Sun Cluster Data Services Developer's Guide for Solaris OS

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

Part No: 820–4680–10
January 2009, Revision A

Copyright 2009 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more U.S. patents or pending patent applications in the U.S. and in other countries.

U.S. Government Rights – Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

This distribution may include materials developed by third parties.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, the Solaris logo, the Java Coffee Cup logo, docs.sun.com, NetBeans, Java, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. or its subsidiaries in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc. ORACLE is a registered trademark of Oracle Corporation.

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

Products covered by and information contained in this publication are controlled by U.S. Export Control laws and may be subject to the export or import laws in other countries. Nuclear, missile, chemical or biological weapons or nuclear maritime end uses or end users, whether direct or indirect, are strictly prohibited. Export or reexport to countries subject to U.S. embargo or to entities identified on U.S. export exclusion lists, including, but not limited to, the denied persons and specially designated nationals lists is strictly prohibited.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright 2009 Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 U.S.A. Tous droits réservés.

Sun Microsystems, Inc. détient les droits de propriété intellectuelle relatifs à la technologie incorporée dans le produit qui est décrit dans ce document. En particulier, et ce sans limitation, ces droits de propriété intellectuelle peuvent inclure un ou plusieurs brevets américains ou des applications de brevet en attente aux Etats-Unis et dans d'autres pays.

Cette distribution peut comprendre des composants développés par des tierces personnes.

Certaines composants de ce produit peuvent être dérivées du logiciel Berkeley BSD, licenciés par l'Université de Californie. UNIX est une marque déposée aux Etats-Unis et dans d'autres pays; elle est licenciée exclusivement par X/Open Company, Ltd.

Sun, Sun Microsystems, le logo Sun, le logo Solaris, le logo Java Coffee Cup, docs.sun.com, NetBeans, Java et Solaris sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc., ou ses filiales, aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc. ORACLE est une marque déposée registre de Oracle Corporation.

L'interface d'utilisation graphique OPEN LOOK et Sun a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui, en outre, se conforment aux licences écrites de Sun.

Les produits qui font l'objet de cette publication et les informations qu'il contient sont régis par la legislation américaine en matière de contrôle des exportations et peuvent être soumis au droit d'autres pays dans le domaine des exportations et importations. Les utilisations finales, ou utilisateurs finaux, pour des armes nucléaires, des missiles, des armes chimiques ou biologiques ou pour le nucléaire maritime, directement ou indirectement, sont strictement interdites. Les exportations ou réexportations vers des pays sous embargo des Etats-Unis, ou vers des entités figurant sur les listes d'exclusion d'exportation américaines, y compris, mais de manière non exclusive, la liste de personnes qui font objet d'un ordre de ne pas participer, d'une façon directe ou indirecte, aux exportations des produits ou des services qui sont régis par la legislation américaine en matière de contrôle des exportations et la liste de ressortissants spécifiquement designés, sont rigoureusement interdites.

LA DOCUMENTATION EST FOURNIE "EN L'ETAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFACON.

081111@21288

Contents

Preface 13

1 **Overview of Resource Management** 19
Sun Cluster Application Environment 19
Resource Group Manager Model 21
Description of a Resource Type 21
Description of a Resource 22
Description of a Resource Group 22
Resource Group Manager 23
Callback Methods 23
Programming Interfaces 24
Resource Management API 24
Data Service Development Library 25
Sun Cluster Agent Builder 25
Resource Group Manager Administrative Interface 26
Sun Cluster Manager 26
`clsetup` Utility 26
Administrative Commands 27

2 **Developing a Data Service** 29
Analyzing the Application for Suitability 29
Determining the Interface to Use 31
Setting Up the Development Environment for Writing a Data Service 32
▼ How to Set Up the Development Environment 33
Transferring a Data Service to a Cluster 34
Setting Resource and Resource Type Properties 34
Declaring Resource Type Properties 34

Declaring Resource Properties ... 37
Declaring Extension Properties ... 42
Implementing Callback Methods ... 44
Accessing Resource and Resource Group Property Information ... 44
Idempotence of Methods ... 44
How Methods Are Invoked in Zones ... 45
Generic Data Service ... 45
Controlling an Application ... 45
Starting and Stopping a Resource ... 45
Using the Optional `Init`, `Fini`, and `Boot` Methods ... 48
Monitoring a Resource ... 50
Implementing Monitors and Methods That Execute Exclusively in the Global Zone ... 51
Adding Message Logging to a Resource ... 53
Providing Process Management ... 53
Providing Administrative Support for a Resource ... 54
Implementing a Failover Resource ... 55
Implementing a Scalable Resource ... 55
Validation Checks for Scalable Services ... 58
Writing and Testing Data Services ... 59
Using TCP Keep-Alives to Protect the Server ... 59
Testing HA Data Services ... 60
Coordinating Dependencies Between Resources ... 60

3 Resource Management API Reference ... 63
RMAPI Access Methods ... 63
RMAPI Shell Commands ... 63
C Functions ... 65
RMAPI Callback Methods ... 68
Arguments That You Can Provide to Callback Methods ... 69
Callback Method Exit Codes ... 69
Control and Initialization Callback Methods ... 70
Administrative Support Methods ... 73
Net-Relative Callback Methods ... 73
Monitor Control Callback Methods ... 74

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 #$upgrade and #$upgrade_from Directives 77
Changing the RT_version in an RTR File 78
Resource Type Names in Previous Versions of Sun Cluster 79
What Happens When a Cluster Administrator Upgrades 79
Implementing Resource Type Monitor Code 80
Determining Installation Requirements and Packaging 80
Before You Change the RTR File 81
Changing Monitor Code 81
Changing Method Code 82
Determining the Packaging Scheme to Use 82
Documentation to Provide for a Modified Resource Type 83
Information About What to Do Before Installing an Upgrade 84
Information About When to Upgrade Resources 84
Information About Changes to Resource Properties 85

5 **Sample Data Service** 87
Overview of the Sample Data Service 87
Defining the Resource Type Registration File 88
Overview of the RTR File 88
Resource Type Properties in the Sample RTR File 89
Resource Properties in the Sample RTR File 90
Providing Common Functionality to All Methods 94
Identifying the Command Interpreter and Exporting the Path 94
Declaring the PMF_TAG and SYSLOG_TAG Variables 95
Parsing the Function Arguments 96
Generating Error Messages 97
Obtaining Property Information 98
Controlling the Data Service 99
How the Start Method Works 99
How the Stop Method Works 102
Defining a Fault Monitor 104

How the Probe Program Works ... 105
How the Monitor_start Method Works ... 110
How the Monitor_stop Method Works ... 111
How the Monitor_check Method Works ... 113
Handling Property Updates ... 114
How the Validate Method Works ... 114
How the Update Method Works ... 118

