検証設定の妥当性を検査します。

DSDL は、リソース、リソースタイプ、リソースグループのプロパティ、および、よく使用される拡張プロパティを取得するための関数セットを提供します。これらの関数は、次のような規則に従って、プロパティへのアクセスを標準化しています。

- 各関数は、`scds_initialize` から戻されるハンドル引数だけを取ります。
- 各関数は特定のプロパティに対応します。つまり、関数の戻り値のタイプは取得するプロパティ値のタイプに一致します。
- 値は `scds_initialize` によってあらかじめ算出されているため、関数はエラーを戻しません。新しい値がコマンド行で渡された場合を除き、関数は RGM から値を取得します。

データサービスの起動と停止

Start メソッドは、クラスタノード上でデータサービスを起動するために必要なアクションを実行します。通常、このようなアクションには、リソースプロパティの取得、アプリケーション固有の実行可能ファイルおよび構成ファイルの格納先の特定、および適切なコマンド行引数を用いたアプリケーションの起動が含まれます。

scds_initialize 関数はリソース構成を取得します。Start メソッドはプロパティ用の DSDL 関数を使用して、アプリケーションを起動するのに必要な構成ディレクトリや構成ファイルを識別するための特定のプロパティ (Confdir_list など) の値を取得します。

Start メソッドは、scds_pmf_start を呼び出して、プロセス監視機能 (PMF) の制御下でアプリケーションを起動します。PMF を使用すると、プロセスに適用する監視レベルを指定したり、異常終了したプロセスを再起動したりできます。DSDL で実装する Start メソッドの例については、142 ページの「xfnts_start メソッド」を参照してください。

Stop メソッドは呼び出し回数に依存しないように実装されていなければなりません。つまり、アプリケーションが動作していないときにノード上で呼び出された場合でも、正常終了する必要があります。Stop メソッドが失敗した場合、停止するリソースが STOP_FAILED 状態に設定され、クラスタの再起動を招いてしまう可能性があります。

リソースが STOP_FAILED 状態になるのを防止するために、Stop メソッドはあらゆる手段を構じてリソースを停止する必要があります。scds_pmf_stop 関数は、段階的にリソースを停止しようとします。まず、SIGTERM シグナルを使用してリソースを停止しようとします。これに失敗した場合は、SIGKILL シグナルを使用します。詳細は、scds_pmf_stop(3HA) のマニュアルページを参照してください。

障害モニターの実装

DSDL は、事前に定義されたモデルを提供することによって、障害モニターを実装する際の煩雑さをほとんど取り除きます。リソースがノード上で起動すると、Monitor_start メソッドは PMF の制御下で障害モニターを起動します。リソースがノード上で動作している間、障害モニターは無限ループを実行します。次に、DSDL 障害モニターのロジックの概要を示します。

- scds_fm_sleep 関数は Thorough_probe_interval プロパティを使用して、検証を行う期間を決定します。この期間中に PMF がアプリケーションプロセスの失敗を決定した場合、リソースは再起動されます。

- 検証機能自身は、障害の重要度を示す値を戻します。この値の範囲は、0 (障害なし) から 100 (致命的な障害) までです。
- 検証機能が戻した値は、scds_action 関数に送信されます。scds_action 関数は、Retry_interval プロパティの期間中に、障害の履歴を累積します。
- scds_action 関数は、次に示すような、障害が発生した場合の処置を決定します。
  - 累積した障害が 100 より少ない場合は、何もしません。
  - 累積した障害が 100 に到達した場合 (完全な障害)、データサービスを再起動します。Retry_interval を超えた場合、障害の履歴をリセットします。
  - Retry_interval で指定された期間中に、再起動の回数が Retry_count プロパティを上回った場合、データサービスをフェイルオーバーします。

ネットワークアドレス情報へのアクセス

DSDL は、リソースおよびリソースグループのネットワークアドレス情報を戻す関数を提供します。たとえば、scds_get_netaddr_list は、リソースが使用するネットワークアドレスリソースを取得して、障害モニターがアプリケーションを検証できるようにします。

また、DSDL は TCP ベースの監視を行う関数セットも提供します。通常、このような関数はサービスとの間に単純なソケット接続を確立し、サービスのデータを読み書きした後で、サービスとの接続を切断します。検証の結果を DSDL の scds_fm_action 関数に送信し、次に実行すべき処理を決定できます。

TCP ベースの障害監視の例については、156 ページの「xfnts_validate メソッド」を参照してください。

実装したリソースタイプのデバッグ

DSDL は、データサービスをデバッグするときに役立つ組み込み機能を提供します。

DSDL の scds_syslog_debug() ユーティリティーは、実装したリソースタイプにデバッグ文を追加するための基本的なフレームワークを提供します。デバッグレベル (1 から 9 までの数字) は、各クラスタノード上のリソースタイプごとに動的に設定できます。ファイル /var/cluster/rgm/rt/*rtname*/loglevel は、1 から 9 までの整数だけが含まれているファイルであり、すべてのリソースタイプコールバックメ

ソッドはこのファイルを読み取ります。DSDL の `scds_initialize()` ルーチンはこのファイルを読み取って、内部デバッグレベルを指定されたレベルに設定します。デフォルトのデバッグレベルは 0 であり、この場合、データサービスはデバッグメッセージを記録しません。

`scds_syslog_debug()` 関数は、`LOG_DEBUG` の優先順位において、`scha_cluster_getlogfacility()` 関数から戻された機能を使用します。このようなデバッグメッセージは `/etc/syslog.conf()` で構成できます。

`scds_syslog` ユーティリティーを使用すると、いくつかのデバッグメッセージをリソースタイプの通常の動作 (おそらくは LOG_INFO 優先順位) における情報メッセージとして使用することができます。第 8 章 のサンプル DSDL アプリケーションでは、`scds_syslog_debug` と `scds_syslog` 関数が多用されています。

高可用性ローカルファイルシステムの有効化

`HAStoragePlus` リソースタイプを使用すると、ローカルファイルシステムを Sun Cluster 環境内で高可用性にすることができます。このためには、ローカルファイルシステムのパーティションを広域ディスクグループ内に配置しなければなりません。また、アフィニティスイッチオーバーを有効にし、Sun Cluster 環境をフェイルオーバー用に構成する必要もあります。これによって、多重ホストディスクに直接接続された任意のホストから、多重ホストディスク上の任意のファイルシステムにアクセスできるようになります。入出力が多いデータサービスでは、高可用性のローカルファイルシステムを使用することを強く推奨します。`HAStoragePlus` リソースタイプの構成については、『*Sun Cluster 3.1* データサービスの計画と管理』の「高可用性ローカルファイルの実現」を参照してください。

第 7 章

リソースタイプの設計

この章では、リソースタイプの設計や実装で DSDL を通常どのように使用するかについて説明します。また、リソース構成を検証したり、リソースの開始、停止、および監視を行なったりするためのリソースタイプの設計についても説明します。そして、最後に、リソースタイプのコールバックメソッドを DSDL を使って導入する方法を説明します。

詳細は、`rt_callbacks(1HA)` のマニュアルページを参照してください。

これらの作業を行うには、リソースのプロパティ設定値にアクセスできなければなりません。DSDL ユーティリティー `scds_initialize()` を使用すると、統一された方法でリソースプロパティにアクセスできます。この機能は、各コールバックメソッドの始めの部分で呼び出す必要があります。このユーティリティー関数は、クラスタフレームワークからリソースのすべてのプロパティを取り出します。これによって、これらのプロパティは、`scds_get`*name*`()` 関数群からアクセスできるようになります。

この章の内容は次のとおりです。

RTR ファイル

RTR (Resource Type Registration 、リソースタイプ登録) ファイルは、リソースタイプの重要コンポーネントです。Sun Cluster は、リソースタイプの詳細な情報をこのファイルから取得します。この詳細情報には、この実装に必要なプロパティや、それらのデータタイプやデフォルト値、リソースタイプの実装に必要なコールバックメソッドのファイルシステムパス、システム定義プロパティのさまざまな設定値などがあります。

ほとんどのリソースタイプ実装では、DSDL に添付されるサンプル RTR ファイルだけで十分なはずです。必要な作業は、基本的な要素 (リソースタイプ名、リソースタイプのコールバックメソッドのパス名など) の編集だけです。リソースタイプを実装する際に新しいプロパティが必要な場合は、そのプロパティをリソースタイプ実装のリソースタイプ登録 (RTR) ファイルに拡張プロパティとして宣言します。新しいプロパティには、DSDL の `scds_get_ext_property()` ユーティリティーを使ってアクセスできます。

`Validate` メソッド

リソースタイプ実装の `Validate` メソッドは、 1) リソースタイプの新しいリソースが作成されているときや、2) リソースまたはリソースグループのプロパティが更新されているときにそれぞれ RGM から呼び出されます。この 2 つの操作は、リソースの `Validate` メソッドに渡されるコマンド行オプション `-c` (作成) と `-u` (更新) によって区別されます。

`Validate` メソッドは、リソースタイププロパティ `INIT_NODES` の値で定義されるノード群の各ノードに対して呼び出されます。たとえば、`INIT_NODES` に `RG_PRIMARIES` が設定されている場合、`Validate` は、そのリソースのリソースグループを収容できる (その主ノードになりうる) 各ノードに対して呼び出されます。`INIT_NODES` が `RT_INSTALLED_NODES` に設定されている場合、 `Validate` は、リソースタイプソフトウェアがインストールされている各ノード (通常は、クラスタのすべてのノード) に対して呼び出されます。`INIT_NODES` のデフォルト値は `RG_PRIMARIES` です (`rt_reg`(4) のマニュアルページを参照)。`Validate` メソッドが呼び出される時点では、RGM はまだリソースを作成していません (作成コールバックの場合)。あるいは、更新するプロパティの更新値をまだ適用していません (更新コールバックの場合)。リソースタイプ実装の `Validate` コールバックメソッドの目的は、リソースの新しい設定値 (リソースに対して指定された新しいプロパティ設定値) がそのリソースタイプにとって有効であるかどうかを検査することにあります。

注 – HAStoragePlus によって管理されるローカルファイルシステムを使用している場合は、scds_hasp_check を使って HAStoragePlus リソースの状態を検査します。この情報は、そのリソースが依存するすべての SUNW.HAStoragePlus(5) リソースの状態 (オンラインかどうか) から、そのリソースに定義されたシステムプロパティ Resource_dependencies または Resource_dependencies_weak を使って取得されます。scds_hasp_check 呼び出しから返される状態コードの完全なリストについては、scds_hasp_check(3HA) のマニュアルページを参照してください。

DSDL 関数 scds_initialize() は、リソースの作成や更新をそれぞれ次のように処理します。

- リソースの作成では、コマンド行から渡された新しいリソースプロパティを解析します。これによって、リソースタイプの開発者は、リソースプロパティの新しい値を、そのリソースががすでにシステムに作成されているかのように使用できます。
- リソースやリソースグループの更新では、管理者によって更新されようとしているプロパティの新しい値をコマンド行から読み込み、残りのプロパティ (値が更新されないもの) をリソース管理 API を使って Sun Cluster から読み込みます。ただし、リソースタイプの開発者は、DSDL を使用する限り、このような初期作業を行う必要はありません。さらに、開発者は、リソースのすべてのプロパティが使用可能であるものとして、リソースの検証を行うことができます。

次の図に示す svc_validate() は、リソースプロパティの検証を行う関数です。この関数は、scds_get_*name*() 関数群を使って、検証しようとするプロパティを検査します。リソースの設定が有効ならこの関数から戻りコード 0 が返されるとすると、リソースタイプの Validate メソッドは、次のコード部分のようになります。

```
in
tmain(int argc, char *argv[])
{
   scds_handle_t handle;
   int rc;

   if (scds_initialize(&handle, argc, argv)!= SCHA_ERR_NOERR) {
   return (1);   /* 初期化エラー */
   }
   rc = svc_validate(handle);
   scds_close(&handle);
   return (rc);
}
```

さらに、検証関数は、リソースの設定が有効でない場合は、その理由を記録する必要があります。svc_validate() 関数の例 (詳細は省略) は、次のようになります (実際的な検証ルーチンについては、次の章を参照)。

```
int
svc_validate(scds_handle_t handle)
{
   scha_str_array_t *confdirs;
   struct stat    statbuf;
```

```
   confdirs = scds_get_confdir_list(handle);
   if (stat(confdirs->str_array[0], &statbuf) == -1) {
   return (1);   /* 無効なリソースプロパティ設定 */
   }
   return (0);   /* 有効な設定 */
}
```

このように、リソースタイプの開発者は、`svc_validate()` 関数を使用することだけに集中できます。リソースタイプ実装の典型的な例としては、app.conf というアプリケーション構成ファイルを `Confdir_list` プロパティの下に置く処理があります。この処理は、`Confdir_list` プロパティから取り出した適切なパス名に対して `stat()` システム呼び出しを実行することによって実装できます。

Start メソッド

リソースタイプ実装の `Start` コールバックメソッドは、特定のクラスタノードのリソースを開始するときに RGM によって呼び出されます。リソースグループ名とリソース名、およびリソースタイプ名はコマンド行から渡されます。`Start` メソッドは、クラスタノードでデータサービスリソースを開始するために必要なアクションを行います。通常、このようなアクションには、リソースプロパティの取得や、アプリケーション固有の実行可能ファイルと構成ファイル (または、どちらか) の場所の特定、適切なコマンド行引数を使用したアプリケーションの起動などがあります。

DSDL では、リソース構成ファイルが `scds_initialize()` ユーティリティーによってすでに取得されています。アプリケーションの起動アクションは、`svc_start()` 関数に指定できます。さらに、アプリケーションが実際に起動されたかどうかを確認するために、`svc_wait()` 関数を呼び出すことができます。`Start` メソッドのコード (詳細は省略) は、次のようになります。

```
int
main(int argc, char *argv[])
{
   scds_handle_t handle;

   if (scds_initialize(&handle, argc, argv)!= SCHA_ERR_NOERR) {
   return (1);   /* 初期化エラー */
   }
   if (svc_validate(handle) != 0) {
   return (1);   /* 無効な設定 */
   }
   if (svc_start(handle) != 0) {
   return (1);   /* 起動に失敗 */
   }
   return (svc_wait(handle));
}
```

この起動メソッドの実装では、svc_validate() を呼び出してリソース構成を検証します。検証結果が正しくない場合は、リソース構成とアプリケーション構成が一致していないか、このクラスタノードのシステムに関して何らかの問題があることを示しています。たとえば、リソースに必要な広域ファイルシステムが現在このクラスタノードで使用できない可能性などが考えられます。その場合には、このクラスタノードでこのリソースを起動しても意味がないので、RGM を使って別のノードのリソースを起動すべきです。ただし、この場合、svc_validate()は十分に限定的であるものとします。その場合、このルーチンは、アプリケーションが必要とするリソースがあるかどうかをそのクラスタノードだけで検査します。そうでないと、このリソースはすべてのクラスタノードで起動に失敗し、START_FAILED の状態になる可能性があります。この状態については、scswitch(1M) のマニュアルページおよび『*Sun Cluster 3.1* データサービスの計画と管理』を参照してください。

svc_start() 関数は、このノードでリソースの起動に成功した場合は戻りコード 0 を、問題を検出した場合は 0 以外の戻りコードをそれぞれ返す必要があります。この関数から 0 以外の値が返されると、RGM は、このリソースを別のクラスタノードで起動しようと試みます。

DSDL を最大限に活用するには、svc_start() 関数で scds_pmf_start() ユーティリティーを使って、アプリケーションを PMF (プロセス管理機能) のもとで起動できます。このユーティリティーは、PMF の障害コールバックアクション機能 (pmfadm(1M) の -a アクションフラグを参照) を使って、プロセス障害の検出を実装します。

Stop メソッド

リソースタイプ実装の Stop コールバックメソッドは、特定のクラスタノードでアプリケーションを停止するときに RGM によって呼び出されます。Stop メソッドのコールバックが有効であるためには、次の条件が必要です。

- Stop メソッドは結果に依存しない命令 *(idempotent)* でなければなりません。つまり、Stop メソッドは、そのノードで Start メソッドが正常に終了していなくても、RGM から呼び出されることがあります。したがって、Stop メソッドは、そのクラスタノードでアプリケーションが動作していない場合でも (したがって、特別な処理が必要ない場合でも)、正常に (終了コード 0 で) 終了しなければなりません。
- リソースタイプの Stop メソッドが特定のクラスタノードで失敗に終わると (戻りコードが 0 以外だと)、そのリソースタイプは STOP_FAILED の状態になります。この場合、リソースの Failover_mode 設定によっては、このクラスタノードが RGM によって強制的に再起動されることがあります。したがって、Stop メソッドの設計時には、アプリケーションを停止する手段をメソッドに組み込んでおくことが重要です。たとえば、アプリケーションが停止しない場合は、 SIGKILL などを使って、アプリケーションを強制的かつ即時に停止する必要があります。さらに、この処理は一定の時間内に行われなければなりません。Stop_timeout で

設定した時間が経過すると、停止が失敗したものとみなされ、リソースは STOP_FAILED の状態になるからです。

ほとんどのアプリケーションには、DSDL ユーティリティー scds_pmf_stop() で十分なはずです。このユーティリティーは、まず、アプリケーションが PMF の scds_pmf_start() で起動されたものとみなして、アプリケーションを SIGTERM で「静かに」停止しようとします。これで停止しない場合は、プロセスに対して SIGKILL を適用します。このユーティリティーの詳細については、206 ページの「PMF 関数」を参照してください。

アプリケーションを停止するそのアプリケーション固有の関数を svc_stop() とし、これまで使用してきたコードモデルに従うとするなら、Stop メソッドは、次のように実装できます。svc_stop() の実装で scds_pmf_stop() が使用されているかどうかは、ここでは関係ありません。それが使用されているかどうかは、アプリケーションが PMF のもとで Start メソッドによって起動されているかどうかに依存します。

```
if (scds_initialize(&handle, argc, argv)!= SCHA_ERR_NOERR)
{
   return (1);   /* 初期化エラー */
}
return (svc_stop(handle));
```

Stop メソッドの実装では、svc_validate() メソッドは使用されません。システムに問題があったとしても、Stop メソッドは、このノードでこのアプリケーションを停止すべきだからです。

Monitor_start メソッド

RGM は、Monitor_start メソッドを呼び出して、リソースに対する障害モニターを起動します。障害モニターは、このリソースによって管理されているアプリケーションの状態を監視します。リソースタイプの実装では、通常、障害モニターはバックグラウンドで動作する独立したデーモンとして実装されます。このデーモンの起動には、適切な引数をもつ Monitor_start コールバックメソッドが使用されます。

モニターデーモン自体は障害が発生しやすいため (たとえば、モニターは、アプリケーションを、監視されない状態にしたまま停止することがある)、モニターデーモンは、PMF を使って起動すべきです。DSDL ユーティリティー scds_pmf_start() には、障害モニターを起動する機能が組み込まれています。このユーティリティーは、モニターデーモンプログラムの相対パス名 (リソースタイプコールバックメソッド実装の場所を表す RT_basedir との相対パス) を使用します。さらに、ユーティリティーは、DSDL によって管理される Monitor_retry_interval と Monitor_retry_count 拡張プロパティを使って、デーモンが際限なく再起動されるのを防止します。モニターデーモンのコマンド行構文には、コールバックメソッド

に対して定義されたコマンド行構文と同じものが使用されます (`-R` *resource* `-G` *resource_group* `-T` *resource_type*)。ただし、モニターデーモンが RGM から直接呼び出されることはありません。このユーティリティーでは、モニターデーモン実装自体が `scds_initialize()` ユーティリティーを使って独自の環境を設定できます。したがって、主な作業は、モニターデーモン自体を設計することです。

Monitor_stop メソッド

RGM は、`Monitor_stop` メソッドを使って、`Monitor_start` メソッドで起動された障害モニターデーモンを停止します。このコールバックメソッドの失敗は、`Stop` メソッドの失敗とまったく同じように処理されます。したがって、`Monitor_stop` メソッドは、`Stop` メソッドと同じように強固なものでなければなりません。

障害モニターデーモンを `scds_pmf_start()` ユーティリティーを使って起動したら、`scds_pmf_stop()` ユーティリティーで停止する必要があります。

Monitor_check メソッド

クラスタノードが特定のリソースを支配できるかどうかを確認するために (つまり、そのリソースによって管理されるアプリケーションがそのノードで正常に動作するかどうかを確認するために)、そのリソースの `Monitor_check` コールバックメソッドがそのリソースのノードで呼び出されます。通常、この状況では、アプリケーションに必要なすべてのシステムリソースが本当にクラスタノードで使用可能かどうかが確認されます。128 ページの「`Validate` メソッド」 で述べたように、開発者が使用する `svc_validate()` 関数では、少なくともこの確認が行われなければなりません。

リソースタイプ実装によって管理されているアプリケーションによっては、`Monitor_check` メソッドでその他の作業を行うことがあります。`Monitor_check` メソッドは、並行して実行中のその他のメソッドと競合しない方法で実装する必要があります。DSDL を使用する場合には、リソースプロパティに対するアプリケーション固有の検証を行なうために作成された `svc_validate()` 関数を `Monitor_check` メソッドで活用することをお勧めします。

Update メソッド

RGM は、リソースタイプ実装の Update メソッドを呼び出して、システム管理者が行なったすべての変更をアクティブリソースの構成に適用します。Update メソッドは、そのリソースがオンラインになっているすべてのノードに対して呼び出されます。

リソースの構成に対して行われた変更は、リソースタイプ実装にとって必ず有効なものです。RGM は、リソースタイプの Update メソッドを呼び出す前に Validate メソッドを呼び出すからです。Validate メソッドは、リソースやリソースグループのプロパティが変更される前に呼び出されます。したがって、Validate メソッドは新しい変更を拒否できます。変更が適用されると、Update メソッドが呼び出され、新しい設定値がアクティブ (オンライン) リソースに通知されます。

リソースタイプの開発者は、どのプロパティを動的に変更できるようにするかを慎重に決定し、RTR ファイルでこれらのプロパティに TUNABLE = ANYTIME を設定する必要があります。通常、障害モニターデーモンによって使用される、リソースタイプ実装のプロパティは、すべて動的に変更できるように設定できます。ただし、Update メソッドの実装が少なくともモニターデーモンを再起動できなければなりません。

このようなプロパティの候補には次のものがあります。

- Thorough_Probe_Interval
- Retry_Count
- Retry_Interval
- Monitor_retry_count
- Monitor_retry_interval
- Probe_timeout

これらのプロパティは、障害モニターデーモンがサービスの状態検査をどのような頻度でどのように行うかや、どのような履歴間隔でエラーを追跡するか、PMF によってどのような再起動しきい値が設定されるかなどに影響します。DSDL には、これらのプロパティの更新を行なうための scds_pmf_restart() ユーティリティーが備わっています。

リソースプロパティを動的に更新できなければならないがプロパティの変更によって動作中のアプリケーションに影響が及ぶ可能性があるという場合は、適切なアクションを行なうことによって、プロパティに対する変更が動作中のアプリケーションインスタンスに正しく適用されるようにしなければなりません。DSDL には、現在のところ、この問題をサポートする機能はありません。変更されたプロパティをコマンド行から Update に渡すことはできません (Validate に渡すことは可能)。

Init、Fini、 Boot の各メソッド

これらのメソッドは、「1 度だけのアクション」を行うためのものです (リソース管理 API 仕様の定義を参照)。DSDL のサンプル実装には、これらのメソッドの使い方は示されていません。しかし、これらのメソッドを使用する必要がある場合には、DSDL のすべての機能をこれらのメソッドでも使用できます。通常、「1 度だけのアクション」を使用するリソースタイプ実装では、`Init` メソッドと `Boot` メソッドはまったく同じように機能します。`Fini` メソッドは、一般に、`Init` メソッドや `Boot` メソッドのアクションを「取り消す」ためのアクションに使用されます。

障害モニターデーモンの設計

DSDL を使用したリソースタイプ実装の障害モニターデーモンには、通常、次の役割があります。

- 管理されているアプリケーションの状態を定期的に監視します。モニターデーモンのこの役割はアプリケーションに大きく依存します。したがって、リソースタイプによって大幅に異なることがあります。DSDL には、TCP に基づく簡単なサービスの状態を検査するいくつかのユーティリティー関数が組み込まれています。HTTP、NNTP、IMAP、POP3 など、ASCII ベースのプロトコルを実装するアプリケーションは、これらのユーティリティーを使って実装できます。
- アプリケーションによって検出された問題をリソースプロパティ `Retry_interval` や `Retry_count` を使って追跡します。さらに、アプリケーションが異常停止した場合には、PMF アクションスクリプトを使ってサービスを再起動すべきかどうかや、アプリケーションの障害が頻繁に発生するためにフェイルオーバーを考慮すべきかどうかを判断します。DSDL では、この機構の使用を助けるユーティリティーとして `scds_fm_action()` と `scds_fm_sleep()` が提供されます。
- アプリケーションを再起動するか、リソースを含むリソースグループのフェイルオーバーを試みるなど、適切なアクションを実行します。DSDL ユーティリティー `scds_fm_action()` には、このようなアルゴリズムが使用されています。そのために、このアルゴリズムは、過去の Retry_interval で指定した秒数の間に起った検証障害の累積を計算します。
- リソースの状態を更新します。これによって、`scstat` コマンドやクラスタ管理 GUI からアプリケーションの状態を知ることができます。

DSDL ユーティリティーの設計では、障害モニターデーモンの主要ループは次のようになっています。

DSDL を使って実装された障害モニターでは、

- アプリケーションプロセスの異常停止は、scds_fm_sleep() によって比較的迅速に検出されます。これは、PMF によるプロセス停止の通知が非同期に行われるためです。これは、障害モニターが時々リープから復帰してサービスの状態を検査し、アプリケーションの停止を検出する方法と対象的です。DSDL を使用した障害モニターでは障害検出時間が大幅に短縮されるため、サービスの可用性が向上します。
- RGM が scha_control(3HA) API によるサービスのフェイルオーバーを拒否すると、scds_fm_action() は、現在の障害履歴を「リセット」(消去) します。このようにするのは、障害履歴が Retry_count の値をすでに超えているからです。さらに、モニターデーモンは、次のサイクルでスリープから復帰した後に、デーモンの状態検査を正常に完了できないと、scha_control() を再び呼び出そうとするはずです。しかし、前回のサイクルで呼び出しが拒否され状況が依然として残っていれば、この呼び出しは今回も拒否されるはずです。履歴がリセットされていれば、障害モニターは、少なくとも、次のサイクルでアプリケーションの再起動などによってその状況を内部的に訂正しようとします。
- 再起動が失敗に終わった場合、scds_fm_action() は、アプリケーション障害履歴をリセットしません。これは、状況が訂正されなければ、scha_control() が間もなく呼び出される可能性が高いからです。
- ユーティリティー scds_fm_action() は、障害履歴に従って、SCHA_RSSTATUS_OK、SCHA_RSSTATUS_DEGRADED、SCHA_RSSTATUS_FAULTED のどれかをリソースステータスに設定します。これによって、ステータスをクラスタシステム管理から使用できるようになります。

ほとんどの場合、アプリケーション固有の状態検査アクションは、スタンドアロンの別個のユーティリティー (たとえば、svc_probe()) として実装してから、この汎用的なメインループに統合できます。

```
for (;;) {

   / * 正常な検証と検証の間の thorough_probe_interval
   *  だけスリープする。*/
   (void) scds_fm_sleep(scds_handle,
   scds_get_rs_thorough_probe_interval(scds_handle));

   /* 使用するすべてのipaddress を検証する。
   * 次の各要素を繰り返し検証する。
   * 1. 使用するすべてのネットリソース
   * 2. 特定のリソースのすべてのipaddresses
   * 検証するipaddress ごとに
   * 障害履歴を計算する。*/
   probe_result = 0;
    /* すべてのリソースを繰り返し調べて、
    * svc_probe() の呼び出しに使用する各IP アドレスを取得する。*/
   for (ip = 0; ip < netaddr->num_netaddrs; ip++) {
    /* 状態を検証する必要があるホスト名とポート
    * を取得する。
    */
   hostname = netaddr->netaddrs[ip].hostname;
   port = netaddr->netaddrs[ip].port_proto.port;
   /*
   * HA-XFS は1 つのポートしかサポートしないため、
```

```
* ポート配列の最初のエントリから
* ポート値を取得する。
*/
ht1 = gethrtime(); /* Latch probe start time */
probe_result = svc_probe(scds_handle,

hostname, port, timeout);
/*
* サービス検証履歴を更新し、
* 必要に応じてアクションを実行する。
* 検証終了時刻を保存する。
*/
ht2 = gethrtime();
/* ミリ秒に変換する。*/
dt = (ulong_t)((ht2 - ht1) / 1e6);

/*
* 障害履歴を計算し、
* 必要に応じてアクションを実行する。
*/
(void) scds_fm_action(scds_handle,
probe_result, (long)dt);
}       /* 各ネットワークリソース */
}       /* 検証を続ける。*/
```

第 8 章

サンプル DSDL リソースタイプの実装

この章では、DSDL で実装したサンプルのリソースタイプ SUNW.xfnts について説明します。データサービスは C 言語で作成されています。使用するアプリケーションは TCP/IP ベースのサービスである X Font Server です。

この章の内容は、次のとおりです。

X Font Server について

X Font Server は、フォントファイルをクライアントに提供する、簡単な TCP/IP ベースのサービスです。クライアントはサーバーに接続してフォントセットを要求します。サーバーはフォントファイルをディスクから読み取って、クライアントにサービスを提供します。X Font Server デーモンはサーバーバイナリである /usr/openwin/bin/xfs から構成されます。このデーモンは通常、inetd から起動されますが、このサンプルでは、/etc/inetd.conf ファイル内の適切なエントリが (たとえば、fsadmin -d コマンドによって) 無効にされているものと想定します。したがって、デーモンは Sun Cluster だけの制御下にあります。

X Font Server の構成ファイル

デフォルトでは、X Font Server はその構成情報をファイル /usr/openwin/lib/X11/fontserver.cfg から読み取ります。 このファイルのカタログエントリには、デーモンがサービスを提供できるフォントディレクトリのリストが入っています。クラスタ管理者は広域ファイルシステム上のフォントディレクトリの格納先を指定できます。こうすることによって、システム上でフォントデータベースのコピーを 1 つだけ保持すれば済むので、Sun Cluster 上の X Font Server の使用を最適化できます。広域ファイルシステム上のフォントディレクトリの格納先を指定するには、fontserver.cfg を編集して、フォントディレクトリの新しいパスを反映させる必要があります。

構成を簡単にするために、管理者は構成ファイル自身も広域ファイルシステム上に配置できます。xfs デーモンはデフォルトの格納先 (このファイルの組み込み場所) を変更するためのコマンド行引数を提供します。SUNW.xfnts リソースタイプは、次のコマンドを使用して、Sun Cluster の制御下でデーモンを起動します。

```
/usr/openwin/bin/xfs -config <location_of_cfg_file>/fontserver.cfg \
-port <portnumber>
```

SUNW.xfnts v リソースタイプの実装では、Confdir_list プロパティを使用して、fontserver.cfg 構成ファイルの格納場所を管理できます。

TCP ポート番号

xfs サーバーデーモンが通信する TCP ポートの番号は、一般に「fs」ポート (通常、/etc/services ファイルで 7100 と定義されている) です。ただし、xfs コマンド行で -port オプションを指定することにより、システム管理者はデフォルトの設定を変更できます。SUNW.xfnts リソースタイプの Port_list プロパティを使用すると、デフォルト値を設定したり、xfs コマンド行で -port オプションを指定できるようになります。RTR ファイルにおいて、このプロパティのデフォルト値を 7100/tcp と定義します。SUNW.xfnts の Start メソッドで、Port_list を xfs コマンド行の -port オプションに渡します。このようにすると、このリソースタイプのユーザーはポート番号を指定する必要がなくなります。つまり、デフォルトのポートが 7100/tcp になります。ただし、リソースタイプを構成するときに、Port_list プロパティに異なる値を指定することにより、別のポートを指定することも可能です。

命名規約

次の命名規則を覚えておけば、サンプルコード内で関数とメソッドを簡単に識別できます。

- RMAPI 関数の名前は、scha_ で始まります。

- DSDL 関数の名前は、scds_ で始まります。
- コールバックメソッドの名前は、xfnts_ で始まります。
- ユーザー定義関数の名前は、svc_ で始まります。

SUNW.xfnts の RTR ファイル

この節では、SUNW.xfnts の RTR ファイルで宣言されている、いくつかの重要なプロパティについて説明します。各プロパティの目的については説明しません。プロパティの詳細については、34 ページの「リソースとリソースタイププロパティの設定」を参照してください。

次に示すように、Confdir_list 拡張プロパティは構成ディレクトリ (または、ディレクトリのリスト) を指定します。

```
{
        PROPERTY = Confdir_list;
        EXTENSION;
        STRINGARRAY;
        TUNABLE = AT_CREATION;
        DESCRIPTION = "The Configuration Directory Path(s)";
}
```

Confdir_list プロパティには、デフォルト値は設定されていません。クラスタ管理者はリソースを作成するときに、構成 ディレクトリを指定する必要があります。「TUNABLE = AT_CREATION」が指定されているので、作成時以降、この値を変更することはできません。

次に示すように、Port_list プロパティは、サーバーデーモンがリッスンするポートを指定します。

```
{
        PROPERTY = Port_list;
        DEFAULT = 7100/tcp;
        TUNABLE = AT_CREATION;
}
```

このプロパティにはデフォルト値が設定されているため、クラスタ管理者はリソースを作成するときに、新しい値を指定するか、デフォルト値を使用するかを選択します。「TUNABLE = AT_CREATION」が指定されているので、作成時以降、この値を変更することはできません。

scds_initialize() 関数

DSDL では、各コールバックメソッドがメソッドの開始時に scds_initialize (3HA) 関数を呼び出す必要があります。この関数は次の作業を行います。

- フレームワークがデータサービスメソッドに渡すコマンド行引数 (argc と argv) を検査および処理します。そのため、データサービスメソッドは、コマンド行引数について追加の処理を実行する必要はありません。
- 他の DSDL 関数が使用できるように内部データ構造を設定します。
- ロギング環境を初期化します。
- 障害モニターの検証設定の妥当性を検査します。

scds_close() 関数を使用すると、scds_initialize() が割り当てたリソースを再利用できます。

xfnts_start メソッド

データサービスリソースを含むリソースグループがオンラインになったとき、あるいは、リソースが有効になったとき、RGM はそのクラスタノード上で Start メソッドを呼び出します。サンプルの SUNW.xfnts リソースタイプでは、xfnts_start メソッドが当該ノード上で xfs デーモンを起動します。

xfnts_start メソッドは scds_pmf_start() を呼び出して、PMF の制御下でデーモンを起動します。PMF は、自動障害通知、再起動機能、および障害モニターとの統合を提供します。

注 – xfnts_start は、scds_initialize() を最初に呼び出します。これによって、いくつかのハウスキーピング関数が実行されます。詳細については、142 142 ページの「scds_initialize() 関数」と、scds_initialize(3HA) のマニュアルページを参照してください。

起動前のサービスの検証

次に示すように、xfnts_start メソッドは X Font Server を起動する前に svc_validate() を呼び出して、xfs デーモンをサポートするための適切な構成が存在していることを確認します。詳細については、156 ページの「xfnts_validate メソッド」を参照してください。

```
rc = svc_validate(scds_handle);
   if (rc != 0) {
      scds_syslog(LOG_ERR,
          "Failed to validate configuration.");
      return (rc);
   }
```

サービスの起動

xfnts_start メソッドは、xfnts.c で定義されている svc_start() メソッドを呼び出して、xfs デーモンを起動します。ここでは、svc_start() について説明します。

以下に、xfs デーモンを起動するためのコマンドを示します。

```
xfs -config config_directory/fontserver.cfg -port port_number
```

Confdir_list 拡張プロパティには *config_directory* を指定します。一方、Port_list システムプロパティには *port_number* を指定します。クラスタ管理者はデータサービスを構成するときに、これらのプロパティの値を指定します。

次に示すように、xfnts_start メソッドはこれらのプロパティを文字列配列として宣言し、scds_get_ext_confdir_list() と scds_get_port_list() 関数 (scds_property_functions(3HA)) で説明されている) を使用して、管理者が設定した値を取得します。

```
scha_str_array_t *confdirs;
scds_port_list_t    *portlist;
scha_err_t   err;

   /* Confdir_list プロパティから構成ディレクトリを取得する。*/
   confdirs = scds_get_ext_confdir_list(scds_handle);

   (void) sprintf(xfnts_conf, "%s/fontserver.cfg", confdirs->str_array[0]);

   /* Port_list プロパティから XFS が使用するポートを取得する。*/
   err = scds_get_port_list(scds_handle, &portlist);
   if (err != SCHA_ERR_NOERR) {
      scds_syslog(LOG_ERR,
          "Could not access property Port_list.");
      return (1);
   }
```

confdirs 変数は配列の最初の要素 (0) を指していることに注意してください。

次に示すように、xfnts_start メソッドは sprintf を使用して、xfs 用のコマンド行を形成します。

```
/* xfs デーモンを起動するコマンドを構築する。*/
   (void) sprintf(cmd,
       "/usr/openwin/bin/xfs -config %s -port %d 2>/dev/null",
```

```
        xfnts_conf, portlist->ports[0].port);
```

出力が dev/null にリダイレクトされていることに注意してください。こうすることによって、デーモンが生成するメッセージが抑制されます。

次に示すように、xfnts_start メソッドは xfs コマンド行を scds_pmf_start() に渡して、PMF の制御下でデータサービスを起動します。

```
scds_syslog(LOG_INFO, "Issuing a start request.");
   err = scds_pmf_start(scds_handle, SCDS_PMF_TYPE_SVC,
      SCDS_PMF_SINGLE_INSTANCE, cmd, -1);

   if (err == SCHA_ERR_NOERR) {
      scds_syslog(LOG_INFO,
          "Start command completed successfully.");
   } else {
      scds_syslog(LOG_ERR,
          "Failed to start HA-XFS ");
   }
```

scds_pmf_start() を呼び出すときは、次のことに注意してください。

- SCDS_PMF_TYPE_SVC パラメータには、データサービスアプリケーションとして起動するプログラムを指定します。このメソッドは他のタイプのアプリケーション (障害モニターなど) も起動できます。
- SCDS_PMF_SINGLE_INSTANCE パラメータには、これが単一インスタンスのリソースであることを指定します。
- cmd パラメータは、すでに生成されているコマンド行を示します。
- 最後のパラメータである -1 には、子プロセスの監視レベルを指定します。-1 は、PMF がすべての子プロセスを親プロセスと同様に監視することを示します。

次に示すように、svc_pmf_start() はportlist 構造体に割り当てられているメモリーを解放してから戻ります。

```
scds_free_port_list(portlist);
return (err);
```

svc_start() からの復帰

svc_start() が正常終了したときでも、使用するアプリケーションが起動に失敗した可能性があります。そのため、svc_start() はアプリケーションを検証して、アプリケーションが動作していることを確認してから、正常終了のメッセージを戻す必要があります。このとき、アプリケーションがただちに利用できない理由として、アプリケーションの起動にはある程度時間がかかるということを考慮しておく必要があります。次に示すように、svc_start() メソッドは xfnts.c で定義されている svc_wait() を呼び出して、アプリケーションが動作していることを確認します。

```
/* サービスが完全に起動するまで待つ。*/
   scds_syslog_debug(DBG_LEVEL_HIGH,
```

```
        "Calling svc_wait to verify that service has started.");

    rc = svc_wait(scds_handle);

    scds_syslog_debug(DBG_LEVEL_HIGH,
        "Returned from svc_wait");

    if (rc == 0) {
       scds_syslog(LOG_INFO, "Successfully started the service.");
    } else {
       scds_syslog(LOG_ERR, "Failed to start the service.");
    }
```

次に示すように、svc_wait() 関数は scds_get_netaddr_list(3HA) を呼び出して、アプリケーションを検証するのに必要なネットワークアドレスリソースを取得します。

```
/* 検証に使用するネットワークリソースを取得する。*/
    if (scds_get_netaddr_list(scds_handle, &netaddr)) {
       scds_syslog(LOG_ERR,
           "No network address resources found in resource group.");
       return (1);
    }

    /* ネットワークリソースが存在しない場合は、エラーを戻す。*/
    if (netaddr == NULL || netaddr->num_netaddrs == 0) {
       scds_syslog(LOG_ERR,
           "No network address resource in resource group.");
       return (1);
    }
```

次に示すように、svc_wait() は start_timeout と stop_timeout 値を取得します。

```
svc_start_timeout = scds_get_rs_start_timeout(scds_handle)
    probe_timeout = scds_get_ext_probe_timeout(scds_handle)
```

サーバーの起動に時間がかかることを考慮して、svc_wait() は scds_svc_wait() を呼び出して、start_timeout 値の 3 % であるタイムアウト値を渡します。次に、svc_wait() は svc_probe() を呼び出して、アプリケーションが起動していることを確認します。svc_probe() メソッドは指定されたポート上でサーバーとの単純ソケット接続を確立します。ポートへの接続が失敗した場合、svc_probe() は値 100 を戻して、致命的な障害であることを示します。ポートとの接続は確立したが、切断に失敗した場合、svc_probe() は値 50 を戻します。

svc_probe() が完全にまたは部分的に失敗した場合、svc_wait() は scds_svc_wait() をタイムアウト値 5 で呼び出します。scds_svc_wait() メソッドは、検証の周期を 5 秒ごとに制限します。また、このメソッドはサービスを起

動しようとした回数も数えます。この回数がリソースの Retry_interval プロパティで指定された期限内にリソースの Retry_count プロパティの値を超えた場合、scds_svc_wait() メソッドは失敗します。この場合、svc_start() 関数も失敗します。

```
#define    SVC_CONNECT_TIMEOUT_PCT    95
#define    SVC_WAIT_PCT       3
   if (scds_svc_wait(scds_handle, (svc_start_timeout * SVC_WAIT_PCT)/100)
      != SCHA_ERR_NOERR) {

      scds_syslog(LOG_ERR, "Service failed to start.");
      return (1);
   }

   do {
      /*
       * ネットワークリソースのIP アドレスとportname 上で
       * データサービスを検証する。
       */
      rc = svc_probe(scds_handle,
          netaddr->netaddrs[0].hostname,
          netaddr->netaddrs[0].port_proto.port, probe_timeout);
      if (rc == SCHA_ERR_NOERR) {
         /* 成功。リソースを解放して終了。*/
         scds_free_netaddr_list(netaddr);
         return (0);
      }

       /* サービスが何度も失敗する場合は、
        * iscds_svc_wait() を呼び出す。
        */
      if (scds_svc_wait(scds_handle, SVC_WAIT_TIME)
         != SCHA_ERR_NOERR) {
         scds_syslog(LOG_ERR, "Service failed to start.");
         return (1);
      }

   /* RGM のタイムアウトを待ってプログラムを終了する。*/
   } while (1);
```

注 – xfnts_start メソッドは終了する前に scds_close() を呼び出して、scds_initialize() が割り当てたリソースを再利用します。詳細については、142 142 ページの「scds_initialize() 関数」と scds_close(3HA) のマニュアルページを参照してください。

---

xfnts_stop メソッド

xfnts_start メソッドは scds_pmf_start() を使用して PMF の制御下でサービスを起動するため、xfnts_stop は scds_pmf_stop() を使用してサービスを中止します。

注 – xfnts_stop は scds_initialize() を最初に呼び出します。これによって、いくつかのハウスキーピング関数が実行されます。詳細については、scds_initialize(3HA) のマニュアルページを参照してください。

次に示すように、xfnts_stop メソッドは、xfnts.c で定義されている svc_stop () メソッドを呼び出します。

```
scds_syslog(LOG_ERR, "Issuing a stop request.");
   err = scds_pmf_stop(scds_handle,
       SCDS_PMF_TYPE_SVC, SCDS_PMF_SINGLE_INSTANCE, SIGTERM,
       scds_get_rs_stop_timeout(scds_handle));

   if (err != SCHA_ERR_NOERR) {
      scds_syslog(LOG_ERR,
          "Failed to stop HA-XFS.");
      return (1);
   }

   scds_syslog(LOG_INFO,
       "Successfully stopped HA-XFS.");
   return (SCHA_ERR_NOERR); /* 正常に停止。*/
```

svc_stop() から scds_pmf_stop() 関数を呼び出すときは、次のことに注意してください。

- SCDS_PMF_TYPE_SVC パラメータには、データサービスアプリケーションとして停止するプログラムを指定します。このメソッドは他のタイプのアプリケーション (障害モニターなど) も停止できます。
- SCDS_PMF_SINGLE_INSTANCE パラメータには、シグナルを指定します。
- SIGTERM パラメータには、リソースインスタンスを停止するのに使用するシグナルを指定します。このシグナルでインスタンスを停止できなかった場合、scds_pmf_stop() は SIGKILL を送信してインスタンスを停止しようとします。このシグナルでもインスタンスを停止できなかった場合、タイムアウトエラーで戻ります。詳細については、scds_pmf_stop(3HA) のマニュアルページを参照してください。
- タイムアウト値は、リソースの Stop_timeout プロパティの値を示します。

注 – xfnts_stop メソッドは終了する前に scds_close() を呼び出して、scds_initialize() が割り当てたリソースを再利用します。詳細については、142 142 ページの「scds_initialize() 関数」と scds_close(3HA) のマニュアルページを参照してください。

xfnts_monitor_start メソッド

リソースがノード上で起動したあと、RGM はそのノード上で Monitor_start メソッドを呼び出して障害モニターを起動します。xfnts_monitor_start メソッドは scds_pmf_start() を使用して PMF の制御下でモニターデーモンを起動します。

注 – xfnts_monitor_start は、scds_initialize() を最初に呼び出します。これによって、いくつかのハウスキーピング関数が実行されます。詳細については、142 142 ページの「scds_initialize() 関数」と、scds_initialize(3HA) のマニュアルページを参照してください。

次に示すように、xfnts_monitor_start メソッドは、xfnts.c に定義されている mon_start メソッドを呼び出します。

```
scds_syslog_debug(DBG_LEVEL_HIGH,
      "Calling Monitor_start method for resource <%s>.",
      scds_get_resource_name(scds_handle));

    /* scds_pmf_start を呼び出し、検証の名前を渡す。*/
   err = scds_pmf_start(scds_handle, SCDS_PMF_TYPE_MON,
       SCDS_PMF_SINGLE_INSTANCE, "xfnts_probe", 0);

   if (err != SCHA_ERR_NOERR) {
      scds_syslog(LOG_ERR,
          "Failed to start fault monitor.");
      return (1);
   }

   scds_syslog(LOG_INFO,
       "Started the fault monitor.");

   return (SCHA_ERR_NOERR); /* モニターを正常に起動。*/
}
```

svc_mon_start() から scds_pmf_start() 関数を呼び出すときは、次のことに注意してください。

- SCDS_PMF_TYPE_MON パラメータには、障害モニターとして起動するプログラムを指定します。このメソッドは他のタイプのアプリケーション (データサービスなど) も起動できます。
- SCDS_PMF_SINGLE_INSTANCE パラメータには、これが単一インスタンスのリソースであることを指定します。
- xfnts_probe パラメータには、起動するモニターデーモンを指定します。このモニターデーモンは、他のコールバックプログラムと同じディレクトリに存在するものと想定されます。
- 最後のパラメータである 0 は、子プロセスの監視レベルを指定します。この場合、モニターデーモンだけを監視することを示します。

注 – xfnts_monitor_start メソッドは終了する前に scds_close() を呼び出して、scds_initialize() が割り当てたリソースを再利用します。詳細については、142 142 ページの「scds_initialize() 関数」と scds_close(3HA) のマニュアルページを参照してください。

---

xfnts_monitor_stop メソッド

xfnts_monitor_start メソッドは scds_pmf_start() を使用して PMF の制御下でモニターデーモンを起動するので、xfnts_monitor_stop は scds_pmf_stop() を使用してモニターデーモンを停止します。

