

Oracle® Solaris Cluster Data Services Developer's Guide

ORACLE®

Part No: E56681
September 2015

Part No: E56681

Copyright © 2000, 2015, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info> or visit <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs> if you are hearing impaired.

Référence: E56681

Copyright © 2000, 2015, Oracle et/ou ses affiliés. Tous droits réservés.

Ce logiciel et la documentation qui l'accompagne sont protégés par les lois sur la propriété intellectuelle. Ils sont concédés sous licence et soumis à des restrictions d'utilisation et de divulgation. Sauf stipulation expresse de votre contrat de licence ou de la loi, vous ne pouvez pas copier, reproduire, traduire, diffuser, modifier, accorder de licence, transmettre, distribuer, exposer, exécuter, publier ou afficher le logiciel, même partiellement, sous quelque forme et par quelque procédé que ce soit. Par ailleurs, il est interdit de procéder à toute ingénierie inverse du logiciel, de le désassembler ou de le décompiler, excepté à des fins d'interopérabilité avec des logiciels tiers ou tel que prescrit par la loi.

Les informations fournies dans ce document sont susceptibles de modification sans préavis. Par ailleurs, Oracle Corporation ne garantit pas qu'elles soient exemptes d'erreurs et vous invite, le cas échéant, à lui en faire part par écrit.

Si ce logiciel, ou la documentation qui l'accompagne, est livré sous licence au Gouvernement des Etats-Unis, ou à quiconque qui aurait souscrit la licence de ce logiciel pour le compte du Gouvernement des Etats-Unis, la notice suivante s'applique :

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

Ce logiciel ou matériel a été développé pour un usage général dans le cadre d'applications de gestion des informations. Ce logiciel ou matériel n'est pas conçu ni n'est destiné à être utilisé dans des applications à risque, notamment dans des applications pouvant causer un risque de dommages corporels. Si vous utilisez ce logiciel ou ce matériel dans le cadre d'applications dangereuses, il est de votre responsabilité de prendre toutes les mesures de secours, de sauvegarde, de redondance et autres mesures nécessaires à son utilisation dans des conditions optimales de sécurité. Oracle Corporation et ses affiliés déclinent toute responsabilité quant aux dommages causés par l'utilisation de ce logiciel ou matériel pour des applications dangereuses.

Oracle et Java sont des marques déposées d'Oracle Corporation et/ou de ses affiliés. Tout autre nom mentionné peut correspondre à des marques appartenant à d'autres propriétaires qu'Oracle.

Intel et Intel Xeon sont des marques ou des marques déposées d'Intel Corporation. Toutes les marques SPARC sont utilisées sous licence et sont des marques ou des marques déposées de SPARC International, Inc. AMD, Opteron, le logo AMD et le logo AMD Opteron sont des marques ou des marques déposées d'Advanced Micro Devices. UNIX est une marque déposée de The Open Group.

Ce logiciel ou matériel et la documentation qui l'accompagne peuvent fournir des informations ou des liens donnant accès à des contenus, des produits et des services émanant de tiers. Oracle Corporation et ses affiliés déclinent toute responsabilité ou garantie expresse quant aux contenus, produits ou services émanant de tiers, sauf mention contraire stipulée dans un contrat entre vous et Oracle. En aucun cas, Oracle Corporation et ses affiliés ne sauraient être tenus pour responsables des pertes subies, des coûts occasionnés ou des dommages causés par l'accès à des contenus, produits ou services tiers, ou à leur utilisation, sauf mention contraire stipulée dans un contrat entre vous et Oracle.

Accès aux services de support Oracle

Les clients Oracle qui ont souscrit un contrat de support ont accès au support électronique via My Oracle Support. Pour plus d'informations, visitez le site <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info> ou le site <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs> si vous êtes malentendant.

Contents

Using This Documentation	13
1 Overview of Resource Management	15
Oracle Solaris Cluster Application Environment	15
Resource Group Manager Model	17
Description of a Resource Type	17
Description of a Resource	18
Description of a Resource Group	18
Resource Group Manager	19
Callback Methods	19
Programming Interfaces	21
Resource Management API	21
Data Service Development Library	21
Oracle Solaris Cluster Agent Builder	22
Resource Group Manager Administrative Interface	22
Oracle Solaris Cluster Manager	22
clsetup Utility	23
Administrative Commands	24
2 Developing a Data Service	25
Analyzing the Application for Suitability	25
Determining the Interface to Use	27
Setting Up the Development Environment for Writing a Data Service	28
▼ How to Set Up the Development Environment	29
Transferring a Data Service to a Cluster	30
Setting Standard Properties	30
Cluster Properties	30
Resource Type Properties	31

Resource Properties	31
Resource Group Properties	31
Resource Property Attributes	31
Node List Properties	31
Setting Resource and Resource Type Properties	32
Declaring Resource Type Properties	32
Declaring Resource Properties	35
Declaring Extension Properties	39
Implementing Callback Methods	41
Accessing Resource and Resource Group Property Information	41
Idempotence of Methods	42
How Methods Are Invoked in Zones	42
Generic Data Service	42
Controlling an Application	43
Starting and Stopping a Resource	43
Using the Optional Init, Fini, and Boot Methods	46
Monitoring a Resource	48
Implementing Monitors and Methods That Execute Exclusively in the Global Zone	49
Adding Message Logging to a Resource	50
Providing Process Management	50
Providing Administrative Support for a Resource	51
Implementing a Failover Resource	52
Implementing a Scalable Resource	53
Validation Checks for Scalable Services	55
Writing and Testing Data Services	56
Using TCP Keep-Alives to Protect the Server	56
Testing HA Data Services	57
Coordinating Dependencies Between Resources	57
Legal RGM Names	59
3 Resource Management API Reference	61
RMAPI Access Methods	61
RMAPI Shell Commands	61
C Functions	63
RMAPI Callback Methods	67
Arguments That You Can Provide to Callback Methods	67

Callback Method Exit Codes	68
Control and Initialization Callback Methods	68
Administrative Support Methods	71
Net-Relative Callback Methods	72
Monitor Control Callback Methods	72
4 Modifying a Resource Type	75
Overview of Modifying a Resource Type	75
Setting Up the Contents of the Resource Type Registration File	76
Resource Type Name	76
Specifying the # <i>\$upgrade</i> and # <i>\$upgrade_from</i> Directives	77
Changing the <i>RT_version</i> in an RTR File	78
What Happens When a Cluster Administrator Upgrades	79
Determining Installation Requirements and Packaging	80
Before You Change the RTR File	80
Changing Monitor Code	81
Changing Method Code	81
Determining the Packaging Scheme to Use	81
Documentation to Provide for a Modified Resource Type	82
Information About What to Do Before Installing an Upgrade	83
Information About When to Upgrade Resources	83
Information About Changes to Resource Properties	84
5 Sample Data Service	85
Overview of the Sample Data Service	85
Defining the Resource Type Registration File	86
Overview of the RTR File	86
Resource Type Properties in the Sample RTR File	87
Resource Properties in the Sample RTR File	88
Providing Common Functionality to All Methods	92
Identifying the Command Interpreter and Exporting the Path	92
Declaring the <i>PMF_TAG</i> and <i>SYSLOG_TAG</i> Variables	93
Parsing the Function Arguments	94
Generating Error Messages	95
Obtaining Property Information	96
Controlling the Data Service	97

How the Start Method Works	97
How the Stop Method Works	100
Defining a Fault Monitor	103
How the Probe Program Works	103
How the Monitor_start Method Works	109
How the Monitor_stop Method Works	110
How the Monitor_check Method Works	112
Handling Property Updates	113
How the Validate Method Works	113
How the Update Method Works	118
6 Data Service Development Library	121
DSDL Overview	121
Managing Configuration Properties	121
Starting and Stopping a Data Service	122
Implementing a Fault Monitor	123
Accessing Network Address Information	124
Debugging the Resource Type Implementation	124
Enabling Highly Available Local File Systems	124
7 Designing Resource Types	127
Resource Type Registration File	127
Validate Method	128
Start Method	130
Stop Method	131
Monitor_start Method	132
Monitor_stop Method	133
Monitor_check Method	133
Update Method	134
Description of Init, Fini, and Boot Methods	135
Designing the Fault Monitor Daemon	135
Supporting Resource Types That Perform Resource Migration From Their Stop Method	137
SCHA_TARGET_NODES Query	138
Pre_evict Resource Property	140
Example of a Giveover of a Resource Group With Live Migration	141

Error Handling	142
8 Sample DSDL Resource Type Implementation	143
X Font Server	143
X Font Server Configuration File	144
TCP Port Number	144
ORCL.xfnts RTR File	145
Naming Conventions for Functions and Callback Methods	145
scds_initialize() Function	146
xfnts_start Method	146
Validating the Service Before Starting the X Font Server	146
Starting the Service With svc_start()	147
Returning From svc_start()	148
xfnts_stop Method	151
xfnts_monitor_start Method	152
xfnts_monitor_stop Method	153
xfnts_monitor_check Method	154
ORCL.xfnts Fault Monitor	154
xfnts_probe Main Loop	155
svc_probe() Function	157
Determining the Fault Monitor Action	160
xfnts_validate Method	160
xfnts_update Method	163
9 Oracle Solaris Cluster Agent Builder	165
Agent Builder Overview	165
Before You Use Agent Builder	166
Using Agent Builder	167
Analyzing the Application	167
Installing and Configuring Agent Builder	168
Agent Builder Screens	168
Starting Agent Builder	169
Navigating Agent Builder	169
Using the Create Screen	172
Using the Configure Screen	173
Using the Agent Builder Korn Shell-Based \$hostnames Variable	175

Using Property Variables	176
Reusing Code That You Create With Agent Builder	178
▼ How to Use the Command-Line Version of Agent Builder	180
Directory Structure That Agent Builder Creates	180
Agent Builder Output	181
Source and Binary Files	182
Utility Scripts and Man Pages That Agent Builder Creates	183
Support Files That Agent Builder Creates	184
Package Directory That Agent Builder Creates	184
rtconfig File	185
10 DSDL API Functions	187
General-Purpose Functions	187
Initialization Functions	187
Retrieval Functions	188
Failover and Restart Functions	188
Execution Functions	189
Property Functions	189
Network Resource Access Functions	189
Host Name Functions	190
Port List Functions	190
Network Address Functions	190
Fault Monitoring Using TCP Connections Functions	191
PMF Functions	191
Fault Monitor Functions	192
Utility Functions	192
11 Cluster Reconfiguration Notification Protocol	193
CRNP Concepts	193
How the CRNP Works	194
CRNP Semantics	195
CRNP Message Types	196
How a Client Registers With the Server	197
Assumptions About How Administrators Set Up the Server	197
How the Server Identifies a Client	197
How SC_CALLBACK_REG Messages Are Passed Between a Client and the Server	197

How the Server Replies to a Client	199
Contents of an SC_REPLY Message	200
How a Client Is to Handle Error Conditions	200
How the Server Delivers Events to a Client	201
How the Delivery of Events Is Guaranteed	202
Contents of an SC_EVENT Message	202
How the CRNP Authenticates Clients and the Server	203
Example of Creating a Java Application That Uses the CRNP	204
▼ How to Set Up Your Environment	205
▼ How to Start Developing Your Application	205
▼ How to Parse the Command-Line Arguments	207
▼ How to Define the Event Reception Thread	208
▼ How to Register and Unregister Callbacks	209
▼ How to Generate the XML	210
▼ How to Create the Registration and Unregistration Messages	214
▼ How to Set Up the XML Parser	217
▼ How to Parse the Registration Reply	217
▼ How to Parse the Callback Events	219
▼ How to Run the Application	223
12 Security for Data Services	225
Storing Application Passwords in Private Strings	225
Invoking Application Programs with Least Privilege	227
Using the resource_security Property	227
Using the application_user Property	228
Using the scha_check_app_user Command	229
Executing the Application Under a Specified Oracle Solaris Project	229
A Sample Data Service Code Listings	231
Resource Type Registration File Listing	231
Start Method Code Listing	234
Stop Method Code Listing	237
gettime Utility Code Listing	240
PROBE Program Code Listing	240
Monitor_start Method Code Listing	246
Monitor_stop Method Code Listing	248

Monitor_check Method Code Listing	250
Validate Method Code Listing	252
Update Method Code Listing	256
B DSDL Sample Resource Type Code Listings	259
xfnts.c File Listing	259
xfnts_monitor_check Method Code Listing	272
xfnts_monitor_start Method Code Listing	273
xfnts_monitor_stop Method Code Listing	274
xfnts_probe Method Code Listing	275
xfnts_start Method Code Listing	278
xfnts_stop Method Code Listing	280
xfnts_update Method Code Listing	281
xfnts_validate Method Code Listing	283
C Requirements for Non-Cluster-Aware Applications	285
Multihosted Data	285
Using Symbolic Links for Multihosted Data Placement	286
Host Names	287
Multihomed Hosts	287
Binding to INADDR_ANY as Opposed to Binding to Specific IP Addresses	288
Client Retry	289
D Document Type Definitions for the CRNP	291
SC_CALLBACK_REG XML DTD	291
NVPAIR XML DTD	293
SC_REPLY XML DTD	294
SC_EVENT XML DTD	295
E CrnpClient.java Application	297
Contents of CrnpClient.java	297
Index	323

Using This Documentation

- **Overview** – Contains information about the Resource Management API to develop Oracle Solaris Cluster data services on both SPARC and x86 based systems.
- **Audience** – Experienced system administrators with extensive knowledge of Oracle software and hardware.
- **Required knowledge** – Knowledge of the Oracle Solaris operating system, of Oracle Solaris Cluster software, and expertise with the volume manager software that is used with Oracle Solaris Cluster software.

Product Documentation Library

Documentation and resources for this product and related products are available at <http://www.oracle.com/pls/topic/lookup?ctx=E56676-01>.

Feedback

Provide feedback about this documentation at <http://www.oracle.com/goto/docfeedback>.

◆◆◆ CHAPTER 1

Overview of Resource Management

This book provides guidelines for creating a resource type for a software application, such as Oracle Database, Oracle GlassFish Server, or DNS. As such, this book is intended for developers of resource types.

To understand the contents of this book, you must be thoroughly familiar with the concepts that are presented in the *Oracle Solaris Cluster 4.3 Concepts Guide*.

This chapter provides an overview of the concepts that you need to understand to develop a data service. This chapter covers the following topics:

- “Oracle Solaris Cluster Application Environment” on page 15
- “Resource Group Manager Model” on page 17
- “Resource Group Manager” on page 19
- “Callback Methods” on page 19
- “Programming Interfaces” on page 21
- “Resource Group Manager Administrative Interface” on page 22

Note - This book uses the terms *resource type* and *data service* interchangeably. The term *agent*, though rarely used in this book, is equivalent to a resource type or collection of related resource types.

Oracle Solaris Cluster Application Environment

The Oracle Solaris Cluster system enables applications to be run and administered as highly available and scalable resources. The Resource Group Manager (RGM) provides the mechanism for high availability and scalability.

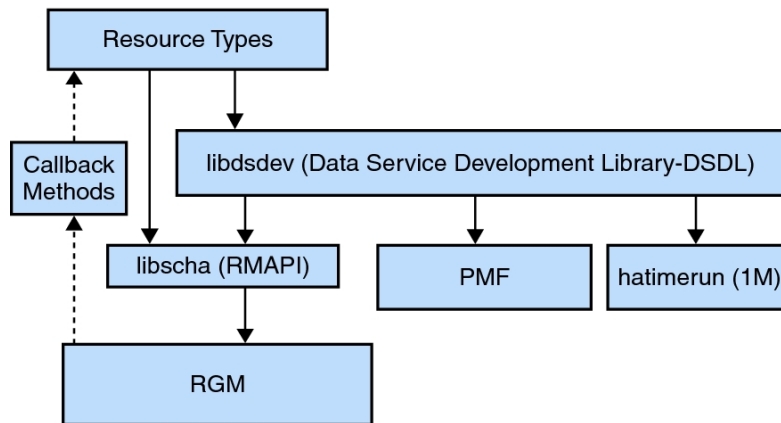
The following elements form the programming interface to this facility:

- A set of callback methods that you write that enable the RGM to control an application in the cluster.

- The Resource Management API (RMAPI), a set of low-level API commands and functions that you can use to write the callback methods. These APIs are implemented in the `libscha.so` library.
- Process Monitor Facility (PMF) for monitoring and restarting processes in the cluster.
- The Data Service Development Library (DSDL), a set of library functions that encapsulates the low-level API and process-management functionality at a higher level. The DSDL adds some additional functionality to ease the writing of callback methods. These functions are implemented in the `libdsdev.so` library.

The following figure shows the interrelationship of these elements.

FIGURE 1 Programming Architecture of the Oracle Solaris Cluster Application Environment



The Oracle Solaris Cluster Agent Builder, which is described in [Chapter 9, “Oracle Solaris Cluster Agent Builder”](#), is a tool in the Oracle Solaris Cluster package that automates the process of creating a data service. Agent Builder generates data service code in either C (by using DSDL functions to write the callback methods) or in the Korn (ksh) shell command language (by using low-level API commands to write the callback methods). Agent Builder also has an option to generate a set of driving scripts to be used with the Generic Data Service (GDS).

The RGM runs as a daemon on each cluster node and automatically starts and stops resources on selected Oracle Solaris hosts according to preconfigured policies. The RGM makes a resource highly available in the event of a node failure or reboot. The RGM does so by stopping the resource on the affected node and starting it on another node. The RGM also automatically starts and stops resource-specific monitors. These monitors detect resource failures and relocate failing resources onto other nodes or monitor other aspects of resource performance.

The RGM supports both failover resources and scalable resources. A failover resource can be online on only one node at a time. A scalable resource can be online on multiple nodes simultaneously.

Resource Group Manager Model

This section introduces some fundamental terminology and explains in more detail the RGM and its associated interfaces.

The RGM handles three major kinds of interrelated objects: resource types, resources, and resource groups. One way to introduce these objects is by means of an example, as follows.

You implement a resource type, `ha-oracle`, that makes an existing Oracle DBMS application highly available. An end user defines separate databases for marketing, engineering, and finance, each of which is a resource of type `ha-oracle`. The cluster administrator places these resources in separate resource groups so that they can run on different nodes and fail over independently. You create a second resource type, `ha-calendar`, to implement a highly available calendar server that requires an Oracle database. The cluster administrator places the resource for the finance calendar into the same resource group as the finance database resource. The cluster administrator does so to ensure that both resources run on the same node and fail over together.

Description of a Resource Type

A *resource type* consists of the following elements:

- A software component to be run in the cluster
- Control programs that are used as callback methods by the RGM to manage the software component as a cluster resource
- A set of properties that form part of the static configuration for the software component on the cluster

The RGM uses resource type properties to manage resources of a particular type.

Note - In addition to a software application, a resource type can represent other system resources, such as network addresses.

You specify the properties for the resource type and set property values in a resource type registration (RTR) file. The RTR file follows the format that is described in [“Setting Resource](#)

and [Resource Type Properties](#)” on page 32 and in the `rt_reg(4)` man page. See also [“Defining the Resource Type Registration File”](#) on page 86 for a description of a sample RTR file.

[“Resource Type Properties”](#) on page 31 provides a list of the resource type properties.

The cluster administrator installs and registers the resource type implementation and underlying application on a cluster. The registration procedure enters the information from the RTR file into the cluster configuration. The *Oracle Solaris Cluster 4.3 Data Services Planning and Administration Guide* describes the procedure for registering a data service.

Description of a Resource

A *resource* inherits the properties and values of its resource type. In addition, you can declare resource properties in the RTR file. [“Resource Properties”](#) on page 31 contains a list of resource properties, each of which can take a different value for each resource of this type.

The cluster administrator can change the values of particular properties depending on how the properties are specified in the RTR file. For example, property definitions can specify a range of allowable values. Property definitions can also specify when the property is tunable: never, any time, at creation (when the resource is added to the cluster), or when the resource is disabled. Within these specifications, the cluster administrator can make changes to properties by using administration commands.

The cluster administrator can create many resources of the same type, with each resource having its own name and set of property values, so that more than one instance of the underlying application can run in the cluster. Each instantiation requires a unique name within the cluster.

Description of a Resource Group

Each resource must be configured in a resource group. The RGM brings all resources in a group online and offline together on the same node. When the RGM brings a resource group online or offline, it runs callback methods on the individual resources in the group.

The nodes where a resource group is currently online are called its *primaries* or *primary nodes*. A resource group is *mastered* by each of its primaries. Each resource group has an associated `NodeList` property that identifies all *potential primaries* or *masters* of the resource group. The cluster administrator sets the `NodeList` property, which otherwise includes all cluster nodes by default.

A resource group also has a set of properties. These properties include configuration properties that can be set by the cluster administrator and dynamic properties, set by the RGM, that reflect the active state of the resource group.

The RGM defines two types of resource groups: failover and scalable. A failover resource group can be online on only one node at any time. A scalable resource group can be online on multiple nodes simultaneously. The RGM provides a set of properties to support the creation of each type of resource group. See [“Transferring a Data Service to a Cluster” on page 30](#) and [“Implementing Callback Methods” on page 41](#) for details about these properties.

[“Resource Group Properties” on page 31](#) contains a list of resource group properties.

Resource Group Manager

The Resource Group Manager (RGM) is implemented as a daemon, `rgmd`, that runs on each global-cluster node. All of the `rgmd` processes communicate with each other and act as a single cluster-wide facility.

The RGM supports the following functions:

- Whenever a node fails, the RGM attempts to maintain the availability of all managed resource groups. The RGM does so by automatically bringing them online on correct masters.
- If a particular resource fails, its monitor program can request that the resource group be restarted on the same master or switched to a new master.
- The cluster administrator can issue an administrative command to request one of the following actions:
 - Change mastery of a resource group.
 - Enable or disable a particular resource within a resource group.
 - Create, delete, or modify a resource type, a resource, or a resource group.

Whenever the RGM activates configuration changes, it coordinates its actions across all member nodes of the cluster. This kind of activity is known as a *reconfiguration*. To effect a state change on an individual resource, the RGM runs a resource type-specific callback method on that resource.

Callback Methods

The Oracle Solaris Cluster framework uses a callback mechanism to provide communication between a data service and the RGM. The framework defines a set of callback methods,

including their arguments and return values, and the circumstances under which the RGM calls each method.

You create a data service by coding a set of individual callback methods and implementing each method as a control program that the RGM can call. That is, the data service does not consist of a single executable, but a number of executable scripts (ksh) or binaries (C), each of which the RGM can call directly.

Callback methods are registered with the RGM through the RTR file. In the RTR file you identify the program for each method that you have implemented for the data service. When a cluster administrator registers the data service on a cluster, the RGM reads the RTR file, which provides the identity of the callback programs and other information.

The only required callback methods for a resource type are a start method (Start or Pernet_start) and a stop method (Stop or Postnet_stop).

The callback methods can be grouped into the following categories:

- Control and initialization methods
 - The Start and Stop methods start and stop resources in a group that is being brought online or offline.
 - The Init, Fini, and Boot methods execute initialization and termination code on resources.
- Administrative support methods
 - The Validate method verifies properties that are set by administrative action.
 - The Update method updates the property settings of an online resource.
- Net-relative methods

Pernet_start and Postnet_stop perform special startup or shutdown operations before network addresses in the same resource group are configured to go up or after they are configured to go down.
- Monitor control methods
 - Monitor_start and Monitor_stop start or stop the monitor for a resource.
 - Monitor_check assesses the reliability of a node before a resource group is moved to the node.

See [Chapter 3, “Resource Management API Reference”](#) and the `rt_callbacks(1HA)` man page for more information about the callback methods. Also see [Chapter 5, “Sample Data Service”](#) and [Chapter 8, “Sample DSDL Resource Type Implementation”](#) for callback methods in sample data services.

Programming Interfaces

For writing data service code, the resource management architecture provides a low-level or base API, a higher-level library that is built on top of the base API, and Oracle Solaris Cluster Agent Builder, a tool that automatically generates a data service from basic input that you provide.

Resource Management API

The Resource Management API (RMAPI) provides a set of low-level functions that enable a data service to access information about the resource types, resources, and resource groups in the system, to request a local restart or failover, and to set the resource status. You access these functions through the `libscha.so` library. The RMAPI provides these callback methods both in the form of shell commands and in the form of C functions. See the [scha_calls\(3HA\)](#) man page and [Chapter 3, “Resource Management API Reference”](#) for more information about the RMAPI functions. Also see [Chapter 5, “Sample Data Service”](#) for examples of how to use these functions in sample data service callback methods.

Data Service Development Library

Built on top of the RMAPI is the Data Service Development Library (DSDL), which provides a higher-level integrated framework while retaining the underlying *method-callback model* of the RGM. The `libdsdev.so` library contains the DSDL functions.

The DSDL brings together various facilities for data service development, including the following:

- `libscha.so`. The low-level resource management APIs.
- **PMF**. The Process Monitor Facility (PMF), which provides a means of monitoring processes and their descendants, and restarting them if they die. See the [pmfadm\(1M\)](#) and [rpc.pmfd\(1M\)](#) man pages.
- `/usr/cluster/bin/hatimerun`. A facility for running programs under a timeout. See the [hatimerun\(1M\)](#) man page.

For the majority of applications, the DSDL provides most or all of the functionality you need to build a data service. Note, however, that the DSDL does not replace the low-level API but encapsulates and extends it. In fact, many DSDL functions call the `libscha.so` functions. Likewise, you can directly call `libscha.so` functions while using the DSDL to code the bulk of your data service.

See [Chapter 6, “Data Service Development Library”](#) and the `scha_calls(3HA)` man page for more information about the DSDL.

Oracle Solaris Cluster Agent Builder

Agent Builder is a tool that automates the creation of a data service. You input basic information about the target application and the data service to be created. Agent Builder generates a data service, which includes GDS driver scripts or source and executable code (C or Korn shell) with a customized RTR file and an Oracle Solaris SVR4 package.

For most applications, you can use Agent Builder to generate a complete data service with only minor manual changes on your part. Applications with more sophisticated requirements, such as adding validation checks for additional properties, might require work that Agent Builder cannot do. However, even in these cases, you might be able to use Agent Builder to generate the bulk of the code and manually code the rest. At a minimum, you can use Agent Builder to generate the Oracle Solaris package for you.

Resource Group Manager Administrative Interface

Oracle Solaris Cluster provides both a graphical user interface (GUI) and a set of commands for administering a cluster.

Oracle Solaris Cluster Manager

Oracle Solaris Cluster Manager is a web-based tool that enables you to perform the following tasks:

- Install a cluster.
- Administer a cluster.
- Create and configure resources and resource groups.
- Configure data services with the Oracle Solaris Cluster software.

See the [Oracle Solaris Cluster 4.3 Software Installation Guide](#) for instructions on how to install Oracle Oracle Solaris Cluster Manager and how to use Oracle Solaris Cluster Manager to install cluster software. Oracle Solaris Cluster Manager provides online help for most unique administrative tasks.

clsetup Utility

You can perform most Oracle Solaris Cluster administration tasks interactively with the `clsetup(1CL)` utility.

You can administer the following Oracle Solaris Cluster elements with the `clsetup` utility:

- Quorum
- Resource groups
- Data services
- Cluster interconnect
- Device groups and volumes
- Private host names
- New nodes
- Other cluster tasks

You can also perform the following operations with the `clsetup` utility:

- Create a resource group
- Create a zone cluster
- Add a network resource to a resource group
- Add a data service resource to a resource group
- Add a file system to a zone cluster
- Add a storage device to a zone cluster
- Add a network address to a zone cluster
- Register a resource type
- Bring a resource group online or offline
- Switchover a resource group
- Suspend or resume the automatic recovery actions of a resource group
- Enable or disable a resource
- Change resource group properties
- Change resource properties
- Remove a resource from a resource group
- Remove a resource group
- Clear the `Stop_failed` error flag from a resource

Administrative Commands

The Oracle Solaris Cluster commands for administering RGM objects are `clresourcetype`, `clresourcegroup`, `clresource`, `clnode`, and `cluster`.

The `clresourcetype`, `clresourcegroup`, and `clresource` commands enable you to view, create, configure, and delete a resource type, a resource group, and the resource objects that are used by the RGM. These commands are part of the administrative interface for the cluster, but are not designed to be used in the same programming context as the application interface that is described in the rest of this chapter. However, the `clresourcetype`, `clresourcegroup`, and `clresource` commands are the tools for constructing the cluster configuration in which the API operates. Understanding the administrative interface sets the context for understanding the application interface. See the [clresourcetype\(1CL\)](#), [clresourcegroup\(1CL\)](#), and [clresource\(1CL\)](#) man pages for details about the administrative tasks that you can perform with these commands.

◆◆◆ 2 CHAPTER 2

Developing a Data Service

This chapter tells you how to make an application highly available or scalable, and provides detailed information about developing a data service.

This chapter covers the following topics:

- [“Analyzing the Application for Suitability” on page 25](#)
- [“Determining the Interface to Use” on page 27](#)
- [“Setting Up the Development Environment for Writing a Data Service” on page 28](#)
- [“Setting Standard Properties” on page 30](#)
- [“Node List Properties” on page 31](#)
- [“Setting Resource and Resource Type Properties” on page 32](#)
- [“Implementing Callback Methods” on page 41](#)
- [“Generic Data Service” on page 42](#)
- [“Controlling an Application” on page 43](#)
- [“Monitoring a Resource” on page 48](#)
- [“Adding Message Logging to a Resource” on page 50](#)
- [“Providing Process Management” on page 50](#)
- [“Providing Administrative Support for a Resource” on page 51](#)
- [“Implementing a Failover Resource” on page 52](#)
- [“Implementing a Scalable Resource” on page 53](#)
- [“Writing and Testing Data Services” on page 56](#)
- [“Legal RGM Names” on page 59](#)

Analyzing the Application for Suitability

The first step in creating a data service is to determine whether the target application satisfies the requirements for being made highly available or scalable. If the application fails to meet all requirements, you might be able to modify the application source code to make it highly available or scalable.

The list that follows summarizes the requirements for an application to be made highly available or scalable. If you need more detail or if you need to modify the application source code, see [Appendix A, “Sample Data Service Code Listings”](#).

Note - A scalable service must meet all the following conditions for high availability as well as some additional criteria, which follow the list.

- Both network-aware (client-server model) and non-network-aware (client-less) applications are potential candidates for being made highly available or scalable in the Oracle Solaris Cluster environment. However, Oracle Solaris Cluster cannot provide enhanced availability in timesharing environments in which applications are run on a server that is accessed through telnet or rlogin.
- The application must be crash tolerant. That is, the application must recover disk data (if necessary) when it is started after an unexpected failure of a node. Furthermore, the recovery time after a crash must be bounded. Crash tolerance is a prerequisite for making an application highly available because the ability to recover the disk and restart the application is a data integrity issue. The data service is not required to be able to recover connections.
- The application must not depend upon the physical host name of the node on which it is running. See [“Host Names” on page 287](#) for additional information.
- The application must operate correctly in environments in which multiple IP addresses are configured to go up. Examples include environments with multihomed hosts, in which the node is located on more than one public network, and environments with nodes on which multiple, logical interfaces are configured to go up on one hardware interface.
- To be highly available, the application data must be located on a highly available local file system. See [“Multihosted Data” on page 285](#).

If the application uses a hard-wired path name for the location of the data, you could change that path to a symbolic link that points to a location in the cluster file system, without changing application source code. See [“Using Symbolic Links for Multihosted Data Placement” on page 286](#) for additional information.

- Application binaries and libraries can be located locally on each node or in the cluster file system. The advantage of being located in the cluster file system is that a single installation is sufficient.
- The client should have some capacity to retry a query automatically if the first attempt times out. If the application and the protocol already handle a single server's crashing and rebooting, they also can handle the containing resource group's being failed over or switched over. See [“Client Retry” on page 289](#) for additional information.
- The application must not have UNIX® domain sockets or named pipes in the cluster file system.

Additionally, scalable services must meet the following requirements:

- The application must have the ability to run multiple instances, all operating on the same application data in the cluster file system.
- The application must provide data consistency for simultaneous access from multiple nodes.
- The application must implement sufficient locking with a globally visible mechanism, such as the cluster file system.

For a scalable service, application characteristics also determine the load-balancing policy. For example, the load-balancing policy `Lb_weighted`, which allows any instance to respond to client requests, does not work for an application that makes use of an in-memory cache on the server for client connections. In this case, specify a load-balancing policy that restricts a given client's traffic to one instance of the application. The load-balancing policies `Lb_sticky` and `Lb_sticky_wild` repeatedly send all requests by a client to the same application instance, where they can make use of an in-memory cache. Note that if multiple client requests come in from different clients, the RGM distributes the requests among the instances of the service. See [“Implementing a Failover Resource” on page 52](#) for more information about setting the load-balancing policy for scalable data services.

Determining the Interface to Use

The Oracle Solaris Cluster developer support IPS package (`ha-cluster/developer/api`) provides two sets of interfaces for coding data service methods:

- The Resource Management API (RMAPI), a set of low-level functions (in the `libscha.so` library)
- The Data Services Development Library (DSDL), a set of higher-level functions (in the `libdsdev.so` library) that encapsulate the functionality of the RMAPI and provide some additional functionality

Oracle Solaris Cluster software provides the following additional packages to aid data service development:

- Oracle Solaris Cluster Agent Builder (`ha-cluster/developer/agent-builder`), a tool that automates the creation of a data service. For more information, see [Chapter 9, “Oracle Solaris Cluster Agent Builder”](#).
- The Generic Data Service or GDS (`ha-cluster/ha-service/gds`), a precompiled data service on which to base the data service you develop. For more information, see [Oracle Solaris Cluster Generic Data Service \(GDS\) Guide](#).

Here is the recommended approach to developing a data service:

1. Decide whether to use the Generic Data Service (GDS) method or to code in C or the Korn shell. If you decide to use the Korn shell, you cannot use the DSDL, which provides a C interface only.
2. Run Agent Builder, specify the requested information, and generate a data service, which includes source and executable code, an RTR file, and a package.
3. If the generated data service requires customizing, you can add DSDL code to the generated source files. Agent Builder indicates, with comments, specific places in the source files where you can add your own code.
4. If the code requires further customizing to support the target application, you can add RMAPI functions to the existing source code.

In practice, you can take numerous approaches to creating a data service. For example, rather than add your own code to specific places in the code that is generated by Agent Builder, you could entirely replace one of the generated methods or the generated monitor program with a program that you write from scratch using DSDL or RMAPI functions.

However, regardless of how you proceed, in almost every case, starting with Agent Builder using the GDS method makes sense, for the following reasons:

- The code that is generated by Agent Builder using the GDS method, while generic in nature, has been tested in numerous data services.
- Agent Builder generates an RTR file, a `Makefile`, a package for the resource, and other support files for the data service. Even if you use none of the data service code, using these other files can save you considerable work.
- You can modify the generated code.

Note - Unlike the RMAPI, which provides a set of C functions and a set of commands for use in scripts, the DSDL provides a C function interface only. Therefore, if you specify Korn shell (ksh) output in Agent Builder, the generated source code makes calls to RMAPI because there are no DSDL ksh commands.

Setting Up the Development Environment for Writing a Data Service

Before you begin to develop your data service, you must install the Oracle Solaris Cluster development IPS package (`ha-cluster/developer/api`) to have access to the Oracle Solaris Cluster header and library files. Although this package is already installed on all cluster nodes, you typically develop your data service on a separate, noncluster development machine, rather

than on a cluster node. In this typical case, you must use the `pkg install` command to install the `ha-cluster/developer/api` IPS package on your development machine.

When compiling and linking your code, you must set particular options to identify the header and library files.

Note - You cannot mix compatibility-mode compiled C++ code and standard-mode compiled C++ code in the Oracle Solaris Operating System and Oracle Solaris Cluster products. Consequently, if you intend to create a C++ based data service for use on Oracle Solaris Cluster software, you must use the standard mode to compile that data service.

When you have finished development (on a noncluster node), you can transfer the completed data service to a cluster for testing.

The procedures in this section describe how to complete the following tasks:

- Install the Oracle Solaris Cluster development IPS package (`ha-cluster/developer/api`) and set the correct compiler and linker options.
- Transfer the data service to a cluster.

▼ How to Set Up the Development Environment

This procedure describes how to install the `ha-cluster/developer/api` IPS package and set the compiler and linker options for data service development.

1. **Assume the root role that provides `solaris.cluster.modify` RBAC authorization.**
2. **Ensure that the `solaris` and `ha-cluster` publishers are valid.**

```
# pkg publisher
PUBLISHER          TYPE    STATUS  URI
solaris             origin  online  solaris-repository
ha-cluster          origin  online  ha-cluster-repository
```

For information about setting the `solaris` publisher, see [“Checking and Setting Repository Properties” in *Copying and Creating Package Repositories in Oracle Solaris 11.3*](#).

3. **Install the `ha-cluster/developer/api` package in the current directory.**

```
# pkg install ha-cluster/developer/api
```

4. **Verify that the package installed successfully.**

```
phys-schost% pkg info ha-cluster/developer/api
```

Installation is successful if output shows that State is Installed.

5. In the makefile, specify compiler and linker options that identify the include and library files for your data service code.

Specify the `-I` option to identify the Oracle Solaris Cluster header files, the `-L` option to specify the compile-time library search path on the development system, and the `-R` option to specify the library search path to the runtime linker in the cluster.

```
# Makefile for sample data service
...

-I /usr/cluster/include

-L /usr/cluster/lib

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

Transferring a Data Service to a Cluster

When you have completed the data service on a development machine, you must transfer the data service to a cluster for testing. To reduce the chance of error during the transfer, combine the data service code and the RTR file into a package. Then, install the package on the Oracle Solaris hosts on which you want to run the service.

Note - Agent Builder creates a System V Revision 4 (SVR4) package automatically.

Setting Standard Properties

You can specify standard cluster, resource type, resource, and resource group properties when configuring data services. Resource property attributes are also available to change system-defined properties and create extension properties.

Cluster Properties

Cluster properties are used to manage data services. For information about cluster properties, see the [cluster\(1CL\)](#) man page.

Resource Type Properties

Resource type properties cannot be updated by administrative utilities with the exception of `Installed_nodes` and `RT_system`. `Installed_nodes` cannot be declared in the RTR file and can only be set by the cluster administrator. `RT_system` can be assigned an initial value in the RTR file, and can also be set by the cluster administrator.

For information about each resource type property that is defined by Oracle Solaris Cluster software, see the [rt_properties\(5\)](#) man page.

Resource Properties

For information about each resource property that is defined by Oracle Solaris Cluster software, see the [r_properties\(5\)](#) man page.

Resource Group Properties

For information about each resource group property that is defined by Oracle Solaris Cluster software, see the [rg_properties\(5\)](#) man page.

Resource Property Attributes

You can use resource property attributes to change system-defined properties or to create extension properties. For information about each property, see the [property_attributes\(5\)](#) man page.

Node List Properties

You can specify the following node list properties when configuring data services:

- `Installed_nodes` Property – See the [rt_properties\(5\)](#) man page for more information.
- `Nodelist` Property – See the [rg_properties\(5\)](#) man page for more information.
- `Auxnodelist` Property – See the [clressharedaddress\(1CL\)](#) man page for more information.

Setting Resource and Resource Type Properties

Oracle Solaris Cluster software provides a set of resource type properties and resource properties that you use to define the static configuration of a data service. Resource type properties specify the type of the resource, its version, the version of the API, as well as the paths to each of the callback methods. “[Resource Type Properties](#)” on page 31 lists all the resource type properties.

Resource properties, such as `Failover_mode`, `Thorough_probe_interval`, and method timeouts, also define the static configuration of the resource. Dynamic resource properties, such as `Resource_state` and `Status`, reflect the active state of a managed resource. “[Resource Properties](#)” on page 31 describes the resource properties.

You declare the resource type and resource properties in the resource type registration (RTR) file, which is an essential component of a data service. The RTR file defines the initial configuration of the data service at the time that the cluster administrator registers the data service with the Oracle Solaris Cluster software.

Use Agent Builder to generate the RTR file for your data service. Agent Builder declares the set of properties that are both useful and required for any data service. For example, particular properties, such as `Resource_type`, must be declared in the RTR file. Otherwise, registration of the data service fails. Other properties, although not required, are not available to a cluster administrator unless you declare them in the RTR file. Some properties are available whether you declare them or not because the RGM defines them and provides default values. To avoid this level of complexity, use Agent Builder to guarantee the generation of a correct RTR file. Later, you can edit the RTR file to change specific values if necessary.

The rest of this section shows a sample RTR file, which was created by Agent Builder.

Declaring Resource Type Properties

The cluster administrator cannot configure the resource type properties that you declare in the RTR file. They become part of the permanent configuration of the resource type.

Note - Only a cluster administrator can configure the resource type property `Installed_nodes`. You cannot declare `Installed_nodes` in the RTR file.

The syntax of resource type declarations is as follows:

```
property-name = value;
```

Note - Property names for resource groups, resources, and resource types are *not* case sensitive. You can use any combination of uppercase and lowercase letters when you specify property names.

These are resource type declarations in the RTR file for a sample (smpl) data service:

```
# Oracle Solaris 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 = ORCL;
RT_description = "Sample Service on Oracle Solaris Cluster";

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

Init_nodes = RG_PRIMARYES;

RT_basedir=/opt/ORCLsmpl/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;
```

Tip - You must declare the `Resource_type` property as the first entry in the RTR file. Otherwise, registration of the resource type fails.

The first set of resource type declarations provide basic information about the resource type.

`Resource_type` and `Vendor_id`

Provide a name for the resource type. You can specify the resource type name with the `Resource_type` property alone (smpl) or by using the `Vendor_id` property as a prefix with a period (.) separating it from the resource type (ORCL.smpl), as shown in the sample. If you use `Vendor_id`, make it the stock market symbol of the company that is defining the resource type. The resource type name must be unique in the cluster.

Note - By convention, the resource type name (*vendoridApplicationname*) is used as the package name. The combination of vendor ID and application name can exceed nine characters.

Agent Builder, on the other hand, in all cases explicitly generates the package name from the resource type name, so it enforces the nine-character limit.

`RT_description`

Briefly describes the resource type.

`RT_version`

Identifies the version of the sample data service.

`API_version`

Identifies the version of the API. For example, `API_version = 11` indicates that the data service can be registered on any version of Oracle Solaris Cluster starting with Oracle Solaris Cluster 3.3, assuming that the application is compatible with that version of Oracle Solaris Cluster software. However, `API_version = 11` also indicates that the data service cannot be registered on any version of Oracle Solaris Cluster that was released before Oracle Solaris Cluster 3.3. This property is described in more detail under the entry for `API_version` in [“Resource Type Properties” on page 31](#).

`Failover = FALSE`

Indicates that the data service can run in a resource group that can be online on multiple nodes at the same time. In other words, this declaration specifies a multi-master data service. This property is described in more detail under the entry for `Failover` in [“Resource Type Properties” on page 31](#).

`Start, Stop, and Validate`

Provide the paths to the respective callback method programs that are called by the RGM. These paths are relative to the directory that is specified by `RT_basedir`.

The remaining resource type declarations provide configuration information.

`Init_nodes = RG_PRIMARYES`

Specifies that the RGM call the `Init`, `Boot`, `Fini`, and `Validate` methods only on nodes that can master the data service. The nodes that are specified by `RG_PRIMARYES` are a subset of all nodes on which the data service is installed. Set the value to `RT_INSTALLED_NODES` to specify that the RGM call these methods on all nodes on which the data service is installed.

`RT_basedir`

Points to `/opt/ORCLsample/bin` as the directory path to complete relative paths, such as callback method paths.

Start, Stop, and Validate

Provide the paths to the respective callback method programs that are called by the RGM. These paths are relative to the directory that is specified by `RT_basedir`.

See the [rt_properties\(5\)](#) man page for more information about resource type properties. See the [clresourcetype\(1CL\)](#) man page for more information about registering resource types in the global cluster or in a zone cluster.

Declaring Resource Properties

As with resource type properties, you declare resource properties in the RTR file. By convention, resource property declarations follow the resource type declarations in the RTR file. The syntax for resource declarations is a set of attribute value pairs enclosed by braces (`{ }`):

```
{
attribute = value;
attribute = value;
.
.
.
attribute = value;
}
```

For resource properties that are provided by Oracle Solaris Cluster, which are called *system-defined* properties, you can change specific attributes in the RTR file. For example, Oracle Solaris Cluster provides default values for method timeout properties for each callback method. In the RTR file, you can specify different default values.

You can also define new resource properties in the RTR file, which are called *extension* properties, by using a set of property attributes that are provided by Oracle Solaris Cluster. [“Resource Property Attributes” on page 31](#) lists the attributes for changing and defining resource properties. Extension property declarations follow the system-defined property declarations in the RTR file.

The first set of system-defined resource properties specifies timeout values for the callback methods.

...

```
# Resource property declarations appear as a list of bracketed
# entries after the resource type declarations. The property
# name declaration must be the first attribute after the open
# curly bracket of a resource property entry.
#
# Set minimum and default for method timeouts.
```

```
{
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;
}
```

The name of the property (`PROPERTY = value`) must be the first attribute for each resource-property declaration. You can configure resource properties within limits that are defined by the property attributes in the RTR file. For example, the default value for each method timeout in the sample is 300 seconds. The cluster administrator can change this value. However, the minimum allowable value, specified by the `MIN` attribute, is 60 seconds. [“Resource Property Attributes” on page 31](#) contains a list of resource property attributes.

The next set of resource properties defines properties that have specific uses in the data service.

```
{
PROPERTY = Failover_mode;
DEFAULT=SOFT;
```

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

# The number of retries to be done within a certain period before concluding
# that the application cannot be successfully started on this node.
{
PROPERTY = Retry_count;
MAX=10;
DEFAULT=2;
TUNABLE = ANYTIME;
}

# Set Retry_interval as a multiple of 60 since it is converted from seconds
# to minutes, rounding up. For example, a value of 50 (seconds)
# is converted to 1 minute. Use this property to time the number of
# retries (Retry_count).
{
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 = ANYTIME;
DEFAULT = ;
}
```

These resource-property declarations include the TUNABLE attribute. This attribute limits the occasions on which the cluster administrator can change the value of the property with which this attribute is associated. For example, the value AT_CREATION means that the cluster administrator can only specify the value when the resource is created and cannot change the value later.

For most of these properties, you can accept the default values as generated by Agent Builder unless you have a reason to change them. Information about these properties follows. For additional information, see [“Resource Properties” on page 31](#) or the `r_properties(5)` man page.

`Failover_mode` Indicates whether the RGM should relocate the resource group or abort the node in the case of a failure of a Start or Stop method.

`Thorough_probe_interval`, `Retry_count`, and `Retry_interval` Used in the fault monitor. Tunable equals ANYTIME, so a cluster administrator can adjust them if the fault monitor is not functioning optimally.

`Network_resources_used` A list of logical-hostname or shared-address resources on which this resource has a dependency. This list contains all network-address resources that appear in the properties `Resource_dependencies`, `Resource_dependencies_weak`, `Resource_dependencies_restart`, or `Resource_dependencies_offline_restart`.

The RGM automatically creates this property if the `Scalable` property is declared in the RTR file. If the `Scalable` property is not declared in the RTR file, `Network_resources_used` is unavailable unless it is explicitly declared in the RTR file.

If you do not assign a value to the `Network_resources_used` property, its value is updated automatically by the RGM, based on the setting of the resource-dependencies properties. You do not need to set this property directly. Instead, set the `Resource_dependencies`, `Resource_dependencies_offline_restart`, `Resource_dependencies_restart`, or `Resource_dependencies_weak` property. If per-node dependencies are specified, the derived value of the `Network_resources_used` property includes only those dependencies

which are in effect on the local node. The value might differ on each node.

For simplicity, avoid setting a value for the `Network_resources_used` property. Set only the resource dependency properties, and treat the `Network_resources_used` property as a read-only property. For more information, see the [r_properties\(5\)](#) man page.

<code>Scalable</code>	<p>If set to TRUE, indicates that the resource uses the network load balancing feature of Oracle Solaris Cluster software. In this case, the resource must be configured with a dependency upon a shared address resource. See “Implementing a Scalable Resource” on page 53 for additional information about how to use this property.</p> <p>In this example, the tunability of the <code>Scalable</code> property is declared to be <code>AT_CREATION</code>, which means that this property may be set to either TRUE or FALSE when the resource is created. This implies that the resource type implementation is capable of supporting either a scalable or non-scalable mode of operation for the resource.</p>
<code>Load_balancing_policy</code> and <code>Load_balancing_weights</code>	<p>These properties are declared automatically when the <code>Scalable</code> property is declared. Therefore, their appearance in the RTR file is optional unless you want to override the default tunability. These properties are used only when the <code>Scalable</code> property is set to TRUE.</p>
<code>Port_list</code>	<p>Identifies the list of ports on which the application is listening. Agent Builder declares this property so that a cluster administrator can specify a list of ports when the cluster administrator configures the data service.</p>

Declaring Extension Properties

Extension properties appear at the end of the sample RTR file.

```
# Extension Properties
#
# The cluster administrator must set the value of this property to point to the
# directory that contains the configuration files used by the application.
# For this application, smpl, specify the path of the configuration file on
# PXFS (typically named conf).
{
PROPERTY = Confdir_list;
EXTENSION;
STRINGARRAY;
```

```

TUNABLE = AT_CREATION;
DESCRIPTION = "The Configuration Directory Path(s)";
}

# The following two properties control restart of the fault monitor.
{
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.";
}
# Time out value in seconds for the probe.
{
PROPERTY = Probe_timeout;
EXTENSION;
INT;
DEFAULT = 120;
TUNABLE = ANYTIME;
DESCRIPTION = "Time out value for the probe (seconds)";
}

# Child process monitoring level for PMF (-C option of pmfadm).
# Default of -1 means to not use the -C option of pmfadm.
# A value of 0 or greater indicates the desired level of child-process.
# monitoring.
{
PROPERTY = Child_mon_level;
EXTENSION;
INT;
DEFAULT = -1;
TUNABLE = ANYTIME;
DESCRIPTION = "Child monitoring level for PMF";
}
# User added code -- BEGIN VVVVVVVVVVVV
# User added code -- END   ^^^^^^^^^^^^^

```

Agent Builder creates the following extension properties, which are useful for most data services.

`Confdir_list` Specifies the path to the application configuration directory, which is useful information for many applications. The cluster administrator can provide the location of this directory when the cluster administrator configures the data service.

`Monitor_retry_count`, `Monitor_retry_interval`, and `Probe_timeout` Controls the restarts of the fault monitor itself, not the server daemon.

`Child_mon_level` Sets the level of monitoring to be carried out by the PMF. See the [pmfadm\(1M\)](#) man page for more information.

You can create additional extension properties in the area that is delimited by the `User` added code comments.

Implementing Callback Methods

This section provides general information that pertains to implementing the callback methods.

Accessing Resource and Resource Group Property Information

Generally, callback methods require access to the properties of the resource. The RMAPI provides both shell commands and C functions that you can use in callback methods to access the system-defined and extension properties of resources. See the [scha_resource_open\(3HA\)](#) man page.