6 Data Service Development Library ... 121
DSDL Overview ... 121
Managing Configuration Properties ... 122
Starting and Stopping a Data Service ... 123
Implementing a Fault Monitor ... 123
Accessing Network Address Information ... 124
Debugging the Resource Type Implementation ... 124
Enabling Highly Available Local File Systems ... 125

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

8 Sample DSDL Resource Type Implementation ... 139
X Font Server ... 139
X Font Server Configuration File ... 140
TCP Port Number ... 140
SUNW.xfnts RTR File ... 140

Naming Conventions for Functions and Callback Methods 141
scds_initialize() Function 141
xfnts_start Method 142
Validating the Service Before Starting the X Font Server 142
Starting the Service With svc_start() 142
Returning From svc_start() 144
xfnts_stop Method 146
xfnts_monitor_start Method 147
xfnts_monitor_stop Method 148
xfnts_monitor_check Method 149
SUNW.xfnts Fault Monitor 150
xfonts_probe Main Loop 151
svc_probe() Function 152
Determining the Fault Monitor Action 156
xfnts_validate Method 156
xfnts_update Method 159

9 Sun Cluster Agent Builder 161
Agent Builder Overview 161
Before You Use Agent Builder 162
Using Agent Builder 163
Analyzing the Application 163
Installing and Configuring Agent Builder 164
Agent Builder Screens 164
Starting Agent Builder 165
Navigating Agent Builder 166
Using the Create Screen 169
Using the Configure Screen 171
Using the Agent Builder Korn Shell-Based $hostnames Variable 174
Using Property Variables 174
Reusing Code That You Create With Agent Builder 177
▼ How to Use the Command-Line Version of Agent Builder 178
Directory Structure That Agent Builder Creates 179
Agent Builder Output 180
Source and Binary Files 180

Utility Scripts and Man Pages That Sun Cluster Agent Builder Creates 181
Support Files That Agent Builder Creates 182
Package Directory That Agent Builder Creates 183
`rtconfig` File 183
Cluster Agent Module for Agent Builder 184
▼ How to Install and Set Up the Cluster Agent Module 184
▼ How to Start the Cluster Agent Module 185
Using the Cluster Agent Module 187
Differences Between the Cluster Agent Module and Agent Builder 188

10 Generic Data Services 189
Generic Data Services Concepts 189
Precompiled Resource Type 190
Advantages and Disadvantages of Using the GDS 190
Ways to Create a Service That Uses the GDS 191
How the GDS Logs Events 191
Required GDS Properties 192
Optional GDS Properties 193
Using Agent Builder to Create a Service That Uses the GDS 196
Creating and Configuring GDS-Based Scripts 196
Output From Agent Builder 201
Using Sun Cluster Administration Commands to Create a Service That Uses the GDS 202
▼ How to Use Sun Cluster Administration Commands to Create a Highly Available Service That Uses the GDS 202
▼ How to Use Sun Cluster Administration Commands to Create a Scalable Service That Uses the GDS 203
Command-Line Interface for Agent Builder 204
▼ How to Use the Command-Line Version of Agent Builder to Create a Service That Uses GDS 204

11 DSDL API Functions 207
General-Purpose Functions 207
Initialization Functions 208
Retrieval Functions 208
Failover and Restart Functions 208

Execution Functions 209
Property Functions 209
Network Resource Access Functions 209
Host Name Functions 210
Port List Functions 210
Network Address Functions 210
Fault Monitoring Using TCP Connections Functions 210
PMF Functions 211
Fault Monitor Functions 212
Utility Functions 212

12 Cluster Reconfiguration Notification Protocol 213
CRNP Concepts 213
How the CRNP Works 214
CRNP Semantics 215
CRNP Message Types 215
How a Client Registers With the Server 217
Assumptions About How Administrators Set Up the Server 217
How the Server Identifies a Client 217
How SC_CALLBACK_REG Messages Are Passed Between a Client and the Server 217
How the Server Replies to a Client 219
Contents of an SC_REPLY Message 219
How a Client Is to Handle Error Conditions 220
How the Server Delivers Events to a Client 221
How the Delivery of Events Is Guaranteed 221
Contents of an SC_EVENT Message 222
How the CRNP Authenticates Clients and the Server 223
Example of Creating a Java Application That Uses the CRNP 224
▼ How to Set Up Your Environment 224
▼ How to Start Developing Your Application 225
▼ How to Parse the Command-Line Arguments 227
▼ How to Define the Event Reception Thread 227
▼ How to Register and Unregister Callbacks 228
▼ How to Generate the XML 229
▼ How to Create the Registration and Unregistration Messages 233

▼ How to Set Up the XML Parser 236
▼ How to Parse the Registration Reply 236
▼ How to Parse the Callback Events 238
▼ How to Run the Application 242

A Standard Properties 243
Resource Type Properties 243
Resource Properties 253
Resource Group Properties 273
Resource Property Attributes 287

B Sample Data Service Code Listings 291
Resource Type Registration File Listing 291
`Start` Method Code Listing 295
`Stop` Method Code Listing 298
`gettime` Utility Code Listing 300
`PROBE` Program Code Listing 301
`Monitor_start` Method Code Listing 307
`Monitor_stop` Method Code Listing 309
`Monitor_check` Method Code Listing 311
`Validate` Method Code Listing 313
`Update` Method Code Listing 317

C DSDL Sample Resource Type Code Listings 319
`xfnts.c` File Listing 319
`xfnts_monitor_check` Method Code Listing 333
`xfnts_monitor_start` Method Code Listing 334
`xfnts_monitor_stop` Method Code Listing 335
`xfnts_probe` Method Code Listing 336
`xfnts_start` Method Code Listing 339
`xfnts_stop` Method Code Listing 340
`xfnts_update` Method Code Listing 341
`xfnts_validate` Method Code Listing 343

D Legal RGM Names and Values 345
RGM Legal Names 345
Rules for Names Except Resource Type Names 345
Format of Resource Type Names 346
RGM Values 347

E Requirements for Non-Cluster Aware Applications 349
Multihosted Data 349
Using Symbolic Links for Multihosted Data Placement 350
Host Names 351
Multihomed Hosts 351
Binding to `INADDR_ANY` as Opposed to Binding to Specific IP Addresses 352
Client Retry 353

F Document Type Definitions for the CRNP 355
`SC_CALLBACK_REG` XML DTD 355
`NVPAIR` XML DTD 357
`SC_REPLY` XML DTD 358
`SC_EVENT` XML DTD 359

G `CrnpClient.java` Application 361
Contents of `CrnpClient.java` 361

Index 387

Preface

The *Sun Cluster Data Services Developer's Guide for Solaris OS* contains information about using the Resource Management API to develop Sun™ Cluster data serviceson both SPARC® and x86 based systems.