注 – xfnts_monitor_stop は、 scds_initialize() を最初に呼び出します。これによって、いくつかのハウスキーピング関数が実行されます。詳細については、142 142 ページの「scds_initialize() 関数」と scds_initialize(3HA) のマニュアルページを参照してください。

次に示すように、xfnts_monitor_stop() メソッドは xfnts.c で定義されている mon_stop メソッドを呼び出します。

```
scds_syslog_debug(DBG_LEVEL_HIGH,
      "Calling scds_pmf_stop method");

   err = scds_pmf_stop(scds_handle, SCDS_PMF_TYPE_MON,
       SCDS_PMF_SINGLE_INSTANCE, SIGKILL,
       scds_get_rs_monitor_stop_timeout(scds_handle));

   if (err != SCHA_ERR_NOERR) {
      scds_syslog(LOG_ERR,
          "Failed to stop fault monitor.");
```

```
        return (1);
    }

    scds_syslog(LOG_INFO,
        "Stopped the fault monitor.");

    return (SCHA_ERR_NOERR); /* モニターを正常に停止。*/
}
```

svc_mon_stop() から scds_pmf_stop() 関数を呼び出すときは、次のことに注意してください。

- SCDS_PMF_TYPE_MON パラメータには、障害モニターとして停止するプログラムを指定します。このメソッドは他のタイプのアプリケーション (データサービスなど) も停止できます。
- SCDS_PMF_SINGLE_INSTANCE パラメータには、これが単一インスタンスのリソースであることを指定します。
- SIGKILL パラメータには、リソースインスタンスを停止するのに使用するシグナルを指定します。このシグナルでインスタンスを停止できなかった場合、scds_pmf_stop() はタイムアウトエラーで戻ります。詳細については、scds_pmf_stop(3HA) のマニュアルページを参照してください。
- タイムアウト値は、リソースの Monitor_stop_timeout プロパティの値を示します。

注 – xfnts_monitor_stop メソッドは終了する前に scds_close() を呼び出して、scds_initialize() が割り当てたリソースを再利用します。詳細については、142 142 ページの「scds_initialize() 関数」と scds_close(3HA) のマニュアルページを参照してください。

---

xfnts_monitor_check メソッド

障害モニターがリソースが属するリソースグループを別のノードにフェイルオーバーしようとするたびに、RGM は Monitor_check メソッドを呼び出します。xfnts_monitor_check メソッドは svc_validate() メソッドを呼び出して xfs デーモンをサポートするための適切な構成が存在していることを確認します。詳細については、156 ページの「xfnts_validate メソッド」を参照してください。次に、xfnts_monitor_check のコードを示します。

```
    /* RGM から渡された引数を処理し、syslog を初期化する。*/
    if (scds_initialize(&scds_handle, argc, argv) != SCHA_ERR_NOERR)
{
        scds_syslog(LOG_ERR, "Failed to initialize the handle.");
        return (1);
```

```
    }

    rc =  svc_validate(scds_handle);
    scds_syslog_debug(DBG_LEVEL_HIGH,
        "monitor_check method "
        "was called and returned <%d>.", rc);

    /* scds_initialize が割り当てたすべてのメモリーを解放する。*/
    scds_close(&scds_handle);

    /* モニター検査の一環として実行した検証メソッドの結果を戻す。*/
    return (rc);
}
```

SUNW.xfnts 障害モニター

リソースがノード上で起動したあと、RGM は、PROBE メソッドを直接呼び出すのではなく、Monitor_start メソッドを呼び出してモニターを起動します。xfnts_monitor_start メソッドは PMF の制御下で障害モニターを起動します。xfnts_monitor_stop メソッドは障害モニターを停止します。

SUNW.xfnts 障害モニターは、次の処理を実行します。

- 単純な TCP ベースのサービス (xfs など) を検査するために特別に設計されたユーティリティーを使用して、定期的に xfs サーバーデーモンの状態を監視します。
- (Retry_count と Retry_interval プロパティを使用して) ある期間内にアプリケーションが遭遇した問題を追跡し、アプリケーションが完全に失敗した場合に、データサービスを再起動するか、フェイルオーバーするかどうかを決定します。scds_fm_action() と scds_fm_sleep() 関数は、この追跡および決定機構の組み込みサポートを提供します。
- scds_fm_action() を使用して、フェイルオーバーまたは再起動の決定を実装します。
- リソースの状態を更新して、管理ツールや GUI で利用できるようにします。

xfonts_probe のメインループ

xfonts_probe メソッドは ループを実行します。ループを実行する前に、xfonts_probe は次の処理を行います。

- 次に示すように、xfnts リソース用のネットワークアドレスリソースを取得します。

```
/* 当該リソース用に利用できる IP アドレスを取得する。*/
    if (scds_get_netaddr_list(scds_handle, &netaddr)) {
       scds_syslog(LOG_ERR,
```

```
            "No network address resource in resource group.");
        scds_close(&scds_handle);
        return (1);
    }

    /* ネットワークリソースが存在しない場合、エラーを戻す。*/
    if (netaddr == NULL || netaddr->num_netaddrs == 0) {
        scds_syslog(LOG_ERR,
            "No network address resource in resource group.");
        return (1);
    }
```

- scds_fm_sleep() を呼び出し、タイムアウト値として Thorough_probe_interval の値を渡します。検証を実行する間、検証機能は Thorough_probe_interval で指定された期間、休止状態になります。

```
timeout = scds_get_ext_probe_timeout(scds_handle);

    for (;;) {
        /*
         * 連続する検証の間、Throrough_probe_interval で指定された期間、
         * 休眠状態になる。
         */
        (void) scds_fm_sleep(scds_handle,
            scds_get_rs_thorough_probe_interval(scds_handle));
```

xfnts_probe メソッドは次のようなループを実装します。

```
for (ip = 0; ip < netaddr->num_netaddrs; ip++) {
          /*
           * 状態を監視するホスト名とポートを取得する。
           */
          hostname = netaddr->netaddrs[ip].hostname;
          port = netaddr->netaddrs[ip].port_proto.port;
          /*
           * HA-XFS がサポートするポートは 1 つだけなので、
           * ポート値はポートの配列の最初の
           * エントリから取得する。
           */
          ht1 = gethrtime(); /* Latch probe start time */
          scds_syslog(LOG_INFO, "Probing the service on port: %d.", port);

          probe_result =
          svc_probe(scds_handle, hostname, port, timeout);

          /*
           * サービス検証履歴を更新し、
           * 必要に応じて、アクションを行う。
           * 検証終了時間を取得する。
           */
          ht2 = gethrtime();

          /* ミリ秒に変換する。*/
```

```
        dt = (ulong_t)((ht2 - ht1) / 1e6);

        /*
         * 障害の履歴を計算し、必要に応じて
         * アクションを実行する。
         */
        (void) scds_fm_action(scds_handle,
            probe_result, (long)dt);
    }   /* ネットワークリソースごと */
}     /* 検証を永続的に繰り返す。*/
```

svc_probe() 関数は検証ロジックを実装します。svc_probe() からの戻り値は scds_fm_action() に渡されます。そして scds_fm_action() は、アプリケーションを再起動するか、リソースグループをフェイルオーバーするか、あるいは何もしないかを決定します。

svc_probe() 関数

svc_probe() 関数は、scds_fm_tcp_connect() を呼び出すことによって、指定されたポートとの単純ソケット接続を確立します。接続に失敗した場合、svc_probe() は 100 の値を戻して、致命的な障害であることを示します。接続には成功したが、切断に失敗した場合、svc_probe() は 50 の値を戻して、部分的な障害であることを示します。接続と切断の両方に成功した場合、svc_probe() は 0 の値を戻して、成功したことを示します。

次に、svc_probe() のコードを示します。

```
int svc_probe(scds_handle_t scds_handle,
char *hostname, int port, int timeout)
{
   int  rc;
   hrtime_t   t1, t2;
   int    sock;
   char   testcmd[2048];
   int    time_used, time_remaining;
   time_t      connect_timeout;


   /*
    * データサービスを検証するには、port_list プロパティに指定された、
    * XFS データサービスを提供するホスト上にあるポートとのソケット接続
    * を確立する。
    * 指定されたポート上でリスンするように構成されたXFS サービスが接続に
    * 応答した場合、検証が成功したと判断する。probe_timeout プロパティに
    * 設定された期間待機しても応答がない場合、検証が失敗したと判断する。
    */

    /*
     * SVC_CONNECT_TIMEOUT_PCT をタイムアウトの
     * 百分率として使用し、ポートと接続する。
```

```
  */
connect_timeout = (SVC_CONNECT_TIMEOUT_PCT * timeout)/100;
t1 = (hrtime_t)(gethrtime()/1E9);

/*
 * 検証機能は、指定されたホスト名とポートを接続する。
 * 接続は、probe_timeout 値の 95% に達するとタイムアウトになる。
 */
rc = scds_fm_tcp_connect(scds_handle, &sock, hostname, port,
    connect_timeout);
if (rc) {
   scds_syslog(LOG_ERR,
       "Failed to connect to port <%d> of resource <%s>.",
       port, scds_get_resource_name(scds_handle));
   /* 致命的な障害 */
   return (SCDS_PROBE_COMPLETE_FAILURE);
}

t2 = (hrtime_t)(gethrtime()/1E9);

/*
 * 接続にかかる実際の時間を計算する。この値は、
 * 接続に割り当てられた時間を示す connect_timeout 以下
 * である必要がある。接続に割り当てられた時間をすべて
 * 使い切った場合、probe_timeout に残った値が当該関数に
 * 渡され、切断タイムアウトとして使用される。
 * それ以外の場合、接続呼び出しで残った時間が
 * 切断タイムアウトに追加される。
 */

time_used = (int)(t2 - t1);

/*
 * 残った時間(タイムアウトから接続にかかった時間を引いた値) を
 * 切断に使用する。
 */

time_remaining = timeout - (int)time_used;

/*
 * すべての時間を使い切った場合、ハードコーディングされた小さな
 * タイムアウト値を使用して、切断しようとする。
 * これによって、fd リークを防ぐ。
 */
if (time_remaining <= 0) {
   scds_syslog_debug(DBG_LEVEL_LOW,
       "svc_probe used entire timeout of "
       "%d seconds during connect operation and exceeded the "
       "timeout by %d seconds. Attempting disconnect with timeout"
       " %d ",
       connect_timeout,
       abs(time_used),
       SVC_DISCONNECT_TIMEOUT_SECONDS);

   time_remaining = SVC_DISCONNECT_TIMEOUT_SECONDS;
```

```
}

/*
 * 切断に失敗した場合、部分的な障害を戻す。
 * 理由：接続呼び出しは成功した。これは、
 * アプリケーションが正常に動作していることを意味する。
 * 切断が失敗した原因は、アプリケーションがハングしたか、
 * 負荷が高いためである。
 * 後者の場合、アプリケーションが停止したとは宣言しない
 * (つまり、致命的な障害を戻さない)。その代わりに、部分的な
 * 障害であると宣言する。この状態が続く場合、切断呼び出しは
 * 再び失敗し、アプリケーションは再起動される。
 */
rc = scds_fm_tcp_disconnect(scds_handle, sock, time_remaining);
if (rc != SCHA_ERR_NOERR) {
   scds_syslog(LOG_ERR,
       "Failed to disconnect to port %d of resource %s.",
       port, scds_get_resource_name(scds_handle));
   /* 部分的な障害 */
   return (SCDS_PROBE_COMPLETE_FAILURE/2);
}

t2 = (hrtime_t)(gethrtime()/1E9);
time_used = (int)(t2 - t1);
time_remaining = timeout - time_used;

/*
 * 時間が残っていない場合、fsinfo による完全な
 * テストを行わない。その代わりに、
 * SCDS_PROBE_COMPLETE_FAILURE/2 を戻す。これによって、
 * このタイムアウトが続く場合、サーバーは再起動される。
 */
if (time_remaining <= 0) {
   scds_syslog(LOG_ERR, "Probe timed out.");
   return (SCDS_PROBE_COMPLETE_FAILURE/2);
}

/*
 * ポートとの接続と切断に成功した。
 * fsinfo コマンドを実行して、
 * サーバーの状態を完全に検査する。
 * stdout をリダイレクトする。そうでない場合、
 * fsinfo からの出力はコンソールに送られる。
 */
(void) sprintf(testcmd,
    "/usr/openwin/bin/fsinfo -server %s:%d> /dev/null",
    hostname, port);
scds_syslog_debug(DBG_LEVEL_HIGH,
    "Checking the server status with %s.", testcmd);
if (scds_timerun(scds_handle, testcmd, time_remaining,
   SIGKILL, &rc) != SCHA_ERR_NOERR || rc != 0) {

   scds_syslog(LOG_ERR,
      "Failed to check server status with command <%s>",
      testcmd);
```

```
        return (SCDS_PROBE_COMPLETE_FAILURE/2);
    }
    return (0);
}
```

svc_probe() は終了時に、成功 (0)、部分的な障害 (50)、または致命的な障害 (100) を戻します。xfnts_probe メソッドはこの値を scds_fm_action() に渡します。

障害モニターのアクションの決定

xfnts_probe メソッドは scds_fm_action() を呼び出して、行うべきアクションを決定します。scds_fm_action() のロジックは次のとおりです。

- Retry_interval プロパティで指定された期間中に、障害の履歴を累積します。
- 累積した障害が 100 に到達した場合 (致命的な障害)、データサービスを再起動します。Retry_interval を超えた場合、障害の履歴をリセットします。
- Retry_interval で指定された期間中に、再起動の回数が Retry_count プロパティを上回った場合、データサービスをフェイルオーバーします。

たとえば、検証機能が xfs サーバーに正常に接続したが、切断に失敗したものと想定します。これは、サーバーは動作しているが、ハングしていたり、一時的に過負荷状態になっている可能性を示しています。切断に失敗すると、scds_fm_action() に部分的な障害 (50) が送信されます。この値は、データサービスを再起動するしきい値を下回っていますが、値は障害の履歴に記録されます。

次回の検証でもサーバーが切断に失敗した場合、scds_fm_action() が保持している障害の履歴に値 50 が再度追加されます。累積した障害の履歴が 100 になるので、scds_fm_action() はデータサービスを再起動します。

xfnts_validate メソッド

リソースが作成されたとき、および、リソースまたは (リソースを含む) リソースグループのプロパティが管理アクションによって更新されるとき、RGM は Validate メソッドを呼び出します。RGM は、作成または更新が行われる前に、Validate メソッドを呼び出します。任意のノード上でメソッドから失敗の終了コードが戻ると、作成または更新は取り消されます。

RGM が Validate メソッドを呼び出すのは、管理アクションがリソースまたはグループのプロパティを変更したときだけです。RGM がプロパティを設定したときや、モニターがリソースプロパティ Status と Status_msg を設定したときではありません。

注 – PROBE メソッドがデータサービスを新しいノードにフェイルオーバーする場合、Monitor_check メソッドも明示的に Validate メソッドを呼び出します。

RGM は、他のメソッドに渡す追加の引数 (更新されるプロパティと値を含む) を指定して、Validate メソッドを呼び出します。xfnts_validate の開始時に実行される scds_initialize() の呼び出しにより、RGM が xfnts_validate に渡したすべての引数が解析され、その情報が scds_handle パラメータに格納されます。この情報は、xfnts_validate が呼び出すサブルーチンによって使用されます。

xfnts_validate メソッドは svc_validate() を呼び出して、次のことを検証します。

- Confdir_list プロパティがリソース用に設定されており、単一のディレクトリが定義されているかどうか。

```
scha_str_array_t *confdirs;
   confdirs = scds_get_ext_confdir_list(scds_handle);

/* Confdir_list 拡張プロパティが存在しない場合、エラーを戻す。*/
   if (confdirs == NULL || confdirs->array_cnt != 1) {
      scds_syslog(LOG_ERR,
          "Property Confdir_list is not set properly.");
      return (1); /* 検証に失敗 */
   }
```

- Confdir_list で指定されたディレクトリに fontserver.cfg ファイルが存在しているかどうか。

```
(void) sprintf(xfnts_conf, "%s/fontserver.cfg", confdirs->str_array[0]);

   if (stat(xfnts_conf, &statbuf) != 0) {
      /*
       * errno.h プロトタイプには void 引数がないので、
       * lint エラーが抑制される。
       */
      scds_syslog(LOG_ERR,
          "Failed to access file <%s> : <%s>",
          xfnts_conf, strerror(errno));   /*lint !e746 */
      return (1);
   }
```

- サーバーデーモンバイナリがクラスタノード上でアクセスできるかどうか。

```
if (stat("/usr/openwin/bin/xfs", &statbuf) != 0) {
      scds_syslog(LOG_ERR,
          "Cannot access XFS binary : <%s> ", strerror(errno));
      return (1);
   }
```

- Port_list プロパティが単一のポートを指定しているかどうか。

```
scds_port_list_t   *portlist;
   err = scds_get_port_list(scds_handle, &portlist);
   if (err != SCHA_ERR_NOERR) {
      scds_syslog(LOG_ERR,
          "Could not access property Port_list: %s.",
         scds_error_string(err));
      return (1); /* 検証に失敗 */
   }

#ifdef TEST
   if (portlist->num_ports != 1) {
      scds_syslog(LOG_ERR,
          "Property Port_list must have only one value.");
      scds_free_port_list(portlist);
      return (1); /* 検証に失敗 */
   }
#endif
```

- データサービスが属するリソースグループに、少なくとも 1 つのネットワークアドレスリソースが属しているかどうか。

```
scds_net_resource_list_t *snrlp;
   if ((err = scds_get_rs_hostnames(scds_handle, &snrlp))
      != SCHA_ERR_NOERR) {
      scds_syslog(LOG_ERR,
          "No network address resource in resource group: %s.",
         scds_error_string(err));
      return (1); /* 検証に失敗 */
   }

   /* ネットワークアドレスリソースが存在しない場合エラーを戻す。*/
   if (snrlp == NULL || snrlp->num_netresources == 0) {
      scds_syslog(LOG_ERR,
          "No network address resource in resource group.");
      rc = 1;
      goto finished;
   }
```

次に示すように、`svc_validate()` は戻る前に、割り当てられているすべてのリソースを解放します。

```
finished:
   scds_free_net_list(snrlp);
   scds_free_port_list(portlist);

   return (rc); /* 検証結果を戻す。*/
```

注 – xfnts_validate メソッドは終了する前に scds_close() を呼び出して、scds_initialize() が割り当てたリソースを再利用します。詳細については、142 142 ページの「scds_initialize() 関数」と scds_close(3HA) のマニュアルページを参照してください。

xfnts_update メソッド

プロパティが変更された場合、RGM は Update メソッドを呼び出して、そのことを動作中のリソースに通知します。xfnts データサービスにおいて変更可能なプロパティは、障害モニターに関連したものだけです。したがって、プロパティが更新されたとき、xfnts_update メソッドは scds_pmf_restart_fm() を呼び出して、障害モニターを再起動します。

```
/*
 * 障害モニターがすでに動作していることを検査し、動作している場合、
 * 障害モニターを停止および再起動する。scds_pmf_restart_fm() への
 * 2 番目のパラメータは、再起動する必要がある障害モニターの
 * インスタンスを一意に識別する。
 */

 scds_syslog(LOG_INFO, "Restarting the fault monitor.");
 result = scds_pmf_restart_fm(scds_handle, 0);
 if (result != SCHA_ERR_NOERR) {
    scds_syslog(LOG_ERR,
        "Failed to restart fault monitor.");
   /* scds_initialize が割り当てたすべてのメモリーを解放する。*
    scds_close(&scds_handle);
    return (1);
 }

 scds_syslog(LOG_INFO,
 "Completed successfully.");
```

注 – scds_pmf_restart_fm() への 2 番目のパラメータは、複数のインスタンスが存在する場合に、再起動する障害モニターのインスタンスを一意に識別します。この例の場合、値 0 は障害モニターのインスタンスが 1 つしか存在しないことを示します。

第 9 章

SunPlex Agent Builder

この章では、Resource Group Manager (RGM) の管理下で動作するリソースタイプ (データサービス) の作成を自動化するツール、SunPlex™ Agent Builder と Agent Builder 用の Cluster Agent モジュール について説明します。リソースタイプとは、アプリケーションが RGM の制御下にあるクラスタ環境で動作できるようにするアプリケーションのラッパーのことです。

Agent Builder は、アプリケーションや作成したいリソースタイプの種類に関する簡単な情報を入力するための画面ベースのインタフェースを提供します。Agent Builder は、入力された情報に基づいて、次のソフトウェアを生成します。

- フェイルオーバーリソースタイプまたはスケーラブルリソースタイプのメソッドコールバックに対応する複数のソースファイル。C、Korn シェル (`ksh`)、汎用データサービス (Generic Data Service: GDS) など。
- C シェルまたは Korn シェルのソースを生成する場合は、カスタマイズされたリソースタイプ登録 (Resource Type Registration: RTR) ファイル。
- リソースタイプのインスタンス (リソース) を起動、停止、および削除するためのカスタマイズされたユーティリティースクリプト。また、これらのファイルの使用方法を説明するカスタマイズされたマニュアルページ。
- C のソースを生成する場合はバイナリを含む Solaris パッケージとユーティリティースクリプト。C または Korn シェルのソースを生成する場合は RTR ファイルを含む Solaris パッケージとユーティリティースクリプト。

Agent Builder は、クライアントとの通信にネットワークを使用するネットワーク対応アプリケーションと、非ネットワーク対応のスタンドアロンアプリケーションをサポートします。Agent Builder を使って、プロセス監視機能 (PMF) によって個別に監視および再起動される複数の独立したプロセスツリーを持つアプリケーション用のリソースタイプを生成できます。170 ページの「複数の独立したプロセスツリーを持つリソースタイプの作成」を参照してください。

この章では、次のトピックについて説明します。

Agent Builder の使用

この節では、Agent Builder の使用方法と、Agent Builder を使用する前に実行する作業について説明します。リソースタイプコードを生成したあとで Agent Builder を活用する方法についても説明します。

アプリケーションの分析

Agent Builder を使用する前に、アプリケーションが高可用性またはスケーラビリティを備えるための要件を満たしているかどうかを判定します。この分析はアプリケーションの実行時特性だけに基づくものなので、Agent Builder はこの分析を行うことができません。詳細については、29 ページの「アプリケーションの適合性の分析」を参照してください。

Agent Builder は必ずしもアプリケーション用の完全なリソースタイプを作成できるわけではありませんが、ほとんどの場合、少なくとも部分的なソリューションを提供します。たとえば、より複雑なアプリケーションでは、Agent Builder がデフォルトで生成しないコード (プロパティの妥当性検査を追加するコードや Agent Builder がエクスポーズしないパラメータを調節するコード) を追加しなければならない場合もあります。このような場合、生成されたコードまたは RTR ファイルを修正する必要があります。Agent Builder は、このような柔軟性を提供するように設計されています。

Agent Builder は、ソースファイル内において独自のリソースタイプコードを追加できる場所にコメント文を埋め込みます。ソースコードを修正した後、Agent Builder が生成した Makefile を使用すれば、ソースコードを再コンパイルし、リソースタイプパッケージを生成し直すことができます。

Agent Builder が生成したリソースタイプコードを使用せずに、リソースタイプコードを完全に作成し直す場合でも、Agent Builder が生成した Makefile やディレクトリ構造を使用すれば、独自のリソースタイプ用の Solaris パッケージを作成できます。

Agent Builder のインストールと構成

Agent Builder を個別にインストールする必要はありません。Agent Builder は、Sun Cluster ソフトウェアの標準インストールでデフォルトでインストールされる `SUNWscdev` パッケージに含まれています。詳細は、『*Sun Cluster 3.1 10/03* ソフトウェアのインストール』を参照してください。Agent Builder を使用する前に、次のことを確認してください。

- $PATH 変数に Java が指定されていること。Agent Builder は Java (Java Development Kit バージョン 1.3.1 以上) に依存しています。$PATH 変数に Java が指定されていない場合、scdsbuilder はエラーメッセージを戻します。
- Solaris 8 以降の「Developer System Support」ソフトウェアグループがインストールされていること。
- $PATH 変数に cc コンパイラが指定されていること。Agent Builder は、$PATH 変数内の最初の cc を、リソースタイプの C バイナリコードを生成するコンパイラとします。cc が $PATH に存在しない場合、Agent Builderは C コードを生成するオプションを無効にします。詳細は、165 ページの「作成画面の使用」を参照してください。

注 – Agent Builder では、標準の cc コンパイラ以外のコンパイラも使用できます。このためには、$PATH において、cc から別のコンパイラ (gcc など) にシンボリックリンクを作成します。もう 1 つの方法は、Makefile におけるコンパイラ指定を変更して (現在は、CC=cc)、別のコンパイラへの完全パスを指定します。たとえば、エージェント が生成する Makefile において、CC=cc を CC=*pathname*/gcc に変更します。この場合、エージェントを直接実行することはできません。代わりに、make や make pkg コマンドを使用して、データサービスコードとパッケージを生成する必要があります。

Agent Builder の起動

Agent Builder は次のコマンドで起動します。

```
% /usr/cluster/bin/scdsbuilder
```

次の図は、エージェント の初期画面です。

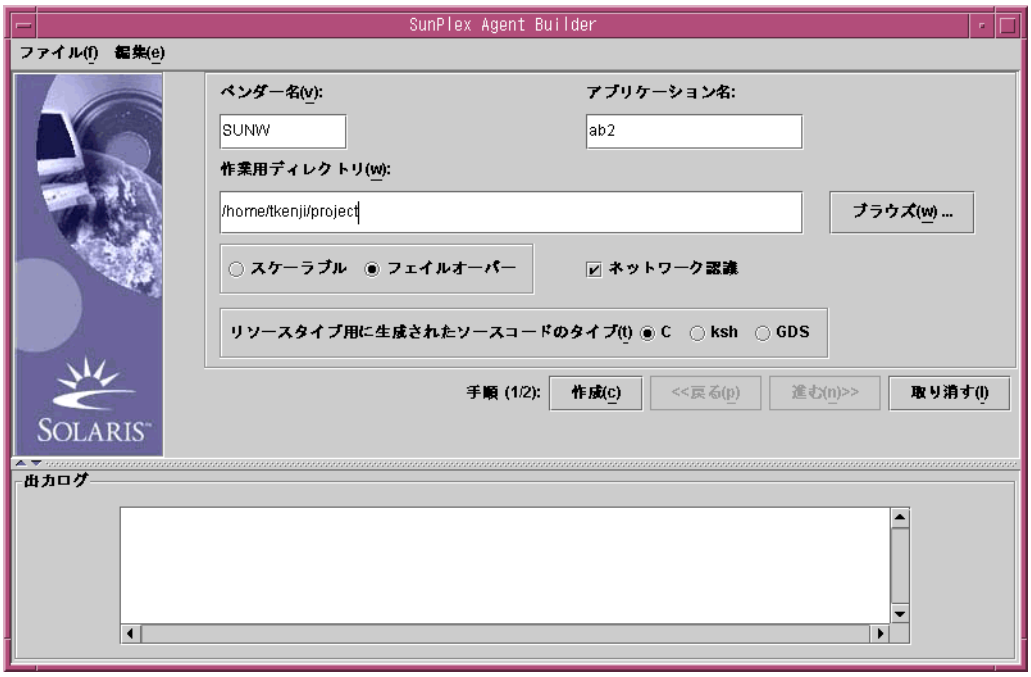


図 **9–1** 初期画面

注 – GUI バージョンにアクセスできない場合は、コマンド行インタフェースバージョンの エージェント を使用できます。詳細については、173 ページの「コマンド行バージョンの Agent Builder の使用」を参照してください。

Agent Builder では、次の 2 つの画面を使用して、新しいリソースタイプを作成します。

1. 作成 — この画面では、リソースタイプ名、リソースタイプテンプレートを作成および構成する作業ディレクトリなどの基本情報を指定して、生成されたファイル用のリソースタイプを作成します。作成するリソースの種類 (スケーラブルまたはフェイルオーバー) 、ベースアプリケーションがネットワーク対応かどうか (つまり、ネットワークを使用してクライアントと通信するかどうか)、生成するコードのタイプ (C、`ksh` または GDS) も指定できます。GDS の詳細については、第 10 章を参照してください。この画面に必要な情報をすべて入力し、「作成」を選択すると、対応する出力が生成されます。その後、構成画面に進みます。
2. 構成 — この画面では、アプリケーションを起動するコマンドを提供する必要があります。オプションとして、アプリケーションを停止するコマンドや検証するコマンドも提供できます。これらのコマンドを指定しない場合、生成される出力コードは、シグナルを使用してアプリケーションを停止し、デフォルトの検証メカニズムを使用してアプリケーションを検証します。検証コマンドについては、167 ページ

の「構成画面の使用」を参照してください。また、この画面では、上記の各コマンドのタイムアウト値も変更できます。

注 – 既存のリソースタイプの作業ディレクトリから起動する場合、Agent Builder は、作成および構成画面を既存のリソースタイプの値に初期化します。

Agent Builder の画面上のボタンやメニューの使用方法については、178 ページの「Agent Builder のナビゲーション」を参照してください。

作成画面の使用

リソースタイプを作成する最初の段階では、Agent Builder を起動したときに表示される作成画面に必要な情報を入力します。すると、次の画面が表示されます。

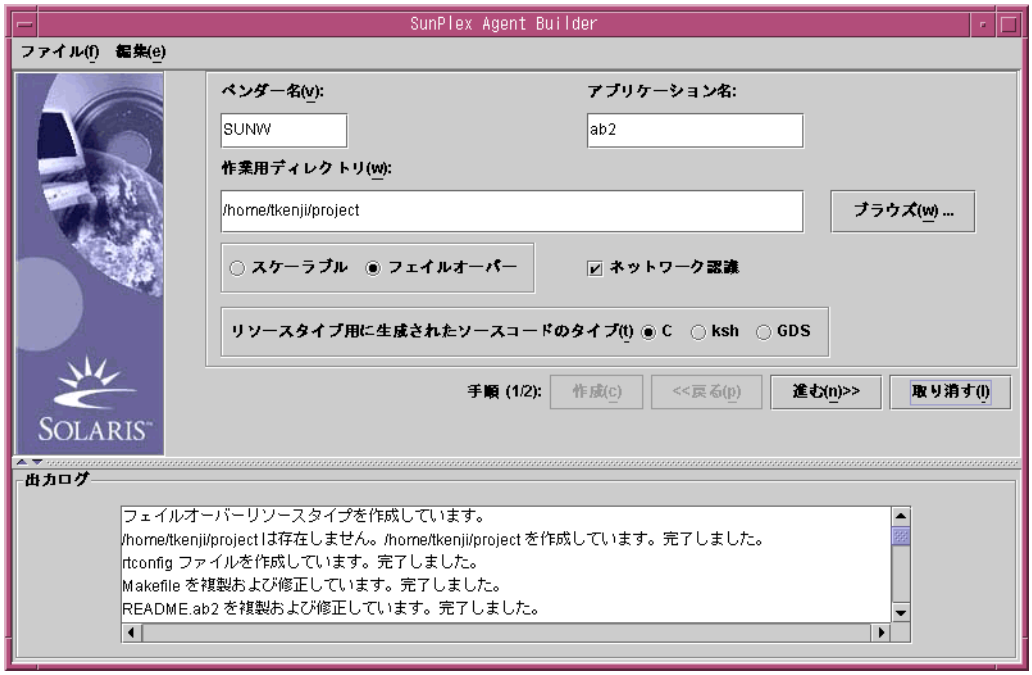


図 **9–2** 作成画面

作成画面には、次のフィールド、ラジオボタン、およびチェックボックスがあります。

- ベンダー名 — リソースタイプのベンダーを識別する名前。通常、ベンダーの略号を指定します。ベンダーを一意に識別する名前であれば、どのような名前でも有効です。英数字文字だけを使用します。
- アプリケーション名 — リソースタイプ名。英数字文字だけを使用します。

注 – ベンダー名とアプリケーション名の両方で、リソースタイプの完全な名前が形成されます。完全な名前は 9 文字を超えてはなりません。

- 作業用ディレクトリ — Agent Builder は、このディレクトリの下に、ターゲットリソースタイプ用のすべてのファイルを格納するディレクトリ構造を作成します。1 つの作業ディレクトリには 1 つのリソースタイプしか作成できません。Agent Builder は、このフィールドを Agent Builder が起動されたディレクトリのパスで初期化します。ただし、別のディレクトリ名を入力したり、「ブラウズ」ボタンを使用して異なるディレクトリを指定することもできます。

  Agent Builder は、作業ディレクトリの下にリソースタイプ名を持つサブディレクトリを作成します。たとえば、ベンダー名が `SUNW` で、アプリケーション名が `ftp` である場合、Agent Builder はこのサブディレクトリに `SUNWftp` という名前を付けます。

  Agent Builder は、ターゲットリソースタイプのすべてのディレクトリとファイルをこのサブディレクトリの下に置きます(173 ページの「ディレクトリ構造」を参照)。
- スケーラブルまたはフェイルオーバー —ターゲットリソースタイプがスケーラブルまたはフェイルオーバーのどちらであるかを指定します。
- ネットワーク認識 —ベースアプリケーションがネットワーク対応かどうかを指定します。つまり、アプリケーションがネットワークを使用してクライアントと通信するかどうかを指定します。ネットワーク対応であれば、チェックボックスにチェックマークを入れます。非ネットワーク対応であれば、チェックボックスをそのままにします。Korn シェルコードの場合、アプリケーションはネットワーク対応でなければなりません。したがって、「**ksh**」または「**GDS**」チェックボックスにチェックマークを入れた場合、このチェックボックスは自動的に無効になります。
- **C, ksh**—生成されるソースコードの言語を指定します。このオプションは、どちらか 一方しか指定できません。ただし、Agent Builder を使用すれば、`ksh` 用に生成されたコードでリソースタイプを作成した後、同じ情報を再利用して C 言語のコードを作成できます。詳細については、171 ページの「既存のリソースタイプのクローンの作成」 を参照してください。
- **GDS** — このサービスが汎用データサービスであることを示します。汎用データサービスの詳しい作成および構成方法については、第 10 章を参照してください。

注 – cc コンパイラが $PATH に存在しない場合、「C」オプションボタンは無効になり、「ksh」ボタンにチェックマークが入ります。異なるコンパイラを指定する方法については、162 ページの「Agent Builder のインストールと構成」の最後にある注を参照してください。

必要な情報を入力した後、「作成」ボタンをクリックします。画面の一番下にある「出力ログ」には、162 ページの「Agent Builder のインストールと構成」 が行なったアクションが表示されます。「編集」メニューの「出力ログを保存」コマンドを使用すれば、出力ログ内の情報を保存できます。

終了したなら、162 ページの「Agent Builder のインストールと構成」 は成功メッセージまたは警告メッセージを表示します。警告メッセージは作成段階が完了しなかったことを示します。その場合は、出力ログの情報から原因を調べます。

Agent Builder が成功メッセージを表示した場合は、「進む」ボタンをクリックして「構成」画面に進むことができます。「構成」画面では、リソースタイプの生成を完結することができます。

注 – 完全なリソースタイプを生成するには、2 段階の作業が必要ですが、最初の段階 (つまり、作成) が完了した後に Agent Builder を終了しても、入力した情報や Agent Builder で作成した内容が失われることはありません。詳細については、171 ページの「完成した作業内容の再利用」を参照してください。

構成画面の使用

リソースタイプを作成する最初の段階では、Agent Builder を起動したときに表示される作成画面に必要な情報を入力します。すると、次の画面が表示されます。リソースタイプの作成が完了していなければ、構成画面にはアクセスできません。

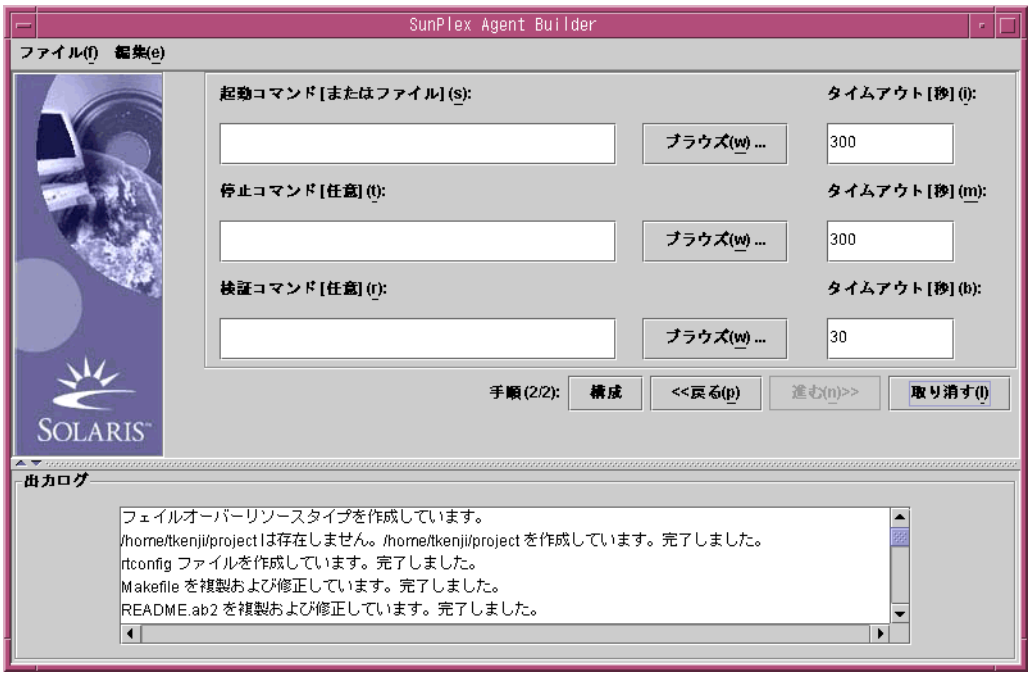


図 **9–3** 構成画面

構成画面には、次のフィールドがあります。

- 起動コマンド — ベースアプリケーションを起動するために任意の UNIX シェルに渡すことができる完全なコマンド行。このコマンドは必ず指定する必要があります。このフィールドにコマンドを入力するか、「ブラウズ」ボタンを使用して、アプリケーションを起動するコマンドが記述されているファイルを指定します。

  完全なコマンド行には、アプリケーションを起動するのに必要なすべての要素が含まれていなければなりません。たとえば、ホスト名、ポート番号、構成ファイルへのパスなどです。コマンド行にホスト名を指定する必要があるアプリケーションの場合、Agent Builder が定義する `$hostnames` 変数を使用できます。詳細については、170 ページの「Agent Builder の `$hostnames` 変数の使用」を参照してください。

  コマンドは二重引用符 ("") で囲んではなりません。

注 – ベースアプリケーションが複数の独立したプロセスツリーを持ち、各プロセスツリーが PMF の制御下で独自のタグによって起動される場合、単一のコマンドは指定できません。代わりに、各プロセスツリーを起動するための個々のコマンドを記述したテキストファイルを作成し、そのファイルへのパスを「起動コマンド」テキストフィールドに指定する必要があります。このファイルが適切に機能するために必要な特性については、170 ページの「複数の独立したプロセスツリーを持つリソースタイプの作成」を参照してください。

- 停止コマンド —ベースアプリケーションを停止するために任意の UNIX シェルに渡すことができる完全なコマンド行。このフィールドにコマンドを入力するか、「ブラウズ」ボタンを使用して、アプリケーションを停止するコマンドが記述されているファイルを指定します。コマンド行にホスト名を指定する必要があるアプリケーションの場合、Agent Builder が定義する `$hostnames` 変数を使用できます。詳細については、170 ページの「Agent Builder の `$hostnames` 変数の使用」を参照してください。

  このコマンドは省略可能です。停止コマンドを指定しない場合、生成されるコードは 、次に示すように、`Stop` メソッドでシグナルを使用して、アプリケーションを停止します。

  - `Stop` メソッドは `SIGTERM` を送信してアプリケーションを停止しようとします。そして、アプリケーション用のタイムアウト値の 80% だけ待機して、停止しない場合は終了します。
  - SIGTERM シグナルが失敗した場合、`Stop` メソッドは `SIGKILL` を送信して、アプリケーションを停止しようとします。そして、アプリケーション用のタイムアウト値の 15% だけ待機して、停止しない場合は終了します 。
  - SIGKILL シグナルが失敗した場合、`Stop` メソッドは 失敗します。タイムアウト値の残りの 5% はオーバーヘッドとなります。

注意 – 停止コマンドは、アプリケーションが完全に停止するまで戻らないことに注意してください。

- 検証コマンド —定期的に実行され、アプリケーションの状態を検査して、0 (正常) から 100 (致命的な障害) の範囲の終了状態を戻すコマンド。このコマンドは省略可能です。このフィールドにコマンドの完全パスを入力するか、「ブラウズ」ボタンを使用して、アプリケーションを検証するコマンドが記述されているファイルを指定します。

  通常は、単にベースアプリケーションのクライアントを指定します。検証コマンドを指定しない場合、生成されるコードは、リソースが使用するポートへの接続と切断を試みます。接続と切断に成功すれば、アプリケーションの状態が正常であると判断します。検証コマンドを使用できるのはネットワーク対応アプリケーションだけです。Agent Builder は常に検証コマンドを生成しますが、非ネットワーク対応アプリケーションでは検証コマンドを無効にします。

検証コマンド行にホスト名を指定する必要があるアプリケーションでは、Agent Builder が定義する `$hostnames` 変数を使用できます。詳細については、170 ページの「Agent Builder の `$hostnames` 変数の使用」を参照してください。

- タイムアウト **(**コマンドごと**) —**各コマンドのタイムアウト値 (秒単位)。新しい値を指定するか、Agent Builder が提供するデフォルト値を受け入れます。起動コマンドと停止コマンドのデフォルト値は 300 秒です。検証コマンドのデフォルト値は 30 秒です。

Agent Builder の `$hostnames` 変数の使用

多くのアプリケーション (特に、ネットワーク対応アプリケーション) では、アプリケーションが通信し、顧客の要求に対してサービスを提供するホスト名をコマンド行に指定して、アプリケーションに渡す必要があります。そのため、多くの場合、ホスト名は、構成画面において、ターゲットリソースタイプの起動、停止、および検証コマンドに指定する必要があるパラメータです。ただし、アプリケーションが通信するホスト名はクラスタに固有です。つまり、ホスト名は、リソースがクラスタ上で動作するときに決定され、Agent Builder でリソースタイプコードを生成するときに決定することはできません。

この問題を解決するために、Agent Builder は `$hostnames` 変数を提供します。この変数を使用すると、起動、停止、および検証コマンドのコマンド行にホスト名を指定できます。`$hostnames` 変数を指定する方法は、実際のホスト名を指定する方法と同じです。たとえば、次のようになります。

```
/opt/network_aware/echo_server -p port_no -l $hostnames
```

ターゲットリソースタイプのリソースがあるクラスタ上で動作するとき、(リソースの `Network_resources_used` リソースプロパティで) そのリソースに構成されている LogicalHostname または SharedAddress ホスト名が `$hostnames` 変数の値に置き換えられます。

`Network_resources_used`プロパティに複数のホスト名を構成している場合、すべてのホスト名をコンマで区切って `$hostnames` 変数に指定します。

複数の独立したプロセスツリーを持つリソースタイプの作成

Agent Builder は、複数の独立したプロセスツリーを持つアプリケーション用のリソースタイプを作成できます。プロセスツリーが独立しているということは、PMF が各プロセスツリーを個別に監視および起動することを意味しています。PMF は独自のタグを使用して各プロセスツリーを起動します。

注 – 複数の独立したプロセスツリーを持つリソースタイプを Agent Builder で作成できるのは、生成されるソースコードとして C を指定する場合だけです。Agent Builder を使用して `ksh` または GDS 用にこれらのリソースタイプを作成することはできません。`ksh` または GDS 用にこれらのリソースタイプを作成するには、手動でそれらのコードを作成する必要があります。

複数の独立したプロセスツリーを持つベースアプリケーションの場合、1 つのコマンド行だけでアプリケーションを起動することはできません。代わりに、アプリケーションの各プロセスツリーを起動するコマンドへの完全パスを行ごとに記述したテキストファイルを作成します。このファイルには空白行を含めることはできません。そして、このファイルへのパスを構成画面の「起動コマンド」テキストフィールドに指定します。

また、このテキストファイルに実行権が設定されていないことを確認する必要があります。これにより、Agent Builder は、このファイルが複数のプロセスツリーを起動するためのものであり、単に複数のコマンドが記述されている実行可能スクリプトではないことを認識できます。このテキストファイルに実行権を与えた場合、リソースはクラスタ上で動作するように見えますが、すべてのコマンドが 1 つの PMF タグ下で起動されるため、PMF は各プロセスツリーを個別に監視および再起動することはできません。

完成した作業内容の再利用

Agent Builder を使用すると、次に示すように、完成した作業内容を再利用できます。

- Agent Builder で作成した既存のリソースタイプのクローンを作成できます。
- Agent Builder が生成したソースコードを編集して、そのコードを再コンパイルすれば、新しいパッケージを作成できます。

既存のリソースタイプのクローンの作成

Agent Builder で作成した既存のリソースタイプのクローンを作成するには、次の手順に従います。

1. 既存のリソースタイプを **Agent Builder** にロードします。これは、次の **2** つの方法で行えます。

   a. **(Agent Builder** で作成した**)** 既存のリソースタイプ用の作業ディレクトリ **(`rtconfig` ファイルが格納されているディレクトリ)** から **Agent Builder** を起動します。すると、**Agent Builder** の作成画面と構成画面に、そのリソースタイプ用の値がロードされます。

   b. 「ファイル」メニューの「リソースタイプをロード」コマンドを使用します。

2. 作成画面で作業ディレクトリを変更します。

   ディレクトリを選択するときは、「ブラウズ」ボタンを使用する必要があります。つまり、新しいディレクトリ名を入力するだけでは十分ではありません。ディレクトリを選択したあと、Agent Builder は「作成」ボタンを有効に戻します。

3. 必要な変更を行います。

   この手順は、リソースタイプ用に生成されたコードのタイプを変更するときに使用できます。たとえば、最初は `ksh` バージョンのリソースタイプを作成していたが、あとで C バージョンのリソースタイプが必要になった場合などです。この場合、既存の `ksh` バージョンのリソースタイプをロードし、出力用の言語を C に変更すると、Agent Builder は C バージョンのリソースタイプを構築します。

4. リソースタイプのクローンを作成します。

   「作成」を選択して、リソースタイプを作成します。「進む」を選択して、構成画面に進みます。「構成」を選択して、リソースタイプを構成します。最後に、「取り消す」を押して、終了します。

生成されたソースコードの編集

リソースタイプを作成するプロセスを簡単にするために、Agent Builder は入力数を制限しています。必然的に、生成されるリソースタイプの範囲も制限されます。したがって、より複雑な機能、たとえば、追加のプロパティの妥当性を検査したり、Agent Builder がエクスポーズしないパラメータを調整したりする機能を追加するには、生成されたソースコードまたは RTR ファイルを修正する必要があります。

ソースファイルは *install_directory*/*rt_name*/`src` ディレクトリに置かれます。Agent Builder は、ソースコード内においてコードを追加できる場所にコメント文を埋め込みます。このようなコメントの形式は次のとおりです (C コードの場合)。

```
/* User added code -- BEGIN vvvvvvvvvvvvvvv */
/* User added code -- END   ^^^^^^^^^^^^^^^ */
```

注 – これらのコメント文は Korn シェルコードの場合も同じです。ただし、コメント行の先頭にはシャープ記号 (#) が使用されます。

たとえば、*rt_name*`.h` は、異なるプログラムが使用するすべてのユーティリティールーチンを宣言します。宣言リストの最後はコメント文になっており、ここでは自分のコードに追加したいルーチンを宣言できます。

install_directory/*rt_name*/`src` ディレクトリには、適切なターゲットとともに、Makefile も生成されます。`make` コマンドを使用すると、ソースコードを再コンパイルできます。また、`make pkg` コマンドを使用すると、リソースタイプパッケージを生成し直すことができます。