The DSDL provides a set of C functions (one function for each property) to access system-defined properties, and a function to access extension properties. See the [scds_property_functions\(3HA\)](#) and [scds_get_ext_property\(3HA\)](#) man pages.

You cannot use the property mechanism to store dynamic state information for a data service because no API functions are available for setting resource properties other than `Status` and `Status_msg`. Rather, you should store dynamic state information in global files.

Note - The cluster administrator can set particular resource properties by using the `clresource` command or through a graphical administrative command or interface. However, do not call `clresource` from any callback method because `clresource` fails during cluster reconfiguration, that is, when the RGM calls the method.

Idempotence of Methods

In general, the RGM does not call a method more than once in succession on the same resource with the same arguments. However, if a Start method fails, the RGM can call a Stop method on a resource even though the resource was never started. Likewise, a resource daemon could die of its own accord and the RGM might still run its Stop method on it. The same scenarios apply to the Monitor_start and Monitor_stop methods.

For these reasons, you must build *idempotence* into your Stop and Monitor_stop methods. In other words, repeated calls to Stop or Monitor_stop on the same resource with the same arguments must achieve the same results as a single call.

One implication of idempotence is that Stop and Monitor_stop must return 0 (success) even if the resource or monitor is already stopped and no work is to be done.

Note - The Init, Fini, Boot, and Update methods must also be idempotent. A Start method need not be idempotent.

How Methods Are Invoked in Zones

If declared in the RTR file, the Global_zone resource type property indicates whether the methods of a resource type execute in the global zone. If the Global_zone property equals TRUE, methods execute in the global zone.

If the resource is configured in the global zone, the `-Z zonename` option is not invoked.

The Global_zone resource type property is described in more detail in [“Resource Type Properties” on page 31](#) and in the `rt_properties(5)` man page.

Generic Data Service

A generic data service (GDS) is a mechanism for making simple applications highly available or scalable by plugging them into the Oracle Solaris Cluster Resource Group Manager (RGM) framework. This mechanism does not require the coding of a data service, which is the typical approach for making an application highly available or scalable.

The GDS model relies on a precompiled resource type, `SUNW.gds`, to interact with the RGM framework. See [Oracle Solaris Cluster Generic Data Service \(GDS\) Guide](#) for additional information.

Controlling an Application

Callback methods enable the RGM to take control of the underlying resource (that is, the application). For example, callback methods enable the RGM to take control of the underlying resource when a node joins or leaves the cluster.

Starting and Stopping a Resource

A resource type implementation requires, at a minimum, a `Start` method and a `Stop` method.

Using Start and Stop Methods

The RGM calls a resource type's method programs at correct times and on the correct nodes for bringing resource groups offline and online. For example, after the crash of a cluster node, the RGM moves any resource groups that are mastered by that node onto a new node. In this case, you must implement a `Start` method to provide the RGM with, among other things, a way of restarting each resource on the surviving host node.

A `Start` method must not return until the resource has been started and is available on the local node. Be certain that resource types that require a long initialization period have sufficiently long timeouts set on their `Start` methods. To ensure sufficient timeouts, set the default and minimum values for the `Start_timeout` property in the RTR file.

You must implement a `Stop` method for situations in which the RGM takes a resource group offline. For example, suppose a resource group is taken offline in ZoneA on Host1 and brought back online in ZoneB on Host2. While taking the resource group offline, the RGM calls the `Stop` method on resources in the resource group to stop all activity in ZoneA on Host1. After the `Stop` methods for all resources have completed in ZoneA on Host1, the RGM brings the resource group back online in ZoneB on Host2.

A `Stop` method must not return until the resource has completely stopped all its activity on the local node and has completely shut down. The safest implementation of a `Stop` method terminates all processes on the local node that are related to the resource. Resource types that require a long time to shut down need sufficiently long timeouts set on their `Stop` methods. Set the `Stop_timeout` property in the RTR file.

If an RGM method callback times out, the method's process group is killed by a `SIGTERM` signal. If the process group does not exit within 10 seconds, a `SIGKILL` is executed.

Although method core dumps are not generated when a method times out, system core dumps are generated when a node is rebooted by the RGM due to a resource's `Failover_mode` property

being set to `HARD` and the resource method being stuck in the kernel and failing to exit after a `SIGKILL`.

Note - Avoid writing data service methods that create a new process group. If your data service method must create a new process group, write a signal handler for the `SIGTERM` and `SIGABRT` signals. Also, ensure that your signal handler forwards the `SIGTERM` or `SIGABRT` signal to the child process group or groups before the signal handler terminates the process. Writing a signal handler for these signals increases the likelihood that all processes that are spawned by your method are correctly terminated.

Failure or timeout of a `Stop` method causes the resource group to enter an error state that requires the cluster administrator to intervene. To avoid this state, the `Stop` and `Monitor_stop` method implementations must attempt to recover from all possible error conditions. Ideally, these methods must exit with 0 (success) error status, having successfully stopped all activity of the resource and its monitor on the local node.

Deciding Which Start and Stop Methods to Use

This section provides some tips about when to use the `Start` and `Stop` methods as opposed to using the `Prenet_start` and `Postnet_stop` methods. You must have in-depth knowledge of both the client and the data service's client-server networking protocol to decide the correct methods to use.

Services that use network address resources might require that start or stop steps be done in a particular order. This order must be relative to the logical host name address configuration. The optional callback methods `Prenet_start` and `Postnet_stop` enable a resource type implementation to perform special startup and shutdown operations before and after network addresses in the same resource group are configured to go up or configured to go down.

The RGM calls methods that plumb the network addresses (but do not configure network addresses to go up) before calling the data service's `Prenet_start` method. The RGM calls methods that unplug the network addresses after calling the data service's `Postnet_stop` methods.

The sequence is as follows when the RGM takes a resource group online:

1. Plumb network addresses.
2. Call the data service's `Prenet_start` method (if any).
3. Configure network addresses to go up.
4. Call the data service's `Start` method (if any).

The reverse happens when the RGM takes a resource group offline:

1. Call the data service's `Stop` method (if any).

2. Configure network addresses to go down.
3. Call the data service's `Postnet_stop` method (if any).
4. Unplumb network addresses.

When deciding whether to use the `Start`, `Stop`, `Prenet_start`, or `Postnet_stop` methods, first consider the server side. When bringing online a resource group that contains both data service application resources and network address resources, the RGM calls methods to configure the network addresses to go up before it calls the data service resource `Start` methods. Therefore, if a data service requires network addresses to be configured to go up at the time it starts, use the `Start` method to start the data service.

Likewise, when bringing offline a resource group that contains both data service resources and network address resources, the RGM calls methods to configure the network addresses to go down after it calls the data service resource `Stop` methods. Therefore, if a data service requires network addresses to be configured to go up at the time it stops, use the `Stop` method to stop the data service.

For example, to start or stop a data service, you might have to run the data service's administrative utilities or libraries. Sometimes, the data service has administrative utilities or libraries that use a client-server networking interface to perform the administration. That is, an administrative utility makes a call to the server daemon, so the network address might need to be up to use the administrative utility or library. Use the `Start` and `Stop` methods in this scenario.

If the data service requires that the network addresses be configured to go down at the time it starts and stops, use the `Prenet_start` and `Postnet_stop` methods to start and stop the data service. Consider whether your client software is to respond differently, depending on whether the network address or the data service comes online first after a cluster reconfiguration (either `scha_control()` with the `SCHA_GIVEOVER` argument or a switchover with the `clnode evacuate` command). For example, the client implementation might perform the fewest retries, giving up soon after determining that the data service port is not available.

If the data service does not require the network address to be configured to go up when it starts, start the data service before the network interface is configured to go up. Starting the data service in this way ensures that the data service is able to respond immediately to client requests as soon as the network address has been configured to go up. As a result, clients are less likely to stop retrying. In this scenario, use the `Prenet_start` method rather than the `Start` method to start the data service.

If you use the `Postnet_stop` method, the data service resource is still up at the point the network address is configured to be down. Only after the network address is configured to go down is the `Postnet_stop` method run. As a result, the data service's TCP or UDP service port, or its RPC program number, always appears to be available to clients on the network, except when the network address is also not responding.

Note - If you install an RPC service in the cluster, the service must not use the following program numbers: 100141, 100142, and 100248. These numbers are reserved for the Oracle Solaris Cluster daemons `rgmd_receptionist`, `fed`, and `pmfd`, respectively. If the RPC service that you install uses one of these program numbers, change the program number of that RPC service.

The decision to use the `Start` and `Stop` methods as opposed to the `Prenet_start` and `Postnet_stop` methods, or to use both, must take into account the requirements and behavior of both the server and client.

Using the Optional `Init`, `Fini`, and `Boot` Methods

Three optional methods, `Init`, `Fini`, and `Boot`, enable the RGM to execute initialization and termination code on a resource.

Using the `Init` Method

The RGM executes the `Init` method to perform a one-time initialization of the resource when the resource becomes managed as a result of one of the following conditions:

- The resource group in which the resource is located is switched from an unmanaged to a managed state.
- The resource is created in a resource group that is already managed.

Using the `Fini` Method

The RGM executes the `Fini` method to clean up after a resource when that resource is no longer managed by the RGM. The `Fini` method usually undoes any initializations that were performed by the `Init` method.

The RGM executes `Fini` on the node where the resource becomes unmanaged when the following situations arise:

- The resource group that contains the resource is switched to an unmanaged state. In this case, the RGM executes the `Fini` method on all nodes in the node list.
- The resource is deleted from a managed resource group. In this case, the RGM executes the `Fini` method on all nodes in the node list.
- A node is deleted from the node list of the resource group that contains the resource. In this case, the RGM executes the `Fini` method on only the deleted node.

A “node list” is either the resource group's `Nodelist` or the resource type's `Installed_nodes` list. Whether “node list” refers to the resource group's `Nodelist` or the resource type's `Installed_nodes` list depends on the setting of the resource type's `Init_nodes` property. You can set the `Init_nodes` property to `RG_PRIMARYES` or `RT_INSTALLED_NODE`. For most resource types, `Init_nodes` is set to `RG_PRIMARYES`, the default. In this case, both the `Init` and `Fini` methods are executed on the nodes that are specified in the resource group's `Nodelist`.

The type of initialization that the `Init` method performs defines the type of cleanup that the `Fini` method that you implement needs to perform, as follows:

- Cleanup of node-specific configuration.
- Cleanup of cluster-wide configuration.

Guidelines for Implementing a `Fini` Method

The `Fini` method that you implement needs to determine whether to perform only cleanup of node-specific configuration or cleanup of both node-specific and cluster-wide configuration.

When a resource becomes unmanaged on only a particular node, the `Fini` method can clean up local, node-specific configuration. However, the `Fini` method must not clean up global, cluster-wide configuration, because the resource remains managed on other nodes. If the resource becomes unmanaged cluster-wide, the `Fini` method can perform cleanup of both node-specific and global configuration. Your `Fini` method code can distinguish these two cases by determining whether the resource group's node list contains the local node on which your `Fini` method is executing.

If the local node appears in the resource group's node list, the resource is being deleted or is moving to an unmanaged state. The resource is no longer active on any node. In this case, your `Fini` method needs to clean up any node-specific configuration on the local node as well as cluster-wide configuration.

If the local node does not appear in the resource group's node list, your `Fini` method can clean up node-specific configuration on the local node. However, your `Fini` method must not clean up cluster-wide configuration. In this case, the resource remains active on other nodes.

You must also code the `Fini` method so that it is idempotent. In other words, even if the `Fini` method has cleaned up a resource during a previous execution, subsequent calls to the `Fini` method exit successfully.

Using the `Boot` Method

The RGM executes the `Boot` method on nodes that join the cluster, that is, that have just been booted or rebooted.

The `Boot` method normally performs the same initialization as `Init`. You must code the `Boot` method so that it is idempotent. In other words, even if the `Boot` method has initialized the resource during a previous execution, subsequent calls to the `Boot` method exit successfully.

Monitoring a Resource

Typically, you implement monitors to run periodic fault probes on resources to detect whether the probed resources are working correctly. If a fault probe fails, the monitor can attempt to restart locally or request failover of the affected resource group. The monitor requests the failover by calling the `scha_control()` or `scha_control_zone()` RMAPI function or the `scds_fm_action()` DSDL function.

You can also monitor the performance of a resource and tune or report performance. Writing a resource type-specific fault monitor is optional. Even if you choose not to write such a fault monitor, the resource type benefits from the basic monitoring of the cluster that Oracle Solaris Cluster software itself does. Oracle Solaris Cluster software detects failures of the host hardware, gross failures of the host's operating system, and failures of a host to be able to communicate on its public networks.

The RGM provides callbacks for automatically starting and stopping resource monitors. When bringing a resource offline, the RGM calls the `Monitor_stop` method to stop the resource's monitor on the local nodes before stopping the resource itself. When bringing a resource online, the RGM calls the `Monitor_start` method after the resource itself has been started.

The `scha_control()` or `scha_control_zone()` RMAPI function and the `scds_fm_action()` DSDL function (which calls `scha_control()`) enable resource monitors to request the failover of a resource group to a different node. As one of its sanity checks, `scha_control()` and `scha_control_zone()` call `Monitor_check` (if defined) to determine whether the requested node is reliable enough to master the resource group that contains the resource. If `Monitor_check` reports back that the node is not reliable, or the method times out, the RGM looks for a different node to honor the failover request. If `Monitor_check` fails on all nodes, the failover is canceled.

The resource monitor can set the `Status` and `Status_msg` properties to reflect the monitor's view of the resource state. Use the `scha_resource_setstatus()` or `scha_resource_setstatus_zone()` RMAPI function, the `scha_resource_setstatus` command, or the `scds_fm_action()` DSDL function to set these properties.

Note - Although the `Status` and `Status_msg` properties are of particular use to a resource monitor, any program can set these properties.

See “[Defining a Fault Monitor](#)” on page 103 for an example of a fault monitor that is implemented with the RMAPI. See “[ORCL.xfnts Fault Monitor](#)” on page 154 for an example of a fault monitor that is implemented with the DSDL. See the *Oracle Solaris Cluster 4.3 Data Services Planning and Administration Guide* for information about fault monitors that are built into data services that are supplied by Oracle.

Implementing Monitors and Methods That Execute Exclusively in the Global Zone

Most resource types execute their methods in whatever node appears in the resource group's node list. A few resource types must execute all of their methods in the global zone, even when the resource group is configured in a zone-cluster node. This is necessary for resource types that manage system resources, such as network addresses or disks, which can only be managed from the global zone. These resource types are identified by setting the `Global_zone` property to `TRUE` in the resource type registration (RTR) file.



Caution - Do not register a resource type for which the `Global_zone` property is set to `TRUE` unless the resource type comes from a known and trusted source. Resource types for which this property is set to `TRUE` circumvent zone isolation and present a risk.

A resource type that declares `Global_zone=TRUE` might also declare the `Global_zone_override` resource property. In this case, the value of the `Global_zone_override` property supersedes the value of the `Global_zone` property for that resource. The `Global_zone_override` property is described in more detail in “[Resource Properties](#)” on page 31 and the `r_properties(5)` man page.

If the `Global_zone` resource type property is not set to `TRUE`, monitors and methods execute in whatever nodes are listed in the resource group's node list.

The `scha_control()` and `scha_resource_setstatus()` functions and the `scha_control` and `scha_resource_setstatus` commands operate implicitly on the node from which the function or command is run. If the `Global_zone` resource type property equals `TRUE`, these functions and commands need to be invoked differently when the resource is configured in a zone cluster.

When the resource is configured in a zone-cluster node, the value of the `zonename` operand is passed to the resource type method by the `-Z` option. If your method or monitor invokes one of these functions or commands without the correct handling, it incorrectly operates on the global zone. Your method or monitor should operate on the zone-cluster node in which the resource that is included in the resource group's node list is configured.

To ensure that your method or monitor code handles these conditions correctly, check that it does the following:

- Specifies the `-Z zonename` option in calls to the `scha_control` and `scha_resource_setstatus` commands. Use the same value for `zonename` that the RGM passes to the data service method with the `-Z` option.
- Includes calls to the `scha_control_zone()` function rather than to the `scha_control()` function. Ensure that your call passes the `zonename` operand that was passed by the `-Z` option.
- Includes calls to the `scha_resource_setstatus_zone()` function rather than to the `scha_resource_setstatus()` function. Ensure that your call passes the `zonename` operand that was passed by the `-Z` option.

The DSDL functions inherently handle the `-Z zonename` option.

You can use the DSDL function `scds_get_zone_name()` to query the name of the zone that is passed to the method in the `-Z zonename` command-line option. If no `-Z zonename` is passed, the `scds_get_zone_name()` function returns `NULL`.

Adding Message Logging to a Resource

If you want to record status messages in the same log file as other cluster messages, use the convenience function `scha_cluster_getlogfacility()` to retrieve the facility number that is being used to log cluster messages.

Use this facility number with the regular Oracle Solaris `syslog()` function to write messages to the cluster log. You can also access the cluster log facility information through the generic `scha_cluster_get()` interface.

Providing Process Management

The RMAPI and the DSDL provide process management facilities to implement resource monitors and resource control callbacks. The RMAPI defines the following facilities:

Process Monitor Facility (PMF): <code>pmfadm</code> and <code>rpc.pmf</code>	Provides a means of monitoring processes and their descendants, and restarting processes if they die. The facility consists of the <code>pmfadm</code> command for starting and controlling monitored processes, and the <code>rpc.pmf</code> daemon.
---	---

The DSDL provides a set of functions (preceded by the name `scds_pmf_`) to implement the PMF functionality. See “[PMF Functions](#)” on page 191 for an overview of the DSDL PMF functionality and for a list of the individual functions.

The [pmfadm\(1M\)](#) and [rpc.pmf\(1M\)](#) man pages describe this command and daemon in more detail.

`/usr/cluster/
bin/halockrun`

A program for running a child program while holding a file lock. This command is convenient to use in shell scripts.

The [halockrun\(1M\)](#) man page describes this command in more detail.

`/usr/cluster/
bin/hatimerun`

A program for running a child program under timeout control. This command is convenient to use in shell scripts.

The DSDL provides the `scds_hatimerun()` function to implement the features of the `hatimerun` command.

The [hatimerun\(1M\)](#) man page describes this command in more detail.

Providing Administrative Support for a Resource

Actions that cluster administrators perform on resources include setting and changing resource properties. The API defines the `Validate` and `Update` callback methods so that you can create code that hooks into these administrative actions.

The RGM calls the optional `Validate` method when a resource is created. The RGM also calls the `Validate` method when a cluster administrator updates the properties of the resource or its containing group. The RGM passes the property values for the resource and its resource group to the `Validate` method. The RGM calls `Validate` on the set of cluster nodes that is indicated by the `Init_nodes` property of the resource's type. See “[Resource Type Properties](#)” on page 31 or the [rt_properties\(5\)](#) man page for information about `Init_nodes`. The RGM calls `Validate` before the creation or the update is applied. A failure exit code from the method on any node causes the creation or the update to fail.

The RGM calls `Validate` only when the cluster administrator changes resource or resource group properties, not when the RGM sets properties, or when a monitor sets the `Status` and `Status_msg` resource properties.

The RGM calls the optional `Update` method to notify a running resource that properties have been changed. The RGM runs `Update` after the cluster administrator succeeds in setting properties of a resource or its group. The RGM calls this method on nodes where the resource is

online. This method can use the API access functions to read property values that might affect an active resource and adjust the running resource accordingly.

Implementing a Failover Resource

A failover resource group contains network addresses, such as the built-in resource types `LogicalHostname` and `SharedAddress`, and failover resources, such as the data service application resources for a failover data service. The network address resources, along with their dependent data service resources, move between cluster nodes when data services fail over or are switched over. The RGM provides a number of properties that support implementation of a failover resource.

In a global cluster, a failover resource group can fail over to a node on another Oracle Solaris host or on the same Oracle Solaris host. A failover resource group cannot fail over in this way in a zone cluster. However, if the host fails, the failing over of this resource group to a node on the same host does not provide high availability. Nonetheless, you might find this failing over of a resource group to a node on the same host useful in testing or prototyping.

Set the Boolean `Failover` resource type property to `TRUE` to restrict the resource from being configured in a resource group that can be online on more than one node at a time. The default for this property is `FALSE`, so you must declare it as `TRUE` in the RTR file for a failover resource.

The `Scalable` resource property determines if the resource uses the cluster shared address facility. For a failover resource, set `Scalable` to `FALSE` because a failover resource does not use shared addresses.

The `RG_mode` resource group property enables the cluster administrator to identify a resource group as failover or scalable. If `RG_mode` is `FAILOVER`, the RGM sets the `Maximum primaries` property of the group to 1. The RGM also restricts the resource group to being mastered by a single node. The RGM does not allow a resource whose `Failover` property is `TRUE` to be created in a resource group whose `RG_mode` is `SCALABLE`.

The `Implicit_network_dependencies` resource group property specifies that the RGM should enforce implicit strong dependencies of non-network address resources on all network address resources (`LogicalHostname` and `SharedAddress`) within the group. As a result, the `Start` methods of the non-network address (data service) resources in the group are not called until the network addresses in the group are configured to go up. The `Implicit_network_dependencies` property defaults to `TRUE`.

Implementing a Scalable Resource

A scalable resource can be online on more than one node simultaneously. Scalable resources include data services such as Oracle Solaris Cluster HA for Oracle GlassFish Server and Oracle Solaris Cluster HA for Apache.

The RGM provides a number of properties that support the implementation of a scalable resource.

Set the Boolean `Failover` resource type property to `FALSE`, to allow the resource to be configured in a resource group that can be online on more than one node at a time.

The `Scalable` resource property determines if the resource uses the cluster shared address facility. Set this property to `TRUE` because a scalable service uses a shared address resource to make the multiple instances of the scalable service appear as a single service to the client.

The `RG_mode` property enables the cluster administrator to identify a resource group as failover or scalable. If `RG_mode` is `SCALABLE`, the RGM allows `Maximum primaries` to be assigned a value greater than 1. The resource group can be mastered by multiple nodes simultaneously. The RGM allows a resource whose `Failover` property is `FALSE` to be instantiated in a resource group whose `RG_mode` is `SCALABLE`.

The cluster administrator creates a scalable resource group to contain scalable service resources and a separate failover resource group to contain the shared address resources upon which the scalable resource depends.

The resource group that contains the scalable service resource has an implicit dependency on the resource group that contains the shared address resource. This dependency ensures that the network address is brought up before the scalable data service is started. The cluster administrator need not set any explicit dependency for this.

When you declare the `Scalable` property in the RTR file for a resource, the RGM automatically creates the following set of scalable properties for the resource.

`Network_resources_used`

Identifies the shared-address resources on which this resource has a dependency. This list contains all network-address resources that appear in the properties `Resource_dependencies`, `Resource_dependencies_weak`, `Resource_dependencies_restart`, or `Resource_dependencies_offline_restart`.

The RGM automatically creates this property if the `Scalable` property is declared in the RTR file. If the `Scalable` property is not declared in the RTR file, `Network_resources_used` is unavailable unless it is explicitly declared in the RTR file.

If you do not assign a value to the `Network_resources_used` property, its value is updated automatically by the RGM, based on the setting of the resource-dependencies properties. You do not need to set this property directly. Instead, set the `Resource_dependencies`, `Resource_dependencies_offline_restart`, `Resource_dependencies_restart`, or `Resource_dependencies_weak` property.

`Load_balancing_policy`

Specifies the load-balancing policy for the resource. You can explicitly set the policy in the RTR file (or allow the default `LB_WEIGHTED`). In either case, the cluster administrator can change the value when he or she creates the resource (unless you set `Tunable` for `Load_balancing_policy` to `NONE` or `FALSE` in the RTR file). These are the legal values that you can use:

`LB_WEIGHTED`

The load is distributed among various nodes according to the weights that are set in the `Load_balancing_weights` property.

`LB_STICKY`

A given client (identified by the client IP address) of the scalable service is always sent to the same node of the cluster.

`LB_STICKY_WILD`

A given client (identified by the client's IP address) that connects to an IP address of a wildcard sticky service is always sent to the same cluster node regardless of the port number to which it is coming.

For a scalable service with a `Load_balancing_policy` of `LB_STICKY` or `LB_STICKY_WILD`, changing `Load_balancing_weights` while the service is online can cause existing client affinities to be reset. In this case, a different node might service a subsequent client request, even if the client had been previously serviced by another node in the cluster.

Similarly, starting a new instance of the service on a cluster might reset existing client affinities.

`Load_balancing_weights`

Specifies the load to be sent to each node. The format is *weight@node, weight@node*. *weight* is an integer that reflects the relative portion of load that is distributed to the specified *node*. The fraction of load that is distributed to a node is the weight for this node divided by the sum of all weights of active instances. For example, `1@1, 3@2` specifies that node 1 receives one-fourth of the load and node 2 receives three-fourths of the load.

Port_list

Identifies the ports on which the application is listening. This property defaults to the empty string. You can provide a list of ports in the RTR file. Otherwise, the cluster administrator must provide the actual list of ports when creating the resource.

You can create a data service that the cluster administrator can configure to be either scalable or failover. To do so, declare both the `Failover` resource type property and the `Scalable` resource property as `FALSE` in the data service's RTR file. Specify the `Scalable` property to be tunable at creation.

The `Failover` property value `FALSE` allows the resource to be configured in a scalable resource group. The cluster administrator can enable shared addresses by changing the value of `Scalable` to `TRUE` when he or she creates the resource, to create a scalable service.

On the other hand, even though `Failover` is set to `FALSE`, the cluster administrator can configure the resource in a failover resource group to implement a failover service. The cluster administrator does not change the value of `Scalable`, which is `FALSE`. To support this scenario, you should provide a check in the `Validate` method on the `Scalable` property. If `Scalable` is `FALSE`, verify that the resource is configured into a failover resource group.

The [Oracle Solaris Cluster 4.3 Concepts Guide](#) contains additional information about scalable resources.

Validation Checks for Scalable Services

Whenever you create or update a resource with the `scalable` property set to `TRUE`, the RGM validates various resource properties. If you do not configure the properties correctly, the RGM rejects the attempted update or creation.

The RGM performs the following checks:

- The scalable resource must declare a resource dependency on one or more existing shared address resources.

Every node in the `NodeList` for the resource group that contains the scalable resource must appear in the `NetIfList` property of the `SharedAddress` resource.

The `NodeList` of the scalable resource group must be a subset of, or the same as, the combination, or union, of the following two node lists:

- The `NodeList` for the resource group that contains the `SharedAddress` resource.
- The `NodeList` that is listed in the `AuxNodeList` property for the `SharedAddress` resource.

Note - If you include all nodes in the node list for the scalable resource's resource group in the node list for the shared address' resource group, you do not need to set the `AuxNodeList` property.

- The `Port_list` property must not be empty and must contain a list of port-protocol pairs. You must append a slash (/) to each port number, followed by the protocol that is being used by that port. For example:

```
Port_list=80/tcp6,40/udp6
```

You can specify the following protocol values:

- `tcp`, for TCP IPv4
- `tcp6`, for TCP IPv6
- `udp`, for UDP IPv4
- `udp6`, for UDP IPv6

Writing and Testing Data Services

This section describes how to write and test a data service. Topics that are covered include using TCP keep-alives to protect the server, testing highly available data services, and coordinating dependencies between resources.

Using TCP Keep-Alives to Protect the Server

On the server side, using TCP keep-alives protects the server from wasting system resources for a down (or network-partitioned) client. If these resources are not cleaned up in a server that stays up long enough, the wasted resources eventually grow without bound as clients crash and reboot.

If the client-server communication uses a TCP stream, both the client and the server should enable the TCP keep-alive mechanism. This provision applies even in the non-HA, single-server case.

Other connection-oriented protocols might also have a keep-alive mechanism.

On the client side, using TCP keep-alives enables the client to be notified when a network address resource has failed over or switched over from one physical host to another physical host. That transfer of the network address resource breaks the TCP connection. However, unless

the client has enabled the keep-alive, it does not necessarily learn of the connection break if the connection happens to be quiescent at the time.

For example, suppose the client is waiting for a response from the server to a long-running request, and the client's request message has already arrived at the server and has been acknowledged at the TCP layer. In this situation, the client's TCP module has no need to keep retransmitting the request. Also, the client application is blocked, waiting for a response to the request.

Where possible, in addition to using the TCP keep-alive mechanism, the client application also must perform its own periodic keep-alive at its level. The TCP keep-alive mechanism is not perfect in all possible boundary cases. Using an application-level keep-alive typically requires that the client-server protocol support a null operation or at least an efficient read-only operation, such as a status operation.

Testing HA Data Services

This section provides suggestions about how to test a data service implementation in a highly-available environment. The test cases are suggestions and are not exhaustive. You need access to a test-bed Oracle Solaris Cluster configuration so that the testing work does not affect production machines.

Test that your HA data service behaves correctly in all cases where a resource group is moved between physical hosts. These cases include system crashes and the use of the `clnode` command. Test that client machines continue to get service after these events.

Test the idempotence of the methods. For example, replace each method temporarily with a short shell script that calls the original method two or more times.

Coordinating Dependencies Between Resources

Sometimes one client-server data service makes requests on another client-server data service while fulfilling a request for a client. For example, data service A depends on data service B if, for A to provide its service, B must provide its service. Oracle Solaris Cluster provides for this requirement by permitting resource dependencies to be configured within a resource group. The dependencies affect the order in which Oracle Solaris Cluster starts and stops data services. See the [r_properties\(5\)](#) man page.

If resources of your resource type depend on resources of another type, you need to instruct the cluster administrator to configure the resources and resource groups correctly. As an alternative, provide scripts or tools to correctly configure them.

Decide whether to use explicit resource dependencies, or omit them and poll for the availability of other data services in your HA data service's code. If the dependent and depended-on resource can run on different nodes, configure them in separate resource groups. In this case, polling is required because configuring resource dependencies across groups is not possible.

Some data services store no data directly themselves. Instead, they depend on another back-end data service to store all their data. Such a data service translates all read and update requests into calls on the back-end data service. For example, consider a hypothetical client-server appointment calendar service that keeps all of its data in an SQL database, such as Oracle. The appointment calendar service uses its own client-server network protocol. For example, it might have defined its protocol using an RPC specification language, such as ONC RPC.

In the Oracle Solaris Cluster environment, you can use HA for Oracle to make the back-end Oracle database highly available. Then, you can write simple methods for starting and stopping the appointment calendar daemon. The cluster administrator registers the appointment calendar resource type with Oracle Solaris Cluster software.

If the HA for Oracle resource is to run on a different node than the appointment calendar resource, the cluster administrator configures them into two separate resource groups. The cluster administrator consequently makes the appointment calendar resource dependent on the HA for Oracle resource.

The cluster administrator makes the resources dependent by doing either of the following:

- Configuring the appointment calendar resource in the same resource group as the HA-ORACLE resource.
- Specifying a strong positive affinity between the two resource groups in which each resource is located.

You specify this affinity by using the `RG_affinities` property with the `clresource` command.

The calendar data service daemon, after it has been started, might poll while waiting for the Oracle database to become available. The calendar resource type's `Start` method usually returns success in this case. If the `Start` method blocks indefinitely, however, this method moves its resource group into a busy state. This busy state prevents any further state changes, such as edits, failovers, or switchovers on the resource group. If the calendar resource's `Start` method times out or exits with a nonzero status, its timing out or nonzero exit status might cause the resource group to ping-pong between two or more nodes while the Oracle database remains unavailable.

Legal RGM Names

To see the requirements for legal characters for Resource Group Manager (RGM) names and values, see [“Legal RGM Names”](#) in *Oracle Solaris Cluster 4.3 Data Services Planning and Administration Guide* .

◆◆◆ CHAPTER 3

Resource Management API Reference

This chapter lists and briefly describes the access functions and callback methods that make up the Resource Management API (RMAPI). However, the definitive reference for these functions and methods is the RMAPI man pages.

This chapter covers the following topics:

- [“RMAPI Access Methods” on page 61](#)
- [“RMAPI Callback Methods” on page 67](#)

RMAPI Access Methods

The API provides functions to access resource type, resource, and resource group properties, and other cluster information. These functions are provided both in the form of shell commands and C functions, which enable you to implement control programs as shell scripts or as C programs.

RMAPI Shell Commands

Shell commands are used in shell script implementations of the callback methods for resource types that represent services that are controlled by the cluster's RGM.

You can use these commands to complete the following tasks:

- Access information about resource types, resources, resource groups, and clusters.
- With a monitor, set the `Status` and `Status_msg` properties of a resource.
- Request the restart or relocation of a resource group.

Note - Although this section provides brief descriptions of the shell commands, the 1HA man pages provide the definitive reference for the shell commands. A man page of the same name is associated with each command, unless otherwise noted.

RMAPI Resource Commands

You can access information about a resource or set the `Status` and `Status_msg` properties of a resource with these commands.

`scha_resource_get`

Accesses information about a resource or resource type that is under the control of the RGM. This command provides the same information as the `scha_resource_get()` C function. For details, see the [scha_resource_get\(1HA\)](#) man page.

`scha_resource_setstatus`

Sets the `Status` and `Status_msg` properties of a resource that is under the control of the RGM. This command is used by the resource's monitor to indicate the state of the resource as perceived by the monitor. This command provides the same functionality as the `scha_resource_setstatus()` C function. This command is described in more detail in the [scha_resource_setstatus\(1HA\)](#) man page.

Note - Although `scha_resource_setstatus()` is of particular use to a resource monitor, any program can call it.

Resource Type Command

`scha_resourcetype_get`

Accesses information about a resource type that is registered with the RGM. This command provides the same functionality as the `scha_resourcetype_get()` C function. This command is described in more detail in the [scha_resourcegroup_open\(3HA\)](#) and the [scha_resourcetype_get\(1HA\)](#) man pages.

Resource Group Commands

You can access information about or restart a resource group with these commands.

`scha_resourcegroup_get`

Accesses information about a resource group that is under the control of the RGM. This command provides the same functionality as the `scha_resourcegroup_get()` C function. This command is described in more detail in the [scha_resourcegroup_get\(1HA\)](#) and the [scha_resourcegroup_open\(3HA\)](#) man pages.

`scha_control`

Requests the restart of a resource group that is under the control of the RGM or its relocation to a different node. This command provides the same functionality as the `scha_control()` and `scha_control_zone()` C functions. This command is described in more detail in the [scha_control\(1HA\)](#) man page.

Cluster Command

`scha_cluster_get`

Accesses information about a cluster, such as the cluster name, node name, zone name, IDs, states, and resource groups. This command provides the same information as the `scha_cluster_get()` C function. This command is described in more detail in the [scha_cluster_open\(3HA\)](#) and the [scha_cluster_get\(1HA\)](#) man pages.

C Functions

C functions are used in C program implementations of the callback methods for resource types that represent services that are controlled by the cluster's RGM.

You can use these functions to complete the following tasks:

- Access information about resource types, resources, resource groups, and clusters.
- Set the `Status` and `Status_msg` properties of a resource.
- Request the restart or relocation of a resource group.
- Convert an error code to a related error message.

Note - Although this section provides brief descriptions of the C functions, the 3HA man pages provide the definitive reference for the C functions. A man page of the same name is associated with each function, unless otherwise noted. See the [scha_calls\(3HA\)](#) man page for information about the output arguments and return codes of the C functions.

Resource Functions

These functions access information about a resource that is managed by the RGM or indicate the state of the resource as perceived by the monitor.

`scha_resource_open()`, `scha_resource_get()`, and `scha_resource_close()`

These functions access information about a resource that is managed by the RGM. The `scha_resource_open()` function initializes access to a resource and returns a

handle for `scha_resource_get()`, which accesses the resource information. The `scha_resource_close()` function invalidates the handle and frees memory that is allocated for `scha_resource_get()` return values.

A resource can change, through cluster reconfiguration or administrative action, after `scha_resource_open()` returns the resource's handle. As a result, the information that `scha_resource_get()` obtains through the handle might be inaccurate. In cases of cluster reconfiguration or administrative action on a resource, the RGM returns the `scha_err_seqid` error code to `scha_resource_get()` to indicate that information about the resource might have changed. This error message is nonfatal. The function returns successfully. You can choose to ignore the message and accept the returned information. Alternatively, you can close the current handle and open a new handle to access information about the resource.

One man page describes these three functions. For information about these three functions, see the [scha_resource_open\(3HA\)](#) man page.

`scha_resource_setstatus()`

Sets the `Status` and `Status_msg` properties of a resource that is under the control of the RGM. The resource's monitor uses this function to indicate the resource's state.

Note - Although `scha_resource_setstatus()` is of particular use to a resource monitor, any program can call it.

`scha_resource_setstatus_zone()`

Like the `scha_resource_setstatus()` function, sets the `Status` and `Status_msg` properties of a resource that is under the control of the RGM. The resource's monitor uses this function to indicate the resource's state. However, this function also specifies the name of the zone in which the method is configured to run.

Note - Although `scha_resource_setstatus_zone()` is of particular use to a resource monitor, any program can call it.

Resource Type Functions

These functions access information about a resource type that is registered with the RGM.

`scha_resourcetype_open()`, `scha_resourcetype_get()`, and `scha_resourcetype_close()`

The `scha_resourcetype_open()` function initializes access to a resource and returns a handle for `scha_resourcetype_get()`, which accesses the resource type information. The `scha_resourcetype_close()` function invalidates the handle and frees memory that is allocated for `scha_resourcetype_get()` return values.

A resource type can change, through cluster reconfiguration or administrative action, after `scha_resourcetype_open()` returns the resource type's handle. As a result, the information that `scha_resourcetype_get()` obtains through the handle might be inaccurate. In cases of cluster reconfiguration or administrative action on a resource type, the RGM returns the `scha_err_seqid` error code to `scha_resourcetype_get()` to indicate that information about the resource type might have changed. This error message is nonfatal. The function returns successfully. You can choose to ignore the message and accept the returned information. Alternatively, you can close the current handle and open a new handle to access information about the resource type.

One man page describes these three functions. For more information about these three functions, see the [scha_resourcetype_open\(3HA\)](#) man page.

Resource Group Functions

You can access information about a resource group or restart a resource group with these functions.

`scha_resourcegroup_open()`, `scha_resourcegroup_get()`, and `scha_resourcegroup_close()`

These functions access information about a resource group that is managed by the RGM. The `scha_resourcegroup_open()` function initializes access to a resource group and returns a handle for `scha_resourcegroup_get()`, which accesses the resource group information. The `scha_resourcegroup_close()` function invalidates the handle and frees memory that is allocated for `scha_resourcegroup_get()` return values.

A resource group can change, through cluster reconfiguration or administrative action, after `scha_resourcegroup_open()` returns the resource group's handle. As a result, the information that `scha_resourcegroup_get()` obtains through the handle might be inaccurate. In cases of cluster reconfiguration or administrative action on a resource group, the RGM returns the `scha_err_seqid` error code to `scha_resourcegroup_get()` to indicate that information about the resource group might have changed. This error message is nonfatal. The function returns successfully. You can choose to ignore the message and accept the returned information. Alternatively, you can close the current handle and open a new handle to access information about the resource group.

One man page describes these three functions. For more information about these three functions, see the [scha_resourcegroup_open\(3HA\)](#) man page.

`scha_control()` and `scha_control_zone()`

Requests the restart of a resource group that is under the control of the RGM or its relocation to a different node. These functions are described in more detail in the [scha_control\(3HA\)](#) man page.

Cluster Functions

These functions access or return information about a cluster.

`scha_cluster_open()`, `scha_cluster_get()`, and `scha_cluster_close()`

These functions access information about a cluster, such as the cluster name, node name, zone name, IDs, states, and resource groups.

A cluster can change, through reconfiguration or administrative action, after `scha_cluster_open()` returns the cluster's handle. As a result, the information that `scha_cluster_get()` obtains through the handle might be inaccurate. In cases of reconfiguration or administrative action on a cluster, the RGM returns the `scha_err_seqid` error code to `scha_cluster_get()` to indicate that information about the cluster might have changed. This error message is nonfatal. The function returns successfully. You can choose to ignore the message and accept the returned information. Alternatively, you can close the current handle and open a new handle to access information about the cluster.

One man page describes these three functions. For more information about these three functions, see the [scha_cluster_open\(3HA\)](#) man page.

`scha_cluster_getlogfacility()`

Returns the number of the system log facility that is being used as the cluster log. Uses the returned value with the `syslog()` Oracle Solaris function to record events and status messages to the cluster log. This function is described in more detail in the [scha_cluster_getlogfacility\(3HA\)](#) man page.

`scha_cluster_getnodename()`

Returns the name of the cluster node on which the function is called. This function is described in more detail in the [scha_cluster_getnodename\(3HA\)](#) man page.

Utility Function

This function converts an error code to an error message.

`scha_strerror()`

Translates an error code that is returned by one of the `scha_` functions to a corresponding error message. Use this function with the `logger` command to log messages in the Oracle Solaris system log (`syslog`). This function is described in more detail in the [scha_strerror\(3HA\)](#) man page.

RMAPI Callback Methods

Callback methods are the key elements that are provided by the API for implementing a resource type. Callback methods enable the RGM to control resources in the cluster in the event of a change in cluster membership, such as the failure of a node.

Note - The callback methods are executed by the RGM with root role or the greatest RBAC role permissions because the client programs control HA services in the cluster system. Install and administer these methods with restrictive file ownership and permissions. Specifically, give these methods a privileged owner, such as bin or root, and do not make them writable.

This section describes callback method arguments and exit codes.

Callback methods in the following categories are described:

- Control and initialization methods
- Administrative support methods
- Net-relative methods
- Monitor control methods

Note - This section provides brief descriptions of the callback methods, including the point at which the method is run and the expected effect on the resource. However, the [rt_callbacks\(1HA\)](#) man page is the definitive reference for the callback methods.

Arguments That You Can Provide to Callback Methods

The RGM runs callback methods, as follows:

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

The method is the path name of the program that is registered as the Start, Stop, or other callback. The callback methods of a resource type are declared in its registration file.

All callback method arguments are passed as flagged values, as follows:

- -R indicates the name of the resource instance
- -T indicates the type of the resource
- -G indicates the group into which the resource is configured

Use the arguments with access functions to retrieve information about the resource.

The `Validate` method is called with additional arguments that include the property values of the resource and resource group on which it is called.

The [scha_calls\(3HA\)](#) man page contains more information.

Callback Method Exit Codes

All callback methods have the same exit codes. These exit codes are defined to specify the effect of the method invocation on the resource state. The [scha_calls\(3HA\)](#) man page describes these exit codes in more detail.

The two major categories of exit codes are as follows:

- `0` – The method succeeded
- Any nonzero value – The method failed

The RGM also handles abnormal failures of callback method execution, such as timeouts and core dumps.

Method implementations must output failure information by using `syslog()` on each node. Output written to `stdout` or `stderr` is not guaranteed to be delivered to the user, although it is currently displayed on the console of the local node.

Control and Initialization Callback Methods

The primary control and initialization callback methods start and stop a resource. Other methods execute initialization and termination code on a resource.

Start

The RGM runs this method on a cluster node when the resource group that contains the resource is brought online on that node. This method activates the resource on that node.

A `Start` method should not exit until the resource that it activates has been started and is available on the local node. Therefore, before exiting, the `Start` method should poll the resource to determine that it has started. In addition, you should set a sufficiently long timeout value for this method. For example, particular resources, such as database daemons, take more time to start, and thus require that the method have a longer timeout value.

The way in which the RGM responds to failure of the `Start` method depends on the setting of the `Failover_mode` property.

The `Start_timeout` property in the resource type registration (RTR) file sets the timeout value for a resource's Start method.

Stop

The RGM runs this required method on a cluster node when the resource group that contains the resource is brought offline on that node. This method deactivates the resource if it is active.

A Stop method should not exit until the resource that it controls has completely stopped all its activity on the local node and has closed all file descriptors. Otherwise, because the RGM assumes the resource has stopped when, in fact, it is still active, data corruption can result. The safest way to avoid data corruption is to terminate all processes on the local node that is related to the resource.

Before exiting, the Stop method should poll the resource to determine that it has stopped. In addition, you should set a sufficiently long timeout value for this method. For example, particular resources, such as database daemons, take more time to stop, and thus require that the method have a longer timeout value.

If an RGM method callback times out, the method's process group is killed by a SIGTERM signal. If the process group does not exit within 10 seconds, a SIGKILL is executed.

Although method core dumps are not generated when a method times out, system core dumps are generated when a node is rebooted by the RGM due to a resource's `Failover_mode` property being set to HARD and the resource method being stuck in the kernel and failing to exit after a SIGKILL.

Note - Avoid writing data service methods that create a new process group. If your data service method must create a new process group, write a signal handler for the SIGTERM and SIGABRT signals. Also, ensure that your signal handler forwards the SIGTERM or SIGABRT signal to the child process group or groups before the signal handler terminates the process. Writing a signal handler for these signals increases the likelihood that all processes that are spawned by your method are correctly terminated.

The way in which the RGM responds to failure of the Stop method depends on the setting of the `Failover_mode` property. See [“Resource Properties” on page 31](#).

The `Stop_timeout` property in the RTR file sets the timeout value for a resource's Stop method.

Init

The RGM runs this optional method to perform a one-time initialization of the resource when the resource becomes managed. The RGM runs this method when its resource group is switched from an unmanaged to a managed state or when the resource is created in a resource group that is already managed. The method is called on nodes that are identified by the `Init_nodes` resource property.

Fini

The RGM executes the `Fini` method to clean up after a resource when that resource is no longer managed by the RGM. The `Fini` method usually undoes any initializations that were performed by the `Init` method.

The RGM executes `Fini` on each node on which the resource becomes unmanaged when the following situations arise:

- The resource group that contains the resource is switched to an unmanaged state. In this case, the RGM executes the `Fini` method on all nodes in the node list.
- The resource is deleted from a managed resource group. In this case, the RGM executes the `Fini` method on all nodes in the node list.
- A node is deleted from the node list of the resource group that contains the resource. In this case, the RGM executes the `Fini` method on only the deleted node.

A “node list” is either the resource group's `Nodelist` or the resource type's `Installed_nodes` list. Whether “node list” refers to the resource group's `Nodelist` or the resource type's `Installed_nodes` list depends on the setting of the resource type's `Init_nodes` property. The `Init_nodes` property can be set to `RG_PRIMARYES` or `RT_INSTALLED_NODES`. For most resource types, `Init_nodes` is set to `RG_PRIMARYES`, the default. In this case, both the `Init` and `Fini` methods are executed on the nodes that are specified in the resource group's `Nodelist`.

The type of initialization that the `Init` method performs defines the type of cleanup that the `Fini` method that you implement needs to perform, as follows:

- Cleanup of node-specific configuration.
- Cleanup of cluster-wide configuration.

The `Fini` method that you implement needs to determine whether to perform only cleanup of node-specific configuration or cleanup of both node-specific and cluster-wide configuration.

When a resource becomes unmanaged on only a particular node, the `Fini` method can clean up local, node-specific configuration. However, the `Fini` method must not clean up global, cluster-wide configuration, because the resource remains managed on other nodes. If the resource becomes unmanaged cluster-wide, the `Fini` method can perform cleanup of both node-specific and global configuration. Your `Fini` method code can distinguish these two cases by determining whether the resource group's node list contains the local node on which your `Fini` method is executing.

If the local node appears in the resource group's node list, the resource is being deleted or is moving to an unmanaged state. The resource is no longer active on any node. In this case, your `Fini` method needs to clean up any node-specific configuration on the local node as well as cluster-wide configuration.

If the local node does not appear in the resource group's node list, your `Fini` method can clean up node-specific configuration on the local node. However, your `Fini` method must not clean up cluster-wide configuration. In this case, the resource remains active on other nodes.

You must also code the `Fini` method so that it is idempotent. In other words, even if the `Fini` method has cleaned up a resource during a previous execution, subsequent calls to the `Fini` method exit successfully.

Boot

The RGM runs this optional method, which is similar to `Init`, to initialize the resource on nodes that join the cluster after the resource group that contains the resource has already been put under the management of the RGM. The RGM runs this method on nodes that are identified by the `Init_nodes` resource property. The `Boot` method is called when the node joins or rejoins the cluster as a result of being booted or rebooted.

If the `Global_zone` resource type property equals `TRUE`, methods execute in the global zone even if the resource group that contains the resource is configured to run in a zone-cluster node.

Note - Failure of the `Init`, `Fini`, or `Boot` methods causes an error message to be written to the system log. However, management of the resource by the RGM is not otherwise affected.

Administrative Support Methods

Administrative actions on resources include setting and changing resource properties. The `Validate` and `Update` callback methods enable a resource type implementation to carry out these administrative actions.

Validate

The RGM calls this optional method when a resource is created and when the cluster administrator updates the properties of the resource or its containing resource group. This method is called on the set of cluster nodes that are identified by the `Init_nodes` property of the resource's type. The `Validate` method is called before the creation or the update is applied. A failure exit code from the method on any node causes the creation or the update to be canceled.

`Validate` is called only when resource or resource group properties are changed by the cluster administrator. This method is not called when the RGM sets properties, nor when a monitor sets the `Status` and `Status_msg` resource properties.

Update

The RGM runs this optional method to notify a running resource that properties have been changed. The RGM runs Update after an administrative action succeeds in setting properties of a resource or its group. This method is called on nodes where the resource is online. The method uses the API access functions to read property values that might affect an active resource and to adjust the running resource accordingly.

Note - Failure of the Update method causes an error message to be written to the system log. However, management of the resource by the RGM is not otherwise affected.

Net-Relative Callback Methods

Services that use network address resources might require that start or stop steps be carried out in a particular order relative to the network address configuration. The following optional callback methods, `Prenet_start` and `Postnet_stop`, enable a resource type implementation to carry out special startup and shutdown actions before and after a related network address is configured or unconfigured.

`Prenet_start`

This optional method is called to carry out special startup actions before network addresses in the same resource group are configured.

`Postnet_stop`

This optional method is called to carry out special shutdown actions after network addresses in the same resource group are configured down.

Monitor Control Callback Methods

A resource type implementation optionally can include a program to monitor the performance of a resource, report on its status, or take action when a resource fails. The `Monitor_start`, `Monitor_stop`, and `Monitor_check` methods support the implementation of a resource monitor in a resource type implementation.

`Monitor_start`

This optional method is called to start a monitor for the resource after the resource is started.

`Monitor_stop`

This optional method is called to stop a resource's monitor before the resource is stopped.

Monitor_check

This optional method is called to assess the reliability of a node before a resource group is relocated to that node. You must implement the `Monitor_check` method so that it does not conflict with the concurrent running of another method.

Modifying a Resource Type

This chapter discusses the issues that you need to understand to modify a resource type. Information about the means by which you enable a cluster administrator to upgrade a resource is also included.