Note – This Sun Cluster release supports systems that use the SPARC and x86 families of processor architectures: UltraSPARC, SPARC64, AMD64, and Intel 64. In this document, x86 refers to the larger family of 64-bit x86 compatible products. Information in this document pertains to all platforms unless otherwise specified.

Who Should Use This Book

This document is intended for experienced developers with extensive knowledge of Sun software and hardware. The information in this book assumes that you have knowledge of the Solaris Operating System.

How This Book Is Organized

The *Sun Cluster Data Services Developer's Guide for Solaris OS* contains the following chapters and appendixes:

Chapter 1, "Overview of Resource Management," provides an overview of the concepts that you need to develop a data service.

Chapter 2, "Developing a Data Service," provides detailed information about developing a data service.

Chapter 3, "Resource Management API Reference," provides a reference to the access functions and callback methods that make up the Resource Management API (RMAPI).

Chapter 4, "Modifying a Resource Type," 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.

Chapter 5, "Sample Data Service," provides a sample Sun Cluster data service for the `in.named` application.

Chapter 6, "Data Service Development Library," provides an overview of the application programming interfaces that make up the Data Services Development Library (DSDL).

Chapter 7, "Designing Resource Types," explains the typical use of the DSDL in designing and implementing resource types.

Chapter 8, "Sample DSDL Resource Type Implementation," describes a sample resource type that is implemented with the DSDL.

Chapter 9, "Sun Cluster Agent Builder," describes Sun Cluster Agent Builder.

Chapter 10, "Generic Data Services," describes how to create a generic data service.

Chapter 11, "DSDL API Functions," describes the DSDL API functions.

Chapter 12, "Cluster Reconfiguration Notification Protocol," provides information about the Cluster Reconfiguration Notification Protocol (CRNP). The CRNP enables failover and scalable applications to be "cluster aware."

Appendix A, "Standard Properties," describes the standard resource type, resource, and resource group properties.

Appendix B, "Sample Data Service Code Listings," provides the complete code for each method in the sample data service.

Appendix C, "DSDL Sample Resource Type Code Listings," lists the complete code for each method in the `SUNW.xfnts` resource type.

Appendix D, "Legal RGM Names and Values," lists the requirements for legal characters for Resource Group Manager (RGM) names and values.

Appendix E, "Requirements for Non-Cluster Aware Applications," list the requirements for ordinary, non-cluster aware applications to be candidates for high availability.

Appendix F, "Document Type Definitions for the CRNP," lists the document type definitions for the CRNP.

Appendix G, "`CrnpClient.java` Application," shows the complete `CrnpClient.java` application that is discussed in Chapter 12, "Cluster Reconfiguration Notification Protocol."

Related Documentation

Information about related Sun Cluster topics is available in the documentation that is listed in the following table. All Sun Cluster documentation is available at http://docs.sun.com.

Topic	Documentation
Overview	*Sun Cluster Overview for Solaris OS*
	Sun Cluster 3.2 1/09 Documentation Center
Concepts	*Sun Cluster Concepts Guide for Solaris OS*
Hardware installation and administration	*Sun Cluster 3.1 - 3.2 Hardware Administration Manual for Solaris OS*
	Individual hardware administration guides
Software installation	*Sun Cluster Software Installation Guide for Solaris OS*
	Sun Cluster Quick Start Guide for Solaris OS
Data service installation and administration	*Sun Cluster Data Services Planning and Administration Guide for Solaris OS*
	Individual data service guides
Data service development	*Sun Cluster Data Services Developer's Guide for Solaris OS*
System administration	*Sun Cluster System Administration Guide for Solaris OS*
	Sun Cluster Quick Reference
Software upgrade	*Sun Cluster Upgrade Guide for Solaris OS*
Error messages	*Sun Cluster Error Messages Guide for Solaris OS*
Command and function references	*Sun Cluster Reference Manual for Solaris OS*
	Sun Cluster Data Services Reference Manual for Solaris OS
	Sun Cluster Quorum Server Reference Manual for Solaris OS

For a complete list of Sun Cluster documentation, see the release notes for your release of Sun Cluster software at http://wikis.sun.com/display/SunCluster/Home/.

Getting Help

If you have problems installing or using the Sun Cluster software, contact your service provider and provide the following information:

- Your name and email address
- Your company name, address, and phone number
- The model number and serial number of your systems
- The release number of the operating system (for example, the Solaris 10 OS)
- The release number of Sun Cluster software (for example, 3.2 1/09)

Use the following commands to gather information about your systems for your service provider.

Command	Function
`prtconf -v`	Displays the size of the system memory and reports information about peripheral devices
`psrinfo -v`	Displays information about processors
`showrev -p`	Reports which patches are installed
SPARC: `prtdiag -v`	Displays system diagnostic information
`/usr/cluster/bin/clnode show-rev`	Displays Sun Cluster release and package version information

Also have available the contents of the `/var/adm/messages` file.

Documentation, Support, and Training

The Sun web site provides information about the following additional resources:

- Documentation (`http://www.sun.com/documentation/`)
- Support (`http://www.sun.com/support/`)
- Training (`http://www.sun.com/training/`)

Typographic Conventions

The following table describes the typographic conventions that are used in this book.

TABLE P–1 Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file. **Note:** Some emphasized items appear bold online.

Shell Prompts in Command Examples

The following table shows the default UNIX® system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell	`machine_name%`
C shell for superuser	`machine_name#`
Bourne shell and Korn shell	`$`
Bourne shell and Korn shell for superuser	`#`

CHAPTER 1

Overview of Resource Management

This book provides guidelines for creating a resource type for a software application, such as Oracle®, Sun Java™ System Web Server (formerly Sun ONE Web 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 *Sun Cluster Concepts Guide for Solaris OS*.

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

- "Sun Cluster Application Environment" on page 19
- "Resource Group Manager Model" on page 21
- "Resource Group Manager" on page 23
- "Callback Methods" on page 23
- "Programming Interfaces" on page 24
- "Resource Group Manager Administrative Interface" on page 26

Note – This book uses the terms *resource type* and *data service* interchangeably. The term *agent*, though rarely used in this book, is equivalent to *resource type* and *data service*.

Sun Cluster Application Environment

The Sun 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.