RTR ファイルは *install_directory*/*rt_name*/etc ディレクトリに置かれます。RTR ファイルは標準のテキストエディタで編集できます。RTR ファイルの詳細については、34 ページの「リソースとリソースタイププロパティの設定」を参照してください。プロパティの詳細については、付録 A を参照してください。

コマンド行バージョンの Agent Builder の使用

コマンド行バージョンの Agent Builder では、グラフィカルユーザーインタフェースと同様に、2 段階の入力手順が必要です。ただし、グラフィカルユーザーインタフェースに情報を入力する代わりに、scdscreate(1HA) および scdsconfig (1HA) コマンドにパラメータを渡します。

コマンド行バージョンの Agent Builder の使用方法は次のとおりです。

1. アプリケーションに高可用性またはスケーラビリティを持たせるため、**scdscreate** を使って **Sun Cluster** リソースタイプテンプレートを作成します。
2. **scdsconfigure** を使って、**scdscreate** で作成したリソースタイプテンプレートを構成します。
3. 作業ディレクトリの **pkg** サブディレクトリに移動します。
4. **pkgadd(1M)** コマンドを使って、**scdscreate** で作成したパッケージをインストールします。
5. 必要に応じて、生成されたソースコードを編集します。
6. 起動スクリプトを実行します。

ディレクトリ構造

Agent Builder は、ターゲットリソースタイプ用に生成するすべてのファイルを格納するためのディレクトリ構造を作成します。作業ディレクトリは 作成画面で指定します。開発するリソースタイプごとに異なるインストールディレクトリを指定する必要があります。Agent Builder は、作業ディレクトリの下に、作成画面で入力されたベンダー名とリソースタイプ名を連結した名前を持つサブディレクトリを作成します。たとえば、SUNW というベンダー名を指定し、ftp というリソースタイプを作成した場合、Agent Builder は SUNWftp というディレクトリを作業ディレクトリの下に作成します。

Agent Builder は、このサブディレクトリの下に、次のようなディレクトリを作成し、各ディレクトリにファイルを配置します。

ディレクトリ名	内容
bin	C 出力の場合、ソースファイルからコンパイルしたバイナリファイルが格納されます。ksh の場合、src ディレクトリと同じファイルが格納されます。
etc	RTR ファイルが格納されます。Agent Builder はベンダー名とアプリケーション名をピリオド (.) で区切って連結し、RTR ファイル名を形成します。たとえば、ベンダー名が SUNW で、リソースタイプ名が ftp である場合、RTR ファイル名は SUNW.ftp となります。
man	start、stop、および remove ユーティリティースクリプト用にカスタマイズされたマニュアルページ (man1m) が格納されます。たとえば、startftp(1M)、stopftp(1M)、および removeftp(1M) が格納されます。 これらのマニュアルページを表示するには、 man -M オプションでパスを指定します。次に例を示します。 man -M *install_directory*/SUNWftp/man removeftp .
pkg	最終的なパッケージが格納されます。
src	Agent Builder によって生成されたソースファイルが格納されます。
util	Agent Builder によって生成された start、stop、および remove ユーティリティースクリプトが格納されます。176 ページの「ユーティリティースクリプトとマニュアルページ」を参照してください。Agent Builder は、これらのスクリプト名にアプリケーション名を追加します。たとえば、startftp、stopftp、および removeftp のようになります。

出力

この節では、Agent Builder の出力について説明します。

ソースファイルとバイナリファイル

リソースグループと、最終的にはクラスタ上のリソースを管理する Resource Group Manager (RGM) は、コールバックモデル上で動作します。つまり、特定のイベント (ノードの障害など) が発生したとき、RGM は、当該ノード上で動作しているリソースごとにリソースタイプのメソッドを呼び出します。たとえば、RGM は Stop メソッドを呼び出して、当該ノード上で動作しているリソースを停止します。次に、Stop メソッドを呼び出して、異なるノード上でリソースを起動します。このモデルの詳細については、21 ページの「RGM モデル」、23 ページの「コールバックメソッド」、および rt_callbacks(1HA) のマニュアルページを参照してください。

このモデルをサポートするため、Agent Builder は、 *install_directory*/*rt_name*/bin ディレクトリに、コールバックメソッドとして機能する 8 つの実行可能プログラム (C) またはスクリプト (ksh) を生成します。

注 – 厳密には、障害モニターを実装する *rt_name*_probe プログラムはコールバックプログラムではありません。RGM は *rt_name*_probe を直接呼び出すのではなく、*rt_name*_monitor_start および *rt_name*_monitor_stop を呼び出します。そして、これらのメソッドが *rt_name*_probe を呼び出すことによって、障害モニターが起動および停止されます。

Agent Builder が生成する8 つのメソッドは次のとおりです。

- *rt_name*_monitor_check
- *rt_name*_monitor_start
- *rt_name*_monitor_stop
- *rt_name*_probe
- *rt_name*_svc_start
- *rt_name*_svc_stop
- *rt_name*_update
- *rt_name*_validate

各メソッドに固有な情報については、rt_callbacks(1HA) のマニュアルページを参照してください。

Agent Builder は、*install_directory*/*rt_name*/src ディレクトリ (C 出力の場合) に、次のファイルを生成します。

- ヘッダーファイル (*rt_name*.h)
- すべてのメソッドに共通なコードが記述されているソースファイル (*rt_name*.c
- 共通なコード用のオブジェクトファイル (*rt_name*.o)
- 各メソッド用のソースファイル (*.c)
- 各メソッド用のオブジェクトファイル (*.o)

Agent Builder は *rt_name*.o ファイルを各メソッドの .o ファイルにリンクして、実行可能ファイルを *install_directory*/*rt_name*/bin ディレクトリに作成します。

ksh 出力の場合、*install_directory*/*rt_name*/bin と *install_directory*/*rt_name*/src ディレクトリは同じです。各ディレクトリには、7 つのコールバックメソッドと PROBE メソッドに対応する 8 つの実行可能スクリプトが格納されます。

注 – ksh 出力には、2 つのコンパイル済みユーティリティープログラム (gettime と gethostnames) が含まれます。これらのプログラムは、時間を取得して検証を行うのに必要なコールバックメソッドです。

ソースコードを編集して、make コマンドを実行すると、コードを再コンパイルできます。さらに、再コンパイル後、make pkgコマンドを実行すると、新しいパッケージを生成できます。ソースコードの修正をサポートするために、Agent Builder はソースコード中の適切な場所に、コードを追加するためのコメント文を埋め込みます。172 ページの「生成されたソースコードの編集」を参照してください。

ユーティリティースクリプトとマニュアルページ

リソースタイプを生成し、そのパッケージをクラスタにインストール後、リソースタイプのインスタンス (リソース) をクラスタ上で実行する必要があります。一般に、リソースを実行するには、管理コマンドまたは SunPlex Manager を使用します。Agent Builder は、ターゲットリソースタイプのリソースを起動するためのカスタマイズされたユーティリティスクリプトに加え、リソースを停止および削除するスクリプトも生成します。これら 3 つのスクリプトは *install_directory*/*rt_name*/util ディレクトリに格納されており、次のような処理を行います。

- 起動スクリプト - リソースタイプを登録し、必要なリソースグループとリソースを作成します。また、アプリケーションがネットワーク上のクライアントと通信するためのネットワークアドレスリソース (LocalHostname または SharedAddress) も作成します。
- 停止スクリプト — リソースを停止し、無効にします。
- 削除スクリプト — 起動スクリプトの処理を無効にします。つまり、リソース、リソースグループ、およびターゲットリソースタイプを停止して、システムから削除します。

注 – これらのスクリプトは内部的な規則を使用して、リソースとリソースグループの名前付けを行います。そのため、削除スクリプトを使用できるリソースは、対応する起動スクリプトで起動されたリソースだけです。

Agent Builder は、スクリプト名にアプリケーション名を追加することにより、スクリプトの名前付けを行います。たとえば、アプリケーション名が ftp の場合、各スクリプトは startftp、stopftp、および removeftp になります。

Agent Builder は、各ユーティリティースクリプト用のマニュアルページを *install_directory*/*rt_name*/man/man1m ディレクトリに格納します。これらのマニュアルページにはスクリプトに渡す必要があるパラメータについての説明が記載されているので、各スクリプトを起動する前に、これらのマニュアルページをお読みください。

これらのマニュアルページを表示するには、man コマンドに -M オプションを付けて、上記のマニュアルページが格納されているディレクトリへのパスを指定する必要があります。たとえば、ベンダーが SUNW で、アプリケーション名が ftp である場合、startftp(1M) のマニュアルページを表示するには、次のコマンドを使用します。

```
man -M install_directory/SUNWftp/man startftp
```

クラスタ管理者は、マニュアルページユーティリティースクリプトも利用できます。Agent Builder が生成したパッケージをクラスタ上にインストールすると、ユーティリティースクリプト用のマニュアルページは、 /opt/*rt_name*/man ディレクトリに格納されます。たとえば、startftp(1M) のマニュアルページを表示するには、次のコマンドを使用します。

```
man -M /opt/SUNWftp/man startftp
```

サポートファイル

Agent Builder はサポートファイル (pkginfo、postinstall、postremove、preremove など) を *install_directory*/*rt_name*/etc ディレクトリに格納します。このディレクトリには、Resource Type Registration (RTR) ファイルも格納されます。RTR ファイルは、ターゲットリソースタイプが利用できるリソースとリソースタイププロパティを宣言して、リソースをクラスタに登録するときにプロパティ値を初期化します。詳細については、34 ページの「リソースとリソースタイププロパティの設定」を参照してください。RTR ファイルの名前は、ベンダー名とリソースタイプ名をピリオドで区切って連結したものです。 (たとえば、SUNW.ftp)。

RTR ファイルは、ソースコードを再コンパイルしなくても、標準のテキストエディタで編集および修正できます。ただし、make pkg コマンドを使用して、パッケージを再構築する必要があります。

パッケージディレクトリ

install_directory/*rt_name*/pkg ディレクトリには、Solaris パッケージが格納されます。パッケージの名前は、ベンダー名とアプリケーション名を連結したものです (たとえば、SUNWftp)。*install_directory*/*rt_name*/src ディレクトリ内の Makefile は、新しいパッケージを作成するのに役立ちます。たとえば、ソースファイルを修正し、コードを再コンパイルした場合、あるいは、パッケージユーティリティースクリプトを修正した場合、make pkg コマンドを使用して新しいパッケージを作成します。

パッケージをクラスタから削除する場合、同時に複数のノードから pkgrm コマンドを実行しようとすると、pkgrm コマンドが失敗する可能性があります。この問題を解決するには、次の 2 つの方法があります。

- クラスタの いずれかのノードで remove*rt_name* スクリプトを実行してから、任意のノードで pkgrm を実行します。
- クラスタの 1 つのノードで pkgrm を実行して、必要なすべてのクリーンナップ処理を行なった後、(必要であれば同時に) 残りのノードで pkgrm を実行します。

同時に複数のノードから pkgrm を実行しようとして失敗した場合は、再度いずれかのノードで pkgrm を実行した後、残りのノードで pkgrm を実行します。

rtconfig ファイル

C または ksh ソースコードを生成する場合、Agent Builder は、作業ディレクトリ内に構成ファイル rtconfig を生成します。このファイルには、作成および構成画面でユーザーが入力した情報が含まれています。既存のリソースタイプの作業ディレクトリから Agent Builder を起動した場合、または、「ファイル」メニューの「リソースタイプをロード」 コマンドを使って既存のリソースタイプをロードした場合、Agent Builder は rtconfig ファイルを読み取り、作成および構成画面に、既存のリソースタイプ用としてユーザーが指定した情報を自動入力します。この機能は、既存のリソースタイプのクローンを作成したい場合に便利です。詳細については、171 ページの「既存のリソースタイプのクローンの作成」を参照してください。

Agent Builder のナビゲーション

Agent Builder のナビゲーションは操作が簡単でわかりやすいものです。Agent Builder は 2 段階形式のウィザードであり、各段階に対応した画面 (作成画面と構成画面) を提供します 。各画面では、次のように情報を入力します。

- フィールドに情報を入力。
- ディレクトリ構造をブラウズして、ファイルまたはディレクトリを選択。
- 相互に排他的なラジオボタンセットの 1 つを選択 (たとえば、「スケーラブル」または「フェイルオーバー」)。
- チェックボックスのオン/オフ。たとえば、「ネットワーク認識」ボックスにチェックマークを入れると、ベースアプリケーションがネットワーク対応であることを指定します。チェックマークを入れなければ、アプリケーションが非ネットワーク対応であることを指定します。

各画面の下にあるボタンを使用すると、作業を完了したり、次の画面に進んだり、以前の画面に戻ったり、Agent Builder を終了したりできます。Agent Builder は、必要に応じて、これらのボタンを強調表示またはグレー表示します。

たとえば、作成画面において、必要なフィールドに入力し、希望のオプションにチェックマークを付けてから、画面の下にある「作成」ボタンをクリックします。この時点で、以前の画面は存在しないので、「戻る」ボタンはグレー表示されます。また、この作業が完成するまで次の手順には進めないので、「進む」ボタンもグレー表示されます。

Agent Builder は、画面の下にある出力ログ領域に進捗メッセージを表示します。作業が終了したとき、Agent Builder は成功メッセージまたは警告メッセージを出力ログに表示します。このとき、「進む」ボタンが強調表示されます。あるいは、この画面が最後の画面である場合、「キャンセル」ボタンだけが強調表示されます。

「キャンセル」 ボタンを押すと、いつでも Agent Builder を終了できます。

「ブラウズ」ボタン

Agent Builder のフィールドの中には、情報を直接入力することも、「ブラウズ」ボタンをクリックしてディレクトリ構造をブラウズし、ファイルまたはディレクトリを選択することも可能なフィールドがあります。

「ブラウズ」をクリックすると、次のような画面が表示されます。

フォルダをダブルクリックすると、フォルダが開きます。ファイルを強調表示すると、そのファイル名が「ファイル名」ボックスに表示されます。適切なファイルを選択し、強調表示してから、「選択」をクリックします。

注 – ディレクトリをブラウズしている場合は、ディレクトリを強調表示して、「開く」ボタンを選択します。サブディレクトリがない場合、Agent Builder はブラウズウィンドウを閉じて、強調表示されたディレクトリ名を適切なフィールドに表示します。サブディレクトリがある場合、「閉じる」ボタンをクリックすると、ブラウズウィンドウが閉じて、以前の画面に戻ります。Agent Builder は、強調表示したディレクトリ名を適切なフィールドに表示します。

画面の右上隅にあるアイコンは、次のような処理を行います。

ディレクトリツリーの 1 つ上のレベルに移動します。

ホームフォルダに戻ります。

現在選択しているフォルダの下に新しいフォルダを作成します。

ビューを切り替えます。将来のために予約されています。

メニュー

Agent Builder には、「ファイル」と「編集」メニューがあります。

「ファイル」メニュー

「ファイル」メニューでは、次の2 つのコマンドを使用できます。

- リソースタイプをロード —既存のリソースタイプをロードします。Agent Builder が提供するブラウズ画面を使用して、既存のリソースタイプ用の作業ディレクトリを選択します。Agent Builder を起動したディレクトリにリソースタイプが存在する場合、Agent Builder は自動的にそのリソースタイプをロードします。「リソースタイプをロード」コマンドを使用すると、任意のディレクトリから Agent Builder を起動した後、既存のリソースタイプを選択して、新しいリソースタイプを作成するためのテンプレートとして使用できます。詳細については、171 ページ

の「既存のリソースタイプのクローンの作成」を参照してください。

- 終了 — Agent Builder を終了します。Create 画面または Configure 画面で「Cancel」をクリックしても、Agent Builder を終了できます。

「編集」メニュー

「編集」メニューでは、出力ログを消去または保存するコマンドを使用できます。

- 出力ログをクリア — 出力ログから情報を消去します。「作成」または「構成」を選択するたびに、Agent Builder は状態メッセージを出力ログに追加します。対話形式で繰り返してソースコードを修正し、Agent Builder で出力を生成し直しているときに、出力の生成ごとに状態メッセージを記録したい場合は、出力ログを使用するたびにログファイルの内容を保存および消去できます。
- 出力ログを保存 —ログの出力をファイルに保存します。Agent Builder が提供するブラウズ画面を使用すると、ディレクトリを選択して、ファイル名を指定できます。

Agent Builder の Cluster Agent モジュール

Agent Builder の Cluster Agent モジュールは、NetBeans™ モジュールです。この Cluster Agent モジュールを使用することで、 Sun™ ONE Studio 製品のユーザーは統合された開発環境で Sun Cluster ソフトウェアのリソースタイプ (データサービス) を作成できます。この Cluster Agent モジュールは、ユーザーが作成するリソースタイプの種類を説明する画面ベースのインタフェースを提供します。

注 – Sun ONE Studio 製品の設定、インストール、使用についての詳細は、Sun ONE Studio マニュアルに記載されています。

▼ Cluster Agent モジュールのインストールと設定

Cluster Agent モジュールは、Sun Cluster ソフトウェアのインストール時にインストールされます。Sun Cluster のインストールツールは、Cluster Agent モジュールファイルを `/usr/cluster/lib/scdsbuilder` の `scdsbuilder.jar` に配置します。Sun ONE Studio ソフトウェアで Cluster Agent モジュールを使用するには、このファイルに対してシンボリックリンクを作成する必要があります。

注 – また、Cluster Agent モジュールを実行する予定のシステムに Sun Cluster 製品、Sun ONE Studio 製品、および Java ™ 1.4 がすでにインストールされ、それらが使用可能な状況でなければなりません。

1. ユーザー全員が **Cluster Agent** モジュールを使用できるようにするか、あるいは自分だけが使用できるようにするかを決定します。
   - ユーザー全員に使用を認めるには、スーパーユーザーになるか、あるいは同等の役割を宣言し、汎用的なモジュールディレクトリにシンボリックリンクを作成します。

     ```
     # cd /opt/s1studio/ee/modules
     # ln -s /usr/cluster/lib/scdsbuilder/scdsbuilder.jar
     ```

     注 – Sun ONE Studio ソフトウェアを `/opt/s1studio/ee` 以外のディレクトリにすでにインストールしてある場合は、このディレクトリパスを、使用したパスに読み替えてください。

   - 自分だけが使用できるようにするには、 自分の `modules` サブディレクトリにシンボリックリンクを作成します。

     ```
     % cd ~your-home-dir/ffjuser40ee/modules
     % ln -s /usr/cluster/lib/scdsbuilder/scdsbuilder.jar
     ```

2. **Sun ONE Studio** ソフトウェアを停止し、再起動します。

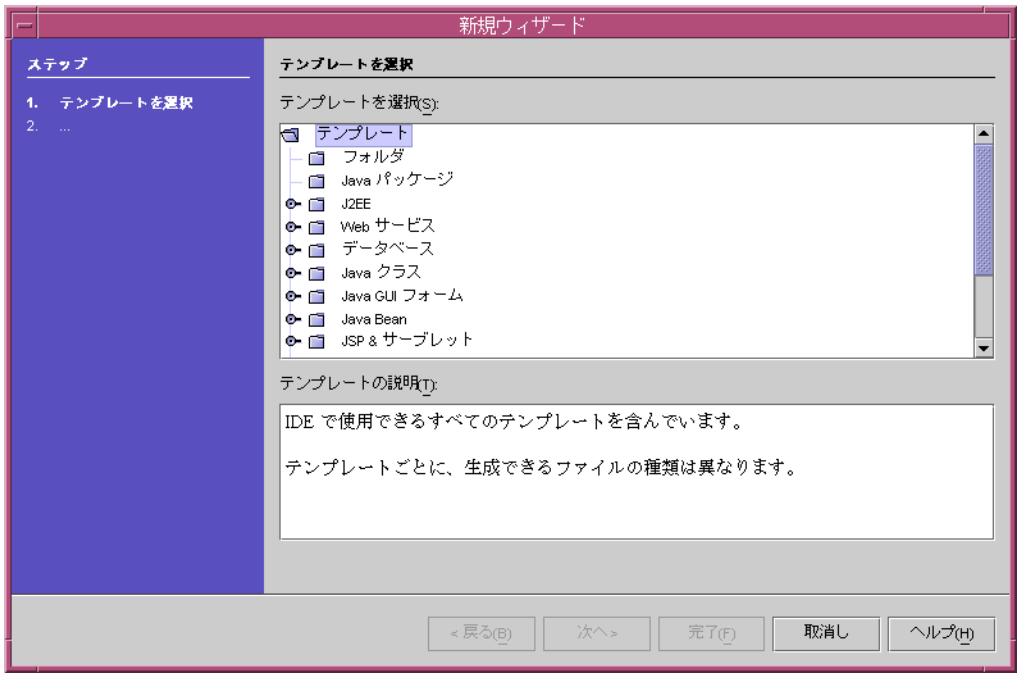
▼ Cluster Agent モジュールの起動

次に、Sun ONE Studio ソフトウェアから Cluster Agent モジュールを起動する手順を示します。

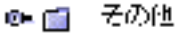
1. **Sun ONE Studio** の「ファイル」メニューから「新規」を選択するか、あるいはツールバーの「新規」アイコンをクリックします。

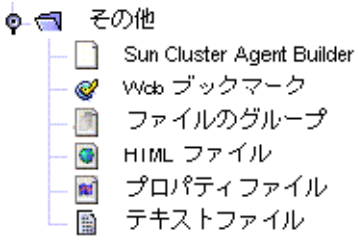
   「新規ウィザード」画面が表示されます。



2. 「テンプレートを選択」ウィンドウで、必要に応じて下方向へスクロールし、その他フォルダの横に表示されている鍵マークをクリックします。

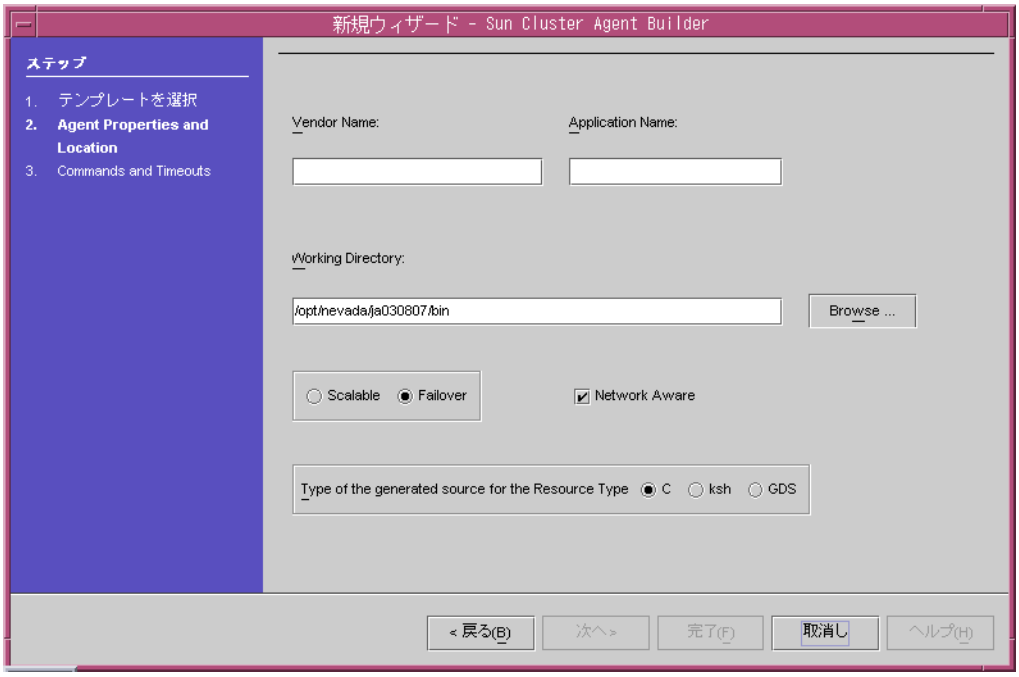


その他フォルダのメニューオプションが示されます。



3. その他メニューから **Sun Cluster Agent Builder** を選択し、「次へ」をクリックします。

Sun One Studio の Cluster Agent モジュールが起動します。最初の「新規ウィザード - Sun Cluster Agent Builder」画面が表示されます。



Cluster Agent モジュールの使用

Cluster Agent モジュールは、Agent Builder ソフトウェアと同様に使用できます。インタフェースは英語版の Agent Builder ソフトウェアと全く同じです。たとえば次の図では、英語版 Agent Builder ソフトウェアの「Create」画面と Cluster Agent モジュールの最初の「新規ウィザード - Sun Cluster Agent Builder」画面には同じフィールドと選択肢が存在することがわかります。

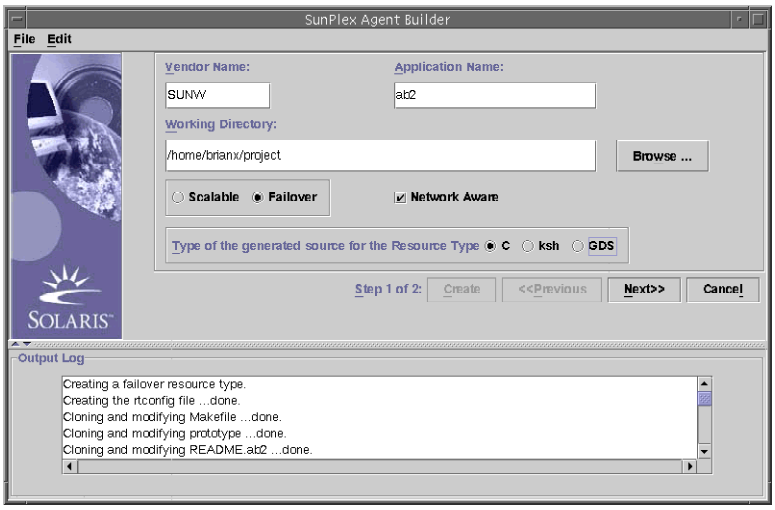


図 **9–4** 英語版 Agent Builder ソフトウェアの作成画面

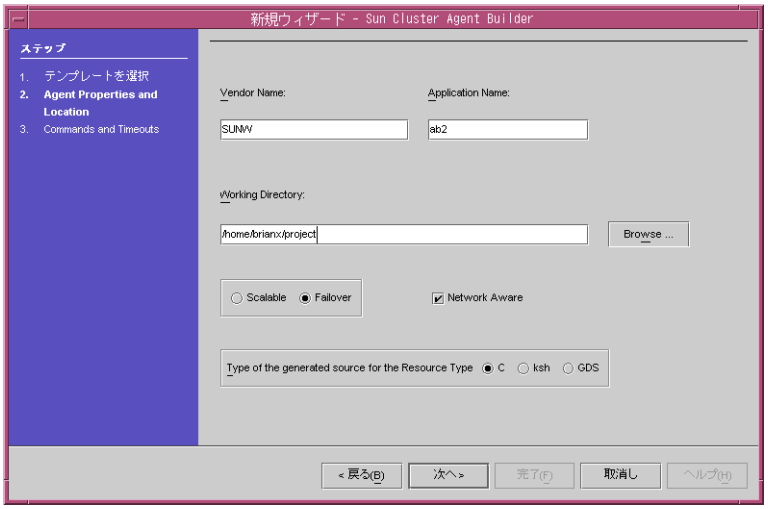


図 **9–5** Cluster Agent モジュールの「新規ウィザード - Sun Cluster Agent Builder」画面

Cluster Agent モジュールと Agent Builder の違い

Cluster Agent モジュールと Agent Builder は似ていますが、小さな違いがいくつかあります。

- Cluster Agent モジュールでは、2 つ目の「新規ウィザード - Sun Cluster Agent Builder」画面で「完了」をクリックした時点でリソースタイプの作成と構成が完了します。最初の「新規ウィザード - Sun Cluster Agent Builder」画面で「次へ」をクリックした時点ではリソースタイプは作成されません。

  英語版 Agent Builder では、「Create」画面で「Create」をクリックした時点でとリソースタイプがただちに作成され、「Configure」画面で「Configure」をクリックする時点で構成が行われます。

- 英語版 Agent Builder の「Output Log」ウィンドウに表示される情報は、Sun ONE Studio 製品では別の出力ウィンドウで表示されます。

第 10 章

汎用データサービス

この章では、最初に汎用データサービス (GDS) の概要について説明した後に、SunPlex Agent Builder または標準的な Sun Cluster 管理コマンドを使って GDS ベースのサービスを作成する方法について説明します。

GDS の概要

汎用データサービス (GDS) とは、ネットワーク対応のシンプルなアプリケーションを Sun Cluster Resource Group Management フレームワークにプラグインすることによって、これらのアプリケーションを可用性の高いものにしたり、スケーラブルなものにするための機構です。この機構では、アプリケーションを可用性の高いものにしたり、スケーラブルなものにするために通常必要になるエージェントのコーディングは必要ありません。

GDS は、あらかじめコンパイルされた単一のデータサービスです。このアプローチでは、コールバックメソッド (`rt_callbacks(1HA)`) の実装やリソースタイプ登録ファイル (`rt_reg(4)`) など、コンパイル済みのデータサービスやそのコンポーネントを変更することはできません。

コンパイル済みリソースタイプ

汎用データサービスのリソースタイプ SUNW.gds は、 SUNWscgds パッケージに含まれています。このパッケージは、クラスタのインストール時に scinstall(1M) ユーティリティでインストールされます。SUNWscgds パッケージには次のファイルが格納されています。

```
# pkgchk -v SUNWscgds

/opt/SUNWscgds
/opt/SUNWscgds/bin
/opt/SUNWscgds/bin/gds_monitor_check
/opt/SUNWscgds/bin/gds_monitor_start
/opt/SUNWscgds/bin/gds_monitor_stop
/opt/SUNWscgds/bin/gds_probe
/opt/SUNWscgds/bin/gds_svc_start
/opt/SUNWscgds/bin/gds_svc_stop
/opt/SUNWscgds/bin/gds_update
/opt/SUNWscgds/bin/gds_validate
/opt/SUNWscgds/etc
/opt/SUNWscgds/etc/SUNW.gds
```

GDS を使用する利点

GDS には、SunPlex Agent Builder が生成するソースコードモデル (scdscreate (1HA) のマニュアルページを参照) や標準的な Sun Cluster 管理コマンドを使用するのに比べ、次の利点があります。

- GDS は使い易いデータサービスです。
- GDS とそのメソッドはコンパイル済みであるため、変更できません。
- SunPlex Agent Builder を使って、アプリケーションを起動するスクリプトを生成できます。これらのスクリプトは、複数のクラスタ間で再利用できる Solaris でインストール可能なパッケージとしてパッケージ化されます。

GDS を使用するサービスの作成方法

GDS を使用するサービスの作成方法は 2 通りあります。

- SunPlex Agent Builder を使用
- 標準的な Sun Cluster 管理コマンドを使用

GDS と SunPlex Agent Builder

SunPlex Agent Builder を使用し、生成されるソースコードのタイプとして GDS を選択します。特定のアプリケーションのリソースを設定する起動スクリプト群を生成するためにユーザーの入力が必要です。

GDS と標準的な Sun Cluster 管理コマンド

この方法では SUNWscgds にあるコンパイル済みデータサービスコードを使用しますが、システム管理者は、標準的な Sun Cluster 管理コマンド (scrgadm(1M) と scswitch(1M)) を使って、リソースの作成と構成を行う必要があります。

GDS ベースのサービスを作成する方法の選択

198 ページの「Sun Cluster 管理コマンドを使って GDS ベースの高可用性サービスを作成する方法」や 198 ページの「標準的な Sun Cluster 管理コマンドを使って GDS ベースのスケーラブルサービスを作成する方法」の手順からわかるように、適切な scrgadm や scswitch コマンドを実行するためには、かなりの分量の入力を行う必要があります。

/GDS と SunPlex Agent Builder を使用する方法では、この処理が簡単になります。この方法では、生成される起動スクリプトがユーザーに代わって scrgadm と scswitch コマンドを出力するからです。

GDS の使用が適さない場合

GDS には多くの利点がありますが、GDS の使用が適さない場合もあります。GDS の使用が適切でないのは次のような場合です。

- コンパイル済みリソースタイプを使用する場合よりも高度な制御が必要な場合。たとえば拡張プロパティを追加する場合や、デフォルト値を変更する場合など
- 特別な機能を追加するためにソースコードを変更する必要がある場合
- 複数のプロセスツリーを使用する場合
- ネットワーク対応でないアプリケーションを使用する場合

GDS の必須プロパティ

次のプロパティを指定する必要があります。

- Start_command (拡張プロパティ)
- Port_list

Start_command 拡張プロパティ

Start_command 拡張プロパティに指定される起動コマンドは、アプリケーションの起動を行います。このコマンドは、引数を備えた完全な UNIX コマンドでなければなりません。コマンドは、アプリケーションを起動するシェルに直接渡されます。

Port_list プロパティ

Port_list プロパティは、アプリケーションが待機するポート群を指定したものです。Port_list プロパティは、SunPlex Agent Builder によって生成される start スクリプトか、scrgadm コマンド (標準的な Sun Cluster 管理コマンドを使用する場合) に指定されていなければなりません。

GDS のオプションプロパティ

GDS のオプションプロパティには次のものがあります。

- Network_resources_used
- Stop_command (拡張プロパティ)
- Probe_command (拡張プロパティ)
- Start_timeout
- Stop_timeout
- Probe_timeout (拡張プロパティ)
- Child_mon_level (標準的な管理コマンドだけで使用される拡張プロパティ)
- Failover_enabled (拡張プロパティ)
- Stop_signal (拡張プロパティ)

Network_resources_used プロパティ

このプロパティのデフォルト値は Null です。アプリケーションが 1 つ以上の特定のアドレスにバインドする必要がある場合は、このプロパティを指定する必要があります。このプロパティを省略するか、このプロパティが Null の場合、アプリケーションはすべてのアドレスに対して待機するものとみなされます。

GDS リソースを作成する場合は、 LogicalHostname か SharedAddress リソースがあらかじめ構成されていなければなりません。LogicalHostname または SharedAddress リソースの構成方法については、『*Sun Cluster 3.1* データサービスのインストールと構成』を参照してください。

値を指定する場合は、1 つまたは複数のリソース名を指定します。個々のリソース名には、1 つ以上の LogicalHostname か 1 つ以上の SharedAddress を指定できます。詳細は、r_properties(5) のマニュアルページを参照してください。

Stop_command プロパティ

stop コマンドは、アプリケーションを停止し、アプリケーションが完全に停止した後で終了します。このコマンドは、アプリケーションを停止するシェルに直接渡される完全な UNIX コマンドでなければなりません。

Stop_command が指定されていると、GDS 停止メソッドは、停止タイムアウトの 80% を指定して停止コマンドを起動します。さらに、GDS 停止メソッドは、停止コマンドの起動結果がどうであれ、停止タイムアウトの 15% を指定して SIGKILL を送信します。タイムアウトの残り 5% は、処理のオーバーヘッドのために使用されます。

stop コマンドが省略されていると、GDS は、 Stop_signal に指定されたシグナルを使ってアプリケーションを停止します。

Probe_command プロパティ

probe コマンドは、特定のアプリケーションの状態を周期的にチェックします。このコマンドは、引数を備えた完全な UNIX コマンドでなければなりません。コマンドは、アプリケーションの状態をチェックするシェルに直接渡されます。アプリケーションの状態が正常であれば、検証コマンドは終了ステータスとして 0 を返します。

検証コマンドの終了ステータスは、アプリケーションの障害の重大度を判断するために使用されます。終了ステータス (probe ステータス) は、0 (正常) から 100 (全面的な障害) までの整数でなければなりません。probe ステータスは 201 という特別な値をもつことがあります。この場合には、Failover_enabled が false に設定されている場合を除き、アプリケーションのフェイルオーバーが直ちに行なわれます。 GDS プローブアルゴリズム (scds_fm_action(3HA) のマニュアルページを参照) は、この probe ステータスを使って、アプリケーションをローカルに再起動するか別のノードにフェイルオーバーするかを決定します。終了ステータス 201 なら、アプリケーションは直ちにフェイルオーバーされます。

probe コマンドが省略されていると、GDS は、Newtork_resources_used プロパティか、 scds_get_netaddr_list(3HA) の出力から得られる IP アドレス群を使ってアプリケーションに接続する独自の簡単な検証を行います。この検証では、接続に成功すると、接続を直ちに切り離します。接続と切り離しが両方とも正常なら、アプリケーションは正常に動作しているものとみなされます。

注 – GDS 提供の検証は、全機能を備えたアプリケーション固有の検証を代替するものではありません。

Start_timeout プロパティ

このプロパティでは、start コマンドの起動タイムアウトを指定します (詳細は、189 189 ページの「Start_command 拡張プロパティ」 を参照)。Start_timeout のデフォルトは 300 秒です。

Stop_timeout プロパティ

このプロパティでは、stop コマンドの停止タイムアウトを指定します (詳細は、190 190 ページの「Stop_command プロパティ」 を参照)。Stop_timeout のデフォルトは 300 秒です。

Probe_timeout プロパティ

このプロパティでは、probe コマンドのプローブタイムアウトを指定します (詳細は、191 191 ページの「Probe_command プロパティ」を参照)。Probe_timeout のデフォルトは 30 秒です。

Child_mon_level プロパティ

このプロパティでは、PMF を通してどのプロセスを監視するかを制御します。このプロパティは、フォークされた子プロセスをどのようなレベルで監視するかを表します。これは、pmfadm(1M) コマンドの -C 引数と同等です。

このプロパティを省略するか、このプロパティにデフォルト値の -1 を指定することは、pmfadm コマンドで --C オプションを省略するのと同じ効果があります。つまり、すべての子プロセスとその子孫プロセスが監視されます。

注 – このオプションは、標準的な Sun Cluster 管理コマンドを使用するときだけ指定できます。SunPlex Agent Builder を使用するときには指定できません。

Failover_enabled プロパティ

ブール値のこの拡張プロパティでは、リソースのフェイルオーバー動作を制御します。この拡張プロパティに true を設定すると、アプリケーションは、再起動回数が retry_interval 秒間に retry_count を超えるとフェイルオーバーされます。

この拡張プロパティに false を設定すると、再起動回数が retry_interval 秒間に retry_count を超えてもアプリケーションの再起動やフェイルオーバーは行われません。

この拡張プロパティを使用すれば、アプリケーションリソースがリソースグループのフェイルオーバーを引き起こすことを防止できます。デフォルトは true です。

Stop_signal プロパティ

GDS は、整数値のこの拡張プロパティを使って、PMF によるアプリケーションの停止に使用するシグナルを判別します。 指定できる整数値については、signal (3HEAD) のマニュアルページを参照してください。デフォルトは 15 (SIGTERM) です。

SunPlex Agent Builder を使って GDS ベースのサービスを作成

SunPlex Agent Builder を使ってGDS ベースのサービスを作成できます。SunPlex Agent Builder の詳細については、第 9 章を参照してください。

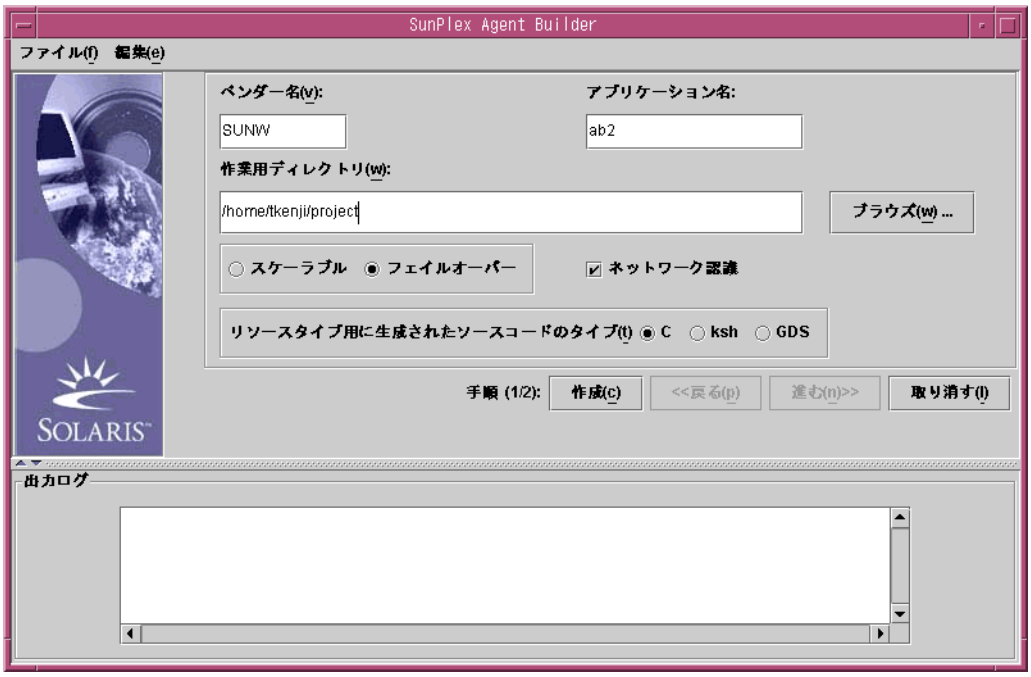
Agent Builder を使って GDS ベースのサービスを作成

▼ Agent Builder を使って GDS ベースのサービスを作成する方法

1. **SunPlex Agent Builder** を起動します。

   ```
   # /usr/cluster/bin/scdsbuilder
   ```

2. 「**SunPlex Agent Builder**」パネルが表示されます。

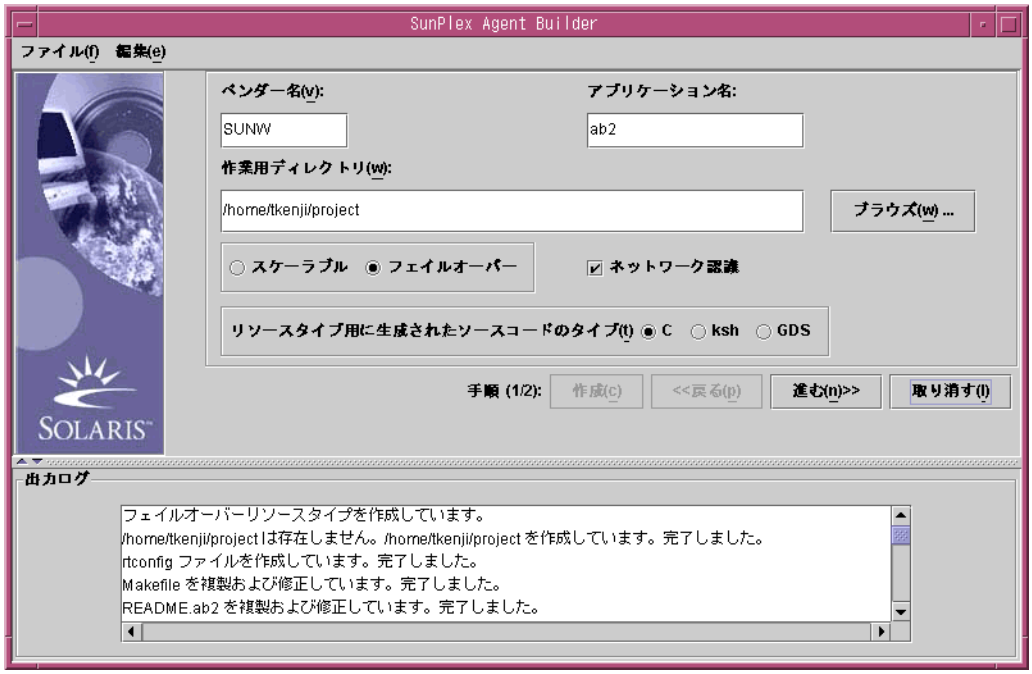


3. 「ベンダー名」にベンダー名を入力します。
4. 「アプリケーション名」にアプリケーション名を入力します。

注 – 「ベンダー名」と「アプリケーション名」の合計は 9 文字以内でなければなりません。この組み合わせは、起動スクリプトのパッケージ名として使用されます。

---

5. 作業ディレクトリに移動します。
   「ブラウズ」のプルダウンを使用すれば、パスを入力する代わりにディレクトリを選択することができます。
6. データサービスがスケーラブルなのかフェイルオーバーなのかを選択します。
   GDS を作成するときには「ネットワーク認識」がデフォルトですので、これを選択する必要はありません。
7. 「**GDS**」を選択します。
8. 「作成」ボタンをクリックして起動スクリプトを作成します。
9. サービスの作成結果が「**SunPlex Agent Builder**」パネルに表示されます。「作成」ボタンが無効になります。「進む」ボタンを押します。



▼ 起動スクリプトを構成する方法

起動スクリプトを作成したなら、SunPlex Agent Builder を使って新しいサービスを構成する必要があります。

1. 「進む」ボタンをクリックすると、構成パネルが表示されます。
2. 起動コマンドの場所を入力するか、「ブラウズ」ボタンを使って起動コマンドの場所を選択します。
3. **(**省略可能**)** 停止コマンドを入力するか、「ブラウズ」ボタンを使って停止コマンドの場所を選択します。
4. **(**省略可能**)** 検証コマンドを入力するか、「ブラウズ」ボタンを使って検証コマンドの場所を選択します。
5. **(**省略可能**)** 起動、停止、検証コマンドのタイムアウト値を指定します。
6. 「構成」をクリックして起動スクリプトの構成を開始します。

注 – このパッケージ名は、ベンダー名とアプリケーション名が結合したものです。

起動スクリプトのパッケージが作成され、次の場所に格納されます。

<working-dir>/*<vendor_name><application>*/pkg
たとえば、/export/wdir/NETapp/pkg のようになります。

7. 完成したパッケージをクラスタのすべてのノードにインストールします。

```
# cd /export/wdir/NETapp/pkg
# pkgadd -d . NETapp
```

8. **pkgadd** の実行で次のファイルがインストールされます。

```
/opt/NETapp
/opt/NETapp/README.app
/opt/NETapp/man
/opt/NETapp/man/man1m
/opt/NETapp/man/man1m/removeapp.1m
/opt/NETapp/man/man1m/startapp.1m
/opt/NETapp/man/man1m/stopapp.1m
/opt/NETapp/man/man1m/app_config.1m
/opt/NETapp/util
/opt/NETapp/util/removeapp
/opt/NETapp/util/startapp
/opt/NETapp/util/stopapp
/opt/NETapp/util/app_config
```

注 – マニュアルページとスクリプト名は、上で入力したアプリケーション名の前にスクリプト名を付けたものです。たとえば、startapp のようになります。

マニュアルページを表示するには、マニュアルページへのパスを指定する必要があります。たとえば、startapp のマニュアルページを表示する場合は、次のように入力します。

```
# man -M /opt/NETapp/man startapp
```

9. クラスタのいずれかのノードでリソースを構成し、アプリケーションを起動します。

/opt/NETapp/util/startapp -h *<logichostname>* **-p** *<port and protocol list>*
起動スクリプトの引数は、リソースのタイプがフェイルオーバーかスケーラブルかで異なります。カスタマイズしたマニュアルページを検査するか、起動スクリプトを引数なしで実行して引数リストを入手してください。

```
# /opt/NETapp/util/startapp
 The resource name of LogicalHostname or SharedAddress must be specified.
 For failover services:
 Usage: startapp -h <logical host name>
```

```
          -p <port and protocol list>
          [-n <ipmpgroup/adapter list>]
For scalable services:
Usage: startapp
          -h <shared address name>
          -p <port and protocol list>
          [-l <load balancing policy>]
          [-n <ipmpgroup/adapter list>]
          [-w <load balancing weights>]
```

SunPlex Agent Builder の出力

SunPlex Agent Builder は 3 つの起動スクリプトと、パッケージ作成時の入力内容に基づいた構成ファイルを生成します。構成ファイルには、リソースグループとリソースタイプの名前が指定されます。