This chapter covers the following topics:

- [“Overview of Modifying a Resource Type” on page 75](#)
- [“Setting Up the Contents of the Resource Type Registration File” on page 76](#)
- [“What Happens When a Cluster Administrator Upgrades” on page 79](#)
- [“Determining Installation Requirements and Packaging” on page 80](#)
- [“Documentation to Provide for a Modified Resource Type” on page 82](#)

Overview of Modifying a Resource Type

Cluster administrators must be able to carry out the following tasks:

- Install and register a new version of an existing resource type
- Allow the registration of multiple versions of a given resource type
- Upgrade an existing resource to a new version of the resource type without having to delete and recreate the resource

A resource type that you intend to upgrade is called an *upgrade-aware* resource type.

Elements of an existing resource type that you might change are as follows:

- Attributes of resource type properties
- The set of declared resource properties, including standard and extension properties
- Attributes of resource properties, such as `default`, `min`, `max`, `arraymin`, `arraymax`, or `tunability`
- The set of declared methods

- The implementation of methods or monitors

Note - You do not necessarily have to modify a resource type when you modify application code.

You need to understand the requirements for providing the tools that will enable a cluster administrator to upgrade a resource type. This chapter tells you what you need to know to set up these tools.

Setting Up the Contents of the Resource Type Registration File

This section describes how to set up a resource type registration file.

This section covers the following topics:

- [“Resource Type Name” on page 76](#)
- [“Specifying the # \$\\$\$ upgrade and # \$\\$\$ upgrade_from Directives” on page 77](#)
- [“Changing the RT_version in an RTR File” on page 78](#)

Resource Type Name

The three components of a resource type name are properties that are specified in the RTR file as *vendor-id*, *resource-type*, and *rt-version*. The `clresourcetype(1CL)` command inserts the period and the colon delimiters to create the name of the resource type:

```
vendor-id.resource-type:rt-version
```

The *vendor-id* prefix serves to distinguish between two registration files of the same name that different companies provide. To ensure that the *vendor-id* is unique, use the stock symbol of the company when creating the resource type. The *rt-version* distinguishes between multiple registered versions (upgrades) of the same base resource type.

You can obtain the fully qualified resource type name by typing the following command:

```
# scha_resource_get -O Type -R resource-name -G resource-group-name
```

The format of resource type names is described in [“Format of Resource Type Names” in Oracle Solaris Cluster 4.3 Data Services Planning and Administration Guide](#) .

Specifying the `#$upgrade` and `#$upgrade_from` Directives

Existing resources can be upgraded at run time to a new upgrade-aware version of their resource type. However, if the new resource type version is non-upgrade-aware, you would have to delete and recreate any existing resources of that type to move them to the new version. Therefore, new resource types should always include the `#$upgrade` directive.

To ensure that the resource type that you are modifying is upgrade-aware, include the `#$upgrade` directive in the resource type's RTR file. After the `#$upgrade` directive, add zero or more `#$upgrade_from` directives for each earlier version of the resource type that you want to support.

The `#$upgrade` and `#$upgrade_from` directives must appear between the resource type property declarations and the resource declarations sections in the RTR file. See the [rt_reg\(4\)](#) man page.

EXAMPLE 1 `#$upgrade_from` Directive in an RTR File

```
#$upgrade
#$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
```

The format of the `#$upgrade_from` directive is as follows:

```
#$upgrade_from version tunability
```

version

The `RT_version`. If any resource type does not have a version, or for versions other than what you defined previously in the RTR file, specify the empty string (“”).

tunability

The conditions under which, or when, the cluster administrator can upgrade the specified `RT_version`.

Use the following tunability values in the `#$upgrade_from` directives:

ANYTIME

Use when there are no restrictions on when the cluster administrator can upgrade the resource. The resource can be completely online during the upgrade.

WHEN_UNMONITORED

Use when the new resource type version's methods are as follows:

- The Update, Stop, Monitor_check, and Postnet_stop methods are compatible with the older resource type version's starting methods (Prenet_stop and Start)
- The Fini method is compatible with the Init method of older versions

The cluster administrator must only stop the resource monitor program before upgrading.

WHEN_OFFLINE

Use when the new resource type version's Update, Stop, Monitor_check, or Postnet_stop method is as follows:

- Compatible with the Init method of an older version
- Incompatible with an older resource type version's starting methods (Prenet_stop and Start)

The cluster administrator must take the resource offline before upgrading.

WHEN_DISABLED

Similar to WHEN_OFFLINE. However, the cluster administrator must disable the resource before upgrading.

WHEN_UNMANAGED

Use when the new resource type version's Fini method is incompatible with the Init method of an older version. The cluster administrator must switch the existing resource group to the unmanaged state before upgrading.

If a version of the resource type does not appear in the list of #supgrade_from directives, the RGM imposes the tunability of WHEN_UNMANAGED to that version by default.

AT_CREATION

Use to prevent existing resources from being upgraded to the new version of the resource type. The cluster administrator must delete and recreate a resource.

Changing the RT_version in an RTR File

You only need to change the RT_version property in an RTR file whenever the contents of the RTR file change. Choose a value for this property that clearly indicates that this version of the resource type is the latest version.

Do *not* include the following characters in the `RT_version` string in the RTR file or registration of the resource type fails:

- Space
- Tab
- Slash (/)
- Backslash (\)
- Asterisk (*)
- Question mark (?)
- Comma (,)
- Semicolon (;)
- Left square bracket ([)
- Right square bracket (])

The `RT_version` property is required.

What Happens When a Cluster Administrator Upgrades

Here is what the cluster administrator must do or what happens when he or she upgrades a resource type:

- If the existing resource property attributes do not satisfy the validation conditions of the new version of the resource type, the cluster administrator must provide valid values.
The cluster administrator must provide valid values under the following conditions:
 - When the new version of the resource type does not have a default value and uses a property that is not declared in the earlier version.
 - When the existing resource uses a property whose value is undeclared or invalid in the new version. Declared properties that are undeclared in a new version of a resource type are deleted from the resource.
- Any attempt to upgrade from an unsupported version of a resource type fails.
- After an upgrade, resources inherit the resource property attributes for all properties from the new version of the resource type.
- If you change the default value of a resource type in the RTR file, the new default value is inherited by existing resources. The new default value is inherited even if the property is declared tunable only `AT_CREATION` or `WHEN_DISABLED`. A property of the same type that the cluster administrator creates also inherits this default value. However, if the cluster administrator specifies a new default value for the property, the new default value overrides the default value that is specified in the RTR file.

Note - The cluster administrator can overcome this limitation by specifying values for the properties and thus overriding the defaults.

Determining Installation Requirements and Packaging

Keep the following two requirements in mind when determining installation requirements and packaging for resource type packages:

- When a new resource type is registered, its RTR file must be accessible on disk.
- When a resource of the new type is created, all declared method path names and the monitor program for the new type must be on disk and be executable. The old method and monitor programs must remain in place as long as the resource is in use.

To determine the correct packaging to use, consider the following questions:

- Does the RTR file change?
- Does the default value or tunability of a property change?
- Does the min or max value of a property change?
- Does the upgrade add or delete properties?
- Does the monitor code change?
- Does the method code change?
- Are the new methods, the monitor code, or both compatible with the previous versions?

The answers to these questions will help you determine the correct packaging to use for your new resource type.

Before You Change the RTR File

You do not necessarily need to create new method or monitor code when you modify a resource type. For example, you might only change the default value or tunability of a resource property. In this instance, because you do not change the method code, you only require a new valid path name to a readable RTR file.

If you do not need to reregister the old resource type, the new RTR file can overwrite the previous version. Otherwise, place the new RTR file in a new path.

If the upgrade changes the default value or tunability of a property, use the `Validate` method for the new version of the resource type to verify that the existing property attributes are valid

for the new resource type. If they are not, the cluster administrator can change the properties of an existing resource to the correct values. If the upgrade changes the `min`, `max`, or `type` attributes of a property, the `cl_resourcetype(1CL)` command automatically validates these constraints when the cluster administrator upgrades the resource type.

If the upgrade adds a new property or deletes an old property, you probably need to change callback methods or monitor code.

Changing Monitor Code

If you change only the monitor code for a resource type, the package installation can overwrite the monitor binaries.

Changing Method Code

If you change only the method code in a resource type, you must determine whether the new method code is compatible with the old method code. The answer to this question determines whether the new method code must be stored in a new path or whether the old methods can be overwritten.

If you can apply the new `Stop`, `Postnet_stop`, and `Fini` methods (if declared) to resources that were initialized or started by the old versions of the `Start`, `Prenet_stop`, or `Init` methods, the old methods can be overwritten with the new methods.

If applying a new default value to a property causes a method such as `Stop`, `Postnet_stop`, or `Fini` to fail, the cluster administrator must accordingly restrict the state of the resource when the resource type is upgraded.

You enable the cluster administrator to restrict the state of the resource when it is upgraded by limiting the tunability of the `Type_version` property.

One approach to packaging is to include all earlier versions of a resource type that are still supported in the package. This approach permits the new version of a package to replace the old version of the package, without overwriting or deleting the old paths to the methods. You must decide the number of previous versions to support.

Determining the Packaging Scheme to Use

The following table summarizes the packaging schemes to use for your new resource types.

TABLE 1 Determining the Packaging Scheme to Use

Type of Change	Tunability Value	Packaging Scheme
Make property changes in only the RTR file.	ANYTIME	Deliver only new RTR file.
Update the methods.	ANYTIME	Place the updated methods in a different path than the old methods.
Install the new monitor program.	WHEN_UNMONITORED	Overwrite only the previous version of the monitor.
Update the methods. The new Update and Stop methods are incompatible with the old Start methods.	WHEN_OFFLINE	Place the updated methods in a different path than the old methods.
Update the methods and add new properties to the RTR file. The new methods require new properties. The goal is to allow the containing resource group to remain online but prevent the resource from coming online if the resource group moves from the offline state to the online state on a node.	WHEN_DISABLED	Overwrite the previous versions of the methods.
Update the methods and add new properties to the RTR file. New methods do not require new properties.	ANYTIME	Overwrite the previous versions of the methods.
Update the methods. The new <code>Fin</code> method is incompatible with the old <code>Init</code> method.	WHEN_UNMANAGED	Place the updated methods in a different path than the old methods.
Update the methods. No changes are made to the RTR file.	Not applicable. No changes are made to the RTR file.	Overwrite the previous versions of the methods. Because you made no changes to the RTR file, the resource does not need to be registered or upgraded.

Documentation to Provide for a Modified Resource Type

Instructions that tell the cluster administrator how to upgrade a resource type are provided in [“Upgrading a Resource Type” in Oracle Solaris Cluster 4.3 Data Services Planning and Administration Guide](#). To enable the cluster administrator to upgrade a resource type that you modify, supplement these instructions with additional information, as described in this section.

Generally, when you create a new resource type, you need to provide documentation that does the following:

- Describes the properties that you add, change, or delete
- Describes how to make the properties conform to the new requirements
- States the tunability constraints on resources

- Calls out any new default property attributes
- Informs the cluster administrator that he or she can set existing resource properties to their correct values if necessary

Information About What to Do Before Installing an Upgrade

Explain to the cluster administrator what he or she must do before installing the upgrade package on a node, as follows:

- If the upgrade package overwrites existing methods or monitors, tell the cluster administrator to unmonitor all resources.
- If the upgrade package updates only the RTR file, leaving the method and monitor code unchanged, it is not necessary to unmonitor resources.

Information About When to Upgrade Resources

Explain to the cluster administrator when he or she can upgrade resources to a new version of the resource type.

The conditions under which the cluster administrator can upgrade the resource type depend on the tunability of the `#$upgrade_from` directive for each version of the resource in the RTR file, as follows:

- Any time (ANYTIME)
- Only when the resource is unmonitored (WHEN_UNMONITORED)
- Only when the resource is offline (WHEN_OFFLINE)
- Only when the resource is disabled (WHEN_DISABLED)
- Only when the resource group is unmanaged (WHEN_UNMANAGED)

EXAMPLE 2 How `#$upgrade_from` Defines When a Cluster Administrator Can Upgrade

This example shows how the tunability of the `#$upgrade_from` directive affects the conditions under which the cluster administrator can upgrade a resource to a new version of a resource type.

```
#$upgrade_from "1.1" WHEN_OFFLINE
#$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
    
```

Version	When the Cluster Administrator Can Upgrade a Resource
1.1, 1.2, or 1.3	Only when the resource is offline
2.0	Only when the resource is unmonitored
2.1	Any time
All other versions	Only when the resource group is unmanaged

Information About Changes to Resource Properties

Describe any changes that you have made to the resource type that require the cluster administrator to modify properties of existing resources when he or she upgrades.

Possible changes that you can make include the following:

- Default settings of existing resource type properties that you have changed
- New extension properties of the resource type that you have introduced
- Existing properties of the resource type that you have withdrawn
- Changes to the set of standard properties that you have declared for the resource type
- Attributes of resource properties such as `min`, `max`, `arraymin`, `arraymax`, `default`, and `tunability` that you have changed
- Changes to the set of methods that you have declared
- Implementation of methods or the fault monitor that you have changed

Sample Data Service

This chapter describes a sample Oracle Solaris Cluster data service, HA for DNS, for the `in.` named application. The `in.named` daemon is the Oracle Solaris implementation of the Domain Name Service (DNS). The sample data service demonstrates how to make an application highly available, using the Resource Management API.

The Resource Management API supports a shell script interface and a C program interface. The sample application in this chapter is written using the shell script interface.

This chapter covers the following topics:

- [“Overview of the Sample Data Service” on page 85](#)
- [“Defining the Resource Type Registration File” on page 86](#)
- [“Providing Common Functionality to All Methods” on page 92](#)
- [“Controlling the Data Service” on page 97](#)
- [“Defining a Fault Monitor” on page 103](#)
- [“Handling Property Updates” on page 113](#)

Overview of the Sample Data Service

The sample data service starts, stops, restarts, and switches the DNS application among the nodes of the cluster in response to cluster events, such as administrative action, application failure, or node failure.

Application restart is managed by the Process Monitor Facility (PMF). If the number of applications that die exceeds the failure count within the failure time window, the fault monitor fails over the resource group that contains the application resource to another node.

The sample data service provides fault monitoring in the form of a `PROBE` method that uses the `nslookup` command to ensure that the application is healthy. If the probe detects a hung DNS service, the probe tries to correct the situation by restarting the DNS application locally. If

restarting the DNS application locally does not improve the situation and the probe repeatedly detects problems with the service, the probe attempts to fail over the service to another node in the cluster.

Specifically, the sample data service includes the following elements:

- A resource type registration file that defines the static properties of the data service.
- A `Start` callback method that is run by the RGM to start the `in.named` daemon when the resource group that contains the HA-DNS data service is brought online.
- A `Stop` callback method that is run by the RGM to stop the `in.named` daemon when the resource group that contains HA-DNS goes offline.
- A fault monitor to check the availability of the service by verifying that the DNS server is running. The fault monitor is implemented by a user-defined `PROBE` method, and is started and stopped by the `Monitor_start` and `Monitor_stop` callback methods.
- A `Validate` callback method that is run by the RGM to validate that the configuration directory for the service is accessible.
- An `Update` callback method that is run by the RGM to restart the fault monitor when the cluster administrator changes the value of a resource property.

Defining the Resource Type Registration File

The resource type registration (RTR) file in this example defines the static configuration of the DNS resource type. Resources of this type inherit the properties that are defined in the RTR file.

The information in the RTR file is read by the Resource Group Manager (RGM) when the cluster administrator registers the HA-DNS data service. By convention, you place the RTR file in the `/opt/cluster/lib/rgm/rtreg/` directory. Note that the package installer places the RTR file that Agent Builder creates in this directory as well.

Overview of the RTR File

The RTR file follows a well-defined format. Resource type properties are defined first in the file, system-defined resource properties are defined next, and extension properties are defined last. See the `rt_reg(4)` man page and [“Setting Resource and Resource Type Properties” on page 32](#) for more information.

The following sections describe the specific properties in the sample RTR file. These sections provide listings of different parts of the file. For a complete listing of the contents of the sample RTR file, see [“Resource Type Registration File Listing” on page 231](#).

Resource Type Properties in the Sample RTR File

The sample RTR file begins with comments followed by resource type properties that define the HA-DNS configuration, as shown in the following listing. The example assumes that previous versions of the resource type exist, so `#$upgrade` directives are included. See [“Specifying the # \\$upgrade and # \\$upgrade_from Directives” on page 77](#) for more information.

Note - Property names for resource groups, resources, and resource types are *not* case sensitive. You can use any combination of uppercase and lowercase letters when you specify property names.

```
#
# Copyright (c) 1998, 2011, Oracle and/or its affiliates.
# All rights reserved.
#
# Registration information for Domain Name Service (DNS)
#

#pragma ident "@(#)ORCL.sample 1.3 00/05/24"

Resource_type = "sample";
Vendor_id = ORCL;
RT_description = "Domain Name Service on Oracle Solaris Cluster";

RT_version = "3";
API_version = 2;
Failover = TRUE;

RT_basedir=/opt/ORCLsample/bin;
Pkglst = ORCLsample;

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;

#$upgrade
#$upgrade_from "1" when_disabled
#$upgrade_from "2" anytime
```

Tip - You must declare the `Resource_type` property as the first entry in the RTR file. Otherwise, registration of the resource type fails.

The following information describes these properties:

- You can specify the resource type name by the `Resource_type` property alone (`sample`) or by using the `vendor-id` as a prefix, followed by a period (`.`), followed by the resource type property (`ORCL.sample`).
If you specify `vendor-id`, use the stock exchange symbol of the company that is defining the resource type. The resource type name must be unique in the cluster.
- The `RT_version` property identifies the version of the sample data service as specified by the vendor.
- The `API_version` property identifies the Oracle Solaris Cluster version. For example, `API_version = 11` indicates that the data service can be registered on any version of Oracle Solaris Cluster starting with 3.3, assuming that the application is compatible with that version of Oracle Solaris Cluster software. However, `API_version = 11` also indicates that the data service cannot be registered on any version of Oracle Solaris Cluster that was released before 3.3. This property is described in more detail in the [rt_properties\(5\)](#) man page.
- `Failover = TRUE` indicates that the data service cannot run in a resource group that can be online on multiple nodes at the same time.
- `RT_basedir` points to `/opt/ORCLsample/bin` as the directory path to complete relative paths, such as callback method paths.
- `Start`, `Stop`, and `Validate` provide the paths to the respective callback method programs that are run by the RGM. These paths are relative to the directory that is specified by `RT_basedir`.
- `Pkglist` identifies `ORCLsample` as the package that contains the sample data service installation.

Resource type properties that are not specified in this RTR file, such as `Single_instance`, `Init_nodes`, and `Installed_nodes`, are set to their default values. “[Resource Type Properties](#)” on page 31 contains a complete list of the resource type properties, including their default values.

The cluster administrator cannot change the values for resource type properties in the RTR file.

Resource Properties in the Sample RTR File

By convention, you declare resource properties after the resource type properties in the RTR file. Resource properties include system-defined properties that are provided by the Oracle

Solaris Cluster software and extension properties that you define. For either type, you can specify a number of property attributes that are supplied by the Oracle Solaris Cluster software, such as minimum, maximum, and default values.

System-Defined Properties in the RTR File

The following listing shows the system-defined properties in a sample RTR file.

```
# A list of bracketed resource property declarations follows the
# resource type declarations. The property-name declaration must be
# the first attribute after the open curly bracket of each entry.

# The <method>_timeout properties set the value in seconds after which
# the RGM concludes invocation of the method has failed.

# The MIN value for all method timeouts is set to 60 seconds. This
# prevents administrators from setting shorter timeouts, which do not
# improve switchover/failover performance, and can lead to undesired
# RGM actions (false failovers, node reboot, or moving the resource group
# to ERROR_STOP_FAILED state, requiring operator intervention). Setting
# too-short method timeouts leads to a *decrease* in overall availability
# of the data service.
{
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;
}

# The number of retries to be done within a certain period before concluding
# that the application cannot be successfully started on this node.
{
PROPERTY = Retry_count;
MIN=0;
MAX=10;
DEFAULT=2;
TUNABLE = ANYTIME;
}

# Set Retry_interval as a multiple of 60 since it is converted from seconds
# to minutes, rounding up. For example, a value of 50 (seconds)
# is converted to 1 minute. Use this property to time the number of
# retries (Retry_count).
{
PROPERTY = Retry_interval;
MIN=60;
MAX=3600;
DEFAULT=300;
TUNABLE = ANYTIME;
}

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

```

Although the Oracle Solaris Cluster software provides the system-defined properties, you can set different default values by using resource property attributes. See [“Resource Property Attributes” on page 31](#) for a complete list of attributes that are available to you to apply to resource properties.

Note the following points about the system-defined resource properties in the sample RTR file:

- Oracle Solaris Cluster provides a minimum value (1 second) and a default value (3600 seconds, or one hour) for all timeouts. The sample RTR file changes the minimum timeout to 60 seconds and the default value to 300 seconds. A cluster administrator can accept this default value or change the value of the timeout to another value, 60 or greater. Oracle Solaris Cluster has no maximum allowed value.
- The TUNABLE attribute for the properties `Thorough_probe_interval`, `Retry_count`, and `Retry_interval`, are set to `ANYTIME`. These settings indicate that the cluster administrator can change the value of these properties, even when the data service is running. These properties are used by the fault monitor implemented by the sample data service. The sample data service implements an `Update` method to stop and restart the fault monitor when these or other resource properties are changed by administrative action. See [“How the Update Method Works” on page 118](#).
- Resource properties are classified as follows:
 - **Required.** The cluster administrator must specify a value when creating a resource.
 - **Optional.** If the cluster administrator does not specify a value, the system supplies a default value.
 - **Conditional.** The RGM creates the property only if it is declared in the RTR file.

The fault monitor of the sample data service makes use of the `Thorough_probe_interval`, `Retry_count`, `Retry_interval`, and `Network_resources_used` conditional properties, so you need to declare them in the RTR file. See the [`r_properties\(5\)` man page](#) or [“Resource Properties” on page 31](#) for information about how properties are classified.

Extension Properties in the RTR File

At the end of the sample RTR file are extension properties, as shown in this listing.

```
# Extension Properties

# The cluster administrator must set the value of this property to point to the
# directory that contains the configuration files used by the application.
# For this application, DNS, specify the path of the DNS configuration file on
# PXFS (typically named.conf).
{
PROPERTY = Confdir;
EXTENSION;
STRING;
TUNABLE = AT_CREATION;
DESCRIPTION = "The Configuration Directory Path";
}

# Time out value in seconds before declaring the probe as failed.
```

```
{  
PROPERTY = Probe_timeout;  
EXTENSION;  
INT;  
DEFAULT = 120;  
TUNABLE = ANYTIME;  
DESCRIPTION = "Time out value for the probe (seconds)";  
}
```

The sample RTR file defines two extension properties, `Confdir` and `Probe_timeout`. The `Confdir` property specifies the path to the DNS configuration directory. This directory contains the `in.named` file, which DNS requires to operate successfully. The sample data service's `Start` and `Validate` methods use this property to verify that the configuration directory and the `in.named` file are accessible before starting DNS.

When the data service is configured, the `Validate` method verifies that the new directory is accessible.

The sample data service's `PROBE` method is not an Oracle Solaris Cluster callback method but a user-defined method. Therefore, Oracle Solaris Cluster does not provide a `Probe_timeout` property for it. You need to define an extension property in the RTR file to enable a cluster administrator to configure a `Probe_timeout` value.

Providing Common Functionality to All Methods

This section describes the following functionality that is used in all callback methods of the sample data service:

- [“Identifying the Command Interpreter and Exporting the Path” on page 92](#)
- [“Declaring the `PMF_TAG` and `SYSLOG_TAG` Variables” on page 93](#)
- [“Parsing the Function Arguments” on page 94](#)
- [“Generating Error Messages” on page 95](#)
- [“Obtaining Property Information” on page 96](#)

Identifying the Command Interpreter and Exporting the Path

The first line of a shell script must identify the command interpreter. Each method script in the sample data service identifies the command interpreter, as follows:

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

All method scripts in the sample application export the path to the Oracle Solaris Cluster binaries and libraries rather than relying on the user's PATH settings.

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

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

Declaring the PMF_TAG and SYSLOG_TAG Variables

All the method scripts, except `Validate`, use the `pmfadm` command to start or to stop either the data service or the monitor, and to pass the name of the resource. Each script defines a variable, `PMF_TAG`, that can be passed to the `pmfadm` command to identify either the data service or the monitor.

Likewise, each method script uses the `logger` command to log messages in the system log. Each script defines a variable, `SYSLOG_TAG`, that can be passed to `logger` with the `-t` option to identify the resource type, resource name, and resource group of the resource for which the message is being logged.

All methods define `SYSLOG_TAG` in the same way, as shown in the following sample code. The `dns_probe`, `dns_svc_start`, `dns_svc_stop`, and `dns_monitor_check` methods define `PMF_TAG` as follows (the use of `pmfadm` and `logger` is from the `dns_svc_stop` method).

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

PMF_TAG=$RESOURCE_NAME.named

SYSLOG_TAG=$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME

# Send a SIGTERM signal to the data service and wait for 80% of the
# total timeout value.
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"
```

The `dns_monitor_start`, `dns_monitor_stop`, and `dns_update` methods define `PMF_TAG` as follows (the use of `pmfadm` is from the `dns_monitor_stop` method):

```
#####
```

```
# MAIN
#####

PMF_TAG=$RESOURCE_NAME.monitor
SYSLOG_TAG=$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME
...
# See if the monitor is running, and if so, kill it.
if pmfadm -q $PMF_TAG.monitor; then
pmfadm -s $PMF_TAG.monitor KILL
```

Parsing the Function Arguments

The RGM runs all of the callback methods, except `Validate`, as follows:

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

The method name is the path name of the program that implements the callback method. A data service specifies the path name for each method in the RTR file. These path names are relative to the directory that is specified by the `RT_basedir` property, also in the RTR file. For example, in the sample data service's RTR file, the base directory and method names are specified as follows:

```
RT_basedir=/opt/ORCLsample/bin;
Start = dns_svc_start;
Stop = dns_svc_stop;
...
```

All callback method arguments are passed as flagged values. The `-R` argument indicates the name of the resource instance. The `-T` argument indicates the type of the resource. The `-G` argument indicates the group into which the resource is configured. See the [rt_callbacks\(1HA\)](#) man page for more information about callback methods.

Note - The `Validate` method is called with additional arguments, that is, the property values of the resource and resource group on which it is called. See [“Handling Property Updates” on page 113](#) for more information.

Each callback method needs a function to parse the arguments that the function is passed. Because the callbacks are all passed the same arguments, the data service provides a single parse function that is used in all the callbacks in the application.

The following sample shows the `parse_args()` function that is used for the callback methods in the sample application.

```
#####
# Parse program arguments.
```

```

#
function parse_args # [args ...]
{
typeset opt

while getopts 'R:G:T:' opt
do
case "$opt" in
R)
# Name of the DNS resource.
RESOURCE_NAME=$OPTARG
;;
G)
# Name of the resource group in which the resource is
# configured.
RESOURCEGROUP_NAME=$OPTARG
;;
T)
# Name of the resource type.
RESOURCETYPE_NAME=$OPTARG
;;
*)
logger -p ${SYSLOG_FACILITY}.err \
-t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME] \
"ERROR: Option $OPTARG unknown"
exit 1
;;
esac
done
}

```

Note - Although the PROBE method in the sample application is user defined (not an Oracle Solaris Cluster callback method), it is called with the same arguments as the callback methods. Therefore, this method contains a parse function that is identical to the one that is used by the other callback methods.

The parse function is called in MAIN as:

```
parse_args "$@"
```

Generating Error Messages

Callback methods should use the `syslog()` function to output error messages to end users. All callback methods in the sample data service use the `scha_cluster_get` command to retrieve the number of the `syslog()` function that is used for the cluster log, as follows:

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

The value is stored in a shell variable, `SYSLOG_FACILITY`, and can be used as the facility of the `logger` command to log messages in the cluster log. For example, the `Start` method in the sample data service retrieves the `syslog()` function and logs a message that the data service has been started, as follows:

```
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
```

See the [scha_cluster_get\(1HA\)](#) man page for more information.

Obtaining Property Information

Most callback methods need to obtain information about resource and resource type properties of the data service. The API provides the `scha_resource_get()` function for this purpose.

Both system-defined properties and extension properties are available. System-defined properties are predefined. You define extension properties in the RTR file.

When you use `scha_resource_get()` to obtain the value of a system-defined property, you specify the name of the property with the `-O` option. The command returns only the *value* of the property. For example, in the sample data service, the `Monitor_start` method needs to locate the probe program so it can start it. The probe program is located in the base directory for the data service, which is pointed to by the `RT_basedir` property. The `Monitor_start` method retrieves the value of `RT_basedir` and places it in the `RT_BASEDIR` variable, as follows:

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

For extension properties, you must use the `-O` option to specify that the property is an extension property. You must also supply the name of the property as the last argument. For extension properties, the command returns both the *type* and *value* of the property. For example, in the sample data service, the probe program retrieves the type and value of the `Probe_timeout` extension property, and uses the `awk` command to put the value only in the `PROBE_TIMEOUT` shell variable, as follows:

```
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}'`
```

Controlling the Data Service

A data service must provide a `Start` or `Prenet_start` method to activate the application daemon in the cluster, and a `Stop` or `Postnet_stop` method to stop the application daemon in the cluster. The sample data service implements a `Start` and a `Stop` method. See [“Deciding Which Start and Stop Methods to Use” on page 44](#) for information about when to use `Prenet_start` and `Postnet_stop` instead.

How the Start Method Works

The RGM runs the `Start` method on a cluster node when the resource group that contains the data service resource is brought online on that node or when the resource group is already online and the resource is enabled. In the sample application, the `Start` method activates the `in.named` DNS daemon on the cluster node on that host.

This section describes the major pieces of the `Start` method for the sample application. This section does not describe functionality that is common to all callback methods, such as the `parse_args()` function. This section also does not describe using the `syslog()` function. Common functionality is described in [“Providing Common Functionality to All Methods” on page 92](#).

For the complete listing of the `Start` method, see [“Start Method Code Listing” on page 234](#).

What the Start Method Does

Before attempting to start DNS, the `Start` method in the sample data service verifies that the configuration directory and configuration file (`named.conf`) are accessible and available. Information in `named.conf` is essential to the successful operation of DNS.

This callback method uses the PMF (`pmfadm`) to start the DNS daemon (`in.named`). If DNS crashes or fails to start, the PMF attempts to start the DNS daemon a prescribed number of times during a specified interval. The number of retries and the interval are specified by properties in the data service's RTR file.

Verifying the Configuration

In order to operate, DNS requires information from the `named.conf` file in the configuration directory. Therefore, the `Start` method performs some sanity checks to verify that the directory and file are accessible before attempting to start DNS.

The `Confdir` extension property provides the path to the configuration directory. The property itself is defined in the RTR file. However, the cluster administrator specifies the actual location when the cluster administrator configures the data service.

In the sample data service, the `Start` method retrieves the location of the configuration directory by using the `scha_resource_get()` function.

Note - Because `Confdir` is an extension property, `scha_resource_get()` returns both the type and value. The `awk` command retrieves just the value and places that value in a shell variable, `CONFIG_DIR`.

```
# find the value of Confdir set by the cluster administrator at the time of
# adding the resource.
config_info=`scha_resource_get -O Extension -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAME Confdir`

# scha_resource_get returns the "type" as well as the "value" for the
# extension properties. Get only the value of the extension property
CONFIG_DIR=`echo $config_info | awk '{print $2}'`
```

The `Start` method uses the value of `CONFIG_DIR` to verify that the directory is accessible. If it is not accessible, `Start` logs an error message and exits with an error status. See [“Start Exit Status” on page 100](#).

```
# Check if $CONFIG_DIR is accessible.
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
```

Before starting the application daemon, this method performs a final check to verify that the `named.conf` file is present. If the file is not present, `Start` logs an error message and exits with an error status.

```
# Change to the $CONFIG_DIR directory in case there are relative
# pathnames in the data files.
cd $CONFIG_DIR
```

```

# Check that the named.conf file is present in the $CONFIG_DIR directory
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

```

Starting the Application

This method uses the process manager facility (`pmfadm`) to start the application. The `pmfadm` command enables you to set the number of times to try to restart the application during a specified time frame. The RTR file contains two properties: `Retry_count` specifies the number of times to attempt restarting an application, and `Retry_interval` specifies the time period over which to do so.

The `Start` method retrieves the values of `Retry_count` and `Retry_interval` by using the `scha_resource_get()` function and stores their values in shell variables. The `Start` method passes these values to `pmfadm` by using the `-n` and `-t` options.

```

# Get the value for retry count from the RTR file.
RETRY_CNT=`scha_resource_get -O Retry_count -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAME`
# Get the value for retry interval from the RTR file. This value is in seconds
# and must be converted to minutes for passing to pmfadm. Note that the
# conversion rounds up; for example, 50 seconds rounds up to 1 minute.
((RETRY_INTRVAL=`scha_resource_get -O Retry_interval -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAME` / 60))

# Start the in.named daemon under the control of PMF. Let it crash and restart
# up to $RETRY_COUNT times in a period of $RETRY_INTRVAL; if it crashes
# more often than that, PMF will cease trying to restart it.
# If there is a process already registered under the tag
# <PMF_TAG>, then PMF sends out an alert message that the
# process is already running.
pmfadm -c $PMF_TAG -n $RETRY_CNT -t $RETRY_INTRVAL \
/usr/sbin/in.named -c named.conf

# Log a message indicating that HA-DNS has been started.
if [ $? -eq 0 ]; then
logger -p ${SYSLOG_FACILITY}.err \
-t [${SYSLOG_TAG}] \
"${ARGV0} HA-DNS successfully started"
fi
exit 0

```

Start Exit Status

A Start method should not exit with success until the underlying application is actually running and is available, particularly if other data services depend on it. One way to verify success is to probe the application to make sure that it is running before exiting the Start method. For a complex application, such as a database, be certain to set the value for the `Start_timeout` property in the RTR file sufficiently high to allow time for the application to initialize and recover from a crash.

Note - Because the application resource (DNS) in the sample data service starts quickly, the sample data service does not poll to verify that it is running before exiting with success.

If this method fails to start DNS and exits with failure status, the RGM checks the `Failover_mode` property, which determines how to react. The sample data service does not explicitly set the `Failover_mode` property, so this property has the default value `NONE` (unless the cluster administrator overrides the default value and specifies a different value). In this case, the RGM takes no action other than to set the state of the data service. The cluster administrator needs to initiate a restart on the same node or a fail over to a different node.

How the Stop Method Works

The RGM runs the Stop method on a cluster node when the resource group that contains the HA-DNS resource is brought offline on that node or if the resource group is online and the resource is disabled. This method stops the `in.named` (DNS) daemon on that node.

This section describes the major pieces of the Stop method for the sample application. This section does not describe functionality that is common to all callback methods, such as the `parse_args()` function. This section also does not describe using the `syslog()` function. Common functionality is described in [“Providing Common Functionality to All Methods” on page 92](#).

For the complete listing of the Stop method, see [“Stop Method Code Listing” on page 237](#).

What the Stop Method Does

There are two primary considerations when attempting to stop the data service. The first is to provide an orderly shutdown. Sending a `SIGTERM` signal through `pmfadm` is the best way to accomplish an orderly shutdown.

The second consideration is to ensure that the data service is actually stopped to avoid putting it in `Stop_failed` state. The best way to accomplish putting the data service in this state is to send a `SIGKILL` signal through `pmfadm`.

The `Stop` method in the sample data service takes both of these considerations into account. It first sends a `SIGTERM` signal. If this signal fails to stop the data service, the method sends a `SIGKILL` signal.

Before attempting to stop DNS, this `Stop` method verifies that the process is actually running. If the process is running, `Stop` uses the PMF (`pmfadm`) to stop the process.

This `Stop` method is guaranteed to be idempotent. Although the RGM should not call a `Stop` method twice without first starting the data service with a call to its `Start` method, the RGM could call a `Stop` method on a resource even though the resource was never started or the resource died of its own accord. Therefore, this `Stop` method exits with success even if DNS is not running.

Stopping the Application

The `Stop` method provides a two-tiered approach to stopping the data service: an orderly or smooth approach using a `SIGTERM` signal through `pmfadm` and an abrupt or hard approach using a `SIGKILL` signal. The `Stop` method obtains the `Stop_timeout` value (the amount of time in which the `Stop` method must return). `Stop` allocates 80 percent of this time to stopping smoothly and 15 percent to stopping abruptly (5 percent is reserved), as shown in the following sample code.

```
STOP_TIMEOUT=`scha_resource_get -O STOP_TIMEOUT -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAME`
((SMOOTH_TIMEOUT=$STOP_TIMEOUT * 80/100))
((HARD_TIMEOUT=$STOP_TIMEOUT * 15/100))
```

The `Stop` method uses `pmfadm -q` to verify that the DNS daemon is running. If the DNS daemon is running, `Stop` first uses `pmfadm -s` to send a `TERM` signal to terminate the DNS process. If this signal fails to terminate the process after 80 percent of the timeout value has expired, `Stop` sends a `SIGKILL` signal. If this signal also fails to terminate the process within 15 percent of the timeout value, the method logs an error message and exits with an error status.

If `pmfadm` terminates the process, the method logs a message that the process has stopped and exits with success.

If the DNS process is not running, the method logs a message that it is not running and exits with success anyway. The following code sample shows how `Stop` uses `pmfadm` to stop the DNS process.

```
# See if in.named is running, and if so, kill it.
if pmfadm -q $PMF_TAG; then
# Send a SIGTERM signal to the data service and wait for 80% of the
# total timeout value.
pmfadm -s $RESOURCE_NAME.named -w $SMOOTH_TIMEOUT TERM
if [ $? -ne 0 ]; then
logger -p ${SYSLOG_FACILITY}.err \
-t [$RESOURCE_TYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME] \
"${ARGV0} Failed to stop HA-DNS with SIGTERM; Retry with \
SIGKILL"

# Since the data service did not stop with a SIGTERM signal, use
# SIGKILL now and wait for another 15% of the total timeout value.
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 UNSUCCESSFUL"
exit 1
fi
fi
else
# The data service is not running as of now. Log a message and
# exit success.
logger -p ${SYSLOG_FACILITY}.err \
-t [$SYSLOG_TAG] \
"HA-DNS is not started"

# Even if HA-DNS is not running, exit success to avoid putting
# the data service resource in STOP_FAILED State.
exit 0
fi

# Could successfully stop DNS. Log a message and exit success.
logger -p ${SYSLOG_FACILITY}.err \
-t [$RESOURCE_TYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME] \
"HA-DNS successfully stopped"
exit 0
```

Stop Exit Status

A Stop method should not exit with success until the underlying application is actually stopped, particularly if other data services depend on it. Failure to do so can result in data corruption.

For a complex application, such as a database, be certain to set the value for the `Stop_timeout` property in the RTR file sufficiently high to allow time for the application to clean up while stopping.

If this method fails to stop DNS and exits with failure status, the RGM checks the `Failover_mode` property, which determines how to react. The sample data service does not explicitly set the `Failover_mode` property, so this property has the default value `NONE` (unless the cluster administrator overrides the default value and specifies a different value). In this case, the RGM takes no action other than to set the state of the data service to `Stop_failed`. The cluster administrator needs to stop the application forcibly and clear the `Stop_failed` state.

Defining a Fault Monitor

The sample application implements a basic fault monitor to monitor the reliability of the DNS resource (`in.named`).

The fault monitor consists of the following elements:

- `dns_probe`, a user-defined program that uses `nslookup` to verify that the DNS resource that is controlled by the sample data service is running. If DNS is not running, this method attempts to restart it locally, or depending on the number of restart attempts, requests that the RGM relocate the data service to a different node.
- `dns_monitor_start`, a callback method that starts `dns_probe`. The RGM automatically calls `dns_monitor_start` after the sample data service is brought online if monitoring is enabled.
- `dns_monitor_stop`, a callback method that stops `dns_probe`. The RGM automatically calls `dns_monitor_stop` before bringing the sample data service offline.
- `dns_monitor_check`, a callback method that calls the `Validate` method to verify that the configuration directory is available when the `PROBE` program fails over the data service to a new node.

How the Probe Program Works

The `dns_probe` program implements a continuously running process that verifies that the DNS resource that is controlled by the sample data service is running. The `dns_probe` is started by the `dns_monitor_start` method, which is automatically run by the RGM after the sample data service is brought online. The data service is stopped by the `dns_monitor_stop` method, which the RGM runs before the RGM brings the sample data service offline.

This section describes the major pieces of the `PROBE` method for the sample application. It does not describe functionality that is common to all callback methods, such as

the `parse_args()` function. This section also does not describe using the `syslog()` function. Common functionality is described in [“Providing Common Functionality to All Methods” on page 92](#).

For the complete listing of the PROBE method, see [“PROBE Program Code Listing” on page 240](#).

What the Probe Program Does

The probe runs in an infinite loop. It uses `nslookup` to verify that the correct DNS resource is running. If DNS is running, the probe sleeps for a prescribed interval (set by the `Thorough_probe_interval` system-defined property) and checks again. If DNS is not running, this program attempts to restart it locally, or depending on the number of restart attempts, requests that the RGM relocate the data service to a different node.

Obtaining Property Values

This program requires the values of the following properties:

- `Thorough_probe_interval` – To set the period during which the probe sleeps
- `Probe_timeout` – To enforce the timeout value of the probe on the `nslookup` command that does the probing
- `Network_resources_used` – To obtain the IP address on which DNS is running
- `Retry_count` and `Retry_interval` – To determine the number of restart attempts and the period over which to count them
- `RT_basedir` – To obtain the directory that contains the PROBE program and the `gettime` utility

The `scha_resource_get()` function obtains the values of these properties and stores them in shell variables, as follows:

```
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_INTRVAL=`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`
```

Note - For system-defined properties, such as `Thorough_probe_interval`, the `scha_resource_get()` function returns the value only. For extension properties, such as `Probe_timeout`, the `scha_resource_get()` function returns the type and value. Use the `awk` command to obtain the value only.

Checking the Reliability of the Service

The probe itself is an infinite `while` loop of `nslookup` commands. Before the `while` loop, a temporary file is set up to hold the `nslookup` replies. The `probefail` and `retries` variables are initialized to 0.

```
# Set up a temporary file for the nslookup replies.
DNSPROBEFILE=/var/cluster/run/.$RESOURCE_NAME.probe
probefail=0
retries=0
```

The `while` loop carries out the following tasks:

- Sets the sleep interval for the probe
- Uses `/usr/cluster/bin/hatimerun` to start `nslookup`, passes the `Probe_timeout` value, and identifies the target host
- Sets the `probefail` variable based on the success or failure of the `nslookup` return code
- If `probefail` is set to 1 (failure), verifies that the reply to `nslookup` came from the sample data service and not some other DNS server

Here is the `while` loop code.

```
while :
do
# The interval at which the probe needs to run is specified in the
# property THOROUGH_PROBE_INTERVAL. Therefore, set the probe to sleep
# for a duration of THOROUGH_PROBE_INTERVAL.
sleep $PROBE_INTERVAL

# Run an nslookup command of the IP address on which DNS is serving.
```

```
/usr/cluster/bin/hatimerun -t $PROBE_TIMEOUT /usr/sbin/nslookup $DNS_HOST $DNS_HOST \  
> $DNSPROBEFILE 2>&1  
  
retcode=$?  
if [ $retcode -ne 0 ]; then  
  probefail=1  
fi  
  
# Make sure that the reply to nslookup comes from the HA-DNS  
# server and not from another nameserver mentioned in the  
# /etc/resolv.conf file.  
if [ $probfail -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
```

Comparing Restart With Failover

If the `probfail` variable is something other than 0 (success), the `nslookup` command timed out or the reply came from a server other than the sample service's DNS. In either case, the DNS server is not functioning as expected and the fault monitor calls the `decide_restart_or_failover()` function to determine whether to restart the data service locally or request that the RGM relocate the data service to a different node. If the `probfail` variable is 0, a message is generated that the probe was successful.

```
  if [ $probfail -ne 0 ]; then  
    decide_restart_or_failover  
  else  
    logger -p ${SYSLOG_FACILITY}.err\  
    -t [${SYSLOG_TAG}]\  
    "${ARGV0} Probe for resource HA-DNS successful"  
  fi
```

The `decide_restart_or_failover()` function uses a time window (`Retry_interval`) and a failure count (`Retry_count`) to determine whether to restart DNS locally or request that the RGM relocate the data service to a different node. This function implements the following conditional logic. The code listing for `decide_restart_or_failover()` in [“PROBE Program Code Listing” on page 240](#) contains the code.

- If this is the first failure, restart the data service. Log an error message and bump the counter in the `retries` variable.
- If this is not the first failure, but the window has been exceeded, restart the data service. Log an error message, reset the counter, and slide the window.
- If the time is still within the window and the retry counter has been exceeded, fail over to another node. If the failover does not succeed, log an error and exit the probe program with status 1 (failure).
- If time is still within the window but the retry counter has not been exceeded, restart the data service. Log an error message and bump the counter in the `retries` variable.

If the number of restarts reaches the limit during the time interval, the function requests that the RGM relocate the data service to a different node. If the number of restarts is under the limit, or the interval has been exceeded so the count begins again, the function attempts to restart DNS on the same node.

Note the following points about this function:

- The `gettime` utility is used to track the time between restarts. This is a C program that is located in the `(RT_basedir)` directory.
- The `Retry_count` and `Retry_interval` system-defined resource properties determine the number of restart attempts and the time interval over which to count. These properties default to two attempts in a period of 5 minutes (300 seconds) in the `RTR` file, although the cluster administrator can change these values.
- The `restart_service()` function is called to attempt to restart the data service on the same node. See the next section, [“Restarting the Data Service” on page 107](#), for information about this function.
- The `scha_control()` API function, with the `SCHA_GIVEOVER` argument, brings the resource group that contains the sample data service offline and back online on a different node.

Restarting the Data Service

The `restart_service()` function is called by `decide_restart_or_failover()` to attempt to restart the data service on the same node.

This function executes the following logic:

- Determines if the data service is still registered under the PMF.
 - If the service is still registered, the function carries out the following actions:
 - Obtains the Stop method name and the `Stop_timeout` value for the data service
 - Uses `/usr/cluster/bin/hatimerun` to start the Stop method for the data service, passing the `Stop_timeout` value

- If the data service is successfully stopped, obtains the Start method name and the Start_timeout value for the data service
- Uses hatimerun to start the Start method for the data service, passing the Start_timeout value
- If the data service is no longer registered under the PMF, the implication is that the data service has exceeded the maximum number of allowable retries under the PMF. The scha_control command is run with the GIVEOVER argument to fail over the data service to a different node.

```
function restart_service
{

# To restart the data service, first verify that the
# data service itself is still registered under PMF.
pmfadm -q $PMF_TAG
if [[ $? -eq 0 ]]; then
# Since the TAG for the data service is still registered under
# PMF, first stop the data service and start it back up again.

# Obtain the Stop method name and the STOP_TIMEOUT value for
# this resource.
STOP_TIMEOUT=`scha_resource_get -O STOP_TIMEOUT \
-R $RESOURCE_NAME -G $RESOURCEGROUP_NAME`
STOP_METHOD=`scha_resource_get -O STOP \
-R $RESOURCE_NAME -G $RESOURCEGROUP_NAME`
/usr/cluster/bin/hatimerun -t $STOP_TIMEOUT $RT_BASEDIR/$STOP_METHOD \
-R $RESOURCE_NAME -G $RESOURCEGROUP_NAME \
-T $RESOURCE_TYPE_NAME

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

# Obtain the START method name and the START_TIMEOUT value for
# this resource.
START_TIMEOUT=`scha_resource_get -O START_TIMEOUT \
-R $RESOURCE_NAME -G $RESOURCEGROUP_NAME`
START_METHOD=`scha_resource_get -O START \
-R $RESOURCE_NAME -G $RESOURCEGROUP_NAME`
/usr/cluster/bin/hatimerun -t $START_TIMEOUT $RT_BASEDIR/$START_METHOD \
-R $RESOURCE_NAME -G $RESOURCEGROUP_NAME \
-T $RESOURCE_TYPE_NAME

if [[ $? -ne 0 ]]; then
logger-p ${SYSLOG_FACILITY}.err -t [${SYSLOG_TAG}] \
```

```

"${ARGV0} Start method failed."
return 1
fi

else
# The absence of the TAG for the dataservice
# implies that the data service has already
# exceeded the maximum retries allowed under PMF.
# Therefore, do not attempt to restart the
# data service again, but try to failover
# to another node in the cluster.
scha_control -O GIVEOVER -G $RESOURCEGROUP_NAME \
-R $RESOURCE_NAME
fi

return 0
}

```

Probe Exit Status

The sample data service's PROBE program exits with failure if attempts to restart locally fail and the attempt to fail over to a different node fails as well. This program logs the message `Failover attempt failed.`

How the Monitor_start Method Works

The RGM calls the `Monitor_start` method to start the `dns_probe` method after the sample data service is brought online.

This section describes the major pieces of the `Monitor_start` method for the sample application. This section does not describe functionality that is common to all callback methods, such as the `parse_args()` function. This section also does not describe using the `syslog()` function. Common functionality is described in [“Providing Common Functionality to All Methods” on page 92](#).

For the complete listing of the `Monitor_start` method, see [“Monitor_start Method Code Listing” on page 246](#).

What the Monitor_start Method Does

This method uses the PMF (`pmfadm`) to start the probe.

Starting the Probe

The `Monitor_start` method obtains the value of the `RT_basedir` property to construct the full path name for the `PROBE` program. This method starts the probe by using the infinite retries option of `pmfadm` (`-n -1, -t -1`), which means that if the probe fails to start, the PMF tries to start it an infinite number of times over an infinite period of time.

```
# Find where the probe program resides by obtaining the value of the
# RT_basedir property of the resource.
RT_BASEDIR=`scha_resource_get -O RT_basedir -R $RESOURCE_NAME -G \
$RESOURCEGROUP_NAME`

# Start the probe for the data service under PMF. Use the infinite retries
# option to start the probe. Pass the resource name, type, and group to the
# probe program.
pmfadm -c $RESOURCE_NAME.monitor -n -1 -t -1 \
$RT_BASEDIR/dns_probe -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME \
-T $RESOURCETYPE_NAME
```

How the `Monitor_stop` Method Works

The RGM calls the `Monitor_stop` method to stop execution of `dns_probe` when the sample data service is brought offline.

This section describes the major pieces of the `Monitor_stop` method for the sample application. This section does not describe functionality that is common to all callback methods, such as the `parse_args()` function. This section also does not describe using the `syslog()` function. Common functionality is described in [“Providing Common Functionality to All Methods” on page 92](#).

For the complete listing of the `Monitor_stop` method, see [“Monitor_stop Method Code Listing” on page 248](#).

What the `Monitor_stop` Method Does

This method uses the PMF (`pmfadm`) to check whether the probe is running, and if so, to stop it.

Stopping the Monitor

The `Monitor_stop` method uses `pmfadm -q` to see if the probe is running, and if so, uses `pmfadm -s` to stop it. If the probe is already stopped, the method exits successfully anyway, which guarantees the idempotence of the method.



Caution - Be certain to use the KILL signal with `pmfadm` to stop the probe and not a signal that can be masked, such as `TERM`. Otherwise, the `Monitor_stop` method can hang indefinitely and eventually time out. The reason is that the `PROBE` method calls `scha_control()` when it is necessary to restart or fail over the data service. When `scha_control()` calls `Monitor_stop` as part of the process of bringing the data service offline, if `Monitor_stop` uses a signal that can be masked, `Monitor_stop` hangs waiting for `scha_control()` to complete, and `scha_control()` hangs waiting for `Monitor_stop` to complete.