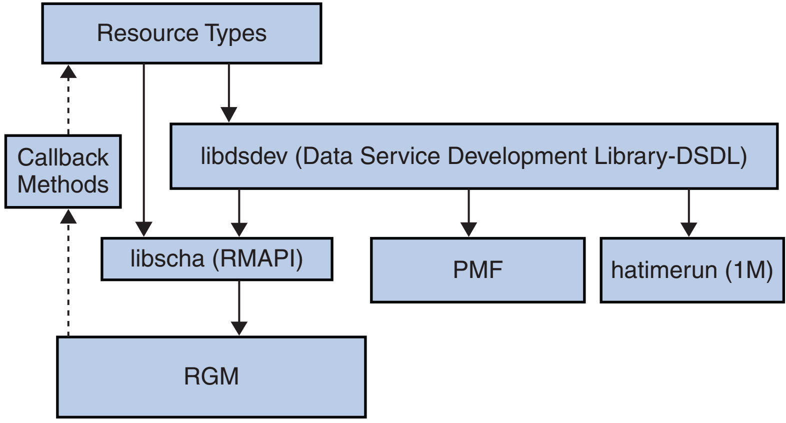


FIGURE 1–1 Programming Architecture of the Sun Cluster Application Environment

Sun Cluster Agent Builder, which is described in Chapter 9, "Sun Cluster Agent Builder," is a tool in the Sun 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).

The RGM runs as a daemon on each cluster node and automatically starts and stops resources on selected Solaris host 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. However, a scalable resource that uses a shared address to balance the service load between nodes can be online in only one node per Solaris host.

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 application to be run in the cluster
- Control programs that are used as callback methods by the RGM to manage the application as a cluster resource
- A set of properties that form part of the static configuration of a 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 34 and in the `rt_reg(4)` man page. See also "Defining the Resource Type Registration File" on page 88 for a description of a sample RTR file.

"Resource Type Properties" on page 243 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 *Sun Cluster Data Services Planning and Administration Guide for Solaris OS* 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 253 contains a list of resource properties.

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.

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 34 and "Implementing Callback Methods" on page 44 for details about these properties.

"Resource Group Properties" on page 273 contains a list of resource group properties.

Resource Group Manager

The Resource Group Manager (RGM) is implemented as a daemon, `rgmd`, that runs on the global-cluster voting node in a cluster. 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 Sun 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 `Prenet_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

  `Prenet_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 Sun 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.
- `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.

Sun 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 source and executable code (C or Korn shell), a customized RTR file, and a Solaris 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 Solaris package for you.

Resource Group Manager Administrative Interface

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

Sun Cluster Manager

Sun 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 Sun Cluster software.

See the *Sun Cluster Software Installation Guide for Solaris OS* for instructions on how to install Sun Cluster Manager and how to use Sun Cluster Manager to install cluster software. Sun Cluster Manager provides online help for most unique administrative tasks.

`clsetup` Utility

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

You can administer the following Sun 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
- Add a network resource to a resource group
- Add a data service resource to a resource group
- 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 Sun 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 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.

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 29
- “Determining the Interface to Use” on page 31
- “Setting Up the Development Environment for Writing a Data Service” on page 32
- “Setting Resource and Resource Type Properties” on page 34
- “Implementing Callback Methods” on page 44
- “Generic Data Service” on page 45
- “Controlling an Application” on page 45
- “Monitoring a Resource” on page 50
- “Adding Message Logging to a Resource” on page 53
- “Providing Process Management” on page 53
- “Providing Administrative Support for a Resource” on page 54
- “Implementing a Failover Resource” on page 55
- “Implementing a Scalable Resource” on page 55
- “Writing and Testing Data Services” 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 B, “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 nonnetwork-aware (client-less) applications are potential candidates for being made highly available or scalable in the Sun Cluster environment. However, Sun 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 351 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 349.

  If the application uses a hardwired 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 350 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 353 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 55 for more information about setting the load-balancing policy for scalable data services.

Determining the Interface to Use

The Sun Cluster developer support package (`SUNWscdev`) 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

Also included in the Sun Cluster developer support package is Sun Cluster Agent Builder, a tool that automates the creation of a data service.

Here is the recommended approach to developing a data service:

1. Decide whether 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 makes sense, for the following reasons:

- The code that is generated by Agent Builder, 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 Sun Cluster development package (`SUNWscdev`) to have access to the Sun Cluster header and library files. Although this package is already installed on all cluster nodes, you typically develop your data service on a separate, non-cluster development machine, rather than on a cluster node. In this typical case, you must use the `pkgadd` command to install the `SUNWscdev` package on your development machine.

Note – On the development machine, ensure that you are using the Developer or Entire Distribution software group of the Solaris 9 OS or the Solaris 10 OS.

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 Solaris Operating System and Sun Cluster products.

Consequently, if you intend to create a C++ based data service for use on Sun Cluster, you must compile that data service, as follows:

- For Sun Cluster 3.0 and prior versions, use the compatibility mode.
- Starting with Sun Cluster 3.1, use the standard mode.

When you have finished development (on a non-cluster 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 Sun Cluster development package (`SUNWscdev`) 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 `SUNWscdev` package and set the compiler and linker options for data service development.

1. **Become superuser or assume a role that provides `solaris.cluster.modify` RBAC authorization.**

2. **Change directory to the CD-ROM directory that you want.**

   ```
   # cd cd-rom-directory
   ```

3. **Install the `SUNWscdev` package in the current directory.**

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

4. **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 Sun 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 Solaris hosts on which you want to run the service.

Note – Agent Builder creates this package automatically.

Setting Resource and Resource Type Properties

Sun Cluster 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 243 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 253 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 Sun 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:

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

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

Init_nodes = RG_PRIMARIES;

RT_basedir=/opt/SUNWsmpl/bin;

Start           =    smpl_svc_start;
Stop            =    smpl_svc_stop;

Validate        =    smpl_validate;
Update          =    smpl_update;