生成される起動スクリプトは次のとおりです。

- 起動スクリプト: リソースを構成し、RGM 制御のもとでアプリケーションを起動します。
- 停止スクリプト: アプリケーションを停止し、リソースやリソースグループを停止します。
- 削除スクリプト: 起動スクリプトによって作成されたリソースやリソースグループを削除します。

これらの起動スクリプトのインタフェースや動作は、SunPlex Agent Builder によって非 GDS ベースのエージェント用に作成されるユーティリティスクリプトのものと同じです。これらのスクリプトは、複数のクラスタで再利用できる Solaris にインストール可能なパッケージとしてパッケージ化されます。

構成ファイルをカスタマイズすれば、リソースグループの名前など、一般には `scrgadm` コマンドへの入力として指定されるパラメータを独自に設定できます。スクリプトをカスタマイズしないと、SunPlex Agent Builder が `scrgadm` のパラメータに対し妥当なデフォルトを 設定します。

標準的な Sun Cluster 管理コマンドを使って GDS ベースのサービスを作成

この節では、これらのパラメータが実際に GDS に入力されるまでの手順を示します。GDS の使用や管理は、`scrgadm` や `scswitch` など、すでにある Sun Cluster 管理コマンドを通して行われます。

起動スクリプトに適切な機能が含まれている場合は、この節で述べる低位レベルの管理コマンドを入力する必要はありません。ただし、GDS ベースのリソースをより細かく制御したい場合は、このような低位レベルのコマンドを入力することもできます。起動スクリプトでは、これらのコマンドが実際に実行されます。

▼ Sun Cluster 管理コマンドを使って GDS ベースの高可用性サービスを作成する方法

1. リソースタイプ **SUNW.gds** を登録します。

```
# scrgadm -a -t SUNW.gds
```

2. **LogicalHostname** リソースとフェイルオーバーサービス自体を含むリソースグループを作成します。

```
# scrgadm -a -g haapp_rg
```

3. **LogicalHostname** リソースのリソースを作成します。

```
# scrgadm -a -L -g haapp_rs -l hhead
```

4. フェイルオーバーサービス自体のリソースを作成します。

```
# scrgadm -a -j haapp_rs -g haapp_rg -t SUNW.gds \
         -y Scalable=false -y Start_timeout=120 \
         -y Stop_timeout=120 -x Probe_timeout=120 \
         -y Port_list="2222/tcp" \
         -x Start_command="/export/ha/appctl/start" \
         -x Stop_command="/export/ha/appctl/stop" \
         -x Probe_command="/export/app/bin/probe" \
         -x Child_mon_level=0 -y Network_resources_used=hhead \
         -x Failover_enabled=true -x Stop_signal=9
```

5. リソースグループ **haapp_rg** をオンラインにします。

```
# scswitch -Z -g haapp_rg
```

▼ 標準的な Sun Cluster 管理コマンドを使って GDS ベースのスケーラブルサービスを作成する方法

1. リソースタイプ **SUNW.gds** を登録します。

```
# scrgadm -a -t SUNW.gds
```

2. **SharedAddress** リソースのリソースグループを作成します。

```
# scrgadm -a -g sa_rg
```

3. **SharedAddress** リソースを **sa_rg** に作成します。

```
# scrgadm -a -S -g sa_rg -l hhead
```

4. スケーラブルサービスのリソースグループを作成します。

```
# scrgadm -a -g app_rg -y Maximum_primaries=2 \
      -y Desired_primaries=2 -y RG_dependencies=sa_rg
```

5. スケーラブルサービス自体のリソースグループを作成します。

```
# scrgadm -a -j app_rs -g app_rg -t SUNW.gds \
       -y Scalable=true -y Start_timeout=120 \
       -y Stop_timeout=120 -x Probe_timeout=120 \
       -y Port_list="2222/tcp" \
       -x Start_command="/export/app/bin/start" \
       -x Stop_command="/export/app/bin/stop" \
       -x Probe_command="/export/app/bin/probe" \
       -x Child_mon_level=0 -y Network_resource_used=hhead \
       -x Failover_enabled=true -x Stop_signal=9
```

6. ネットワークリソースを含むリソースグループをオンラインにします。

```
 # scswitch -Z -g sa_rg
```

7. リソースグループ **app_rg** をオンラインにします。

```
# scswitch -Z -g app_rg
```

SunPlex Agent Builder のコマンド行インタフェース

SunPlex Agent Builder には、GUI インタフェースと同等の機能をもつコマンド行インタフェースがあります。このインタフェースは、`scsdcreate(1HA)` と `scdsconfig(1HA)` からなります。この節では、GUI ベースの手順 によるものと同じ機能をコマンド行インタフェースを使って行います。

▼ コマンド行バージョンの Agent Builder を使用して GDS ベースのサービスを作成する方法

1. サービスを作成します。

   フェイルオーバーサービスの場合:

```
# scdscreate -g -V NET -T app -d /export/wdir
```

   スケーラブルサービスの場合:

```
# scdscreate -g -s -V NET -T app -d /export/wdir
```

注 – *–d* パラメータは任意です。このパラメータを指定しない場合は、作業ディレクトリとして現在のディレクトリが使用されます。

2. サービスを構成します。

```
# scdsconfig -s "/export/app/bin/start' -t "/export/app/bin/stop" \
-m "/export/app/bin/probe" -d /export/wdir
```

注 – start コマンドだけが必須で、他のパラメータはすべて任意です。

3. 完成したパッケージをクラスタのすべてのノードにインストールします。

```
# cd /export/wdir/NETapp/pkg
# pkgadd -d . NETapp
```

4. **pkgadd** の実行で次のファイルがインストールされます。

```
/opt/NETapp
/opt/NETapp/README.app
/opt/NETapp/man
/opt/NETapp/man/man1m
/opt/NETapp/man/man1m/removeapp.1m
/opt/NETapp/man/man1m/startapp.1m
/opt/NETapp/man/man1m/stopapp.1m
/opt/NETapp/man/man1m/app_config.1m
/opt/NETapp/util
/opt/NETapp/util/removeapp
/opt/NETapp/util/startapp
/opt/NETapp/util/stopapp
/opt/NETapp/util/app_config
```

注 – マニュアルページとスクリプト名は、上で入力したアプリケーション名の前にスクリプト名を付けたものです。たとえば、`startapp` のようになります。

マニュアルページを表示するには、マニュアルページへのパスを指定する必要があります。たとえば、`startapp` のマニュアルページを表示する場合は、次のように入力します。

```
# man -M /opt/NETapp/man startapp
```

5. クラスタのいずれかのノードでリソースを構成し、アプリケーションを起動します。

```
# /opt/NETapp/util/startapp -h <logichostname> -p <port and protocol list>
```

起動スクリプトの引数は、リソースのタイプがフェイルオーバーかスケーラブルかで異なります。カスタマイズしたマニュアルページを検査するか、起動スクリプト

を引数なしで実行して引数リストを入手してください。

```
# /opt/NETapp/util/startapp
 The resource name of LogicalHostname or SharedAddress must be specified.
 For failover services:
 Usage: startapp -h <logical host name>
           -p <port and protocol list>
           [-n <ipmpgroup/adapter list>]
 For scalable services:
 Usage: startapp
           -h <shared address name>
           -p <port and protocol list>
           [-l <load balancing policy>]
           [-n <ipmpgroup/adapter list>]
           [-w <load balancing weights>]
```

第 11 章

データサービス開発ライブラリのリファレンス

この章では、 データサービス開発ライブラリ (Data Service Development Library: DSDL) の API 関数の一覧を示し、概要を述べます。個々の DSDL 関数の詳細については、そのマニュアルページ (3HA) を参照してください。DSDL は C 言語用のインタフェースだけを定義します。スクリプト用の DSDL インタフェースはありません。

DSDL が提供する関数は、次のカテゴリに分類されます。

DSDL 関数

この節では、DSDL 関数の各カテゴリを簡単に説明します。DSDL 関数を定義するリファレンスについては、個々のマニュアルページ (3HA) を参照してください。

汎用関数

このカテゴリの関数は、さまざまな機能を提供します。これらの関数は、次のような処理を行います。

- DSDL 環境を初期化します。
- リソース、リソースタイプ、およびリソースグループの名前、ならびに、拡張プロパティの値を取得します。
- リソースグループをフェイルオーバーおよび再起動し、リソースを再起動します。

- エラー文字列をエラーメッセージに変換します。
- タイムアウトを適用してコマンドを実行します。

次の関数は、呼び出しメソッドを初期化します。

- `scds_initialize` – リソースを割り当て、DSDL 環境を初期化します。
- `scds_close` – `scds_initialize` が割り当てたリソースを解放します。

次の関数は、リソース、リソースタイプ、リソースグループ、および拡張プロパティについての情報を取得します。

- `scds_get_resource_name` – 呼び出しプログラム用のリソース名を取得します。
- `scds_get_resource_type_name` – 呼び出しプログラム用のリソースタイプ名を取得します。
- `scds_get_resource_group_name` – 呼び出しプログラム用のリソースグループ名を取得します。
- `scds_get_ext_property` – 指定した拡張プロパティの値を取得します。
- `scds_free_ext_property` – `scds_get_ext_property` が割り当てたメモリーを解放します。

次の関数は、リソースが使用している SUNW.HAStoragePlus リソースについての状態情報を取得します。

- `scds_hasp_check` – リソースが使用している `SUNW.HAStoragePlus` リソースについての状態情報を取得します。当該リソース用に定義されている `Resource_dependencies` または `Resource_dependencies_weak` のシステム属性を使用することによって、当該リソースが依存しているすべての `SUNW.HAStoragePlus` リソース状態 (オンラインであるか、オンラインでないか) についての情報が得られます。

  `SUNW.HAStoragePlus` の詳細は、`SUNW.HAStoragePlus(5)` を参照してください。

次の関数は、リソースまたはリソースグループをフェイルオーバーまたは再起動します。

- `scds_failover_rg` – リソースグループをフェイルオーバーします。
- `scds_restart_rg` – リソースグループを再起動します。
- `scds_restart_resource` – リソースを再起動します。

次の 2 つの関数は、タイムアウトを適用してコマンドを実行し、エラーコードをエラーメッセージに変換します。

- `scds_timerun` – タイムアウトを適用してコマンドを実行します。
- `scds_error_string` – エラーコードをエラーメッセージに変換します。

プロパティ関数

このカテゴリの関数は、関連するリソース、リソースグループ、およびリソースタイプ (よく使用される一部の拡張プロパティも含む) に固有なプロパティにアクセスするのに有用な API を提供します。 DSDL は、`scds_initialize` を使用して、コマンド行引数を解析します。`scds_initialize`(3HA) 関数は、関連するリソース、リソースグループ、およびリソースタイプの様々なプロパティをキャッシュに入れます。

これらすべての関数については、`scds_property_functions`(3HA) のマニュアルページを参照してください。このカテゴリには、次の関数が含まれます。

- `scds_get_rs_`*property_name*
- `scds_get_rg_`*property_name*
- `scds_get_rt_`*property_name*
- `scds_get_ext_`*property_name*

ネットワークリソースアクセス関数

このカテゴリの関数は、リソースおよびリソースグループが使用するネットワークリソースを、取得、出力、および解放します。ここで説明する `scds_get_*` 関数は、RMAPI 関数を使用して `Network_resources_used` や `Port_list` などのプロパティを照会しなくても、ネットワークリソースを取得できる便利な方法を提供します。`scds_print_`*name*`()` 関数は、`scds_get_ `*name*`()` 関数から戻されたデータ構造から値を出力します。`scds_free_ `*name*`()` 関数は、`scds_get_`*name*`()` 関数が割り当てたメモリーを解放します。

次の関数は、ホスト名に関連した処理を行います。

- `scds_get_rg_hostnames` – ネットワークグループ内のネットワークリソースが使用するホスト名のリストを取得します。
- `scds_get_rs_hostnames` – リソースが使用するホスト名のリストを取得します。
- `scds_print_net_list` – `scds_get_rg_hostnames` または `scds_get_rs_hostnames` が戻したホスト名のリストの内容を出力します。
- `scds_free_net_list` – `scds_get_rg_hostnames` または `scds_get_rs_hostnames` が割り当てたメモリーを解放します。

次の関数は、ポートリストに関連した処理を行います。

- `scds_get_port_list` – リソースが使用するポートとプロトコルのペアのリストを取得します。
- `scds_print_port_list` – `scds_get_port_list` が戻したポートとプロトコルのペアのリストの内容を出力します。
- `scds_free_port_list` – `scds_get_port_list` が割り当てたメモリーを解放します。

次の関数は、ネットワークアドレスに関連した処理を行います。

- `scds_get_netaddr_list` – リソースが使用するネットワークアドレスのリストを取得します。
- `scds_print_netaddr_list` – `scds_get_netaddr_list` が戻したネットワークアドレスのリストの内容を出力します。
- `scds_free_netaddr_list` – `scds_get_netaddr_list` が割り当てたメモリーを解放します。

TCP 接続を使用する障害監視

このカテゴリの関数は、TCP ベースの監視を行います。通常、障害モニターはこれらの関数を使用して、サービスとの単純ソケット接続を確立し、サービスのデータを読み書きしてサービスの状態を確認した後、サービスとの接続を切断します。

このカテゴリには、次の関数が含まれます。

- `scds_tcp_connect` – プロセスとの TCP 接続を確立します。
- `scds_tcp_read` – TCP 接続を使用して、監視対象のプロセスからデータを読み取ります。
- `scds_tcp_write` – TCP 接続を使用して、監視対象のプロセスにデータを書き込みます。
- `scds_simple_probe` – プロセスとの TCP 接続を確立および終了することによって、プロセスを検証します。
- `scds_tcp_disconnect` – 監視対象のプロセスとの接続を終了します。

PMF 関数

このカテゴリの関数は、PMF 機能をカプセル化します。PMF 経由の監視における DSDL モデルは、`pmfadm(1M)` に対して、暗黙のタグ値を作成および使用します。また、PMF 機能は、`Restart_interval`、`Retry_count`、および `action_script` 用の暗黙値も使用します (`pmfadm` の `-t`、`-n`、および `-a` オプション)。最も重要な点は、DSDL が、PMF によって検出されたプロセス停止履歴を、障害モニターによって検出されたアプリケーション障害履歴に結びつけ、再起動またはフェイルオーバーのどちらを行うかを決定することです。

このカテゴリには、次の関数が含まれます。

- `scds_pmf_get_status` – 指定されたインスタンスが PMF の制御下で監視されているかどうかを判定します。
- `scds_pmf_restart_fm` – PMF を使用して障害モニターを再起動します。
- `scds_pmf_signal` – 指定されたシグナルを PMF の制御下で動作しているプロセスツリーに送信します。

- scds_pmf_start – 指定されたプログラム (障害モニターを含む) を PMF の制御下で実行します。
- scds_pmf_stop – PMF の制御下で動作しているプロセスを終了します。
- scds_stop_monitoring — PMF の制御下で動作しているプロセスの監視を停止します。

障害監視関数

このカテゴリの関数は、障害履歴を保持し、その履歴を Retry_count および Retry_interval プロパティと関連付けて評価することにより、障害監視の事前定義モデルを提供します。

このカテゴリには、次の関数が含まれます。

- scds_fm_sleep – 障害モニター制御ソケット上でメッセージを待ちます。
- scds_fm_action – 検証終了後にアクションを実行します。
- scds_fm_print_probes – 検証状態情報をシステムログに書き込みます。

ユーティリティ関数

このカテゴリの関数は、メッセージやデバッグ用メッセージをシステムログに書き込みます。このカテゴリには、次の 2 つの関数が含まれます。

- scds_syslog – メッセージをシステムログに書き込みます。
- scds_syslog_debug – デバッグ用メッセージをシステムログに書き込みます。

第 12 章

CRNP

この章では、Cluster Reconfiguration Notification Protocol (CRNP) について説明します。CRNP を使用することで、フェイルオーバー用のアプリケーションや拡張性のあるアプリケーションを「クラスタ対応」として設定できます。具体的には、Sun Cluster 再構成イベントにアプリケーションを登録し、それらのイベントの後続の非同期通知を受け取ることができます。イベント通知の受信登録が可能なのは、クラスタの内部で動作するデータサービスと、クラスタの外部で動作するアプリケーションです。イベントは、クラスタ内のメンバーシップに変化があった場合と、リソースグループまたはリソースの状態に変化があった場合に生成されます。

CRNP の概要

CRNP は、クラスタ再構成イベントの生成、クラスタへの配信、それらのイベントを要求しているクライアントへの送信を行うメカニズムとデーモンを提供します。

クライアントとの通信を行うのは、`cl_apid` デーモンです。クラスタ再構成イベントの生成は、Sun Cluster Resource Group Manager (RGM) によって行われます。これらのデーモンは、`sysevent(1M)` を使用して各ローカルノードにイベントを転送します。`cl_apid` デーモンは、TCP/IP 上で XML (Extensible Markup Language) を使用して要求クライアントとの通信を行います。

次の図は、CRNP コンポーネント間のイベントの流れを簡単に示したものです。この図では、一方のクライアントはクラスタノード 2 で動作し、他方のクライアントはクラスタに属していないコンピュータ上で動作しています。

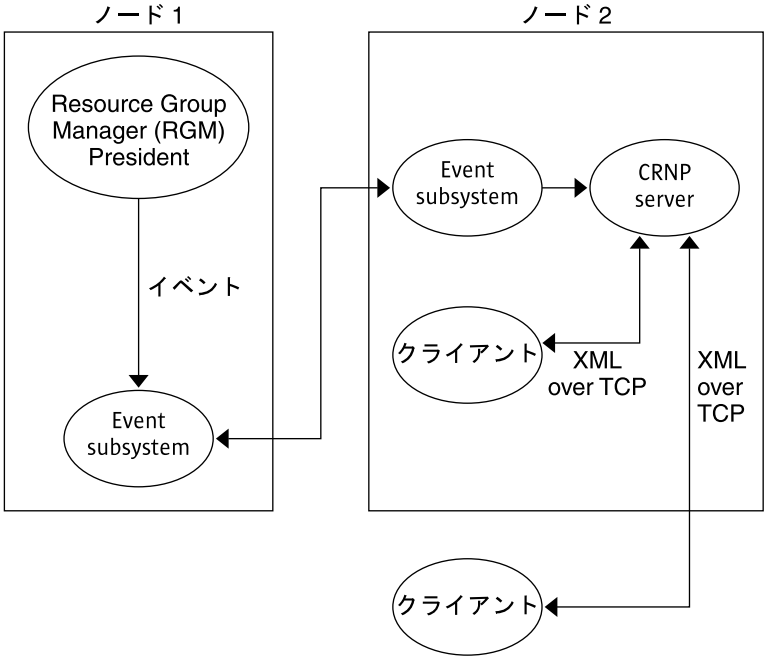


図 **12–1** CRNP の動作

CRNP プロトコルの概要

CRNP は、標準の 7 層 OSI (Open System Interconnect) プロトコルスタックにおけるアプリケーション層、プレゼンテーション層、およびセッション層を定義します。トランスポート層は TCP でなければならず、ネットワーク層は IP でなければなりません。CRNP は、データリンク層および物理層とは無関係です。CRNP 内で交換されるアプリケーション層メッセージはすべて、XML 1.0 をベースとしたものです。

CRNP プロトコルのセマンティクス

クライアントは、サーバーへ登録メッセージ (`SC_CALLBACK_RG`) を送信することによって通信を開始します。この登録メッセージは、通知を受信したいイベントタイプと、イベントの配信先として使用できるポートを指定するものです。登録用接続のソース IP と指定ポートから、コールバックアドレスが構成されます。

クライアントが配信を希望しているイベントがクラスタ内で生成されるたびに、サーバーはこのコールバックアドレス (IP とポート) を持つクライアントと通信を行い、イベント (SC_EVENT) をクライアントに配信します。サーバーには、そのクラスタ内で稼動している高可用マシンが使用されます。サーバーは、クラスタの再起動後も維持されるストレージにクライアントの登録情報を格納します。

登録解除を行う場合、クライアントはサーバーに登録メッセージ (REMOVE_CLIENT メッセージが入った SC_CALLBACK_RG) を送信します。サーバーから SC_REPLY メッセージを受け取ったあとで、クライアントは接続を閉じることができます。

次の図は、クライアントとサーバー間の通信の流れを示します。

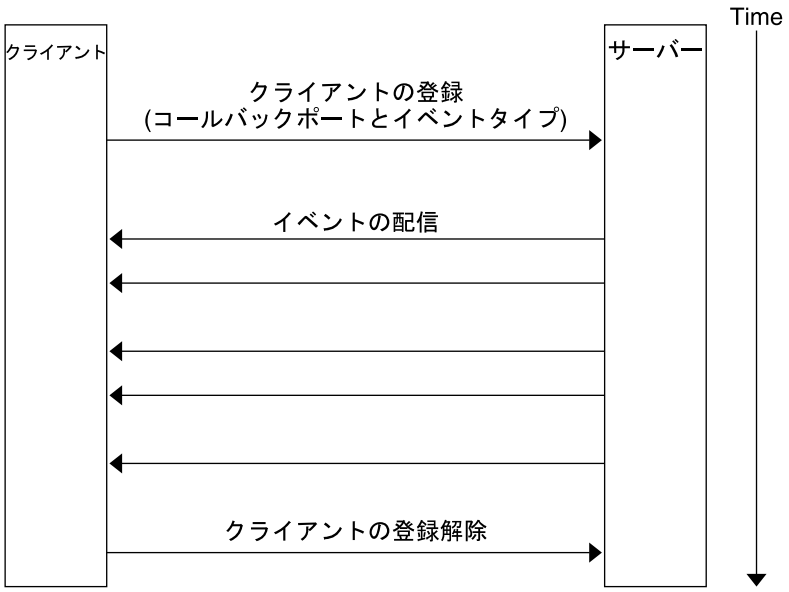


図 **12–2** クライアントとサーバー間の通信の流れ

CRNP が使用するメッセージのタイプ

CRNP は、3 種類のメッセージを使用します。これらはすべて、次の表に示すように XML ベースのメッセージです。これらのメッセージタイプの内容と使用法の詳細は、この章で後述します。

メッセージのタイプ	説明
SC_CALLBACK_REG	このメッセージには、4 つのフォーム、ADD_CLIENT、REMOVE_CLIENT、ADD_EVENTS、および REMOVE_EVENTS を指定できます。これらの各フォームには、次の情報が含まれます。 ■ プロトコルバージョン ■ ASCII 形式で示されたコールバックポート (バイナリ形式ではない) ADD_CLIENT、ADD_EVENTS 、REMOVE_EVENTS の各フォームには制約のないイベントタイプリストも含まれ、それぞれに次の情報が含まれます。 ■ イベントクラス ■ イベントサブクラス (省略可能) ■ 名前と値がペアになったリスト (省略可能) イベントクラスとイベントサブクラスにより一意の「イベントタイプ」が定義されます。SC_CALLBACK_REG のクラスを生成する DTD (document type definition: ドキュメントタイプ定義) は SC_CALLBACK_REG です。この DTD の詳細は、付録 F を参照してください。
SC_EVENT	このメッセージには次の情報が含まれます。 ■ プロトコルバージョン ■ イベントクラス ■ イベントサブクラス ■ ベンダー ■ パブリッシャー ■ 名前と値のペアリスト (名前と値をペアにした 0 個以上のデータ構造) ■ 名前 (文字列) ■ 値 (文字列または文字列配列) SC_EVENT 内の値はタイプとしては分類されていません。SC_EVENT のクラスを生成する DTD (ドキュメントタイプ定義) は SC_EVENT です。この DTD の詳細は、付録 F を参照してください。
SC_REPLY	このメッセージには次の情報が含まれます。 ■ プロトコルバージョン ■ エラーコード ■ エラーメッセージ: SC_REPLY のクラスを生成する DTD (ドキュメントタイプ定義) は SC_REPLY です。この DTD の詳細は、付録 F を参照してください。

クライアントをサーバーに登録する方法

この節では、サーバーの設定、クライアントの識別、アプリケーション層とセッション層での情報送信、エラー状況などについて説明します。

管理者によるサーバー設定の前提

システム管理者は、汎用 IP アドレス (クラスタ内の特定のマシン専用でない IP アドレス) とポート番号を使用してサーバーを構成し、クライアントとなるマシンにこのネットワークアドレスを公開する必要があります。CRNP では、クライアントがこのサーバー名をどのように取得するかは定義されていません。管理者は、ネーミングサービスを使用することも (この場合、クライアントは動的にサーバーのネットワークアドレスを検出できる)、ネットワーク名を構成ファイルに追加してクライアントに読み取らせることもできます。サーバーは、クラスタ内でフェイルオーバーリソースタイプとして動作します。

サーバーによるクライアントの識別方法

各クライアントは、そのコールバックアドレス (IP アドレスとポート番号) で識別されます。ポートは SC_CALLBACK_REG メッセージで指定され、IP アドレスは登録用の TCP 接続から取得されます。CRNP は、同じコールバックアドレスを持つ後続の SC_CALLBACK_REG メッセージは同じクライアントから送信されたと想定します。これは、メッセージの送信元であるソースポートが異なる場合でも同様です。

クライアントとサーバー間での SC_CALLBACK_REG メッセージの受け渡し方法

クライアントは、サーバーの IP アドレスとポート番号に対して TCP 接続を開くことによって登録を開始します。TCP 接続が確立され書き込みの用意ができたところで、クライアントはその登録メッセージを送信する必要があります。 この登録メッセージは正しい書式の SC_CALLBACK_REG メッセージでなければならず、メッセージの前後に余分なバイトを含めることはできません。

バイトがすべてストリームに書き込まれたあと、クライアントはサーバーから応答を受け取ることができるように接続をオープン状態に保つ必要があります。クライアントが不正な書式のメッセージを送信した場合、サーバーはそのクライアントを登録せず、クライアントに対してエラー応答を送信します。サーバーが応答を送信する前にクライアントがソケット接続を閉じた場合、サーバーはそのクライアントを正常なクライアントとして登録します。

クライアントは、いつでもサーバーと通信を行うことができます。サーバーと通信を行うごとに、クライアントは SC_CALLBACK_REG メッセージを送信する必要があります。書式が不正なメッセージ、順不同のメッセージ、無効なメッセージなどを受け取った場合、サーバーはクライアントにエラー応答を送信します。

クライアントは、それ自体が ADD_CLIENT メッセージを送信するまでは ADD_EVENTS、REMOVE_EVENTS 、REMOVE_CLIENT メッセージを送信できません。また、ADD_CLIENT メッセージを送信しないかぎり REMOVE_CLIENT メッセージも送信できません。

クライアントが ADD_CLIENT メッセージを送信したが、そのクライアントがすでに登録されていたという場合は、サーバーがこのメッセージを黙認することがあります。このような場合、サーバーは報告なしに古いクライアント登録を削除し、2 つめの ADD_CLIENT メッセージに指定された新しいクライアント登録に置き換えます。

通常、クライアントはその起動時に ADD_CLIENT メッセージを送信することによって、サーバーに一度だけ登録を行います。また、登録の解除もサーバーに REMOVE_CLIENT メッセージを送信して一度だけ行います。しかし、CRNP はクライアントが必要に応じてイベントタイプリストを動的に変更できるだけの柔軟性を備えています。

SC_CALLBACK_REG メッセージの概念

ADD_CLIENT、ADD_EVENTS、および REMOVE_EVENTS メッセージには、それぞれイベントリストが含まれます。次の表は、CRNP が受け付けるイベントタイプを、必要となる名前と値のペアと共に示して説明しています。

クライアントが以下の作業のどちらか一方を行うと、

- まだ登録が行われていないイベントタイプを 1 つ以上指定する REMOVE_EVENTS メッセージを送信する
- 同じイベントタイプを 2 度登録する

サーバーはクライアントに通知することなくこれらのメッセージを無視します。

クラスとサブクラス	名前と値のペア	説明
EC_Cluster ESC_cluster_membership	必須: なし (省略可能) なし	クラスタメンバーシップの変更 (ノードの停止または結合) に関連するあらゆるイベントに登録する
EC_Cluster ESC_cluster_rg_state	次の条件で 1 つ必要: rg_name 値のタイプ: 文字列 (省略可能) なし	リソースグループ *name* のあらゆる状態変更イベントに登録する
EC_Cluster ESC_cluster_r_state	次の条件で 1 つ必要: r_name 値のタイプ: 文字列 (省略可能) なし	リソース *name* のあらゆる状態変更イベントに登録する
EC_Cluster なし	必須: なし 省略可能: なし	あらゆる Sun Cluster イベントに登録する

クライアントに対するサーバーの応答方法

登録を処理したあと、サーバーは SC_REPLY メッセージを送信します。この送信は、登録要求を行なったクライアントがオープン状態に維持している TCP 接続に対して行われます。このあとサーバーはこの接続を閉じます。クライアントは、サーバーから SC_REPLY メッセージを受信するまで TCP 接続をオープン状態に保つ必要があります。

次に、クライアント側の作業例を示します。

1. サーバーに対して TCP 接続を開きます。
2. 接続が「writeable (書き込み可能)」になるまで待機します。
3. SC_CALLBACK_REG メッセージ (このメッセージには ADD_CLIENT メッセージが入っている) を送信します。
4. SC_REPLY メッセージの到着を待機します。
5. SC_REPLY メッセージを受け取ります。
6. サーバーが接続を閉じたことを示すインジケータを受信します (ソケットから 0 バイトを読み取る)。
7. 接続を閉じます。

その後クライアントは以下の作業を行います。

1. サーバーに対して TCP 接続を開きます。
2. 接続が「writeable (書き込み可能)」になるまで待機します。
3. SC_CALLBACK_REG メッセージ (このメッセージには REMOVE_CLIENT メッセージが入っている) を送信します。
4. SC_REPLY メッセージの到着を待機します。
5. SC_REPLY メッセージを受け取ります。
6. サーバーが接続を閉じたことを示すインジケータを受信します (ソケットから 0 バイトを読み取る)。
7. 接続を閉じます。

クライアントから SC_CALLBACK_REG メッセージを受け取るたびに、サーバーは同じ接続に SC_REPLY メッセージを送信します。このメッセージは、処理が正常に完了したか失敗したかを示すものです。SC_REPLY メッセージの XML ドキュメントタイプ定義とこのメッセージ内で示されるエラーメッセージについては、324 ページの「SC_REPLY XML DTD」を参照してください。

SC_REPLY メッセージの内容

SC_REPLY メッセージは、処理が正常に完了したか失敗したかを示します。このメッセージには、CRNP プロトコルメッセージのバージョン、ステータスコード、およびステータスコードの詳細を説明したステータスメッセージが含まれます。次の表は、ステータスコードの値を説明しています。

ステータスコード	説明
OK	メッセージは正常に処理されました。
RETRY	一時的なエラーのためにクライアントの登録はサーバーに拒否されました。クライアントは別のパラメータを使用して登録をもう一度試す必要があります。
LOW_RESOURCE	クラスタのリソースが少ないため、クライアントはあとでもう一度試すか、システム管理者にクラスタのリソースを増やしてもらう必要があります。
SYSTEM_ERROR	重大な問題が発生しました。クラスタのシステム管理者に連絡してください。
FAIL	承認の失敗などの問題が発生し、登録が失敗しました。
MALFORMED	XML 要求の形式が正しくないため解析が失敗しました。
INVALID	XML 要求が無効です (XML 仕様を満たしていない)。
VERSION_TOO_HIGH	メッセージのバージョンが高すぎて、メッセージを正常に処理できませんでした。
VERSION_TOO_LOW	メッセージのバージョンが低すぎて、メッセージを正常に処理できませんでした。

クライアントによるエラー状況の処理

通常、SC_CALLBACK_REG メッセージを送信するクライアントは登録の成功または失敗を知らせる応答を受け取ります。

しかし、クライアントが登録を試みる際にサーバーからの SC_REPLY メッセージの送信を妨げるエラーが発生することがあります。この場合、エラーが発生する前に登録が正常に完了することも、登録が失敗することも、あるいは登録処理が行われないまま終了することもあります。

サーバーはクラスタ上でフェイルオーバー (高可用) サーバーとして機能するため、このエラーがサービスの終了を意味するわけではありません。実際、サーバーは新しく登録されたクライアントに対してすぐにイベント送信を開始できます。

これらの状況を修復するには、クライアントは次の 2 つの作業を行う必要があります。

- SC_REPLY メッセージを待機している登録用接続にアプリケーションレベルのタイムアウトを強制します (このあと、登録を再試行する必要がある)。
- イベントコールバックの登録を行う前に、イベント配信用のコールバック IP アドレスとポート番号で待機を開始します。クライアントは、登録確認メッセージとイベント配信を同時に待機することになります。確認メッセージを受信する前にイベントを受信し始めた場合は、クライアントはそのまま登録接続を閉じる必要があります。

サーバーがクライアントにイベントを配信する方法

クラスタ内でイベントが生成されると、CRNP サーバーはそのタイプのイベントを要求したすべてのクライアントにイベントの配信を行います。この配信では、クライアントのコールバックアドレスに SC_EVENT メッセージが送信されます。各イベントの配信は、新たな TCP 接続で行われます。

クライアントが ADD_CLIENT メッセージまたは ADD_EVENT メッセージが入った SC_CALLBACK_REG メッセージを通してイベントタイプの配信登録を行うと、サーバーはただちにクライアントに対してそのタイプの最新イベントを送信します。続いてクライアントは、後続のイベントを送信するシステムの現在の状態を検出できます。

クライアントに対して TCP 接続を開始する際に、サーバーはその接続に SC_EVENT メッセージを 1 つだけ送信します。続いてサーバーは全二重通信を閉じます。

クライアントは次のような作業を行います。

1. サーバーが TCP 接続を開始するのを待機します。
2. サーバーからの着信接続を受け入れます。
3. SC_EVENT メッセージの到着を待機します。
4. SC_EVENT メッセージを読み取ります。
5. サーバーが接続を閉じたことを示すインジケータを受信します (ソケットから 0 バイトを読み取る)。
6. 接続を閉じます。

すべてのクライアントが登録を終了した時点で、それらのクライアントはイベント配信のための着信接続を受け入れるために常にコールバックアドレス (IP アドレスとポート番号) で待機する必要があります。

クライアントとの通信に失敗してイベントを配信できなかった場合、サーバーはユーザーが設定してある回数と周期に従ってイベントの配信を繰り返し試みます。それらの試行がすべて失敗に終わった場合、そのクライアントはサーバーのクライアントリストから削除されます。 イベントをそれ以上受け取るためには、クライアントは ADD_CLIENT メッセージが入った SC_CALLBACK_REG メッセージを別途送信して登録をもう一度行う必要があります。

イベント配信の保証

クラスタ内では、クライアントごとに配信順序を守るという方法で、トータル的にイベント生成を順序付けます。たとえば、クラスタ内でイベント A の生成後イベント B が生成された場合、クライアント X はイベント A を受け取ってからイベント B を受け取ります。しかし、全クライアントに対するイベント配信の全体的な順序付けは保持されません。つまり、クライアント Y はクライアント X がイベント A を受け取る前にイベント A と B の両方を受け取る可能性があります。この方法では、低速のクライアントのために全クライアントへの配信が停滞するということがありません。

サーバーが配信するイベントはすべて (サブクラス用の最初のイベントとサーバーエラーのあとに発生するイベントを除く)、クラスタが生成する実際のイベントに応答して発生します。ただし、クラスタで生成されるイベントを見逃すようなエラーが発生する場合は、サーバーは各イベントタイプの現在のシステム状態を示すイベントをそれらのイベントタイプごとに生成します。各イベントは、そのイベントタイプの配信登録を行なったクライアントに送信されます。

イベント配信は、「1 回以上」というセマンティクスに従って行われます。つまり、サーバーは 1 台のクライアントに対して同じイベントを複数回送信できます。この許可は、サーバーが一時的に停止して復帰した際に、クライアントが最新の情報を受け取ったかどうかをサーバーが判断できないという場合に不可欠なものです。

SC_EVENT メッセージの内容

SC_EVENT メッセージには、クラスタ内で生成されて SC_EVENT XML メッセージ形式に合うように変換された実際のメッセージが入っています。次の表は、CRNP が配信するイベントタイプ (名前と値のペア、パブリッシャー、ベンダーなど) を説明したものです。

クラスとサブクラス	パブリッシャーとベンダー	名前と値のペア	注
EC_Cluster ESC_cluster_membership	パブリッシャー: rgm ベンダー: SUNW	名前: node_list 値のタイプ: 文字配列 名前: state_list 値のタイプ: 文字配列	state_list の配列要素は、node_list の配列要素と同期をとるように配置されます。つまり、node_list 配列内で最初に出現しているノードの状態は、state_list 配列の先頭に示されます。 state_list には、ASCII 形式の数字だけが入っています。各数字は、クラスタにおけるそのノードの現在のインカーネーション番号を示します。この番号が前のメッセージで受信した番号と同じである場合、ノードとクラスタの関係は変化していません (離脱、結合、または再結合が行われていない)。インカーネーション番号が –1 の場合、ノードはクラスタのメンバーではありません。インカーネーション番号が負の値以外の数字である場合、ノードはクラスタのメンバーです。 ev_ で始まるほかの名前や、それらの名前に関連した値が存在する場合がありますが、クライアントによる使用を意図したものではありません。

クラスとサブクラス	パブリッシャーとベンダー	名前と値のペア	注
EC_Cluster ESC_cluster_rg_state	パブリッシャー: rgm ベンダー: SUNW	名前: rg_name 値のタイプ: 文字列 名前: node_list 値のタイプ: 文字配列 名前: state_list 値のタイプ: 文字配列	state_list の配列要素は、node_list の配列要素と同期をとるように配置されます。つまり、node_list 配列内で最初に出現しているノードの状態は、state_list 配列の先頭に示されます。 state_list には、リソースグループの状態を示す文字列が入っています。有効なのは、scha_cmds(1HA) コマンドで取得できる値です。 ev_ で始まるほかの名前や、それらの名前に関連した値が存在する場合がありますが、クライアント使用を意図したものではありません。
EC_Cluster ESC_cluster_r_state	パブリッシャー: rgm ベンダー: SUNW	次の条件で 3 つ必要: 名前 r_name 値のタイプ: 文字列 名前 node_list 値のタイプ: 文字配列 名前 state_list 値のタイプ: 文字配列	state_list の配列要素は、node_list の配列要素と同期をとるように配置されます。つまり、node_list 配列内で最初に出現しているノードの状態は、state_list 配列の先頭に示されます。 state_list には、リソースの状態を示す文字列が入っています。有効なのは、 scha_cmds (1HA) コマンドで取得できる値です。 ev_ で始まるほかの名前や、それらの名前に関連した値が存在する場合がありますが、クライアント使用を意図したものではありません。

CRNP によるクライアントとサーバーの認証

サーバーは、TCP ラッパーを使用してクライアントの認証を行います。この場合、登録メッセージのソース IP アドレス (これはイベントの配信先であるコールバック IP アドレスとしても使用される) がサーバー側の「許可されたユーザー」リストに含まれていなければなりません。ソース IP アドレスと登録メッセージが「拒否されたクライアント」リストに存在してはなりません。ソース IP アドレスと登録メッセージがリスト中に存在しない場合、サーバーは要求を拒否し、クライアントに対してエラー応答を返します。

サーバーが `SC_CALLBACK_REG ADD_CLIENT` メッセージを受けとる場合、そのクライアントの後続の `SC_CALLBACK_REG` メッセージには最初のメッセージ内のものと同じソース IP アドレスが含まれていなければなりません。この条件を満たさない `SC_CALLBACK_REG` を受信した場合、CRNP サーバーは次のどちらかを選択します。

- 要求を無視し、クライアントにエラー応答を送信する
- その要求が新しいクライアントからのものであると想定する (`SC_CALLBACK_REG` メッセージの内容にもとづいて判断)

このセキュリティメカニズムは、正規クライアントの登録の解除を試みるサービス拒否攻撃の防止に役立ちます。

クライアントも、同様のサーバー認証を行う必要があります。クライアントは、それ自体が使用した登録 IP アドレスおよびポート番号と同じソース IP アドレスおよびポート番号を持つサーバーからのイベント配信を受け入れるだけです。

CRNP サービスのクライアントはクラスタを保護するファイアウォール内に配置されるのが一般的なため、CRNP にセキュリティメカニズムは提供されていません。

CRNP を使用する Java アプリケーションの作成

以下の例は、CRNP を使用する `CrnpClient` というシンプルな Java アプリケーションを作成する方法を示しています。このアプリケーションでは、クラスタ上の CRNP サーバーへのイベントコールバックの登録、イベントコールバックの待機、イベントの処理 (内容の出力) を行い、終了前にイベントコールバック要求の登録解除を行います。

この例を参照する場合は、以下の点に注意してください。

- このアプリケーション例は、JAXP (XML 処理用の Java API) による XML 生成と解析を行います。この例は JAXP の使用方法を説明したものではありません。JAXP の詳細は、`http://java.sun.com/xml/jaxp/index.html` で説明しています。
- この例は、付録 G に示されている完全なアプリケーションコードを断片的に示したものです。この章の例は個々の概念を効果的に示すことをねらっており、付録 G に示されている完全なアプリケーションコードと多少異なります。
- また、簡潔に示すため、この章の例ではコード例からコメントを除いてあります。付録 G に示されている完全なアプリケーションコードにはコメントが含まれています。
- この例に示しているアプリケーションは終了するだけでほとんどのエラー状況に対応できるものですが、ユーザーが実際に使用するアプリケーションではエラーを徹底的に処理する必要があります。

▼ 環境の設定

まず、環境の設定を行う必要があります。

1. **JAXP** と、正しいバージョンの **Java** コンパイラおよび **Virtual Machine** をダウンロードし、インストールを行います。

   作業手順は、`http://java.sun.com/xml/jaxp/index.html` に示されています。

   注 – この例は、バージョン 1.3.1 以降の Java を必要とします。

   ---

2. コンパイラが **JAXP** クラスを見つけることができるように、コンパイルのコマンド行に必ず **`classpath`** を指定する必要があります。ソースファイルが置かれているディレクトリから、次のように入力します。

   ```
   % javac -classpath JAXP_ROOT/dom.jar:JAXP_ROOTjaxp-api. \
   jar:JAXP_ROOTsax.jar:JAXP_ROOTxalan.jar:JAXP_ROOT/xercesImpl \
   .jar:JAXP_ROOT/xsltc.jar -sourcepath . SOURCE_FILENAME.java
   ```

   上記コマンドの *JAXP_ROOT* には、JAXP jar ファイルが置かれているディレクトリの絶対パスまたは相対パスを指定してください。 *SOURCE_FILENAME* には、Java ソースファイルの名前を指定してください。

3. アプリケーションの実行時に、アプリケーションが適切な **JAXP** クラスファイルを読み込むことができるように **`classpath`** を指定します **(`classpath` の最初のパスは現在のディレクトリ)**。

   ```
   java -cp .:JAXP_ROOT/dom.jar:JAXP_ROOTjaxp-api. \
   jar:JAXP_ROOTsax.jar:JAXP_ROOTxalan.jar:JAXP_ROOT/xercesImpl \
   .jar:JAXP_ROOT/xsltc.jar SOURCE_FILENAME ARGUMENTS
   ```

   以上で環境の構成が終了し、アプリケーションの開発を行える状況となります。

▼ 作業の開始

サンプルアプリケーションのこの段階では、コマンド行引数を解析して CrnpClient オブジェクトの構築を行うメインメソッドを使用し、CrnpClient という基本的なクラスを作成します。このオブジェクトは、コマンド行引数をこのクラスに渡し、ユーザーがアプリケーションを終了するのを待って CrnpClient で shutdown を呼び出し、その後終了します。

CrnpClient クラスのコンストラクタは、以下の作業を実行する必要があります。

- オブジェクトを処理する XML を設定する
- イベントコールバックを待機するスレッドを作成する
- CRNP サーバーと通信し、イベントコールバックを受け取る登録をする

● 上記のロジックを実装する **Java** コードを作成します。

次の例は、CrnpClient クラスのスケルトンコードを示しています。コンストラクタ内で参照される 4 つのヘルパーメソッドと停止メソッドの実装はあとで示します。ここでは、ユーザーが必要とするパッケージをすべてインポートするコードを示しています。

```
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import org.w3c.dom.*;

import java.net.*;
import java.io.*;
import java.util.*;


class CrnpClient
{
        public static void main(String []args)
        {
                InetAddress regIp = null;
                int regPort = 0, localPort = 0;

                try {
                        regIp = InetAddress.getByName(args[0]);
                        regPort = (new Integer(args[1])).intValue();
                        localPort = (new Integer(args[2])).intValue();
                } catch (UnknownHostException e) {
                        System.out.println(e);
                        System.exit(1);
                }

                CrnpClient client = new CrnpClient(regIp, regPort, localPort,
                    args);
                System.out.println("Hit return to terminate demo...");
                try {
```

```
                System.in.read();
        } catch (IOException e) {
                System.out.println(e.toString());
        }
        client.shutdown();
        System.exit(0);
}

public CrnpClient(InetAddress regIpIn, int regPortIn, int localPortIn,
    String []clArgs)
{
        try {
                regIp = regIpIn;
                regPort = regPortIn;
                localPort = localPortIn;
                regs = clArgs;

                setupXmlProcessing();
                createEvtRecepThr();
                registerCallbacks();

        } catch (Exception e) {
                System.out.println(e.toString());
                System.exit(1);
        }
}

public void shutdown()
{
        try {
                unregister();
        } catch (Exception e) {
                System.out.println(e);
                System.exit(1);
        }
}

private InetAddress regIp;
private int regPort;
private EventReceptionThread evtThr;
private String regs[];

public int localPort;
public DocumentBuilderFactory dbf;
}
```

メンバー変数についての詳細は後述します。

▼ コマンド行引数の解析

- コマンド行引数の解析方法については、付録 **G** 内のコードを参照してください。

▼ イベント受信スレッドの定義

イベント受信はコード内で個別のスレッドで行われるようにする必要があります。これは、イベントスレッドがイベントコールバックを待機している間アプリケーションが継続してほかの作業を行えるようにするためです。

注 – XML の設定については後述します。

---

1. コード内で、**`ServerSocket`** を作成してソケットにイベントが到着するのを待機する **`EventReceptionThread`** という **`Thread`** サブクラスを定義します。

   サンプルコードのこの部分では、イベントの読み取りもイベントの処理も行われません。イベントの読み取りと処理については後述します。
   `EventReceptionThread` は、ワイルドカード IP アドレス上に `ServerSocket` を作成します。`EventReceptionThread` は、`CrnpClient` オブジェクトにイベントを送信して処理できるように、`CrnpClient` オブジェクトに対する参照も維持します。

```
class EventReceptionThread extends Thread
{
        public EventReceptionThread(CrnpClient clientIn) throws IOException
        {
                client = clientIn;
                listeningSock = new ServerSocket(client.localPort, 50,
                    InetAddress.getLocalHost());
        }

        public void run()
        {
                try {
                        DocumentBuilder db = client.dbf.newDocumentBuilder();
                        db.setErrorHandler(new DefaultHandler());

                        while(true) {
                                Socket sock = listeningSock.accept();
                                // ソケットストリームからイベントを作成し、処理する。
                                sock.close();
                        }
                        // 到達不能

                } catch (Exception e) {
                        System.out.println(e);
                        System.exit(1);
                }
        }

        /* プライベートメンバー変数 */
        private ServerSocket listeningSock;
        private CrnpClient client;
}
```

2. 以上で、**EventReceptionThread** クラスがどのように動作するか確認ができました。次は、**createEvtRecepThr** オブジェクトを構築します。

```
private void createEvtRecepThr() throws Exception
{
        evtThr = new EventReceptionThread(this);
        evtThr.start();
}
```

▼ コールバックの登録と登録解除

登録は以下の作業によって行います。

- 登録用の IP アドレスとポートに対して基本的な TCP ソケットを開く
- XML 登録メッセージを作成する
- ソケット上で XML 登録メッセージを送信する
- ソケットから XML 応答メッセージを読み取る
- ソケットを閉じる