```
# See if the monitor is running, and if so, kill it.
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
# could successfully stop the monitor. Log a message.
logger -p ${SYSLOG_FACILITY}.err \
-t [${SYSLOG_TAG}] \
"${ARGV0} Monitor for resource " $RESOURCE_NAME \
" successfully stopped"
fi
fi
exit 0
```

Monitor_stop Exit Status

The `Monitor_stop` method logs an error message if it cannot stop the `PROBE` method. The RGM puts the sample data service into `MONITOR_FAILED` state on the primary node, which can panic the node.

`Monitor_stop` should not exit before the probe has been stopped.

How the Monitor_check Method Works

The RGM calls the Monitor_check method whenever the PROBE method attempts to fail over the resource group that contains the data service to a new node.

This section describes the major pieces of the Monitor_check method for the sample application. This section does not describe functionality that is common to all callback methods, such as the parse_args() function. This section also does not describe using the syslog() function. Common functionality is described in [“Providing Common Functionality to All Methods” on page 92](#).

For the complete listing of the Monitor_check method, see [“Monitor_check Method Code Listing” on page 250](#).

The Monitor_check method must be implemented so that it does not conflict with other methods that are running concurrently.

The Monitor_check method calls the Validate method to verify that the DNS configuration directory is available on the new node. The Confdir extension property points to the DNS configuration directory. Therefore, Monitor_check obtains the path and name for the Validate method and the value of Confdir. It passes this value to Validate, as shown in the following listing.

```
# Obtain the full path for the Validate method from
# the RT_basedir property of the resource type.
RT_BASEDIR=`scha_resource_get -O RT_basedir -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAME`

# Obtain the name of the Validate method for this resource.
VALIDATE_METHOD=`scha_resource_get -O Validate \
-R $RESOURCE_NAME -G $RESOURCEGROUP_NAME`

# Obtain the value of the Confdir property in order to start the
# data service. Use the resource name and the resource group entered to
# obtain the Confdir value set at the time of adding the resource.
config_info=`scha_resource_get -O Extension -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAME Confdir`

# scha_resource_get returns the type as well as the value for extension
# properties. Use awk to get only the value of the extension property.
CONFIG_DIR=`echo $config_info | awk '{print $2}'`

# Call the validate method so that the dataservice can be failed over
# successfully to the new node.
$RT_BASEDIR/$VALIDATE_METHOD -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME \
-T $RESOURCETYPE_NAME -x Confdir=$CONFIG_DIR
```


See [“How the Validate Method Works” on page 113](#) to see how the sample application verifies the suitability of a node for hosting the data service.

Handling Property Updates

The sample data service implements `Validate` and `Update` methods to handle the updating of properties by a cluster administrator.

How the Validate Method Works

The RGM calls the `Validate` method when a resource is created and when administrative action updates the properties of the resource or its containing group. The RGM calls `Validate` before the creation or update is applied, and a failure exit code from the method on any node causes the creation or update to be canceled.

The RGM calls `Validate` only when resource or resource group properties are changed by the cluster administrator, not when the RGM sets properties or when a monitor sets the resource properties `Status` and `Status_msg`.

Note - The `Monitor_check` method also explicitly calls the `Validate` method whenever the `PROBE` method attempts to fail over the data service to a new node.

What the Validate Method Does

The RGM calls `Validate` with additional arguments to those that are passed to other methods, including the properties and values that are being updated. Therefore, this method in the sample data service must implement a different `parse_args()` function to handle the additional arguments.

The `Validate` method in the sample data service verifies a single property, the `Confdir` extension property. This property points to the DNS configuration directory, which is critical to the successful operation of DNS.

Note - Because the configuration directory cannot be changed while DNS is running, the `Confdir` property is declared in the RTR file as `TUNABLE = AT_CREATION`. Therefore, the `Validate` method is never called to verify the `Confdir` property as the result of an update, but only when the data service resource is being created.

If `Confdir` is one of the properties that the RGM passes to `Validate`, the `parse_args()` function retrieves and saves its value. `Validate` verifies that the directory pointed to by the new value of `Confdir` is accessible and that the `named.conf` file exists in that directory and contains data.

If the `parse_args()` function cannot retrieve the value of `Confdir` from the command-line arguments that are passed by the RGM, `Validate` still attempts to validate the `Confdir` property. `Validate` uses `scha_resource_get()` to obtain the value of `Confdir` from the static configuration. `Validate` performs the same checks to verify that the configuration directory is accessible and contains a `named.conf` file that is not empty.

If `Validate` exits with failure, the update or creation of all properties, not just `Confdir`, fails.

Validate Method Parsing Function

Because the RGM passes the `Validate` method a different set of arguments than the other callback methods, `Validate` requires a different function for parsing arguments than the other methods. See the [rt_callbacks\(1HA\)](#) man page for more information about the arguments that are passed to `Validate` and the other callback methods. The following code sample shows the `Validate parse_args()` function.

```
#####  
# Parse Validate arguments.  
#  
function parse_args # [args...]  
{  
  
typeset opt  
while getopt 'cur:x:g:R:T:G:X:' opt  
do  
case "$opt" in  
R)  
# Name of the DNS resource.  
RESOURCE_NAME=$OPTARG  
;;  
G)  
# Name of the resource group in which the resource is  
# configured.  
RESOURCEGROUP_NAME=$OPTARG  
;;  
T)  
# Name of the resource type.  
RESOURCETYPE_NAME=$OPTARG  
;;  
X)  

```

```
# Per-node extension property setting. The format of the
# option argument is "propertyname{nodeid}=propertyvalue".
# For example, there might be two -X options with the following
# arguments:
#   myprop{1}=myvalue
#   myprop{2}=othervalue
# representing the setting of property 'myprop' on nodes 1
# and 2.
# In most cases, the -X arguments can be ignored. Instead
# use the -x argument to get the property setting for the
# local node.
;;
r)
# The method is not accessing any system defined
# properties so this is a no-op
;;
g)
# The method is not accessing any resource group
# properties, so this is a no-op
;;
c)
# Indicates the Validate method is being called while
# creating the resource, so this flag is a no-op.
;;
u)
# Indicates the updating of a property when the
# resource already exists. If the update is to the
# Confdir property then Confdir should appear in the
# command-line arguments. If it does not, the method must
# look for it specifically using scha_resource_get.
UPDATE_PROPERTY=1
;;
x)
# Extension property list. Separate the property and
# value pairs using "=" as the separator.
PROPERTY=`echo $OPTARG | awk -F= '{print $1}'`
VAL=`echo $OPTARG | awk -F= '{print $2}'`
# If the Confdir extension property is found on the
# command line, note its value.
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
}

```

As with the `parse_args()` function for other methods, this function provides a flag (R) to capture the resource name, (G) to capture the resource group name, and (T) to capture the resource type that is passed by the RGM.

The `r` flag (which indicates a system-defined property), `g` flag (which indicates a resource group property), and the `c` flag (which indicates that the validation is occurring during creation of the resource) are ignored. They are ignored because this method is being called to validate an extension property when the resource is being updated.

The `u` flag sets the value of the `UPDATE_PROPERTY` shell variable to 1 (TRUE). The `x` flag captures the names and values of the properties that are being updated. If `Confdir` is one of the properties being updated, its value is placed in the `CONFDIR` shell variable, and the variable `CONFDIR_FOUND` is set to 1 (TRUE).

Validating Confdir

In its `MAIN` function, `Validate` first sets the `CONFDIR` variable to the empty string and `UPDATE_PROPERTY` and `CONFDIR_FOUND` to 0.

```

CONFDIR=""
UPDATE_PROPERTY=0
CONFDIR_FOUND=0

```

`Validate` calls `parse_args()` to parse the arguments that are passed by the RGM.

```

parse_args "$@"

```

`Validate` checks if `Validate` is being called as the result of an update of properties. `Validate` also checks if the `Confdir` extension property was on the command line. `Validate` verifies that the `Confdir` property has a value, and if not, exits with failure status and an error message.

```

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

```

```

# Verify that the Confdir property has a value. If not there is a failure

```

```
# and exit with status 1
if [[ -z $CONFDIR ]]; then
logger -p ${SYSLOG_FACILITY}.err \
"${ARGV0} Validate method for resource "$RESOURCE_NAME " failed"
exit 1
fi
```

Note - Specifically, the preceding code checks if `Validate` is being called as the result of an update (`$UPDATE_PROPERTY == 1`) and if the property was *not* found on the command line (`CONFDIR_FOUND == 0`). In this case, the code retrieves the existing value of `Confdir` by using `scha_resource_get()`. If `Confdir` was found on the command line (`CONFDIR_FOUND == 1`), the value of `CONFDIR` comes from the `parse_args()` function, not from `scha_resource_get()`.

The `Validate` method uses the value of `CONFDIR` to verify that the directory is accessible. If the directory is not accessible, `Validate` logs an error message and exits with error status.

```
# Check if $CONFDIR is accessible.
if [ ! -d $CONFDIR ]; then
logger -p ${SYSLOG_FACILITY}.err \
-t [${SYSLOG_TAG}] \
"${ARGV0} Directory $CONFDIR missing or not mounted"
exit 1
fi
```

Before validating the update of the `Confdir` property, `Validate` performs a final check to verify that the `named.conf` file is present. If the file is not present, the method logs an error message and exits with error status.

```
# Check that the named.conf file is present in the Confdir directory
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
```

If the final check is passed, `Validate` logs a message that indicates success and exits with success status.

```
# Log a message indicating that the Validate method was successful.
logger -p ${SYSLOG_FACILITY}.err \
-t [${SYSLOG_TAG}] \
"${ARGV0} Validate method for resource "$RESOURCE_NAME \
" completed successfully"

exit 0
```

Validate Exit Status

If `Validate` exits with success (0), `Confdir` is created with the new value. If `Validate` exits with failure (1), `Confdir` and any other properties are not created and a message that indicates the reason is generated.

How the Update Method Works

The RGM runs the `Update` method to notify a running resource that its properties have been changed. The RGM runs `Update` after the cluster administrator succeeds in setting properties of a resource or its group. This method is called on nodes where the resource is online.

What the Update Method Does

The `Update` method does not update properties. The RGM updates properties. The `Update` method notifies running processes that an update has occurred. The only process in the sample data service that is affected by a property update is the fault monitor. Consequently, the fault monitor process is the process that the `Update` method stops and restarts.

The `Update` method must verify that the fault monitor is running and then kill it by using the `pmfadm` command. The method obtains the location of the probe program that implements the fault monitor, and restarts it by using the `pmfadm` command.

Stopping the Monitor With Update

The `Update` method uses `pmfadm -q` to verify that the monitor is running, and if so, kills it with `pmfadm -s TERM`. If the monitor is successfully terminated, a message to that effect is sent to the cluster administrator. If the monitor cannot be stopped, `Update` exits with failure status and sends an error message to the cluster administrator.

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

# Kill the monitor that is running already
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
# could successfully stop DNS. Log a message.
logger -p ${SYSLOG_FACILITY}.err \
-t [$RESOURCE_TYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME] \
"Monitor for HA-DNS successfully stopped"
fi
```

Restarting the Monitor

To restart the monitor, the Update method must locate the script that implements the probe program. The probe program is located in the base directory for the data service, which is pointed to by the `RT_basedir` property. Update retrieves the value of `RT_basedir` and stores it in the `RT_BASEDIR` variable, as follows.

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

Update uses the value of `RT_BASEDIR` with `pmfadm` to restart the `dns_probe` program. If successful, Update exits with success and sends a message to that effect to the cluster administrator. If `pmfadm` cannot start the probe program, Update exits with failure status and logs an error message.

Update Exit Status

Update method failure causes the resource to be put into an “update failed” state. This state has no effect on RGM management of the resource, but indicates the failure of the update action to administration tools through the `syslog()` function.

Data Service Development Library

This chapter provides an overview of the application programming interfaces that constitute the Data Service Development Library (DSDL). The DSDL is implemented in the `libdsdev.so` library and is included in the Oracle Solaris Cluster package.

This chapter covers the following topics:

- [“DSDL Overview” on page 121](#)
- [“Managing Configuration Properties” on page 121](#)
- [“Starting and Stopping a Data Service” on page 122](#)
- [“Implementing a Fault Monitor” on page 123](#)
- [“Accessing Network Address Information” on page 124](#)
- [“Debugging the Resource Type Implementation” on page 124](#)
- [“Enabling Highly Available Local File Systems” on page 124](#)

DSDL Overview

The DSDL API is layered on top of the Resource Management Application Programming Interface (RMAPI). As such, the DSDL API does not supersede the RMAPI but rather encapsulates and extends the RMAPI functionality. The DSDL simplifies data service development by providing predetermined solutions to specific Oracle Solaris Cluster integration issues. Consequently, you can devote the majority of development time to the high availability and scalability issues that are intrinsic to your application. You spend less time integrating the application startup, shutdown, and monitor procedures with Oracle Solaris Cluster.

Managing Configuration Properties

All callback methods require access to the configuration properties.

The DSDL supports access to properties in these ways:

- Initializing the environment
- Providing a set of convenience functions to retrieve property values

The `scds_initialize()` function, which must be called at the beginning of each callback method, does the following:

- Checks and processes the command-line arguments (`argc` and `argv[]`) that the RGM passes to the callback method, obviating the need for you to write a command-line parsing function.
- Sets up internal data structures for use by other DSDL functions. For example, the convenience functions that retrieve property values from the RGM store the values in these structures. Likewise, values from the command line, which take precedence over values retrieved from the RGM, are stored in these data structures.
- Initializes the logging environment and validates fault monitor probe settings.

Note - For the `Validate` method, `scds_initialize()` parses the property values that are passed on the command line, obviating the need to write a parse function for `Validate`.

The DSDL provides sets of functions to retrieve resource type, resource, and resource group properties as well as commonly used extension properties.

These functions standardize access to properties by using the following conventions:

- Each function takes only a handle argument (returned by `scds_initialize()`).
- Each function corresponds to a particular property. The return value type of the function matches the type of the property value that it retrieves.
- Functions do not return errors as the values have been precomputed by `scds_initialize()`. Functions retrieve values from the RGM unless a new value is passed on the command line.

Starting and Stopping a Data Service

A `Start` method performs the actions that are required to start a data service on a cluster node. Typically, these actions include retrieving the resource properties, locating application-specific executable and configuration files, and starting the application with the correct command-line arguments.

The `scds_initialize()` function retrieves the resource configuration. The `Start` method can use property convenience functions to retrieve values for specific properties, such as `Confdir_list`, that identify the configuration directories and files for the application to start.

A Start method can call `scds_pmf_start()` to start an application under control of the Process Monitor Facility (PMF). The PMF enables you to specify the level of monitoring to apply to the process and provides the ability to restart the process in case of failure. See “[xfnts_start Method](#)” on page 146 for an example of a Start method that is implemented with the DSDL.

A Stop method must be idempotent so that the Stop method exits with success even if it is called on a node when the application is not running. If the Stop method fails, the resource that is being stopped is set to the `STOP_FAILED` state, which can cause the cluster to perform a hard reboot.

To avoid putting the resource in the `STOP_FAILED` state, the Stop method must make every effort to stop the resource. The `scds_pmf_stop()` function provides a phased attempt to stop the resource. This function first attempts to stop the resource by using a `SIGTERM` signal, and if this fails, uses a `SIGKILL` signal. See the [scds_pmf_stop\(3HA\)](#) man page for more information.

Implementing a Fault Monitor

The DSDL absorbs much of the complexity of implementing a fault monitor by providing a predetermined model. A `Monitor_start` method starts the fault monitor, under the control of the PMF, when the resource starts on a node. The fault monitor runs in a loop as long as the resource is running on the node.

The high-level logic of a DSDL fault monitor is as follows:

- The `scds_fm_sleep()` function uses the `Thorough_probe_interval` property to determine the amount of time between probes. Any application process failures that are detected by the PMF during this interval lead to a restart of the resource.
- The probe itself returns a value that indicates the severity of failures, from 0, no failure, to 100 complete failure.
- The probe return value is sent to the `scds_action()` function, which maintains a cumulative failure history within the interval of the `Retry_interval` property.
- The `scds_action()` function determines what to do in the event of a failure, as follows:
 - If the cumulative failure is below 100, do nothing.
 - If the cumulative failure reaches 100 (complete failure), restart the data service. If `Retry_interval` is exceeded, reset the history.
 - If the number of restarts exceeds the value of the `Retry_count` property, within the time specified by `Retry_interval`, fail over the data service.

Accessing Network Address Information

The DSDL provides convenience functions to return network address information for resources and resource groups. For example, the `scds_get_netaddr_list()` retrieves the network address resources that are used by a resource, enabling a fault monitor to probe the application.

The DSDL also provides a set of functions for TCP-based monitoring. Typically, these functions establish a simple socket connect to a service, read and write data to the service, and disconnect from the service. The result of the probe can be sent to the DSDL `scds_fm_action()` function to determine the action to take.

See [“xfnts_validate Method” on page 160](#) for an example of TCP-based fault monitoring.

Debugging the Resource Type Implementation

The DSDL has built-in features to help you debug your data service.

The DSDL utility `scds_syslog_debug()` provides a basic framework for adding debugging statements to the resource type implementation. The debugging *level* (a number between 1-9) can be dynamically set for each resource type implementation on each cluster node. A file named `/var/cluster/rgm/rt/rtname/loglevel`, which contains only an integer between 1 and 9, is read by all resource type callback methods. The DSDL function `scds_initialize()` reads this file and sets the debug level internally to the specified level. The default debug level 0 specifies that the data service is not to log debugging messages.

The `scds_syslog_debug()` function uses the facility that is returned by the `scha_cluster_getlogfacility()` function at a priority of `LOG_DEBUG`. You can configure these debug messages in the `/etc/syslog.conf` file.

You can turn some debugging messages into information messages for regular operation of the resource type (perhaps at `LOG_INFO` priority) by using the `scds_syslog()` function. Note that the sample DSDL application in [Chapter 8, “Sample DSDL Resource Type Implementation”](#) includes calls to the `scds_syslog_debug()` and `scds_syslog()` functions.

Enabling Highly Available Local File Systems

You can use the `SUNW.HASStoragePlus` resource type to make a local file system highly available within an Oracle Solaris Cluster environment.

Note - Local file systems include the UNIX File System (UFS), Quick File System (QFS), and Oracle Solaris ZFS.

The local file system partitions must be located on global disk groups. Affinity switchovers must be enabled, and the Oracle Solaris Cluster environment must be configured for failover. This setup enables the cluster administrator to make any file system that is located on multihost disks accessible from any host that is directly connected to those multihost disks. You use a highly available local file system for selected I/O intensive data services. [“Enabling Highly Available Local File Systems”](#) in *Oracle Solaris Cluster 4.3 Data Services Planning and Administration Guide* contains information about configuring the SUNW.HASStoragePlus resource type.

Designing Resource Types

This chapter explains the typical use of the Data Service Development Library (DSDL) in designing and implementing resource types. This chapter also focuses on designing the resource type to validate the resource configuration, and to start, stop, and monitor the resource. In addition, this chapter describes how to use the DSDL to implement the resource type callback methods.

See the [rt_callbacks\(1HA\)](#) man page for additional information.

You need access to the resource's property settings to complete these tasks. The DSDL utility `scds_initialize()` provides a uniform way to access these resource properties. This function is designed to be called at the beginning of each callback method. This utility function retrieves all the properties for a resource from the cluster framework and makes it available to the family of `scds_getname()` functions.

This chapter covers the following topics:

- “Resource Type Registration File” on page 127
- “Validate Method” on page 128
- “Start Method” on page 130
- “Stop Method” on page 131
- “Monitor_start Method” on page 132
- “Monitor_stop Method” on page 133
- “Monitor_check Method” on page 133
- “Update Method” on page 134
- “Description of Init, Fini, and Boot Methods” on page 135
- “Designing the Fault Monitor Daemon” on page 135

Resource Type Registration File

The Resource Type Registration (RTR) file specifies the details about the resource type to the Oracle Solaris Cluster software.

Details include information as follows:

- Properties that are needed by the implementation
- The data types and default values of those properties
- The file system path for the callback methods for the resource type implementation
- Various settings for the system-defined properties

The sample RTR file that is shipped with the DSDL is sufficient for most resource type implementations. You need only edit some basic elements, such as the resource type name and the path name of the resource type callback methods. If a new property is needed to implement the resource type, you can declare it as an extension property in the RTR file of the resource type implementation, and access the new property by using the DSDL `scds_get_ext_property()` function.

Validate Method

The purpose of the `Validate` callback method of a resource type implementation is to check that the proposed resource settings (as specified by the proposed property settings on the resource) are acceptable to the resource type.

The `Validate` method of a resource type implementation is called by the Resource Group Manager (RGM) under one of the following two conditions:

- A new resource of the resource type is being created
- A property of the resource or resource group is being updated

These two scenarios can be distinguished by the presence of the command-line option `-c` (create) or `-u` (update) that is passed to the `Validate` method of the resource.

The `Validate` method is called on each node of a set of nodes, where the set of nodes is defined by the value of the resource type property `Init_nodes`. If `Init_nodes` is set to `RG_PRIMARYES`, `Validate` is called on each node that can host (be a primary of) the resource group that contains the resource. If `Init_nodes` is set to `RT_INSTALLED_NODES`, `Validate` is called on each node where the resource type software is installed, typically all nodes in the cluster.

The default value of `Init_nodes` is `RG_PRIMARYES` (see the [rt_reg\(4\)](#) man page). At the point the `Validate` method is called, the RGM has not yet created the resource (in the case of creation callback) or has not yet applied the updated values of the properties that are being updated (in the case of update callback).

Note - If you are using local file systems that are managed by the `SUNW.HASStoragePlus` resource type, you use the `scds_hasp_check()` function to check the state of that resource type. This information is obtained from the state (online or otherwise) of all `SUNW.HASStoragePlus` resources on which the resource depends. See the [scds_hasp_check\(3HA\)](#) man page for a complete list of status codes that are returned by the `scds_hasp_check()` function.

The DSDL function `scds_initialize()` handles these situations in the following manner:

- If the resource is being created, `scds_initialize()` parses the proposed resource properties, as they are passed on the command line. The proposed values of resource properties are therefore available to you as though the resource was already created in the system.
- If the resource or resource group is being updated, the proposed values of the properties that are being updated by the cluster administrator are read in from the command line. The remaining properties (whose values are not being updated) are read in from Oracle Solaris Cluster by using the Resource Management API. If you are using the DSDL, you do not need to concern yourself with these tasks. You can validate a resource as if all the properties of the resource were available.

Suppose the function that implements the validation of a resource's properties is called `svc_validate()`, which uses the `scds_get_name()` family of functions to look at the property to be validated. Assuming that an acceptable resource setting is represented by a 0 return code from this function, the `Validate` method of the resource type can thus be represented by the following code fragment:

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

    if (scds_initialize(&handle, argc, argv) != SCHA_ERR_NOERR) {
        return (1); /* Initialization Error */
    }
    rc = svc_validate(handle);
    scds_close(&handle);
    return (rc);
}
```

The validation function should also log the reason why the validation of the resource failed. However, by leaving out that detail ([Chapter 8, “Sample DSDL Resource Type Implementation”](#) contains a more realistic treatment of a validation function), you can implement a simpler example `svc_validate()` function, as follows:

```
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); /* Invalid resource property setting */
    }
    return (0); /* Acceptable setting */
}
```

Thus, you must concern yourself with only the implementation of the `svc_validate()` function.

Start Method

The `Start` callback method of a resource type implementation is called by the RGM on a chosen cluster node to start the resource. The resource group name, the resource name, and resource type name are passed on the command line. The `Start` method performs the actions that are needed to start a data service resource in the cluster node. Typically this involves retrieving the resource properties, locating the application specific executable file, configuration files, or both, and starting the application with the correct command-line arguments.

With the DSDL, the resource configuration is already retrieved by the `scds_initialize()` function. The startup action for the application can be contained in a function `svc_start()`. Another function, `svc_wait()`, can be called to verify that the application actually starts. The simplified code for the `Start` method is as follows:

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

    if (scds_initialize(&handle, argc, argv) != SCHA_ERR_NOERR) {
        return (1); /* Initialization Error */
    }
    if (svc_validate(handle) != 0) {
        return (1); /* Invalid settings */
    }
    if (svc_start(handle) != 0) {
        return (1); /* Start failed */
    }
    return (svc_wait(handle));
}
```

This start method implementation calls `svc_validate()` to validate the resource configuration. If it fails, either the resource configuration and application configuration do not match or there is currently a problem on this cluster node with regard to the system. For example, a cluster file system that is needed by the resource might currently not be available on this cluster node. In this case, it is futile to attempt to start the resource on this cluster node. It is better to let the RGM attempt to start the resource on a different node.

Note, however, that the preceding statement assumes that `svc_validate()` is sufficiently conservative, checking only for resources on the cluster node that are required by the application. Otherwise, the resource might fail to start on all cluster nodes and thus enter a `START_FAILED` state. See the [Oracle Solaris Cluster 4.3 Data Services Planning and Administration Guide](#) for an explanation of this state.

The `svc_start()` function must return 0 for a successful startup of the resource on the node. If the startup function encounters a problem, it must return nonzero. Upon failure of this function, the RGM attempts to start the resource on a different cluster node.

To take advantage of the DSDL as much as possible, the `svc_start()` function can call the `scds_pmf_start()` utility to start the application under the Process Monitor Facility (PMF). This utility also uses the failure callback action feature of the PMF to detect process failure. See the description of the `-a` action argument in the [pmfadm\(1M\)](#) man page for more information.

Stop Method

The Stop callback method of a resource type implementation is called by the RGM on a cluster node to stop the application.

The callback semantics for the Stop method demand the following factors:

- The Stop method must be *idempotent* because the Stop method can be called by the RGM even if the Start method did not complete successfully on the node. Thus, the Stop method must succeed (exit zero) even if the application is not currently running on the cluster node and there is no work for it to do.
- If the Stop method of the resource type fails (exits nonzero) on a cluster node, the resource that is being stopped enters the `STOP_FAILED` state. Depending on the `Failover_mode` setting on the resource, this condition might lead the RGM to perform a hard reboot of the cluster node.

Thus, you must design the Stop method so that this method definitely stops the application. You might even need to resort to using `SIGKILL` to kill the application abruptly if the application otherwise fails to terminate.

You must also ensure that this method stops the application in a timely fashion because the framework treats expiry of the `Stop_timeout` property as a stop failure, and consequently puts the resource in a `STOP_FAILED` state.

The DSDL utility `scds_pmf_stop()` should suffice for most applications as it first attempts to softly stop the application with `SIGTERM`. This function then delivers a `SIGKILL` to the process. This function assumes that the application was started under the PMF with `scds_pmf_start()`. See “[PMF Functions](#)” on page 191 for details about this utility.

Assuming that the application-specific function that stops the application is called `svc_stop()`, implement the `Stop` method as follows:

```
if (scds_initialize(&handle, argc, argv) != SCHA_ERR_NOERR)
{
    return (1); /* Initialization Error */
}
return (svc_stop(handle));
```

Whether or not the implementation of the preceding `svc_stop()` function includes the `scds_pmf_stop()` function is irrelevant. Your decision to include the `scds_pmf_stop()` function depends on whether or not the application was started under the PMF through the `Start` method.

The `svc_validate()` method is not used in the implementation of the `Stop` method because, even if the system is currently experiencing a problem, the `Stop` method should attempt to stop the application on this node.

Monitor_start Method

The RGM calls the `Monitor_start` method to start a fault monitor for the resource. Fault monitors monitor the health of the application that is being managed by the resource. Resource type implementations typically implement a fault monitor as a separate daemon that runs in the background. The `Monitor_start` callback method is used to start this daemon with the correct arguments.

Because the monitor daemon itself is prone to failures (for example, it could die, leaving the application unmonitored), you should use the PMF to start the monitor daemon. The DSDL utility `scds_pmf_start()` has built-in support for starting fault monitors. This utility uses the path name that is relative to the `RT_basedir` for the location of the resource type callback method implementations of the monitor daemon program. This utility uses the `Monitor_retry_interval` and `Monitor_retry_count` extension properties that are managed by the DSDL to prevent unlimited restarts of the daemon.

This utility also imposes the same command-line syntax as defined for all callback methods (that is, `-R resource -G resource-group -T resource-type`) onto the monitor daemon, although the monitor daemon is never called directly by the RGM. Finally, this utility also allows the monitor daemon implementation itself to enable the `scds_initialize()` utility to set up its own environment. The main effort is in designing the monitor daemon itself.

Monitor_stop Method

The RGM calls the `Monitor_stop` method to stop the fault monitor daemon that was started with the `Monitor_start` method. Failure of this callback method is treated in exactly the same fashion as failure of the `Stop` method. Therefore, the `Monitor_stop` method must be idempotent and just as robust as the `Stop` method.

If you use the `scds_pmf_start()` utility to start the fault monitor daemon, use the `scds_pmf_stop()` utility to stop it.

Monitor_check Method

The RGM runs the `Monitor_check` callback method on a resource on a node for the specified resource to ascertain whether the cluster node is capable of mastering the resource. In other words, the RGM runs this method to determine whether the application that is being managed by the resource can run successfully on the node.

Typically, this situation involves ensuring that all the system resources that are required by the application are indeed available on the cluster node. As discussed in [“Validate Method” on page 128](#), the function `svc_validate()` that you implement is intended to ascertain at least that.

Depending on the specific application that is being managed by the resource type implementation, the `Monitor_check` method can be written to carry out additional tasks. The `Monitor_check` method must be implemented so that it does not conflict with other methods that are running concurrently. If you are using the DSDL, the `Monitor_check` method should call the `svc_validate()` function, which implements application-specific validation of resource properties.

Update Method

The RGM calls the `Update` method of a resource type implementation to apply any changes that were made by the cluster administrator to the configuration of the active resource. The `Update` method is only called on nodes (if any) where the resource is currently online.

The changes that have just been made to the resource configuration are guaranteed to be acceptable to the resource type implementation because the RGM runs the `Validate` method of the resource type before it runs the `Update` method. The `Validate` method is called before the resource or resource group properties are changed, and the `Validate` method can veto the proposed changes. The `Update` method is called after the changes have been applied to give the active (online) resource the opportunity to take notice of the new settings.

You must carefully determine the properties that you want to be able to update dynamically, and mark those with the `TUNABLE = ANYTIME` setting in the RTR file. Typically, you can specify that you want to be able to dynamically update any property of a resource type implementation that the fault monitor daemon uses. However, the implementation of the `Update` method must at least restart the monitor daemon.

Possible properties that you can use are as follows:

- `Thorough_probe_interval`
- `Retry_count`
- `Retry_interval`
- `Monitor_retry_count`
- `Monitor_retry_interval`
- `Probe_timeout`

These properties affect the way a fault monitor daemon checks the health of the service, how often the daemon performs checks, the history interval that the daemon uses to keep track of the errors, and the restart thresholds that are set by the PMF. To implement updates of these properties, the utility `scds_pmf_restart()` is provided in the DSDL.

If you need to be able to dynamically update a resource property, but the modification of that property might affect the running application, you need to implement the correct actions. You must ensure that the updates to that property are correctly applied to any running instances of the application. Currently, you cannot use the DSDL to dynamically update a resource property in this way. You cannot pass the modified properties to `Update` on the command line (as you can with `Validate`).

Description of Init, Fini, and Boot Methods

These methods are *one-time action* methods as defined by the Resource Management API specifications. The sample implementation that is included with the DSDL does not illustrate the use of these methods. However, all the facilities in the DSDL are available to these methods as well, should you need these methods. Typically, the `Init` and the `Boot` methods would be exactly the same for a resource type implementation to implement a *one-time action*. The `Fini` method typically would perform an action that *undoes* the action of the `Init` or `Boot` methods.

Designing the Fault Monitor Daemon

Resource type implementations that use the DSDL typically have a fault monitor daemon that carries out the following responsibilities:

- Periodically monitors the health of the application that is being managed. This particular responsibility of a monitor daemon largely depends on the particular application and can vary widely from resource type to resource type. The DSDL contains some built-in utility functions that perform health checks for simple TCP-based services. You can use these utilities to implement applications that use ASCII-based protocols, such as HTTP, NNTP, IMAP, and POP3.
- Keeps track of the problems that are encountered by the application by using the resource properties `Retry_interval` and `Retry_count`. When the application fails completely, the fault monitor needs to determine whether the PMF action script should restart the service or whether the application failures have accumulated so rapidly that a failover needs to be carried out. The DSDL utilities `scds_fm_action()` and `scds_fm_sleep()` are intended to aid you in implementing this mechanism.
- Takes action, typically either restarting the application or attempting a failover of the containing resource group. The DSDL utility `scds_fm_action()` implements this algorithm. This utility computes the current accumulation of probe failures in the past number of `Retry_interval` seconds for this purpose.
- Updates the resource state so that the state of the application's health is available to the Oracle Solaris Cluster administrative commands, as well as to the cluster management GUI. .

The DSDL utilities are designed so that the main loop of the fault monitor daemon can be represented by the pseudo code at the end of this section.

Keep the following factors in mind when you implement a fault monitor with the DSDL:

- `scds_fm_sleep()` detects the death of an application process rapidly because notification of the application process's death through the PMF is asynchronous. Thus, the fault detection

time is reduced significantly, thereby increasing the availability of the service. A fault monitor might otherwise wake up every so often to check on a service's health and find that the application process has died.

- If the RGM rejects the attempt to fail over the service with the `scha_control` API, `scds_fm_action()` *resets*, or forgets, its current failure history. This function resets its current failure history because its history already exceeds `Retry_count`. If the monitor daemon wakes up in the next iteration and is unable to successfully complete its health check of the daemon, the monitor daemon again attempts to call the `scha_control()` function. That call is probably rejected once again, as the situation that led to its rejection in the last iteration is still valid. Resetting the history ensures that the fault monitor at least attempts to correct the situation locally (for example, through restarting the application) in the next iteration.
- `scds_fm_action()` does *not* reset application failure history in case of restart failures, as you would typically like to issue `scha_control()` quickly thereafter if the situation does not correct itself.
- The utility `scds_fm_action()` updates the resource status to `SCHA_RSSTATUS_OK`, `SCHA_RSSTATUS_DEGRADED`, or `SCHA_RSSTATUS_FAULTED` depending on the failure history. This status is consequently available to cluster system management.

In most cases, you can implement the application-specific health check action in a separate stand-alone utility (`svc_probe()`, for example). You can integrate it with the following generic main loop.

```
for (;;) {
/* sleep for a duration of thorough_probe_interval between
 * successive probes.
 */
(void) scds_fm_sleep(scds_handle,
scds_get_rs_thorough_probe_interval(scds_handle));
/* Now probe all ipaddress we use. Loop over
 * 1. All net resources we use.
 * 2. All ipaddresses in a given resource.
 * For each of the ipaddress that is probed,
 * compute the failure history.
 */
probe_result = 0;
/* Iterate through the all resources to get each
 * IP address to use for calling svc_probe()
 */
for (ip = 0; ip < netaddr->num_netaddrs; ip++) {
/* Grab the hostname and port on which the
 * health has to be monitored.
 */
hostname = netaddr->netaddrs[ip].hostname;
port = netaddr->netaddrs[ip].port_proto.port;
```



```

/*
 * HA-XFS supports only one port and
 * hence obtain the port value from the
 * first entry in the array of ports.
 */
ht1 = gethrtime();
/* Latch probe start time */
probe_result = svc_probe(scds_handle, hostname, port, timeout);
/*
 * Update service probe history,
 * take action if necessary.
 * Latch probe end time.
 */
ht2 = gethrtime();
/* Convert to milliseconds */
dt = (ulong_t)((ht2 - ht1) / 1e6);
/*
 * Compute failure history and take
 * action if needed
 */
(void) scds_fm_action(scds_handle,
probe_result, (long)dt);
}      /* Each net resource */
}      /* Keep probing forever */

```

Supporting Resource Types That Perform Resource Migration From Their Stop Method

Most RGM resource types separate their stopping and starting actions using the Stop or Postnet_stop and Start or Prenet_start methods. In some cases, such as in Oracle VM Server for SPARC that supports live migration, an entire switchover of the resource is performed by the Stop method. For example, if an Oracle VM Server for SPARC logical domain starts out on node1 and you execute a switchover to node2, the RGM executes the Stop method on node1 and the resource goes offline. In addition, during the execution of the Stop method on node1, the Oracle VM Server for SPARC logical domain is switched over to node2 using live migration in which case the Start method executed on node2 is a no-op.

A problem can occur if another resource group declares a strong negative (SN) affinity for the resource's group or if the RGM load limits are in use. In such a case, the logical domain actually migrates onto the target node before the RGM can evict any resource group or groups that need to be evicted due to strong negative affinities or load limits. Hence, there is a time interval in which the hard load limit or SN affinity is violated, resulting in overload of the node and consequent failure of the live migration attempt.

The following two features allow a data service that performs a switchover using live migration from the Stop method, to find out the target node of the switchover and to trigger any required resource group evictions caused by strong negative affinities or load limits before the switchover is initiated.

- SCHA_TARGET_NODES query
- Pre_evict resource property

SCHA_TARGET_NODES Query

The SCHA_TARGET_NODES query tag for `scha_resourcegroup_get` returns a list of target node names to which the given resource group is currently in the process of switching over. This query returns a meaningful value while the resource group is in PENDING_OFFLINE state on its current master or in PENDING_ONLINE state on the target node of the switchover. This query is typically called from a Stop or Postnet_stop method of a resource within the switching resource group. The output argument type is `scha_str_array_t**`. The corresponding operation tag for the command line form of `scha_resourcegroup_get` is Target_nodes. See the [scha_resourcegroup_get\(1HA\)](#) and the [scha_resourcegroup_open\(3HA\)](#) man pages for more information.

Note the following points about the SCHA_TARGET_NODES query tag:

- If a switchover is not in progress on the given resource group, the Target_nodes query returns an empty list.
- If a switchover is in progress on a multi-mastered resource group, in which the Maximum primaries setting is greater than 1, the list returned by the Target_node query contains all nodes in the targeted mastery of the resource group, excluding current masters. For a single-mastered resource group, the returned list contains at most one target node.
- The SCHA_TARGET_NODES query returns the target nodes for failbacks, `scha_control` giveovers, or user-initiated resource group switch or resource group remaster commands. Resource group switches include `clresourcegroup switch` or `clresourcegroup online` commands.
- The SCHA_TARGET_NODES query tag does not return the destination nodes for resource groups that are taken offline due to a node evacuation, a resource group eviction, a strong affinity, or a Start method failure. In such cases, the SCHA_TARGET_NODES query returns an empty list.

The SCHA_TARGET_NODES query is called by a stopping method of a resource to determine the target node for a live migration. Note that the return value of an empty list indicates that the resource group is going offline but it does not necessarily imply that the resource group

will remain offline. For example, in the case of a start failure or node evacuation of a single-mastered resource group, the `SCHA_TARGET_NODES` query returns an empty list but the resource group might still fail over to another node. As a result of this limitation, a data service might not be able to perform live migration for node evacuations or for failovers due to the failure of Start methods.

If a resource group is switching nodes due to a strong affinity only, the `SCHA_TARGET_NODES` query returns an empty list. For example, if RG1 declares a strong positive (`++` or `+++`) affinity for RG2 and you switch RG2 from node1 to node2, RG1 is also switched from node1 to node2. However, a `SCHA_TARGET_NODES` query on RG1 will return an empty list, which would prevent live migration from being applied. If RG1 and RG2 are in the same global cluster or zone cluster, the Oracle Solaris Cluster administrator can work around this issue by using the `clresourcegroup switch` command to switch both RG1 and RG2 together, instead of relying on the strong positive affinity. Having both resource groups as operands of the `clresourcegroup switch` command allows the `Target_nodes` query to work correctly on both of them.

EXAMPLE 3 C Form of the `SCHA_TARGET_NODES` Query

```
#include <scha.h>...
scha_err_t err;
scha_str_array_t *node_list;
scha_resourcegroup_t handle;
int ix;
char *rgname = "example_RG";
err = scha_resourcegroup_open(rgname, &handle);
err = scha_resourcegroup_get(handle, SCHA_TARGET_NODES, &node_list);
if (err == SCHA_ERR_NOERR) {
    for (ix = 0; ix < node_list->array_cnt; ix++) {
        printf("Group %s target node: %s\n", rgname,
            node_list->str_array[ix]);
    }
}
...
```

EXAMPLE 4 CLI Form of the `SCHA_TARGET_NODES` Query

```
# scha_resourcegroup_get -G mygroup -O Target_nodes
node2
#
```

Pre_evict Resource Property

The `Pre_evict` property allows the RGM to prestart a resource group, RG1, on a target node before RG1 has actually switched offline from its current master. The prestart action evicts the resource groups from the target node prior to RG1 executing its switchover onto the target node.

The prestart action is executed before the live migration so that the target node will satisfy the necessary load and affinity prerequisites for hosting the migrating resource group.

Note the following points for the `Pre_evict` resource property:

- The RGM evicts RG2 from the target node prior to executing the switchover of RG1 under the following conditions:
 - If a resource group RG1 contains a resource with `Pre_evict` set to `TRUE`
 - If RG1 is being switched from one node to another
 - If RG1 preempts another resource group RG2 on the target node due to strong negative affinities or loadlimits and causes the eviction of RG2

The eviction is completed before the execution of any stopping methods of resources in RG1.

- During the pre-eviction action, the state of RG1 on its current master is set to `PENDING_OFFLINE`. The status message of each resource in RG1 that has `Pre_evict=TRUE` is set on RG1's current master node to `Waiting for resource group evictions on node %s`, where `%s` indicates the name of the switchover target node. The status message is localized to the system locale that is applicable to `rgmd`'s environment, which is normally the default system locale.
- The pre-eviction action is taken only for single-mastered resource groups, that is for resource groups with the `Maximum primaries` property set to 1. For a resource group with `Maximum primaries` property set to greater than 1, the `Pre_evict` settings of its resources is ignored, and prestart/eviction actions do not take place on the target node or nodes.
- If RG1 is only being brought online on a new master but not being switched off of any current master, resource group evictions take place on the new master before bringing RG1's resources online, regardless of the setting of the `Pre_evict` resource property. In such a case, pre-eviction is equivalent to the default eviction because there is no stopping phase for RG1.
- If RG1 goes offline due to a `Startmethod` failure, a resource group eviction, a strong affinity, or a node evacuation, the prestart/eviction action will not take place regardless of the setting of `Pre_evict`.

For example, if RG1 declares a strong positive (`++` or `+++`) affinity for RG2 and you switch RG2 from `node1` to `node2`, RG1 is also switched from `node1` to `node2` but pre-eviction is not performed on RG1. This prevents live migration from being applied to RG1. If RG1 and RG2

are in the same global cluster or zone cluster, the Oracle Solaris Cluster administrator can work around this issue by using the `clrrsourcegroup switch` command to switch both RG1 and RG2 together instead of relying on the strong positive affinity. Having both resource groups as operands of the `clrrsourcegroup switch` command allows the pre- eviction feature to work correctly on both of them.

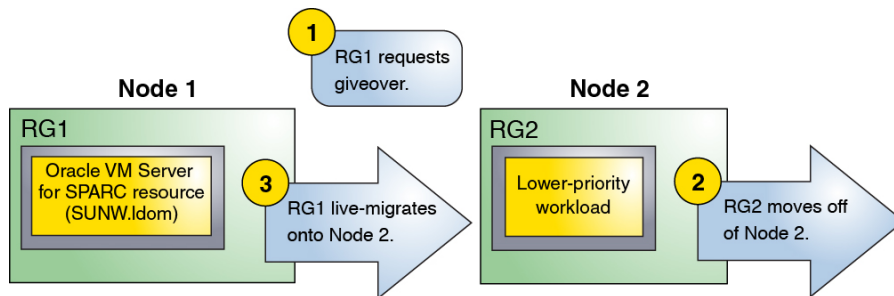
Note that in such cases where pre- eviction is not performed, the evictions are done in the default manner. Evictions occur after stopping the switching resource group on its original master and before starting it on the target node.

For information about the `Pre_evict` resource property, see the [r_properties\(5\)](#) man page.

Example of a Giveover of a Resource Group With Live Migration

The following figure shows an example of a giveover of the resource group RG1, which contains an Oracle VM Server for SPARC resource.

FIGURE 2 Example of a Giveover of a Resource Group



The giveover switches RG1 from Node1 to Node2, which results in eviction of resource group RG2 from Node2.

With the default RGM behavior, RG1 would execute its stopping methods on Node1 before RG2 is evicted from Node2.

With the newly-implemented behavior, RG2 is evicted from Node2 before RG1 even begins to stop on Node1. This clears the way for RG1 to use live migration without confronting the

workload imposed by RG2 on Node2. The Stop method of the resource type, SUNW.ldom, can use the SCHA_TARGET_NODES query to find out the switchover target node, Node2, to which the live migration needs to be performed.

Error Handling

During a switchover of a resource group in which the prestart and eviction actions are triggered, if the evicted resource group encounters Stop method failures, the switchover attempt returns failure and the switchover operand remains on its current master.

Sample DSDL Resource Type Implementation

This chapter describes a sample resource type, `ORCL.xfnts`, which is implemented with the Data Service Development Library (DSDL). This data service is written in C. The underlying application is the X Font Server, a TCP/IP-based service. [Appendix B, “DSDL Sample Resource Type Code Listings”](#) contains the complete code for each method in the `ORCL.xfnts` resource type.

This chapter covers the following topics:

- “X Font Server” on page 143
- “ORCL.xfnts RTR File” on page 145
- “Naming Conventions for Functions and Callback Methods” on page 145
- “`scds_initialize()` Function” on page 146
- “`xfnts_start` Method” on page 146
- “`xfnts_stop` Method” on page 151
- “`xfnts_monitor_start` Method” on page 152
- “`xfnts_monitor_stop` Method” on page 153
- “`xfnts_monitor_check` Method” on page 154
- “ORCL.xfnts Fault Monitor” on page 154
- “`xfnts_validate` Method” on page 160
- “`xfnts_update` Method” on page 163

X Font Server

The X Font Server is a TCP/IP-based service that serves font files to its clients. Clients connect to the server to request a font set, and the server reads the font files off the disk and serves them to the clients. The X Font Server daemon consists of a server binary at `/usr/bin/xfns`. The daemon is normally started from `inetd`. However, for the current sample, assume that the correct entry in the `/etc/inetd.conf` file has been disabled (for example, by using the `fsadmin -d` command) so that the daemon is under sole control of the Oracle Solaris Cluster software.

X Font Server Configuration File

By default, the X Font Server reads its configuration information from the file `/usr/openwin/lib/X11/fontserver.cfg`. The catalog entry in this file contains a list of font directories that are available to the daemon for serving. The cluster administrator can locate the font directories in the cluster file system. This location optimizes the use of the X Font Server on Oracle Solaris Cluster by maintaining a single copy of the font's database on the system. If the cluster administrator wants to change the location, the cluster administrator must edit `fontserver.cfg` to reflect the new paths for the font directories.

For ease of configuration, the cluster administrator can also place the configuration file itself in the cluster file system. The `xfs` daemon provides command-line arguments that override the default, built-in location of this file. The `ORCL.xfnts` resource type uses the following command to start the daemon under the control of the Oracle Solaris Cluster software.

```
/usr/bin/xfs -config location-of-configuration-file/fontserver.cfg \  
-port port-number
```

In the `ORCL.xfnts` resource type implementation, you can use the `Confdir_list` property to manage the location of the `fontserver.cfg` configuration file.

TCP Port Number

The TCP port number on which the `xfs` server daemon listens is normally the “fs” port, typically defined as `7100` in the `/etc/services` file. However, the `-port` option that the cluster administrator includes with the `xfs` command enables the cluster administrator to override the default setting.

You can use the `Port_list` property in the `ORCL.xfnts` resource type to set the default value and to enable the cluster administrator to use the `-port` option with the `xfs` command. You define the default value of this property as `7100/tcp` in the RTR file. In the `ORCL.xfnts` Start method, you pass `Port_list` to the `-port` option on the `xfs` command line. Consequently, a user of this resource type is not required to specify a port number (the port defaults to `7100/tcp`). The cluster administrator can specify a different value for the `Port_list` property when the cluster administrator configures the resource type.

ORCL.xfnts RTR File

This section describes several key properties in the ORCL.xfnts RTR file. It does not describe the purpose of each property in the file. For such a description, see [“Setting Resource and Resource Type Properties” on page 32](#).

The `Confdir_list` extension property identifies the configuration directory (or a list of directories), as follows:

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

The `Confdir_list` property does not specify a default value. The cluster administrator must specify a directory name when the resource is created. This value cannot be changed later because tunability is limited to `AT_CREATION`.

The `Port_list` property identifies the port on which the application listens, as follows:

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

Because the property declares a default value, the cluster administrator can specify a new value or accept the default value when the resource is created. No one can change this value later because tunability is limited to `AT_CREATION`.

Naming Conventions for Functions and Callback Methods

You can identify the various pieces of the sample code by knowing these conventions:

- RMAPI functions begin with `scha_`.
- DSDL functions begin with `scds_`.
- Callback methods begin with `xfnts_`.
- User-written functions begin with `svc_`.

scds_initialize() Function

The DSDL requires that each callback method call the `scds_initialize()` function at the beginning of the method.

This function performs the following operations:

- Checks and processes the command-line arguments (`argc` and `argv`) that the framework passes to the data service method. The method does not have to process any additional command-line arguments.
- Sets up internal data structures for use by the other functions in the DSDL.
- Initializes the logging environment.
- Validates fault monitor probe settings.

Use the `scds_close()` function to reclaim the resources that are allocated by `scds_initialize()`.

xfnts_start Method

The RGM runs the Start method on a cluster node when the resource group that contains the data service resource is brought online on that node or when the resource is enabled. In the `ORCL.xfnts` sample resource type, the `xfnts_start` method activates the `xfns` daemon on that node.

The `xfnts_start` method calls `scds_pmf_start()` to start the daemon under the PMF. The PMF provides automatic failure notification and restart features, as well as integration with the fault monitor.

Note - The first call in `xfnts_start` is to `scds_initialize()`, which performs some necessary *housekeeping* functions. “[scds_initialize\(\) Function](#)” on page 146 and the [scds_initialize\(3HA\)](#) man page contain more information.