Monitor_start   =    smpl_monitor_start;
Monitor_stop    =    smpl_monitor_stop;
Monitor_check   =    smpl_monitor_check;
```

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 (`SUNW.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. Starting with the Solaris 9 OS, 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 = 2` indicates that the data service can be installed on any version of Sun Cluster starting with Sun Cluster 3.0. `API_version = 7` indicates that the data service can be installed on any version of Sun Cluster starting with Sun Cluster 3.2. However, `API_version = 7` also indicates that the data service cannot be installed on any version of Sun Cluster that was released before Sun Cluster 3.2. This property is described in more detail under the entry for `API_version` in "Resource Type Properties" on page 243.

`Failover = TRUE`
: Indicates that the data service cannot run in a resource group that can be online on multiple nodes at the same time. In other words, this declaration specifies a failover data service. This property is described in more detail under the entry for `Failover` in "Resource Type Properties" on page 243.

`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_PRIMARIES`
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_PRIMARIES` 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/SUNWsample/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`.

Declaring Resource Type Properties for a Zone Cluster

You (and the cluster administrator) can register a resource type for use in a particular zone cluster by creating an RTR file under the zone root path. To correctly configure this RTR file, ensure that it meets the following conditions:

- The `Global_zone` property is either set to `FALSE` or not set at all in the RTR file. If you do not specify the `Global_zone` property, the property is set to `FALSE` by default.
- The RTR file is not of the logical hostname or shared address type.

You can also register a resource type for a zone cluster by placing an RTR file in the `/usr/cluster/lib/rgm/rtreg/` directory. The cluster administrator cannot configure the resource type properties that you declare in an RTR file in this directory.

Resource types that are defined in RTR files in the `/opt/cluster/lib/rgm/rtreg/` directory are for the exclusive use of the global 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 Sun Cluster, which are called *system-defined* properties, you can change specific attributes in the RTR file. For example, Sun Cluster provides default values for method timeout properties for each callback method. In the RTR file, you can specify different default values.

If an RGM method callback times out, the method's process tree is killed by a `SIGABRT` signal (not a `SIGTERM` signal). As a result, all members of the process group generate a core dump file in the `/var/cluster/core` directory or in a subdirectory of the `/var/cluster/core` directory on the node on which the method exceeded its timeout. This core dump file is generated to enable you to determine why your method exceeded its timeout.

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.

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 Sun Cluster. "Resource Property Attributes" on page 287 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 287 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 253 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.

  To maintain compatibility with earlier releases of Sun Cluster software, you can still set the value of the Network_resources_used property directly. If you set the value of the Network_resources_used property directly, the value of the Network_resources_used property is no longer derived from the settings of the resource-dependencies properties. If you add a resource name to the Network_resources_used property, the resource name is automatically added to the Resource_dependencies property as well. The only way to remove that dependency is to remove it from the Network_resources_used property. If you are not sure whether a network-resource dependency was originally added to the Resource_dependencies property or to the Network_resources_used property, remove the dependency from both properties.

Scalable
: Set to FALSE to indicate that this resource does not use the cluster networking (shared address) facility. If you set this property to FALSE, the resource type property Failover must

be set to TRUE to indicate a failover service. See "Transferring a Data Service to a Cluster" on page 34 and "Implementing Callback Methods" on page 44 for additional information about how to use this property.

`Load_balancing_policy` and `Load_balancing_weights`
Automatically declares these properties. However, these properties have no use in a failover resource type.

`Port_list`
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.

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_get(1HA)` and `scha_resource_get(3HA)` man pages.

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 even if the resource group that contains the resource is configured to run in a non-global zone.

If the resource for which `Global_zone` equals `TRUE` is configured in a non-global zone, methods that are invoked in the global zone are invoked with the `-Z` *zonename* option. The *zonename* operand specifies the Solaris zone name of the non-global zone in which the resource is actually configured. The value of this operand is passed to the method.

If the resource is configured in the global zone, the `-Z` *zonename* option is not invoked and the non-global zone name is not passed to the method.

The `Global_zone` resource type property is described in more detail in "Resource Type Properties" on page 243 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 Sun 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 Chapter 10, "Generic Data Services," 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 tree is killed by a `SIGABRT` signal (not a `SIGTERM` signal). As a result, all members of the process group generate a core dump file in the `/var/cluster/core` directory or in a subdirectory of the `/var/cluster/core` directory on the node on which the method exceeded its timeout. This core dump file is generated to enable you to determine why your method exceeded its timeout.

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 unplumb 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 Sun 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_PRIMARIES or RT_INSTALLED_NODE. For most resource types, Init_nodes is set to RG_PRIMARIES, 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 Sun Cluster itself does. Sun Cluster 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.

Although the RGM does not call a resource monitor directly, the RGM does provide for automatically starting monitors for resources. 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 104 for an example of a fault monitor that is implemented with the RMAPI. See "SUNW.xfnts Fault Monitor" on page 150 for an example of a fault monitor that is implemented with the DSDL. See the *Sun Cluster Data Services Planning and Administration Guide for Solaris OS* for information about fault monitors that are built into data services that are supplied by Sun.

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 non-global zone, that is, either a zone-cluster node or a global-cluster non-voting 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 253 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 non-global zone.

When the resource is configured in a non-global zone, 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 non-global zone 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.

If a resource for which the `Global_zone` resource type property equals `TRUE` invokes `scha_cluster_get()` with the `ZONE_LOCAL` query *optag* value, it returns the name of the global zone. In this case, the calling code must concatenate the string `:`*zonename* to the local node name to obtain the zone in which the resource is actually configured. The `zonename` is the same zone name that is passed down to the method in the `-Z` *zonename* command-line option. If there is no `-Z` option in the command line, the resource group is configured in the global zone and you do not need to concatenate a zone name to the node name.

Similarly, if the calling code queries, for example, the state of a resource in the non-global zone, it must invoke `scha_resource_get()` with the `RESOURCE_STATE_NODE` *optag* value rather than the `RESOURCE_STATE` *optag* value. In this case, the `RESOURCE_STATE` *optag* value queries in the global zone rather than in the non-global zone in which the resource is actually configured.

The DSDL functions inherently handle the `-Z` *zonename* option. Therefore, the `scds_initialize()` function obtains the relevant resource and resource group properties for the non-global zone in which a resource is actually configured. Other DSDL queries operate implicitly on that node.

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`.

Multiple `Boot` methods might run simultaneously in the global zone if both of the following conditions occur:

- The `Nodelist` for a resource group contains two or more nodes on the same Solaris host.

Note – You can configure two or more nodes on a global-cluster node only. You can configure only one node in a zone cluster on each Solaris host.

- That same resource group contains one or more resources for which the `Global_zone` property is set to `TRUE`.

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 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): `pmfadm` and `rpc.pmfd`
: Provides a means of monitoring processes and their descendants, and restarting processes if they die. The facility consists of the `pmfadm` command for starting and controlling monitored processes, and the `rpc.pmfd` daemon.

  The DSDL provides a set of functions (preceded by the name `scds_pmf_`) to implement the PMF functionality. See “PMF Functions” on page 211 for an overview of the DSDL PMF functionality and for a list of the individual functions.

The pmfadm(1M) and rpc.pmfd(1M) man pages describe this command and daemon in more detail.

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.

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 243 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 Solaris host or on the same 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 nonnetwork address resources on all network address resources (LogicalHostname and SharedAddress) within the group. As a result, the Start methods of the nonnetwork 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. You can configure a scalable resource (which uses network load-balancing) to run on a global-cluster non-voting node as well. However, you can run such a scalable resource in only one node per Solaris host. Scalable resources include data services such as Sun Cluster HA for Sun Java System Web Server (formerly Sun Cluster HA for Sun ONE Web Server) and Sun 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 cluster administrator uses the RG_dependencies resource group property to specify the order in which resource groups are brought online and offline on a node. This ordering is important for a scalable service because the scalable resources and the shared address resources upon which they depend are located in different resource groups. A scalable data service requires that its network address (shared address) resources be configured to go up before the scalable data service is started. Therefore, the cluster administrator must set the RG_dependencies property (of the resource group that contains the scalable service) to include the resource group that contains the shared address resources.

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 ¼ of the load and node 2 receives ¾ 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 *Sun Cluster Concepts Guide for Solaris OS* 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 `RG_dependencies` property of the resource group that contains the scalable resource must include the resource groups of all shared address resources that are listed in the scalable resource's `Network_resources_used` 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 Sun Cluster configuration so that the testing work does not affect production machines.

Test your HA data service on global-cluster non-voting nodes on a single Solaris host rather than on all Solaris hosts in the cluster. Once you determine that your data service works as expected in the global-cluster non-voting nodes, you can then test it on the entire cluster. Even if it's ill-behaved, a HA data service that runs in a global-cluster non-voting node on a host probably will not perturb the operation of data services that are running in other nodes or on other hosts.

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. Sun Cluster provides for this requirement by permitting resource dependencies to be configured within a resource group. The dependencies affect the order in which Sun 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 Sun Cluster environment, you can use HA-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 Sun Cluster.

If the HA-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-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.

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 63 – In the form of shell script commands and C functions
- “RMAPI Callback Methods” on page 68 – Described in the `rt_callbacks(1HA)` man page

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_resourcetype_get`(1HA) man page.

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_resourcetype_get()` C function. This command is described in more detail in the `scha_resourcegroup_get`(1HA) man page.

`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_get(1HA) man page.

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 non-fatal. 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. You can access this man page through any of the individual functions scha_resource_open(3HA), scha_resource_get(3HA), or scha_resource_close(3HA).

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 non-fatal. 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. You can access this man page through any of the individual functions `scha_resourcetype_open(3HA)`, `scha_resourcetype_get(3HA)`, or `scha_resourcetype_close(3HA)`.

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 non-fatal. 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. You can access this man page through any of the individual functions `scha_resourcegroup_open(3HA)`, `scha_resourcegroup_get(3HA)`, and `scha_resourcegroup_close(3HA)`.

`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)` and `scha_control_zone(3HA)` man pages.

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 non-fatal. 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. You can access this man page through any of the individual functions `scha_cluster_open`(3HA), `scha_cluster_get`(3HA), and `scha_cluster_close`(3HA).

`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()` 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 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 superuser 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 tree is killed by a `SIGABRT` signal (not a `SIGTERM` signal). As a result, all members of the process group generate a core dump file in the `/var/cluster/core` directory or in a subdirectory of the `/var/cluster/core` directory on the node on which the method exceeded its timeout. This core dump file is generated to enable you to determine why your method exceeded its timeout.

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 253.

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_PRIMARIES or RT_INSTALLED_NODES. For most resource types, Init_nodes is set to RG_PRIMARIES, 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-cluster voting node even if the resource group that contains the resource is configured to run in a global-cluster non-voting 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.

CHAPTER 4

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
- "Implementing Resource Type Monitor Code" on page 80
- "Determining Installation Requirements and Packaging" on page 80
- "Documentation to Provide for a Modified Resource Type" on page 83

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 re-create 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 Names in Previous Versions of Sun Cluster" on page 79

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
```