1. 上記のロジックを実装する **Java** コードを作成します。

   以下の例は、`CrnpClient` クラスの `registerCallbacks` メソッド (`CrnpClient` コンストラクタによって呼び出される) の実装を示しています。`createRegistrationString()` と `readRegistrationReply()` の呼び出しの詳細は後述します。

   `regIp` と `regPort` は、コンストラクタによって設定されるオブジェクトメンバーです。

```
private void registerCallbacks() throws Exception
{
        Socket sock = new Socket(regIp, regPort);
        String xmlStr = createRegistrationString();
        PrintStream ps = new
                PrintStream(sock.getOutputStream());
        ps.print(xmlStr);
        readRegistrationReply(sock.getInputStream();
        sock.close();
}
```

2. **unregister** メソッドを実装します。このメソッドは、**CrnpClient** の **shutdown** メソッドによって呼び出されます。**createUnregistrationString** の実装の詳細は後述します。

```
private void unregister() throws Exception
{
        Socket sock = new Socket(regIp, regPort);
        String xmlStr = createUnregistrationString();
        PrintStream ps = new PrintStream(sock.getOutputStream());
        ps.print(xmlStr);
        readRegistrationReply(sock.getInputStream());
        sock.close();
}
```

▼ XML の生成

以上で、アプリケーション構造の設定と、通信用のコードの作成が終了しました。次は、XML の生成と解析を行うコードを作成します。初めに、SC_CALLBACK_REG XML 登録メッセージを生成するコードを作成します。

SC_CALLBACK_REG メッセージは、登録のタイプ (ADD_CLIENT、REMOVE_CLIENT、 ADD_EVENTS、または REMOVE_EVENTS)、コールバックポート、および要求するイベントの一覧から構成されます。各イベントはクラスとサブクラスから構成され、名前と値のペアリストが続きます。

この例のこの段階では、登録タイプ、コールバックポート、および登録イベントの一覧を格納する CallbackReg クラスを作成します。このクラスは、それ自体を SC_CALLBACK_REG XML メッセージにシリアル化することもできます。

このクラスには、クラスメンバーから SC_CALLBACK_REG XML メッセージ文字列を作成する convertToXml という興味深いメソッドがあります。このメソッドを使用したコードの詳細は、http://java.sun.com/xml/jaxp/index.html の JAXP ドキュメントに記載されています。

次に、Event クラスの実装を示します。CallbackReg クラスは、イベントを 1 つ保存してそのイベントを XML Element に変換できる Event クラスを使用します。

1. 上記のロジックを実装する **Java** コードを作成します。

```
class CallbackReg
{
        public static final int ADD_CLIENT = 0;
        public static final int ADD_EVENTS = 1;
        public static final int REMOVE_EVENTS = 2;
        public static final int REMOVE_CLIENT = 3;

        public CallbackReg()
        {
                port = null;
                regType = null;
                regEvents = new Vector();
        }

        public void setPort(String portIn)
        {
                port = portIn;
        }

        public void setRegType(int regTypeIn)
        {
                switch (regTypeIn) {
                case ADD_CLIENT:
                        regType = "ADD_CLIENT";
                        break;
                case ADD_EVENTS:
                        regType = "ADD_EVENTS";
                        break;
```

```
        case REMOVE_CLIENT:
                regType = "REMOVE_CLIENT";
                break;
        case REMOVE_EVENTS:
                regType = "REMOVE_EVENTS";
                break;
        default:
                System.out.println("Error, invalid regType " +
                    regTypeIn);
                regType = "ADD_CLIENT";
                break;
        }
}

public void addRegEvent(Event regEvent)
{
        regEvents.add(regEvent);
}

public String convertToXml()
{
        Document document = null;
        DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();
        try {
                DocumentBuilder builder = factory.newDocumentBuilder();
                document = builder.newDocument();
        } catch (ParserConfigurationException pce) {
                // 指定されたオプションを持つパーサーを構築できない。
                pce.printStackTrace();
                System.exit(1);
        }

        // root 要素を作成する。
        Element root = (Element) document.createElement(
            "SC_CALLBACK_REG");

        // 属性を追加する。
        root.setAttribute("VERSION", "1.0");
        root.setAttribute("PORT", port);
        root.setAttribute("regType", regType);

        // イベントを追加する。
        for (int i = 0; i < regEvents.size(); i++) {
                Event tempEvent = (Event)
                    (regEvents.elementAt(i));
                root.appendChild(tempEvent.createXmlElement(
                    document));
        }
        document.appendChild(root);

        // 全体を文字列に変換する。
        DOMSource domSource = new DOMSource(document);
        StringWriter strWrite = new StringWriter();
        StreamResult streamResult = new StreamResult(strWrite);
```

```
                TransformerFactory tf = TransformerFactory.newInstance();
                try {
                        Transformer transformer = tf.newTransformer();
                        transformer.transform(domSource, streamResult);
                } catch (TransformerException e) {
                        System.out.println(e.toString());
                        return ("");
                }
                return (strWrite.toString());
        }

        private String port;
        private String regType;
        private Vector regEvents;
}
```

2. **Event** クラスと **NVPair** クラスを実装します。

   CallbackReg クラスは、NVPair クラスを使用する Event クラスを使用します。

```
class Event
{

        public Event()
        {
                regClass = regSubclass = null;
                nvpairs = new Vector();
        }

        public void setClass(String classIn)
        {
                regClass = classIn;
        }

        public void setSubclass(String subclassIn)
        {
                regSubclass = subclassIn;
        }

        public void addNvpair(NVPair nvpair)
        {
                nvpairs.add(nvpair);
        }

        public Element createXmlElement(Document doc)
        {
                Element event = (Element)
                    doc.createElement("SC_EVENT_REG");
                event.setAttribute("CLASS", regClass);
                if (regSubclass != null) {
                        event.setAttribute("SUBCLASS", regSubclass);
                }
                for (int i = 0; i < nvpairs.size(); i++) {
                          NVPair tempNv = (NVPair)
                              (nvpairs.elementAt(i));
```

```
                        event.appendChild(tempNv.createXmlElement(
                            doc));
                }
                return (event);
        }

        private String regClass, regSubclass;
        private Vector nvpairs;
}

class NVPair
{
        public NVPair()
        {
                name = value = null;
        }

        public void setName(String nameIn)
        {
                name = nameIn;
        }

        public void setValue(String valueIn)
        {
                value = valueIn;
        }

        public Element createXmlElement(Document doc)
        {
                Element nvpair = (Element)
                    doc.createElement("NVPAIR");
                Element eName = doc.createElement("NAME");
                Node nameData = doc.createCDATASection(name);
                eName.appendChild(nameData);
                nvpair.appendChild(eName);
                Element eValue = doc.createElement("VALUE");
                Node valueData = doc.createCDATASection(value);
                eValue.appendChild(valueData);
                nvpair.appendChild(eValue);

                return (nvpair);
        }

        private String name, value;
}
```

▼ 登録メッセージと登録解除メッセージの作成

XML メッセージを生成するヘルパークラスの作成が終了したところで、次は `createRegistrationString` メソッドを実装します。このメソッドは、`registerCallbacks` メソッド (詳細は 226 ページの「コールバックの登録と登録解除」) によって呼び出されます。

createRegistrationString は、CallbackReg オブジェクトを構築し、その登録タイプとポートを設定します。続いて、createRegistrationString は、createAllEvent、createMembershipEvent、 createRgEvent、および createREvent ヘルパーメソッドを使用して各種のイベントを構築します。各イベントは、CallbackReg オブジェクトが作成されたあとでこのオブジェクトに追加されます。最後に、createRegistrationString は CallbackReg オブジェクト上で convertToXml メソッドを呼び出し、String 形式の XML メッセージを取得します。

regs メンバー変数は、ユーザーがアプリケーションに指定するコマンド行引数を格納します。5 つ目以降の引数は、アプリケーションが登録を行うイベントを指定します。4 つ目の引数は登録のタイプを指定しますが、この例では無視されています。付録 G に挙げられている完全なコードでは、この 4 つ目の引数の使用方法も示されています。

1. 上記のロジックを実装する **Java** コードを作成します。

```
private String createRegistrationString() throws Exception
{
        CallbackReg cbReg = new CallbackReg();
        cbReg.setPort("" + localPort);

        cbReg.setRegType(CallbackReg.ADD_CLIENT);

        // イベントを追加する。
        for (int i = 4; i < regs.length; i++) {
                if (regs[i].equals("M")) {
                        cbReg.addRegEvent(
                            createMembershipEvent());
                } else if (regs[i].equals("A")) {
                        cbReg.addRegEvent(
                            createAllEvent());
                } else if (regs[i].substring(0,2).equals("RG")) {
                        cbReg.addRegEvent(createRgEvent(
                            regs[i].substring(3)));
                } else if (regs[i].substring(0,1).equals("R")) {
                        cbReg.addRegEvent(createREvent(
                            regs[i].substring(2)));
                }
        }

        String xmlStr = cbReg.convertToXml();
        return (xmlStr);
}

private Event createAllEvent()
{
        Event allEvent = new Event();
        allEvent.setClass("EC_Cluster");
        return (allEvent);
}

private Event createMembershipEvent()
```

```
{
        Event membershipEvent = new Event();
        membershipEvent.setClass("EC_Cluster");
        membershipEvent.setSubclass("ESC_cluster_membership");
        return (membershipEvent);
}

private Event createRgEvent(String rgname)
{
        Event rgStateEvent = new Event();
        rgStateEvent.setClass("EC_Cluster");
        rgStateEvent.setSubclass("ESC_cluster_rg_state");

        NVPair rgNvpair = new NVPair();
        rgNvpair.setName("rg_name");
        rgNvpair.setValue(rgname);
        rgStateEvent.addNvpair(rgNvpair);

        return (rgStateEvent);
}

private Event createREvent(String rname)
{
        Event rStateEvent = new Event();
        rStateEvent.setClass("EC_Cluster");
        rStateEvent.setSubclass("ESC_cluster_r_state");

        NVPair rNvpair = new NVPair();
        rNvpair.setName("r_name");
        rNvpair.setValue(rname);
        rStateEvent.addNvpair(rNvpair);

        return (rStateEvent);
}
```

2. 登録解除文字列を作成します。

   イベントを指定する必要がない分、登録解除文字列の作成は登録文字列の作成よりも簡単です。

```
private String createUnregistrationString() throws Exception
{
        CallbackReg cbReg = new CallbackReg();
        cbReg.setPort("" + localPort);
        cbReg.setRegType(CallbackReg.REMOVE_CLIENT);
        String xmlStr = cbReg.convertToXml();
        return (xmlStr);
}
```

▼ XML パーサーの設定

以上で、アプリケーションの通信用コードと XML 生成コードの生成が終わります。最後のステップとして、登録応答とイベントコールバックの解析と処理を行います。`CrnpClient` コンストラクタは `setupXmlProcessing` メソッドを呼び出します。

このメソッドは、 DocumentBuilderFactory オブジェクトを作成し、そのオブジェクトに各種の解析プロパティを設定します。このメソッドの詳細は、 http://java.sun.com/xml/jaxp/index.html の JAXP ドキュメントに記載されています。

- 上記のロジックを実装する **Java** コードを作成します。

```
private void setupXmlProcessing() throws Exception
{
        dbf = DocumentBuilderFactory.newInstance();

        // わざわざ検証を行う必要はない。
        dbf.setValidating(false);
        dbf.setExpandEntityReferences(false);

        // コメントと空白文字は無視したい。
        dbf.setIgnoringComments(true);
        dbf.setIgnoringElementContentWhitespace(true);

        // CDATA セクションを TEXT ノードに結合する。
        dbf.setCoalescing(true);
}
```

▼ 登録応答の解析

登録メッセージまたは登録解除メッセージに応答して CRNP サーバーが送信する SC_REPLY XML メッセージを解析するには、 RegReply ヘルパークラスが必要です。このクラスは、XML ドキュメントから構築できます。このクラスは、ステータスコードとステータスメッセージのアクセッサを提供します。サーバーからの XML ストリームを解析するには、新しい XML ドキュメントを作成してそのドキュメントの解析メソッドを使用する必要があります (このメソッドの詳細は http://java.sun.com/xml/jaxp/index.html の JAXP ドキュメントを参照)。

1. 上記のロジックを実装する **Java** コードを作成します。

   readRegistrationReply メソッドは、新しい RegReply クラスを使用します。

```
private void readRegistrationReply(InputStream stream) throws Exception
{
        // ドキュメントビルダーを作成する。
        DocumentBuilder db = dbf.newDocumentBuilder();
        db.setErrorHandler(new DefaultHandler());

        // 入力ファイルを解析する。
        Document doc = db.parse(stream);

        RegReply reply = new RegReply(doc);
        reply.print(System.out);
}
```

2. **RegReply** クラスを実装します。

retrieveValues メソッドは XML ドキュメント内の DOM ツリーを回り、ステータスコードとステータスメッセージを抽出します。詳細は、http://java.sun.com/xml/jaxp/index.html の JAXP ドキュメントに記載されています。

```
class RegReply
{
        public RegReply(Document doc)
        {
                retrieveValues(doc);
        }

        public String getStatusCode()
        {
                return (statusCode);
        }

        public String getStatusMsg()
        {
                return (statusMsg);
        }
        public void print(PrintStream out)
        {
                out.println(statusCode + ": " +
                    (statusMsg != null ? statusMsg : ""));
        }

        private void retrieveValues(Document doc)
        {
                Node n;
                NodeList nl;
                String nodeName;

                // SC_REPLY 要素を見つける。
                nl = doc.getElementsByTagName("SC_REPLY");
                if (nl.getLength() != 1) {
                        System.out.println("Error in parsing: can't find "
                            + "SC_REPLY node.");
                        return;
                }

                n = nl.item(0);

                // statusCode 属性の値を取得する。
                statusCode = ((Element)n).getAttribute("STATUS_CODE");

                // SC_STATUS_MSG 要素を検出する。
                nl = ((Element)n).getElementsByTagName("SC_STATUS_MSG");
                if (nl.getLength() != 1) {
                        System.out.println("Error in parsing: can't find "
                            + "SC_STATUS_MSG node.");
                        return;
                }
                // TEXT セクションを取得する (存在する場合)。
                n = nl.item(0).getFirstChild();
```

```
            if (n == null || n.getNodeType() != Node.TEXT_NODE) {
                    // 1 つも存在しなくてもエラーではないため、そのまま戻る。
                    return;
            }

            // 値を取得する。
            statusMsg = n.getNodeValue();
        }

        private String statusCode;
        private String statusMsg;
}
```

▼ コールバックイベントの解析

最後のステップは、実際のコールバックイベントの解析と処理です。この作業をスムーズに行うため、227 ページの「XML の生成」で作成した Event クラスを変更します。このクラスを使用して XML ドキュメントから Event を構築し、XML Element を作成できます。この変更は、XML ドキュメントを受け付ける別のコンストラクタ、retrieveValues メソッド、2 つの新たなメンバー変数 (vendor と publisher) 、全フィールドのアクセッサメソッド、および出力メソッドを必要とします。

1. 上記のロジックを実装する **Java** コードを作成します。

   このコードは、233 ページの「登録応答の解析」で説明しているRegReply クラスのコードに似ていることに注目してください。

```
public Event(Document doc)
{
        nvpairs = new Vector();
        retrieveValues(doc);
}
public void print(PrintStream out)
{
        out.println("\tCLASS=" + regClass);
        out.println("\tSUBCLASS=" + regSubclass);
        out.println("\tVENDOR=" + vendor);
        out.println("\tPUBLISHER=" + publisher);
        for (int i = 0; i < nvpairs.size(); i++) {
                NVPair tempNv = (NVPair)
                    (nvpairs.elementAt(i));
                out.print("\t\t");
                tempNv.print(out);
        }
}

private void retrieveValues(Document doc)
{
        Node n;
        NodeList nl;
        String nodeName;
```

```
                // SC_EVENT 要素を検出する。
                nl = doc.getElementsByTagName("SC_EVENT");
                if (nl.getLength() != 1) {
                        System.out.println("Error in parsing: can't find
"
                            + "SC_EVENT node.");
                        return;
                }

                n = nl.item(0);

                //
                // CLASS、SUBCLASS、VENDOR、および PUBLISHER
                // 属性の値を取得する。
                //
                regClass = ((Element)n).getAttribute("CLASS");
                regSubclass = ((Element)n).getAttribute("SUBCLASS");
                publisher = ((Element)n).getAttribute("PUBLISHER");
                vendor = ((Element)n).getAttribute("VENDOR");

                // すべての nv ペアを取得する。
                for (Node child = n.getFirstChild(); child != null;
                    child = child.getNextSibling())
                {
                       nvpairs.add(new NVPair((Element)child));
                }
        }

        public String getRegClass()
        {
                return (regClass);
        }

        public String getSubclass()
        {
                return (regSubclass);
        }

        public String getVendor()
        {
                return (vendor);
        }

        public String getPublisher()
        {
                return (publisher);
        }

        public Vector getNvpairs()
        {
                return (nvpairs);
        }

        private String vendor, publisher;
```

2. **XML** 解析をサポートする、**`NVPair`** クラスのコンストラクタとメソッドを別途実装します。

   手順 1 で Event クラスに変更を加えたため、NVPair クラスにも類似した変更を加える必要があります。

```
        public NVPair(Element elem)
        {
                retrieveValues(elem);
        }
        public void print(PrintStream out)
        {
                out.println("NAME=" + name + " VALUE=" + value);
        }
        private void retrieveValues(Element elem)
        {
                Node n;
                NodeList nl;
                String nodeName;

                // NAME 要素を検出する。
                nl = elem.getElementsByTagName("NAME");
                if (nl.getLength() != 1) {
                        System.out.println("Error in parsing: can't find
"
                            + "NAME node.");
                        return;
                }
                // TEXT セクションを取得する。
                n = nl.item(0).getFirstChild();
                if (n == null || n.getNodeType() != Node.TEXT_NODE) {
                        System.out.println("Error in parsing: can't find
"
                            + "TEXT section.");
                        return;
                }

                // 値を取得する。
                name = n.getNodeValue();

                // ここで値要素を取得する。
                nl = elem.getElementsByTagName("VALUE");
                if (nl.getLength() != 1) {
                        System.out.println("Error in parsing: can't find
"
                            + "VALUE node.");
                        return;
                }
                // TEXT セクションを取得する。
                n = nl.item(0).getFirstChild();
                if (n == null || n.getNodeType() != Node.TEXT_NODE) {
                System.out.println("Error in parsing: can't find "
                            + "TEXT section.");
                        return;
                }
```

```
                // 値を取得する。
                value = n.getNodeValue();
                }

        public String getName()
        {
                return (name);
        }

        public String getValue()
        {
                return (value);
        }
}
```

3. **EventReceptionThread** でイベントコールバックを待機する **while** ループを実装します **(EventReceptionThread** については **225** ページの「イベント受信スレッドの定義」を参照**)**。

```
while(true) {
                Socket sock = listeningSock.accept();
                Document doc = db.parse(sock.getInputStream());
                Event event = new Event(doc);
                client.processEvent(event);
                sock.close();
        }
```

▼ アプリケーションの実行

- アプリケーションを実行します。

  # **java CrnpClient** *crnpHost crnpPort localPort* **...**

  完全な CrnpClient アプリケーションコードは、付録 G に示されています。

付録 A

標準プロパティ

この付録では、標準リソースタイプ、リソースグループ、リソースプロパティについて説明します。また、システム定義プロパティの変更および拡張プロパティの作成に使用するリソースプロパティ属性についても説明します。

この付録は、次のような節から構成されています。

注 – `True` や `False` などのプロパティ値は、大文字と小文字は区別されません。

リソースタイププロパティ

以下の表に、Sun Cluster によって定義されるリソースタイププロパティを示します。プロパティ値は、次のように分類されます (「カテゴリ」列)。

- 必須 — Resource Type Registration (RTR) ファイル内に利用値を必要とするプロパティです。値がない場合は、プロパティが属するオブジェクトを作成できません。空白文字または空の文字列を値として指定することはできません。
- 条件付き — このプロパティが存在するためには、RTR ファイル内で宣言する必要があります。宣言がない場合、RGM はこのプロパティを作成しません。したがって、管理ユーティリティで利用できません。 空白文字または空の文字列を値として指定できます。プロパティが RTR ファイル内で宣言されており、値が指定されていない場合には、RGM はデフォルト値を使用します。

- 条件付き / 明示 — このプロパティが存在するためには、明示的に値を指定して、RTR ファイル内で宣言する必要があります。宣言がない場合、RGM はこのプロパティを作成しません。したがって、管理ユーティリティで利用できません。 空白文字または空の文字列を値として指定することはできません。
- 任意 — プロパティを RTR ファイル内で宣言できます。宣言しない場合は、RGM はこのプロパティを作成し、デフォルト値を使用します。プロパティが RTR ファイル内で宣言されており、値が指定されていない場合は、RGM は、プロパティが RTR ファイル内で宣言されないときのデフォルト値と同じ値を使用します。

リソースタイププロパティは、Installed_nodes を除き、管理ユーティリティによって更新することができません。Installed_nodes は、RTR ファイル内で宣言できないため、管理者が設定する必要があります。

表 **A–1** リソースタイププロパティ

プロパティ名	説明	更新の可否	カテゴリ
Allow_hosts (文字配列型)	クラスタ再構成イベントを受信するために cl_apid デーモンを使用して登録を行うことができるクライアントの設定を制御します。このプロパティは、通常 ipaddress/masklength の形式で登録可能なクライアントのサブネットを定義します。たとえば 129.99.77.0/24 の設定では、サブネット 129.99.77 上のクライアントがイベント登録できます。また、192.9.84.231/32 では、クライアント 192.9.84.231 だけがイベント登録できます。このプロパティは CRNP にセキュリティを提供します。cl_apid デーモンの詳細は、SUNW.Event(5) のマニュアルページを参照してください。 さらに、以下の特殊キーワードが認識されます。LOCAL は、クラスタの直接接続されたサブネット上に存在する全クライアントです。ALL では、すべてのクライアントが登録可能です。Allow_hosts と Deny_hosts プロパティの両方で 1 つのエントリと一致するクライアントの場合、この実装で登録することはできません。 デフォルトは LOCAL です。	不可	任意
API_version (整数)	このリソースタイプの実装が使用するリソース管理 API のバージョン。 SC 3.1 のデフォルトは 2 です。	不可	任意

表 **A–1** リソースタイププロパティ　(続き)

プロパティ名	説明	更新の可否	カテゴリ
Boot (文字列)	任意のコールバックメソッドです。RGM がノード上で呼び出すプログラムのパスを指定します。このプログラムは、このリソース型が管理対象になっているとき、クラスタの結合または再結合を行います。このメソッドは、Init メソッドと同様に、このタイプのリソースに対し、初期化アクションを行う必要があります。	不可	条件付き / 明示
Client_retry_count (整数型)	外部クライアントとの通信で行われる cl_apid デーモンの試行回数を制御します。Client_retry_count の試行回数内に応答しなかったクライアントはタイムアウトになり、クラスタ再構成イベントの受信資格を持つクライアントの登録リストから削除されます。続いてクライアントは、クラスタ再構成イベントを受け取ることができる登録済みクライアントのリストから削除されます。削除されたクライアントは、再びイベントを受信するために再登録する必要があります。実装による再試行の頻度についての詳細は、Client_retry_interval プロパティの説明を参照してください。cl_apid デーモンの詳細は、SUNW.Event(5) のマニュアルページを参照してください。 デフォルト値は 3 です。	可	任意
Client_retry_interval (整数型)	cl_apid デーモンが 応答のない外部クライアントとの通信を行う時間の長さを秒単位で指定します。この時間内に、最大 Client_retry_count 回のクライアント接続が試行されます。cl_apid デーモンの詳細は、SUNW.Event(5) のマニュアルページを参照してください。 デフォルトは 1800 です。	可	任意
Client_timeout (整数型)	cl_apid デーモンが外部クライアントとの通信で使用するタイムアウト値を秒単位で指定します。しかし、cl_apid デーモンは、調整可能な回数だけクライアントとの通信を試みます。このプロパティの調整方法の詳細は、Client_retry_count プロパティと Client_retry_interval プロパティの説明を参照してください。cl_apid デーモンの詳細は、SUNW.Event(5) のマニュアルページを参照してください。 デフォルトは 60 です。	可	任意

表 **A–1** リソースタイププロパティ　(続き)

プロパティ名	説明	更新の可否	カテゴリ
Deny_hosts (文字配列型)	クラスタ再構成イベントの受信候補として登録できないクライアントの設定を制御します。アクセスを確定するために、このプロパティの設定は Allow_hosts リストの設定に優先します。このプロパティの形式は、Allow_hosts プロパティで定義された形式と同じです。このプロパティは CRNP にセキュリティを提供します。 デフォルトは NULL です。	可	任意
Failover (ブール値)	True は、複数のノード上で同時にオンラインになることのできる任意のグループで、このタイプのリソースを構成できないことを示します。デフォルトは、False です。	N	任意
Fini (文字列)	任意のコールバックメソッドです。この型のリソースを RGM 管理の対象外にするとき RGM によって呼び出されるプログラムのパスです。	不可	条件付き / 明示
Init (文字列)	任意のコールバックメソッドです。この型のリソースを RGM 管理対象にするとき RGM によって呼び出されるプログラムのパスです。	不可	条件付き / 明示
Init_nodes (列挙)	値には、RG_primaries (リソースをマスターできるノードだけ)、または RT_installed_nodes (リソースタイプがインストールされるすべてのノード) を指定できます。RGM が Init、Fini、 Boot、Validate メソッドをコールするノードを示します。 デフォルト値は、RG_primaries です。	不可	任意
Installed_nodes (文字配列)	リソースタイプの実行が許可されるクラスタノード名のリスト。このプロパティは RGM によって自動的に作成されます。クラスタ管理者は値を設定できます。RTR ファイル内には宣言できません。 デフォルトは、すべてのクラスタノードです。	可	クラスタ管理者による構成が可能です。

表 **A–1** リソースタイププロパティ　(続き)

プロパティ名	説明	更新の可否	カテゴリ
`Max_clients` (整数型)	クラスタイベントの通知を受けるように `cl_apid` デーモンを使用して登録できるクライアントの最大数を制御します。この数を超えるクライアントによるイベント登録の試行はアプリケーションによって拒否されます。個々のクライアント登録にはクラスタ上のリソースが使用されるため、このプロパティの値を調整することで、外部クライアントによるクラスタ上のリソース使用を制御することができます。`cl_apid` デーモンの詳細は、`SUNW.Event(5)` のマニュアルページを参照してください。 デフォルトは 1000 です。	可	任意
`Monitor_check` (文字列)	任意のコールバックメソッド。障害モニターの要求によってこのリソース型のフェイルオーバーを実行する前に、RGM によって呼び出されるプログラムのパスです。	不可	条件付き / 明示
`Monitor_start` (文字列)	任意のコールバックメソッド。この型のリソースの障害モニターを起動するために RGM によって呼び出されるプログラムのパスです。	不可	条件付き / 明示
`Monitor_stop` (文字列)	`Monitor_start` が設定されている場合、必須のコールバックメソッドになります。この型のリソースの障害モニターを停止するために RGM によって呼び出されるプログラムのパスです。	不可	条件付き / 明示
各クラスタノード上の `Num_resource_ restarts` (整数)。	このプロパティは、RGM によって、このノード上のこのリソースに対して過去 *n* 秒間 (*n* はリソースの `Retry_interval` プロパティの値) に実行された `scha_control RESTART` 呼び出しの回数に設定されます。リソースタイプが `Retry_interval` プロパティを宣言していない場合、この型のリソースは `Num_resource_restarts` プロパティを使用できません。	不可	照会のみ
`Pkglist` (文字配列)	リソースタイプのインストールに含まれている任意のパッケージリスト。	不可	条件付き / 明示

表 **A–1** リソースタイププロパティ　(続き)

プロパティ名	説明	更新の可否	カテゴリ
`Postnet_stop` (文字列)	任意のコールバックメソッド。このリソースタイプがネットワークアドレスリソース (`Network_resources_used`) に依存している場合、このネットワークアドレスリソースの `Stop` メソッドの呼び出し後に RGM によって呼び出されるプログラムのパスを指定します。このメソッドは、ネットワークインタフェースの停止設定に続いて、必要な `Stop` アクションを行います。	不可	条件付き / 明示
`Prenet_start` (文字列)	任意のコールバックメソッド。このリソースタイプがネットワークアドレスリソース (`Network_resources_used`) に依存している場合、このネットワークアドレスリソースの `Start` メソッドの呼び出し前に RGM によって呼び出されるプログラムのパスを指定します。ネットワークインタフェースが起動に構成される前に必要な `Start` アクションを行う必要があります。	不可	条件付き / 明示

`Resource_type` (文字列)	リソースタイプの名前。現在登録されているリソースタイプ名を表示するには、次のコマンドを使用します。
	`scrgadm` `-p`
	Sun Cluster 3.1 以降、リソースタイプ名は次の形式をとります。
	`vendor_id.resource_type:version`
	リソースタイプ名は、RTR ファイル内に指定された 3 つのプロパティ *Vendor_id*、*Resource_type*、*RT_version* で構成されます。`scrgadm` コマンドでは、区切り文字としてピリオドとコロンを使用します。リソースタイプ名の最後の部分、`RT_version` には、`RT_version` プロパティと同じ値が入ります。重複を防ぐため、*Vendor_id* には、リソースタイプの作成元の会社のストックシンボルを使用することをお勧めします。Sun Cluster 3.1 以前に作成されたリソースタイプ名は次の形式をとります。
	`vendor_id.resource_type`
	デフォルトは空の文字列です。
		不可	必須

表 **A–1** リソースタイププロパティ　(続き)

プロパティ名	説明	更新の可否	カテゴリ
`RT_basedir` (文字列)	コールバックメソッドの相対パスのを補完するディレクトリパスです。このパスは、リソースタイプパッケージのインストール場所に設定します。スラッシュ (/) で開始する完全なパスを指定する必要があります。すべてのメソッドパス名が絶対パスの場合は、指定しなくてもかまいません。	不可	必須 (絶対パスでないメソッドパスがある場合)
`RT_description` (文字列)	リソース型の簡単な説明です。 デフォルトは空の文字列です。	不可	条件付き
`RT_version` (文字列)	Sun Cluster 3.1 以降、このリソースタイプの実装に必要なバージョンを指定します。`RT_version` は、完全なリソースタイプ名の末尾の部分です。	不可	条件付き / 明示
`Single_instance` (ブール値)	`True` の場合、このリソースタイプはクラスタ内に 1 つだけ存在できます。RGM は、同時に 1 つのこのリソースタイプだけに、クラスタ全体に渡っての実行を許可します。 デフォルト値は、`False` です。	不可	任意
`Start` (文字列)	コールバックメソッド。 この型のリソースを起動するために RGM によって呼び出されるプログラムのパスです。	不可	RTR ファイルで `Prenet_start` メソッドが宣言されていないかぎり必須
`Stop` (文字列)	コールバックメソッド。 この型のリソースを停止するために RGM によって呼び出されるプログラムのパスです。	不可	RTR ファイルで `Postnet_stop` メソッドが宣言されていないかぎり必須
`Update` (文字列)	任意のコールバックメソッド。この型の実行中のリソースのプロパティが変更されたとき RGM によって呼び出されるプログラムのパスです。	不可	条件付き / 明示
`Validate` (文字列)	任意のコールバックメソッドです。この型のリソースのプロパティ値を検査するために呼び出されるプログラムのパスを指定します。	不可	条件付き / 明示
`Vendor_ID` (文字列)	`Resource_type` を参照してください。	不可	条件付き

リソースプロパティ

Sun Cluster で定義されるリソースプロパティについては表 A–2 を参照してください。プロパティ値は、次のように分類されます (「カテゴリ」列)。

- 必須 — 管理者は、管理ユーティリィティでリソースを作成するときに、必ず値を指定する必要があります。
- 任意 — 管理者がリソースグループの作成時に値を指定しない場合、システムがデフォルト値を提供します。
- 条件付き — プロパティが RTR ファイルで宣言されている場合にのみ、RGM がプロパティを作成します。宣言されていない場合プロパティは存在せず、システム管理者はこれを利用できません。RTR ファイルで宣言されている条件付きのプロパティは、デフォルト値が RTR ファイル内で指定されているかどうかによって、必須または任意になります。詳細については、各条件付きプロパティの説明を参照してください。
- 照会のみ— 管理ツールから直接設定できません。

表 A–2 の「更新の可否」列では、リソースプロパティが更新可能かどうか、更新可能な場合はいつ更新できるかを示しています。

`None` または `False`	不可 (None)
`True` または `Anytime`	任意の時点 (Anytime)
`At_creation`	リソースをクラスタに追加するとき
`When_disabled`	リソースが無効なとき

表 **A–2** リソースプロパティ

プロパティ名	説明	更新の可否	カテゴリ
`Affinity_timeout` (整数)	リソース内のサービスのクライアント IP アドレスからの接続は、この時間 (秒数) 内に同じサーバーノードに送信されます。 このプロパティは、`Load_balancing_policy` が `Lb_sticky` または `Lb_sticky_wild` の場合にかぎり有効です。さらに、`Weak_affinity` を False (デフォルト値) に設定する必要があります。 このプロパティは、スケーラブルサービス専用です。	任意の時点 (Anytime)	任意

表 **A–2** リソースプロパティ　(続き)

プロパティ名	説明	更新の可否	カテゴリ
`Cheap_probe_ interval` (整数)	リソースの即時障害検証の呼び出しの間隔 (秒数)。このプロパティは、RGM のみが作成でき、RTR ファイル内で宣言されている場合には、管理者はこのプロパティを利用できます。 デフォルト値が RTR ファイルで指定されている場合、このプロパティは任意です。リソースタイプファイル内に `Tunable` 属性が指定されていない場合、このプロパティの `Tunable` 値は `When_disabled` になります。 `Default` 属性が RTR ファイルのプロパティ宣言に指定されていない場合は、このプロパティは必須です。	無効時 (When_ disabled)	条件付き
拡張プロパティ	そのリソースのタイプの RTR ファイルで宣言される拡張プロパティ。リソースタイプの実装によって、これらのプロパティを定義します。拡張プロパティに設定可能な各属性については表 A–4 を参照してください。	特定のプロパティに依存	条件付き
`Failover_mode` (列挙)	設定可能な値は None、Soft、Hard です。リソース上の `Start`、`Stop`、または`Monitor_stop` メソッド呼び出しに失敗した場合、RGM がリソースグループを再配置するか、あるいはノードを終了するかを制御します。`None` は、RGM が単にリソース状態をメソッド失敗に設定し、オペレータの介入を待つことを示します。`Soft` は、RGM が、`Start` メソッドの失敗時にはリソースのグループを別のノードに再配置するが、`Stop` または `Monitor_stop` メソッドの失敗時にはリソースを `STOP_FAILED` 状態、リソースグループを `ERROR_STOP_FAILED` 状態にしてオペレータの介入を待つことを示します。`Stop` または `Monitor_stop` の失敗時には、`None` と `Soft` のどちらに設定していても同じ結果になります。`Hard` は、`Start` メソッドが失敗したときに、グループの再配置を行い、`Stop` または `Monitor_stop` メソッドが失敗したときに、クラスタノードを異常終了させることで、リソースの強制的な停止を行うことを示します。 デフォルトは `None` です。	任意の時点 (Anytime)	オプション

表 **A–2** リソースプロパティ　(続き)

プロパティ名	説明	更新の可否	カテゴリ
`Load_balancing_ policy` (文字列)	使用する負荷均衡ポリシーを定義する文字列。このプロパティは、スケーラブルサービス専用です。RTR ファイルに `Scalable` プロパティが宣言されている場合、RGM は自動的にこのプロパティを作成します。`Load_balancing_policy` は、次の値をとることができます。 `Lb_weighted` (デフォルト)。`Load_balancing_weights` プロパティで設定されているウエイトに従って、さまざまなノードに負荷が分散されます。`Lb_sticky`。スケーラブルサービスの指定のクライアント (クライアントの IP アドレスで識別される) は、常に同じクラスタノードに送信されます。 `Lb_sticky_wild`。指定のクライアント (クライアントの IP アドレスで識別される) はワイルドカードスティッキーサービスの IP アドレスに接続され、送信時に使用されるポート番号とは無関係に、常に同じクラスタノードに送信されます。 デフォルト値は、`Lb_weighted` です。	作成時 (At_ creation)	条件付き / 任意
`Load_balancing_weights` (文字配列)	このプロパティは、スケーラブルサービスに対してのみ使用します。RTR ファイルに `Scalable` プロパティが宣言されている場合、RGM は自動的にこのプロパティを作成します。 形式は、「*weight@node,weight@node*」になります。ここで、*weight* は、指定したノード (*node*) に対する負荷分散の相対的な割り当てを示す整数になります。ノードに分散される負荷の割合は、すべてのウエイトの合計でこのノードのウエイトを割った値になります。たとえば、`1@1,3@2` は、ノード 1 に負荷の 1/4 が割り当てられ、ノード 2 に負荷の 3/4 が割り当てられることを意味します。デフォルトの空の文字列 ("") は、一定の分散を指定します。明示的にウエイトを割り当てられていないノードのウエイトは、デフォルトで 1 になります。 `Tunable` 属性がリソースタイプファイルに指定されていない場合は、プロパティの `Tunable` 値は `Anytime` (任意の時点) になります。このプロパティを変更すると、新しい接続時にのみ分散が変更されます。 デフォルト値は、空の文字列 ("") です。	任意の時点 (Anytime)	条件付き / 任意

表 **A–2** リソースプロパティ　(続き)

プロパティ名	説明	更新の可否	カテゴリ
リソースタイプの各コールバックメソッドの *method*_timeout (整数)	RGM がメソッドの呼び出しに失敗したと判断するまでの時間 (秒)。 メソッド自身が RTR ファイルで宣言されている場合、デフォルトは、3,600 秒 (1 時間) です。	任意の時点 (Anytime)	条件付 / 任意
Monitored_ switch (列挙)	クラスタ管理者が管理ユーティリィティを使用してモニターを有効または無効にすると、RGM によって Enabled または Disabled に設定されます。Disabled に設定されると、再び有効に設定されるまで、モニターは Start メソッドを呼び出しません。リソースが、モニターのコールバックメソッドを持っていない場合は、このプロパティは存在しません。 デフォルトは Enabled です。	不可 (None)	照会のみ
Network_ resources_ used (文字配列)	リソースが使用する論理ホスト名または共有アドレスネットワークリソースのリスト。スケーラブルサービスの場合、このプロパティは別のリソースグループに存在する共有アドレスリソースを参照する必要があります。フェイルオーバーサービスの場合、このプロパティは同じリソースグループに存在する論理ホスト名または共有アドレスを参照します。RTR ファイルに Scalable プロパティが宣言されている場合、RGM は自動的にこのプロパティを作成します。 Scalable が RTR ファイルで宣言されていない場合、Network_resources_used は RTR ファイルで明示的に宣言されていないかぎり使用できません。 Tunable 属性がリソースタイプファイルに指定されていない場合は、プロパティの Tunable 値は、 At_creation (作成時) になります。	作成時 (At_ creation)	条件付き / 必須
On_off_switch (列挙)	クラスタ管理者が管理ユーティリィティを使用してリソースを有効または無効にすると、RGM によって Enabled または Disabled に設定されます。無効に設定されると、再び有効に設定されるまで、リソースはコールバックを呼び出しません。 デフォルトは、Disabled です。	不可 (None)	照会のみ

表 **A–2** リソースプロパティ　(続き)

プロパティ名	説明	更新の可否	カテゴリ
`Port_list` (文字配列)	サーバーが待機するポートの番号リストです。各ポート番号の末尾に、そのポートが使用しているプロトコルが追加されます (例:`Port_list=80/tcp`)。`Scalable` プロパティが RTR ファイルで宣言されている場合、RGM は自動的に `Port_list` を作成します。それ以外の場合、このプロパティは RTR ファイルで明示的に宣言されていないかぎり使用できません。 Apache 用にこのプロパティを設定する方法は、『*Sun Cluster 3.1 Data Service for Apache* ガイド』を参照してください。	作成時 (At_creation)	条件付き / 必須
`R_description` (文字列)	リソースの簡単な説明。 デフォルトは空の文字列です。	任意の時点 (Anytime)	任意
`Resource_name` (文字列)	リソースインスタンスの名前。この名前はクラスタ構成内で一意にする必要があります。リソースが作成されたあとで変更はできません。	不可 (None)	必須
`Resource_project_name` (文字列)	リソースに関連付けられた Solaris プロジェクト名。このプロパティは、CPU の共有、クラスタデータサービスのリソースプールといった Solaris のリソース管理機能に適用できます。RGM は、リソースをオンラインにすると、このプロジェクト名を持つ関連プロセスを起動します。このプロパティが指定されていない場合、プロジェクト名は、リソースが属しているリソースグループの `RG_project_name` プロパティから取得されます。`rg_properties`(5) を参照してください。どちらのプロパティも指定されなかった場合、RGM は事前定義済みのプロジェクト名 `default` を使用します。プロジェクトデータベース内に存在するプロジェクト名を指定する必要があります。また、`root` ユーザーは、このプロジェクトのメンバーとして構成されている必要があります。このプロパティは Solaris 9 以降でサポートされます。 注 – このプロパティへの変更を有効にするには、リソースを再起動する必要があります。 デフォルトは null です。	任意の時点 (Anytime)	任意

表 **A–2** リソースプロパティ　(続き)

プロパティ名	説明	更新の可否	カテゴリ
各クラスタノードの `Resource_state` (列挙)	RGM が判断した各クラスタノード上のリソースの状態。`Online`、`Offline`、`Stop_failed` 、`Start_failed`、`Monitor_failed`、`Online_not_monitored`、`Detached`。 このプロパティは、ユーザーは構成できません。	不可 (None)	照会のみ
`Retry_count` (整数)	リソースの起動に失敗した場合にモニターが再起動を試みる試行回数です。このプロパティは、RGM のみが作成でき、RTR ファイルで宣言されている場合は、管理者は利用できます。デフォルト値が RTR ファイルで指定されている場合は、このプロパティは任意です。 リソースタイプファイル内で `Tunable` 属性が指定されていない場合は、プロパティの `Tunable` 値は、`When_disabled` (無効化にするとき) になります。 RTR ファイルのプロパティ宣言内に `Default` 属性が指定されていない場合、このプロパティは必須です。	無効時 (When_disabled)	条件付き
`Retry_interval` (整数)	失敗したリソースを再起動するまでの秒数。リソースモニターは、`Retry_count` と共にこのプロパティを使用します。このプロパティは、RGM のみが作成でき、RTR ファイルで宣言されている場合は、管理者は利用できます。RTR ファイルにデフォルト値が指定されている場合、このプロパティは任意です。 リソースタイプファイル内に `Tunable` 属性が指定されていない場合、このプロパティの `Tunable` 値は `When_disabled` になります。 `Default` 属性が RTR ファイルのプロパティ宣言に指定されていない場合は、このプロパティは必須です。	無効時 (When_disabled)	条件付き

表 **A–2** リソースプロパティ　(続き)

プロパティ名	説明	更新の可否	カテゴリ
`Scalable` (ブール値)	リソースがスケーラブルかどうかを示します。このプロパティが RTR ファイルで宣言されている場合は、そのタイプのリソースに対して、RGM は、次のスケーラブルサービスプロパティを自動的に作成します。`Network_resources_used`、`Port_list`、`Load_balancing_policy`、`Load_balancing_weights`。これらのプロパティは、RTR ファイルで明示的に宣言されないかぎり、デフォルト値を持ちます。RTR ファイルで宣言されている場合、`Scalable` のデフォルトは `True` です。 このプロパティが RTR ファイルで宣言されている場合、`Tunable` 属性は、`At_creation` (作成時) に設定する必要があります。設定しなければ、リソースの生成に失敗します。 RTR ファイルにこのプロパティが宣言されていない場合、リソースはスケーラブルにはなりません。したがって、クラスタ管理者はこのプロパティを調整することができず、RGM はスケーラブルサービスプロパティを設定しません。ただし、必要に応じて、明示的に `Network_resources_used` および `Port_list` プロパティを RTR ファイルで宣言できます。これらのプロパティは、スケーラブルサービスだけでなく、非スケーラブルサービスでも有用です。	作成時 (At_creation)	任意
各クラスタノードの `Status` (列挙)	リソースモニターによって設定されます。指定可能な値は、 `degraded`、`faulted`、`unknown`、`offline` です。RGM は、リソースがオンラインになると、値を `unknown` に設定し、オフラインになると `offline` に設定します。	不可 (None)	照会のみ
各クラスタノードの `Status_msg` (文字列)	リソースモニターによって、`Status` プロパティと同時に設定されます。このプロパティは、各ノードのリソースごとに設定可能です。RGM は、リソースがオフラインになると、このプロパティに空の文字列を設定します。	不可 (None)	照会のみ

表 **A–2** リソースプロパティ　(続き)

プロパティ名	説明	更新の可否	カテゴリ
`Thorough_probe_ interval` (整数)	高オーバーヘッドのリソース障害検証の呼び出し間隔 (秒)。このプロパティは、RGM のみが作成でき、RTR ファイルで宣言されている場合は、管理者は利用できます。デフォルト値が RTR ファイルで指定されている場合は、このプロパティは任意です。 リソースタイプファイル内に `Tunable` 属性が指定されていない場合、このプロパティの `Tunable` 値は `When_disabled` になります。 `Default` 属性が RTR ファイルのプロパティ宣言に指定されていない場合は、このプロパティは必須です。	無効時 (When_ disabled)	条件付き
`Type` (文字列)	このリソースがインスタントであるリソースタイプ。	不可 (None)	必須
`Type_version` (文字列)	現在このリソースに関連付けられているリソースタイプのバージョンを指定します。このプロパティは RTR ファイル内に宣言できません。したがって、RGM によって自動的に作成されます。このプロパティの値は、リソースの型の `RT_version` プロパティと等しくなります。リソースの作成時、`Type_version` プロパティはリソースタイプ名の接尾辞として表示されるだけで、明示的には指定されません。リソースを編集すると、`Type_version` の値が変更されます。 次の項目から派生: ■ 現在のリソースタイプのバージョン ■ RTR ファイル内の `#$upgrade_from` ディレクティブ	説明を参照	説明を参照
`UDP_affinity` (ブール値)	true の場合、指定のクライアントからの UDP トラフィックはすべて現在クライアントの TCP トラフィックを処理しているサーバーノードに送信されます。 このプロパティは、`Load_balancing_policy` が `Lb_sticky` または `Lb_sticky_wild` の場合にかぎり有効です。さらに、`Weak_affinity` を False (デフォルト値) に設定する必要があります。 このプロパティは、スケーラブルサービス専用です。	無効時 (When_ disabled)	任意

表 **A–2** リソースプロパティ　(続き)

プロパティ名	説明	更新の可否	カテゴリ
`Weak_affinity` (ブール値)	true の場合、弱い形式のクライアントアフィニティが有効になります。これにより、指定のクライアントからの接続を同じサーバーノードに送信できます。ただし、次の場合は例外です。 ■ 障害モニターの再起動、リソースのフェイルオーバーまたはスイッチオーバー、障害発生後のノードのクラスタへの再接続などによるサーバーリスナーの起動時 ■ 管理アクションによるスケーラブルリソースの `Load_balancing_weights` の変更時 メモリーの消費とプロセッササイクルの点で、弱いアフィニティによるオーバーヘッドはデフォルトの形式よりも低くなります。 このプロパティは、`Load_balancing_policy` が `Lb_sticky` または `Lb_sticky_wild` の場合にかぎり有効です。 このプロパティは、スケーラブルサービス専用です。	無効時 (When_disabled)	オプション

リソースグループプロパティ

以下の表に、Sun Cluster によって定義されるリソースタイププロパティを示します。プロパティ値は、次のように分類されます (「カテゴリ」列)。

- 必須 — 管理者は、管理ユーティリィティでリソースグループを作成するときに、必ず値を指定する必要があります。
- 任意 — 管理者がリソースグループの作成時に値を指定しない場合、システムがデフォルト値を提供します。
- 照会のみ— 管理ツールから直接設定できません。