Validating the Service Before Starting the X Font Server

Before the `xfnts_start` method attempts to start the X Font Server, it calls `svc_validate()` to verify that a correct configuration is in place to support the `xfns` daemon.

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

See “[xfnts_validate Method](#)” on page 160 for details.

Starting the Service With `svc_start()`

The `xfnts_start` method calls the `svc_start()` method, which is defined in the `xfnts.c` file, to start the xfs daemon. This section describes `svc_start()`.

The command to start the xfs daemon is as follows:

```
# xfs -config config-directory/fontserver.cfg -port port-number
```

The `Confdir_list` extension property identifies the *config-directory* while the `Port_list` system property identifies the *port-number*. The cluster administrator provides specific values for these properties when he or she configures the data service.

The `xfnts_start` method declares these properties as string arrays. The `xfnts_start` method obtains the values that the cluster administrator sets by using the `scds_get_ext_confdir_list()` and `scds_get_port_list()` functions. These functions are described in the [scds_property_functions\(3HA\)](#) man page.

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

/* get the configuration directory from the confdir_list property */
confdirs = scds_get_ext_confdir_list(scds_handle);

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

/* obtain the port to be used by XFS from the Port_list property */
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);
}
```

Note that the `confdirs` variable points to the first element (0) of the array.

The `xfnts_start` method uses `sprintf()` to form the command line for `xfns`.

```
/* Construct the command to start the xfs daemon. */
(void) sprintf(cmd,
"/usr/bin/xfns -config %s -port %d 2>/dev/null",
xfnts_conf, portlist->ports[0].port);
```

Note that the output is redirected to `/dev/null` to suppress messages that are generated by the daemon.

The `xfnts_start` method passes the `xfns` command line to `scds_pmf_start()` to start the data service under the control of the PMF.

```
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 ");
}
```

Note the following points about the call to `scds_pmf_start()`:

- The `SCDS_PMF_TYPE_SVC` argument identifies the program to start as a data service application. This method can also start a fault monitor or some other type of application.
- The `SCDS_PMF_SINGLE_INSTANCE` argument identifies this as a single-instance resource.
- The `cmd` argument is the command line that was generated previously.
- The final argument, `-1`, specifies the child monitoring level. The `-1` value specifies that the PMF monitor all children as well as the original process.

Before returning, `scds_pmf_start()` frees the memory that is allocated for the `portlist` structure.

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

Returning From `svc_start()`

Even when `svc_start()` returns successfully, the underlying application might have failed to start. Therefore, `svc_start()` must probe the application to verify that it is running before returning a success message. The probe must also take into account that the application might

not be immediately available because it takes some time to start. The `svc_start()` method calls `svc_wait()`, which is defined in the `xfnts.c` file, to verify that the application is running.

```
/* Wait for the service to start up fully */
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.");
}
```

The `svc_wait()` function calls `scds_get_netaddr_list()` to obtain the network address resources that are needed to probe the application.

```
/* obtain the network resource to use for probing */
if (scds_get_netaddr_list(scds_handle, &netaddr) {
scds_syslog(LOG_ERR,
"No network address resources found in resource group.");
return (1);
}

/* Return an error if there are no network resources */
if (netaddr == NULL || netaddr->num_netaddrs == 0) {
scds_syslog(LOG_ERR,
"No network address resource in resource group.");
return (1);
}
```

The `svc_wait()` function obtains the `Start_timeout` and `Stop_timeout` values.

```
svc_start_timeout = scds_get_rs_start_timeout(scds_handle)
probe_timeout = scds_get_ext_probe_timeout(scds_handle)
```

To account for the time the server might take to start, `svc_wait()` calls `scds_svc_wait()` and passes a timeout value equivalent to three percent of the `Start_timeout` value. The `svc_wait()` function calls the `svc_probe()` function to verify that the application has started. The `svc_probe()` method makes a simple socket connection to the server on the specified port. If it fails to connect to the port, `svc_probe()` returns a value of 100, which indicates a total failure. If the connect goes through but the disconnect to the port fails, `svc_probe()` returns a value of 50.

On failure or partial failure of `svc_probe()`, `svc_wait()` calls `scds_svc_wait()` with a timeout value of 5. The `scds_svc_wait()` method limits the frequency of the probes to every five seconds. This method also counts the number of attempts to start the service. If the number of attempts exceeds the value of the `Retry_count` property of the resource within the period that is specified by the `Retry_interval` property of the resource, the `scds_svc_wait()` function returns failure. In this case, the `svc_start()` function also returns failure.

```
#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 {
    /*
    * probe the data service on the IP address of the
    * network resource and the portname
    */
    rc = svc_probe(scds_handle,
        netaddr->netaddrs[0].hostname,
        netaddr->netaddrs[0].port_proto.port, probe_timeout);
    if (rc == SCHA_ERR_NOERR) {
        /* Success. Free up resources and return */
        scds_free_netaddr_list(netaddr);
        return (0);
    }

    /* Call scds_svc_wait() so that if service fails too
    if (scds_svc_wait(scds_handle, SVC_WAIT_TIME)
        != SCHA_ERR_NOERR) {
        scds_syslog(LOG_ERR, "Service failed to start.");
        return (1);
    }

    /* Rely on RGM to timeout and terminate the program */
} while (1);
```

Note - Before it exits, the `xfnts_start` method calls `scds_close()` to reclaim resources that are allocated by `scds_initialize()`. [“`scds_initialize\(\)` Function” on page 146](#) and the [`scds_close\(3HA\)` man page](#) contain more information.

xfnts_stop Method

Because the `xfnts_start` method uses `scds_pmf_start()` to start the service under the PMF, `xfnts_stop` uses `scds_pmf_stop()` to stop the service.

Note - The first call in `xfnts_stop` is to `scds_initialize()`, which performs some necessary *housekeeping* functions. “[scds_initialize\(\) Function](#)” on page 146 and the [scds_initialize\(3HA\)](#) man page contain more information.

The `xfnts_stop` method calls the `svc_stop()` method, which is defined in the `xfnts.c` file, as follows:

```
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); /* Successfully stopped */
```

Note the following points about the call in `svc_stop()` to the `scds_pmf_stop()` function:

- The `SCDS_PMF_TYPE_SVC` argument identifies the program to stop as a data service application. This method can also stop a fault monitor or some other type of application.
- The `SCDS_PMF_SINGLE_INSTANCE` argument identifies the signal.
- The `SIGTERM` argument identifies the signal to use to stop the resource instance. If this signal fails to stop the instance, `scds_pmf_stop()` sends `SIGKILL` to stop the instance, and if that fails, returns with a timeout error. See the [scds_pmf_stop\(3HA\)](#) man page for details.
- The timeout value is that of the `Stop_timeout` property of the resource.

Note - Before it exits, the `xfnts_stop` method calls `scds_close()` to reclaim resources that are allocated by `scds_initialize()`. “[scds_initialize\(\) Function](#)” on page 146 and the [scds_close\(3HA\)](#) man page contain more information.

xfnts_monitor_start Method

The RGM calls the `Monitor_start` method on a node to start the fault monitor after a resource is started on the node. The `xfnts_monitor_start` method uses `scds_pmf_start()` to start the monitor daemon under the PMF.

Note - The first call in `xfnts_monitor_start` is to `scds_initialize()`, which performs some necessary *housekeeping* functions. “[scds_initialize\(\) Function](#)” on page 146 and the [scds_initialize\(3HA\)](#) man page contain more information.

The `xfnts_monitor_start` method calls the `mon_start` method, which is defined in the `xfnts.c` file, as follows:

```
scds_syslog_debug(DBG_LEVEL_HIGH,
"Calling Monitor_start method for resource <%s>.",
scds_get_resource_name(scds_handle));

/* Call scds_pmf_start and pass the name of the probe. */
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); /* Successfully started Monitor */
}
```

Note the following points about the call in `svc_mon_start()` to the `scds_pmf_start()` function:

- The `SCDS_PMF_TYPE_MON` argument identifies the program to start as a fault monitor. This method can also start a data service or some other type of application.
- The `SCDS_PMF_SINGLE_INSTANCE` argument identifies this as a single-instance resource.
- The `xfnts_probe` argument identifies the monitor daemon to start. The monitor daemon is assumed to be located in the same directory as the other callback programs.
- The final argument, `0`, specifies the child monitoring level. In this case, this value specifies that the PMF monitor the monitor daemon only.

Note - Before it exits, the `xfnts_monitor_start` method calls `scds_close()` to reclaim resources that were allocated by `scds_initialize()`. “[scds_initialize\(\) Function](#)” on page 146 and the [scds_close\(3HA\)](#) man page contain more information.

xfnts_monitor_stop Method

Because the `xfnts_monitor_start` method uses `scds_pmf_start()` to start the monitor daemon under the PMF, `xfnts_monitor_stop` uses `scds_pmf_stop()` to stop the monitor daemon.

Note - The first call in `xfnts_monitor_stop` is to `scds_initialize()`, which performs some necessary *housekeeping* functions. “[scds_initialize\(\) Function](#)” on page 146 and the [scds_initialize\(3HA\)](#) man page contain more information.

The `xfnts_monitor_stop()` method calls the `mon_stop` method, which is defined in the `xfnts.c` file, as follows:

```
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); /* Successfully stopped monitor */
}
```

Note the following points about the call in `svc_mon_stop()` to the `scds_pmf_stop()` function:

- The `SCDS_PMF_TYPE_MON` argument identifies the program to stop as a fault monitor. This method can also stop a data service or some other type of application.
- The `SCDS_PMF_SINGLE_INSTANCE` argument identifies this as a single-instance resource.

- The SIGKILL argument identifies the signal to use to stop the resource instance. If this signal fails to stop the instance, `scds_pmf_stop()` returns with a timeout error. See the [scds_pmf_stop\(3HA\)](#) man page for details.
- The timeout value is that of the `Monitor_stop_timeout` property of the resource.

Note - Before it exits, the `xfnts_monitor_stop` method calls `scds_close()` to reclaim resources that were allocated by `scds_initialize()`. “[scds_initialize\(\) Function](#)” on page 146 and the [scds_close\(3HA\)](#) man page contain more information.

xfnts_monitor_check Method

The RGM calls the `Monitor_check` method whenever the fault monitor attempts to fail over the resource group that contains the resource to another node. The `xfnts_monitor_check` method calls the `svc_validate()` method to verify that a correct configuration is in place to support the `xfns` daemon. See “[xfnts_validate Method](#)” on page 160 for details. The code for `xfnts_monitor_check` is as follows:

```
/* Process the arguments passed by RGM and initialize syslog */
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);

/* Free up all the memory allocated by scds_initialize */
scds_close(&scds_handle);

/* Return the result of validate method run as part of monitor check */
return (rc);
}
```

ORCL.xfnts Fault Monitor

The RGM does not directly call the `PROBE` method, but rather calls the `Monitor_start` method to start the monitor after a resource is started on a node. The `xfnts_monitor_start` method

starts the fault monitor under the control of the PMF. The `xfnts_monitor_stop` method stops the fault monitor.

The `ORCL.xfnts` fault monitor performs the following operations:

- Periodically monitors the health of the `xfns` server daemon by using utilities that are specifically designed to check simple TCP-based services, such as `xfns`.
- Tracks problems that the application encounters within a time window (using the `Retry_count` and `Retry_interval` properties) and decides whether to restart or fail over the data service if the application fails completely. The `scds_fm_action()` and `scds_fm_sleep()` functions provide built-in support for this tracking and decision mechanism.
- Implements the failover or restart decision by using `scds_fm_action()`.
- Updates the resource state and makes the resource state available to administrative tools and GUIs.

`xfnts_probe` Main Loop

The `xfnts_probe` method implements a loop.

Before implementing the loop, `xfnts_probe` performs the following operations:

- Retrieves the network address resources for the `xfnts` resource, as follows:

```
/* Get the ip addresses available for this resource */
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);
}
```

```
/* Return an error if there are no network resources */
if (netaddr == NULL || netaddr->num_netaddrs == 0) {
scds_syslog(LOG_ERR,
"No network address resource in resource group.");
return (1);
}
```

- Calls `scds_fm_sleep()` and passes the value of `Thorough_probe_interval` as the timeout value. The probe sleeps for the value of `Thorough_probe_interval` between probes, as follows:

```
timeout = scds_get_ext_probe_timeout(scds_handle);
```

```
for (;;) {
/*
 * sleep for a duration of thorough_probe_interval between
 * successive probes.
 */
(void) scds_fm_sleep(scds_handle,
scds_get_rs_thorough_probe_interval(scds_handle));
```

The `xfnts_probe` method implements the following loop:

```
for (ip = 0; ip < netaddr->num_netaddrs; ip++) {
/*
 * Grab the hostname and port on which the
 * health has to be monitored.
 */
hostname = netaddr->netaddrs[ip].hostname;
port = netaddr->netaddrs[ip].port_proto.port;
/*
 * HA-XFS supports only one port and
 * hence obtain the port value from the
 * first entry in the array of ports.
 */
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);

/*
 * Update service probe history,
 * take action if necessary.
 * Latch probe end time.
 */
ht2 = gethrtime();

/* Convert to milliseconds */
dt = (ulong_t)((ht2 - ht1) / 1e6);

/*
 * Compute failure history and take
 * action if needed
 */
(void) scds_fm_action(scds_handle,
probe_result, (long)dt);
} /* Each net resource */
} /* Keep probing forever */
```

The `svc_probe()` function implements the probe logic. The return value from `svc_probe()` is passed to `scds_fm_action()`, which determines whether to restart the application, fail over the resource group, or do nothing.

svc_probe() Function

The `svc_probe()` function makes a simple socket connection to the specified port by calling `scds_fm_tcp_connect()`. If the connect fails, `svc_probe()` returns a value of `100`, which indicates a complete failure. If the connect succeeds, but the disconnect fails, `svc_probe()` returns a value of `50`, which indicates a partial failure. If the connect and disconnect both succeed, `svc_probe()` returns a value of `0`, which indicates success.

The code for `svc_probe()` is as follows:

```
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;

/*
* probe the data service by doing a socket connection to the port
* specified in the port_list property to the host that is
* serving the XFS data service. If the XFS service which is configured
* to listen on the specified port, replies to the connection, then
* the probe is successful. Else we will wait for a time period set
* in probe_timeout property before concluding that the probe failed.
*/

/*
* Use the SVC_CONNECT_TIMEOUT_PCT percentage of timeout
* to connect to the port
*/
connect_timeout = (SVC_CONNECT_TIMEOUT_PCT * timeout)/100;
t1 = (hrtime_t)(gethrtime()/1E9);

/*
* the probe makes a connection to the specified hostname and port.
* The connection is timed for 95% of the actual probe_timeout.
*/
```

```
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));
/* this is a complete failure */
return (SCDS_PROBE_COMPLETE_FAILURE);
}

t2 = (hrtime_t)(gethrtime()/1E9);

/*
* Compute the actual time it took to connect. This should be less than
* or equal to connect_timeout, the time allocated to connect.
* If the connect uses all the time that is allocated for it,
* then the remaining value from the probe_timeout that is passed to
* this function will be used as disconnect timeout. Otherwise, the
* the remaining time from the connect call will also be added to
* the disconnect timeout.
*
*/

time_used = (int)(t2 - t1);

/*
* Use the remaining time(timeout - time_took_to_connect) to disconnect
*/

time_remaining = timeout - (int)time_used;

/*
* If all the time is used up, use a small hardcoded timeout
* to still try to disconnect. This will avoid the fd leak.
*/
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;
}
```

```

/*
 * Return partial failure in case of disconnection failure.
 * Reason: The connect call is successful, which means
 * the application is alive. A disconnection failure
 * could happen due to a hung application or heavy load.
 * If it is the later case, don't declare the application
 * as dead by returning complete failure. Instead, declare
 * it as partial failure. If this situation persists, the
 * disconnect call will fail again and the application will be
 * restarted.
 */
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));
    /* this is a partial failure */
    return (SCDS_PROBE_COMPLETE_FAILURE/2);
}

t2 = (hrtime_t)(gethrtime()/1E9);
time_used = (int)(t2 - t1);
time_remaining = timeout - time_used;

/*
 * If there is no time left, don't do the full test with
 * fsinfo. Return SCDS_PROBE_COMPLETE_FAILURE/2
 * instead. This will make sure that if this timeout
 * persists, server will be restarted.
 */
if (time_remaining <= 0) {
    scds_syslog(LOG_ERR, "Probe timed out.");
    return (SCDS_PROBE_COMPLETE_FAILURE/2);
}

/*
 * The connection and disconnection to port is successful,
 * Run the fsinfo command to perform a full check of
 * server health.
 * Redirect stdout, otherwise the output from fsinfo
 * ends up on the console.
 */
(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);  
}
```

When finished, `svc_probe()` returns a value that indicates success (0), partial failure (50), or complete failure (100). The `xfnts_probe` method passes this value to `scds_fm_action()`.

Determining the Fault Monitor Action

The `xfnts_probe` method calls `scds_fm_action()` to determine the action to take.

The logic in `scds_fm_action()` is as follows:

- Maintain a cumulative failure history within the value of the `Retry_interval` property.
- If the cumulative failure reaches 100 (complete failure), restart the data service. If `Retry_interval` is exceeded, reset the history.
- If the number of restarts exceeds the value of the `Retry_count` property, within the time specified by `Retry_interval`, fail over the data service.

For example, suppose the probe makes a connection to the xfs server, but fails to disconnect. This indicates that the server is running, but could be hung or just under a temporary load. The failure to disconnect sends a partial (50) failure to `scds_fm_action()`. This value is below the threshold for restarting the data service, but the value is maintained in the failure history.

If during the next probe the server again fails to disconnect, a value of 50 is added to the failure history maintained by `scds_fm_action()`. The cumulative failure value is now 100, so `scds_fm_action()` restarts the data service.

xfnts_validate Method

The RGM calls the `Validate` method when a resource is created and when a cluster administrator updates the properties of the resource or its containing group. The RGM calls the `Validate` method before the creation or update is applied. A failure exit code from the method on any node causes the creation or update to be canceled.

The RGM calls `Validate` only when a cluster administrator changes resource or resource group properties or when a monitor sets the `Status` and `Status_msg` resource properties. The RGM does not call `Validate` when the RGM sets properties.

Note - The `Monitor_check` method also explicitly calls the `Validate` method whenever the `PROBE` method attempts to fail over the data service to a new node.

The RGM calls `Validate` with additional arguments to those that are passed to other methods, including the properties and values that are being updated. The call to `scds_initialize()` at the beginning of `xfnts_validate` parses all the arguments that the RGM passes to `xfnts_validate` and stores the information in the `scds_handle` argument. The subroutines that `xfnts_validate` calls make use of this information.

The `xfnts_validate` method calls `svc_validate()`, which verifies the following conditions:

- The `Confdir_list` property has been set for the resource and defines a single directory.

```
scha_str_array_t *confdirs;
confdirs = scds_get_ext_confdir_list(scds_handle);

/* Return error if there is no confdir_list extension property */
if (confdirs == NULL || confdirs->array_cnt != 1) {
scds_syslog(LOG_ERR,
"Property Confdir_list is not set properly.");
return (1); /* Validation failure */
}
```

- The directory that is specified by `Confdir_list` contains the `fontserver.cfg` file.

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

if (stat(xfnts_conf, &statbuf) != 0) {
/*
* suppress lint error because errno.h prototype
* is missing void arg
*/
scds_syslog(LOG_ERR,
"Failed to access file <%s> : <%s>",
xfnts_conf, strerror(errno)); /*lint !e746 */
return (1);
}
```

- The server daemon binary is accessible on the cluster node.

```
if (stat("/usr/bin/xfns", &statbuf) != 0) {
scds_syslog(LOG_ERR,
```

```
"Cannot access XFS binary : <%s> ", strerror(errno));
return (1);
}
```

- The Port_list property specifies a single port.

```
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); /* Validation Failure */
}
```

```
#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); /* Validation Failure */
}
#endif
```

- The resource group that contains the data service also contains at least one network address resource.

```
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); /* Validation Failure */
}
```

```
/* Return an error if there are no network address resources */
if (snrlp == NULL || snrlp->num_netresources == 0) {
scds_syslog(LOG_ERR,
"No network address resource in resource group.");
rc = 1;
goto finished;
}
```

Before it returns, svc_validate() frees all allocated resources.

```
finished:
scds_free_net_list(snrlp);
scds_free_port_list(portlist);

return (rc); /* return result of validation */
```

Note - Before it exits, the `xfnts_validate` method calls `scds_close()` to reclaim resources that were allocated by `scds_initialize()`. “[scds_initialize\(\) Function](#)” on page 146 and the [scds_close\(3HA\)](#) man page contain more information.

xfnts_update Method

The RGM calls the Update method to notify a running resource that its properties have changed. The only properties that can be changed for the `xfnts` data service pertain to the fault monitor. Therefore, whenever a property is updated, the `xfnts_update` method calls `scds_pmf_restart_fm()` to restart the fault monitor.

```
/* check if the Fault monitor is already running and if so stop
 * and restart it. The second parameter to scds_pmf_restart_fm()
 * uniquely identifies the instance of the fault monitor that needs
 * to be restarted.
 */
```

```
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.");
/* Free up all the memory allocated by scds_initialize */
scds_close(&scds_handle);
return (1);
}
```

```
scds_syslog(LOG_INFO,
"Completed successfully.");
```

Note - The second argument to `scds_pmf_restart_fm()` uniquely identifies the instance of the fault monitor to be restarted if there are multiple instances. The value `0` in the example indicates that there is only one instance of the fault monitor.

Oracle Solaris Cluster Agent Builder

This chapter describes Oracle Solaris Cluster Agent Builder, which automates the creation of resource types, or data services, to be run under the control of the Resource Group Manager (RGM). A *resource type* is a wrapper around an application that enables that application to run in a clustered environment, under control of the RGM.

This chapter covers the following topics:

- [“Agent Builder Overview” on page 165](#)
- [“Before You Use Agent Builder” on page 166](#)
- [“Using Agent Builder” on page 167](#)
- [“Directory Structure That Agent Builder Creates” on page 180](#)
- [“Agent Builder Output” on page 181](#)

Agent Builder Overview

Agent Builder provides a graphical user interface (GUI) for specifying information about your application and the kind of resource type that you want to create. Agent Builder supports network-aware applications and non-network-aware applications. *Network-aware applications* use the network to communicate with clients. *Non-network-aware applications* are standalone applications.

Note - If the GUI version of Agent Builder is not accessible, you can access Agent Builder through a command-line interface. See [“How to Use the Command-Line Version of Agent Builder” on page 180](#).

Based on the information you specify, Agent Builder generates the following software:

- A set of C, Korn shell (ksh), or generic data service (GDS) source files for a failover or scalable resource type that corresponds to the resource type's method callbacks. These files are intended for both network-aware (client-server model) and non-network-aware (clientless) applications.

- A customized Resource Type Registration (RTR) file (if you generate C or Korn shell source code).
- Customized utility scripts for starting, stopping, and removing an instance (resource) of the resource type, as well as customized man pages that document how to use each one of these files.
- An Oracle Solaris SVR4 package that includes the binaries (if you generate C source code), an RTR file (if you generate C or Korn shell source code), and the utility scripts.

Agent Builder also enables you to generate a resource type for an application that has multiple independent process trees that the Process Monitor Facility (PMF) must monitor and restart individually.

Before You Use Agent Builder

Before you use Agent Builder, you need to know how to create resource types with multiple independent process trees.

Agent Builder can create resource types for applications that have more than one independent process tree. These process trees are independent in the sense that the PMF monitors and starts them individually. The PMF starts each process tree with its own tag.

Note - Agent Builder enables you to create resource types with multiple independent process trees only if the generated source code that you specify is C or GDS. You cannot use Agent Builder to create these resource types for the Korn shell. To create these resource types for the Korn shell, you must manually write the code.

In the case of a base application with multiple independent process trees, you cannot specify a single command line to start the application. Rather, you must create a text file, with each line specifying the full path to a command to start one of the application's process trees. This file must not contain any empty lines. You specify this text file in the Start Command text field on the Agent Builder Configure screen.

Ensuring that this file does not have execute permissions enables Agent Builder to distinguish this file. The purpose of this file is to start multiple process trees from a simple executable script that contains multiple commands. If this text file is given execute permissions, the resources come up with no problems or errors on a cluster. However, all the commands are started under one PMF tag. As a result, the PMF is unable to monitor and restart the process trees individually.

Using Agent Builder

This section describes how to use Agent Builder. In addition, this section includes tasks that you must complete before you can use Agent Builder. This section also explains ways that you can take advantage of Agent Builder after you generate your resource type code.

This section covers the following topics:

- [“Analyzing the Application” on page 167](#)
- [“Installing and Configuring Agent Builder” on page 168](#)
- [“Agent Builder Screens” on page 168](#)
- [“Starting Agent Builder” on page 169](#)
- [“Navigating Agent Builder” on page 169](#)
- [“Using the Create Screen” on page 172](#)
- [“Using the Configure Screen” on page 173](#)
- [“Using the Agent Builder Korn Shell-Based `\$hostnames` Variable” on page 175](#)
- [“Using Property Variables” on page 176](#)
- [“Reusing Code That You Create With Agent Builder” on page 178](#)
- [“How to Use the Command-Line Version of Agent Builder” on page 180](#)

Analyzing the Application

Before using Agent Builder, you must determine whether the application that you intend to make highly available or scalable meets the required criteria. Agent Builder cannot perform this analysis, which is based solely on the runtime characteristics of the application. [“Analyzing the Application for Suitability” on page 25](#) provides more information about this topic.

Agent Builder might not always be able to create a complete resource type for your application. However, in most cases, Agent Builder provides at least a partial solution. For example, more sophisticated applications might require additional code that Agent Builder does not generate by default. Examples of additional code include code that adds validation checks for additional properties or that tunes parameters that Agent Builder does not expose. In these cases, you must make changes to the generated source code or to the RTR file. Agent Builder is designed to provide just this kind of flexibility.

Agent Builder places comments at particular points in the generated source code where you can add your own resource type code. After making changes to the source code, you can use the makefile that Agent Builder generates to recompile the source code and regenerate the resource type package.

Even if you write your entire resource type code without using any code that is generated by Agent Builder, you can use the makefile and structure that Agent Builder provides to create the Oracle Solaris package for your resource type.

Installing and Configuring Agent Builder

Agent Builder requires no special installation. Agent Builder is included in the `ha-cluster/developer/api` IPS package, which is installed by default when you install the Oracle Solaris Cluster software. The [Oracle Solaris Cluster 4.3 Software Installation Guide](#) contains more information.

Before you use Agent Builder, verify the following requirements:

- The Java runtime environment is included in your `$PATH` variable.
- The `cc` compiler is included in your `$PATH` variable. Agent Builder uses the first occurrence of `cc` in your `$PATH` variable to identify the compiler with which to generate C binary code for the resource type. If `cc` is not included in `$PATH`, Agent Builder disables the option to generate C code. See [“Using the Create Screen” on page 172](#).

Note - You can use a different compiler with Agent Builder than the standard `cc` compiler. To use a different compiler, create a symbolic link in `$PATH` from `cc` to a different compiler, such as `gcc`. Or, change the compiler specification in the makefile (currently, `CC=cc`) to the complete path for a different compiler. For example, in the makefile that is generated by Agent Builder, change `CC=cc` to `CC=pathname/gcc`. In this case, you cannot run Agent Builder directly. Instead, you must use the `make` and `make pkg` commands to generate data service code and the package.

Agent Builder Screens

Agent Builder is a two-step wizard with a corresponding screen for each step.

Agent Builder provides the following two screens to guide you through the process of creating a new resource type:

1. **Create screen.** On this screen, you provide basic information about the resource type to create, such as its name and the working directory for the generated files. The working directory is where you create and configure the resource type template.

You also specify the following information:

- The kind of resource to create (scalable or failover)
- Whether the base application is network aware (that is, if it uses the network to communicate with its clients)

- The type of code to generate (C, Korn shell (ksh), or GDS)

For information about GDS, see [Oracle Solaris Cluster Generic Data Service \(GDS\) Guide](#). You must provide all the information on this screen and select Create to generate the corresponding output. Then, you can display the Configure screen.

2. **Configure screen.** On this screen, you must specify the full command line that can be passed to any UNIX shell to start your base application. Optionally, you can provide commands to stop and to probe your application. If you do not specify these two commands, the generated output uses signals to stop the application and provides a default probe mechanism. See the description of the probe command in [“Using the Configure Screen” on page 173](#). The Configure screen also enables you to change the timeout values for each of these three commands: start, stop, probe.

Starting Agent Builder

Note - If the GUI version of Agent Builder is not accessible, you can access Agent Builder through a command-line interface. See [“How to Use the Command-Line Version of Agent Builder” on page 180](#).

If you start Agent Builder from the working directory for an existing resource type, Agent Builder initializes the Create and Configure screens to the values of the existing resource type.

Start Agent Builder by typing the following command:

```
% /usr/cluster/bin/scdsbuilder
```

The Create screen appears.

Navigating Agent Builder

You enter information on the Create and Configure screens by performing the following operations:

- Typing information in a field
- Browsing your directory structure and selecting a file or directory
- Selecting one of a set of mutually exclusive radio buttons, for example, selecting Scalable or Failover
- Selecting the Network Aware check box to identify the base application as network aware, or leaving this box empty to identify a non-network-aware application

The buttons at the bottom of each screen enable you to complete the task, move to the next or previous screen, or exit Agent Builder. Agent Builder emphasizes or grays out these buttons, as necessary.

For example, when you have filled in the fields and selected the preferred options on the Create screen, click Create at the bottom of the screen. Previous and Next are grayed out because no previous screen exists and you cannot go to the next step before you complete this step.



Agent Builder displays progress messages in the Output Log area at the bottom of the screen. When Agent Builder finishes, it displays a success message or a warning message. Next is highlighted, or if this is the last screen, only Cancel is highlighted.

You can click Cancel at any time to exit Agent Builder.

Browse Command


Some Agent Builder fields enable you to type information in them. Other fields enable you to click Browse to browse a directory structure and select a file or a directory.




When you click Browse, a screen appears that lists the files and directory folders.

Double-click a folder to open it. When you move the cursor to a file, the file's name appears in the File Name field. Click Select when you have located and moved the cursor to the file that you want.

Note - If you are browsing for a directory, move the cursor to the directory that you want and click Open. If the directory contains no subdirectories, Agent Builder closes the browse window and places the name of the directory to which you moved the cursor in the appropriate field. If this directory has subdirectories, click Close to close the browse window and redisplay the previous screen. Agent Builder places the name of the directory to which you moved the cursor in the appropriate field.

The icons in the upper right corner of the Browse screen do the following:

Icon	Purpose
	This icon moves you up one level in the directory tree.

Icon	Purpose
	This icon returns you to the home folder.
	This icon creates a new folder under the currently selected folder.
	This icon, for toggling between different views, is reserved for future use.

Agent Builder Menus

Agent Builder provides File and Edit drop-down menus.

Agent Builder File Menu

The File menu contains two options:

- **Load Resource Type.** Loads an existing resource type. Agent Builder provides a browse screen from which you select the working directory for an existing resource type. If a resource type exists in the directory from which you start Agent Builder, Agent Builder automatically loads the resource type. Load Resource Type enables you to start Agent Builder from any directory and select an existing resource type to use as a template for creating a new resource type. See [“Reusing Code That You Create With Agent Builder” on page 178](#).
- **Exit.** Exits Agent Builder. You can also exit by clicking Cancel on the Create or the Configure screen.

Agent Builder Edit Menu

The Edit menu contains two options:

- **Clear Output Log.** Clears the information from the output log. Each time you select Create or Configure, Agent Builder appends status messages to the output log. If you are iteratively making changes to your source code and regenerating output in Agent Builder and want to segregate the status messages, you can save and clear the log file before each use.
- **Save Log File.** Saves the log output to a file. Agent Builder provides a browse screen that enables you to select the directory and specify a file name.

Using the Create Screen

The first step in creating a resource type is to complete the Create screen, which appears when you start Agent Builder.

The Create screen contains the following fields, radio buttons, and check box:

- **Vendor Name.** A name that identifies the vendor of the resource type. Typically, you specify the stock symbol of the vendor. However, any name that uniquely identifies the vendor is valid. Use alphanumeric characters only.
- **Application Name.** The name of the resource type. Use alphanumeric characters only.

Note - Together, the vendor name and application name make up the full name of the resource type.

- **RT Version.** The version of the generated resource's type. The RT Version distinguishes between multiple registered versions, or upgrades, of the same base resource type.

You cannot use the following characters in the RT Version field:

- Space
- Tab
- Slash (/)
- Backslash (\)
- Asterisk (*)
- Question mark (?)
- Comma (,)
- Semicolon (;)
- Left square bracket ([)
- Right square bracket (])
- **Working Directory.** The directory under which Agent Builder creates a directory structure to contain all the files that are created for the target resource type. You can create only one resource type in any one working directory. Agent Builder initializes this field to the path of the directory from which you started Agent Builder. However, you can type a different name or use Browse to locate a different directory.

Under the working directory, Agent Builder creates a subdirectory with the resource type name. For example, if ORCL is the vendor name and ftp is the application name, Agent Builder names this subdirectory ORCLftp.

Agent Builder places all the directories and files for the target resource type under this subdirectory. See [“Directory Structure That Agent Builder Creates” on page 180](#).

- **Scalable or Failover.** Specify whether the target resource type is failover or scalable.

- **Network Aware.** Specify whether the base application is network aware, that is, if it uses the network to communicate with its clients. Select the Network Aware check box to specify network aware, or do not select the check box to specify non-network aware.
- **C, ksh.** Specify the language of the generated source code. Although these options are mutually exclusive, with Agent Builder you can create a resource type with Korn shell-generated code and reuse the same information to create C generated code. See [“Reusing Code That You Create With Agent Builder” on page 178](#).
- **GDS.** Specify that this service is a generic data service. [Oracle Solaris Cluster Generic Data Service \(GDS\) Guide](#) contains more detailed information about creating and configuring a generic data service.

Note - If the `cc` compiler is not in your `$PATH` variable, Agent Builder grays out the C radio button and allows you to select the ksh radio button. To specify a different compiler, see the note at the end of [“Installing and Configuring Agent Builder” on page 168](#).

After you have specified the required information, click Create. The Output Log area at the bottom of the screen shows the actions that Agent Builder performs. You can choose Save Output Log from the Edit menu to save the information in the output log.

When finished, Agent Builder displays either a success message or a warning message.

- If Agent Builder was unable to complete this step, examine the output log for details.
- If Agent Builder completes successfully, click Next to display the Configure screen. The Configure screen enables you to finish generating the resource type.

Note - Although generation of a complete resource type is a two-step process, you can exit Agent Builder after completing the first step (create) without losing the information that you have specified or the work that Agent Builder has completed. See [“Reusing Code That You Create With Agent Builder” on page 178](#).

Using the Configure Screen

The Configure screen appears after Agent Builder finishes creating the resource type and you click Next on the Create screen. You cannot access the Configure screen before the resource type has been created.

The Configure screen contains the following fields:

- **Start Command.** The complete command line that can be passed to any UNIX shell to start the base application. You must specify a start command. You can type the command in the field provided, or use Browse to locate a file that contains the command to start the application.

The complete command line must include everything necessary to start the application, such as host names, port numbers, a path to configuration files. You can also specify property variables, which are described in [“Using Property Variables” on page 176](#). If your Korn shell-based application requires a host name to be specified on the command line, you can use the `$hostnames` variable that Agent Builder defines. See [“Using the Agent Builder Korn Shell-Based `\$hostnames` Variable” on page 175](#).

Do not enclose the command in double quotation marks (“”).

Note - If the base application has multiple independent process trees, each of which is started with its own tag under Process Monitor Facility (PMF) control, you cannot specify a single command. Rather, you must create a text file that contains individual commands to start each process tree, and specify the path to this file in the Start Command text field. See [“Before You Use Agent Builder” on page 166](#). This section lists some special characteristics that this file requires to work correctly.

- **Stop Command.** The complete command line that can be passed to any UNIX shell to stop the base application. You can type the command in the field provided, or use Browse to locate a file that contains the command to stop the application. You can also specify property variables, which are described in [“Using Property Variables” on page 176](#). If your Korn shell-based application requires a host name to be specified on the command line, you can use the `$hostnames` variable that Agent Builder defines. See [“Using the Agent Builder Korn Shell-Based `\$hostnames` Variable” on page 175](#).

This command is optional.

If you do not specify a stop command, the generated code uses signals (in the Stop method) to stop the application, as follows:

- The Stop method sends SIGTERM to stop the application and waits for 80 percent of the timeout value for the application to exit.
- If the SIGTERM signal is unsuccessful, the Stop method sends SIGKILL to stop the application and waits for 15 percent of the timeout value for the application to exit.
- If SIGKILL is unsuccessful, the Stop method exits unsuccessfully. The remaining 5 percent of the timeout value is considered overhead.



Caution - Be certain the stop command does not return before the application has stopped completely.

- **Probe Command.** A command that can be run periodically to check the health of the application and return an exit status between 0 (success) and 100 (complete failure). This command is optional. You can type the complete path to the command, or use Browse to locate a file that contains the commands to probe the application.

Typically, you specify a simple client of the base application. If you do not specify a probe command, the generated code simply connects to and disconnects from the port that is used by the resource. If the connect and disconnect succeed, the generated code declares the application healthy. You can also specify property variables, which are described in [“Using Property Variables” on page 176](#). If your Korn shell-based application requires that you specify a host name on the probe command line, you can use the `$hostnames` variable that Agent Builder defines. See [“Using the Agent Builder Korn Shell-Based `\$hostnames` Variable” on page 175](#).

Do not enclose the command in double quotation marks (“”).

- **Timeout.** A timeout value, in seconds, for each command. You can specify a new value, or accept the default value that Agent Builder provides. The default value is 300 seconds for start and stop and 30 seconds for probe.

Using the Agent Builder Korn Shell-Based `$hostnames` Variable

For many applications, specifically network-aware applications, the host name on which the application listens and services customer requests must be passed to the application on the command line. In many cases, the host name is an argument that you must specify for start, stop, and probe commands for the target resource type on the Configure screen. However, the host name on which an application listens is cluster specific. The host name is determined when the resource is run on a cluster. The host name cannot be determined when Agent Builder generates your resource type code.

To solve this problem, Agent Builder provides the `$hostnames` variable that you can specify on the command line for the start, stop, and probe commands.

Note - The `$hostnames` variable is supported for use with Korn shell-based services only. The `$hostnames` variable is not supported for use with C-based and GDS-based services.

You specify the `$hostnames` variable exactly as you would an actual host name, for example:

```
% /opt/network_aware/echo_server -p port-no -l $hostnames
```

When a resource of the target resource type is run on a cluster, the `LogicalHostname` or `SharedAddress` host name that is configured for that resource is substituted for the value of the `$hostnames` variable. The host name is configured for that resource in the `Network_resources_used` resource property of the resource.

If you configure the `Network_resources_used` property with multiple host names, the `$hostnames` variable contains all host names, each host name separated by a comma.

Using Property Variables

You can also retrieve the values of selected Oracle Solaris Cluster resource type, resource, and resource group properties from the RGM framework by using property variables. Agent Builder scans your start, probe, or stop command strings for property variables and substitutes these variables with their values before Agent Builder executes the command.

Note - Property variables are not supported for use with Korn shell-based services.

List of Property Variables

This section lists the property variables that you can use. The Oracle Solaris Cluster resource type, resource, and resource group properties are described in [“Setting Standard Properties” on page 30](#).

Resource Property Variables

- HOSTNAMES
- RS_CHEAP_PROBE_INTERVAL
- RS_MONITOR_START_TIMEOUT
- RS_MONITOR_STOP_TIMEOUT
- RS_NAME
- RS_NUM_RESTARTS
- RS_RESOURCE_DEPENDENCIES
- RS_RESOURCE_DEPENDENCIES_WEAK
- RS_RETRY_COUNT
- RS_RETRY_INTRVAL
- RS_SCALABLE
- RS_START_TIMEOUT
- RS_STOP_TIMEOUT
- RS_THOROUGH_PROBE_INTERVAL
- SCHA_STATUS

Resource Type Property Variables

- RT_API_VERSION

- RT_BASEDIR
- RT_FAILOVER
- RT_INSTALLED_NODES
- RT_NAME
- RT_RT_VERSION
- RT_SINGLE_INSTANCE

Resource Group Property Variables

- RG_DESIRED_PRIMARIES
- RG_GLOBAL_RESOURCES_USED
- RG_IMPLICIT_NETWORK_DEPENDENCIES
- RG_MAXIMUM_PRIMARIES
- RG_NAME
- RG_NODELIST
- RG_NUM_RESTARTS
- RG_PATHPREFIX
- RG_PINGPONG_INTERVAL
- RG_RESOURCE_LIST

Syntax of Property Variables

You include a percent sign (%) before a property name to indicate a property variable, as shown in this example:

```
/opt/network_aware/echo_server -t %RS_STOP_TIMEOUT -n %RG_NODELIST
```

Given the preceding example, Agent Builder might interpret these property variables and start the `echo_server` script with the following values:

```
/opt/network_aware/echo_server -t 300 -n phys-node-1,phys-node-2,phys-node-3
```

How Agent Builder Substitutes Property Variables

Agent Builder interprets the types of property variables, as follows:

- An integer is substituted with its actual value (300, for example).

- A Boolean value is substituted with the string TRUE or FALSE.
- A string is substituted with the actual string (phys-node-1, for example).
- A list of strings is substituted with all members in the list, each string separated by a comma (phys-node-1,phys-node-2,phys-node-3, for example).
- A list of integers is substituted with all members in the list, each integer separated by a comma (1,2,3, for example).
- An enumerated type is substituted with its value, in string form.

Reusing Code That You Create With Agent Builder

Agent Builder enables you to reuse completed work in the following ways:

- You can clone an existing resource type that you created with Agent Builder.
- You can edit the source code that Agent Builder generates and recompile the code to create a new package.

▼ How to Clone an Existing Resource Type

Follow this procedure to clone an existing resource type that is generated by Agent Builder.

1. Load an existing resource type into Agent Builder.

Use one of the following methods:

- Start Agent Builder from the working directory for an existing resource type that you created with Agent Builder. Ensure that the working directory contains the `rtconfig` file. Agent Builder loads the values for that resource type in the Create and Configure screens.
- Use the Load Resource Type option from the File drop-down menu.

2. Change the working directory on the Create screen.

You must use Browse to select a directory. Typing a new directory name is not sufficient. After you select a directory, Agent Builder re-enables the Create button.

3. Make the changes that you want to the existing resource type.

You might change the type of code that is generated for the resource type.

For example, if you initially create a Korn shell version of a resource type but find over time that you require a C version, you can do the following:

- Load the existing Korn shell resource type.

- Change the language for the output to C.
 - Click Create to have Agent Builder build a C version of the resource type.
4. **Create the cloned resource type.**
 - a. **Click Create to create the resource type.**
 - b. **Click Next to display the Configure screen.**
 - c. **Click Configure to configure the resource type, and click Cancel to finish.**

Editing the Generated Source Code

To simplify the process of creating a resource type, Agent Builder limits the amount of information that you can specify, which necessarily limits the scope of the generated resource type. Therefore, to add more sophisticated features, you need to modify the generated source code or the RTR file. Examples of additional features include code that adds validation checks for additional properties or that tunes parameters that Agent Builder does not expose.

The source files are in the *install-directory/rt-name/src* directory. Agent Builder embeds comments in the source code where you can add code. These comments are of the form (for C code):

```
/* User added code -- BEGIN vvvvvvvvvvvvvvvvvv */
/* User added code -- END   ^^^^^^^^^^^^^^^^^^ */
```

Note - These comments are identical in Korn shell source code, except the comment mark (#) indicates the beginning of a comment.

For example, *rt-name.h* declares all the utility functions that the different programs use. At the end of the list of declarations are comments that enable you to declare additional functions that you might have added to your code.

Agent Builder also generates the makefile in the *install-directory/rt-name/src* directory with corresponding targets. Use the `make` command to recompile the source code. Use the `make pkg` command to regenerate the resource type package.

The RTR file is in the *install-directory/rt-name/etc* directory. You can edit the RTR file with a standard text editor. See [“Setting Resource and Resource Type Properties” on page 32](#) for more information about the RTR file. See [“Setting Standard Properties” on page 30](#) for information about properties.

▼ How to Use the Command-Line Version of Agent Builder

The command-line version of Agent Builder follows the same basic process as the GUI. However, instead of typing information in the GUI, you pass arguments to the `scdscreate` and `scdsconfig` commands. See the [scdscreate\(1HA\)](#) and [scdsconfig\(1HA\)](#) man pages for more information.

Follow these steps to use the command-line version of Agent Builder.

1. **Use `scdscreate` to create an Oracle Solaris Cluster resource type template for making an application highly available or scalable.**
2. **Use `scdsconfig` to configure the resource type template that you created with `scdscreate`.**
You can specify property variables. Property variables are described in “Using Property Variables” on page 176.
3. **Change directories to the `pkg` subdirectory in the working directory.**
4. **(Optional) Edit the generated source code.**
5. **On each node, use the `pkgadd` command to install the packages that you created with `scdscreate`.**

```
# pkgadd -d . package-name
```

Note - This instruction applies to the SVR4 package that Agent Builder creates. If you need an IPS version of the package, use the `pkgsend` command to convert your SVR4 agent package to an IPS package, and use the `pkg install` command to install the IPS package. For more information, see the [pkgsend\(1\)](#) and [pkg\(1\)](#) man pages.

6. **Run the start script.**

Directory Structure That Agent Builder Creates

Agent Builder creates a directory structure to hold all the files that it generates for the target resource type. You specify the working directory on the Create screen. You must specify

separate install directories for any additional resource types that you develop. Under the working directory, Agent Builder creates a subdirectory whose name is a concatenation of the vendor name and the resource type name. For example, if you specify ORCL as the vendor name and create a resource type called ftp, Agent Builder creates a directory called ORCLftp under the working directory.

Under this subdirectory, Agent Builder creates and populates the directories that are listed in the following table.

Directory Name	Contents
bin	For C output, contains the binary files that are compiled from the source files. For Korn shell output, contains the same files as the src directory.
etc	Contains the RTR file. Agent Builder concatenates the vendor name and application name, separated by a period (.), to form the RTR file name. For example, if the vendor name is ORCL and the name of the resource type is ftp, the name of the RTR file is ORCL.ftp.
man	Contains customized man pages for the start, stop, and remove utility scripts, for example, startftp(1M), stopftp(1M), and removeftp(1M). To view these man pages, specify the path with the man -M option. For example: % man -M install-directory/ORCLftp/man removeftp
pkg	Contains the final Oracle Solaris package that includes the created data service.
src	Contains the source files that Agent Builder generates.
util	Contains the start, stop, and remove utility scripts that Agent Builder generates. See “Utility Scripts and Man Pages That Agent Builder Creates” on page 183 . Agent Builder appends the application name to each of these script names, for example, startftp, stopftp, and removeftp.

Agent Builder Output

This section describes the output that Agent Builder generates.

This section covers the following topics:

- [“Source and Binary Files” on page 182](#)
- [“Utility Scripts and Man Pages That Agent Builder Creates” on page 183](#)
- [“Support Files That Agent Builder Creates” on page 184](#)
- [“Package Directory That Agent Builder Creates” on page 184](#)
- [“rtconfig File” on page 185](#)

Source and Binary Files

The Resource Group Manager (RGM) manages resource groups and ultimately resources on a cluster. The RGM works on a callback model. When specific events happen, such as a node failure, the RGM calls the resource type's methods for each of the resources that are running on the affected node. For example, the RGM calls the Stop method to stop a resource that is running on the affected node, and calls the resource's Start method to start the resource on a different node. See [“Resource Group Manager Model” on page 17](#), [“Callback Methods” on page 19](#), and the [rt_callbacks\(1HA\)](#) man page for more information about this model.

To support this model, Agent Builder generates eight executable C programs or Korn shell scripts in the *install-directory/rt-name/bin* directory. These programs or shell scripts serve as callback methods.

Note - Strictly speaking, the *rt-name_probe* program, which implements a fault monitor, is not a callback program. The RGM does not directly call *rt-name_probe*. Instead, the RGM calls *rt-name_monitor_start* and *rt-name_monitor_stop*. These methods start and stop the fault monitor by calling *rt-name_probe*.

Here are the eight methods that Agent Builder generates:

- *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*

See the [rt_callbacks\(1HA\)](#) man page for specific information about each method.

In the *install-directory/rt-name/src* directory (C output), Agent Builder generates the following files:

- A header file (*rt-name.h*)
- A source file (*rt-name.c*) that contains code that is common to all methods
- An object file (*rt-name.o*) for the common code
- Source files (**.c*) for each method

- Object files (*.o) for each method

Agent Builder links the *rt-name.o* file to each of the method .o files to create the executable files in the *install-directory/rt-name/bin* directory.

For Korn shell output, the *install-directory/rt-name/bin* and *install-directory/rt-name/src* directories are identical. Each directory contains the eight executable scripts that correspond to the seven callback methods and the Probe method.

Note - The Korn shell output includes two compiled utility programs, *gettime* and *gethostnames*. Particular callback methods require these methods for getting the time and for probing.

You can edit the source code, run the *make* command to recompile the code, and when you are finished, run the *make pkg* command to generate a new package. To support making changes to the source code, Agent Builder embeds comments in the source code at correct locations where you can add code. See [“Editing the Generated Source Code” on page 179](#).

Utility Scripts and Man Pages That Agent Builder Creates

Once you have generated a resource type and installed its package on a cluster, you must still get an instance (resource) of the resource type that is running on a cluster. Generally, to get an instance, you use administrative commands or Oracle Solaris Cluster Manager. However, as a convenience, Agent Builder generates a customized utility script for this purpose as well as scripts for stopping and removing a resource of the target resource type.

These three scripts, which are located in the *install-directory/rt-name/util* directory, do the following:

- **Start script.** Registers the resource type, and creates the necessary resource groups and resources. This script also creates the network address resource (*LogicalHostname* or *SharedAddress*) that enables the application to communicate with the clients on the network.
- **Stop script.** Stops the resource.
- **Remove script.** Undoes the work of the start script. That is, this script stops and removes the resources, resource groups, and the target resource type from the system.

Note - You can only use the remove script with a resource that was started by the corresponding start script because these scripts use internal conventions to name resources and resource groups.

Agent Builder names these scripts by appending the application name to the script names. For example, if the application name is `ftp`, the scripts are called `startftp`, `stopftp`, and `removeftp`.

Agent Builder provides man pages in the `install-directory/rt-name/man/man1m` directory for each utility script. You should read these man pages before you start these scripts because they document the arguments that you need to pass to the script.