Resource type names that you registered prior to Sun Cluster 3.1 continue to use this syntax:

vendor-id`.`*resource-type*

The format of resource type names is described in "Format of Resource Type Names" on page 346.

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

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 4–1 `#$upgrade_from` Directive in an RTR File

```
#$upgrade_from   "1.1"   WHEN_OFFLINE
#$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 `#$upgrade_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 re-create 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, which is optional in Sun Cluster 3.0, is mandatory starting with the Sun Cluster 3.1 release.

Resource Type Names in Previous Versions of Sun Cluster

Resource type names in Sun Cluster 3.0 do not contain the version suffix, as shown here:

vendor-id.*resource-type*

The name of a resource type that you registered in Sun Cluster 3.0 retains this syntax in Sun Cluster 3.1 and Sun Cluster 3.2. If you register an RTR file in Sun Cluster 3.1 or Sun Cluster 3.2 that omits the `#$upgrade` directive, the resource type name also follows this syntax.

The cluster administrator can register RTR files by using the `#$upgrade` directive or the `#$upgrade_from` directive in Sun Cluster 3.0. However, upgrading existing resources to new resource types in Sun Cluster 3.0 is not supported.

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 – Resources that were created in Sun Cluster 3.0 do not inherit new default resource property attributes from the resource type when they are upgraded to a later version of Sun Cluster. This limitation applies only to Sun Cluster 3.1 clusters that are upgraded from Sun Cluster 3.0 clusters. The cluster administrator can overcome this limitation by specifying values for the properties and thus overriding the defaults.

Implementing Resource Type Monitor Code

The cluster administrator can register an upgrade-aware resource type in Sun Cluster 3.0. However, Sun Cluster records the resource type name without the version suffix. To run correctly in Sun Cluster 3.0 and Sun Cluster 3.1, the monitor code for this resource type must be able to handle both naming conventions:

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

The format of resource type names is described in "Format of Resource Type Names" on page 346.

The cluster administrator cannot register the same version of the resource type twice under two different names. To enable the monitor code to determine the correct name, call these commands in the monitor code:

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

Then, compare the output values with `vers`. Only one of these commands succeeds for a particular value of `vers`.

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 `clresourcetype(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 4–1 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.

TABLE 4–1 Determining the Packaging Scheme to Use *(Continued)*

Type of Change	Tunability Value	Packaging Scheme
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 `Fini` method is incompatible with the old `Init` 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 *Sun Cluster Data Services Planning and Administration Guide for Solaris OS*. 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, instruct the cluster administrator to reboot the node in noncluster mode.
- If the upgrade package updates only the monitor code and leaves the method code unchanged, tell the cluster administrator to keep the node running in cluster mode. Also tell the cluster administrator to turn off monitoring of all resource types.
- If the upgrade package updates only the RTR file, leaving the method and monitor code unchanged, tell the cluster administrator to keep the node running in cluster mode. Also tell the cluster administrator to keep monitoring turned on for all resource types.

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 4–2 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
```

EXAMPLE 4–2 How `#$upgrade_from` Defines When a Cluster Administrator Can Upgrade *(Continued)*

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

CHAPTER 5

Sample Data Service

This chapter describes a sample Sun Cluster data service, HA-DNS, for the `in.named` application. The `in.named` daemon is the 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 87
- "Defining the Resource Type Registration File" on page 88
- "Providing Common Functionality to All Methods" on page 94
- "Controlling the Data Service" on page 99
- "Defining a Fault Monitor" on page 104
- "Handling Property Updates" on page 114

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 34 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 291.

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.