「更新の可否」列は、初期設定後に、そのプロパティが更新可能かどうかを示しています。

表 **A–3** リソースグループプロパティ

プロパティ名	説明	更新の可否	概要
Auto_start_on_new_cluster (ブール値)	このプロパティを使用すると、新しいクラスタを形成するとき、Resource Group の自動起動を無効にすることができます。 デフォルトは TRUE です。TRUE の場合、クラスタが再起動するとき、Resource Group Manager はリソースグループを自動的に起動して、Desired_primaries を有効にしようと試みます。FALSE に設定されている場合、クラスタのすべてのノードが同時に再起動したとき、Resource Group Manager はリソースグループを自動的に起動しません。	可	任意
Desired_ primaries (整数)	グループが同時にオンラインになることができるノードの数。 デフォルトは 1 です。RG_mode プロパティが Failover の場合、このプロパティの値を 1 より大きく設定することはできません。RG_mode プロパティが Scalable の場合は、1 より大きな値を設定できます。	可	任意
Failback (ブール値)	クラスタメンバーシップが変更されたとき、グループがオンラインになるノードセットを再計算するかどうかを指定するブール値。再計算により、RGM は優先度の低いノードをオフラインにし、優先度の高いノードをオンラインにすることができます。 デフォルトは、False です。	可	任意
Global_ resources_ used (文字配列)	クラスタファイルシステムがこのリソースグループ内のリソースによって使用されるかどうかを指定します。管理者はアスタリスク (*) か空文字列 (“”) を指定できます。すべてのグローバルリソースを指定するときはアスタリスク、グローバルリソースを一切指定しない場合は空文字列を指定します。 デフォルトでは、すべての広域リソースです。	可	任意

表 **A–3** リソースグループプロパティ　(続き)

プロパティ名	説明	更新の可否	概要
`Implicit_network_dependencies` (ブール値)	`True` の場合に、グループ内のネットワークアドレスリソースに対し、非ネットワークアドレスリソースの暗黙の強い依存性を RGM が強制することを指定するブール値。ネットワークアドレスリソースには、論理ホスト名と共有アドレスリソース型があります。 スケーラブルリソースグループの場合、ネットワークアドレスリソースを含んでいないため、このプロパティは効果がありません。 デフォルトは、`True` です。	可	任意
`Maximum_ primaries` (整数)	グループが同時にオンラインになることのできるノードの最大数。 デフォルトは 1 です。`RG_mode` プロパティが `Failover` の場合、このプロパティの値を 1 より大きく設定することはできません。`RG_mode` プロパティが `Scalable` の場合は、1 より大きな値を設定できます。	可	任意
`Nodelist` (文字配列)	優先順位に従ってグループをオンラインにできるクラスタノードのリスト。これらのノードは、リソースグループの潜在的主ノードまたはマスターです。 デフォルトは、すべてのクラスタノードのリストになります。	可	任意
`Pathprefix` (文字列)	グループ内のリソースが書き込めるクラスタファイルシステムにあるディレクトリは、重要な管理ファイルを書き込めます。一部のリソースの必須プロパティです。各リソースグループの `Pathprefix` は、一意にする必要があります。 デフォルトは空の文字列です。	可	任意

表 **A–3** リソースグループプロパティ　(続き)

プロパティ名	説明	更新の可否	概要
Pingpong_ interval (整数)	再構成が生じた場合、あるいは scha_control -O GIVEOVER コマンドまたは SCHA_GIVEOVER 引数が指定された scha_control() 関数の実行結果として、どのノードでリソースグループをオンラインにするかを判断するときに RGM が使用する負以外の整数値 (秒)。 再構成において、リソースの Start または Prenet_start メソッドがゼロ以外の値で終了、またはタイムアウトによって終了したことが原因で、Pingpong_interval で指定した秒数内に、リソースグループをオンラインにするのを 2 回以上失敗した場合、RGM はそのノードはリソースグループのホストとして不適切だと判断し、別のマスターを探します。 リソースの scha_control コマンドまたは scha_control() 関数の呼び出しの場合、特定のノード上で Pingpong_interval 秒以内にリソースグループがオフラインにならないと、このノードはリソースグループのホストとして不適切と判断され、別のノードから scha_control() が呼び出されます。 デフォルト値は、3,600 秒 (1 時間) です。	可	任意
Resource_ list (文字配列)	グループに含まれるリソースのリスト。管理者はこのプロパティを直接設定しません。このプロパティは、管理者がリソースグループにリソースを追加したり、リソースを削除したときに、RGM によって更新されます。 デフォルトは、空のリストです。	不可	照会のみ
RG_ description (文字列)	リソースグループの簡単な説明。 デフォルトは空の文字列です。	可	任意

表 **A–3** リソースグループプロパティ　(続き)

プロパティ名	説明	更新の可否	概要
RG_mode (列挙)	リソースグループがフェイルオーバーグループかスケーラブルグループかを指定します。このプロパティの値が Failover の場合、RGM はグループの Maximum_primaries プロパティを 1 に設定し、そのリソースグループをマスターするのを単一のノードに制限します。 このプロパティの値が Scalable の場合、RGM は Maximum_primaries プロパティが 1 より大きい値を持つことを許可し、複数のノードで同時にそのグループをマスターできるようにします。Failover プロパティの値が True のリソースを、RG_mode の値が Scalable のリソースグループに追加することはできません。 Maximum_primaries に 1 が設定されている場合のデフォルトは、Failover です。Maximum_primaries に 2 以上が設定されている場合のデフォルトは、Scalable です。	不可	任意
RG_name (文字列型)	リソースグループの名前。この名前は、クラスタ内で一意のものでなければなりません。	不可	必須
RG_project_name (文字列)	リソースグループに関連付けられた Solaris プロジェクト名。このプロパティは、CPU の共有、クラスタデータサービスのリソースプールといった Solaris のリソース管理機能に適用できます。RGM は、リソースグループをオンラインにするとき、Resource_project_name プロパティセットを持たないリソースに対して、このプロジェクトで関連付けられたプロセスを起動します。プロジェクトデータベース内に存在するプロジェクト名を指定する必要があります。また、root ユーザーは、このプロジェクトのメンバーとして構成されている必要があります。 このプロパティは Solaris 9 以降でサポートされます。 注 – このプロパティへの変更を有効にするには、リソースを再起動する必要があります。	任意の時点 (Anytime)	必須

表 **A–3** リソースグループプロパティ　(続き)

プロパティ名	説明	更新の可否	概要
各クラスタノードの RG_state (列挙)	RGM によって Online、Offline、Pending_online、Pending_offline、Pending_online_blocked、Error_stop_failed、または Online_faulted に設定され、各クラスタノード上のグループの状態を示します。 このプロパティは、ユーザーは構成できません。しかし、scswitch(1M) を呼び出すことによって (あるいは同等の scsetup(1M) か SunPlex Manager コマンドを使用して) このプロパティを間接的に設定することは可能です。	不可	照会のみ

リソースプロパティの属性

次の表は、システム定義のプロパティの変更または拡張プロパティの作成に使用できるリソースプロパティ属性を示したものです。

注意 – boolean、enum、int タイプのデフォルト値に、NULL または空の文字列 ("") は指定できません。

表 **A–4** リソースプロパティの属性

プロパティ	説明
Property	リソースプロパティの名前。
Extension	このプロパティを使用すると、RTR ファイルのエントリで、リソースタイプの実装によって定義された拡張プロパティが宣言されていることを示します。使用されない場合は、そのエントリはシステム定義プロパティです。
Description	プロパティを簡潔に記述した注記 (文字列)。RTR ファイル内でシステム定義プロパティに対する Description 属性を設定することはできません。
プロパティのタイプ	指定可能な型は、 string、boolean、int、enum、stringarray です。RTR ファイル内で、システム定義プロパティの型の属性を設定することはできません。タイプは、RTR ファイルのエントリに登録できる、指定可能なプロパティ値とタイプ固有の属性を決定します。enum タイプは、文字列値のセットです。
デフォルト	プロパティのデフォルト値を指定します。

表 **A–4** リソースプロパティの属性　(続き)

プロパティ	説明
調整	クラスタ管理者が、リソースのプロパティ値をいつ設定できるかを示します。管理者がプロパティを設定できないようにするには、None または False に設定します。管理者にプロパティの調整を許可する属性値は、次のとおりです。True または Anytime (任意の時点)、At_creation (リソースの作成時のみ)、When_disabled (リソースがオフラインのとき)。 デフォルトは、 True (Anytime) です。
Enumlist	enum タイプの場合、プロパティに設定できる文字列値のセット。
Min	int タイプの場合、プロパティに設定できる最小値。
Max	int タイプの場合、プロパティに設定できる最大値。
Minlength	string および stringarray タイプの場合、設定できる文字列の長さの最小値。
Maxlength	string および stringarray タイプの場合、設定できる文字列の長さの最大値。
Array_minsize	stringarray タイプの場合、設定できる配列要素の最小数。
Array_maxsize	stringarray タイプの場合、設定できる配列要素の最大数。

付録 B

データサービスのコード例

この付録では、データサービスの各メソッドの完全なコード例を示します。また、リソースタイプ登録 (RTR) ファイルの内容も示します。

この付録に含まれるコードリストは、次のとおりです。

リソースタイプ登録ファイルのリスト

リソースタイプ登録 (RTR) ファイルには、クラスタ管理者がデータサービスを登録するとき、データサービスの初期構成を定義するリソースとリソースタイプのプロパティ宣言が含まれています。

例 **B–1** SUNW.Sample RTR ファイル

```
#
# Copyright (c) 1998-2003 by Sun Microsystems, Inc.
# All rights reserved.
#
# ドメインネームサービス (DNS) の登録情報
#
```

例 **B–1** SUNW.Sample RTR ファイル　(続き)

```
#pragma ident   "@(#)SUNW.sample   1.1   00/05/24 SMI"

RESOURCE_TYPE = "sample";
VENDOR_ID = SUNW;
RT_DESCRIPTION = "Domain Name Service on Sun Cluster";

RT_VERSION ="1.0";
API_VERSION = 2;
FAILOVER = TRUE;

RT_BASEDIR=/opt/SUNWsample/bin;
PKGLIST = SUNWsample;

START               = dns_svc_start;
STOP                = dns_svc_stop;

VALIDATE            = dns_validate;
UPDATE              = dns_update;

MONITOR_START       = dns_monitor_start;
MONITOR_STOP        = dns_monitor_stop;
MONITOR_CHECK       = dns_monitor_check;

# リソースタイプ宣言のあとに、中括弧に囲まれたリソースプロパティ宣言のリスト
# が続く。プロパティ名宣言は、各エントリの左中括弧の直後にある最初
# の属性である必要がある。
#
# <method>_timeout プロパティは、RGM がメソッドの呼び出しが失敗
# したという結論を下すまでの時間 (秒) を設定する。
# すべてのメソッドタイムアウトの MIN 値は 60 秒に設定されている。こ
# れは、管理者が短すぎる時間を設定することを防ぐためである。短すぎ
# る時間を設定すると、スイッチオーバーやフェイルオーバーの性能が上
# がらず、さらには、予期せぬ RGM アクションが発生する可能性がある
# (間違ったフェイルオーバー、ノードの再起動、リソースグループの
# ERROR_STOP_FAILED 状態への移行、オペレータの介入の必要性など)。
# メソッドタイムアウトに短すぎる時間を設定すると、データサービス全
# 体の可用性を下げることになる。
{
   PROPERTY = Start_timeout;
   MIN=60;
   DEFAULT=300;
}

{
             PROPERTY = Stop_timeout;
            MIN=60;
           DEFAULT=300;
}
{
        PROPERTY = Validate_timeout;
        MIN=60;
        DEFAULT=300;
```

例 **B–1** SUNW.Sample RTR ファイル　(続き)

```
}
{
        PROPERTY = Update_timeout;
        MIN=60;
        DEFAULT=300;
}
{
        PROPERTY = Monitor_Start_timeout;
        MIN=60;
        DEFAULT=300;
}
{
        PROPERTY = Monitor_Stop_timeout;
        MIN=60;
        DEFAULT=300;
}
{
        PROPERTY = Thorough_Probe_Interval;
        MIN=1;
        MAX=3600;
        DEFAULT=60;
        TUNABLE = ANYTIME;
}

# 当該ノード上でアプリケーションを正常に起動できないと結論を下すま
# でに、ある期間 (Retry_Interval) に行う再試行の回数。
{
        PROPERTY = Retry_Count;
        MIN=0;
        MAX=10;
        DEFAULT=2;
        TUNABLE = ANYTIME;
}
# Retry_Interval には 60 の倍数を設定する。これは、秒から分に変換さ
# れ、端数が切り上げられるためである。たとえば、50 (秒) という値を
# 指定すると、1 分に変換される。
# このプロパティは再試行数 (Retry_Count) のタイミングを決定する。
{
        PROPERTY = Retry_Interval;
        MIN=60;
        MAX=3600;
        DEFAULT=300;
        TUNABLE = ANYTIME;
}

{
        PROPERTY = Network_resources_used;
        TUNABLE = AT_CREATION;
        DEFAULT = "";
}

#
# 拡張プロパティ
```

例 **B–1** SUNW.Sample RTR ファイル　(続き)

```
# クラスタ管理者はこのプロパティの値を設定して、アプリケーションが使用
# する構成ファイルが入っているディレクトリを示す必要がある。このアプリ
# ケーションの場合、DNS は PXFS (通常は named.conf) 上の DNS 構成ファイ
# ルのパスを指定する。
{
   PROPERTY = Confdir;
   EXTENSION;
   STRING;
   TUNABLE = AT_CREATION;
   DESCRIPTION = "The Configuration Directory Path";
}

# 検証が失敗したと宣言するまでのタイムアウト値 (秒)。
{
        PROPERTY = Probe_timeout;
        EXTENSION;
        INT;
        DEFAULT = 30;
        TUNABLE = ANYTIME;
        DESCRIPTION = "Time out value for the probe (seconds)";
}
```

Start メソッド

データサービスリソースを含むリソースグループがオンラインになったとき、あるいは、リソースが有効になったとき、RGM はそのクラスタノード上で Start メソッドを呼び出します。サンプルのアプリケーションでは、Start メソッドはそのノード上で in.named (DNS) デーモンを起動します。

例 **B–2** dns_svc_start メソッド

```
#!/bin/ksh

#

# HA-DNS の Start メソッド

#

# このメソッドは PMF の制御下でデータサービスを起動する。DNS の
# in.named プロセスを起動する前に、いくつかの妥当性検査を実行する。
#
# データサービスの PMF タグは $RESOURCE_NAME.named である。PMF は、
# 指定された回数 (Retry_count) だけ、サービスを起動しようとする。そ
# して、指定された期間 (Retry_interval) 内で試行回数がこの値を超えた
# 場合、PMF はサービスの起動に失敗したことを報告する。
# Retry_count と Retry_interval は両方とも RTR ファイルに設定されて
```

例 **B–2** dns_svc_start メソッド　(続き)

```
# いるリソースプロパティである。
#pragma ident   "@(#)dns_svc_start   1.1   00/05/24 SMI"



###############################################################################

# プログラム引数を構文解析する。
#
function parse_args # [args ...]
{
        typeset opt

        while getopts 'R:G:T:' opt
        do
                case "$opt" in
                R)
                        # DNS リソース名。
                        RESOURCE_NAME=$OPTARG
                        ;;
                G)
                        # リソースが構成されているリソース
                        # グループの名前。
                        RESOURCEGROUP_NAME=$OPTARG
                        ;;

                T)
                        # リソースタイプ名。
                        RESOURCETYPE_NAME=$OPTARG
                        ;;

                *)
                    logger -p ${SYSLOG_FACILITY}.err \
                    -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME]
\
                    "ERROR: Option $OPTARG unknown"
                     exit 1
                     ;;

                esac
        done
}




###############################################################################
# MAIN
#

###############################################################################
```

例 **B–2** dns_svc_start メソッド　(続き)

```
export PATH=/bin:/usr/bin:/usr/cluster/bin:/usr/sbin:/usr/proc/bin:$PATH


# メッセージの記録に使用する syslog 機能番号を取得する。
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# このメソッドに渡された引数を構文解析する。
parse_args "$@"

PMF_TAG=$RESOURCE_NAME.named

SYSLOG_TAG=$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME

# DNS を起動するため、リソースのConfdir プロパティの値を取得する。
# 入力されたリソース名とリソースグループを使用して、リソースを
# 追加するときにクラスタ管理者が設定したConfdir の値を見つける。
#
config_info=`scha_resource_get -O Extension -R $RESOURCE_NAME

-G $RESOURCEGROUP_NAME Confdir`
# scha_resource_get は拡張プロパティの「タイプ」と「値」を戻す。
# 拡張プロパティの値だけを取得する。
CONFIG_DIR=`echo $config_info | awk '{print $2}'`

# $CONFIG_DIR がアクセス可能であるかどうかを検査する。
if [ ! -d $CONFIG_DIR ]; then
   logger -p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG] \
       "${ARGV0} Directory $CONFIG_DIR missing or not mounted"
   exit 1
fi

# データファイルへの相対パス名が存在する場合、$CONFIG_DIR
# ディレクトリに移動する。
cd $CONFIG_DIR

# named.conf ファイルが$CONFIG_DIR ディレクトリ内に存在するかどうか
# を検査する。
if [ ! -s named.conf ]; then
   logger -p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG] \
       "${ARGV0} File $CONFIG_DIR/named.conf is missing or
empty"
   exit 1
fi

# RTR ファイルからRetry_count の値を取得する。
RETRY_CNT=`scha_resource_get -O Retry_Count -R $RESOURCE_NAME
-G \ $RESOURCEGROUP_NAME
```

例 **B–2** dns_svc_start メソッド　(続き)

```
# RTR ファイルからRetry_interval の値を取得する。この値の単位は秒
# であり、pmfadm に渡すときは分に変換する必要がある。変換時、端数は
# 切り上げられるので注意すること。たとえば、50 秒は1 分に切り上げられる。

((RETRY_INTRVAL = `scha_resource_get -O Retry_Interval
-R $RESOURCE_NAME -G $RESOURCEGROUP_NAMÈ / 60))



# PMF の制御下で in.named デーモンを起動する。$RETRY_INTERVAL の期
# 間、$RETRY_COUNT の回数だけ、クラッシュおよび再起動できる。どちら
# かの値以上クラッシュした場合、PMF は再起動を中止する。
# <$PMF_TAG> というタグですでにプロセスが登録されて
# いる場合、PMF はすでにプロセスが動作していることを示す警告メッセ
# ージを送信する。

echo "Retry interval is "$RETRY_INTRVAL
pmfadm -c $PMF_TAG.named -n $RETRY_CNT -t $RETRY_INTRVAL \
    /usr/sbin/in.named -c named.conf



# HA-DNS が起動していることを示すメッセージを記録する。
if [ $? -eq 0 ]; then
   logger -p ${SYSLOG_FACILITY}.info -t [$SYSLOG_TAG]\
           "${ARGV0} HA-DNS successfully started"
fi
exit 0
```

Stop メソッド

HA-DNS リソースを含むリソースグループがクラスタノード上でオフラインになるとき、あるいは、HA-DNS リソースが無効になるとき、RGM は Stop メソッドを呼び出します。このメソッドは、そのノード上で in.named (DNS) デーモンを停止します。

例 **B–3** dns_svc_stop メソッド

```
#!/bin/ksh
#
# HA-DNS の Stop メソッド
#
# このメソッドは、PMF を使用するデータサービスを停止する。サービス
# が動作していない場合、このメソッドは状態 0 で終了する。その他の値
# は戻さない。リソースは STOP_FAILED 状態になる。
```

例 **B–3** dns_svc_stop メソッド　(続き)

```
#pragma ident   "@(#)dns_svc_stop   1.1   00/05/24 SMI"

###############################################################################
# プログラム引数を構文解析する。
#
function parse_args # [args ...]
{
        typeset opt

        while getopts 'R:G:T:' opt
        do
                case "$opt" in
                R)
                        # DNS リソースの名前。
                        RESOURCE_NAME=$OPTARG
                        ;;
                G)
                        # リソースが構成されているリソース
                        # グループの名前。
                        RESOURCEGROUP_NAME=$OPTARG
                        ;;
                T)
                        # リソースタイプの名前。
                        RESOURCETYPE_NAME=$OPTARG
                        ;;

                *)
                    logger -p ${SYSLOG_FACILITY}.err \
                    -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME]
\
                    "ERROR: Option $OPTARG unknown"
                     exit 1
                     ;;

                esac
        done

}


###############################################################################
# MAIN
#
###############################################################################

export PATH=/bin:/usr/bin:/usr/cluster/bin:/usr/sbin:/usr/proc/bin:$PATH

# メッセージの記録に使用する syslog 機能番号を取得する。
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# このメソッドに渡された引数を構文解析する。
parse_args "$@"
```

例 **B–3** dns_svc_stop メソッド　(続き)

```
PMF_TAG=$RESOURCE_NAME.named
SYSLOG_TAG=$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME

# RTR ファイルから Stop_timeout 値を取得する。
STOP_TIMEOUT=`scha_resource_get -O STOP_TIMEOUT -R $RESOURCE_NAME
-G \ $RESOURCEGROUP_NAMÈ

# PMF 経由で SIGTERM シグナルを使用する規則正しい方法でデータサービ
# スを停止しようとする。SIGTERM がデータサービスを停止できるまで、
# Stop_timeout 値の 80% だけ待つ。停止できない場合、SIGKILL を送信
# して、データサービスを停止しようとする。SIGKILL がデータサービス
# を停止できるまで、Stop_timeout 値の 15% だけ待つ。停止できない場合、
# メソッドは何か異常があったと判断し、0 以外の状態で終了する。
# Stop_timeout の残りの 5% は他の目的のために予約されている。
((SMOOTH_TIMEOUT=$STOP_TIMEOUT * 80/100))

((HARD_TIMEOUT=$STOP_TIMEOUT * 15/100))

# in.named が動作しているかどうかを調べて、動作していれば停止する。
if pmfadm -q $PMF_TAG.named; then
   # SIGTERM シグナルをデータサービスに送信して、合計タイムアウト値
   # の 80% だけ待つ。
   pmfadm -s $PMF_TAG.named -w $SMOOTH_TIMEOUT TERM
   if [ $? -ne 0 ]; then
      logger -p ${SYSLOG_FACILITY}.info -t [SYSLOG_TAG] \
          "${ARGV0} Failed to stop HA-DNS with SIGTERM; Retry
with \
           SIGKILL"

      # SIGTERM シグナルでデータサービスが停止しないので、今度は
      # SIGKILL を使用して、合計タイムアウト値の 15% だけ待つ。
      pmfadm -s $PMF_TAG.named -w $HARD_TIMEOUT KILL
      if [ $? -ne 0 ]; then
          logger -p ${SYSLOG_FACILITY}.err -t [SYSLOG_TAG]
\
          "${ARGV0} Failed to stop HA-DNS; Exiting UNSUCCESFUL"

          exit 1
      fi
fi
else
   # この時点でデータサービスは動作していない。メッセージを記録して、
   # 成功で終了する。
   logger -p ${SYSLOG_FACILITY}.info -t [SYSLOG_TAG] \
           "HA-DNS is not started"

   # HA-DNS が動作していない場合でも、成功で終了し、データサービス
   # リソースが STOP_FAILED 状態にならないようにする。

   exit 0

fi
```

例 **B–3** dns_svc_stop メソッド　(続き)

```
# DNS の停止に成功。メッセージを記録して、成功で終了する。
logger -p ${SYSLOG_FACILITY}.info -t [$SYSLOG_TAG] \
    "HA-DNS successfully stopped"
exit 0
```

gettime ユーティリティ

gettime ユーティリティーは、検証の再起動間の経過時間を PROBE プログラムが追跡するための C プログラムです。このプログラムは、コンパイル後、コールバックメソッドと同じディレクトリ (RT_basedir プロパティが指すディレクトリ) に格納する必要があります。

例 **B–4** gettime.c ユーティリティプログラム

```
#
# このユーティリティープログラムは、データサービスの検証メソッドによ
# って使用され、既知の参照ポイント(基準点) からの経過時間 (秒) を
# 追跡する。このプログラムは、コンパイル後、データサービスのコール
# バックメソッドと同じディレクトリ(RT_basedir) に格納しておくこと。


#pragma ident   "@(#)gettime.c   1.1   00/05/24 SMI"


#include <stdio.h>
#include <sys/types.h>
#include <time.h>

main()
{
    printf("%d\n", time(0));
    exit(0);
}
```

PROBE プログラム

PROBE プログラムは、nslookup(1M) コマンドを使用して、データサービスの可用性を検査します。このプログラムは、Monitor_start コールバックメソッドによって起動され、Monitor_stop コールバックメソッドによって停止されます。

例 **B–5** dns_probe プログラム

```
#!/bin/ksh
#pragma ident   "@(#)dns_probe   1.1   00/04/19 SMI"
#
# HA-DNS の Probe メソッド
#
# このプログラムは、nslookup を使用して、データサービスの可用性を検査
# する。nslookup は DNS サーバーに照会することによって、DNS
# サーバー自身を探す。サーバーが応答しない場合、あるいは、別のサー
# バーが照会に応答した場合、probe メソッドはデータサービスまたはク
# ラスタ内の別のノードになんらかの問題が発生したという結論を下す。
# 検証は、RTR ファイルの THOROUGH_PROBE_INTERVAL で設定さ
# れた間隔で行われる。

#pragma ident   "@(#)dns_probe   1.1   00/05/24 SMI"


################################################################################
# プログラム引数を構文解析する。
#
function parse_args # [args ...]
{
        typeset opt

        while getopts 'R:G:T:' opt
        do
                case "$opt" in
                R)
                        # DNS リソースの名前。
                        RESOURCE_NAME=$OPTARG
                        ;;
                G)
                        # リソースが構成されているリソース
                        # グループの名前。
                        RESOURCEGROUP_NAME=$OPTARG
                        ;;
                T)
                        # リソースタイプの名前。
                        RESOURCETYPE_NAME=$OPTARG
                        ;;

                *)
                    logger -p ${SYSLOG_FACILITY}.err \
                    -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME]
\
                    "ERROR: Option $OPTARG unknown"
                     exit 1
                     ;;

                esac
        done

}
```

例 **B–5** dns_probe プログラム　(続き)

```
#############################################################################
# restart_service ()
#
# この関数は、まずデータサービスの Stop メソッドを呼び出し、
# 次に Start メソッドを呼び出すことによって、データサービスを再起動
# しようとする。データサービスがすでに起動しておらず、
# データサービスのタグが PMF に登録されていない場合、
# この関数はデータサービスをクラスタ内の
# 別のノードにフェイルオーバーする。
#
function restart_service
{

        # データサービスを再起動するには、まず、データサービス自身が
        # PMF 下に登録されているかどうかを確認する。
        pmfadm -q $PMF_TAG
        if [[ $? -eq 0 ]]; then
                # データサービスの TAG が PMF に登録されている場合、
                # データサービスを停止し、起動し直す。

                # 当該リソースの Stop メソッド名と STOP_TIMEOUT 値を取得する。
                STOP_TIMEOUT=`scha_resource_get -O STOP_TIMEOUT
\
                        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAMÈ
                STOP_METHOD=`scha_resource_get -O STOP
\
                        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAMÈ
                hatimerun -t $STOP_TIMEOUT $RT_BASEDIR/$STOP_METHOD
\
                        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME
\
                        -T $RESOURCETYPE_NAME

                if [[ $? -ne 0 ]]; then
                        logger-p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG]
\
                                "${ARGV0} Stop method failed."
                        return 1
                fi

                # 当該リソースの Start メソッド名と START_TIMEOUT 値を取得する。
                START_TIMEOUT=`scha_resource_get -O START_TIMEOUT
\
                        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAMÈ
                START_METHOD=`scha_resource_get -O START
\
                        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAMÈ
                hatimerun -t $START_TIMEOUT $RT_BASEDIR/$START_METHOD
\
                        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME
\
                        -T $RESOURCETYPE_NAME
```

例 **B–5** dns_probe プログラム　(続き)

```
                if [[ $? -ne 0 ]]; then
                        logger-p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG]
\
                                "${ARGV0} Start method
failed."
                        return 1
                fi


        else
                # データサービスの TAG が PMF に登録されていない場合、
                # データサービスが PMF 下で許可されている再試行最大回数を
                # 超えていることを示す。したがって、データサービスを再起動
                # してはならない。その代わりに、同じクラスタ内にある別のノード
                # にフェイルオーバーを試みる。
                scha_control -O GIVEOVER -G $RESOURCEGROUP_NAME
\
                        -R $RESOURCE_NAME
        fi

        return 0
}




##############################################################################
# decide_restart_or_failover ()
#
# この関数は、検証が失敗したときに行うべきアクション、つまり、デー
# タサービスをローカルで再起動するか、クラスタ内の別のノードに
# フェイルオーバーするかを決定する。
{

   # 最初の再起動の試行であるかどうかを検査する。
   if [ $retries -eq 0 ]; then
         # 最初の失敗である。最初の試行の時刻を記録する。
         start_time=`$RT_BASEDIR/gettimè
         retries=`expr $retries + 1`
         # 最初の失敗であるので、データサービスを再起動しようと試行する。
         restart_service
         if [ $? -ne 0 ]; then
            logger -p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG] \
                "${ARGV0} Failed to restart data service."
            exit 1
         fi
   else
      # 最初の失敗ではない。
      current_time=`$RT_BASEDIR/gettimè
      time_diff=`expr $current_time - $start_timè
      if [ $time_diff -ge $RETRY_INTERVAL ]; then
         # この失敗は再試行最大期間後に発生した。
```

例 **B–5** dns_probe プログラム　(続き)

```
          # したがって、再試行カウンタをリセットし、
          # 再試行時間をリセットし、さらに再試行する。
          retries=1
          start_time=$current_time
          # 前回の失敗が Retry_interval よりも以前に発生しているので、
          # データサービスを再起動しようと試行する。
          restart_service
          if [ $? -ne 0 ]; then
             logger -p ${SYSLOG_FACILITY}.err \
                 -t [$SYSLOG_TAG
                 "${ARGV0} Failed to restart HA-DNS."
             exit 1
          fi
       elif [ $retries -ge $RETRY_COUNT ]; then
          # 再試行最大期間内であり、再試行カウンタは満了
          # している。したがって、フェイルオーバーする。
          retries=0
          scha_control -O GIVEOVER -G $RESOURCEGROUP_NAME \
              -R $RESOURCE_NAME
          if [ $? -ne 0 ]; then
             logger -p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG] \
                 "${ARGV0} Failover attempt failed."
             exit 1
          fi
       else
          # 再試行最大期間内であり、再試行カウンタは満了
          # していない。したがって、さらに再試行する。
          retries=`expr $retries + 1`
          restart_service
          if [ $? -ne 0 ]; then
             logger -p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG] \
                 "${ARGV0} Failed to restart HA-DNS."
             exit 1
          fi
       fi
fi
}


##############################################################################
# MAIN
##############################################################################

export PATH=/bin:/usr/bin:/usr/cluster/bin:/usr/sbin:/usr/proc/bin:$PATH

# メッセージの記録に使用する syslog 機能番号を取得する。
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# このメソッドに渡された引数を構文解析する。
parse_args "$@"

PMF_TAG=$RESOURCE_NAME.named
SYSLOG_TAG=$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME
```

例 **B–5** dns_probe プログラム　(続き)

```
# 検証が行われる間隔はシステム定義プロパティ THOROUGH_PROBE_INTERVAL
# に設定されている。scha_resource_get でこのプロパティの値を取得する。
PROBE_INTERVAL=`scha_resource_get -O THOROUGH_PROBE_INTERVAL
-R $RESOURCE_NAME -G $RESOURCEGROUP_NAMÈ

# 検証用のタイムアウト値を取得する。この値は RTR ファイルの
# PROBE_TIMEOUT 拡張プロパティに設定されている。nslookup のデフォル
# トのタイムアウトは 1.5 分。
probe_timeout_info=`scha_resource_get -O Extension -R $RESOURCE_NAME
-G \$RESOURCEGROUP_NAME Probe_timeout`
PROBE_TIMEOUT=`echo $probe_timeout_info | awk '{print $2}'`

# リソースの NETWORK_RESOURCES_USED プロパティの値を取得して、
# DNS がサービスを提供するサーバーを見つける。
DNS_HOST=`scha_resource_get -O NETWORK_RESOURCES_USED -R
$RESOURCE_NAME -G \$RESOURCEGROUP_NAMÈ

# システム定義プロパティ Retry_count から再試行最大回数を取得する。
RETRY_COUNT=`scha_resource_get -O RETRY_COUNT -R $RESOURCE_NAME
-G \$RESOURCEGROUP_NAMÈ

# システム定義プロパティRetry_interval から再試行最大期間を取得する。
RETRY_INTERVAL=`scha_resource_get -O RETRY_INTERVAL -R
$RESOURCE_NAME -G \$RESOURCEGROUP_NAMÈ

# リソースタイプの RT_basedir プロパティから gettime ユーティリティーの
# 完全パスを取得する。
RT_BASEDIR=`scha_resource_get -O RT_BASEDIR -R $RESOURCE_NAME
-G \$RESOURCEGROUP_NAMÈ

# 検証は無限ループで動作し、nslookup コマンドを実行し続ける。
# nslookup 応答用の一時ファイルを設定する。
DNSPROBEFILE=/tmp/.$RESOURCE_NAME.probe
probefail=0
retries=0

while :
do
   # 検証が動作すべき期間は THOROUGH_PROBE_INTERVAL プロパティに指
   # 定されている。したがって、THOROUGH_PROBE_INTERVAL の間、検証
   # プログラムが休眠するように設定する。
   sleep $PROBE_INTERVAL

   # DNS がサービスを提供している IP アドレス上で nslookup コマンド
   # を実行する。
   hatimerun -t $PROBE_TIMEOUT /usr/sbin/nslookup $DNS_HOST $DNS_HOST
\
> $DNSPROBEFILE 2>&1

   retcode=$?
        if [ retcode -ne 0 ]; then
                probefail=1
```

例 **B–5** dns_probe プログラム　(続き)

```
        fi

   # nslookup への応答が HA-DNS サーバーから来ており、
   # /etc/resolv.conf ファイル内に指定されているほかのネームサーバー
   # から来ていないことを確認する。
   if [ $probefail -eq 0 ]; then
      # Get the name of the server that replied to the nslookup query.
                  SERVER=` awk ' $1=="Server:" {
print $2 }' \
                  $DNSPROBEFILE | awk -F. ' { print $1 } ' `
               if [ -z "$SERVER" ];
then
                        probefail=1
               else
                        if [ $SERVER != $DNS_HOST ]; then
                                probefail=1
                        fi
               fi
        fi

   # probefail 変数が 0 以外である場合、nslookup コマンドがタイム
   # アウトしたか、あるいは、別のサーバー (/etc/resolv.conf ファイ
   # ルに指定されている) から照会への応答が来ていることを示す。
   # どちらの場合でも、DNS サーバーは応答していないので、
   # このメソッドは decide_restart_or_failover を呼び出して、
   # データサービスをローカルで起動するか、あるいは、別のノードに
   # フェイルオーバーするかを評価する。

   if [ $probefail -ne 0 ]; then
         decide_restart_or_failover
   else
         logger -p ${SYSLOG_FACILITY}.info -t [$SYSLOG_TAG]\
         "${ARGV0} Probe for resource HA-DNS successful"
   fi
done
```

Monitor_start メソッド

このメソッドは、データサービスの PROBE プログラムを起動します。

例 **B–6** dns_monitor_start メソッド

```
#!/bin/ksh
#
# HA-DNS の Monitor_start メソッド
#
# このメソッドは、PMF の制御下でデータサービスのモニター(検証) を
```

例 **B–6** dns_monitor_start メソッド　(続き)

```
# 起動する。モニターは一定の間隔でデータサービスを検証するプロセス
# で、問題が発生すると、データサービスを同じノード上で再起動するか、
# クラスタ内の別のノードにフェイルオーバーする。モニター用の PMF
# タグは $RESOURCE_NAME.monitor。


#pragma ident   "@(#)dns_monitor_start   1.1   00/05/24 SMI"

################################################################################
# プログラム引数を構文解析する。
#
function parse_args # [args ...]
{
        typeset opt

        while getopts 'R:G:T:' opt
        do
                case "$opt" in
                R)
                        # DNS リソースの名前。
                        RESOURCE_NAME=$OPTARG
                        ;;
                G)
                        # リソースが構成されているリソース
                        # グループの名前。
                        RESOURCEGROUP_NAME=$OPTARG
                        ;;
                T)
                        # リソースタイプの名前。
                        RESOURCETYPE_NAME=$OPTARG
                        ;;

                *)
          logger -p ${SYSLOG_FACILITY}.err \
                    -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME]
\
                     "ERROR: Option $OPTARG unknown"
                      exit 1
                      ;;
                esac
        done

}


################################################################################
# MAIN
#
################################################################################

export PATH=/bin:/usr/bin:/usr/cluster/bin:/usr/sbin:/usr/proc/bin:$PATH
```

例 **B–6** dns_monitor_start メソッド　(続き)

```
# メッセージの記録に使用する syslog 機能を取得する。
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# このメソッドに渡された引数を構文解析する。
parse_args "$@"

PMF_TAG=$RESOURCE_NAME.monitor
SYSLOG_TAG=$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME

# データサービスの RT_BASEDIR プロパティを取得することによって、検
# 証メソッドが存在する場所を見つける。
RT_BASEDIR=`scha_resource_get -O RT_BASEDIR -R $RESOURCE_NAME
-G \$RESOURCEGROUP_NAMÈ

# PMF の制御下でデータサービスの検証を開始する。無限再試行オプショ
# ンを使用して検証メソッドを起動する。リソースの名前、タイプ、および
# グループを検証メソッドに渡す。
pmfadm -c $PMF_TAG.monitor -n -1 -t -1 \
    $RT_BASEDIR/dns_probe -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME
\
    -T $RESOURCETYPE_NAME

# HA-DNS のモニターが起動されたことを示すメッセージを記録する。
if [ $? -eq 0 ]; then
   logger -p ${SYSLOG_FACILITY}.info -t [$SYSLOG_TAG] \
           "${ARGV0} Monitor for HA-DNS successfully started"
fi
exit 0
```

Monitor_stop メソッド

このメソッドは、データサービスの PROBE プログラムを停止します。

例 **B–7** dns_monitor_stop メソッド

```
#!/bin/ksh
#
# HA-DNS の Monitor_stop メソッド
#
# PMF を使用して動作しているモニターを停止する。


#pragma ident   "@(#)dns_monitor_stop   1.1   00/05/24 SMI"


###############################################################################
```

例 **B–7** dns_monitor_stop メソッド　*(続き)*

```
# プログラム引数を構文解析する。
#
function parse_args # [args ...]
{
        typeset opt

        while getopts 'R:G:T:' opt
        do
                case "$opt" in
                R)
                        # DNS リソースの名前。
                        RESOURCE_NAME=$OPTARG
                        ;;
                G)
                        # リソースが構成されているリソース
                        # グループの名前。
                        RESOURCEGROUP_NAME=$OPTARG
                        ;;
                T)
                        # リソースタイプの名前。
                        RESOURCETYPE_NAME=$OPTARG
                        ;;

                *)
                    logger -p ${SYSLOG_FACILITY}.err \
                    -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME]
\
                    "ERROR: Option $OPTARG unknown"
                     exit 1
                     ;;

                esac
        done

}


##############################################################################
# MAIN
#
##############################################################################

export PATH=/bin:/usr/bin:/usr/cluster/bin:/usr/sbin:/usr/proc/bin:$PATH

# メッセージの記録に使用する syslog 機能を取得する。
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# このメソッドに渡された引数を構文解析する。
parse_args "$@"

PMF_TAG=$RESOURCE_NAME.monitor
SYSLOG_TAG=$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME
```

例 **B–7** dns_monitor_stop メソッド　*(続き)*

```
# モニターが動作しているかどうかを調べて、動作していれば停止する。
if pmfadm -q $PMF_TAG.monitor; then
   pmfadm -s $PMF_TAG.monitor KILL
   if [ $? -ne 0 ]; then
      logger -p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG] \
          "${ARGV0} Could not stop monitor for resource " \
          $RESOURCE_NAME
           exit 1
   else
      # モニターは正常に停止している。メッセージを記録する。
      logger -p ${SYSLOG_FACILITY}.info -t [$SYSLOG_TAG]\
          "${ARGV0} Monitor for resource " $RESOURCE_NAME
\
          " successfully stopped"
   fi
fi

exit 0
```

Monitor_check メソッド

このメソッドは、Confdir プロパティが示すディレクトリの存在を確認します。PROBE メソッドがデータサービスを新しいノードにフェイルオーバーするとき、RGM は Monitor_check を呼び出し、また、潜在マスターであるノードを検査します。

例 **B–8** dns_monitor_check メソッド

```
#!/bin/ksh
#
# DNS の Monitor_check メソッド
#
# 障害モニターがデータサービスを新しいノードにフェイルオーバー
# するとき、RGM はこのメソッドを呼び出す。Monitor_check は
# Validate メソッドを呼び出して、新しいノード上で構成ディレク
# トリおよびファイルが利用できるかどうかを確認する。


#pragma ident   "@(#)dns_monitor_check 1.1   00/05/24 SMI"

################################################################################
# プログラム引数を構文解析する。
#
function parse_args # [args ...]
{
   typeset opt
```

例 **B–8** dns_monitor_check メソッド　(続き)

```
   while getopts 'R:G:T:' opt
   do
      case "$opt" in

      R)
      # DNS リソースの名前。
      RESOURCE_NAME=$OPTARG
      ;;

      G)
      # リソースが構成されているリソースグループの名前。
      RESOURCEGROUP_NAME=$OPTARG
      ;;

      T)
      # リソースタイプの名前。
      RESOURCETYPE_NAME=$OPTARG
      ;;

      *)
      logger -p ${SYSLOG_FACILITY}.err \
      -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME]
\
      "ERROR: Option $OPTARG unknown"
      exit 1
      ;;

      esac
   done

}

##############################################################################
# MAIN
##############################################################################

export PATH=/bin:/usr/bin:/usr/cluster/bin:/usr/sbin:/usr/proc/bin:$PATH

# メッセージの記録に使用する syslog 機能を取得する。
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# このメソッドに渡された引数を構文解析する。
parse_args "$@"

PMF_TAG=$RESOURCE_NAME.named
SYSLOG_TAG=$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME

# リソースタイプの RT_BASEDIR プロパティから Validate メソッドの完全パスを
# 取得する。
RT_BASEDIR=`scha_resource_get -O RT_BASEDIR -R $RESOURCE_NAME
\
   -G $RESOURCEGROUP_NAME
```

例 **B–8** dns_monitor_check メソッド　(続き)

```
# 当該リソースの Validate メソッド名を取得する。
VALIDATE_METHOD=`scha_resource_get -O VALIDATE \
   -R $RESOURCE_NAME -G $RESOURCEGROUP_NAMÈ

# データサービスを起動するための Confdir プロパティの値を取得する。
# 入力されたリソース名とリソースグループを使用して、リソースを
# 追加するときに設定した Confdir の値を取得する。
config_info=`scha_resource_get -O Extension -R $RESOURCE_NAME
-G $RESOURCEGROUP_NAME Confdir`

# scha_resource_get は、拡張プロパティの値とともにタイプも戻す。
# awk を使用して、拡張プロパティの値だけを取得する。
CONFIG_DIR=`echo $config_info | awk '{print $2}'`

# Validate メソッドを呼び出して、データサービスを新しいノードに
# フェイルオーバーできるかどうかを確認する。
$RT_BASEDIR/$VALIDATE_METHOD -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME
\
   -T $RESOURCETYPE_NAME -x Confdir=$CONFIG_DIR

# モニター検査が成功したことを示すメッセージを記録する。
if [ $? -eq 0 ]; then
   logger -p ${SYSLOG_FACILITY}.info -t [$SYSLOG_TAG] \
      "${ARGV0} Monitor check for DNS successful."
   exit 0
else
   logger -p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG] \
      "${ARGV0} Monitor check for DNS not successful."
   exit 1
fi
```

Validate メソッド

このメソッドは、Confdir プロパティが示すディレクトリの存在を確認します。RGM がこのメソッドを呼び出すのは、クラスタ管理者がデータサービスを作成したときと、データサービスのプロパティを更新したときです。障害モニターがデータサービスを新しいノードにフェイルオーバーしたときは、Monitor_check メソッドは常にこのメソッドを呼び出します。

例 **B–9** dns_validate メソッド

```
#!/bin/ksh
#
# HA-DNS のValidate メソッド
#
# このメソッドは、リソースの Confdir プロパティを妥当性検査する。
```

例 **B–9** dns_validate メソッド　(続き)

```
# Validate メソッドが呼び出されるのは、リソースが作成されたときと、リソース
# プロパティが更新されたときの 2 つである。リソースが作成されたとき、
# Validate メソッドは -c フラグで呼び出され、すべてのシステム定義プ
# ロパティと拡張プロパティがコマンド行引数として渡される。リソースプロ
# パティが更新されたとき、Validate メソッドは -u フラグで呼び出され、
# 更新されるプロパティのプロパティ/値のペアだけがコマンド行引数とし
# て渡される。
#
# 例：リソースが作成されたとき、コマンド行引数は次のようになる。
#
# dns_validate -c -R <..> -G <...> -T <..>
-r <sysdef-prop=value>...
#       -x <extension-prop=value>.... -g <resourcegroup-prop=value>....
#
# 例：リソースプロパティが更新されたとき、コマンド行引数は次のようになる。
#
# dns_validate -u -R <..> -G <...> -T <..>
-r <sys-prop_being_updated=value>
#   または
# dns_validate -u -R <..> -G <...> -T <..>
-x <extn-prop_being_updated=value>
#


#pragma ident   "@(#)dns_validate   1.1   00/05/24 SMI"

################################################################################
# プログラム引数を構文解析する。
#
function parse_args # [args ...]
{
   typeset opt

   while getopts 'cur:x:g:R:T:G:' opt
   do
                case "$opt" in
                R)
                        # DNS リソースの名前。
                        RESOURCE_NAME=$OPTARG
                        ;;
                G)
                        # リソースが構成されているリソース
                        # グループの名前。
                        RESOURCEGROUP_NAME=$OPTARG
                        ;;
                T)
                        # リソースタイプの名前。
                        RESOURCETYPE_NAME=$OPTARG
                        ;;

                r)
                        # メソッドはシステム定義プロパティにアクセスして
                        # いない。したがって、このフラグは動作なし。
```

例 **B–9** dns_validate メソッド　(続き)

```
                        ;;

                g)
                        # メソッドはリソースグループプロパティにアクセスして
                        # いない。したがって、このフラグは動作なし。
                        ;;

                c)
                        # Validate メソッドがリソースの作成中に呼び出されてい
                        # ることを示す。したがって、このフラグは動作なし。
                        ;;

                u)
                        # リソースがすでに存在しているときは、プロパティの更新
                        # を示す。Confdir プロパティを更新する場合、Confdir
                        # がコマンド行引数に現れるはずである。現れない場合、
                        # メソッドは scha_resource_get を使用して
                        # Confdir を探す必要がある。
                        UPDATE_PROPERTY=1
                        ;;

                x)
                        # 拡張プロパティのリスト。プロパティと値のペア。
                        # 区切り文字は「=」。
                        PROPERTY=`echo $OPTARG | awk -F= '{print $1}'`
                        VAL=`echo $OPTARG | awk -F= '{print $2}'`

                        # Confdir 拡張プロパティがコマンド行上に存在する場合、
                        # その値を記録する。
                        if [ $PROPERTY == "Confdir" ];
                        then
                        CONFDIR=$VAL
                        CONFDIR_FOUND=1
                        fi
                        ;;

                *)
                        logger -p ${SYSLOG_FACILITY}.err \
                        -t [$SYSLOG_TAG] \
                        "ERROR: Option $OPTARG unknown"
                        exit 1
                        ;;
                esac
   done
}

##############################################################################
# MAIN
#
##############################################################################

export PATH=/bin:/usr/bin:/usr/cluster/bin:/usr/sbin:/usr/proc/bin:$PATH
```