To view these man pages, specify the path to this man directory by using the `-M` option with the `man` command. For example, if `ORCL` is the vendor and `ftp` is the application name, type the following command to view the `startftp(1M)` man page:

```
% man -M install-directory/ORCLftp/man startftp
```

The man page utility scripts are also available to the cluster administrator. When an Agent Builder-generated package is installed on a cluster, the man pages for the utility scripts are placed in the `/opt/rt-name/man` directory. For example, type the following command to view the `startftp(1M)` man page:

```
% man -M /opt/ORCLftp/man startftp
```

Support Files That Agent Builder Creates

Agent Builder places support files, such as `pkginfo`, `postinstall`, `postremove`, and `preremove`, in the `install-directory/rt-name/etc` directory. This directory also contains the resource type registration (RTR) file. The RTR file declares resource and resource type properties that are available for the target resource type and initializes property values at the time a resource is registered with a cluster. See [“Setting Resource and Resource Type Properties” on page 32](#) for more information. The RTR file is named as `vendor-name.resource-type-name`, for example, `ORCL.ftp`.

You can edit this file with a standard text editor and make changes without recompiling your source code. However, you must rebuild the package with the `make pkg` command.

Package Directory That Agent Builder Creates

The `install-directory/rt-name/pkg` directory contains an Oracle Solaris SVR4 package. The name of the package is a concatenation of the vendor name and the application name, for example, `ORCLftp`. The makefile in the `install-directory/rt-name/src` directory supports the creation of a new package. For example, if you make changes to the source files and recompile

the code, or you make changes to the package utility scripts, use the `make pkg` command to create a new package.

Note - If you need an IPS version of the package that Agent Builder creates, use the `pkgsend` command to convert your SVR4 agent package to an IPS package, and use the `pkg install` command to install the IPS package. For more information, see the [pkgsend\(1\)](#) and [pkg\(1\)](#) man pages.

When you remove a package from a cluster, the `pkgrm` (SVR4) or `pkg remove` (IPS) command can fail if you attempt to run the command simultaneously from more than one node.

You can solve this problem in one of two ways:

- Run the `remove rt-name` script from one node of the cluster before running the `pkgrm` command from any node.
- Run the `pkgrm` command from one node of the cluster, which takes care of all necessary cleanup operations. Then, run the `pkgrm` command from the remaining nodes, simultaneously if necessary.

If `pkgrm` fails because you attempt to run it simultaneously from multiple nodes, run the command again from one node. Then, run the command from the remaining nodes.

rtconfig File

If you generate C or Korn shell source code in the working directory, Agent Builder generates a configuration file called `rtconfig`. This file contains the information that you specified on the Create and Configure screens. If you start Agent Builder from the working directory for an existing resource type, Agent Builder reads the `rtconfig` file. Agent Builder fills in the Create and Configure screens with the information that you provided for the existing resource type. Agent Builder works similarly if you load an existing resource type by choosing Load Resource Type from the File drop-down menu. This feature is useful if you want to clone an existing resource type. See [“Reusing Code That You Create With Agent Builder” on page 178](#).

DSDL API Functions

This chapter lists and briefly describes the Data Service Development Library (DSDL) API functions. See the individual 3HA man pages for a complete description of each DSDL function. The DSDL provides a C interface only. A script-based DSDL interface is not available.

This chapter covers the following topics:

- “General-Purpose Functions” on page 187
- “Property Functions” on page 189
- “Network Resource Access Functions” on page 189
- “PMF Functions” on page 191
- “Fault Monitor Functions” on page 192
- “Utility Functions” on page 192

General-Purpose Functions

The functions in this section provide a broad range of functionality.

These functions enable you to perform the following operations:

- Initialize the DSDL environment
- Retrieve resource type, resource, and resource group names, and extension property values
- Fail over and restart a resource group, and restart a resource
- Convert error strings to error messages
- Execute a command under a timeout

Initialization Functions

The following functions initialize the calling method:

- `scds_initialize(3HA)` – Allocates resources and initializes the DSDL environment.

- [scds_close\(3HA\)](#) – Frees resources that are allocated by `scds_initialize()`.

Retrieval Functions

The following functions retrieve information about zones, resource types, resources, resource groups, and extension properties:

- [scds_get_current_method_name\(3HA\)](#) – Retrieves the last element of the path name by which a data service method was called.
- [scds_get_zone_name\(3HA\)](#) – Retrieves the name of the zone on whose behalf a method is running.
- [scds_get_resource_type_name\(3HA\)](#) – Retrieves the name of the resource type for the calling program.
- [scds_get_resource_name\(3HA\)](#) – Retrieves the name of the resource for the calling program.
- [scds_get_resource_group_name\(3HA\)](#) – Retrieves the name of the resource group for the calling program.
- [scds_get_ext_property\(3HA\)](#) – Retrieves the value of the specified extension property.
- [scds_free_ext_property\(3HA\)](#) – Frees the memory that is allocated by `scds_get_ext_property()`.

The following function retrieves status information about the `SUNW.HASStoragePlus` resources that are used by a resource:

[scds_hasp_check\(3HA\)](#) – Retrieves status information about `SUNW.HASStoragePlus` resources that are used by a resource. This information is obtained from the state (online or otherwise) of all `SUNW.HASStoragePlus` resources on which the resource depends by using the `Resource_dependencies` or `Resource_dependencies_weak` standard properties that are defined for the resource. See the [SUNW.HASStoragePlus\(5\)](#) man page for more information.

Failover and Restart Functions

The following functions fail over or restart a resource or resource group:

- [scds_failover_rg\(3HA\)](#) – Fails over a resource group.
- [scds_restart_rg\(3HA\)](#) – Restarts a resource group.
- [scds_restart_resource\(3HA\)](#) – Restarts a resource.

Execution Functions

The following functions execute a command under a timeout and convert an error code to an error message:

- [scds_timerun\(3HA\)](#) – Executes a command under a timeout value.
- [scds_error_string\(3HA\)](#) – Translates an error code to an error string. Strings that are returned by `scds_error_string()` are displayed in English. Strings that are returned by `scds_error_string_i18n()` are displayed in the native language that is specified by the `LC_MESSAGES` locale category.
- [scds_svc_wait\(3HA\)](#) - Waits for the specified timeout period for a monitored process to die.

Property Functions

These functions provide convenience APIs for accessing specific properties of the relevant resource type, resource, and resource group, including some commonly used extension properties. The DSDL provides the `scds_initialize()` function to parse the command-line arguments. The library *cache*s the various properties of the relevant resource type, resource, and resource group.

The [scds_property_functions\(3HA\)](#) man page describes these functions, which include the following:

- `scds_get_ext_property-name`
- `scds_get_rg_property-name`
- `scds_get_rs_property-name`
- `scds_get_rt_property-name`

Network Resource Access Functions

The functions listed in this section retrieve, print, and free the network resources that are used by resources and resource groups. The `scds_get_` functions in this section provide a convenient way of retrieving network resources without using the RMAPI functions to query specific properties, such as `Network_resources_used`, `Resource_dependencies`, and `Port_list`. The `scds_print_name()` functions print values from the data structures that are returned by the `scds_get_name()` functions. The `scds_free_name()` functions free the memory that is allocated by the `scds_get_name()` functions.

Host Name Functions

The following functions handle host names:

- [scds_get_rs_hostnames\(3HA\)](#) – Retrieves a list of host names that is used by the resource.
- [scds_get_rg_hostnames\(3HA\)](#) – Retrieves a list of host names that is used by the network resources in a resource group.
- [scds_print_net_list\(3HA\)](#) – Writes the contents of the host name list to [syslog\(3C\)](#). You typically use this function for debugging.
- [scds_free_net_list\(3HA\)](#) – Frees the memory that is allocated by [scds_get_rs_hostnames\(\)](#) or [scds_get_rg_hostnames\(\)](#).

Port List Functions

The following functions handle port lists:

- [scds_get_port_list\(3HA\)](#) – Retrieves a list of port-protocol pairs that is used by a resource.
- [scds_print_port_list\(3HA\)](#) – Writes the contents of the port-protocol list to [syslog\(3C\)](#). You typically use this function for debugging.
- [scds_free_port_list\(3HA\)](#) – Frees the memory that is allocated by [scds_get_port_list\(\)](#).

Network Address Functions

The following functions handle network addresses:

- [scds_get_netaddr_list\(3HA\)](#) – Retrieves a list of network addresses that is used by a resource.
- [scds_print_netaddr_list\(3HA\)](#) – Writes the contents of the network address list to [syslog\(3C\)](#). You typically use this function for debugging.
- [scds_free_netaddr_list\(3HA\)](#) – Frees the memory that is allocated by [scds_get_netaddr_list\(\)](#).

Fault Monitoring Using TCP Connections Functions

The functions in this section enable TCP-based monitoring. Typically, a fault monitor uses these functions to establish a simple socket connection to a service, read and write data to the service to ascertain its status, and disconnect from the service.

These functions include the following:

- [scds_fm_tcp_connect\(3HA\)](#) – Establishes a TCP connection to a process that uses IPv4 addressing only.
- [scds_fm_net_connect\(3HA\)](#) – Establishes a TCP connection to a process that uses either IPv4 or IPv6 addressing.
- [scds_fm_tcp_read\(3HA\)](#) – Uses a TCP connection to read data from the process that is being monitored.
- [scds_fm_tcp_write\(3HA\)](#) – Uses a TCP connection to write data to a process that is being monitored.
- [scds_simple_probe\(3HA\)](#) – Probes a process by establishing and terminating a TCP connection to the process. This function handles only IPv4 addresses.
- [scds_simple_net_probe\(3HA\)](#) – Probes a process by establishing and terminating a TCP connection to the process. This function handles either IPv4 or IPv6 addresses.
- [scds_fm_tcp_disconnect\(3HA\)](#) – Terminates the connection to a process that is being monitored. This function handles only IPv4 addresses.
- [scds_fm_net_disconnect\(3HA\)](#) – Terminates the connection to a process that is being monitored. This function handles either IPv4 or IPv6 addresses.

PMF Functions

These functions encapsulate the Process Monitor Facility (PMF) functionality. The DSDL model for monitoring through the PMF creates and uses implicit *tag* values for `pmfadm`. See the [pmfadm\(1M\)](#) man page for more information.

The PMF facility also uses implicit values for the `Restart_interval`, `Retry_count`, and `action_script` (the `-t`, `-n`, and `-a` options to `pmfadm`). Most important, the DSDL ties the process failure history, as determined by the PMF, into the application failure history as detected by the fault monitor to compute the restart or failover decision.

The set includes the following functions:

- [scds_fm_sleep\(3HA\)](#) – Waits for a message on a fault monitor control socket.

- `scds_pmf_get_status(3HA)` – Determines if the specified instance is being monitored under the PMF's control.
- `scds_pmf_restart_fm(3HA)` – Uses the PMF to restart the fault monitor.
- `scds_pmf_signal(3HA)` – Sends the specified signal to a process tree that is running under the PMF's control.
- `scds_pmf_start(3HA)` – Executes a specified program (including a fault monitor) under the PMF's control. In addition to performing the same operations as the `scds_pmf_start()` function, the `scds_pmf_start_env()` function also passes a provided environment to the executed program.
- `scds_pmf_stop(3HA)` – Terminates a process that is running under the PMF's control.
- `scds_pmf_stop_monitoring(3HA)` – Stops monitoring a process that is running under the PMF's control.

Fault Monitor Functions

The functions in this section provide a predetermined model of fault monitoring by keeping the failure history and evaluating it in conjunction with the `Retry_count` and `Retry_interval` properties.

This set includes the following functions:

- `scds_fm_sleep(3HA)` – Waits for a message on a fault monitor control socket.
- `scds_fm_action(3HA)` – Takes action after a probe completes.
- `scds_fm_print_probes(3HA)` – Writes probe status information to the system log.

Utility Functions

The following functions enable you to write messages and debugging messages to the system log:

- `scds_syslog(3HA)` – Writes messages to the system log.
- `scds_syslog_debug(3HA)` – Writes debugging messages to the system log.

Cluster Reconfiguration Notification Protocol

This chapter provides information about the Cluster Reconfiguration Notification Protocol (CRNP). The CRNP enables failover and scalable applications to be “cluster aware.” More specifically, the CRNP provides a mechanism that enables applications to register for, and receive subsequent asynchronous notification of, Oracle Solaris Cluster reconfiguration events. Data services that run within the cluster and applications that run outside the cluster can register for notification of events. Events are generated when membership in a cluster changes and when the state of a resource group or a resource changes.

Note - The `SUNW.Event` resource type implementation provides highly available CRNP services on Oracle Solaris Cluster. The implementation of this resource type is described in more detail in the [SUNW.Event\(5\)](#) man page.

This chapter covers the following topics:

- “CRNP Concepts” on page 193
- “How a Client Registers With the Server” on page 197
- “How the Server Replies to a Client” on page 199
- “How the Server Delivers Events to a Client” on page 201
- “How the CRNP Authenticates Clients and the Server” on page 203
- “Example of Creating a Java Application That Uses the CRNP” on page 204

CRNP Concepts

The CRNP defines the Application, Presentation, and Session layers of the standard seven-layer Open System Interconnect (OSI) protocol stack. The Transport layer must be TCP, and the Network layer must be IP. The CRNP is independent of the Data Link and Physical layers. All Application layer messages that are exchanged in the CRNP are based on XML 1.0.

Note - You can run the CRNP only on a global-cluster node.

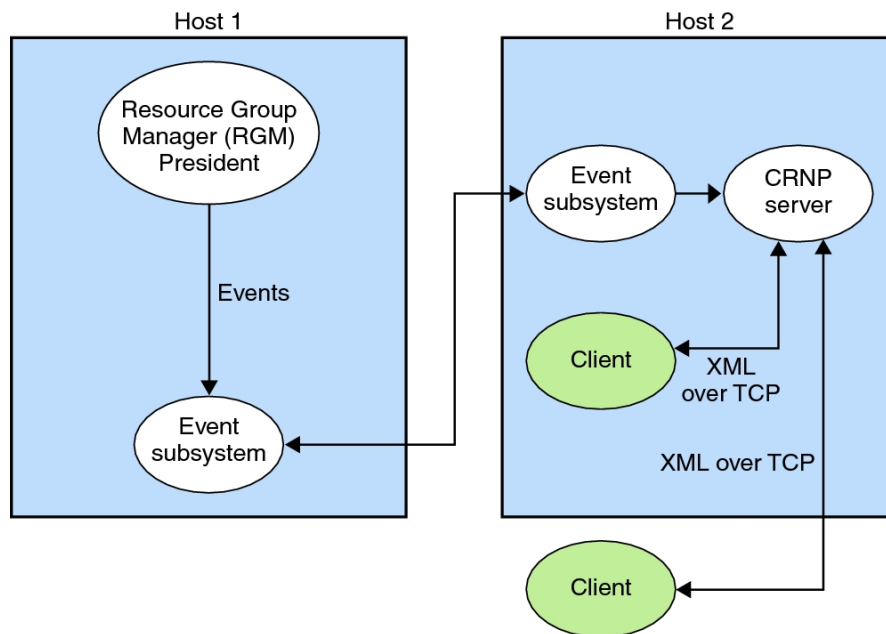
How the CRNP Works

The CRNP provides mechanisms and daemons that generate cluster reconfiguration events, route the events through the cluster, and send them to interested clients.

The `cl_apid` daemon interacts with the clients. The Oracle Solaris Cluster Resource Group Manager (RGM) generates cluster reconfiguration events. This daemon uses `syseventd` to transmit events on each local node. The `cl_apid` daemon uses Extensible Markup Language (XML) over TCP/IP to communicate with interested clients.

The following diagram shows the flow of events between the CRNP components. In this diagram, one client is running on cluster node 2, and the other client is running on a computer that is not part of the cluster.

FIGURE 3 Flow of Events Between CRNP Components



CRNP Semantics

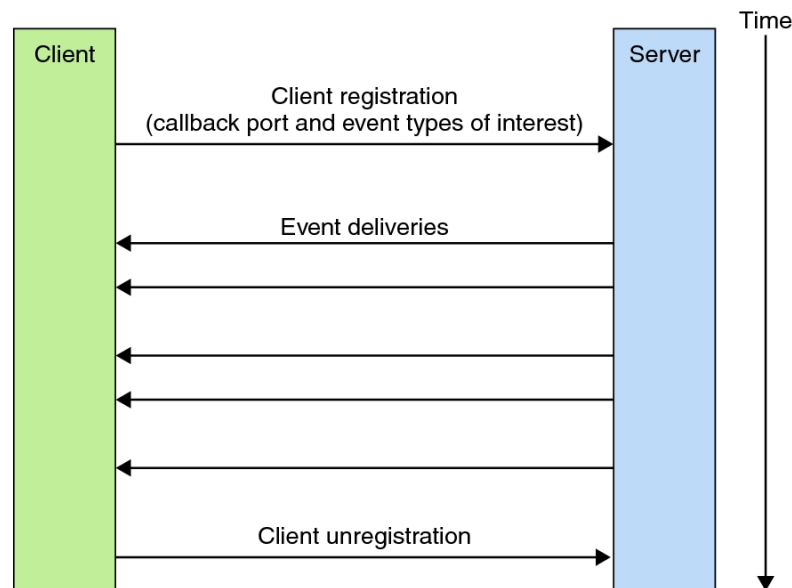
Clients initiate communication by sending a registration message (SC_CALLBACK_RG) to the server. This registration message specifies the event types for which the clients want to receive notification as well as a port to which the events can be delivered. The source IP of the registration connection and the specified port, taken together, form the callback address.

Whenever an event of interest to a client is generated within the cluster, the server contacts the client on its callback address (IP and port) and delivers the event (SC_EVENT) to the client. The server is highly available, running within the cluster itself. The server stores client registrations in storage that persists even after the cluster is rebooted.

Clients unregister by sending a registration message (SC_CALLBACK_RG, which contains a REMOVE_CLIENT message) to the server. After the client receives an SC_REPLY message from the server, the client closes the connection.

The following diagram shows the flow of communication between a client and a server.

FIGURE 4 Flow of Communication Between a Client and a Server



CRNP Message Types

The CRNP uses three types of XML-based messages. Use of these messages is described in the following table. These message types are described in more detail later in this chapter.

CRNP Message Type	Description
SC_CALLBACK_REG	<p>This message takes four forms: ADD_CLIENT, REMOVE_CLIENT, ADD_EVENTS, and REMOVE_EVENTS.</p> <p>Each of these forms contains the following information:</p> <ul style="list-style-type: none"> ■ Protocol version ■ Callback port in ASCII format (not binary format) <p>ADD_CLIENT, ADD_EVENTS, and REMOVE_EVENTS also contain an unbounded list of event types.</p> <p>Each of these forms includes the following information:</p> <ul style="list-style-type: none"> ■ Event class ■ Event subclass (optional) ■ List of the name and value pairs (optional) <p>Together, the event class and event subclass define a unique “event type.” The document type definition (DTD) from which the classes of SC_CALLBACK_REG are generated is SC_CALLBACK_REG. This DTD is described in more detail in Appendix D, “Document Type Definitions for the CRNP”.</p>
SC_REPLY	<p>This message contains the following information:</p> <ul style="list-style-type: none"> ■ Protocol version ■ Error code ■ Error message <p>The DTD from which the classes of SC_REPLY are generated is SC_REPLY. This DTD is described in more detail in Appendix D, “Document Type Definitions for the CRNP”.</p>
SC_EVENT	<p>This message contains the following information:</p> <ul style="list-style-type: none"> ■ Protocol version ■ Event class ■ Event subclass ■ Vendor ■ Publisher ■ Name and value pairs list (0 or more name and value pair data structures) <ul style="list-style-type: none"> ■ Name (string) ■ Value (string or string array) <p>The values in an SC_EVENT are not typed. The DTD from which the classes of SC_EVENT are generated is SC_EVENT. This DTD is described in more detail in Appendix D, “Document Type Definitions for the CRNP”.</p>

How a Client Registers With the Server

This section describes how a cluster administrator sets up the server, how clients are identified, how information is sent over the Application and Session layers, and error conditions.

Assumptions About How Administrators Set Up the Server

The cluster administrator must configure the server with a highly available IP address (one that is not tied to a particular machine in the cluster) and a port number. The cluster administrator must publish this network address to prospective clients. The CRNP does not define how this server name is made available to clients. The cluster administrator either uses a naming service, which enables clients to find the network address of the server dynamically, or adds the network name to a configuration file for the client to read. The server runs within the cluster as a failover resource type.

How the Server Identifies a Client

Each client is uniquely identified by its callback address, that is, its IP address and port number. The port is specified in the `SC_CALLBACK_REG` messages, and the IP address is obtained from the TCP registration connection. The CRNP assumes that subsequent `SC_CALLBACK_REG` messages with the same callback address come from the same client, even if the source port from which the messages are sent is different.

How `SC_CALLBACK_REG` Messages Are Passed Between a Client and the Server

A client initiates a registration by opening a TCP connection to the server's IP address and port number. After the TCP connection is established and ready for writing, the client must send its registration message. The registration message must be one correctly formatted `SC_CALLBACK_REG` message that does not contain extra bytes either before or after the message.

After all the bytes have been written to the stream, the client must keep its connection open to receive the reply from the server. If the client does not format the message correctly, the server does not register the client, and sends an error reply to the client. However, if the client closes the socket connection before the server sends a reply, the server registers the client as usual.

A client can contact the server at any time. Every time a client contacts the server, the client must send an `SC_CALLBACK_REG` message. If the server receives a message that is malformed, out of order, or invalid, the server sends an error reply to the client.

A client cannot send an `ADD_EVENTS`, `REMOVE_EVENTS`, or `REMOVE_CLIENT` message before that client sends an `ADD_CLIENT` message. A client cannot send a `REMOVE_CLIENT` message before that client sends an `ADD_CLIENT` message.

If a client sends an `ADD_CLIENT` message and the client is already registered, the server might tolerate this message. In this situation, the server silently replaces the old client registration with the new client registration that is specified in the second `ADD_CLIENT` message.

In most situations, a client registers with the server once, when the client starts, by sending an `ADD_CLIENT` message. A client unregisters once by sending a `REMOVE_CLIENT` message to the server. However, the CRNP provides more flexibility for those clients that need to modify their event type list dynamically.

Contents of an `SC_CALLBACK_REG` Message

Each `ADD_CLIENT`, `REMOVE_CLIENT`, `ADD_EVENTS`, and `REMOVE_EVENTS` message contains a list of events. The following table describes the event types that the CRNP accepts, including the required name and value pairs.

If a client performs one of the following actions, the server silently ignores these messages:

- Sends a `REMOVE_EVENTS` message that specifies one or more event types for which the client has not previously registered
- Registers for the same event type twice

Class and Subclass	Name and Value Pairs	Description
<code>EC_Cluster</code>	Required: none	Registers for all cluster membership change events (node death or join cluster)
<code>ESC_cluster_membership</code>	Optional: none	
<code>EC_Cluster</code>	One required, as follows:	Registers for all state change events for resource group <i>name</i>
<code>ESC_cluster_rg_state</code>	<i>rg_name</i> Value type: string Optional: none	
<code>EC_Cluster</code>	One required, as follows:	Registers for all state change events for resource <i>name</i>

Class and Subclass	Name and Value Pairs	Description
ESC_cluster_r_state	<i>r_name</i> Value type: string Optional: none	
EC_Cluster	Required: none	Registers for all Oracle Solaris Cluster events
None	Optional: none	

How the Server Replies to a Client

After processing the registration, the server that received the registration request sends the SC_REPLY message on the TCP connection that the client opened. The server closes the connection. The client must keep the TCP connection open until it receives the SC_REPLY message from the server.

For example, the client carries out the following actions:

1. Opens a TCP connection to the server
2. Waits for a connection to be “writable”
3. Sends an SC_CALLBACK_REG message (which contains an ADD_CLIENT message)
4. Waits for an SC_REPLY message from the server
5. Receives an SC_REPLY message from the server
6. Receives an indicator that the server has closed the connection (reads 0 bytes from the socket)
7. Closes the connection

At a later point in time, the client carries out the following actions:

1. Opens a TCP connection to the server
2. Waits for a connection to be “writable”
3. Sends an SC_CALLBACK_REG message (which contains a REMOVE_CLIENT message)
4. Waits for an SC_REPLY message from the server
5. Receives an SC_REPLY message from the server
6. Receives an indicator that the server has closed the connection (reads 0 bytes from the socket)
7. Closes the connection

Each time that the server receives an SC_CALLBACK_REG message from a client, the server sends an SC_REPLY message on the same open connection. This message specifies whether

the operation succeeded or failed. “[SC_REPLY XML DTD](#)” on page 294 contains the XML document type definition of an SC_REPLY message, and the possible error messages that this message can include.

Contents of an SC_REPLY Message

An SC_REPLY message specifies whether an operation succeeded or failed. This message contains the version of the CRNP message, a status code, and a status message, which describes the status code in more detail. The following table describes the possible values for the status code.

Status Code	Description
OK	The message was processed successfully.
RETRY	The registration of the client was rejected by the server due to a transient error. The client should try to register again, with different arguments.
LOW_RESOURCE	Cluster resources are low, and the client can only try again at a later time. The cluster administrator for the cluster can also increase the resources in the cluster.
SYSTEM_ERROR	A serious problem occurred. Contact the cluster administrator for the cluster.
FAIL	Authorization failed or another problem caused the registration to fail.
MALFORMED	The XML request was malformed and could not be parsed.
INVALID	The XML request was invalid , that is, it does not meet the XML specification.
VERSION_TOO_HIGH	The version of the message was too high to process the message successfully.
VERSION_TOO_LOW	The version of the message was too low to process the message successfully.

How a Client Is to Handle Error Conditions

Under normal conditions, a client that sends an SC_CALLBACK_REG message receives a reply that indicates that the registration succeeded or failed.

However, the server can experience an error condition when a client is registering that prohibits the server from sending an SC_REPLY message to the client. In this case, the registration could either have succeeded before the error condition occurred, could have failed, or could not yet have been processed.

Because the server must function as a failover, or highly available, server in the cluster, this error condition does not mean an end to the service. In fact, the server could soon begin sending events to the newly registered client.

To remedy these conditions, your client should perform the following actions:

- Impose an application-level timeout on a registration connection that is waiting for an SC_REPLY message, after which the client needs to retry registering.
- Begin listening on its callback IP address and port number for event deliveries before it registers for the event callbacks. The client should wait for a registration confirmation message and for event deliveries in parallel. If the client begins to receive events before the client receives a confirmation message, the client should silently close the registration connection.

How the Server Delivers Events to a Client

As events are generated within the cluster, the CRNP server delivers them to each client that requested events of those types. The delivery consists of sending an SC_EVENT message to the client's callback address. The delivery of each event occurs on a new TCP connection.

Immediately after a client registers for an event type, through an SC_CALLBACK_REG message that contains an ADD_CLIENT message or an ADD_EVENT message, the server sends the most recent event of that type to the client. The client can determine the current state of the system from which the subsequent events come.

When the server initiates a TCP connection to the client, the server sends exactly one SC_EVENT message on the connection. The server issues a full-duplex close.

For example, the client carries out the following actions:

1. Waits for the server to initiate a TCP connection
2. Accepts the incoming connection from the server
3. Waits for an SC_EVENT message from the server
4. Reads an SC_EVENT message from the server
5. Receives an indicator that the server has closed the connection (reads 0 bytes from the socket)
6. Closes the connection

When all clients have registered, they must listen at their callback address (the IP address and port number) at all times for an incoming event delivery connection.

If the server fails to contact the client to deliver an event, the server tries again to deliver the event the number of times and at the interval that you define. If all attempts fail, the client is removed from the server's list of clients. The client also needs to reregister by sending another

SC_CALLBACK_REG message that contains an ADD_CLIENT message before the client can receive more events.

How the Delivery of Events Is Guaranteed

There is a total ordering of event generation within the cluster that is preserved in the order of delivery to each client. In other words, if event A is generated within the cluster before event B, client X receives event A before that client receives event B. However, the total ordering of event delivery to *all* clients is *not* preserved. That is, client Y could receive both events A and B before client X receives event A. In this way, slow clients do not hold up delivery to all clients.

All events that the server delivers (except the first event for a subclass and events that follow server errors) occur in response to the actual events that the cluster generates, except if the server experiences an error that causes it to miss cluster-generated events. In this case, the server generates an event for each event type that represents the current state of the system for that type. Each event is sent to clients that registered interest in that event type.

Event delivery follows the “at least once” semantics. That is, the server can send the same event to a client more than once. This allowance is necessary in cases in which the server goes down temporarily, and when it comes back up, cannot determine whether the client has received the latest information.

Contents of an SC_EVENT Message

The SC_EVENT message contains the actual message that is generated within the cluster, translated to fit into the SC_EVENT XML message format. The following table describes the event types that the CRNP delivers, including the name and value pairs, publisher, and vendor.

Note - The positions of the array elements for `state_list` are synchronized with those of the `node_list`. That is, the state for the node that is listed first in the `node_list` array is listed first in the `state_list` array.

Additional names starting with `ev_` and their associated values might be present, but are not intended for client use.

Class and Subclass	Publisher and Vendor	Name and Value Pairs
EC_Cluster	Publisher: rgm	Name: node_list
ESC_cluster_membership	Vendor: ORCL	Value type: string array

Class and Subclass	Publisher and Vendor	Name and Value Pairs
		<p>Name: <code>state_list</code></p> <p>The <code>state_list</code> contains only numbers that are represented in ASCII. Each number represents the current incarnation number for that node in the cluster. If the number is the same as the number that was received in a previous message, the node has not changed its relationship to the cluster (departed, joined, or rejoined). If the incarnation number is <code>-1</code>, the node is not a member of the cluster. If the incarnation number is a number other than a negative number, the node is a member of the cluster.</p> <p>Value type: string array</p>
<code>EC_Cluster</code>	Publisher: <code>rgm</code>	Name: <code>rg_name</code>
<code>ESC_cluster_rg_state</code>	Vendor: <code>ORCL</code>	<p>Value type: string</p> <p>Name: <code>node_list</code></p> <p>Value type: string array</p> <p>Name: <code>state_list</code></p> <p>The <code>state_list</code> contains string representations of the state of the resource group. Valid values are those values that you can retrieve with the <code>scha_cmds(1HA)</code> commands.</p> <p>Value type: string array</p>
<code>EC_Cluster</code>	Publisher: <code>rgm</code>	Name: <code>r_name</code>
<code>ESC_cluster_r_state</code>	Vendor: <code>ORCL</code>	<p>Value type: string</p> <p>Name: <code>node_list</code></p> <p>Value type: string array</p> <p>Name: <code>state_list</code></p> <p>The <code>state_list</code> contains string representations of the state of the resource. Valid values are those values that you can retrieve with the <code>scha_cmds(1HA)</code> commands.</p> <p>Value type: string array</p>

How the CRNP Authenticates Clients and the Server

The server authenticates a client by using a form of TCP wrappers. The source IP address of the registration message, which is also used as the callback IP address on which events

are delivered, must be in the list of allowed clients on the server. The source IP address and registration message cannot be in the denied clients list. If the source IP address and registration are not in the list, the server rejects the request and issues an error reply to the client.

When the server receives an `SC_CALLBACK_REG ADD_CLIENT` message, subsequent `SC_CALLBACK_REG` messages for that client must contain a source IP address that is the same as the source IP address in the first message.

If the CRNP server receives an `SC_CALLBACK_REG` that does not meet this requirement, the server performs one of the following actions:

- Ignores the request and sends an error reply to the client
- Assumes that the request comes from a new client, depending on the contents of the `SC_CALLBACK_REG` message

This security mechanism helps to prevent denial of service attacks, where someone attempts to unregister a legitimate client.

Clients should also similarly authenticate the server. Clients need only accept event deliveries from a server whose source IP address and port number are the same as the registration IP address and port number that the client used.

Because clients of the CRNP service are supposed to be located inside a firewall that protects the cluster, the CRNP does not include additional security mechanisms.

Example of Creating a Java Application That Uses the CRNP

The following example illustrates how to develop a simple Java application named `CrnpClient` that uses the CRNP. The application registers for event callbacks with the CRNP server in the cluster, listens for the event callbacks, and processes the events by printing their contents. Before terminating, the application unregisters its request for event callbacks.

Note the following points when reviewing this example:

- The sample application generates and parses XML with the JAXP (Java API for XML Processing). This example does not show you how to use the JAXP. The JAXP is described in more detail at <https://jaxp.java.net/>.
- This example presents pieces of an application, which can be found in its entirety in [Appendix E, “CrnpClient.java Application”](#). To illustrate particular concepts more effectively, the example in this chapter differs slightly from the complete application that is presented in [Appendix E, “CrnpClient.java Application”](#).

- For brevity, comments are excluded from the sample code in this chapter. The complete application in [Appendix E, “CrnpClient.java Application”](#) includes comments.
- The application that is shown in this example handles most error conditions by simply exiting the application. Your actual application needs to handle errors more robustly.

▼ How to Set Up Your Environment

1. **Download and install JAXP and the correct version of the Java compiler and virtual machine.**

You can find instructions at <https://jaxp.java.net/>.

Note - This example requires at least Java 1.3.1.

2. **From the directory in which your source file is located, type the following:**

```
% javac -classpath jaxp-root/dom.jar:jaxp-root/jaxp-api. \
jar:jaxp-rootsax.jar:jaxp-root/xalan.jar:jaxp-root/xercesImpl \
.jar:jaxp-root/xslt.jar -sourcepath . source-filename.java
```

where *jaxp-root* is the absolute or relative path to the directory in which the JAXP jar files are located and *source-filename* is the name of your Java source file.

A `classpath` in your compilation command line ensures that the compiler can find the JAXP classes.

3. **When you run the application, specify the `classpath` so that the application can load the correct JAXP class files.**

Note that the first path in the `classpath` is the current directory.

```
% java -cp .:jaxp-root/dom.jar:jaxp-root/jaxp-api. \
jar:jaxp-rootsax.jar:jaxp-root/xalan.jar:jaxp-root/xercesImpl \
.jar:jaxp-root/xslt.jar source-filename arguments
```

Now that your environment is configured, you can develop your application.

▼ How to Start Developing Your Application

In this part of the example, you create a basic class called `CrnpClient`, with a main method that parses the command-line arguments and constructs a `CrnpClient` object. This object passes

the command-line arguments to the class, waits for the user to terminate the application, calls shutdown on the `CrnpClient`, and exits.

The constructor of the `CrnpClient` class needs to execute the following tasks:

- Set up the XML processing objects.
- Create a thread that listens for event callbacks.
- Contact the CRNP server and register for event callbacks.

● **Create the Java code that implements the preceding logic.**

The following example shows the skeleton code for the `CrnpClient` class. The implementations of the four helper methods that are referenced in the constructor and shutdown methods are shown later in this chapter. Note that the code that imports all the packages that you need is shown.

```
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;
}
```

Member variables are discussed in more detail later in this chapter.

▼ How to Parse the Command-Line Arguments

- To parse the command-line arguments, see the code in [Appendix E, “CrnpClient.java Application”](#).

▼ How to Define the Event Reception Thread

In the code, you need to ensure that event reception is performed in a separate thread so that your application can continue to do other work while the event thread blocks and waits for event callbacks.

Note - Setting up the XML is discussed later in this chapter.

1. **In your code, define a Thread subclass called `EventReceptionThread` that creates a `ServerSocket` and waits for events to arrive on the socket.**

In this part of the example code, events are neither read nor processed. Reading and processing events are discussed later in this chapter. The `EventReceptionThread` creates a `ServerSocket` on a wildcard internet-working protocol address. `EventReceptionThread` also keeps a reference to the `CrnpClient` object so that `EventReceptionThread` can send events to the `CrnpClient` object to process.

```
class EventReceptionThread extends Thread
{
    public EventReceptionThread(CrnpClient clientIn) throws IOException
    {
        client = clientIn;
        listeningSock = new ServerSocket(client.getLocalPort(), 50,
            InetAddress.getLocalHost());
    }

    public void run()
    {
        try {
            DocumentBuilder db = client.dbf.newDocumentBuilder();
            db.setErrorHandler(new DefaultHandler());

            while(true) {
                Socket sock = listeningSock.accept();
                // Construct event from the sock stream and process it
                sock.close();
            }
            // UNREACHABLE

        } catch (Exception e) {
            System.out.println(e);
            System.exit(1);
        }
    }

    /* private member variables */
}
```



```
private ServerSocket listeningSock;
private CrnpClient client;
}
```

2. Construct a createEvtRecepThr object.

```
private void createEvtRecepThr() throws Exception
{
    evtThr = new EventReceptionThread(this);
    evtThr.start();
}
```

▼ How to Register and Unregister Callbacks

The registration task involves the following actions:

- Opening a basic TCP socket to the registration internet-working protocol and port
- Constructing the XML registration message
- Sending the XML registration message on the socket
- Reading the XML reply message off the socket
- Closing the socket

1. Create the Java code that implements the preceding logic.

The following example code shows the implementation of the registerCallbacks method of the CrnpClient class (which is called by the CrnpClient constructor). The calls to createRegistrationString() and readRegistrationReply() are described in more detail later in this chapter.

regIp and regPort are object members that are set up by the constructor.

```
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. Implement the unregister method.

This method is called by the shutdown method of `CrnpClient`. The implementation of `createUnregistrationString` is described in more detail later in this chapter.

```
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();
}
```

▼ How to Generate the XML

Now that you have set up the structure of the application and have written all the networking code, you need to write the code that generates and parses the XML. Start by writing the code that generates the `SC_CALLBACK_REG` XML registration message.

An `SC_CALLBACK_REG` message consists of a registration type (`ADD_CLIENT`, `REMOVE_CLIENT`, `ADD_EVENTS`, or `REMOVE_EVENTS`), a callback port, and a list of events of interest. Each event consists of a class and a subclass, followed by a list of name and value pairs.

In this part of the example, you write a `CallbackReg` class that stores the registration type, callback port, and list of registration events. This class also can serialize itself to an `SC_CALLBACK_REG` XML message.

An interesting method of this class is the `convertToXml` method, which creates an `SC_CALLBACK_REG` XML message string from the class members. The JAXP documentation at <https://jaxp.java.net/> describes the code in this method in more detail.

The implementation of the `Event` class is shown in the following example code. Note that the `CallbackReg` class uses an `Event` class that stores one event and can convert that event to an XML Element.

1. Create the Java code that implements the preceding logic.

```
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) {
// Parser with specified options can't be built
pce.printStackTrace();
System.exit(1);
}

// Create the root element
Element root = (Element) document.createElement("SC_CALLBACK_REG");

// Add the attributes
root.setAttribute("VERSION", "1.0");
root.setAttribute("PORT", port);
root.setAttribute("regType", regType);

// Add the events
for (int i = 0; i < regEvents.size(); i++) {
Event tempEvent = (Event)
(regEvents.elementAt(i));
root.appendChild(tempEvent.createXmlElement(document));
}
document.appendChild(root);

// Convert the whole thing to a string
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. Implement the Event and NVPair classes.

Note that the CallbackReg class uses an Event class, which itself uses an NVPair class.

```
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;
}
```

▼ How to Create the Registration and Unregistration Messages

Now that you have created the helper classes that generate the XML messages, you can write the implementation of the `createRegistrationString` method. This method is called by the `registerCallbacks` method, which is described in [“How to Register and Unregister Callbacks” on page 209](#).

`createRegistrationString` constructs a `CallbackReg` object and sets its registration type and port. Then, `createRegistrationString` constructs various events, by using the `createAllEvent`, `createMembershipEvent`, `createRgEvent`, and `createREvent` helper methods. Each event is added to the `CallbackReg` object after this object is created. Finally,

`createRegistrationString` calls the `convertToXml` method on the `CallbackReg` object to retrieve the XML message in `String` form.

Note that the `regs` member variable stores the command-line arguments that a user provides to the application. The fifth and subsequent arguments specify the events for which the application should register. The fourth argument specifies the type of registration, but is ignored in this example. The complete code in [Appendix E, “CrnpClient.java Application”](#) shows how to use this fourth argument.

1. Create the Java code that implements the preceding logic.

```
private String createRegistrationString() throws Exception
{
    CallbackReg cbReg = new CallbackReg();
    cbReg.setPort("" + localPort);

    cbReg.setRegType(CallbackReg.ADD_CLIENT);

    // add the events
    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. Create the unregistration string.

Creating the unregistration string is easier than creating the registration string because you do not need to accommodate events.

```
private String createUnregistrationString() throws Exception
{
CallbackReg cbReg = new CallbackReg();
cbReg.setPort("" + localPort);
cbReg.setRegType(CallbackReg.REMOVE_CLIENT);
String xmlStr = cbReg.convertToXml();
return (xmlStr);
}
```


▼ How to Set Up the XML Parser

You have now created the networking and XML generation code for the application. The `CrnpClient` constructor calls a `setupXmlProcessing` method. This method creates a `DocumentBuilderFactory` object and sets various parsing properties on that object. The JAXP documentation describes this method in more detail. See <https://jaxp.java.net/>.

● Create the Java code that implements the preceding logic.

```
private void setupXmlProcessing() throws Exception
{
    dbf = DocumentBuilderFactory.newInstance();

    // We don't need to bother validating
    dbf.setValidating(false);
    dbf.setExpandEntityReferences(false);

    // We want to ignore comments and whitespace
    dbf.setIgnoringComments(true);
    dbf.setIgnoringElementContentWhitespace(true);

    // Coalesce CDATA sections into TEXT nodes.
    dbf.setCoalescing(true);
}
```

▼ How to Parse the Registration Reply

To parse the `SC_REPLY` XML message that the CRNP server sends in response to a registration or unregistration message, you need a `RegReply` helper class. You can construct this class from an XML document. This class provides accessors for the status code and status message. To parse the XML stream from the server, you need to create a new XML document and use that document's parse method. The JAXP documentation at <https://jaxp.java.net/> describes this method in more detail.

1. Create the Java code that implements the preceding logic.

Note that the `readRegistrationReply` method uses the new `RegReply` class.

```
private void readRegistrationReply(InputStream stream) throws Exception
{
    // Create the document builder
    DocumentBuilder db = dbf.newDocumentBuilder();
    db.setErrorHandler(new DefaultHandler());
}
```

```
//parse the input file
Document doc = db.parse(stream);

RegReply reply = new RegReply(doc);
reply.print(System.out);
}
```

2. Implement the RegReply class.

Note that the `retrieveValues` method walks the DOM tree in the XML document and pulls out the status code and status message. The JAXP documentation at <https://jaxp.java.net/> contains more detail.

```
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;

// Find the SC_REPLY element.
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);

// Retrieve the value of the statusCode attribute
statusCode = ((Element)n).getAttribute("STATUS_CODE");

// Find the SC_STATUS_MSG element
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;
}
// Get the TEXT section, if there is one.
n = nl.item(0).getFirstChild();
if (n == null || n.getNodeType() != Node.TEXT_NODE) {
// Not an error if there isn't one, so we just silently return.
return;
}

// Retrieve the value
statusMsg = n.getNodeValue();
}

private String statusCode;
private String statusMsg;
}

```

▼ How to Parse the Callback Events

The final step is to parse and process the actual callback events. To aid in this task, you modify the Event class that you created in [“How to Generate the XML” on page 210](#) so that this class can construct an Event from an XML document and create an XML Element. This change requires an additional constructor (that takes an XML document), a retrieveValues method, the addition of two member variables (vendor and publisher), accessor methods for all fields, and finally, a print method.

1. Create the Java code that implements the preceding logic.

Note that this code is similar to the code for the RegReply class that is described in [“How to Parse the Registration Reply” on page 217](#).

```

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;

    // Find the SC_EVENT element.
    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);

    //
    // Retrieve the values of the CLASS, SUBCLASS,
    // VENDOR and PUBLISHER attributes.
    //
    regClass = ((Element)n).getAttribute("CLASS");
    regSubclass = ((Element)n).getAttribute("SUBCLASS");
    publisher = ((Element)n).getAttribute("PUBLISHER");
    vendor = ((Element)n).getAttribute("VENDOR");

    // Retrieve all the nv pairs
    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. Implement the additional constructors and methods for the NVPair class that support the XML parsing.

The changes to the Event class that are shown in [Step 1](#) require similar changes to the NVPair class.

```
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;
```

```
// Find the NAME element
n1 = elem.getElementsByTagName("NAME");
if (n1.getLength() != 1) {
System.out.println("Error in parsing: can't find "
+ "NAME node.");
return;
}
// Get the TEXT section
n = n1.item(0).getFirstChild();
if (n == null || n.getNodeType() != Node.TEXT_NODE) {
System.out.println("Error in parsing: can't find "
+ "TEXT section.");
return;
}

// Retrieve the value
name = n.getNodeValue();

// Now get the value element
n1 = elem.getElementsByTagName("VALUE");
if (n1.getLength() != 1) {
System.out.println("Error in parsing: can't find "
+ "VALUE node.");
return;
}
// Get the TEXT section
n = n1.item(0).getFirstChild();
if (n == null || n.getNodeType() != Node.TEXT_NODE) {
System.out.println("Error in parsing: can't find "
+ "TEXT section.");
return;
}

// Retrieve the value
value = n.getNodeValue();
}

public String getName()
{
return (name);
}

public String getValue()
{
return (value);
}
}
```

3. Implement the while loop in `EventReceptionThread`, which waits for event callbacks.

`EventReceptionThread` is described in [“How to Define the Event Reception Thread” on page 208](#).

```
while(true) {
    Socket sock = listeningSock.accept();
    Document doc = db.parse(sock.getInputStream());
    Event event = new Event(doc);
    client.processEvent(event);
    sock.close();
}
```

▼ How to Run the Application

- 1. Assume the root role that provides `solaris.cluster.modify RBAC` authorization.**
- 2. Run your application.**

```
# java CrnpClient crnpHost crnpPort localPort ...
```

The complete code for the `CrnpClient` application is listed in [Appendix E, “`CrnpClient.java` Application”](#).

Security for Data Services

Oracle Solaris Cluster enables you to enhance the security of your data service using the features described in this chapter.

- “[Storing Application Passwords in Private Strings](#)” on page 225
- “[Invoking Application Programs with Least Privilege](#)” on page 227

Storing Application Passwords in Private Strings

Oracle Solaris Cluster provides the `clpstring` command to enable you to securely store and retrieve private character strings in the cluster file system. These private strings can be used for any purpose, but might typically be used to store a password. For example, a data service agent might retrieve the private string to log in as a privileged user to perform an orderly shut down of the managed application running in the cluster. The private string capability might also be useful for other aspects of the cluster framework, such as to interact with an external storage device.

You create the private string by using the `clpstring` command to assign a unique name to reference the string. The value of the string can be specified in several different ways at the time of creation, and changed at a later time if necessary. The private string is encoded and stored in a protected location that is readable only by root. You can use the `clpstring` command to list the names and other information about private strings that have been created, but the encoded value can only be obtained by using the `scha_cluster_get(1HA)` command.

Use the `clpstring` command for the following administrative tasks:

- Create a private string that is intended to be used by a cluster object instance that may or may not yet exist, using the `create` subcommand.
- Update the value of private string, using the `set` subcommand.
- Delete private strings from the cluster configuration, using the `delete` subcommand.
- Display the specifications of private strings, using the `show` subcommand.

- List the names of private strings, using the `list` subcommand.

You must have `solaris.cluster.modify` role-based access control (RBAC) authorization to use the `clpstring` command to create, update, or delete private strings. You must have `solaris.cluster.read` authorization to show or list information about private strings.

Typical usage of private strings might be similar to the following:

1. On one of the cluster nodes, create the private string name with the `clpstring create` command.

For example, in the following command you are prompted to enter the value of a string you have named `pw_string`. The `-b` option indicates that a resource instance named `resource1` is going to use this private string. The `-t` option indicates that the instance's type is `resource`, which is the default. The `-v` option provides verbose output.

```
# clpstring create -b resource1 -t resource -v pw_string
Enter string value:
Enter string value again:
Private string "pw_string" is created for the global cluster.
```

When you enter the private string value in this way, the characters are not echoed to the screen for increased security.

You can also provide the value of the string by creating a plain text file containing the value and specifying the path to the file by using the `-f` option with the `create` subcommand.

2. The data service agent runs a script to retrieve the private string value using the `scha_cluster_get` command.

The following is a simple example to show how to retrieve a private string for an application:

```
# 'myapp' is an application program that requires a password, which it
# reads from standard input. The cluster administrator has been
# instructed to create a private string 'pw_string' containing the password.
# We use sch_cluster_get to retrieve the password and pipe it into the
# 'myapp' program. The program's output is captured in a file.

scha_cluster_get -O pstring pw_string | myapp > myapp_output_file
```

Note - If you want to change the name of the private string, you must delete the string with `clpstring` command, and then create a new private string.

For more information about the `clpstring` command, see the [clpstring\(1CL\)](#) man page.

Invoking Application Programs with Least Privilege

A data service can invoke an application following the concept of least privilege to decrease the possibility of a security breach. With this approach, you require the application to be executed with the minimal privileges required to perform its tasks.

See the manual [Developer's Guide to Oracle Solaris 11 Security](#) in the [Oracle Solaris 11 documentation library](#) for more information about developing privileged applications in Oracle Solaris 11.

As good security practice, application programs should not be run as root. Executables generally should be owned by root and run as non-root. If an executable binary or script owned by a non-root user can be executed as root, it would be possible for the non-root user to insert malicious operations into that executable, which could be executed with full privileges.

Cluster agent methods should run all external programs using a wrapper to ensure that the external program is executed with the correct username and privilege.

Oracle Solaris Cluster provides the `application_user` and `resource_security` properties and the `scha_check_app_user` command to enable data services to ensure that the application is executed securely. The `scha_check_app_user` command can be called in scripts to verify the username against the configured `application_user` and `resource_security` settings.

See the following sections for information about using resource properties and commands to set up a data service to run with the least privileges required.

- [“Using the resource_security Property” on page 227](#)
- [“Using the application_user Property” on page 228](#)
- [“Using the sch_a_check_app_user Command” on page 229](#)

Using the resource_security Property

In the current Oracle Solaris Cluster release, the execution of a cluster resource is controlled by the setting of a global cluster property called `resource_security`, which is set to `SECURE` by default. This setting ensures that the execution of resource methods is secure.

Resource methods such as `Start` and `Validate` always run as root. If the method executable file is owned by a username other than root or allows write permission for the UNIX group or world, an insecurity exists. In this case, if the `resource_security` property is set to `SECURE`, execution of the resource method fails at run time and an error is returned. If

resource_security has any other setting, the resource method is allowed to execute with a warning message. Permissible values of resource_security are SECURE, WARN, OVERRIDE, or COMPATIBILITY.

A cluster administrator can set the resource_security property for a cluster by using the cluster command, and for a zone cluster by using the clzonecluster command. The clsetup utility can also be used.

If a cluster is upgraded from an earlier release which does not include this property, the resource_security property is set to COMPATIBILITY to ensure that executables can run. However, the administrator should change the setting to SECURE.

For example, the following command sets the property to SECURE on a cluster:

```
# cluster set -p resource_security=SECURE
```

The resource_security setting also modifies the behavior of resource types that declare the application_user resource property, as described in the following section.

Using the application_user Property

A data service can declare the application_user resource property to provide a way for the cluster administrator to specify a non-root Solaris user name to be used for execution of application programs by the data service.