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) 1998-2006 by Sun Microsystems, Inc.
# All rights reserved.
#
# Registration information for Domain Name Service (DNS)
#

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

Resource_type = "sample";
Vendor_id = SUNW;
RT_description = "Domain Name Service on Sun Cluster";

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

RT_basedir=/opt/SUNWsample/bin;
Pkglist = SUNWsample;

Start         =   dns_svc_start;
Stop          =   dns_svc_stop;

Validate      =   dns_validate;
Update        =   dns_update;

Monitor_start =   dns_monitor_start;
Monitor_stop  =   dns_monitor_stop;
Monitor_check =   dns_monitor_check;
```

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 (`SUNW.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 Sun Cluster version. For example, `API_version = 2` indicates that the data service can run on any version of Sun Cluster starting with Sun Cluster 3.0. `API_version = 7` indicates that the data service can be installed on any version of Sun Cluster starting with 3.2. However, `API_version = 7` also indicates that the data service cannot be installed on any version of Sun Cluster that was released before 3.2. This property is described in more detail under the entry for `API_version` in "Resource Type Properties" on page 243.
- `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/SUNWsample/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 `SUNWsample` 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 243 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 Sun Cluster software and extension properties that you define. For either type, you can specify a number of property attributes that are supplied by the Sun 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 Sun Cluster software provides the system-defined properties, you can set different default values by using resource property attributes. See "Resource Property Attributes" on page 287 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:

- Sun 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. Sun 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 253 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 a Sun Cluster callback method but a user-defined method. Therefore, Sun 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 94
- "Declaring the `PMF_TAG` and `SYSLOG_TAG` Variables" on page 95
- "Parsing the Function Arguments" on page 96
- "Generating Error Messages" on page 97
- "Obtaining Property Information" on page 98

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 Sun 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/SUNWsample/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 114 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 a Sun 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 47 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 global-cluster voting 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 94.

For the complete listing of the Start method, see "Start Method Code Listing" on page 295.

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 101.

```
# 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 $RETRY_INTERVAL; if it crashes
# more often than that, PMF will cease trying to restart it.
# If there is a process already registered under the tag
# <$PMF_TAG>, then PMF sends out an alert message that the
# process is already running.
pmfadm -c $PMF_TAAG -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 94.

For the complete listing of the `Stop` method, see "`Stop` Method Code Listing" on page 298.

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 ${SYSLOG_FACILITY}.err \
          -t [$RESOURCETYPE_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 ${SYSLOG_FACILITY}.err \
    -t [$RESOURCETYPE_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 94.

For the complete listing of the PROBE method, see "PROBE Program Code Listing" on page 301.

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_INTERVAL=‘scha_resource_get -O Retry_interval -R $RESOURCE_NAME -G \
$RESOURCEGROUP_NAME‘

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

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=/tmp/.$RESOURCE_NAME.probe
probefail=0
retries=0
```

The `while` loop carries out the following tasks:

- Sets the sleep interval for the probe
- Uses `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.
   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 [ $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
```

Comparing Restart With Failover

If the `probefail` 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 `probefail` variable is 0, a message is generated that the probe was successful.

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

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 301 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 109, 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 `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`
                hatimerun -t $STOP_TIMEOUT $RT_BASEDIR/$STOP_METHOD \
                        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME \
                        -T $RESOURCETYPE_NAME

                if [[ $? -ne 0 ]]; then
```

```
                logger-p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG] \
                        “${ARGV0} Stop method failed.”
                return 1
        fi

        # 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_NAMÈ
        START_METHOD=`scha_resource_get -O START \
                -R $RESOURCE_NAME -G $RESOURCEGROUP_NAMÈ
        hatimerun -t $START_TIMEOUT $RT_BASEDIR/$START_METHOD \
                -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME \
                -T $RESOURCETYPE_NAME

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


else
        # 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 94.

For the complete listing of the Monitor_start method, see "Monitor_start Method Code Listing" on page 307.

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 94.

For the complete listing of the Monitor_stop method, see "Monitor_stop Method Code Listing" on page 309.

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 94.

For the complete listing of the `Monitor_check` method, see "`Monitor_check` Method Code Listing" on page 311.

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_NAMÈ

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

# 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 114 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 getopts 'cur:x:g:R:T:G:' 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
                  ;;
         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 ${SYSLOG_FACILITY}.err \
              -t [$RESOURCETYPE_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.

CHAPTER 6

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 Sun Cluster package.

This chapter covers the following topics:

- "DSDL Overview" on page 121
- "Managing Configuration Properties" on page 122
- "Starting and Stopping a Data Service" on page 123
- "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 125

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 Sun 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 Sun 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 142 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 156 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 `HAStoragePlus` resource type to make a local file system highly available within a Sun Cluster environment.

Note – Local file systems include the UNIX File System (UFS), Quick File System (QFS), Veritas File System (VxFS), and Solaris ZFS (Zettabyte File System).

The local file system partitions must be located on global disk groups. Affinity switchovers must be enabled, and the Sun 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 *Sun Cluster Data Services Planning and Administration Guide for Solaris OS* contains information about configuring the `HAStoragePlus` resource type.

CHAPTER 7

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_get`*name*`()` functions.

This chapter covers the following topics:

- "Resource Type Registration File" on page 128
- "`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 133
- "Description of `Init`, `Fini`, and `Boot` Methods" on page 134
- "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 Sun 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()` utility.

`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_PRIMARIES`, `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_PRIMARIES` (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 HAStoragePlus 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.HAStoragePlus resources on which the resource depends by using the Resource_dependencies or Resource_dependencies_weak system properties that are defined for the resource. 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 Sun 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()` utility. 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 *Sun Cluster Data Services Planning and Administration Guide for Solaris OS* 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 211 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 Sun 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 obtaint 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 */
```

CHAPTER 8

Sample DSDL Resource Type Implementation

This chapter describes a sample resource type, SUNW.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 C, "DSDL Sample Resource Type Code Listings," contains the complete code for each method in the SUNW.xfnts resource type.

This chapter covers the following topics:

- "X Font Server" on page 139
- "SUNW.xfnts RTR File" on page 140
- "Naming Conventions for Functions and Callback Methods" on page 141
- "scds_initialize() Function" on page 141
- "xfnts_start Method" on page 142
- "xfnts_stop Method" on page 146
- "xfnts_monitor_start Method" on page 147
- "xfnts_monitor_stop Method" on page 148
- "xfnts_monitor_check Method" on page 149
- "SUNW.xfnts Fault Monitor" on page 150
- "xfnts_validate Method" on page 156
- "xfnts_update Method" on page 159

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/openwin/bin/xfs. 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 Sun 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 Sun 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 `SUNW.xfnts` resource type uses the following command to start the daemon under the control of the Sun Cluster software.

```
/usr/openwin/bin/xfs -config location-of-configuration-file/fontserver.cfg \
-port port-number
```

In the `SUNW.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 `SUNW.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 `SUNW.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.

`SUNW.xfnts` RTR File

This section describes several key properties in the `SUNW.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 34.

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 SUNW.xfnts sample resource type, the xfnts_start method activates the xfs 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 141 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 xfs 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 156 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 xfs.