例 **B–9** dns_validate メソッド　(続き)

```
# メッセージの記録に使用する syslog 機能を取得する。
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# CONFDIR の値を NULL に設定する。この後、このメソッドは Confdir プロパ
# ティの値を、コマンド行から取得するか、scha_resource_get を使って
# 取得する。
CONFDIR=""
UPDATE_PROPERTY=0
CONFDIR_FOUND=0

# このメソッドに渡された引数を構文解析する。
parse_args "$@"

# プロパティの更新の結果として呼び出されている場合、Validate メソッ
# ドはコマンド行から Confdir 拡張プロパティの値を取得する。そうでな
# い場合、scha_resource_get を使用して Confdir の値を取得する。
if ( (( $UPDATE_PROPERTY == 1 )) &&  (( CONFDIR_FOUND
== 0 )) ); then
   config_info=`scha_resource_get -O Extension -R $RESOURCE_NAME
\
        -G $RESOURCEGROUP_NAME Confdir`
   CONFDIR=`echo $config_info | awk '{print $2}'`
fi

# Confdir プロパティが値を持っているかどうかを確認する。持っていな
# い場合、状態 1 (失敗) で終了する。
if [[ -z $CONFDIR ]]; then
   logger -p ${SYSLOG_FACILITY}.err \
        "${ARGV0} Validate method for resource "$RESOURCE_NAME " failed"
   exit 1
fi

# 実際の Confdir プロパティ値の妥当性検査はここから始まる。

# $CONFDIR がアクセス可能であるかどうかを検査する。
if [ ! -d $CONFDIR ]; then
        logger -p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG]\
            "${ARGV0} Directory $CONFDIR missing or not
mounted"
        exit 1
fi

# named.conf ファイルがConfdir ディレクトリ内に存在するかどうかを
# 検査する。
if [ ! -s $CONFDIR/named.conf ]; then
        logger -p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG]
\
            "${ARGV0} File $CONFDIR/named.conf is missing
or empty"
        exit 1
fi

# Validate メソッドが成功したことを示すメッセージを記録する。
```

例 **B–9** dns_validate メソッド　(続き)

```
logger -p ${SYSLOG_FACILITY}.info -t [$SYSLOG_TAG] \
   "${ARGV0} Validate method for resource "$RESOURCE_NAME
\
   " completed successfully"

exit 0
```

Update メソッド

プロパティが変更された場合、RGM は Update メソッドを呼び出して、そのことを動作中のリソースに通知します。

例 **B–10** dns_update メソッド

```
#!/bin/ksh
#
# HA-DNS の Update メソッド
#
# 実際のプロパティの更新は RGM が行う。更新の影響を受けるのは障害モ
# ニターだけである。したがって、このメソッドは障害モニターを再起動
# する必要がある。


#pragma ident   "@(#)dns_update   1.1   00/05/24 SMI"

################################################################################
# プログラム引数を構文解析する。
#
function parse_args # [args ...]
{
        typeset opt

        while getopts 'R:G:T:' opt
        do
                case "$opt" in
                R)
                        # DNS リソースの名前。
                        RESOURCE_NAME=$OPTARG
                        ;;
                G)
                        # リソースが構成されているリソース
                        # グループの名前。
                        RESOURCEGROUP_NAME=$OPTARG
                        ;;
                T)
                        # リソースタイプの名前。
                        RESOURCETYPE_NAME=$OPTARG
```

例 **B–10** dns_update メソッド　(続き)

```
                        ;;

                *)
                    logger -p ${SYSLOG_FACILITY}.err \
                    -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME]
\
                    "ERROR: Option $OPTARG unknown"
                     exit 1
                     ;;

                esac
        done

}



###############################################################################
# MAIN
#
###############################################################################

export PATH=/bin:/usr/bin:/usr/cluster/bin:/usr/sbin:/usr/proc/bin:$PATH

# メッセージの記録に使用する syslog 機能を取得する。
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# このメソッドに渡された引数を構文解析する。
parse_args "$@"

PMF_TAG=$RESOURCE_NAME.monitor
SYSLOG_TAG=$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME

# リソースの RT_BASEDIR プロパティを取得することによって、検証メソッド
# が存在する場所を見つける。
RT_BASEDIR=`scha_resource_get -O RT_BASEDIR -R $RESOURCE_NAME
-G $RESOURCEGROUP_NAMÈ

# Update メソッドが呼び出されると、RGM は更新されるプロパティの値を
# 更新する。このメソッドは、障害モニター (検証メソッド) が動作し
# ているかどうかを検査し、動作している場合は強制終了し、再起動
# する必要がある。
if pmfadm -q $PMF_TAG.monitor; then

   # すでに動作している障害モニターを強制終了する。
        pmfadm -s $PMF_TAG.monitor TERM
        if [ $? -ne 0 ]; then
                logger -p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG]
\
                    "${ARGV0} Could not stop the monitor"
                exit 1
        else
```

例 **B–10** dns_update メソッド　(続き)

```
                # DNS の停止に成功。メッセージを記録する。
                logger -p ${SYSLOG_FACILITY}.info -t [$SYSLOG_TAG]
\
                    "Monitor for HA-DNS successfully stopped"
        fi

   # モニターを再起動する。
   pmfadm -c $PMF_TAG.monitor -n -1 -t -1 $RT_BASEDIR/dns_probe \
      -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME -T $RESOURCETYPE_NAME
   if [ $? -ne 0 ]; then
          logger -p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG]\
                  "${ARGV0} Could not restart monitor for HA-DNS "
      exit 1
   else
      logger -p ${SYSLOG_FACILITY}.info -t [$SYSLOG_TAG]\
                    "Monitor for HA-DNS successfully restarted"

   fi
fi
exit 0
```

付録 C

サンプル DSDL リソースタイプのコード例

この付録では、SUNW.xfnts リソースタイプの各メソッドの完全なコード例を示します。また、コールバックメソッドが呼び出すサブルーチンのコードを含む、 xfnts.c のコード例を示します。この付録に含まれるコードリストは、次のとおりです。

xfnts.c

このファイルは、SUNW.xfnts メソッドが呼び出すサブルーチンを実装します。

例 **C–1** xfnts.c

```
/*
 * Copyright (c) 1998-2003 by Sun Microsystems, Inc.
 * All rights reserved.
 *
 * xfnts.c - HA-XFS 用の一般的なユーティリティー
 *
 * このユーティリティーは、データサービスと障害モニターの
 * 妥当性検査、起動、および停止を実行するメソッドを持つ。
 * また、データサービスの状態を検証するメソッドも持つ。
 * 検証機能は、成功または失敗だけを戻す。xfnts_probe.c ファイル
 * 内のメソッドで戻された値にもとづいて、アクションが行われる。
```

例 **C–1** xfnts.c　(続き)

```
 *
 */

#pragma ident "@(#)xfnts.c 1.47 01/01/18 SMI"

#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <netinet/in.h>
#include <scha.h>
#include <rgm/libdsdev.h>
#include <errno.h>
#include "xfnts.h"

/*
 * HA-XFS データサービスが完全に起動して動作するまでの
 * 初期タイムアウト。サービスを検証する前に、start_timeout
 * 時間の 3% (SVC_WAIT_PCT) だけ待つ。
 */
#define   SVC_WAIT_PCT      3

/*
 * probe_timeout の 95% の時間でポートと接続する必要がある。
 * また、svc_probe 関数は、残りの時間を使用して、ポートとの接続を切断する。
 */
#define   SVC_CONNECT_TIMEOUT_PCT      95

/*
 * SVC_WAIT_TIME は、svc_wait() で起動している間だけ使用される。
 * svc_wait() では、サービスが起動していることを確認してから戻る必要がある。
 * そのため、svc_probe() を呼び出し、サービスを監視する必要がある。
 * SVC_WAIT_TIME はこのような検証を繰り返す時間間隔である。
 */

#define   SVC_WAIT_TIME      5

/*
 * この値は、probe_timeout に時間が残っていない場合に、
 * 切断タイムアウトとして使用される。
 */

#define   SVC_DISCONNECT_TIMEOUT_SECONDS      2


/*
 * svc_validate():
 *
 * リソース構成に対して、HA-XFS 固有の妥当性検査を行う。
```

例 **C–1** xfnts.c　(続き)

```
 *
 * svc_validate は、次の妥当性を検査する。
 * 1. Confdir_list 拡張プロパティ
 * 2. fontserver.cfg ファイル
 * 3. xfs バイナリ
 * 4. port_list プロパティ
 * 5. ネットワークリソース
 * 6. その他の拡張プロパティ
 *
 * 上記の妥当性検査のいずれかが失敗した場合、0 以上の値を戻す。
 * それ以外の場合、成功を示す 0 を戻す。
 */

int
svc_validate(scds_handle_t scds_handle)
{
   char   xfnts_conf[SCDS_ARRAY_SIZE];
   scha_str_array_t *confdirs;
   scds_net_resource_list_t *snrlp;
   int rc;
   struct stat statbuf;
   scds_port_list_t   *portlist;
   scha_err_t   err;

   /*
    * XFS データサービス用の構成ディレクトリを confdir_list
    * 拡張プロパティから取得する。
    */
   confdirs = scds_get_ext_confdir_list(scds_handle);

   /* confdir_list 拡張プロパティが存在しない場合、エラーを戻す。
    */
   if (confdirs == NULL || confdirs->array_cnt != 1) {
      scds_syslog(LOG_ERR,
          "Property Confdir_list is not set properly.");
      return (1); /* 妥当性検査は失敗。 */
   }

   /*
    * 構成ファイルへのパスを confdir_list 拡張プロパティから構築する。
    * HA-XFS が持つ構成は 1 つだけなので、confdir_list
    * プロパティの最初のエントリを使用する必要がある。
    */
   (void) sprintf(xfnts_conf, "%s/fontserver.cfg",
confdirs->str_array[0]);

   /*
    * HA-XFS の構成ファイルが適切な場所に存在することを確認する。
    * HA-XFS 構成ファイルにアクセスして、アクセス権が
    * 適切に設定されていることを確認する。
    */
   if (stat(xfnts_conf, &statbuf) != 0) {
      /*
```

例 **C–1** xfnts.c　(続き)

```
        * errno.h プロトタイプには void 引数がないので、
        * lint エラーが抑制される。
        */
       scds_syslog(LOG_ERR,
           "Failed to access file <%s> : <%s>",
           xfnts_conf, strerror(errno));   /*lint !e746 */
       return (1);
   }

   /*
    * XFS バイナリが存在し、アクセス権が正しいことを確認する。
    * XFS バイナリは、広域ファイルシステムではなく、
    * ローカルファイルシステム上にあるものと想定する。
    */
   if (stat("/usr/openwin/bin/xfs", &statbuf)
!= 0) {
       scds_syslog(LOG_ERR,
           "Cannot access XFS binary : <%s> ",
strerror(errno));
       return (1);
   }

   /* HA-XFS はポートを 1 つだけ持つ。 */
   err = scds_get_port_list(scds_handle, &portlist);
   if (err != SCHA_ERR_NOERR) {
       scds_syslog(LOG_ERR,
           "Could not access property Port_list: %s.",
          scds_error_string(err));
       return (1); /* 妥当性検査は失敗。 */
   }

#ifdef TEST
   if (portlist->num_ports != 1) {
       scds_syslog(LOG_ERR,
           "Property Port_list must have only one value.");
       scds_free_port_list(portlist);
       return (1); /* 妥当性検査は失敗。 */
   }
#endif

   /*
    * 当該リソースが利用できるネットワークアドレスリソースを
    * 取得しようとして失敗した場合、エラーを戻す。
    */
   if ((err = scds_get_rs_hostnames(scds_handle, &snrlp))
       != SCHA_ERR_NOERR) {
       scds_syslog(LOG_ERR,
           "No network address resource in resource group: %s.",
          scds_error_string(err));
       return (1); /* 妥当性検査は失敗。 */
   }

   /* ネットワークアドレスリソースが存在しない場合、エラーを戻す。 */
```

例 **C–1** xfnts.c　(続き)

```
   if (snrlp == NULL || snrlp->num_netresources == 0) {
      scds_syslog(LOG_ERR,
          "No network address resource in resource group.");
      rc = 1;
      goto finished;
   }

   /* ほかの重要な拡張プロパティが設定されていることを確認する。 */
   if (scds_get_ext_monitor_retry_count(scds_handle) <= 0)
{
      scds_syslog(LOG_ERR,
          "Property Monitor_retry_count is not set.");
      rc = 1; /* 妥当性検査は失敗。 */
      goto finished;
   }
   if (scds_get_ext_monitor_retry_interval(scds_handle) <=
0) {
      scds_syslog(LOG_ERR,
          "Property Monitor_retry_interval is not set.");
      rc = 1; /* 妥当性検査は失敗。 */
      goto finished;
   }

   /* すべての妥当性検査は成功した。 */
   scds_syslog(LOG_INFO, "Successful validation.");
   rc = 0;

finished:
   scds_free_net_list(snrlp);
   scds_free_port_list(portlist);

   return (rc); /* 妥当性検査の結果を戻す。 */
}


/*
 * svc_start():
 *
 * X フォントサーバーを起動する。
 * 成功したときは 0 を戻し、失敗したときは 0 以上の値を戻す。
 *
 * XFS サービスを起動するには、
 * /usr/openwin/bin/xfs -config <fontserver.cfg file> -port <port
 * to listen>listen> コマンドを実行する。
 * XFS は PMF の制御下で起動する。XFS は単一インスタンスのサービスとして
 * 起動する。
 * データサービス用の PMF タグの形式は、<resourcegroupname, resourcename,
 * instance_number.svc>。
 * XFS の場合、インスタンスは 1 つだけなので、
 * タグ内の instance_number は 0 である。
 */

int
```

例 **C–1** xfnts.c　(続き)

```
svc_start(scds_handle_t scds_handle)
{
   char    xfnts_conf[SCDS_ARRAY_SIZE];
   char   cmd[SCDS_ARRAY_SIZE];
   scha_str_array_t *confdirs;
   scds_port_list_t    *portlist;
   scha_err_t   err;

   /* 構成ディレクトリを confdir_list プロパティから取得する。 */
   confdirs = scds_get_ext_confdir_list(scds_handle);

   (void) sprintf(xfnts_conf, "%s/fontserver.cfg",
confdirs->str_array[0]);

   /* XFS が使用するポートを Port_list プロパティから取得する。 */
   err = scds_get_port_list(scds_handle, &portlist);
   if (err != SCHA_ERR_NOERR) {
      scds_syslog(LOG_ERR,
          "Could not access property Port_list.");
      return (1);
   }

   /*
    * HA-XFS を起動するためのコマンドを構築する。
    * 注: XFS を停止している間、XFS デーモンは
    * 次のメッセージを出力する。
    * 「/usr/openwin/bin/xfs notice: terminating」
    * このデーモンメッセージを抑制するには、
    * 出力を /dev/null にリダイレクトする。
    */
   (void) sprintf(cmd,
       "/usr/openwin/bin/xfs -config %s -port %d 2>/dev/null",
       xfnts_conf, portlist->ports[0].port);

   /*
    * HA-XFS を PMF の制御下で起動する。HA-XFS は単一インスタンスの
    * サービスとして起動される。scds_pmf_start 関数の最後の引数は、
    * 監視する子プロセスのレベルを示す。このパラメータが
    * -1 である場合、すべての子プロセスを親プロセスと同様に監視することを示す。
    */
   scds_syslog(LOG_INFO, "Issuing a start request.");
   err = scds_pmf_start(scds_handle, SCDS_PMF_TYPE_SVC,
      SCDS_PMF_SINGLE_INSTANCE, cmd, -1);

   if (err == SCHA_ERR_NOERR) {
      scds_syslog(LOG_INFO,
          "Start command completed successfully.");
   } else {
      scds_syslog(LOG_ERR,
          "Failed to start HA-XFS ");
   }

   scds_free_port_list(portlist);
```

例 **C–1** xfnts.c　(続き)

```
   return (err); /* 成功または失敗の状態を戻す。 */
}


/*
 * svc_stop():
 *
 * XFS サーバーを停止する。
 * 成功したときは0 を戻し、失敗したときは0 以上の値を戻す。
 *
 * svc_stop は、scds_pmf_stop ツールキット関数を呼び出すことによって
 * サーバーを停止する。
 */
int
svc_stop(scds_handle_t scds_handle)
{
   scha_err_t   err;

   /*
    * 停止メソッドが成功できるタイムアウト値を Stop_Timeout
    * (システム定義) プロパティに設定する。
    */
   scds_syslog(LOG_ERR, "Issuing a stop request.");
   err = scds_pmf_stop(scds_handle,
       SCDS_PMF_TYPE_SVC, SCDS_PMF_SINGLE_INSTANCE, SIGTERM,
       scds_get_rs_stop_timeout(scds_handle));

   if (err != SCHA_ERR_NOERR) {
      scds_syslog(LOG_ERR,
          "Failed to stop HA-XFS.");
      return (1);
   }

   scds_syslog(LOG_INFO,
       "Successfully stopped HA-XFS.");
   return (SCHA_ERR_NOERR); /* 停止は成功。 */
}

/*
 * svc_wait():
 *
 * データサービスが完全に起動するまで待ち、動作状態を確認する。
 */

int
svc_wait(scds_handle_t scds_handle)
{
   int rc, svc_start_timeout, probe_timeout;
   scds_netaddr_list_t   *netaddr;

   /* 検証に使用するネットワークリソースを取得する。 */
   if (scds_get_netaddr_list(scds_handle, &netaddr)) {
      scds_syslog(LOG_ERR,
```

例 **C–1** xfnts.c　(続き)

```
            "No network address resources found in resource group.");
        return (1);
    }

    /* ネットワークリソースが存在しない場合、エラーを戻す。 */
    if (netaddr == NULL || netaddr->num_netaddrs == 0) {
        scds_syslog(LOG_ERR,
            "No network address resource in resource group.");
        return (1);
    }

    /*
     * 起動メソッドのタイムアウト、検証を行うポート番号、
     * および検証用のタイムアウト値を取得する。
     */
    svc_start_timeout = scds_get_rs_start_timeout(scds_handle);
    probe_timeout = scds_get_ext_probe_timeout(scds_handle);

    /*
     * データサービスを実際に検証する前に、start_timeout の
     * SVC_WAIT_PCT (%) だけ休止状態になる。これによって、データサービスが
     * 完全に起動し、検証に応答できるようになる。
     * 注: SVC_WAIT_PCT の値はデータサービス
     * ごとに異なる可能性がある。
     * sleep() ではなく scds_svc_wait() を
     * 呼び出すと、サービスが繰り返して失敗する場合には中断して、
     * すぐに戻ることができる。
     */
    if (scds_svc_wait(scds_handle, (svc_start_timeout * SVC_WAIT_PCT)/100)
        != SCHA_ERR_NOERR) {

        scds_syslog(LOG_ERR, "Service failed to start.");
        return (1);
    }

    do {
        /*
         * ネットワークリソースの IP アドレスとポート名で、
         * データサービスを検証する。
         */
        rc = svc_probe(scds_handle,
             netaddr->netaddrs[0].hostname,
             netaddr->netaddrs[0].port_proto.port, probe_timeout);
        if (rc == SCHA_ERR_NOERR) {
            /* 成功。リソースを解放して戻る。 */
            scds_free_netaddr_list(netaddr);
            return (0);
        }

        /*
         * データサービスはまだ起動しようとする。しばらくの
         * 間休止状態になり、もう一度検証を行う。sleep() ではなく
         * scds_svc_wait() を呼び出すと、サービスが
```

例 **C–1** xfnts.c　(続き)

```
        * 繰り返し失敗する場合には中断して、すぐに戻ることができる。
        */
       if (scds_svc_wait(scds_handle, SVC_WAIT_TIME)
          != SCHA_ERR_NOERR) {
          scds_syslog(LOG_ERR, "Service failed to start.");
          return (1);
       }

   /* RGM がタイムアウトするのを待って、プログラムを終了する。 */
   } while (1);

}

/*
 * この関数は、HA-XFS リソース用の障害モニターを起動する。
 * そのためには、検証機能をPMF の制御下で起動する。PMF タグの形式は
 * <RG-name,RS-name,instance_number.mon>。
 * PMF の再起動オプションを使用するが、
 * 無限に再起動しない。代わりに、interval/retry_time を RTR ファイルから取得する。
 */

int
mon_start(scds_handle_t scds_handle)
{
   scha_err_t   err;

   scds_syslog_debug(DBG_LEVEL_HIGH,
      "Calling MONITOR_START method for resource <%s>.",
      scds_get_resource_name(scds_handle));

   /*
    * xfnts_probe 検証機能は、ほかの RT 用のコールバックメソッドが
    * インストールされているのと同じサブディレクトリにあるものと想定する。
    * scds_pmf_start の最後のパラメータは、
    * 監視する子プロセスのレベルを示す。検証機能は PMF の制御下で開始されるので
    * 検証プロセスだけを監視すればよい。
    * したがって、この値は 0 である。
    */
   err = scds_pmf_start(scds_handle, SCDS_PMF_TYPE_MON,
       SCDS_PMF_SINGLE_INSTANCE, "xfnts_probe",
0);

   if (err != SCHA_ERR_NOERR) {
      scds_syslog(LOG_ERR,
          "Failed to start fault monitor.");
      return (1);
   }

   scds_syslog(LOG_INFO,
       "Started the fault monitor.");

   return (SCHA_ERR_NOERR); /* モニターの起動に成功。 */
}
```

例 **C–1** xfnts.c　(続き)

```
/*
 * この関数は、HA-XFS リソース用の障害モニターを停止する。
 * これはPMF 経由で行われる。障害モニター用の PMF タグの形式は、
 * <RG-name_RS-name,instance_number.mon>。
 */

int
mon_stop(scds_handle_t scds_handle)
{

   scha_err_t   err;

   scds_syslog_debug(DBG_LEVEL_HIGH,
      "Calling scds_pmf_stop method");

   err = scds_pmf_stop(scds_handle, SCDS_PMF_TYPE_MON,
       SCDS_PMF_SINGLE_INSTANCE, SIGKILL,
       scds_get_rs_monitor_stop_timeout(scds_handle));

   if (err != SCHA_ERR_NOERR) {
      scds_syslog(LOG_ERR,
          "Failed to stop fault monitor.");
      return (1);
   }

   scds_syslog(LOG_INFO,
       "Stopped the fault monitor.");

   return (SCHA_ERR_NOERR); /* モニターの停止は成功。 */
}

/*
 * svc_probe(): データサービスに固有な検証を行う。
 * 0 (成功) から 100 (致命的な障害) の範囲の整数値を戻す。
 *
 * 検証機能は、リソースの Port_list 拡張プロパティで指定されたポート上で、
 * XFS サーバーとの単純ソケット接続を行い、データサービスを ping する。
 * ポートとの接続に失敗した場合、100 の値を戻して、
 * 致命的な障害であることを示す。ポートとの接続は成功したが
 * 切断が失敗した場合、50 の値を戻して、部分的な障害であることを示す。
 *
 */
int
svc_probe(scds_handle_t scds_handle, char *hostname, int port, int
timeout)
{
   int  rc;
   hrtime_t   t1, t2;
   int    sock;
   char   testcmd[2048];
   int    time_used, time_remaining;
```

例 **C–1** xfnts.c　(続き)

```
    time_t      connect_timeout;


    /*
     * データサービスを検証するには、
     * port_list プロパティで指定されている、XFS データサービスを
     * 提供するホスト上のポートとソケット接続を行う。
     * 指定されたポート上で通信するように構成された
     * XFS サービスが接続に応答した場合、検証が成功したと判断する。
     * probe_timeout プロパティに設定された時間だけ待っても
     * 応答がない場合、検証に失敗したと判断する。
     */

    /*
     * SVC_CONNECT_TIMEOUT_PCT をタイムアウトの
     * 百分率として使用し、ポートと接続する。
     */
    connect_timeout = (SVC_CONNECT_TIMEOUT_PCT * timeout)/100;
    t1 = (hrtime_t)(gethrtime()/1E9);

    /*
     * 検証機能は、指定されたホスト名とポートを使用して接続を行う。
     * 実際には、接続は probe_timeout の 95% の時間でタイムアウトする。
     */
    rc = scds_fm_tcp_connect(scds_handle, &sock, hostname, port,
        connect_timeout);
    if (rc) {
       scds_syslog(LOG_ERR,
           "Failed to connect to port <%d> of resource <%s>.",
           port, scds_get_resource_name(scds_handle));
       /* 致命的な障害。 */
       return (SCDS_PROBE_COMPLETE_FAILURE);
    }

    t2 = (hrtime_t)(gethrtime()/1E9);

    /*
     * 接続にかかる実際の時間を計算する。この値は、
     * 接続に割り当てられた時間を示す connect_timeout 以下である
     * 必要がある。接続に割り当てられた時間をすべて使い切った場合、
     * probe_timeout に残った値が当該関数に渡され、
     * 切断タイムアウトとして使用される。そうでなければ、
     * 接続呼び出しで残った時間は切断タイムアウトに追加される。
     *
     */

    time_used = (int)(t2 - t1);

    /*
     * 残った時間(タイムアウトから接続にかかった時間を引いた値) を切断に
     * 使用する。
     */
```

例 **C–1** xfnts.c　(続き)

```
   time_remaining = timeout - (int)time_used;

   /*
    * すべての時間を使い切った場合、ハードコーディングされた小さな
    * タイムアウトを使用して、切断しようとする。
    * これによって、fd リークを防ぐ。
    */
   if (time_remaining <= 0) {
      scds_syslog_debug(DBG_LEVEL_LOW,
          "svc_probe used entire timeout of "
          "%d seconds during connect operation and exceeded the "
          "timeout by %d seconds. Attempting disconnect with timeout"
          " %d ",
          connect_timeout,
          abs(time_used),
          SVC_DISCONNECT_TIMEOUT_SECONDS);

      time_remaining = SVC_DISCONNECT_TIMEOUT_SECONDS;
   }

   /*
    * 切断に失敗した場合、部分的な障害を戻す。
    * 理由: 接続呼び出しは成功した。これは、アプリケーションが
    * 動作していることを意味する。切断が失敗した原因は、
    * 後者の場合、アプリケーションが停止したとは宣言しない。
    * つまり、致命的な障害を戻さない。その代わりに、
    * 部分的な障害であると宣言する。この状態が続く場合、
    * 切断呼び出しは再び失敗し、アプリケーションは再起動される。
    */
   rc = scds_fm_tcp_disconnect(scds_handle, sock, time_remaining);
   if (rc != SCHA_ERR_NOERR) {
      scds_syslog(LOG_ERR,
          "Failed to disconnect to port %d of resource %s.",
          port, scds_get_resource_name(scds_handle));
      /* 部分的な障害。 */
      return (SCDS_PROBE_COMPLETE_FAILURE/2);
   }

   t2 = (hrtime_t)(gethrtime()/1E9);
   time_used = (int)(t2 - t1);
   time_remaining = timeout - time_used;

   /*
    * 時間が残っていない場合、fsinfo による完全なテストを実行せず、
    * SCDS_PROBE_COMPLETE_FAILURE/2 を戻す。これによって、
    * このタイムアウトが続く場合、
    * サーバーは再起動される。
    */
   if (time_remaining <= 0) {
      scds_syslog(LOG_ERR, "Probe timed out.");
      return (SCDS_PROBE_COMPLETE_FAILURE/2);
   }
```

例 **C–1** xfnts.c　(続き)

```
    /*
     * ポートへの接続および接続の切断に成功。
     * fsinfo コマンドを実行してサーバーの状態を完全に検査する。
     * stdout をリダイレクトする。そうしないと fsinfo の実行結果は
     * コンソールに出力される。
     */
    (void) sprintf(testcmd,
        "/usr/openwin/bin/fsinfo -server %s:%d> /dev/null",
        hostname, port);
    scds_syslog_debug(DBG_LEVEL_HIGH,
        "Checking the server status with %s.", testcmd);
    if (scds_timerun(scds_handle, testcmd, time_remaining,
       SIGKILL, &rc) != SCHA_ERR_NOERR || rc != 0) {

       scds_syslog(LOG_ERR,
          "Failed to check server status with command <%s>",
          testcmd);
       return (SCDS_PROBE_COMPLETE_FAILURE/2);
    }
    return (0);
}
```

xfnts_monitor_check メソッド

このメソッドは、基本的なリソースタイプ構成が有効であることを確認します。

例 **C–2** xfnts_monitor_check.c

```
/*
 * Copyright (c) 1998-2003 by Sun Microsystems, Inc.
 * All rights reserved.
 *
 * xfnts_monitor_check.c - HA-XFS のモニター検査メソッド
 */

#pragma ident "@(#)xfnts_monitor_check.c 1.11 01/01/18
SMI"

#include <rgm/libdsdev.h>
#include "xfnts.h"

/*
 * サービスに対して簡単な妥当性検査を行う。
 */

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

例 **C–2** xfnts_monitor_check.c　(続き)

```
{
   scds_handle_t   scds_handle;
   int   rc;

   /* RGM から渡された引数を処理して、syslog を初期化する。*/
   if (scds_initialize(&scds_handle, argc, argv) != SCHA_ERR_NOERR)
{
      scds_syslog(LOG_ERR, "Failed to initialize the handle.");
      return (1);
   }

   rc =  svc_validate(scds_handle);
   scds_syslog_debug(DBG_LEVEL_HIGH,
       "monitor_check method "
       "was called and returned <%d>.", rc);

   /* scds_initialize が割り当てたすべてのメモリーを解放する。*/
   scds_close(&scds_handle);

   /* モニター検査の一環として実行した検証メソッドの結果を戻す。*/
   return (rc);
}
```

xfnts_monitor_start メソッド

このメソッドは、xfnts_probe メソッドを起動します。

例 **C–3** xfnts_monitor_start.c

```
/*
 * Copyright (c) 1998-2003 by Sun Microsystems, Inc.
 * All rights reserved.
 *
 * xfnts_monitor_start.c - HA-XFS のモニター起動メソッド
 */

#pragma ident "@(#)xfnts_monitor_start.c 1.10 01/01/18
SMI"

#include <rgm/libdsdev.h>
#include "xfnts.h"

/*
 * このメソッドは、HA-XFS リソース用の障害モニターを起動する。
 * そのためには、検証機能を PMF の制御下で起動する。PMF タグの形式は
 * <RG-name, RS-name.mon> である。PMF の再起動オプションを
 * 使用するが、無限に再起動しない。その代わりに、
```

例 C–3 xfnts_monitor_start.c （続き）

```
 * interval/retry_time を RTR ファイルから取得する。
 */

int
main(int argc, char *argv[])
{
   scds_handle_t   scds_handle;
   int   rc;

   /* RGM から渡された引数を処理して、syslog を初期化する。*/
   if (scds_initialize(&scds_handle, argc, argv) != SCHA_ERR_NOERR)
{
      scds_syslog(LOG_ERR, "Failed to initialize the handle.");
      return (1);
   }

   rc = mon_start(scds_handle);

   /* scds_initialize が割り当てたすべてのメモリーを解放する。*/
   scds_close(&scds_handle);

   /* monitor_start メソッドの結果を戻す。*/
   return (rc);
}
```

xfnts_monitor_stop メソッド

このメソッドは、xfnts_probe メソッドを停止します。

例 C–4 xfnts_monitor_stop.c

```
/*
 * Copyright (c) 1998-2003 by Sun Microsystems, Inc.
 * All rights reserved.
 *
 * xfnts_monitor_stop.c -HA-XFS のモニター停止メソッド
 */

#pragma ident "@(#)xfnts_monitor_stop.c 1.9 01/01/18 SMI"

#include <rgm/libdsdev.h>
#include "xfnts.h"

/*
 * このメソッドは、HA-XFS リソース用の障害モニターを停止する。
 * この処理は PMF 経由で行われる。障害モニター用の
 * PMF タグの形式は <RG-name_RS-name.mon> である。
```

例 **C–4** xfnts_monitor_stop.c　(続き)

```
 */

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

   scds_handle_t   scds_handle;
   int    rc;

   /* RGM から渡された引数を処理して、syslog を初期化する。*/
   if (scds_initialize(&scds_handle, argc, argv) != SCHA_ERR_NOERR)
{
      scds_syslog(LOG_ERR, "Failed to initialize the handle.");
      return (1);
   }
   rc = mon_stop(scds_handle);

   /* scds_initialize が割り当てたすべてのメモリーを解放する。*/
   scds_close(&scds_handle);

   /* monitor_stop メソッドの結果を戻す。*/
   return (rc);
}
```

xfnts_probe メソッド

xfnts_probe メソッドは、アプリケーションの可用性を検査して、データサービスをフェイルオーバーするか、再起動するかを決定します。xfnts_probe メソッドは、xfnts_monitor_start コールバックメソッドによって起動され、xfnts_monitor_stop コールバックメソッドによって停止されます。

例 **C–5** xfnts_probe.c+

```
/*
 * Copyright (c) 1998-2003 by Sun Microsystems, Inc.
 * All rights reserved.
 *
 * xfnts_probe.c - HA-XFS の検査
 */

#pragma ident "@(#)xfnts_probe.c 1.26 01/01/18 SMI"

#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <signal.h>
```

例 **C–5** xfnts_probe.c+　(続き)

```
#include <sys/time.h>
#include <sys/socket.h>
#include <strings.h>
#include <rgm/libdsdev.h>
#include "xfnts.h"


/*
 * main():
 * sleep() を実行して、PMF アクションスクリプトが sleep() に割り込むのを
 * 待機する無限ループ。sleep() への割り込みが発生すると、HA-XFS 用の
 * 起動メソッドを呼び出して、再起動する。
 *
 */

int
main(int argc, char *argv[])
{
   int         timeout;
   int         port, ip, probe_result;
   scds_handle_t      scds_handle;

   hrtime_t      ht1, ht2;
   unsigned long      dt;

   scds_netaddr_list_t *netaddr;
   char   *hostname;

   if (scds_initialize(&scds_handle, argc, argv) != SCHA_ERR_NOERR)
{
      scds_syslog(LOG_ERR, "Failed to initialize the handle.");
      return (1);
   }


   /* 当該リソースに利用できる IP アドレスを取得する。*/
   if (scds_get_netaddr_list(scds_handle, &netaddr)) {
      scds_syslog(LOG_ERR,
          "No network address resource in resource group.");
      scds_close(&scds_handle);
      return (1);
   }

   /* ネットワークリソースが存在しない場合、エラーを戻す。*/
   if (netaddr == NULL || netaddr->num_netaddrs == 0) {
      scds_syslog(LOG_ERR,
          "No network address resource in resource group.");
      return (1);
   }


   /*
    * X プロパティからタイムアウト値を設定する。つまり、
```

例 **C–5** xfnts_probe.c+ *(続き)*

```
     * 当該リソース用に構成されたすべてのネットワークリソース間で
     * タイムアウト値を分割するのではなく、検証を行うたびに、
     * 各ネットワークリソースに設定されているタイムアウト値を
     * 取得することを意味する。
     */
    timeout = scds_get_ext_probe_timeout(scds_handle);

    for (;;) {

        /*
         * 連続する検証の間、Throrough_probe_interval
         * の期間、状態になる。
         */
        (void) scds_fm_sleep(scds_handle,
            scds_get_rs_thorough_probe_interval(scds_handle));

        /*
         * 使用するすべての IP アドレスを検証する。
         * 以下をループで検証する。
         * 1. 使用するすべてのネットワークリソース
         * 2. 指定されたリソースのすべての IP アドレス
         * 検証する IP アドレスごとに、障害履歴を計算する。
         */
        probe_result = 0;
        /*
         * すべてのリソースを繰り返し検証して、svc_probe() の
         * 呼び出しに使用する各 IP アドレスを取得する。
         */
        for (ip = 0; ip < netaddr->num_netaddrs; ip++) {
            /*
             * 状態を監視するホスト名とポートを取得する。
             */
            hostname = netaddr->netaddrs[ip].hostname;
            port = netaddr->netaddrs[ip].port_proto.port;
            /*
             * HA-XFS がサポートするポートは 1 つだけなので、
             * ポート値はポートの配列の最初の
             * エントリから取得する。
             */
            ht1 = gethrtime(); /* 検証開始時間を取得する。*/
            scds_syslog(LOG_INFO, "Probing the service on "
                "port: %d.", port);

            probe_result =
            svc_probe(scds_handle, hostname, port, timeout);

            /*
             * サービス検証履歴を更新し、
             * 必要に応じて、アクションを実行する。
             * 検証終了時間を取得する。
             */
            ht2 = gethrtime();
```

例 **C–5** xfnts_probe.c+　（続き）

```
            /* ミリ秒に変換する。*/
            dt = (ulong_t)((ht2 - ht1) / 1e6);

            /*
             * 障害の履歴を計算し、
             * 必要に応じて、アクションを実行する。
             */
            (void) scds_fm_action(scds_handle,
                probe_result, (long)dt);
        }   /* ネットワークリソースごと */
    }    /* 検証を永続的に繰り返す。*/
}
```

xfnts_start メソッド

データサービスリソースを含むリソースグループがオンラインになったとき、あるいは、リソースが有効になったとき、RGM はそのクラスタノード上で Start メソッドを呼び出します。xfnts_start メソッドはそのノード上で xfs デーモンを起動します。

例 **C–6** xfnts_start.c

```
/*
 * Copyright (c) 1998-2003 by Sun Microsystems, Inc.
 * All rights reserved.
 *
 * xfnts_svc_start.c - HA-XFS の起動メソッド
 */

#pragma ident "@(#)xfnts_svc_start.c 1.13 01/01/18 SMI"

#include <rgm/libdsdev.h>
#include "xfnts.h"

/*
 * HA-XFS 用の起動メソッド。リソース設定に対していくつかの
 * 健全性検査を行なったあと、アクションスクリプトを使用して HA-XFS を
 * PMF の制御下で起動する。
 */

int
main(int argc, char *argv[])
{
   scds_handle_t   scds_handle;
   int rc;

   /*
```

例 **C–6** xfnts_start.c　(続き)

```
    * RGM から渡された引数を処理して、syslog を初期化する。
    */

   if (scds_initialize(&scds_handle, argc, argv) != SCHA_ERR_NOERR)
{
      scds_syslog(LOG_ERR, "Failed to initialize the handle.");
      return (1);
   }

   /* 構成の妥当性を検査する。エラーがあれば戻る。*/
   rc = svc_validate(scds_handle);
   if (rc != 0) {
      scds_syslog(LOG_ERR,
          "Failed to validate configuration.");
      return (rc);
   }

   /* データサービスを起動する。失敗した場合、エラーで戻る。*/
   rc = svc_start(scds_handle);
   if (rc != 0) {
      goto finished;
   }

   /* サービスが完全に起動するまで待つ。*/
   scds_syslog_debug(DBG_LEVEL_HIGH,
       "Calling svc_wait to verify that service has started.");

   rc = svc_wait(scds_handle);

   scds_syslog_debug(DBG_LEVEL_HIGH,
       "Returned from svc_wait");

   if (rc == 0) {
      scds_syslog(LOG_INFO, "Successfully started the service.");
   } else {
      scds_syslog(LOG_ERR, "Failed to start the service.");
   }


finished:
   /* 割り当てられた環境リソースを解放する。*/
   scds_close(&scds_handle);

   return (rc);
}
```

xfnts_stop メソッド

HA-XFS リソースを含むリソースグループがクラスタのノード上でオフラインになったとき、あるいは、リソースが無効になったとき、RGM はそのクラスタノード上で Stop メソッドを呼び出します。xfnts_stop メソッドはそのノード上で xfs デーモンを停止します。

例 **C–7** xfnts_stop.c

```
/*
 * Copyright (c) 1998-2003 by Sun Microsystems, Inc.
 * All rights reserved.
 *
 * xfnts_svc_stop.c - HA-XFS の停止メソッド
 */

#pragma ident "@(#)xfnts_svc_stop.c 1.10 01/01/18 SMI"

#include <rgm/libdsdev.h>
#include "xfnts.h"

/*
 * PMF を使用して HA-XFS プロセスを停止する。
 */

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

   scds_handle_t   scds_handle;
   int      rc;

   /* RGM から渡された引数を処理して、syslog を初期化する。*/
   if (scds_initialize(&scds_handle, argc, argv) != SCHA_ERR_NOERR)
{
      scds_syslog(LOG_ERR, "Failed to initialize the handle.");
      return (1);
   }

   rc = svc_stop(scds_handle);

   /* scds_initialize が割り当てたすべてのメモリーを解放する。*/
   scds_close(&scds_handle);

   /* svc_stop メソッドの結果を戻す。*/
   return (rc);
}
```

xfnts_update メソッド

リソースのプロパティが変更されたとき、RGM は Update メソッドを呼び出し、動作中のリソースに通知します。RGM は、管理アクションがリソースまたはそのリソースグループのプロパティの設定に成功したあとに、Update を呼び出します。

例 **C–8** xfnts_update.c

```
#pragma ident "@(#)xfnts_update.c  1.10     01/01/18 SMI"

/*
 * Copyright (c) 1998-2003 by Sun Microsystems, Inc.
 * All rights reserved.
 *
 * xfnts_update.c - HA-XFS の更新メソッド
 */

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <rgm/libdsdev.h>

/*
 * リソースのプロパティが更新された可能性がある。
 * このような更新可能なプロパティはすべて障害モニターに関連するもので
 * あるため、障害モニターを再起動する必要がある。
 */

int
main(int argc, char *argv[])
{
   scds_handle_t   scds_handle;
   scha_err_t   result;

   /* RGM から渡された引数を処理して、syslog を初期化する。*/
   if (scds_initialize(&scds_handle, argc, argv) != SCHA_ERR_NOERR)
{
      scds_syslog(LOG_ERR, "Failed to initialize the handle.");
      return (1);
   }

   /*
    * 障害モニターがすでに動作していることを検査し、
    * 動作している場合、障害モニターを停止および再起動する。
    * scds_pmf_restart_fm() への 2 番目のパラメータは、再起動する
    * 必要がある障害モニターのインスタンスを一意に識別する。
    */

   scds_syslog(LOG_INFO, "Restarting the fault monitor.");
   result = scds_pmf_restart_fm(scds_handle, 0);
   if (result != SCHA_ERR_NOERR) {
```

例 **C–8** xfnts_update.c　(続き)

```
        scds_syslog(LOG_ERR,
            "Failed to restart fault monitor.");
        /* initialize が割り当てたすべてのメモリーを解放する。*/
        scds_close(&scds_handle);
        return (1);
    }

    scds_syslog(LOG_INFO,
        "Completed successfully.");

    /* initialize が割り当てたすべてのメモリーを解放する。*/
    scds_close(&scds_handle);

    return (0);
}
```

xfnts_validate メソッドのコードリスト

xfnts_validate メソッドは、Confdir_list プロパティが示すディレクトリの存在を確認します。RGM がこのメソッドを呼び出すのは、クラスタ管理者がデータサービスを作成したときと、データサービスのプロパティを更新したときです。障害モニターがデータサービスを新しいノードにフェイルオーバーしたときは、Monitor_check メソッドは常にこのメソッドを呼び出します。

例 **C–9** xfnts_validate.c

```
/*
 * Copyright (c) 1998-2003 by Sun Microsystems, Inc.
 * All rights reserved.
 *
 * xfnts_validate.c - HA-XFS の検証メソッド
 */

#pragma ident "@(#)xfnts_validate.c 1.9 01/01/18 SMI"

#include <rgm/libdsdev.h>
#include "xfnts.h"

/*
 * プロパティが正しく設定されていることを確認する。
 */

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

例 **C–9** xfnts_validate.c　(続き)

```
{
   scds_handle_t   scds_handle;
   int   rc;

   /* RGM から渡された引数を処理して、syslog を初期化する。*/
   if (scds_initialize(&scds_handle, argc, argv) != SCHA_ERR_NOERR)
{
      scds_syslog(LOG_ERR, "Failed to initialize the handle.");
      return (1);
   }
   rc = svc_validate(scds_handle);

   /* scds_initialize が割り当てたすべてのメモリーを解放する。*/
   scds_close(&scds_handle);

   /* 検証メソッドの結果を戻す。*/
   return (rc);

}
```

付録 D

RGM の有効な名前と値

この付録では、RGM の名前と値に対する有効な文字の要件を示します。

RGM の有効な名前

RGM の名前は次の 5 つのカテゴリに分類されます。

- リソースグループ名
- リソースタイプ名
- リソース名
- プロパティ名
- 列挙型リテラル名

リソースタイプ名を除いて、すべての名前は次の規則に従う必要があります。

- ASCII であること。
- 英字で始まること。
- 名前に使用できる文字は、英字の大文字と小文字、数字、ハイフン (-)、下線 (_)。
- 255 文字を超えないこと。

リソースタイプ名は、簡単な名前 (RTR ファイルの `Resource_type` プロパティで指定) または完全な名前 (RTR ファイルの `Vendor_id` と `Resource_type` で指定) のどちらでもかまいません。これら両方のプロパティを指定するとき、RGM は、`Vendor_id` と `Resource_type` の間にピリオドを挿入して、完全な名前を作成します。たとえば、`Vendor_id=SUNW` で、`Resource_type=sample` の場合、完全な名前は `SUNW.sample` です。これは、RGM の名前において、ピリオドが有効な文字である場合だけです。

RGM の値

RGM の値は、プロパティ値と記述値という 2 つのカテゴリに分類されます。両方とも、次のような同じ規則が適用されます。

- 値は ASCII であること。
- 値の最大長は 4M - 1 バイト (つまり、4,194,303 バイト) であること。
- NULL、 改行文字、コンマ、セミコロンは、値に使用できない。

付録 E

非クラスタ対応のアプリケーションの要件

通常、非クラスタ対応のアプリケーションの高可用性 (HA) を実現するには、特定の要件を満たす必要があります。そのための要件の一覧が、29 ページの「アプリケーションの適合性の分析」に示されています。この付録では、それらの要件のうち、特定のものについて詳細に説明します。

アプリケーションの高可用性を実現するには、そのリソースをリソースグループで構成します。アプリケーションのデータは、高可用性の広域ファイルシステムに格納されます。したがって、1 つのサーバーが異常終了しても、正常に動作しているサーバーがデータにアクセスできます。『*Sun Cluster 3.1 10/03* の概念』のクラスタファイルシステムに関する情報を参照してください。

ネットワーク上のクライアントがネットワークにアクセスする場合、論理ネットワーク IP アドレスは、データサービスリソースと同じリソースグループにある論理ホスト名リソースで構成されます。データサービスリソースとネットワークアドレスリソースは共にフェイルオーバーします。この場合、データサービスのネットワーククライアントは新しいホスト上のデータサービスリソースにアクセスします。

多重ホストデータ

高可用性の広域ファイルシステムのディスクセットは多重ホスト化されているため、ある物理ホストがクラッシュしても、正常に動作している物理ホストの 1 つがディスクにアクセスできます。アプリケーションの高可用性を実現するには、そのデータが高可用性であること、つまり、そのデータが広域 HA ファイルシステムに格納されていることが必要です。

広域ファイルシステムは、独立したものであるように作成されたディスクグループにマウントされます。ユーザーは、あるディスクグループをマウントされた広域ファイルシステムとして使用し、別のディスクグループをデータサービス (HA Oracle など) で使用する raw デバイスとして使用することもできます。

アプリケーションは、データファイルの位置を示すコマンド行スイッチまたは構成ファイルを持っていることがあります。アプリケーションが固定されたパス名を使用する場合は、アプリケーションのコードを変更せずに、このパス名を広域ファイルシステム内のファイルの位置を指すシンボリックリンクに変更できます。シンボリックリンクの詳しい使用方法については、316 ページの「多重ホストデータを配置するためのシンボリックリンクの使用」を参照してください。