If your agent declares the application_user property, it is expected to set the user ID for execution of application programs according to the settings of the application_user resource property and the resource_security cluster property. You can use the scha_check_app_user command to help with this task. See [“Using the scha_check_app_user Command” on page 229](#) for more information.

If resource_security is set to COMPATIBILITY, the setting of the application_user resource property is ignored and the application user will be the user ID of the caller (usually root). This behavior is compatible with previous releases of Oracle Solaris Cluster.

If resource_security is set to OVERRIDE, the application_user property is ignored and the application user will be the owner of the application program executable file.

If resource_security is set to SECURE or WARN, the application user will be the value of the application_user resource property; however, if application_user is unset or empty, the application user will be the owner of the application program executable file.

Using the `scha_check_app_user` Command

You can use the `scha_check_app_user` command to obtain the configured application user name for a resource that is under the control of the RGM, and check the ownership and permissions on an executable file. The executable file is typically an application program that is intended to be executed by a method or monitor of the resource, using a wrapper such as [su\(1M\)](#) to set the user ID to the configured user. The resource method or monitor should invoke `scha_check_app_user` prior to execution of the application program. Depending on the output of `scha_check_app_user`, the method or monitor should return an error or output a warning message if security-related problems are detected.

See the [scha_check_app_user\(1HA\)](#) man page for more information.

Executing the Application Under a Specified Oracle Solaris Project

If the `su username` command is used in a `Prenet_start` or `Start` method to execute the application program as a specified user, the application runs under the Oracle Solaris project associated with the username, regardless of the setting of the `Resource_project_name` resource property or `RG_project_name` resource group property.

To restore the correct behavior of the `Resource_project_name` and `RG_project_name` properties, execute `newtask(1)` to set the project name after executing the `su` command. The user-specified project name can be determined by querying the `Resource_project_name` resource property and the `RG_project_name` resource group property. If `Resource_project_name` is set to a non-empty value, its value takes precedence and must be used in the `newtask` command. If `Resource_project_name` is null or empty, the value of `RG_project_name` (if non-empty), must be used. If both of these properties are null or empty, your `Start` method can set a particular project name, for example `default`, or can allow the application to execute under the project associated with the username.

Sample Data Service Code Listings

This appendix provides the complete code for each method in the sample data service. It also lists the contents of the resource type registration (RTR) file.

This appendix covers the following topics:

- [“Resource Type Registration File Listing” on page 231](#)
- [“Start Method Code Listing” on page 234](#)
- [“Stop Method Code Listing” on page 237](#)
- [“gettime Utility Code Listing” on page 240](#)
- [“PROBE Program Code Listing” on page 240](#)
- [“Monitor_start Method Code Listing” on page 246](#)
- [“Monitor_stop Method Code Listing” on page 248](#)
- [“Monitor_check Method Code Listing” on page 250](#)
- [“Validate Method Code Listing” on page 252](#)
- [“Update Method Code Listing” on page 256](#)

Resource Type Registration File Listing

The RTR file contains resource and resource type property declarations that define the initial configuration of the data service at the time that the cluster administrator registers the data service.

EXAMPLE 5 ORCL.Sample RTR File

```
#
# Copyright (c) 1998, 2011, Oracle and/or its affiliates.
# All rights reserved.
#
# Registration information for Domain Name Service (DNS)
#
```

```
#pragma ident "@(#)ORCL.sample 1.1 12/01/24"

Resource_type = "sample";
Vendor_id = ORCL;
RT_description = "Domain Name Service on Oracle Solaris Cluster";

RT_version = "1.0";
Failover = TRUE;

RT_basedir=/opt/ORCLsample/bin;
Pkglist = ORCLsample;

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;
# $upgrade
# A list of bracketed resource property declarations follows the
# resource type declarations. The property-name declaration must be
# the first attribute after the open curly bracket of each entry.
#
# The <method>_timeout properties set the value in seconds after which
# the RGM concludes invocation of the method has failed.

# The MIN value for all method timeouts is set to 60 seconds. This
# prevents administrators from setting shorter timeouts, which do not
# improve switchover/failover performance, and can lead to undesired
# RGM actions (false failovers, node reboot, or moving the resource group
# to ERROR_STOP_FAILED state, requiring operator intervention). Setting
# too-short method timeouts leads to a *decrease* in overall availability
# of the data service.
{
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;
}

# The number of retries to be done within a certain period before concluding
# that the application cannot be successfully started on this node.
{
PROPERTY = Retry_count;
MIN=0;
MAX=10;
DEFAULT=2;
TUNABLE = ANYTIME;
}

# Set Retry_interval as a multiple of 60 since it is converted from seconds
# to minutes, rounding up. For example, a value of 50 (seconds)
# is converted to 1 minute. Use this property to time the number of
# retries (Retry_count).
{
PROPERTY = Retry_interval;
MIN=60;
MAX=3600;
DEFAULT=300;
TUNABLE = ANYTIME;
}
```

```
}

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

#
# Extension Properties
#

# The cluster administrator must set the value of this property to point to the
# directory that contains the configuration files used by the application.
# For this application, DNS, specify the path of the DNS configuration file on
# PXFS (typically named.conf).
{
PROPERTY = Confdir;
EXTENSION;
STRING;
TUNABLE = AT_CREATION;
DESCRIPTION = "The Configuration Directory Path";
}

# Time out value in seconds before declaring the probe as failed.
{
PROPERTY = Probe_timeout;
EXTENSION;
INT;
DEFAULT = 30;
TUNABLE = ANYTIME;
DESCRIPTION = "Time out value for the probe (seconds)";
}
```

Start Method Code Listing

The RGM runs the Start method on a cluster node when the resource group that contains the data service resource is brought online on that node. The RGM also does so when the resource is enabled. In the sample application, the Start method activates the `in.named` (DNS) daemon on that node.

EXAMPLE 6 dns_svc_start Method

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

```

#
# Start Method for HA-DNS.
#
# This method starts the data service under the control of PMF. Before starting
# the in.named process for DNS, it performs some sanity checks. The PMF tag for
# the data service is $RESOURCE_NAME.named. PMF tries to start the service a
# specified number of times (Retry_count) and if the number of attempts exceeds
# this value within a specified interval (Retry_interval) PMF reports a failure
# to start the service. Retry_count and Retry_interval are both properties of the
# resource set in the RTR file.

#pragma ident "@(#)dns_svc_start 1.1 12/01/24"

#####
# Parse program arguments.
#
function parse_args # [args ...]
{
typeset opt

while getopts `R:G:T:` opt
do
case "$opt" in
R)
# Name of the DNS resource.
RESOURCE_NAME=$OPTARG
;;
G)
# Name of the resource group in which the resource is
# configured.
RESOURCEGROUP_NAME=$OPTARG
;;
T)
# Name of the resource type.
RESOURCETYPE_NAME=$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  
  
# Obtain the syslog facility to use to log messages.  
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`  
  
# Parse the arguments that have been passed to this method  
parse_args "$@"  
  
PMF_TAG=$RESOURCE_NAME.named  
SYSLOG_TAG=$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME  
  
# Get the value of the Confdir property of the resource in order to start  
# DNS. Using the resource name and the resource group entered, find the value of  
# Confdir value set by the cluster administrator when adding the resource.  
config_info=scha_resource_get -O Extension -R $RESOURCE_NAME \  
-G $RESOURCEGROUP_NAME Confdir`  
# scha_resource_get returns the "type" as well as the "value" for the extension  
# properties. Get only the value of the extension property.  
CONFIG_DIR=`echo $config_info | awk '{print $2}'`  
  
# Check if $CONFIG_DIR is accessible.  
if [ ! -d $CONFIG_DIR ]; then  
logger -p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG] \  
"${ARGV0} Directory $CONFIG_DIR missing or not mounted"  
exit 1  
fi  
  
# Change to the $CONFIG_DIR directory in case there are relative  
# path names in the data files.  
cd $CONFIG_DIR  
  
# Check that the named.conf file is present in the $CONFIG_DIR directory.  
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
```

```

# Get the value for Retry_count from the RTR file.
RETRY_CNT=`scha_resource_get -O Retry_count -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAME`

# Get the value for Retry_interval from the RTR file. Convert this value, which is in
# seconds, to minutes for passing to pmfadm. Note that this is a conversion with
# round-up, for example, 50 seconds rounds up to one minute.
((RETRY_INTRVAL = `scha_resource_get -O Retry_interval -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAME` 60))

# Start the in.named daemon under the control of PMF. Let it crash and restart
# up to $RETRY_COUNT times in a period of $RETRY_INTRVAL; if it crashes
# more often than that, PMF will cease trying to restart it. If there is a
# process already registered under the tag <$PMF_TAG>, then, PMF sends out
# an alert message that the process is already running.
echo "Retry interval is "$RETRY_INTRVAL
pmfadm -c $PMF_TAG.named -n $RETRY_CNT -t $RETRY_INTRVAL \
/usr/sbin/in.named -c named.conf

# Log a message indicating that HA-DNS has been started.
if [ $? -eq 0 ]; then
logger -p ${SYSLOG_FACILITY}.info -t [SYSLOG_TAG] \
"${ARGV0} HA-DNS successfully started"
fi
exit 0

```

Stop Method Code Listing

The RGM runs the Stop method on a cluster node when the resource group that contains the HA-DNS resource is brought offline on that node. The RGM also does so when the resource is disabled. This method stops the `in.named` (DNS) daemon on that node.

EXAMPLE 7 dns_svc_stop Method

```

#!/bin/ksh
#
# Stop method for HA-DNS
#
# Stop the data service using PMF. If the service is not running the
# method exits with status 0 as returning any other value puts the resource
# in STOP_FAILED state.

#pragma ident "@(#)dns_svc_stop 1.1 12/01/24"

```

```
#####
# Parse program arguments.
#
function parse_args # [args ...]
{
typeset opt

while getopts `R:G:T:` opt
do
case "$opt" in
R)
# Name of the DNS resource.
RESOURCE_NAME=$OPTARG
;;
G)
# Name of the resource group in which the resource is
# configured.
RESOURCEGROUP_NAME=$OPTARG
;;
T)
# Name of the resource type.
RESOURCETYPE_NAME=$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

# Obtain the syslog facility to use to log messages.
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# Parse the arguments that have been passed to this method
parse_args "$@"
```

```

PMF_TAG=$RESOURCE_NAME.named
SYSLOG_TAG=$RESOURCE_TYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME

# Obtain the Stop_timeout value from the RTR file.
STOP_TIMEOUT=`scha_resource_get -O STOP_TIMEOUT -R $RESOURCE_NAME -G \
$RESOURCEGROUP_NAME`

# Attempt to stop the data service in an orderly manner using a SIGTERM
# signal through PMF. Wait for up to 80% of the Stop_timeout value to
# see if SIGTERM is successful in stopping the data service. If not, send SIGKILL
# to stop the data service. Use up to 15% of the Stop_timeout value to see
# if SIGKILL is successful. If not, there is a failure and the method exits with
# non-zero status. The remaining 5% of the Stop_timeout is for other uses.
((SMOOTH_TIMEOUT=$STOP_TIMEOUT * 80/100))

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

# See if in.named is running, and if so, kill it.
if pmfadm -q $PMF_TAG.named; then
# Send a SIGTERM signal to the data service and wait for 80% of the
# total timeout value.
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"

# Since the data service did not stop with a SIGTERM signal, use
# SIGKILL now and wait for another 15% of the total timeout value.
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 UNSUCCESSFUL"

exit 1
fi
fi
else
# The data service is not running as of now. Log a message and
# exit success.
logger -p ${SYSLOG_FACILITY}.info -t [SYSLOG_TAG] \
"HA-DNS is not started"

# Even if HA-DNS is not running, exit success to avoid putting
# the data service in STOP_FAILED State.
exit 0
fi

```

```
# Successfully stopped DNS. Log a message and exit success.
logger -p ${SYSLOG_FACILITY}.info -t [SYSLOG_TAG] \
"HA-DNS successfully stopped"
exit 0
```

gettime Utility Code Listing

The `gettime` utility is a C program that is used by the `PROBE` program to track the elapsed time between restarts of the probe. You must compile this program and place it in the same directory as the callback methods, that is, the directory pointed to by the `RT_basedir` property.

EXAMPLE 8 `gettime.c` Utility Program

```
# This utility program, used by the probe method of the data service, tracks
# the elapsed time in seconds from a known reference point (epoch point). It
# must be compiled and placed in the same directory as the data service callback
# methods (RT_basedir).

#pragma ident "@(#)gettime.c 1.1 12/01/24"

#include <stdio.h>
#include <sys/types.h>
#include <time.h>

main()
{
    printf("%d\n", time(0));
    exit(0);
}
```

PROBE Program Code Listing

The `PROBE` program checks the availability of the data service by using `nslookup` commands (see the [nslookup\(1M\)](#) man page). The `Monitor_start` callback method starts this program, and the `Monitor_stop` callback method stops it.

EXAMPLE 9 `dns_probe` Program

```
#!/bin/ksh
#pragma ident "@(#)dns_probe 1.1 12/01/19"
```



```

#
# Probe method for HA-DNS.
#
# This program checks the availability of the data service using nslookup, which
# queries the DNS server to look for the DNS server itself. If the server
# does not respond or if the query is replied to by some other server,
# then the probe concludes that there is some problem with the data service
# and fails the service over to another node in the cluster. Probing is done
# at a specific interval set by THOROUGH_PROBE_INTERVAL in the RTR file.

#pragma ident "@(#)dns_probe 1.1 00/05/24 SMI"

#####
# Parse program arguments.
function parse_args # [args ...]
{
typeset opt

while getopts `R:G:T:` opt
do
case "$opt" in
R)
# Name of the DNS resource.
RESOURCE_NAME=$OPTARG
;;
G)
# Name of the resource group in which the resource is
# configured.
RESOURCEGROUP_NAME=$OPTARG
;;
T)
# Name of the resource type.
RESOURCETYPE_NAME=$OPTARG
;;
*)
logger -p ${SYSLOG_FACILITY}.err \
-t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME] \
"ERROR: Option $OPTARG unknown"
exit 1
;;
esac
done
}

#####
# restart_service ()
#
# This function tries to restart the data service by calling the Stop method

```

```

# followed by the Start method of the dataservice. If the dataservice has
# already died and no tag is registered for the dataservice under PMF,
# then this function fails the service over to another node in the cluster.
#
function restart_service
{
# To restart the dataservice, first, verify that the
# dataservice itself is still registered under PMF.
pmfadm -q $PMF_TAG
if [[ $? -eq 0 ]]; then
# Since the TAG for the dataservice is still registered under
# PMF, first stop the dataservice and start it back up again.
# Obtain the Stop method name and the STOP_TIMEOUT value for
# this resource.
STOP_TIMEOUT=`scha_resource_get -O STOP_TIMEOUT \
-R $RESOURCE_NAME -G $RESOURCEGROUP_NAME`
STOP_METHOD=`scha_resource_get -O STOP \
-R $RESOURCE_NAME -G $RESOURCEGROUP_NAME`
/usr/cluster/bin/hatimerun -t $STOP_TIMEOUT $RT_BASEDIR/$STOP_METHOD \
-R $RESOURCE_NAME -G $RESOURCEGROUP_NAME \
-T $RESOURCE_TYPE_NAME

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

# Obtain the Start method name and the START_TIMEOUT value for
# this resource.
START_TIMEOUT=`scha_resource_get -O START_TIMEOUT \
-R $RESOURCE_NAME -G $RESOURCEGROUP_NAME`
START_METHOD=`scha_resource_get -O START \
-R $RESOURCE_NAME -G $RESOURCEGROUP_NAME`
/usr/cluster/bin/hatimerun -t $START_TIMEOUT $RT_BASEDIR/$START_METHOD \
-R $RESOURCE_NAME -G $RESOURCEGROUP_NAME \
-T $RESOURCE_TYPE_NAME

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

else
# The absence of the TAG for the dataservice
# implies that the dataservice has already
# exceeded the maximum retries allowed under PMF.

```

```

# Therefore, do not attempt to restart the
# dataservice again, but try to failover
# to another node in the cluster.
scha_control -O GIVEOVER -G $RESOURCEGROUP_NAME \
-R $RESOURCE_NAME
fi

return 0
}

#####
# decide_restart_or_failover ()
#
# This function decides the action to be taken upon the failure of a probe:
# restart the data service locally or fail over to another node in the cluster.
#
function decide_restart_or_failover
{

# Check if this is the first restart attempt.
if [ $retries -eq 0 ]; then
# This is the first failure. Note the time of
# this first attempt.
start_time=`$RT_BASEDIR/gettime`
retries=`expr $retries + 1`
# Because this is the first failure, attempt to restart
# the data service.
restart_service
if [ $? -ne 0 ]; then
logger -p ${SYSLOG_FACILITY}.err -t [SYSLOG_TAG] \
"${ARGV0} Failed to restart data service."
exit 1
fi
else
# This is not the first failure
current_time=`$RT_BASEDIR/gettime`
time_diff=`expr $current_time - $start_time`
if [ $time_diff -ge $RETRY_INTRVAL ]; then
# This failure happened after the time window
# elapsed, so reset the retries counter,
# slide the window, and do a retry.
retries=1
start_time=$current_time
# Because the previous failure occurred more than
# Retry_interval ago, attempt to restart the data service.
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
# Still within the time window,
# and the retry counter expired, so fail over.
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
# Still within the time window,
# and the retry counter has not expired,
# so do another retry.
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

# Obtain the syslog facility to use to log messages.
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# Parse the arguments that have been passed to this method
parse_args "$@"

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

# The interval at which probing is to be done is set in the system defined
# property THOROUGH_PROBE_INTERVAL. Obtain the value of this property with
# scha_resource_get
PROBE_INTERVAL=scha_resource_get -O THOROUGH_PROBE_INTERVAL \

```

```
-R $RESOURCE_NAME -G $RESOURCEGROUP_NAME

# Obtain the timeout value allowed for the probe, which is set in the
# PROBE_TIMEOUT extension property in the RTR file. The default timeout for
# nslookup is 1.5 minutes.
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}``

# Identify the server on which DNS is serving by obtaining the value
# of the NETWORK_RESOURCES_USED property of the resource.
DNS_HOST=`scha_resource_get -O NETWORK_RESOURCES_USED -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAME`

# Get the retry count value from the system defined property Retry_count
RETRY_COUNT=`scha_resource_get -O RETRY_COUNT -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAME`

# Get the retry interval value from the system defined property
Retry_interval
RETRY_INTRVAL=scha_resource_get -O RETRY_INTRVAL -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAME`

# Obtain the full path for the gettime utility from the
# RT_basedir property of the resource type.
RT_BASEDIR=scha_resource_get -O RT_basedir -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAME`

# The probe runs in an infinite loop, trying nslookup commands.
# Set up a temporary file for the nslookup replies.
DNSPROBEFILE=/tmp/.$RESOURCE_NAME.probe
probefail=0
retries=0

while :
do
# The interval at which the probe needs to run is specified in the
# property THOROUGH_PROBE_INTERVAL. Therefore, set the probe to sleep for a
# duration of <THOROUGH_PROBE_INTERVAL>
sleep $PROBE_INTERVAL

# Run the probe, which queries the IP address on
# which DNS is serving.
/usr/cluster/bin/hatimerun -t $PROBE_TIMEOUT /usr/sbin/nslookup $DNS_HOST $DNS_HOST \
> $DNSPROBEFILE 2>&1

retcode=$?
if [ retcode -ne 0 ]; then
```

```
probefail=1
fi

# Make sure that the reply to nslookup command comes from the HA-DNS
# server and not from another name server listed in the
# /etc/resolv.conf file.
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

# If the probefail variable is not set to 0, either the nslookup command
# timed out or the reply to the query was came from another server
# (specified in the /etc/resolv.conf file). In either case, the DNS server is
# not responding and the method calls decide_restart_or_failover,
# which evaluates whether to restart the data service or to fail it over
# to another node.

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 Method Code Listing

This method starts the PROBE program for the data service.

EXAMPLE 10 dns_monitor_start Method

```
#!/bin/ksh
#
# Monitor start Method for HA-DNS.
#
```

```

# This method starts the monitor (probe) for the data service under the
# control of PMF. The monitor is a process that probes the data service
# at periodic intervals and if there is a problem restarts it on the same node
# or fails it over to another node in the cluster. The PMF tag for the
# monitor is $RESOURCE_NAME.monitor.

#pragma ident "@(#)dns_monitor_start 1.1 12/01/24"

#####
# Parse program arguments.
#
function parse_args # [args ...]
{
typeset opt

while getopts `R:G:T:` opt
do
case "$opt" in
R)
# Name of the DNS resource.
RESOURCE_NAME=$OPTARG
;;
G)
# Name of the resource group in which the resource is
# configured.
RESOURCEGROUP_NAME=$OPTARG
;;
T)
# Name of the resource type.
RESOURCETYPE_NAME=$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

```

```

# Obtain the syslog facility to use to log messages.
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# Parse the arguments that have been passed to this method
parse_args "$@"

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

# Find where the probe method resides by obtaining the value of the
# RT_basedir property of the data service.
RT_BASEDIR=`scha_resource_get -O RT_basedir -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAME`

# Start the probe for the data service under PMF. Use the infinite retries
# option to start the probe. Pass the resource name, group, and type to the
# probe method.
pmfadm -c $PMF_TAG.monitor -n -1 -t -1 \
$RT_BASEDIR/dns_probe -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME \
-T $RESOURCE_TYPE_NAME

# Log a message indicating that the monitor for HA-DNS has been started.
if [ $? -eq 0 ]; then
logger -p ${SYSLOG_FACILITY}.info -t [SYSLOG_TAG] \
"${ARGV0} Monitor for HA-DNS successfully started"
fi
exit 0

```

Monitor_stop Method Code Listing

This method stops the PROBE program for the data service.

EXAMPLE 11 dns_monitor_stop Method

```

#!/bin/ksh
# Monitor stop method for HA-DNS
# Stops the monitor that is running using PMF.

#pragma ident "@(#)dns_monitor_stop 1.1 12/01/24 "

#####
# Parse program arguments.
#
function parse_args # [args ...]

```



```

{
typeset opt

while getopts `R:G:T:` opt
do
case "$opt" in
R)
# Name of the DNS resource.
RESOURCE_NAME=$OPTARG
;;
G)
# Name of the resource group in which the resource is
# configured.
RESOURCEGROUP_NAME=$OPTARG
;;
T)
# Name of the resource type.
RESOURCETYPE_NAME=$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

# Obtain the syslog facility to use to log messages.
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# Parse the arguments that have been passed to this method
parse_args "$@"

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

# See if the monitor is running, and if so, kill it.
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
# Could successfully stop the monitor. Log a message.
logger -p ${SYSLOG_FACILITY}.info -t [SYSLOG_TAG] \
"${ARGV0} Monitor for resource " $RESOURCE_NAME \
" successfully stopped"
fi
fi
exit 0
```

Monitor_check Method Code Listing

This method verifies the existence of the directory that is pointed to by the `Confdir` property. The RGM calls `Monitor_check` when the `PROBE` method fails over the data service to a new node. The RGM also does so to check nodes that are potential masters.

EXAMPLE 12 dns_monitor_check Method

```
#!/bin/ksh#
# Monitor check Method for DNS.
#
# The RGM calls this method whenever the fault monitor fails the data service
# over to a new node. Monitor_check calls the Validate method to verify
# that the configuration directory and files are available on the new node.

#pragma ident "@(#)dns_monitor_check 1.1 12/01/24"

#####
# Parse program arguments.
function parse_args # [args ...]
{
typeset opt

while getopts `R:G:T:` opt
do
case "$opt" in

R)
# Name of the DNS resource.
RESOURCE_NAME=$OPTARG
```

```

;;

G)
# Name of the resource group in which the resource is
# configured.
RESOURCEGROUP_NAME=$OPTARG
;;

T)
# Name of the resource type.
RESOURCE_TYPE_NAME=$OPTARG
;;

*)
logger -p ${SYSLOG_FACILITY}.err \
-t [$RESOURCE_TYPE_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

# Obtain the syslog facility to use to log messages.
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# Parse the arguments that have been passed to this method.
parse_args "$@"

PMF_TAG=$RESOURCE_NAME.named
SYSLOG_TAG=$RESOURCE_TYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME

# Obtain the full path for the Validate method from
# the RT_basedir property of the resource type.
RT_BASEDIR=`scha_resource_get -O RT_basedir -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAME`

# Obtain the name of the Validate method for this resource.
VALIDATE_METHOD=`scha_resource_get -O VALIDATE -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAME`

```

```

# Obtain the value of the Confdir property in order to start the
# data service. Use the resource name and the resource group entered to
# obtain the Confdir value set at the time of adding the resource.
config_info=`scha_resource_get -O Extension -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAME Confdir`

# scha_resource_get returns the type as well as the value for extension
# properties. Use awk to get only the value of the extension property.
CONFIG_DIR=`echo $config_info | awk '{print $2}'`

# Call the validate method so that the dataservice can be failed over
# successfully to the new node.
$RT_BASEDIR/$VALIDATE_METHOD -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME \
-T $RESOURCETYPE_NAME -x Confdir=$CONFIG_DIR

# Log a message indicating that monitor check was successful.
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 Method Code Listing

This method verifies the existence of the directory that is pointed to by the `Confdir` property. The RGM calls this method when the data service is created. The RGM also calls this method when the cluster administrator updates the data service properties. The `Monitor_check` method calls this method whenever the fault monitor fails over the data service to a new node.

EXAMPLE 13 `dns_validate` Method

```

#!/bin/ksh
# Validate method for HA-DNS.
# This method validates the Confdir property of the resource. The Validate
# method gets called in two scenarios. When the resource is being created and
# when a resource property is getting updated. When the resource is being
# created, this method gets called with the -c flag and all the system-defined
# and extension properties are passed as command-line arguments. When a resource
# property is being updated, the Validate method gets called with the -u flag,
# and only the property/value pair of the property being updated is passed as a

```

```

# command-line argument.
#
# ex: When the resource is being created command args will be
#
# dns_validate -c -R <.> -G <.> -T <.> -r <sysdef-prop=value>...
#       -x <extension-prop=value>.... -g <resourcegroup-prop=value>....
#
# when the resource property is being updated
#
# dns_validate -u -R <.> -G <.> -T <.> -r <sys-prop_being_updated=value>
# OR
# dns_validate -u -R <.> -G <.> -T <.> -x <extn-prop_being_updated=value>

#pragma ident "@(#)dns_validate 1.1 12/01/24 "

#####
# Parse program arguments.
#
function parse_args # [args ...]
{
typeset opt

while getopts `cur:x:g:R:T:G:X:` opt
do
case "$opt" in
R)
# Name of the DNS resource.
RESOURCE_NAME=$OPTARG
;;
G)
# Name of the resource group in which the resource is
# configured.
RESOURCEGROUP_NAME=$OPTARG
;;
T)
# Name of the resource type.
RESOURCETYPE_NAME=$OPTARG
;;
X)
# Per-node extension property setting. The format of the
# option argument is "propertyname{nodeid}=propertyvalue".
# For example, there might be two -X options with the following
# arguments:
#   myprop{1}=myvalue
#   myprop{2}=othervalue
# representing the setting of property 'myprop' on nodes 1
# and 2.
# In most cases, the -X arguments can be ignored. Instead

```

```

# use the -x argument to get the property setting for the
# local node.
;;
r)
#The method is not accessing any system defined
#properties, so this is a no-op.
;;
g)
# The method is not accessing any resource group
# properties, so this is a no-op.
;;
c)
# Indicates the Validate method is being called while
# creating the resource, so this flag is a no-op.
;;
u)
# Indicates the updating of a property when the
# resource already exists. If the update is to the
# Confdir property then Confdir should appear in the
# command-line arguments. If it does not, the method must
# look for it specifically using scha_resource_get.
UPDATE_PROPERTY=1
;;
x)
# Extension property list. Separate the property and
# value pairs using "=" as the separator.
PROPERTY=`echo $OPTARG | awk -F= `{print $1}``
VAL=`echo $OPTARG | awk -F= `{print $2}``

# If the Confdir extension property is found on the
# command line, note its value.
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

# Obtain the syslog facility to use to log messages.
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# Set the Value of CONFDIR to null. Later, this method retrieves the value
# of the Confdir property from the command line or using scha_resource_get.
CONFDIR=""
UPDATE_PROPERTY=0
CONFDIR_FOUND=0

# Parse the arguments that have been passed to this method.
parse_args "$@"

# If the validate method is being called due to the updating of properties
# try to retrieve the value of the Confdir extension property from the command
# line. Otherwise, obtain the value of Confdir using scha_resource_get.
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

# Verify that the Confdir property has a value. If not there is a failure
# and exit with status 1.
if [[ -z $CONFDIR ]]; then
logger -p ${SYSLOG_FACILITY}.err \
"${ARGV0} Validate method for resource "$RESOURCE_NAME " failed"
exit 1
fi

# Now validate the actual Confdir property value.

# Check if $CONFDIR is accessible.
if [ ! -d $CONFDIR ]; then
logger -p ${SYSLOG_FACILITY}.err -t [SYSLOG_TAG] \
"${ARGV0} Directory $CONFDIR missing or not mounted"
exit 1
fi

# Check that the named.conf file is present in the Confdir directory.
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

# Log a message indicating that the Validate method was successful.
logger -p ${SYSLOG_FACILITY}.info -t [SYSLOG_TAG] \
"${ARGV0} Validate method for resource "$RESOURCE_NAME \
" completed successfully"

exit 0
```

Update Method Code Listing

The RGM calls the Update method to notify a running resource that its properties have been changed.

EXAMPLE 14 dns_update Method

```
#!/bin/ksh
# Update method for HA-DNS.
# The actual updates to properties are done by the RGM. Updates affect only
# the fault monitor so this method must restart the fault monitor.

#pragma ident "@(#)dns_update 1.1 12/01/24"

#####
# Parse program arguments.
#
function parse_args # [args ...]
{
typeset opt

while getopts `R:G:T:` opt
do
case "$opt" in
R)
# Name of the DNS resource.
RESOURCE_NAME=$OPTARG
;;
G)
# Name of the resource group in which the resource is
# configured.
RESOURCEGROUP_NAME=$OPTARG
;;
T)

```



```

# Name of the resource type.
RESOURCE_TYPE_NAME=$OPTARG
;;
*)
logger -p ${SYSLOG_FACILITY}.err \
-t [$RESOURCE_TYPE_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

# Obtain the syslog facility to use to log messages.
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# Parse the arguments that have been passed to this method
parse_args "$@"

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

# Find where the probe method resides by obtaining the value of the
# RT_basedir property of the resource.
RT_BASEDIR=`scha_resource_get -O RT_basedir -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAME`

# When the Update method is called, the RGM updates the value of the property
# being updated. This method must check if the fault monitor (probe)
# is running, and if so, kill it and then restart it.
if pmfadm -q $PMF_TAG.monitor; then

# Kill the monitor that is running already
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
# Could successfully stop DNS. Log a message.
logger -p ${SYSLOG_FACILITY}.info -t [$SYSLOG_TAG] \
"Monitor for HA-DNS successfully stopped"
fi
# Restart the monitor.

```

```
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
```

DSDL Sample Resource Type Code Listings

This appendix lists the complete code for each method in the `ORCL.xfnts` resource type. It includes the listing for `xfnts.c`, which contains code for the subroutines that are called by the callback methods. [Chapter 8, “Sample DSDL Resource Type Implementation”](#) describes the sample resource type `ORCL.xfnts` in more detail.

This appendix covers the following topics:

- [“xfnts.c File Listing” on page 259](#)
- [“xfnts_monitor_check Method Code Listing” on page 272](#)
- [“xfnts_monitor_start Method Code Listing” on page 273](#)
- [“xfnts_monitor_stop Method Code Listing” on page 274](#)
- [“xfnts_probe Method Code Listing” on page 275](#)
- [“xfnts_start Method Code Listing” on page 278](#)
- [“xfnts_stop Method Code Listing” on page 280](#)
- [“xfnts_update Method Code Listing” on page 281](#)
- [“xfnts_validate Method Code Listing” on page 283](#)

`xfnts.c` File Listing

This file implements the subroutines that are called by the `ORCL.xfnts` methods.

EXAMPLE 15 `xfnts.c`

```
/*
 * Copyright (c) 1998, 2012, by Oracle and/or its affiliates.
 * All rights reserved.
 *
 * xfnts.c - Common utilities for HA-XFS
 *
 * This utility has the methods for performing the validation, starting and
```

```
* stopping the data service and the fault monitor. It also contains the method
* to probe the health of the data service. The probe just returns either
* success or failure. Action is taken based on this returned value in the
* method found in the file xfnts_probe.c
*
*/
```

```
#pragma ident "@(#)xfnts.c 1.47 12/01/18"
```

```
#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"
```

```
/*
 * The initial timeout allowed for the HAXFS data service to
 * be fully up and running. We will wait for 3 % (SVC_WAIT_PCT)
 * of the start_timeout time before probing the service.
 */
```

```
#define SVC_WAIT_PCT 3
```

```
/*
 * We need to use 95% of probe_timeout to connect to the port and the
 * remaining time is used to disconnect from port in the svc_probe function.
 */
```

```
#define SVC_CONNECT_TIMEOUT_PCT 95
```

```
/*
 * SVC_WAIT_TIME is used only during starting in svc_wait().
 * In svc_wait() we need to be sure that the service is up
 * before returning, thus we need to call svc_probe() to
 * monitor the service. SVC_WAIT_TIME is the time between
 * such probes.
 */
```

```
#define SVC_WAIT_TIME 5
```

```
/*
 * This value will be used as disconnect timeout, if there is no
 * time left from the probe_timeout.
 */
```

```
*/
#define SVC_DISCONNECT_TIMEOUT_SECONDS 2

/*
 * svc_validate():
 *
 * Do HA-XFS specific validation of the resource configuration.
 *
 * svc_validate will check for the following
 * 1. Confdir_list extension property
 * 2. fontserver.cfg file
 * 3. xfs binary
 * 4. port_list property
 * 5. network resources
 * 6. other extension properties
 *
 * If any of the above validation fails then, Return > 0 otherwise return 0 for
 * success
 */

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;

    /*
     * Get the configuration directory for the XFS dataservice from the
     * confdir_list extension property.
     */
    confdirs = scds_get_ext_confdir_list(scds_handle);

    /* Return an error if there is no confdir_list extension property */
    if (confdirs == NULL || confdirs->array_cnt != 1) {
        scds_syslog(LOG_ERR,
            "Property Confdir_list is not set properly.");
        return (1); /* Validation failure */
    }

    /*
     * Construct the path to the configuration file from the extension
     * property confdir_list. Since HA-XFS has only one configuration

```

```
* we will need to use the first entry of the confdir_list property.
*/
(void) sprintf(xfnts_conf, "%s/fontserver.cfg", confdirs->str_array[0]);

/*
 * Check to see if the HA-XFS configuration file is in the right place.
 * Try to access the HA-XFS configuration file and make sure the
 * permissions are set properly
 */
if (stat(xfnts_conf, &statbuf) != 0) {
/*
 * suppress lint error because errno.h prototype
 * is missing void arg
 */
scds_syslog(LOG_ERR,
"Failed to access file <%=s> : <%=s>",
xfnts_conf, strerror(errno)); /*lint !e746 */
return (1);
}

/*
 * Make sure that xfs binary exists and that the permissions
 * are correct. The XFS binary are assumed to be on the local
 * File system and not on the Global File System
 */
if (stat("/usr/bin/xfs", &statbuf) != 0) {
scds_syslog(LOG_ERR,
"Cannot access XFS binary : <%=s> ", strerror(errno));
return (1);
}

/* HA-XFS will have only port */
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); /* Validation Failure */
}

#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); /* Validation Failure */
}
#endif
#endif
```

```
/*
 * Return an error if there is an error when trying to get the
 * available network address resources for this resource
 */
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); /* Validation Failure */
}

/* Return an error if there are no network address resources */
if (snrlp == NULL || snrlp->num_netresources == 0) {
    scds_syslog(LOG_ERR,
        "No network address resource in resource group.");
    rc = 1;
    goto finished;
}

/* Check to make sure other important extension props are set */
if (scds_get_ext_monitor_retry_count(scds_handle) <= 0)
{
    scds_syslog(LOG_ERR,
        "Property Monitor_retry_count is not set.");
    rc = 1; /* Validation Failure */
    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; /* Validation Failure */
    goto finished;
}

/* All validation checks were successful */
scds_syslog(LOG_INFO, "Successful validation.");
rc = 0;

finished:
scds_free_net_list(snrlp);
scds_free_port_list(portlist);

return (rc); /* return result of validation */
}

/*
```

```
* svc_start():
*
* Start up the X font server
* Return 0 on success, > 0 on failures.
*
* The XFS service will be started by running the command
* /usr/bin/xfns -config <fontserver.cfg file> -port <port to listen>
* XFS will be started under PMF. XFS will be started as a single instance
* service. The PMF tag for the data service will be of the form
* <resourcegroupname,resourcenname,instance_number.svc>. In case of XFS, since
* there will be only one instance the instance_number in the tag will be 0.
*/

int
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;

/* get the configuration directory from the confdir_list property */
confdirs = scds_get_ext_confdir_list(scds_handle);

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

/* obtain the port to be used by XFS from the Port_list property */
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);
}

/*
* Construct the command to start HA-XFS.
* NOTE: XFS daemon prints the following message while stopping the XFS
* "/usr/bin/xfns notice: terminating"
* In order to suppress the daemon message,
* the output is redirected to /dev/null.
*/
(void) sprintf(cmd,
"/usr/bin/xfns -config %s -port %d 2>/dev/null",
xfnts_conf, portlist->ports[0].port);

/*
* Start HA-XFS under PMF. Note that HA-XFS is started as a single
```



```
* instance service. The last argument to the scds_pmf_start function
* denotes the level of children to be monitored. A value of -1 for
* this parameter means that all the children along with the original
* process are to be monitored.
*/
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);
return (err); /* return Success/failure status */
}

/*
* svc_stop():
*
* Stop the XFS server
* Return 0 on success, > 0 on failures.
*
* svc_stop will stop the server by calling the toolkit function:
* scds_pmf_stop.
*/
int
svc_stop(scds_handle_t scds_handle)
{
scha_err_t err;

/*
* The timeout value for the stop method to succeed is set in the
* Stop_Timeout (system defined) property
*/
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); /* Successfully stopped */
}

/*
 * svc_wait():
 *
 * wait for the data service to start up fully and make sure it is running
 * healthy
 */

int
svc_wait(scds_handle_t scds_handle)
{
int rc, svc_start_timeout, probe_timeout;
scds_netaddr_list_t *netaddr;

/* obtain the network resource to use for probing */
if (scds_get_netaddr_list(scds_handle, &netaddr) {
scds_syslog(LOG_ERR,
"No network address resources found in resource group.");
return (1);
}

/* Return an error if there are no network resources */
if (netaddr == NULL || netaddr->num_netaddrs == 0) {
scds_syslog(LOG_ERR,
"No network address resource in resource group.");
return (1);
}

/*
 * Get the Start method timeout, port number on which to probe,
 * the Probe timeout value
 */
svc_start_timeout = scds_get_rs_start_timeout(scds_handle);
probe_timeout = scds_get_ext_probe_timeout(scds_handle);

/*
 * sleep for SVC_WAIT_PCT percentage of start_timeout time
 * before actually probing the dataservice. This is to allow
 * the dataservice to be fully up in order to reply to the
 * probe. NOTE: the value for SVC_WAIT_PCT could be different
 * for different data services.
```

```
* Instead of calling sleep(),
* call scds_svc_wait() so that if service fails too
* many times, we give up and return early.
*/
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 {
    /*
    * probe the data service on the IP address of the
    * network resource and the portname
    */
    rc = svc_probe(scds_handle,
        netaddr->netaddrs[0].hostname,
        netaddr->netaddrs[0].port_proto.port, probe_timeout);
    if (rc == SCHA_ERR_NOERR) {
        /* Success. Free up resources and return */
        scds_free_netaddr_list(netaddr);
        return (0);
    }

    /*
    * Dataservice is still trying to come up. Sleep for a while
    * before probing again. Instead of calling sleep(),
    * call scds_svc_wait() so that if service fails too
    * many times, we give up and return early.
    */
    if (scds_svc_wait(scds_handle, SVC_WAIT_TIME)
        != SCHA_ERR_NOERR) {
        scds_syslog(LOG_ERR, "Service failed to start.");
        return (1);
    }

    /* We rely on RGM to timeout and terminate the program */
} while (1);

}

/*
* This function starts the fault monitor for a HA-XFS resource.
* This is done by starting the probe under PMF. The PMF tag
* is derived as <RG-name,RS-name,instance_number.mon>. The restart option
* of PMF is used but not the "infinite restart". Instead
* interval/retry_time is obtained from the RTR file.
```

```
*/

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));

    /*
     * The probe xfnts_probe is assumed to be available in the same
     * subdirectory where the other callback methods for the RT are
     * installed. The last parameter to scds_pmf_start denotes the
     * child monitor level. Since we are starting the probe under PMF
     * we need to monitor the probe process only and hence we are using
     * a value of 0.
     */
    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); /* Successfully started Monitor */
}

/*
 * This function stops the fault monitor for a HA-XFS resource.
 * This is done via PMF. The PMF tag for the fault monitor is
 * constructed based on <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); /* Successfully stopped monitor */
}

/*
 * svc_probe(): Do data service specific probing. Return a float value
 * between 0 (success) and 100(complete failure).
 *
 * The probe does a simple socket connection to the XFS server on the specified
 * port which is configured as the resource extension property (Port_list) and
 * pings the dataservice. If the probe fails to connect to the port, we return
 * a value of 100 indicating that there is a total failure. If the connection
 * goes through and the disconnect to the port fails, then a value of 50 is
 * returned indicating a partial failure.
 */
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;

/*
 * probe the dataservice by doing a socket connection to the port
 * specified in the port_list property to the host that is
 * serving the XFS dataservice. If the XFS service which is configured
 * to listen on the specified port, replies to the connection, then
 * the probe is successful. Else we will wait for a time period set
 * in probe_timeout property before concluding that the probe failed.
 */
```

```
*/

/*
 * Use the SVC_CONNECT_TIMEOUT_PCT percentage of timeout
 * to connect to the port
 */
connect_timeout = (SVC_CONNECT_TIMEOUT_PCT * timeout)/100;
t1 = (hrtime_t)(gethrtime()/1E9);

/*
 * the probe makes a connection to the specified hostname and port.
 * The connection is timed for 95% of the actual probe_timeout.
 */
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));
/* this is a complete failure */
return (SCDS_PROBE_COMPLETE_FAILURE);
}

t2 = (hrtime_t)(gethrtime()/1E9);

/*
 * Compute the actual time it took to connect. This should be less than
 * or equal to connect_timeout, the time allocated to connect.
 * If the connect uses all the time that is allocated for it,
 * then the remaining value from the probe_timeout that is passed to
 * this function will be used as disconnect timeout. Otherwise, the
 * the remaining time from the connect call will also be added to
 * the disconnect timeout.
 *
 */

time_used = (int)(t2 - t1);

/*
 * Use the remaining time(timeout - time_took_to_connect) to disconnect
 */

time_remaining = timeout - (int)time_used;

/*
 * If all the time is used up, use a small hardcoded timeout
 * to still try to disconnect. This will avoid the fd leak.
 */
```

```
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;
}

/*
 * Return partial failure in case of disconnection failure.
 * Reason: The connect call is successful, which means
 * the application is alive. A disconnection failure
 * could happen due to a hung application or heavy load.
 * If it is the later case, don't declare the application
 * as dead by returning complete failure. Instead, declare
 * it as partial failure. If this situation persists, the
 * disconnect call will fail again and the application will be
 * restarted.
 */
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));
/* this is a partial failure */
return (SCDS_PROBE_COMPLETE_FAILURE/2);
}

t2 = (hrtime_t)(gethrtime()/1E9);
time_used = (int)(t2 - t1);
time_remaining = timeout - time_used;

/*
 * If there is no time left, don't do the full test with
 * fsinfo. Return SCDS_PROBE_COMPLETE_FAILURE/2
 * instead. This will make sure that if this timeout
 * persists, server will be restarted.
 */
if (time_remaining <= 0) {
scds_syslog(LOG_ERR, "Probe timed out.");
return (SCDS_PROBE_COMPLETE_FAILURE/2);
}
```

```
/*
 * The connection and disconnection to port is successful,
 * Run the fsinfo command to perform a full check of
 * server health.
 * Redirect stdout, otherwise the output from fsinfo
 * ends up on the console.
 */
(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 Method Code Listing

This method verifies that the basic resource type configuration is valid.

EXAMPLE 16 xfnts_monitor_check.c

```
/*
 * Copyright (c) 1998, 2012, Oracle and/or its affiliates.
 * All rights reserved.
 *
 * xfnts_monitor_check.c - Monitor Check method for HA-XFS
 */

#pragma ident "@(#)xfnts_monitor_check.c 1.11 01/01/18"

#include <rgm/libdsdev.h>
#include "xfnts.h"

/*
 * just make a simple validate check on the service
 */
```



```

int
main(int argc, char *argv[])
{
    scds_handle_t    scds_handle;
    int    rc;

    /* Process the arguments passed by RGM and initialize syslog */
    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);

    /* Free up all the memory allocated by scds_initialize */
    scds_close(&scds_handle);

    /* Return the result of validate method run as part of monitor check */
    return (rc);
}

```

xfnts_monitor_start Method Code Listing

This method starts the xfnts_probe method.

EXAMPLE 17 xfnts_monitor_start.c

```

/*
 * Copyright (c) 1998, 2012, Oracle and/or its affiliates.
 * All rights reserved.
 *
 * xfnts_monitor_start.c - Monitor Start method for HA-XFS
 */

#pragma ident "@(#)xfnts_monitor_start.c 1.10 12/01/18"

#include <rgm/libdsdev.h>
#include "xfnts.h"

/*
 * This method starts the fault monitor for a HA-XFS resource.

```

```
* This is done by starting the probe under PMF. The PMF tag
* is derived as RG-name,RS-name.mon. The restart option of PMF
* is used but not the "infinite restart". Instead
* interval/retry_time is obtained from the RTR file.
*/

int
main(int argc, char *argv[])
{
    scds_handle_t    scds_handle;
    int    rc;

    /* Process arguments passed by RGM and initialize syslog */
    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);

    /* Free up all the memory allocated by scds_initialize */
    scds_close(&scds_handle);

    /* Return the result of monitor_start method */
    return (rc);
}
```

xfnts_monitor_stop Method Code Listing

This method stops the xfnts_probe method.

EXAMPLE 18 xfnts_monitor_stop.c

```
/*
 * Copyright (c) 1998, 2012, Oracle and/or its affiliates.
 * All rights reserved.
 *
 * xfnts_monitor_stop.c - Monitor Stop method for HA-XFS
 */

#pragma ident "@(#)xfnts_monitor_stop.c 1.9 12/01/18"

#include <rgm/libdsdev.h>
#include "xfnts.h"
```

```

/*
 * This method stops the fault monitor for a HA-XFS resource.
 * This is done via PMF. The PMF tag for the fault monitor is
 * constructed based on RG-name_RS-name.mon.
 */

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

scds_handle_t   scds_handle;
int             rc;

/* Process arguments passed by RGM and initialize syslog */
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);

/* Free up all the memory allocated by scds_initialize */
scds_close(&scds_handle);

/* Return the result of monitor stop method */
return (rc);
}

```

xfnts_probe Method Code Listing

The `xfnts_probe` method checks the availability of the application and determines whether to fail over or restart the data service. The `xfnts_monitor_start` callback method starts this program, and the `xfnts_monitor_stop` callback method stops it.

EXAMPLE 19 `xfnts_probe.c`

```

/*
 * Copyright (c) 1998, 2012, Oracle and/or its affiliates.
 * All rights reserved.
 *
 * xfnts_probe.c - Probe for HA-XFS
 */

```

```
#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>
#include <sys/time.h>
#include <sys/socket.h>
#include <strings.h>
#include <rgm/libdsdev.h>
#include "xfnts.h"

/*
 * main():
 * Just an infinite loop which sleep()s for sometime, waiting for
 * the PMF action script to interrupt the sleep(). When interrupted
 * It calls the start method for HA-XFS to restart it.
 *
 */

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);
    }

    /* Get the ip addresses available for this resource */
    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);
    }
}
```

```
/* Return an error if there are no network resources */
if (netaddr == NULL || netaddr->num_netaddrs == 0) {
scds_syslog(LOG_ERR,
"No network address resource in resource group.");
return (1);
}

/*
 * Set the timeout from the X props. This means that each probe
 * iteration will get a full timeout on each network resource
 * without chopping up the timeout between all of the network
 * resources configured for this resource.
 */
timeout = scds_get_ext_probe_timeout(scds_handle);

for (;;) {

/*
 * sleep for a duration of thorough_probe_interval between
 * successive probes.
 */
(void) scds_fm_sleep(scds_handle,
scds_get_rs_thorough_probe_interval(scds_handle));

/*
 * Now probe all ipaddress we use. Loop over
 * 1. All net resources we use.
 * 2. All ipaddresses in a given resource.
 * For each of the ipaddress that is probed,
 * compute the failure history.
 */
probe_result = 0;
/*
 * Iterate through the all resources to get each
 * IP address to use for calling svc_probe()
 */
for (ip = 0; ip < netaddr->num_netaddrs; ip++) {
/*
 * Grab the hostname and port on which the
 * health has to be monitored.
 */
hostname = netaddr->netaddrs[ip].hostname;
port = netaddr->netaddrs[ip].port_proto.port;
/*
 * HA-XFS supports only one port and
 * hence obtain the port value from the
 * first entry in the array of ports.
 */
}
```

```
*/
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);

/*
 * Update service probe history,
 * take action if necessary.
 * Latch probe end time.
 */
ht2 = gethrtime();

/* Convert to milliseconds */
dt = (ulong_t)((ht2 - ht1) / 1e6);

/*
 * Compute failure history and take
 * action if needed
 */
(void) scds_fm_action(scds_handle,
probe_result, (long)dt);
} /* Each net resource */
} /* Keep probing forever */
}
```

xfnts_start Method Code Listing

The RGM runs the Start method on a cluster node when the resource group that contains the data service resource is brought online on that node. The RGM also does so when the resource is enabled. The `xfnts_start` method activates the `xfns` daemon on that node.

EXAMPLE 20 `xfnts_start.c`

```
/*
 * Copyright (c) 1998, 2012, Oracle and/or its affiliates.
 * All rights reserved.
 *
 * xfnts_svc_start.c - Start method for HA-XFS
 */

#pragma ident "@(#)xfnts_svc_start.c 1.13 01/01/18 SMI"
```

```
#include <rgm/libdsdev.h>
#include "xfnts.h"

/*
 * The start method for HA-XFS. Does some sanity checks on
 * the resource settings then starts the HA-XFS under PMF with
 * an action script.
 */

int
main(int argc, char *argv[])
{
    scds_handle_t    scds_handle;
    int rc;

    /*
     * Process all the arguments that have been passed to us from RGM
     * and do some initialization for syslog
     */

    if (scds_initialize(&scds_handle, argc, argv) != SCHA_ERR_NOERR)
    {
        scds_syslog(LOG_ERR, "Failed to initialize the handle.");
        return (1);
    }

    /* Validate the configuration and if there is an error return back */
    rc = svc_validate(scds_handle);
    if (rc != 0) {
        scds_syslog(LOG_ERR,
            "Failed to validate configuration.");
        return (rc);
    }

    /* Start the data service, if it fails return with an error */
    rc = svc_start(scds_handle);
    if (rc != 0) {
        goto finished;
    }

    /* Wait for the service to start up fully */
    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:
/* Free up the Environment resources that were allocated */
scds_close(&scds_handle);

return (rc);
}
```

xfnts_stop Method Code Listing

The RGM runs the Stop method on a cluster node when the resource group that contains the HA-XFS resource is brought offline on that node. The RGM also does so when the resource is disabled. This method stops the xfs daemon on that node.

EXAMPLE 21 xfnts_stop.c

```
/*
 * Copyright (c) 1998, 2012, Oracle and/or its affiliates.
 * All rights reserved.
 *
 * xfnts_svc_stop.c - Stop method for HA-XFS
 */

#pragma ident "@(#)xfnts_svc_stop.c 1.10 01/01/18 SMI"

#include <rgm/libdsdev.h>
#include "xfnts.h"

/*
 * Stops the HA-XFS process using PMF
 */

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



```

scds_handle_t   scds_handle;
int             rc;

/* Process the arguments passed by RGM and initialize syslog */
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);

/* Free up all the memory allocated by scds_initialize */
scds_close(&scds_handle);

/* Return the result of svc_stop method */
return (rc);
}

```

xfnts_update Method Code Listing

The RGM calls the Update method to notify a running resource that its properties have been changed. The RGM runs Update after an administrative action succeeds in setting properties of a resource or its group.