最悪の場合は、実際のデータの位置を示すような何らかの機構を使用するように、アプリケーションのソースコードを変更する必要があります。この作業は、コマンド行スイッチを追加することにより行うことができます。

Sun Cluster は、ボリューム管理ソフトウェアに構成されている UNIX UFS ファイルシステムと HA の raw デバイスの使用をサポートします。インストールおよび構成するとき、システム管理者はどのディスクリソースを UFS ファイルシステムまたは raw デバイス用に使用するかを指定する必要があります。通常、raw デバイスを使用するのは、データベースサーバーとマルチメディアサーバーだけです。

多重ホストデータを配置するためのシンボリックリンクの使用

アプリケーションの中には、そのデータファイルへのパス名が固定されており、しかも、固定されたパス名を変更する機構がないものがあります。このような場合に、シンボリックリンクを使用すればアプリケーションのコードを変更せずに、済ませられる場合もあります。

たとえば、アプリケーションがそのデータファイルに固定されたパス名 `/etc/mydatafile` を指定すると仮定します。このパスは、論理ホストのファイルシステムの 1 つにあるファイルを示す値を持つシンボリックリンクに変更できます。たとえば、`/global/phys-schost-2/mydatafile` へのシンボリックリンクに変更できます。

ただし、データファイルの名前を内容とともに変更するアプリケーション (または、その管理手順) の場合には、シンボリックリンクをこのように使用すると問題が生じる可能性があります。たとえば、アプリケーションが更新を実行するとき、まず、新しい一時ファイル `/etc/mydatafile.new` を作成すると仮定します。次に、このデータベースは `rename(2)` システムコール (または `mv(1)` プログラム) を使用し、この一時ファイルの名前を実際のファイルの名前に変更します。一時ファイルを作成し、その名前を実際のファイルの名前に変更することにより、データサービスは、そのデータファイルの内容が常に適切であるようにします。

この時、`rename(2)` の操作はシンボリックリンクを破壊します。このため、`/etc/mydatafile` という名前は通常ファイルとなり、クラスタの広域ファイルシステムの中ではなく、`/etc` ディレクトリと同じファイルシステムの中に存在することになります。`/etc` ファイルシステムは各ホスト専用であるため、テイクオーバーまたはスイッチオーバー後はデータが利用できなくなります。

このような状況の根本的な問題は、既存のアプリケーションがシンボリックリンクに気付かない、つまり、シンボリックリンクを考慮するように作成されていないことにあります。シンボリックリンクを使用し、データアクセスを論理ホストのファイルシステムにリダイレクトするには、アプリケーション実装がシンボリックリンクを消去しないように動作する必要があります。したがって、シンボリックリンクは、論理ホストのファイルシステムへのデータの配置に関する問題をすべて解決できるわけではありません。

ホスト名

データサービス開発者は、データサービスが動作しているサーバーのホスト名を、データサービスが知る必要があるかどうかを判断する必要があります。知る必要があると判断した場合は、物理ホストではなく、論理ホストのホスト名 (つまり、アプリケーションリソースと同じリソースグループ内にある論理ホスト名リソース内に構成されているホスト名) を使用するようにデータサービスを変更する必要があります。

データサービスのクライアントサーバープロトコルでは、サーバーが自分のホスト名をクライアントへのメッセージの一部としてクライアントに戻すことがあります。このようなプロトコルでは、クライアントは戻されたホスト名をサーバーに接続するときのホスト名として使用できます。戻されたホスト名をテイクオーバーやスイッチオーバーが発生した後にも使用できるようにするには、物理ホストではなく、リソースグループの論理ホスト名を使用する必要があります。物理ホスト名を使用している場合は、論理ホスト名をクライアントに戻すようにデータサービスのコードを変更する必要があります。

多重ホームホスト

多重ホームホストとは、複数のパブリックネットワーク上にあるホストのことです。このようなホストは複数 (つまり、ネットワークごとに 1 つ) のホスト名/IP アドレスのペアを持ちます。Sun Cluster は、1 つのホストが複数のネットワーク上に存在できるように設計されています。1 つのホストが単一のネットワーク上に存在することも可能ですが、このような場合は「多重ホームホスト」とは呼びません。物理ホスト名が複数のホスト名/IP アドレスのペアを持つように、各リソースグループも複数 (つまり、パブリックネットワークごとに 1 つ) のホスト名/IP アドレスのペアを持ちます。Sun Cluster がリソースグループをある物理ホストから別の物理ホストに移動するとき、そのリソースグループに対するホスト名 / IP アドレスのペアもすべて移動します。

リソースグループに対するホスト名/IP アドレスのペアは、リソースグループに含まれる論理ホスト名リソースとして構成されます。このようなネットワークアドレスリソースは、システム管理者がリソースグループを作成および構成するときに指定します。Sun Cluster データサービス API は、このようなホスト名 / IP アドレスのペアを照会する機能を持っています。

Solaris 環境用に書かれているほとんどの市販のデータサービスデーモンは、多重ホームホストを適切に処理できます。ネットワーク通信を行うとき、多くのデータサービスは Solaris のワイルドカードアドレス `INADDR_ANY` にバインドします。すると、`INADDR_ANY` は、すべてのネットワークインタフェースのすべての IP アドレスを自動的に処理します。`INADDR_ANY` は、現在マシンに構成されているすべての IP アドレスに効率的にバインドします。一般的に、`INADDR_ANY` を使用するデータサービスデーモンは、変更しなくても、Sun Cluster 論理ネットワークアドレスを処理できます。

`INADDR_ANY` へのバインドと特定の IP アドレスへのバインド

Sun Cluster の論理ネットワークアドレスの概念では、多重ホーム化されていない環境でも、マシンは複数の IP アドレスを持つことができます。つまり、独自の物理ホストの IP アドレスを 1 つだけ持ち、さらに、現在マスターしているネットワークアドレス (論理ホスト名) リソースごとに 1 つの IP アドレスを持ちます。ネットワークアドレスリソースのマスターになるとき、マシンは動的に追加の IP アドレスを獲得します。ネットワークアドレスリソースのマスターを終了するとき、マシンは動的に IP アドレスを放棄します。

データサービスの中には、`INADDR_ANY` にバインドしていると、Sun Cluster 環境で適切に動作しないもあります。このようなデータサービスは、リソースグループのマスターになるとき、またマスターをやめるときに、バインドしている IP アドレスのセットを動的に変更する必要があります。このようなデータサービスが再バインドする方法の 1 つが、起動メソッドと停止メソッドを使用し、データサービスのデーモンを強制終了および再起動するという方法です。

`Network_resources_used` リソースプロパティを使用すると、エンドユーザーは、アプリケーションリソースをバインドすべきネットワークアドレスリソースを構成できます。この機能が必要なリソースタイプの場合、そのリソースタイプの RTR ファイルで `Network_resources_used` プロパティを宣言する必要があります。

リソースグループをオンラインまたはオフラインにするとき、Sun Cluster は、データサービスリソースメソッドを呼び出す順番に従って、ネットワークアドレスの取り付け (plumb)、 取り外し (unplumb)、「起動」または「停止」への構成を行います。詳細は、45 ページの「`Start` および `Stop` メソッドを使用するかどうかの決定」を参照してください。

データサービスは、Stop メソッドが戻るまでに、リソースグループのネットワークアドレスの使用を終了している必要があります。同様に、データサービスは、Start メソッドが戻るまでに、リソースグループのネットワークアドレスの使用を開始している必要があります。

データサービスが、個々の IP アドレスではなく、INADDR_ANY にバインドする場合、データサービスリソースメソッドが呼び出される順番とネットワークアドレスメソッドが呼び出される順番には重要な関係があります。

データサービスの停止メソッドと起動メソッドでデータサービスのデーモンを終了および再起動する場合、データサービスは適切な時間にネットワークアドレスの使用を停止および開始します。

クライアントの再試行

ネットワーククライアントから見ると、テイクオーバーやスイッチオーバーは、論理ホストに障害が発生し、高速再起動しているように見えます。したがって、クライアントアプリケーションとクライアントサーバープロトコルは、このような場合に何回か再試行するように構成されていることが理想的です。すでに、単一サーバーの障害と高速再起動を処理するように構成されているアプリケーションとプロトコルは、上記のような場合も、リソースグループのテイクオーバーやスイッチオーバーとして処理します。無限に再試行するようなアプリケーションもあります。また、何回も再試行していることをユーザーに通知し、さらに継続するかどうかをユーザーにたずねるような、より洗練されたアプリケーションもあります。

付録 F

CRNP のドキュメントタイプ定義

この付録では、Cluster Reconfiguration Notification Protocol (CRNP) のドキュメントタイプ定義 (DTD) を示します。

SC_CALLBACK_REG XML DTD

注 – SC_CALLBACK_REG と SC_EVENT の両方で使用する NVPAIR データ構造は、1 度だけ定義します。

```
<!— SC_CALLBACK_REG XML format specification
        Copyright 2001-2003 Sun Microsystems, Inc. All rights reserved.
        Use is subject to license terms.

        Intended Use:

A client of the Cluster Reconfiguration Notification Protocol should use this xml format
to register initially with the service, to subsequently register for more events, to
subsequently remove registration of some events, or to remove itself from the service
entirely.

A client is uniquely identified by its callback IP and port.  The port is defined in the
SC_CALLBACK_REG element, and the IP is taken as the source IP of the registration
connection.  The final attribute of the root SC_CALLBACK_REG element is either an
ADD_CLIENT, ADD_EVENTS, REMOVE_CLIENT, or REMOVE_EVENTS, depending on which form of the
message the client is using.

The SC_CALLBACK_REG contains 0 or more SC_EVENT_REG sub-elements.

One SC_EVENT_REG is the specification for one event type.  A client may specify only the
CLASS (an attribute of the SC_EVENT_REG element), or may specify a SUBCLASS (an optional
attribute) for further granularity.  Also, the SC_EVENT_REG has as subelements 0 or more
```

```
NVPAIRs, which can be used to further specify the event.

Thus, the client can specify events to whatever granularity it wants.  Note that a client
cannot both register for and unregister for events in the same message.  However a client
can subscribe to the service and sign up for events in the same message.

Note on versioning: the VERSION attribute of each root element is marked "fixed", which
means that all message adhering to these DTDs must have the version value specified.  If a
new version of the protocol is created, the revised DTDs will have a new value for this
fixed" VERSION attribute, such that all message adhering to the new version must have the
new version number.
—>

<!— SC_CALLBACK_REG definition

The root element of the XML document is a registration message.  A registration message
consists of the callback port and the protocol version as attributes, and either an
ADD_CLIENT, ADD_EVENTS, REMOVE_CLIENT, or REMOVE_EVENTS attribute, specifying the
registration type.  The ADD_CLIENT, ADD_EVENTS, and REMOVE_EVENTS types should have one or
more SC_EVENT_REG subelements.  The REMOVE_CLIENT should not specify an SC_EVENT_REG
subelement.
        ATTRIBUTES:
               VERSION        The CRNP protocol version of the message.
               PORT           The callback port.
               REG_TYPE       The type of registration.  One of:
                       ADD_CLIENT, ADD_EVENTS, REMOVE_CLIENT, REMOVE_EVENTS

        CONTENTS:
SUBELEMENTS: SC_EVENT_REG (0 or more)
—>
<!ELEMENT SC_CALLBACK_REG (SC_EVENT_REG*)>
<!ATTLIST SC_CALLBACK_REG
       VERSION        NMTOKEN                                                  #FIXED
       PORT           NMTOKEN                                                  #REQUIRED
       REG_TYPE       (ADD_CLIENT|ADD_EVENTS|REMOVE_CLIENT|REMOVE_EVENTS)    #REQUIRED

>
<!— SC_EVENT_REG definition

The SC_EVENT_REG defines an event for which the client is either registering or
unregistering interest in receiving event notifications.  The registration can be for any
level of granularity, from only event class down to specific name/value pairs that must be
present.  Thus, the only required attribute is the CLASS.  The SUBCLASS attribute, and the
NVPAIRS sub-elements are optional, for higher granularity.

Registrations that specify name/value pairs are registering interest in notification of
messages from the class/subclass specified with ALL name/value pairs present.
Unregistrations that specify name/value pairs are unregistering interest in notifications
that have EXACTLY those name/value pairs in granularity previously specified.
Unregistrations that do not specify name/value pairs unregister interest in ALL event
notifications of the specified class/subclass.

        ATTRIBUTES:
               CLASS:        The event class for which this element is registering
                             or unregistering interest.
```

```
                SUBCLASS:     The subclass of the event (optional).

        CONTENTS:
                SUBELEMENTS: 0 or more NVPAIRs.
—>

<!ELEMENT SC_EVENT_REG (NVPAIR*)>
<!ATTLIST SC_EVENT_REG
       CLASS          CDATA                #REQUIRED
       SUBCLASS       CDATA                #IMPLIED
>
```

NVPAIR XML DTD

```
<!— NVPAIR XML format specification

        Copyright 2001-2003 Sun Microsystems, Inc. All rights reserved.
        Use is subject to license terms.

        Intended Use:
              An nvpair element is meant to be used in an SC_EVENT or SC_CALLBACK_REG
              element.
—>

<!— NVPAIR definition

       The NVPAIR is a name/value pair to represent arbitrary name/value combinations.
       It is intended to be a direct, generic, translation of the Solaris nvpair_t
       structure used by the sysevent framework.  However, there is no type information
       associated with the name or the value (they are both arbitrary text) in this xml
       element.

       The NVPAIR consists simply of one NAME element and one or more VALUE elements.
       One VALUE element represents a scalar value, while multiple represent an array
       VALUE.

       ATTRIBUTES:

       CONTENTS:
              SUBELEMENTS: NAME(1), VALUE(1 or more)
—>

<!ELEMENT NVPAIR (NAME,VALUE+)>
<!— NAME definition

        The NAME is simply an arbitrary length string.

        ATTRIBUTES:
```

```
        CONTENTS:
                Arbitrary text data.  Should be wrapped with <![CDATA[...]]> to prevent XML
                parsing inside.
—>
<!ELEMENT NAME (#PCDATA)>

<!— VALUE definition
        The VALUE is simply an arbitrary length string.

        ATTRIBUTES:

        CONTENTS:
                Arbitrary text data.  Should be wrapped with <![CDATA[...]]> to prevent XML
                parsing inside.
—>

<!ELEMENT VALUE (#PCDATA)>
```

SC_REPLY XML DTD

```
<!— SC_REPLY XML format specification

        Copyright 2001-2003 Sun Microsystems, Inc. All rights reserved.
        Use is subject to license terms.
—>

<!— SC_REPLY definition

        The root element of the XML document represents a reply to a message.  The reply
        contains a status code and a status message.

        ATTRIBUTES:
                VERSION:        The CRNP protocol version of the message.
                STATUS_CODE:    The return code for the message.  One of the
                                following: OK, RETRY, LOW_RESOURCES, SYSTEM_ERROR, FAIL,
                                MALFORMED, INVALID_XML, VERSION_TOO_HIGH, or
                                VERSION_TOO_LOW.

                CONTENTS:
                        SUBELEMENTS: SC_STATUS_MSG(1)
—>

<!ELEMENT SC_REPLY (SC_STATUS_MSG)>
<!ATTLIST SC_REPLY
        VERSION         NMTOKEN                                         #FIXED   "1.0"
        STATUS_CODE     OK|RETRY|LOW_RESOURCE|SYSTEM_ERROR|FAIL|MALFORMED|INVALID,\
                        VERSION_TOO_HIGH, VERSION_TOO_LOW) #REQUIRED
>
```

```
<!— SC_STATUS_MSG definition
        The SC_STATUS_MSG is simply an arbitrary text string elaborating on the status
        code.  Should be wrapped with <![CDATA[...]]> to prevent XML parsing inside.

        ATTRIBUTES:

        CONTENTS:
                Arbitrary string.
—>

<!ELEMENT SC_STATUS_MSG (#PCDATA)>
```

SC_EVENT XML DTD

注 – SC_CALLBACK_REG と SC_EVENT の両方で使用する NVPAIR データ構造は、1 度だけ定義します。

```
<!— SC_EVENT XML format specification

        Copyright 2001-2003 Sun Microsystems, Inc. All rights reserved.
        Use is subject to license terms.

        The root element of the XML document is intended to be a direct, generic,
        translation of the Solaris sysseventd message format.  It has attributes to
        represent the class, subclass, vendor, and publisher, and contains any number of
        NVPAIR elements.

        ATTRIBUTES:
                VERSION:        The CRNP protocol version of the message.
                CLASS:          The sysevent class of the event
                SUBCLASS:       The subclass of the event
                VENDOR:         The vendor associated with the event
                PUBLISHER:      The publisher of the event
        CONTENTS:
                SUBELEMENTS: NVPAIR (0 or more)
—>

<!ELEMENT SC_EVENT (NVPAIR*)>
<!ATTLIST SC_EVENT
        VERSION         NMTOKEN         #FIXED "1.0"
        CLASS           CDATA           #REQUIRED
        SUBCLASS        CDATA           #REQUIRED
        VENDOR          CDATA           #REQUIRED
        PUBLISHER       CDATA           #REQUIRED
>
```

付録 G

CrnpClient.java アプリケーション

この付録では、CrnpClient.java アプリケーションの完全なコードを示します (詳細は、第 12 章を参照)。

Contents of CrnpClient.java

```
/*
 * CrnpClient.java
 * ================
 *
 * XML 解析についての注意:
 *
 * このプログラムは、Sun Java Architecture for XML Processing (JAXP) API を
 * 使用しています。API ドキュメントや利用についての情報は、
 * http://java.sun.com/xml/jaxp/index.html を参照してください。
 *
 * このプログラムは、Java 1.3.1 以降を対象に作成されています。
 *
 * プログラムの概要:
 *
 * このプログラムのメインスレッドは、CrnpClient オブジェクトを作成し、
 * ユーザーがデモを終了するのを待機し、CrnpClient オブジェクトで
 * shutdown を呼び出し、最後にプログラムを終了します。
 *
 * CrnpClient コンストラクタは、EventReceptionThread オブジェクトを作成し、
 * (コマンド行で指定されたホストとポートを使用して) CRNP サーバーに対して
 * 接続を開き、(コマンド行の指定にもとづいて) 登録メッセージを作成し、登録
 * メッセージを送信し、応答の読み取りと解析を行います。
 *
 * EventReceptionThread は、このプログラムが動作するマシンのホスト名と
 * コマンド行に指定されるポートにバインドされる待機ソケットを作成します。
 * EventReceptionThread は、イベントコールバックの着信を待機し、受信した
 * ソケットストリームから XML ドキュメントを構築し、これを CrnpClient
```

```
 * オブジェクトに返して処理を行わせます。
 *
 * CrnpClient 内の shutdown メソッドは、単に登録解除用の
 * (REMOVE_CLIENT) SC_CALLBACK_REG メッセージを crnp サーバーへ送信するだけ
 * です。
 *
 * エラー処理についての注意：説明を簡潔にするため、このプログラムはほとんどの
 * エラーに対して単に終了するだけですが、実際のアプリケーションではさまざまな
 * 方法でエラー処理がなされます (適宜再試行するなど)。
 */

// JAXP パッケージ
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import org.w3c.dom.*;

// 標準パッケージ
import java.net.*;
import java.io.*;
import java.util.*;

/*
 * class CrnpClient
 * ----------------
 * 上記のファイルヘッダーコメントを参照。
 */
class CrnpClient
{
    /*
     * main
     * ----
     * 実行のエントリポイント main は、コマンド行引数の数を検証し、
     * すべての作業を行う CrnpClient インスタンスを作成する。
     */
    public static void main(String []args)
    {
        InetAddress regIp = null;
        int regPort = 0, localPort = 0;

        /* コマンド行引数の数を検証する */
        if (args.length < 4) {
            System.out.println(
                "Usage: java CrnpClient crnpHost crnpPort "
                + "localPort (-ac | -ae | -re) "
                + "[(M | A | RG=name | R=name) [...]]");
            System.exit(1);
        }


        /*
         * コマンド行には crnp サーバーの IP とポート、待機する
```

```
     * ローカルポート、登録タイプを示す引数が入る。
     */
    try {
        regIp = InetAddress.getByName(args[0]);
        regPort = (new Integer(args[1])).intValue();
        localPort = (new Integer(args[2])).intValue();
    } catch (UnknownHostException e) {
        System.out.println(e);
        System.exit(1);
    }

    // CrnpClient を作成する。
    CrnpClient client = new CrnpClient(regIp, regPort, localPort,
        args);

    // ユーザーがプログラムを終了したくなるまで待機する。
    System.out.println("Hit return to terminate demo...");

    // ユーザーが何か入力するまで読み取りはブロックする。
    try {
        System.in.read();
    } catch (IOException e) {
        System.out.println(e.toString());
    }

    // クライアントを停止する。
    client.shutdown();
    System.exit(0);
}

/*
 * ======================
 * public メソッド
 * ======================
 */

/*
 * CrnpClient コンストラクタ
 * -------------------------
 * crnp サーバーとの通信方法を知るためにコマンド行引数を解析し、
 * イベント受信スレッドを作成し、このスレッドの実行を開始し、XML
 * DocumentBuilderFactory オブジェクトを作成し、最後に crnp
 * サーバーにコールバックの登録を行う。
 */
public CrnpClient(InetAddress regIpIn, int regPortIn, int localPortIn,
    String []clArgs)
{
    try {

        regIp = regIpIn;
        regPort = regPortIn;
        localPort = localPortIn;
        regs = clArgs;

        /*
```

```
             * xml 処理用のドキュメントビルダー
             * ファクトリを設定する。
             */
            setupXmlProcessing();

            /*
             * ServerSocket を作成してこれをローカル IP と
             * ポートにバインドする EventReceptionThread を
             * 作成する。
             */
            createEvtRecepThr();

            /*
             * crnp サーバーに登録する。
             */
            registerCallbacks();

        } catch (Exception e) {
            System.out.println(e.toString());
            System.exit(1);
        }
    }

    /*
     * processEvent
     * ---------------
     * CrnpClient (イベントコールバックを受信する際に
     * EventReceptionThread によって使用される) にコールバックする。
     */
    public void processEvent(Event event)
    {
        /*
         * ここでは、説明の都合上、単純にイベントを System.out
         * に出力。実際のアプリケーションでは、通常、イベント
         * をなんらかの方法で使用する。
         */
        event.print(System.out);
    }

    /*
     * shutdown
     * -------------
     * CRNP サーバーに対する登録を解除する。
     */
    public void shutdown()
    {
        try {
            /* サーバーに登録解除メッセージを送信する */
            unregister();
        } catch (Exception e) {
            System.out.println(e);
            System.exit(1);
        }
    }
```

```
/*
 * ======================
 * private ヘルパーメソッド
 * ======================
 */

/*
 * setupXmlProcessing
 * --------------------
 * xml 応答と xml イベントを解析するためにドキュメント
 * ビルダーファクトリを作成する。
 */
private void setupXmlProcessing() throws Exception
{
    dbf = DocumentBuilderFactory.newInstance();

    // わざわざ検証する必要はない。
    dbf.setValidating(false);
    dbf.setExpandEntityReferences(false);

    // コメントと空白文字は無視したい。
    dbf.setIgnoringComments(true);
    dbf.setIgnoringElementContentWhitespace(true);

    // CDATA セクションを TEXT ノードに結合する。
    dbf.setCoalescing(true);
}

/*
 * createEvtRecepThr
 * -------------------
 * 新しい EventReceptionThread オブジェクトを作成し、待機
 * ソケットがバインドされる IP とポートを保存し、スレッドの
 * 実行を開始する。
 */
private void createEvtRecepThr() throws Exception
{
    /* スレッドオブジェクトを作成する */
    evtThr = new EventReceptionThread(this);

    /*
     * イベント配信コールバックを待機し始めるために
     * スレッドの実行を開始する。
     */
    evtThr.start();
}


/*
 * registerCallbacks
 * ------------------
 * crnp サーバーに対するソケット接続を作成し、
 * イベント登録メッセージを送信する。
 */
```

```
private void registerCallbacks() throws Exception
{
    System.out.println("About to register");

    /*
     * crnp サーバーの登録 IP / ポートに接続されたソケット
     * を作成し、登録情報を送信する。
     */
    Socket sock = new Socket(regIp, regPort);
    String xmlStr = createRegistrationString();
    PrintStream ps = new PrintStream(sock.getOutputStream());
    ps.print(xmlStr);

    /*
     * 応答を読み取る。
     */
    readRegistrationReply(sock.getInputStream());

    /*
     * ソケット接続を閉じる。
     */
    sock.close();
}

/*
 * unregister
 * ----------
 * registerCallbacks の場合と同様に、crnp サーバーに対する
 * ソケット接続を作成し、登録解除メッセージを送信し、
 * サーバーからの応答を待機し、ソケットを閉じる。
 */
private void unregister() throws Exception
{
    System.out.println("About to unregister");

    /*
     * crnp サーバーの登録 IP / ポートに接続された
     * ソケットを作成し、登録解除情報を送信する。
     */
    Socket sock = new Socket(regIp, regPort);
    String xmlStr = createUnregistrationString();
    PrintStream ps = new PrintStream(sock.getOutputStream());
    ps.print(xmlStr);

    /*
     * 応答を読み取る。
     */
    readRegistrationReply(sock.getInputStream());

    /*
     * ソケット接続を閉じる。
     */
    sock.close();
}
```

```
/*
 * createRegistrationString
 * ------------------
 * このプログラムのコマンド行引数にもとづいて CallbackReg
 * オブジェクトを作成し、CallbackReg オブジェクトから XML
 * 文字列を取得する。
 */
private String createRegistrationString() throws Exception
{
    /*
     * 実際の CallbackReg クラスを作成し、ポートを設定する。
     */
    CallbackReg cbReg = new CallbackReg();
    cbReg.setPort("" + localPort);

    // 登録タイプを設定する。
    if (regs[3].equals("-ac")) {
        cbReg.setRegType(CallbackReg.ADD_CLIENT);
    } else if (regs[3].equals("-ae")) {
        cbReg.setRegType(CallbackReg.ADD_EVENTS);
    } else if (regs[3].equals("-re")) {
        cbReg.setRegType(CallbackReg.REMOVE_EVENTS);
    } else {
        System.out.println("Invalid reg type: " + regs[3]);
        System.exit(1);
    }

    // イベントを追加する。
    for (int i = 4; i < regs.length; i++) {
        if (regs[i].equals("M")) {
            cbReg.addRegEvent(
                createMembershipEvent());
        } else if (regs[i].equals("A")) {
            cbReg.addRegEvent(
                createAllEvent());
        } else if (regs[i].substring(0,2).equals("RG")) {
            cbReg.addRegEvent(createRgEvent(
                regs[i].substring(3)));
        } else if (regs[i].substring(0,1).equals("R")) {
            cbReg.addRegEvent(createREvent(
                regs[i].substring(2)));
        }
    }

    String xmlStr = cbReg.convertToXml();
    System.out.println(xmlStr);
    return (xmlStr);
}

/*
 * createAllEvent
 * ----------------
 * クラス EC_Cluster を使用して (サブクラスは使用しない)
 * XML 登録イベントを作成する。
 */
```

```
private Event createAllEvent()
{EC_Cluster
    Event allEvent = new Event();
    allEvent.setClass("EC_Cluster");
    return (allEvent);
}

/*
 * createMembershipEvent
 * ---------------------
 * クラス EC_Cluster、サブクラス ESC_cluster_memberhip を
 * 使用して XML 登録イベントを作成する。
 */
private Event createMembershipEvent()
{
    Event membershipEvent = new Event();
    membershipEvent.setClass("EC_Cluster");
    membershipEvent.setSubclass("ESC_cluster_membership");
    return (membershipEvent);
}

/*
 * createRgEvent
 * ----------------
 * クラス EC_Cluster、サブクラス ESC_cluster_rg_state、
 * および "rg_name" nvpair（入力パラメータにもとづく）を
 * 1 つ使用して XML 登録イベントを作成する。
 */
private Event createRgEvent(String rgname)
{
    /*
     * rgname リソースグループ
     * 用のリソースグループ状態変更イベントを作成する。
     * このイベントタイプには、どのリソースグループに興味
     * があるのかを示すため、名前 / 値ペア（nvpair）を指定
     * する。
     */
    /*
     * イベントオブジェクトを作成し、クラスとサブクラスを設定する。
     */
    Event rgStateEvent = new Event();
    rgStateEvent.setClass("EC_Cluster");
    rgStateEvent.setSubclass("ESC_cluster_rg_state");

    /*
     * nvpair オブジェクトを作成し、これをこのイベントに追加する。
     */
    NVPair rgNvpair = new NVPair();
    rgNvpair.setName("rg_name");
    rgNvpair.setValue(rgname);
    rgStateEvent.addNvpair(rgNvpair);

    return (rgStateEvent);
}
```

```
/*
 * createREvent
 * ----------------
 * クラス EC_Cluster、サブクラス ESC_cluster_r_state、
 * および "r_name" nvpair (入力パラメータにもとづく) を
 * 1 つ使用して XML 登録イベントを作成する。
 */
private Event createREvent(String rname)
{
    /*
     * rgname リソース用の
     * リソース状態変更イベントを作成する。
     * このイベントタイプには、どのリソースグループに興味
     * があるかを示すため、名前 / 値ペア (nvpair) を指定
     * する。
     */
    Event rStateEvent = new Event();
    rStateEvent.setClass("EC_Cluster");
    rStateEvent.setSubclass("ESC_cluster_r_state");

    NVPair rNvpair = new NVPair();
    rNvpair.setName("r_name");
    rNvpair.setValue(rname);
    rStateEvent.addNvpair(rNvpair);

    return (rStateEvent);
}

/*
 * createUnregistrationString
 * ------------------
 * REMOVE_CLIENT CallbackReg オブジェクトを作成し、
 * CallbackReg オブジェクトから XML 文字列を取得する。
 */
private String createUnregistrationString() throws Exception
{
    /*
     * CallbackReg オブジェクトを作成する。
     */
    CallbackReg cbReg = new CallbackReg();
    cbReg.setPort("" + localPort);
    cbReg.setRegType(CallbackReg.REMOVE_CLIENT);

    /*
     * 登録を OutputStream に整列化する。          */
    String xmlStr = cbReg.convertToXml();

    // デバッグのために文字列を出力する。
    System.out.println(xmlStr);
    return (xmlStr);
}


/*
 * readRegistrationReply
```

```
     * ------------------------
     * xml を解析してドキュメントにし、このドキュメントから
     * RegReply オブジェクトを構築し、RegReply オブジェクトを
     * 出力する。実際のアプリケーションでは、通常、RegReply
     * オブジェクトの status_code にもとづいて処理をする。
     */
    private void readRegistrationReply(InputStream stream)
        throws Exception
    {
        // ドキュメントビルダーを作成する。
        DocumentBuilder db = dbf.newDocumentBuilder();

        //
        // 解析前に ErrorHandler を設定する。
        // ここではデフォルトハンドラを使用。
        //
        db.setErrorHandler(new DefaultHandler());

        // 入力ファイルを解析する。
        Document doc = db.parse(stream);

        RegReply reply = new RegReply(doc);
        reply.print(System.out);
    }

    /* private 指定のメンバー変数 */
    private InetAddress regIp;
    private int regPort;
    private EventReceptionThread evtThr;
    private String regs[];

    /* public 指定のメンバー変数 */
    public int localPort;
    public DocumentBuilderFactory dbf;
}

/*
 * クラス EventReceptionThread
 * ---------------------------
 * 上記のファイルヘッダーコメントを参照。
 */
class EventReceptionThread extends Thread
{
    /*
     * EventReceptionThread コンストラクタ
     * -----------------------------------
     * ローカルホスト名とワイルドカードポートにバインドされる
     * 新しい ServerSocket を作成する。
     */
    public EventReceptionThread(CrnpClient clientIn) throws IOException
    {
        /*
         * イベントの取得時に再度呼び返すことができるように、
         * クライアントに対する参照を保持する。
         */
```

```
    client = clientIn;

    /*
     * バインドする IP を指定する。これは、ローカル
     * ホスト IP である。このマシンに複数のパブリック
     * インタフェースが構成されている場合は、
     * InetAddress.getLocalHost によって検出される
     * ものをどれでも使用する。
     *
     */
    listeningSock = new ServerSocket(client.localPort, 50,
        InetAddress.getLocalHost());
        System.out.println(listeningSock);
}

/*
 * run
 * ---
 * Thread.Start メソッドによって呼び出される。
 *
 * ServerSocket で着信接続を待機し、永続的にループする。
 *
 * 各着信接続が受け入れられる際に xml ストリームから
 * Event オブジェクトが作成される。続いてこのオブジェクト
 * が CrnpClient オブジェクトに返されて処理される。
 */
public void run()
{
    /*
     * 永続的にループする。
     */
    try {
        //
        // CrnpClient 内のドキュメントビルダーファクトリを
        // 使用してドキュメントビルダーを作成する。
        //
        DocumentBuilder db = client.dbf.newDocumentBuilder();

        //
        // 解析前に ErrorHandler を設定する。
        // ここではデフォルトハンドラを使用。
        //
        db.setErrorHandler(new DefaultHandler());

        while(true) {
            /* サーバーからのコールバックを待機 */
            Socket sock = listeningSock.accept();

            // 入力ファイルを解析する。
            Document doc = db.parse(sock.getInputStream());

            Event event = new Event(doc);
            client.processEvent(event);

            /* ソケットを閉じる */
```

```
                sock.close();
            }
            // 到達不能

        } catch (Exception e) {
            System.out.println(e);
            System.exit(1);
        }
    }

    /* private 指定のメンバー変数 */
    private ServerSocket listeningSock;
    private CrnpClient client;
}

/*
 * クラス NVPair
 * -----------
 * このクラスは名前 / 値ペア (両方とも文字列) を格納する。
 * このクラスは、そのメンバーから NVPAIR XML メッセージを構築し、
 * NVPAIR XML 要素を解析してそのメンバーにすることができる。
 *
 * NVPAIR の形式仕様では複数の値が許可されているが、ここでは
 * 単純に値は 1 つだけという前提を下す。
 */
class NVPair
{
    /*
     * 2 つのコンストラクタ: 最初のコンストラクタは空の NVPair を
     * 作成する。2 つ目は NVPAIR XML 要素から NVPair を作成する。
     */
    public NVPair()
    {
        name = value = null;
    }

    public NVPair(Element elem)
    {
        retrieveValues(elem);
    }

    /*
     * public 指定のセッター。
     */
    public void setName(String nameIn)
    {
        name = nameIn;
    }

    public void setValue(String valueIn)
    {
        value = valueIn;
    }

    /*
```

```
 * 1 行で名前と値を出力する。
 */
public void print(PrintStream out)
{
    out.println("NAME=" + name + " VALUE=" + value);
}

/*
 * createXmlElement
 * ------------------
 * メンバー変数から NVPAIR XML 要素を作成する。
 * この要素を作成できるように、ドキュメントをパラメータとして
 * 受け付ける。
 */
public Element createXmlElement(Document doc)
{
    // 要素を作成する。
    Element nvpair = (Element)
        doc.createElement("NVPAIR");
    //
    // 名前を追加する。実際の名前は別の
    // CDATA セクションであることに注意。
    //
    Element eName = doc.createElement("NAME");
    Node nameData = doc.createCDATASection(name);
    eName.appendChild(nameData);
    nvpair.appendChild(eName);
    //
    // 値を追加する。実際の値は別の
    // CDATA セクションであることに注意。
    //
    Element eValue = doc.createElement("VALUE");
    Node valueData = doc.createCDATASection(value);
    eValue.appendChild(valueData);
    nvpair.appendChild(eValue);

    return (nvpair);
}

/*
 * retrieveValues
 * ----------------
 * XML 要素を解析して名前と値を取得する。
 */
private void retrieveValues(Element elem)
{
    Node n;
    NodeList nl;

    //
    // NAME 要素を検出する。
    //
    nl = elem.getElementsByTagName("NAME");
    if (nl.getLength() != 1) {
        System.out.println("Error in parsing: can't find "
```

```
            + "NAME node.");
        return;
    }

    //
    // TEXT セクションを取得する。
    //
    n = nl.item(0).getFirstChild();
    if (n == null || n.getNodeType() != Node.TEXT_NODE) {
        System.out.println("Error in parsing: can't find "
            + "TEXT section.");
        return;
    }

    // 値を取得する。
    name = n.getNodeValue();

    //
    // ここで値要素を取得する。
    //
    nl = elem.getElementsByTagName("VALUE");
    if (nl.getLength() != 1) {
        System.out.println("Error in parsing: can't find "
            + "VALUE node.");
        return;
    }

    //
    // TEXT セクションを取得する。
    //
    n = nl.item(0).getFirstChild();
    if (n == null || n.getNodeType() != Node.TEXT_NODE) {
        System.out.println("Error in parsing: can't find "
            + "TEXT section.");
        return;
    }

    // 値を取得する。
    value = n.getNodeValue();
}


/*
 * public 指定のアクセッサ
 */
public String getName()
{
    return (name);
}

public String getValue()
{
    return (value);
}
```

```
    // private 指定のメンバー変数
    private String name, value;
}


/*
 * クラス Event
 * -----------
 * このクラスは、クラス、サブクラス、ベンダー、パブリッシャー、名前 /
 * 値ペアのリストから成るイベントを格納する。このクラスでは、そのメンバー
 * から SC_EVENT_REG XML 要素を作成し、この要素を解析してそのメンバーに
 * することができる。次の非対称性に注意: SC_EVENT 要素を解析するが、
 * 作成するのは SC_EVENT_REG 要素である。これは、SC_EVENT_REG 要素が
 * 登録メッセージ（これは作成の必要がある）内で使用され、SC_EVENT 要素
 * がイベント配信（これは解析の必要がある）内で使用されるためである。
 * 違いは、SC_EVENT_REG 要素にはベンダーとパブリッシャーがないことだけ
 * である。
 */
class Event
{

    /*
     * 2 つのコンストラクタ: 最初のコンストラクタは空のイベントを
     * 作成し、2 つ目は SC_EVENT XML ドキュメントからイベントを
     * 作成する。
     */
    public Event()
    {
        regClass = regSubclass = null;
        nvpairs = new Vector();
    }

    public Event(Document doc)
    {

        nvpairs = new Vector();

        //
        // デバッグで使用できるようにドキュメントを文字列に
        // 変換して出力する。
        //
        DOMSource domSource = new DOMSource(doc);
        StringWriter strWrite = new StringWriter();
        StreamResult streamResult = new StreamResult(strWrite);
        TransformerFactory tf = TransformerFactory.newInstance();
        try {
            Transformer transformer = tf.newTransformer();
            transformer.transform(domSource, streamResult);
        } catch (TransformerException e) {
            System.out.println(e.toString());
            return;
        }
        System.out.println(strWrite.toString());

        // 実際に解析する。
```

```
    retrieveValues(doc);
}

/*
 *public 指定のセッター。
 */
public void setClass(String classIn)
{
    regClass = classIn;
}

public void setSubclass(String subclassIn)
{
    regSubclass = subclassIn;
}

public void addNvpair(NVPair nvpair)
{
    nvpairs.add(nvpair);
}

/*
 * createXmlElement
 * ------------------
 * メンバー変数から SC_EVENT_REG XML 要素を作成する。
 * この要素を作成できるように、ドキュメントをパラメータとして
 * 受け付ける。NVPair createXmlElement 機能を使用。
 */
public Element createXmlElement(Document doc)
{
    Element event = (Element)
        doc.createElement("SC_EVENT_REG");
    event.setAttribute("CLASS", regClass);
    if (regSubclass != null) {
        event.setAttribute("SUBCLASS", regSubclass);
    }
    for (int i = 0; i < nvpairs.size(); i++) {
        NVPair tempNv = (NVPair)
            (nvpairs.elementAt(i));
        event.appendChild(tempNv.createXmlElement(
            doc));
    }
    return (event);
}

/*
 * メンバー変数を複数行に出力する。
 */
public void print(PrintStream out)
{
    out.println("\tCLASS=" + regClass);
    out.println("\tSUBCLASS=" + regSubclass);
    out.println("\tVENDOR=" + vendor);
    out.println("\tPUBLISHER=" + publisher);
    for (int i = 0; i < nvpairs.size(); i++) {
```

```
        NVPair tempNv = (NVPair)
            (nvpairs.elementAt(i));
        out.print("\t\t");
        tempNv.print(out);
    }
}

 /*
 * retrieveValues
 * ----------------
 * XML ドキュメントを解析し、クラス、サブクラス、ベンダー、
 * パブリッシャー、および nvpair を取得する。
 */
private void retrieveValues(Document doc)
{
    Node n;
    NodeList nl;

    //
    // SC_EVENT 要素を検出する。
    //
    nl = doc.getElementsByTagName("SC_EVENT");
    if (nl.getLength() != 1) {
        System.out.println("Error in parsing: can't find "
            + "SC_EVENT node.");
        return;
    }

    n = nl.item(0);

    //
    // CLASS、SUBCLASS、VENDOR、および PUBLISHER
    // 属性の値を取得する。
    //
    regClass = ((Element)n).getAttribute("CLASS");
    regSubclass = ((Element)n).getAttribute("SUBCLASS");
    publisher = ((Element)n).getAttribute("PUBLISHER");
    vendor = ((Element)n).getAttribute("VENDOR");

    //
    // すべての nv ペアを取得する。
    //
    for (Node child = n.getFirstChild(); child != null;
         child = child.getNextSibling())
    {
        nvpairs.add(new NVPair((Element)child));
    }
}

/*
 * public 指定のアクセッサメソッド。
 */
public String getRegClass()
{
    return (regClass);
```

```
    }

    public String getSubclass()
    {
        return (regSubclass);
    }

    public String getVendor()
    {
        return (vendor);
    }

    public String getPublisher()
    {
        return (publisher);
    }

    public Vector getNvpairs()
    {
        return (nvpairs);
    }

    // private 指定のメンバー変数
    private String regClass, regSubclass;
    private Vector nvpairs;
    private String vendor, publisher;
}


/*
 * クラス CallbackReg
 * ------------
 * このクラスは、ポートと登録タイプ (どちらも文字列)、およびイベントリストを
 * 格納する。このクラスは、そのメンバーから SC_CALLBACK_REG XML メッセージ
 * を作成できる。
 *
 * SC_CALLBACK_REG メッセージを解析する必要があるのは CRNP サーバー
 * だけであるため、このクラスで SC_CALLBACK_REG メッセージを解析でき
 * なくてもよい。
 */
class CallbackReg
{
    // setRegType メソッドに便利な定義
    public static final int ADD_CLIENT = 0;
    public static final int ADD_EVENTS = 1;
    public static final int REMOVE_EVENTS = 2;
    public static final int REMOVE_CLIENT = 3;

    public CallbackReg()
    {
        port = null;
        regType = null;
        regEvents = new Vector();
    }
```

```
/*
 * public 指定のセッター。
 */
public void setPort(String portIn)
{
    port = portIn;
}

public void setRegType(int regTypeIn)
{
    switch (regTypeIn) {
    case ADD_CLIENT:
        regType = "ADD_CLIENT";
        break;
    case ADD_EVENTS:
        regType = "ADD_EVENTS";
        break;
    case REMOVE_CLIENT:
        regType = "REMOVE_CLIENT";
        break;
    case REMOVE_EVENTS:
        regType = "REMOVE_EVENTS";
        break;
    default:
        System.out.println("Error, invalid regType " +
            regTypeIn);
        regType = "ADD_CLIENT";
        break;
    }
}

public void addRegEvent(Event regEvent)
{
    regEvents.add(regEvent);
}


/*
 * convertToXml
 * ------------------
 * メンバー変数から SC_CALLBACK_REG XML ドキュメントを構築する。
 * Event createXmlElement 機能を使用。
 */
public String convertToXml()
{
    Document document = null;
    DocumentBuilderFactory factory =
        DocumentBuilderFactory.newInstance();
    try {
        DocumentBuilder builder = factory.newDocumentBuilder();
        document = builder.newDocument();
    } catch (ParserConfigurationException pce) {
        // 指定したオプションを持つパーサーを構築できない。
        pce.printStackTrace();
        System.exit(1);
```

```
        }
        Element root = (Element) document.createElement(
            "SC_CALLBACK_REG");
        root.setAttribute("VERSION", "1.0");
        root.setAttribute("PORT", port);
        root.setAttribute("REG_TYPE", regType);
        for (int i = 0; i < regEvents.size(); i++) {
            Event tempEvent = (Event)
                (regEvents.elementAt(i));
            root.appendChild(tempEvent.createXmlElement(
                document));
        }
        document.appendChild(root);

        //
        // ここでドキュメントを文字列に変換する。
        //
        DOMSource domSource = new DOMSource(document);
        StringWriter strWrite = new StringWriter();
        StreamResult streamResult = new StreamResult(strWrite);
        TransformerFactory tf = TransformerFactory.newInstance();
        try {
            Transformer transformer = tf.newTransformer();
            transformer.transform(domSource, streamResult);
        } catch (TransformerException e) {
            System.out.println(e.toString());
            return ("");
        }
        return (strWrite.toString());
    }

    // private 指定のメンバー変数
    private String port;
    private String regType;
    private Vector regEvents;
}

/*
 * クラス RegReply
 * -----------
 * このクラスは、status_code と status_msg (どちらも文字列) を格納する。
 * このクラスは、SC_REPLY XML 要素を解析し、そのメンバーにできる。
 */
class RegReply
{

    /*
     * 1 つのコンストラクタが XML ドキュメントを受け入れて解析を行う。
     */
    public RegReply(Document doc)
    {
        //
        // ここでドキュメントを文字列に変換する。
        //
        DOMSource domSource = new DOMSource(doc);
```

```
    StringWriter strWrite = new StringWriter();
    StreamResult streamResult = new StreamResult(strWrite);
    TransformerFactory tf = TransformerFactory.newInstance();
    try {
        Transformer transformer = tf.newTransformer();
        transformer.transform(domSource, streamResult);
    } catch (TransformerException e) {
        System.out.println(e.toString());
        return;
    }
    System.out.println(strWrite.toString());

    retrieveValues(doc);
}

/*
 * public 指定のアクセッサ
 */
public String getStatusCode()
{
    return (statusCode);
}

public String getStatusMsg()
{
    return (statusMsg);
}

/*
 * 1 行で情報を出力する。
 */
public void print(PrintStream out)
{
    out.println(statusCode + ": " +
        (statusMsg != null ? statusMsg : ""));
}

 /*
 * retrieveValues
 * ----------------
 * XML ドキュメントを解析し、statusCode と statusMsg を取得する。
 */
private void retrieveValues(Document doc)
{
    Node n;
    NodeList nl;

    //
    // SC_REPLY 要素を検出する。
    //
    nl = doc.getElementsByTagName("SC_REPLY");
    if (nl.getLength() != 1) {
        System.out.println("Error in parsing: can't find "
            + "SC_REPLY node.");
        return;
```

```
        }

        n = nl.item(0);

        // STATUS_CODE 属性の値を取得する。
        statusCode = ((Element)n).getAttribute("STATUS_CODE");

        //
        // SC_STATUS_MSG 要素を検出する。
        //
        nl = ((Element)n).getElementsByTagName("SC_STATUS_MSG");
        if (nl.getLength() != 1) {
            System.out.println("Error in parsing: can't find "
                + "SC_STATUS_MSG node.");
            return;
        }

        //
        // TEXT セクションが存在する場合は、それを取得する。
        //
        n = nl.item(0).getFirstChild();
        if (n == null || n.getNodeType() != Node.TEXT_NODE) {
            // 存在しなくてもエラーではないため、
            // このまま戻る。
            return;
        }

        // 値を取得する。
        statusMsg = n.getNodeValue();
    }

    // private 指定のメンバー変数
    private String statusCode;
    private String statusMsg;
}
```

索引

H

I

J

L

M

P

R

S

T

U

V

W

X

さ

し

す