EXAMPLE 22 xfnts_update.c

```

#pragma ident "@(#)xfnts_update.c 1.10 01/01/18 SMI"

/*
 * Copyright (c) 1998, 2012, Oracle and/or its affiliates.
 * All rights reserved.
 *
 * xfnts_update.c - Update method for HA-XFS
 */

#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <rgm/libdsdev.h>

/*
 * Some of the resource properties might have been updated. All such
 * updatable properties are related to the fault monitor. Hence, just

```

```
* restarting the monitor should be enough.
*/

int
main(int argc, char *argv[])
{
    scds_handle_t    scds_handle;
    scha_err_t      result;

    /* Process the arguments passed by RGM and initialize syslog */
    if (scds_initialize(&scds_handle, argc, argv) != SCHA_ERR_NOERR)
    {
        scds_syslog(LOG_ERR, "Failed to initialize the handle.");
        return (1);
    }

    /*
     * check if the Fault monitor is already running and if so stop and
     * restart it. The second parameter to scds_pmf_restart_fm() uniquely
     * identifies the instance of the fault monitor that needs to be
     * restarted.
     */

    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.");
        /* Free up all the memory allocated by scds_initialize */
        scds_close(&scds_handle);
        return (1);
    }

    scds_syslog(LOG_INFO,
        "Completed successfully.");

    /* Free up all the memory allocated by scds_initialize */
    scds_close(&scds_handle);

    return (0);
}
```

xfnts_validate Method Code Listing

This method verifies the existence of the directory that is pointed to by the `Confdir_list` property. The RGM calls this method when the data service is created and when data service properties are updated by the cluster administrator. The `Monitor_check` method calls this method whenever the fault monitor fails over the data service to a new node.

EXAMPLE 23 `xfnts_validate.c`

```

/*
 * Copyright (c) 1998, 2012, Oracle and/or its affiliates.
 * All rights reserved.
 *
 * xfnts_validate.c - validate method for HA-XFS
 */

#pragma ident "@(#)xfnts_validate.c 1.9 01/01/18 SMI"

#include <rgm/libdsdev.h>
#include "xfnts.h"

/*
 * Check to make sure that the properties have been set properly.
 */

int
main(int argc, char *argv[])
{
    scds_handle_t    scds_handle;
    int    rc;

    /* Process arguments passed by RGM and initialize syslog */
    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);

    /* Free up all the memory allocated by scds_initialize */
    scds_close(&scds_handle);

    /* Return the result of validate method */
    return (rc);
}

```


Requirements for Non-Cluster-Aware Applications

An ordinary, non-cluster-aware application must meet particular requirements to be a candidate for high availability (HA). The section [“Analyzing the Application for Suitability” on page 25](#) lists these requirements. This appendix provides additional details about particular items in that list.

An application is made highly available by configuring its resources into resource groups. The application's data is placed on a highly available cluster file system, making the data accessible by a surviving server in the event that one server fails. See information about cluster file systems in the [Oracle Solaris Cluster 4.3 Concepts Guide](#) .

For network access by clients on the network, a logical network IP address is configured in logical host name resources that are contained in the same resource group as the data service resource. The data service resource and the network address resources fail over together, causing network clients of the data service to access the data service resource on its new host.

This appendix covers the following topics:

- [“Multihosted Data” on page 285](#)
- [“Host Names” on page 287](#)
- [“Multihomed Hosts” on page 287](#)
- [“Binding to INADDR_ANY as Opposed to Binding to Specific IP Addresses” on page 288](#)
- [“Client Retry” on page 289](#)

Multihosted Data

The highly available cluster file systems' devices are multihosted so that when a physical host crashes, one of the surviving hosts can access the device. For an application to be highly available, its data must be highly available. Therefore, the application's data must be located in file systems that can be accessed from multiple cluster nodes. Local file systems that you can

make highly available with Oracle Solaris Cluster include UNIX File System (UFS) , Quick File System (QFS), and Oracle Solaris ZFS.

The cluster file system is mounted on device groups that are created as independent entities. You can choose to use some device groups as mounted cluster file systems and others as raw devices for use with a data service, such as HA for Oracle.

An application might have command-line switches or configuration files that point to the location of the data files. If the application uses hard-wired path names, you could change the path names to symbolic links that point to a file in a cluster file system, without changing the application code. See [“Using Symbolic Links for Multihosted Data Placement” on page 286](#) for a more detailed discussion about using symbolic links.

In the worst case, the application's source code must be modified to provide a mechanism for pointing to the actual data location. You could implement this mechanism by creating additional command-line arguments.

The Oracle Solaris Cluster software supports the use of UNIX UFS and ZFS file systems and HA raw devices that are configured in a volume manager. When installing and configuring the Oracle Solaris Cluster software, the cluster administrator must specify which disk resources to use for UFS or ZFS file systems and which disk resources to use for raw devices. Typically, raw devices are used only by database servers and multimedia servers.

Using Symbolic Links for Multihosted Data Placement

Occasionally, the path names of an application's data files are hard-wired, with no mechanism for overriding the hard-wired path names. To avoid modifying the application code, you can sometimes use symbolic links.

For example, suppose the application names its data file with the hard-wired path name `/etc/mydatafile`. You can change that path from a file to a symbolic link that has its value pointing to a file in one of the logical host's file systems. For example, you can make the path a symbolic link to `/global/phys-schost-2/mydatafile`.

A problem can occur with this use of symbolic links if the application, or one of its administrative procedures, modifies the data file name as well as its contents. For example, suppose that the application performs an update by first creating a new temporary file `/etc/mydatafile.new`. Then, the application renames the temporary file to have the real file name by using the `rename()` system call (or the `mv` command). By creating the temporary file and renaming it to the real file name, the data service is attempting to ensure that its data file contents are always well formed.

Unfortunately, the `rename()` action destroys the symbolic link. The name `/etc/mydatafile` is now a regular file and is in the same file system as the `/etc` directory, not in the cluster's cluster file system. Because the `/etc` file system is private to each host, the data is not available after a failover or switchover.

The underlying problem is that the existing application is not aware of the symbolic link and was not written to handle symbolic links. To use symbolic links to redirect data access into the logical host's file systems, the application implementation must behave in a way that does not obliterate the symbolic links. So, symbolic links are not a complete remedy for the problem of placing data in the cluster's file systems.

Host Names

You must determine whether the data service ever needs to know the host name of the server on which it is running. If so, the data service might need to be modified to use a logical host name, rather than the physical host name. In this sense, a logical host name is a host name that is configured into a logical host name resource that is located in the same resource group as the application resource.

Occasionally, in the client-server protocol for a data service, the server returns its own host name to the client as part of the contents of a message to the client. For such protocols, the client could be depending on this returned host name as the host name to use when contacting the server. For the returned host name to be usable after a failover or switchover, the host name should be a logical host name of the resource group, not the name of the physical host. In this case, you must modify the data service code to return the logical host name to the client.

Multihomed Hosts

The term *multihomed host* describes a host that is located on more than one public network. Such a host has multiple host names and IP addresses. It has one host name–IP address pair for each network. Oracle Solaris Cluster is designed to permit a host to appear on any number of networks, including just one (the non-multihomed case). Just as the physical host name has multiple host name–IP address pairs, each resource group can have multiple host name–IP address pairs, one for each public network. When Oracle Solaris Cluster moves a resource group from one physical host to another physical host, the complete set of host name–IP address pairs for that resource group is moved.

The set of host name–IP address pairs for a resource group is configured as logical host name resources contained in the resource group. These network address resources are specified by

the cluster administrator when the resource group is created and configured. The Oracle Solaris Cluster Data Service API contains facilities for querying these host name–IP address pairs.

Most off-the-shelf data service daemons that have been written for the Oracle Solaris Operating System already handle multihomed hosts correctly. Many data services do all their network communication by binding to the Oracle Solaris wildcard address `INADDR_ANY`. This binding automatically causes the data services to handle all the IP addresses for all the network interfaces. `INADDR_ANY` effectively binds to all IP addresses that are currently configured on the machine. A data service daemon that uses `INADDR_ANY` generally does not need to be changed to handle the Oracle Solaris Cluster logical network addresses.

Binding to `INADDR_ANY` as Opposed to Binding to Specific IP Addresses

Even when non-multihomed hosts are used, the Oracle Solaris Cluster logical network address concept enables the machine to have more than one IP address. The machine has one IP address for its own physical host, and additional IP addresses for each network address (logical host name or shared address) resource that it currently masters. When a machine becomes the master of a network address resource, it dynamically acquires additional IP addresses. When it gives up mastery of a network address resource, it dynamically relinquishes IP addresses.

Some data services cannot work correctly in an Oracle Solaris Cluster environment if they bind to `INADDR_ANY`. For example, if an application needs to use the logical address as the source for outbound traffic, the deprecated flag set by the network address resource will prevent this if the application calls `bind(3SOCKET)` with `INADDR_ANY` as the address. For such an application to work properly in a clustered environment, it must be able to be configured to explicitly bind (`3SOCKET`) to the logical address, which overrides the deprecated flag when the source address for outbound traffic is selected. The application must implement this feature, because the cluster framework cannot alter application behavior.

When the application resource and the network address resource reside in different resource groups, the `Resource_dependencies` property lets you ensure a specific set of network address resources are available when the application resource eventually calls `bind(3SOCKET)`. When the application resource declares a dependency upon the network address resource, Oracle Solaris Cluster starts the network address before starting the application, so that the application that is binding to the network address will not fail to find it. The resource dependency also ensures that when the resource group is going offline, the application is stopped before stopping the network address.

Without a resource dependency declared, the order in which the application and network address resources are started or stopped is nondeterministic.

Network address resource types include `SUNW.LogicalHostname` and `SUNW.SharedAddress`.

To force the application resource group and the network address resource group to always start on the same node, declare a strong positive affinity with delegated failover (a `+++` affinity) of the application resource group upon the network address resource group. For more information, see the “`RG_affinities`” section in the [rg_properties\(5\)](#) man page.

If the application resource and the network address resource are configured in the same resource group, then it is usually not necessary to declare an explicit resource dependency. By default there will be an implicit resource dependency of the application upon the network address. For more information, see the “`Implicit_network_dependencies`” section in the [rg_properties\(5\)](#) man page.

Client Retry

To a network client, a failover or switchover appears to be a crash of the logical host followed by a fast reboot. Ideally, the client application and the client-server protocol are structured to do some amount of retrying. If the application and protocol already handle the case of a single server crashing and rebooting, they can also handle the case of the resource group being taken over or switched over. Some applications might elect to retry endlessly. More sophisticated applications notify the user that a long retry is in progress and enable the user to choose whether to continue.

Document Type Definitions for the CRNP

This appendix includes the following document type definitions (DTDs) for the Cluster Reconfiguration Notification Protocol (CRNP):

- [“SC_CALLBACK_REG XML DTD” on page 291](#)
- [“NVPAIR XML DTD” on page 293](#)
- [“SC_REPLY XML DTD” on page 294](#)
- [“SC_EVENT XML DTD” on page 295](#)

SC_CALLBACK_REG XML DTD

Note - The NVPAIR data structure that is used by both SC_CALLBACK_REG and SC_EVENT is defined only once.

```
<!-- SC_CALLBACK_REG XML format specification
Copyright 2001, 2012, Oracle and/or its affiliates. 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, 2012, Oracle and/or its affiliates. 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 Oracle 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, 2012, Oracle and/or its affiliates. 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

Note - The NVPAIR data structure that is used by both SC_CALLBACK_REG and SC_EVENT is defined only once.

<!-- SC_EVENT XML format specification

Copyright 2001, 2012, Oracle and/or its affiliates. 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 Oracle Solaris syseventd 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
>

◆◆◆ APPENDIX E

CrnpClient.java Application

This appendix shows the complete `CrnpClient.java` application that is discussed in more detail in [Chapter 11, “Cluster Reconfiguration Notification Protocol”](#).

Contents of `CrnpClient.java`

```
/*
 * CrnpClient.java
 * =====
 *
 * Note regarding XML parsing:
 *
 * This program uses the Java Architecture for XML Processing (JAXP) API.
 * See https://jaxp.java.net for API documentation and
 * availability information.
 *
 * This program was written for Java 1.3.1 or higher.
 *
 * Program overview:
 *
 * The main thread of the program creates a CrnpClient object, waits for the
 * user to terminate the demo, then calls shutdown on the CrnpClient object
 * and exits the program.
 *
 * The CrnpClient constructor creates an EventReceptionThread object,
 * opens a connection to the CRNP server (using the host and port specified
 * on the command line), constructs a registration message (based on the
 * command-line specifications), sends the registration message, and reads
 * and parses the reply.
 *
 * The EventReceptionThread creates a listening socket bound to
 * the hostname of the machine on which this program runs, and the port
 * specified on the command line. It waits for an incoming event callback,
 * at which point it constructs an XML Document from the incoming socket
 * stream, which is then passed back to the CrnpClient object to process.
```

```

*
* The shutdown method in the CrnpClient just sends an unregistration
* (REMOVE_CLIENT) SC_CALLBACK_REG message to the crnp server.
*
* Note regarding error handling: for the sake of brevity, this program just
* exits on most errors. Obviously, a real application would attempt to handle
* some errors in various ways, such as retrying when appropriate.
*/

// JAXP packages
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.*;

// standard packages
import java.net.*;
import java.io.*;
import java.util.*;

/*
 * class CrnpClient
 * -----
 * See file header comments above.
 */
class CrnpClient
{
/*
 * main
 * ----
 * The entry point of the execution, main simply verifies the
 * number of command-line arguments, and constructs an instance
 * of a CrnpClient to do all the work.
 */
public static void main(String []args)
{
    InetAddress regIp = null;
    int regPort = 0, localPort = 0;

    /* Verify the number of command-line arguments */
    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);
}

/*
 * We expect the command line to contain the ip/port of the
 * crnp server, the local port on which we should listen, and
 * arguments specifying the type of registration.
 */
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);
}

// Create the CrnpClient
CrnpClient client = new CrnpClient(regIp, regPort, localPort,
args);

// Now wait until the user wants to end the program
System.out.println("Hit return to terminate demo...");

// read will block until the user enters something
try {
    System.in.read();
} catch (IOException e) {
    System.out.println(e.toString());
}

// shutdown the client
client.shutdown();
System.exit(0);
}

/*
 * =====
 * public methods
 * =====
 */

/*
 * CrnpClient constructor
 * -----
 * Parses the command line arguments so we know how to contact
 * the crnp server, creates the event reception thread, and starts it
```

```

* running, creates the XML DocumentBuilderFactory object, and, finally,
* registers for callbacks with the crnp server.
*/
public CrnpClient(InetAddress regIpIn, int regPortIn, int localPortIn,
String []clArgs)
{
try {

regIp = regIpIn;
regPort = regPortIn;
localPort = localPortIn;
regs = clArgs;

/*
* Setup the document builder factory for
* xml processing.
*/
setupXmlProcessing();

/*
* Create the EventReceptionThread, which creates a
* ServerSocket and binds it to a local ip and port.
*/
createEvtRecepThr();

/*
* Register with the crnp server.
*/
registerCallbacks();

} catch (Exception e) {
System.out.println(e.toString());
System.exit(1);
}

/*
* processEvent
* -----
* Callback into the CrnpClient, used by the EventReceptionThread
* when it receives event callbacks.
*/
public void processEvent(Event event)
{
/*
* For demonstration purposes, simply print the event
* to System.out. A real application would obviously make
* use of the event in some way.

```

```
*/
event.print(System.out);
}

/*
 * shutdown
 * -----
 * Unregister from the CRNP server.
 */
public void shutdown()
{
    try {
        /* send an unregistration message to the server */
        unregister();
    } catch (Exception e) {
        System.out.println(e);
        System.exit(1);
    }
}

/*
 * =====
 * private helper methods
 * =====
 */

/*
 * setupXmlProcessing
 * -----
 * Create the document builder factory for
 * parsing the xml replies and events.
 */
private void setupXmlProcessing() throws Exception
{
    dbf = DocumentBuilderFactory.newInstance();

    // We don't need to bother validating
    dbf.setValidating(false);
    dbf.setExpandEntityReferences(false);

    // We want to ignore comments and whitespace
    dbf.setIgnoringComments(true);
    dbf.setIgnoringElementContentWhitespace(true);

    // Coalesce CDATA sections into TEXT nodes.
    dbf.setCoalescing(true);
}
```

```
/*
 * createEvtRecepThr
 * -----
 * Creates a new EventReceptionThread object, saves the ip
 * and port to which its listening socket is bound, and
 * starts the thread running.
 */
private void createEvtRecepThr() throws Exception
{
    /* create the thread object */
    evtThr = new EventReceptionThread(this);

    /*
     * Now start the thread running to begin listening
     * for event delivery callbacks.
     */
    evtThr.start();
}

/*
 * registerCallbacks
 * -----
 * Creates a socket connection to the crnp server and sends
 * an event registration message.
 */
private void registerCallbacks() throws Exception
{
    System.out.println("About to register");

    /*
     * Create a socket connected to the registration ip/port
     * of the crnp server and send the registration information.
     */
    Socket sock = new Socket(regIp, regPort);
    String xmlStr = createRegistrationString();
    PrintStream ps = new PrintStream(sock.getOutputStream());
    ps.print(xmlStr);

    /*
     * Read the reply
     */
    readRegistrationReply(sock.getInputStream());

    /*
     * Close the socket connection.
     */
}
```

```
sock.close();
}

/*
 * unregister
 * -----
 * As in registerCallbacks, we create a socket connection to
 * the crnp server, send the unregistration message, wait for
 * the reply from the server, then close the socket.
 */
private void unregister() throws Exception
{
    System.out.println("About to unregister");

    /*
     * Create a socket connected to the registration ip/port
     * of the crnp server and send the unregistration information.
     */
    Socket sock = new Socket(regIp, regPort);
    String xmlStr = createUnregistrationString();
    PrintStream ps = new PrintStream(sock.getOutputStream());
    ps.print(xmlStr);

    /*
     * Read the reply
     */
    readRegistrationReply(sock.getInputStream());

    /*
     * Close the socket connection.
     */
    sock.close();
}

/*
 * createRegistrationString
 * -----
 * Constructs a CallbackReg object based on the command line arguments
 * to this program, then retrieves the XML string from the CallbackReg
 * object.
 */
private String createRegistrationString() throws Exception
{
    /*
     * create the actual CallbackReg class and set the port.
     */
    CallbackReg cbReg = new CallbackReg();
    cbReg.setPort("" + localPort);
}
```

```
// set the registration type
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);
}

// add the events
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
 * -----
 * Creates an XML registration event with class EC_Cluster, and no
 * subclass.
 */
private Event createAllEvent()
{
    Event allEvent = new Event();
    allEvent.setClass("EC_Cluster");
    return (allEvent);
}

/*
 * createMembershipEvent
 * -----
 * Creates an XML registration event with class EC_Cluster, subclass

```



```
* ESC_cluster_membership.
*/
private Event createMembershipEvent()
{
Event membershipEvent = new Event();
membershipEvent.setClass("EC_Cluster");
membershipEvent.setSubclass("ESC_cluster_membership");
return (membershipEvent);
}

/*
* createRgEvent
* -----
* Creates an XML registration event with class EC_Cluster,
* subclass ESC_cluster_rg_state, and one "rg_name" nvpair (based
* on input parameter).
*/
private Event createRgEvent(String rgname)
{
/*
* Create a Resource Group state change event for the
* rgname Resource Group. Note that we supply
* a name/value pair (nvpair) for this event type, to
* specify in which Resource Group we are interested.
*/
/*
* Construct the event object and set the class and subclass.
*/
Event rgStateEvent = new Event();
rgStateEvent.setClass("EC_Cluster");
rgStateEvent.setSubclass("ESC_cluster_rg_state");

/*
* Create the nvpair object and add it to the Event.
*/
NVPair rgNvpair = new NVPair();
rgNvpair.setName("rg_name");
rgNvpair.setValue(rgname);
rgStateEvent.addNvpair(rgNvpair);

return (rgStateEvent);
}

/*
* createREvent
* -----
* Creates an XML registration event with class EC_Cluster,
* subclass ESC_cluster_r_state, and one "r_name" nvpair (based
```

```
* on input parameter).
*/
private Event createREvent(String rname)
{
    /*
    * Create a Resource state change event for the
    * rname Resource. Note that we supply
    * a name/value pair (nvpair) for this event type, to
    * specify in which Resource Group we are interested.
    */
    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
* -----
* Constructs a REMOVE_CLIENT CallbackReg object, then retrieves
* the XML string from the CallbackReg object.
*/
private String createUnregistrationString() throws Exception
{
    /*
    * Create the CallbackReg object.
    */
    CallbackReg cbReg = new CallbackReg();
    cbReg.setPort("" + localPort);
    cbReg.setRegType(CallbackReg.REMOVE_CLIENT);

    /*
    * we marshall the registration to the OutputStream
    */
    String xmlStr = cbReg.convertToXml();

    // Print the string for debugging purposes
    System.out.println(xmlStr);
    return (xmlStr);
}
```

```

/*
 * readRegistrationReply
 * -----
 * Parse the xml into a Document, construct a RegReply object
 * from the document, and print the RegReply object. Note that
 * a real application would take action based on the status_code
 * of the RegReply object.
 */
private void readRegistrationReply(InputStream stream)
throws Exception
{
    // Create the document builder
    DocumentBuilder db = dbf.newDocumentBuilder();

    //
    // Set an ErrorHandler before parsing
    // Use the default handler.
    //
    db.setErrorHandler(new DefaultHandler());

    //parse the input file
    Document doc = db.parse(stream);

    RegReply reply = new RegReply(doc);
    reply.print(System.out);
}

/* private member variables */
private InetAddress regIp;
private int regPort;
private EventReceptionThread evtThr;
private String regs[];

/* public member variables */
public int localPort;
public DocumentBuilderFactory dbf;
}

/*
 * class EventReceptionThread
 * -----
 * See file header comments above.
 */
class EventReceptionThread extends Thread
{
    /*
     * EventReceptionThread constructor
     * -----

```

```

* Creates a new ServerSocket, bound to the local hostname and
* a wildcard port.
*/
public EventReceptionThread(CrnpClient clientIn) throws IOException
{
    /*
    * keep a reference to the client so we can call it back
    * when we get an event.
    */
    client = clientIn;

    /*
    * Specify the IP to which we should bind. It's
    * simply the local host ip. If there is more
    * than one public interface configured on this
    * machine, we'll go with whichever one
    * InetAddress.getLocalHost comes up with.
    */
    listeningSock = new ServerSocket(client.localPort, 50,
    InetAddress.getLocalHost());
    System.out.println(listeningSock);
}

/*
* run
* ---
* Called by the Thread.Start method.
*
* Loops forever, waiting for incoming connections on the ServerSocket.
*
* As each incoming connection is accepted, an Event object
* is created from the xml stream, which is then passed back to
* the CrnpClient object for processing.
*/
public void run()
{
    /*
    * Loop forever.
    */
    try {
        //
        // Create the document builder using the document
        // builder factory in the CrnpClient.
        //
        DocumentBuilder db = client.dbf.newDocumentBuilder();

        //

```

```
// Set an ErrorHandler before parsing
// Use the default handler.
//
db.setErrorHandler(new DefaultHandler());

while(true) {
    /* wait for a callback from the server */
    Socket sock = listeningSock.accept();

    // parse the input file
    Document doc = db.parse(sock.getInputStream());

    Event event = new Event(doc);
    client.processEvent(event);

    /* close the socket */
    sock.close();
}
// UNREACHABLE

} catch (Exception e) {
    System.out.println(e);
    System.exit(1);
}
}

/* private member variables */
private ServerSocket listeningSock;
private CrnpClient client;
}

/*
 * class NVPair
 * -----
 * This class stores a name/value pair (both Strings). It knows how to
 * construct an NVPAIR XML message from its members, and how to parse
 * an NVPAIR XML Element into its members.
 *
 * Note that the formal specification of an NVPAIR allows for multiple values.
 * We make the simplifying assumption of only one value.
 */
class NVPair
{
    /*
     * Two constructors: the first creates an empty NVPair, the second
     * creates an NVPair from an NVPAIR XML Element.
     */
    public NVPair()
```

```

    {
    name = value = null;
    }

    public NVPair(Element elem)
    {
    retrieveValues(elem);
    }

    /*
    * Public setters.
    */
    public void setName(String nameIn)
    {
    name = nameIn;
    }

    public void setValue(String valueIn)
    {
    value = valueIn;
    }

    /*
    * Prints the name and value on a single line.
    */
    public void print(PrintStream out)
    {
    out.println("NAME=" + name + " VALUE=" + value);
    }

    /*
    * createXmlElement
    * -----
    * Constructs an NVPAIR XML Element from the member variables.
    * Takes the Document as a parameter so that it can create the
    * Element.
    */
    public Element createXmlElement(Document doc)
    {
    // Create the element.
    Element nvpair = (Element)
    doc.createElement("NVPAIR");
    //
    // Add the name. Note that the actual name is
    // a separate CDATA section.
    //
    Element eName = doc.createElement("NAME");
    Node nameData = doc.createCDATASection(name);

```

```
eName.appendChild(nameData);
nvpair.appendChild(eName);
//
// Add the value. Note that the actual value is
// a separate CDATA section.
//
Element eValue = doc.createElement("VALUE");
Node valueData = doc.createCDATASection(value);
eValue.appendChild(valueData);
nvpair.appendChild(eValue);

return (nvpair);
}

/*
 * retrieveValues
 * -----
 * Parse the XML Element to retrieve the name and value.
 */
private void retrieveValues(Element elem)
{
    Node n;
    NodeList nl;

    //
    // Find the NAME element
    //
    nl = elem.getElementsByTagName("NAME");
    if (nl.getLength() != 1) {
        System.out.println("Error in parsing: can't find "
            + "NAME node.");
        return;
    }

    //
    // Get the TEXT section
    //
    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;
    }

    // Retrieve the value
    name = n.getNodeValue();

    //

```

```
// Now get the value element
//
n1 = elem.getElementsByTagName("VALUE");
if (n1.getLength() != 1) {
System.out.println("Error in parsing: can't find "
+ "VALUE node.");
return;
}

//
// Get the TEXT section
//
n = n1.item(0).getFirstChild();
if (n == null || n.getNodeType() != Node.TEXT_NODE) {
System.out.println("Error in parsing: can't find "
+ "TEXT section.");
return;
}

// Retrieve the value
value = n.getNodeValue();
}

/*
 * Public accessors
 */
public String getName()
{
return (name);
}

public String getValue()
{
return (value);
}

// Private member vars
private String name, value;
}

/*
 * class Event
 * -----
 * This class stores an event, which consists of a class, subclass, vendor,
 * publisher, and list of name/value pairs. It knows how to
 * construct an SC_EVENT_REG XML Element from its members, and how to parse
```



```
* an SC_EVENT XML Element into its members. Note that there is an assymetry
* here: we parse SC_EVENT elements, but construct SC_EVENT_REG elements.
* That is because SC_EVENT_REG elements are used in registration messages
* (which we must construct), while SC_EVENT elements are used in event
* deliveries (which we must parse). The only difference is that SC_EVENT_REG
* elements don't have a vendor or publisher.
*/
class Event
{

    /*
    * Two constructors: the first creates an empty Event; the second
    * creates an Event from an SC_EVENT XML Document.
    */
    public Event()
    {
        regClass = regSubclass = null;
        nvpairs = new Vector();
    }

    public Event(Document doc)
    {

        nvpairs = new Vector();

        //
        // Convert the document to a string to print for debugging
        // purposes.
        //
        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());

    // Do the actual parsing.
    retrieveValues(doc);
}

/*
* Public setters.
```

```
*/
public void setClass(String classIn)
{
    regClass = classIn;
}

public void setSubclass(String subclassIn)
{
    regSubclass = subclassIn;
}

public void addNvpair(NVPair nvpair)
{
    nvpairs.add(nvpair);
}

/*
 * createXmlElement
 * -----
 * Constructs an SC_EVENT_REG XML Element from the member variables.
 * Takes the Document as a parameter so that it can create the
 * Element. Relies on the NVPair createXmlElement ability.
 */
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);
}

/*
 * Prints the member vars on multiple lines.
 */
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
 * -----
 * Parse the XML Document to retrieve the class, subclass, vendor,
 * publisher, and nvpairs.
 */
private void retrieveValues(Document doc)
{
    Node n;
    NodeList nl;

    //
    // Find the SC_EVENT element.
    //
    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);

    //
    // Retrieve the values of the CLASS, SUBCLASS,
    // VENDOR and PUBLISHER attributes.
    //
    regClass = ((Element)n).getAttribute("CLASS");
    regSubclass = ((Element)n).getAttribute("SUBCLASS");
    publisher = ((Element)n).getAttribute("PUBLISHER");
    vendor = ((Element)n).getAttribute("VENDOR");

    //
    // Retrieve all the nv pairs
    //
    for (Node child = n.getFirstChild(); child != null;
    child = child.getNextSibling())
    {
        nvpairs.add(new NVPair((Element)child));
    }
}
```

```

    }

    /*
    * Public accessor methods.
    */
    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 member vars.
    private String regClass, regSubclass;
    private Vector nvpairs;
    private String vendor, publisher;
    }

    /*
    * class CallbackReg
    * -----
    * This class stores a port and regType (both Strings), and a list of Events.
    * It knows how to construct an SC_CALLBACK_REG XML message from its members.
    *
    * Note that this class does not need to be able to parse SC_CALLBACK_REG
    * messages, because only the CRNP server must parse SC_CALLBACK_REG
    * messages.
    */
    class CallbackReg

```

```
{
// Useful defines for the setRegType method
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 setters.
*/
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
    * -----
    * Constructs an SC_CALLBACK_REG XML Document from the member
    * variables. Relies on the Event createXmlElement ability.
    */
    public String convertToXml()
    {
        Document document = null;
        DocumentBuilderFactory factory =
        DocumentBuilderFactory.newInstance();
        try {
            DocumentBuilder builder = factory.newDocumentBuilder();
            document = builder.newDocument();
        } catch (ParserConfigurationException pce) {
            // Parser with specified options can't be built
            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);

        //
        // Now convert the document to a string.
        //
        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 member vars
private String port;
private String regType;
private Vector regEvents;
}

/*
 * class RegReply
 * -----
 * This class stores a status_code and status_msg (both Strings).
 * It knows how to parse an SC_REPLY XML Element into its members.
 */
class RegReply
{

/*
 * The only constructor takes an XML Document and parses it.
 */
public RegReply(Document doc)
{
//
// Now convert the document to a string.
//
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 accessors
 */
public String getStatusCode()
{
return (statusCode);
}
```

```
public String getStatusMsg()
{
    return (statusMsg);
}

/*
 * Prints the info on a single line.
 */
public void print(PrintStream out)
{
    out.println(statusCode + ": " +
        (statusMsg != null ? statusMsg : ""));
}

/*
 * retrieveValues
 * -----
 * Parse the XML Document to retrieve the statusCode and statusMsg.
 */
private void retrieveValues(Document doc)
{
    Node n;
    NodeList nl;

    //
    // Find the SC_REPLY element.
    //
    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);

    // Retrieve the value of the STATUS_CODE attribute
    statusCode = ((Element)n).getAttribute("STATUS_CODE");

    //
    // Find the SC_STATUS_MSG element
    //
    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;
    }
}
```



```
    }

    //
    // Get the TEXT section, if there is one.
    //
    n = nl.item(0).getFirstChild();
    if (n == null || n.getNodeType() != Node.TEXT_NODE) {
        // Not an error if there isn't one, so we
        // just silently return.
        return;
    }

    // Retrieve the value
    statusMsg = n.getNodeValue();
}

// private member vars
private String statusCode;
private String statusMsg;
}
```


Index

Numbers and Symbols

- #\$upgrade directive, 77
- #\$upgrade_from directive, 77
 - ANYTIME, 77
 - AT_CREATION, 78
 - tunability values, 77
 - WHEN_DISABLED, 78
 - WHEN_OFFLINE, 78
 - WHEN_UNMANAGED, 78
 - WHEN_UNMONITORED, 78
- (Resource Type Registration)
 - file
 - upgrading, 76

A

- accessing network address
 - with DSDL, 124
- administrative interface
 - RGM (Resource Group Manager), 22
- Agent Builder
 - analyzing the application, 167
 - binary files, 182
 - cloning existing resource type, 178
 - command-line version, 180
 - Configure screen, 173
 - configuring, 168
 - Create screen, 172
 - description, 16, 22
 - directory structure, 180
 - editing generated source code, 179
 - installing, 168
 - man pages, 183

- navigating in, 169
 - Browse, 170
 - Edit menu, 171
 - File menu, 171
 - menus, 171
- package directory, 184
- reusing code, 178
- rtconfig file, 185
- scripts, 183
- source files, 182
- starting, 169
- support files, 184
- using, 167

ANYTIME

- #\$upgrade_from directive, 77
- API, Resource Management *See* RMAPI
- application environment
 - Oracle Solaris Cluster, 15
- arguments
 - RMAPI method, 67
- arraymax
 - resource type upgrade, 75
- arraymin
 - resource type upgrade, 75
- AT_CREATION
 - #\$upgrade_from directive, 78
- attributes
 - resource property, 31

B

- binary files
 - Agent Builder, 182

bind(3SOCKET), 288
Binding to INADDR_ANY, 288
Boot method, using, 47, 71
Browse
 Agent Builder, 170

C

C program functions
 RMAPI, 63
callback methods
 control, 68
 description, 19
 initialization, 68
 Monitor_check, 73
 Monitor_start, 72
 Monitor_stop, 72
 naming conventions, 145
 overview, 15
 Postnet_start, 72
 Prenet_start, 72
 RMAPI, 67
 Update, 72
 using, 51
 Validate, 71
checks
 validating for scalable services, 55
client
 CRNP, 197
cloning existing resource type
 Agent Builder, 178
clsetup
 description, 23
cluster commands
 RMAPI, 63
cluster functions
 RMAPI, 66
cluster properties, 30
 Concentrate_load, 30
Cluster Reconfiguration Notification Protocol *See*
CRNP
code

 changing method, 81
 changing monitor, 81
codes
 RMAPI exit, 68
command line
 Agent Builder, 180
 commands on, 24
commands
 clsetup, 23
 halockrun, 51
 hatimerun, 51
 Oracle Solaris Cluster, 24
 RMAPI resource type, 62
components
 RMAPI, 21
Concentrate_load
 resource property, 30
concepts
 CRNP, 193
Configure screen
 Agent Builder, 173
configuring
 Agent Builder, 168
conventions
 callback method names, 145
 function names, 145
Create screen
 Agent Builder, 172
CRNP (Cluster Reconfiguration Notification Protocol)
 authentication, 203
 client, 197
 client identification process, 197
 communication, 195
 concepts, 193
 description, 194
 error conditions, 200
 example Java application, 204
 function of, 194
 message types, 196
 registration of client and server, 197
 SC_CALLBACK_REG messages, 197
 SC_EVENT, 201, 202
 SC_REPLY, 199, 200

- semantics of protocol, 195
 - server, 197
 - server event delivery, 201
 - server reply, 199
- D**
- daemon
 - designing the fault monitor, 135
 - data service
 - creating
 - analyzing suitability, 25
 - determining the interface, 27
 - sample, 85
 - common functionality, 92
 - controlling the data service, 97
 - defining a fault monitor, 103
 - extension properties in RTR file, 91
 - generating error messages, 95
 - handling property updates, 113
 - Monitor_check method, 112
 - Monitor_start method, 109
 - Monitor_stop method, 110
 - obtaining property information, 96
 - probe program, 103
 - resource properties in RTR file, 88
 - RTR file, 87
 - Start method, 97
 - Stop method, 100
 - Update method, 118
 - Validate method, 113
 - setting up development environment, 28
 - transferring to cluster for testing, 30
 - Data Service Development Library *See* DSDL
 - data services
 - testing, 56
 - testing HA, 57
 - writing, 56
 - debugging resource types with DSDL, 124
 - default property values
 - new value for upgrade, 79
 - when inherited, 79
 - dependencies
 - coordinating between resources, 57
 - directive
 - #\$upgrade, 77
 - #\$upgrade_from, 77, 77
 - default tunability, 78
 - placement in RTR file, 77
 - RT_version, 77
 - tunability constraints, 77
 - directories
 - Agent Builder, 184
 - directory structure
 - Agent Builder, 180
 - distinguishing between multiple registered versions
 - rt-version, 76
 - distinguishing between vendors
 - vendor-id, 76
 - documentation requirements
 - for upgrade, 82
 - tunability constraints, 82
 - DSDL (Data Service Development Library)
 - accessing network address, 124
 - components, 21
 - debugging resource types, 124
 - description, 121, 121
 - enabling HA local file systems, 124
 - fault monitor functions, 192
 - fault monitoring, 191
 - general purpose functions, 187
 - implementing a fault monitor, 123
 - libdsdev.so, 16
 - network resource access functions, 189
 - overview, 16
 - Process Monitor Facility (PMF) functions, 191
 - property functions, 189
 - sample resource type implementation
 - determining the fault monitor action, 160
 - ORCL.xfnts fault monitor, 154
 - ORCL.xfnts RTR file, 145
 - returning from svc_start(), 148
 - scds_initialize() function, 146
 - starting the service, 147
 - svc_probe() function, 157
 - TCP port number, 144

- validating the service, 146
 - X font server, 143
 - X font server configuration file, 144
 - xfnts_monitor_check method, 154
 - xfnts_monitor_start method, 152
 - xfnts_monitor_stop method, 153
 - xfnts_probe main loop, 155
 - xfnts_start method, 146
 - xfnts_stop method, 151
 - xfnts_update method, 163
 - xfnts_validate method, 160
 - starting a data service, 122
 - stopping a data service, 122
 - utility functions, 192
 - where implemented, 16
- E**
- editing generated Agent Builder source code, 179
 - enabling HA local file systems with DSDL, 124
 - error conditions
 - CRNP, 200
 - events
 - guaranteed delivery, 202
 - examples
 - data service, 85
 - Java application that uses CRNP, 204
 - exit codes
 - RMAPI, 68
 - extension properties
 - declaring, 39
- F**
- failover resource
 - implementing, 52
 - fault monitor
 - daemon
 - designing the, 135
 - functions, DSDL, 192
 - ORCL.xfnts, 154
 - files
 - binary in Agent Builder, 182
 - rtconfig, 185
 - source in Agent Builder, 182
 - support in Agent Builder, 184
 - Fini method, guidelines for implementing, 47
 - Fini method, using, 46, 70
 - fully qualified resource type name
 - how obtained, 76
 - functions
 - DSDL fault monitor, 192
 - DSDL network resource access, 189
 - DSDL Process Monitor Facility (PMF), 191
 - DSDL property, 189
 - DSDL utility, 192
 - general purpose DSDL, 187
 - naming conventions, 145
 - RMAPI C program, 63
 - RMAPI cluster, 66
 - RMAPI resource, 63
 - RMAPI resource group, 65
 - RMAPI resource type, 64
 - RMAPI utility, 66
 - scds_initialize(), 146
 - svc_probe(), 157
- G**
- GDS (generic data service)
 - definition, 42
 - generic data service *See* GDS
- H**
- HA data services
 - testing, 57
 - halockrun
 - description, 51
 - handling
 - error, 142
 - hatimerun
 - description, 51

I

- idempotence
 - methods, 42
- implementing
 - fault monitor with DSDL, 123
 - RMAPI, 16
- INADDR_ANY, 288
- Init method, using, 46, 69
- installation requirements
 - resource type packages, 80
- installing Agent Builder, 168
- interface
 - RGM (Resource Group Manager), 22
- interfaces
 - command-line, 24
 - programming, 21

J

- Java
 - sample application that uses CRNP, 204

K

- keep-alives
 - using, 56

L

- libdsdev.so
 - DSDL, 16
- libscha.so
 - RMAPI, 16
- logging
 - adding to a resource, 50

M

- man pages
 - Agent Builder, 183
- master
 - description, 18
- max

- resource type upgrade, 75
- menus
 - Agent Builder, 171
 - Agent Builder Edit, 171
 - Agent Builder File, 171
- message logging
 - adding to a resource, 50
- messages
 - SC_CALLBACK_REG CRNP, 197, 198
 - SC_EVENT CRNP, 201, 202
 - SC_REPLY CRNP, 199, 200
- method arguments
 - RMAPI, 67
- method code
 - changing, 81
- methods
 - Boot, 47, 71, 135
 - callback, 51
 - control, 68
 - initialization, 68
 - Fini, 46, 70, 135
 - Fini, guidelines for implementing, 47
 - idempotence, 42
 - Init, 46, 69, 135
 - Monitor_check, 73, 133
 - Monitor_check callback, 73
 - Monitor_start, 72, 132
 - Monitor_start callback, 72
 - Monitor_stop, 72, 133
 - Monitor_stop callback, 72
 - Postnet_start, 72
 - Postnet_start callback, 72
 - Prenet_start, 72
 - Prenet_start callback, 72
 - Start, 44, 68, 130
 - Stop, 44, 69, 131
 - Update, 51, 72, 134
 - Update callback, 72
 - Validate, 51, 71, 128
 - Validate callback, 71
 - xfnts_monitor_check, 154
 - xfnts_monitor_start, 152

- xfnts_monitor_stop, 153
- xfnts_start, 146
- xfnts_stop, 151
- xfnts_update, 163
- xfnts_validate, 160

min

- resource type upgrade, 75

modifying resource types, 75

monitor code

- changing, 81

Monitor_check method

- compatibility, 78
- using, 73

Monitor_start method

- using, 72

Monitor_stop method

- using, 72

N**naming conventions**

- callback methods, 145
- functions, 145

navigating Agent Builder, 169**network resource access functions**

- DSDL, 189

O**options**

- tunability, 77

Oracle Solaris

- publisher, 29

Oracle Solaris Cluster

- application environment, 15
- commands, 24
- publisher, 29

Oracle Solaris Cluster Agent Builder *See* Agent Builder**Oracle Solaris Cluster Manager**

- description, 22

ORCL.xfnts

- fault monitor, 154

- RTR file, 145

P**package directory**

- Agent Builder, 184

PMF (Process Monitor Facility)

- functions, DSDL, 191
- overview, 16
- purpose, 50

Postnet_start method

- using, 72

Postnet_stop

- compatibility, 78

Prenet_start method

- using, 72

primary nodes, 18**process management, 50****Process Monitor Facility *See* PMF****programming architecture, 16****programming interfaces, 21****properties**

- changing resource, 51
- cluster, 30
- declaring extension, 39
- declaring resource, 35
- declaring resource type, 32
- resource, 31
- resource group, 31
- setting resource, 32, 51
- setting resource type, 32

property attributes

- resource, 31

property functions

- DSDL, 16

property values

- default, 79

property variables, 176

- how Agent Builder substitutes types of, 177
- list of, 176
- list of resource, 176
- list of resource group, 177
- list of resource type, 176

syntax of, 177
 publisher
 Oracle Solaris, 29
 Oracle Solaris Cluster, 29

Q

query
 SCHA_TARGET_NODES, 138

R

registering CRNP clients and servers, 197
 resource
 adding message logging to a, 50
 implementing a failover, 52
 implementing a scalable, 53
 monitoring, 48
 starting, 43
 stopping, 43
 resource commands
 RMAPI, 62
 resource dependencies
 coordinating, 57
 resource functions
 RMAPI, 63
 resource group commands
 RMAPI, 62
 resource group functions
 RMAPI, 65
 Resource Group Manager *See* RGM
 resource group properties, 31
 accessing information about, 41
 resource groups
 description, 18
 failover, 19
 properties, 19
 scalable, 19
 Resource Management API *See* RMAPI
 resource migration
 resource migration from their stop method, 137
 resource properties, 31
 accessing information about, 41

 changing, 51
 declaring, 35
 setting, 32, 51
 resource property
 Pre_evict, 140
 resource property attributes, 31
 resource type
 what happens when upgrading, 79
 resource type names
 obtaining fully qualified, 76
 restrictions, 79, 172
 version suffix, 76
 resource type packages
 installation requirements, 80
 resource type properties
 declaring, 32
 setting, 32
 resource type registration *See* RTR
 resource types
 commands
 RMAPI, 62
 debugging with DSDL, 124
 description, 17
 functions
 RMAPI, 64
 modifying, 75
 multiple versions, 75
 upgrading requirements, 75
resource-type
 upgrading, 76
 resources
 coordinating dependencies between, 57
 description, 18
 reusing code
 Agent Builder, 178
 RGM (Resource Group Manager)
 administrative interface, 22
 description, 19
 handling of resource groups, 17
 handling of resource types, 17
 handling of resources, 17
 purpose, 16
 RMAPI (Resource Management API), 16

- C program functions, 63
 - callback methods, 67
 - cluster commands, 63
 - cluster functions, 66
 - components, 21
 - exit codes, 68
 - libscha.so, 16
 - method arguments, 67
 - resource commands, 62
 - resource functions, 63
 - resource group commands, 62
 - resource group functions, 65
 - resource type commands, 62
 - resource type functions, 64
 - shell commands, 61
 - utility functions, 66
 - where implemented, 16
 - rt-version*
 - upgrading, 76
 - RT_version
 - purpose, 78
 - when to change, 78
 - rtconfig file, 185
 - RTR (Resource Type Registration)
 - description, 20
 - file
 - changing, 80
 - description, 127
 - ORCL.xfnts, 145
- S**
- sample data service
 - common functionality, 92
 - controlling the data service, 97
 - defining a fault monitor, 103
 - extension properties in RTR file, 91
 - generating error messages, 95
 - handling property updates, 113
 - Monitor_check method, 112
 - Monitor_start method, 109
 - Monitor_stop method, 110
 - obtaining property information, 96
 - probe program, 103
 - RTR file, 87
 - sample properties in RTR file, 88
 - Start method, 97
 - Stop method, 100
 - Update method, 118
 - Validate method, 113
 - sample DSDL code
 - determining the fault monitor action, 160
 - ORCL.xfnts fault monitor, 154
 - ORCL.xfnts RTR file, 145
 - returning from svc_start(), 148
 - scds_initialize() function, 146
 - starting the service, 147
 - svc_probe() function, 157
 - TCP port number, 144
 - validating the service, 146
 - X font server, 143
 - X font server configuration file, 144
 - xfnts_monitor_check method, 154
 - xfnts_monitor_start method, 152
 - xfnts_monitor_stop method, 153
 - xfnts_probe main loop, 155
 - xfnts_start method, 146
 - xfnts_stop method, 151
 - xfnts_update method, 163
 - xfnts_validate method, 160
 - SC_CALLBACK_REG
 - contents, 198
 - SC_EVENT
 - contents, 202
 - SC_REPLY
 - contents, 200
 - scalable resource
 - implementing, 53
 - scalable services
 - validating, 55
 - scds_initialize() function, 146
 - screens
 - Configure, 173
 - Create, 172
 - scripts

- Agent Builder, 183
- server
 - CRNP, 197
 - X font
 - configuration file, 144
 - definition, 143
 - xfp
 - port number, 144
- shell commands
 - RMAPI, 61
- source code
 - editing generated Agent Builder, 178
- source files
 - Agent Builder, 182
- Start method, using, 44, 68
- starting a data service with DSDL, 122
- Stop method
 - compatibility, 78
 - using, 44, 69
- stop method
 - resource types that perform resource migration, 137
- stopping a data service with DSDL, 122
- support files
 - Agent Builder, 184
- svc_probe() function, 157

T

- TCP connections
 - using DSDL fault monitoring, 191
- testing
 - data services, 56
 - HA data services, 57
- tunability constraints
 - documentation requirements, 82
- tunability options, 77
 - ANYTIME, 77
 - AT_CREATION, 78
 - WHEN_DISABLED, 78
 - WHEN_OFFLINE, 78
 - WHEN_UNMANAGED, 78
 - WHEN_UNMONITORED, 78

U

- Update method
 - compatibility, 78
 - using, 51, 72
- upgrade aware
 - defined, 76
- upgrades
 - documentation requirements, 82
- upgrading resource types, 75
- utility functions
 - DSDL, 192
 - RMAPI, 66

V

- Validate method
 - using, 51, 71
- validation checks
 - scalable services, 55
- values
 - default property, 79
- variables
 - how Agent Builder substitutes types of property, 177
 - list of property, 176
 - list of resource group property, 177
 - list of resource property, 176
 - list of resource type property, 176
 - property, 176
 - syntax of property, 177
- vendor-id*
 - distinguishing between, 76
 - upgrading, 76

W

- WHEN_DISABLED
 - #\$upgrade_from directive, 78
- WHEN_OFFLINE
 - #\$upgrade_from directive, 78
- WHEN_UNMANAGED
 - #\$upgrade_from directive, 78

WHEN_UNMONITORED

- #\$upgrade_from directive, 78
- writing data services, 56

X

X font server

- configuration file, 144
- definition, 143
- xfnts_monitor_check, 154
- xfnts_monitor_start, 152
- xfnts_monitor_stop, 153
- xfnts_start, 146
- xfnts_stop, 151
- xfnts_update, 163
- xfnts_validate, 160
- xfns server
 - port number, 144