```
/* Construct the command to start the xfs daemon. */
   (void) sprintf(cmd,
       "/usr/openwin/bin/xfs -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 xfs 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, `svc_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 141 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 141 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 141 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 141 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 141 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 141 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 141 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 `xfs` daemon. See "`xfnts_validate` Method" on page 156 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);
}
```

SUNW.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 SUNW.xfnts fault monitor performs the following operations:

- Periodically monitors the health of the xfs server daemon by using utilities that are specifically designed to check simple TCP-based services, such as xfs.
- 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.

xfonts_probe Main Loop

The xfonts_probe method implements a loop.

Before implementing the loop, xfonts_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/openwin/bin/xfs", &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 141 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.

CHAPTER 9

Sun Cluster Agent Builder

This chapter describes Sun Cluster Agent Builder and the Cluster Agent module for Agent Builder. Both tools automate 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 161
- "Before You Use Agent Builder" on page 162
- "Using Agent Builder" on page 163
- "Directory Structure That Agent Builder Creates" on page 179
- "Agent Builder Output" on page 180
- "Cluster Agent Module for Agent Builder" on page 184

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 nonnetwork-aware applications. *Network-aware applications* use the network to communicate with clients. *Nonnetwork-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 178.

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 nonnetwork-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.
- A Solaris 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 163
- "Installing and Configuring Agent Builder" on page 164
- "Agent Builder Screens" on page 164
- "Starting Agent Builder" on page 165
- "Navigating Agent Builder" on page 166
- "Using the Create Screen" on page 169
- "Using the Configure Screen" on page 171
- "Using the Agent Builder Korn Shell-Based `$hostnames` Variable" on page 174
- "Using Property Variables" on page 174
- "Reusing Code That You Create With Agent Builder" on page 177
- "How to Use the Command-Line Version of Agent Builder" on page 178

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 29 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 Solaris package for your resource type.

Installing and Configuring Agent Builder

Agent Builder requires no special installation. Agent Builder is included in the `SUNWscdev` package, which is installed by default when you install the Sun Cluster software. The *Sun Cluster Software Installation Guide for Solaris OS* contains more information.

Before you use Agent Builder, verify the following requirements:

- The Java runtime environment is included in your `$PATH` variable. Agent Builder depends on the Java Development Kit, at least Version 1.3.1. If the Java Development Kit is not included in your `$PATH` variable, the Agent Builder command (`scdsbuilder`) returns and displays an error message.
- You have installed the Developer System Support software group of the Solaris 9 OS or the Solaris 10 OS.
- 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 169.

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 Chapter 10, "Generic Data Services." 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 171. 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 178.

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.

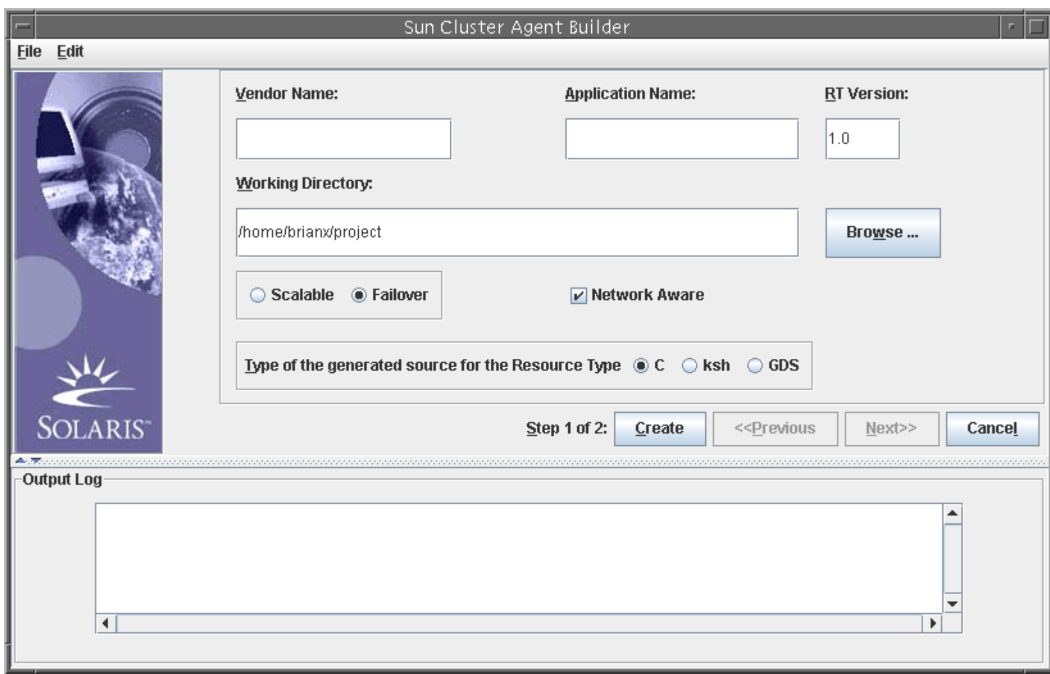


FIGURE 9–1 Create Screen for Agent Builder

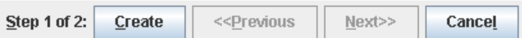
Navigating Agent Builder

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 nonnetwork-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.

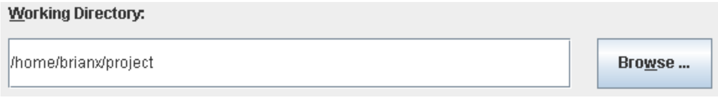


Agent Builder displays progress messages in the Output Log area at the bottom of the screen. When Agent Builder finishes, it displays a success message or a warning message. Next is highlighted, or if this is the last screen, only Cancel is highlighted.

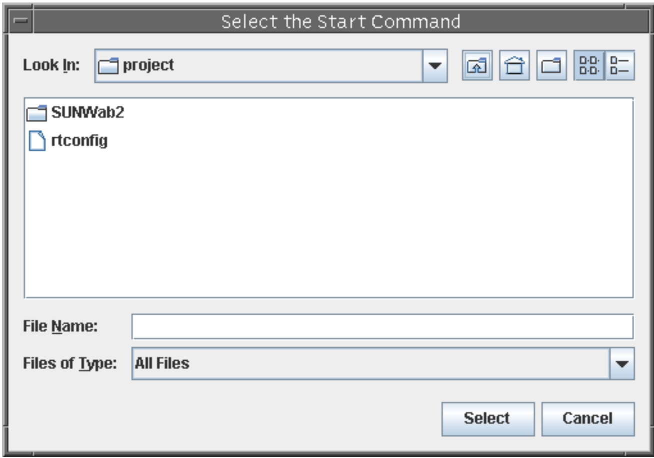
You can click Cancel at any time to exit Agent Builder.

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 similar to this screen appears.



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.
	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 177.
- **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 following figure shows the Create screen after you type information in the fields.

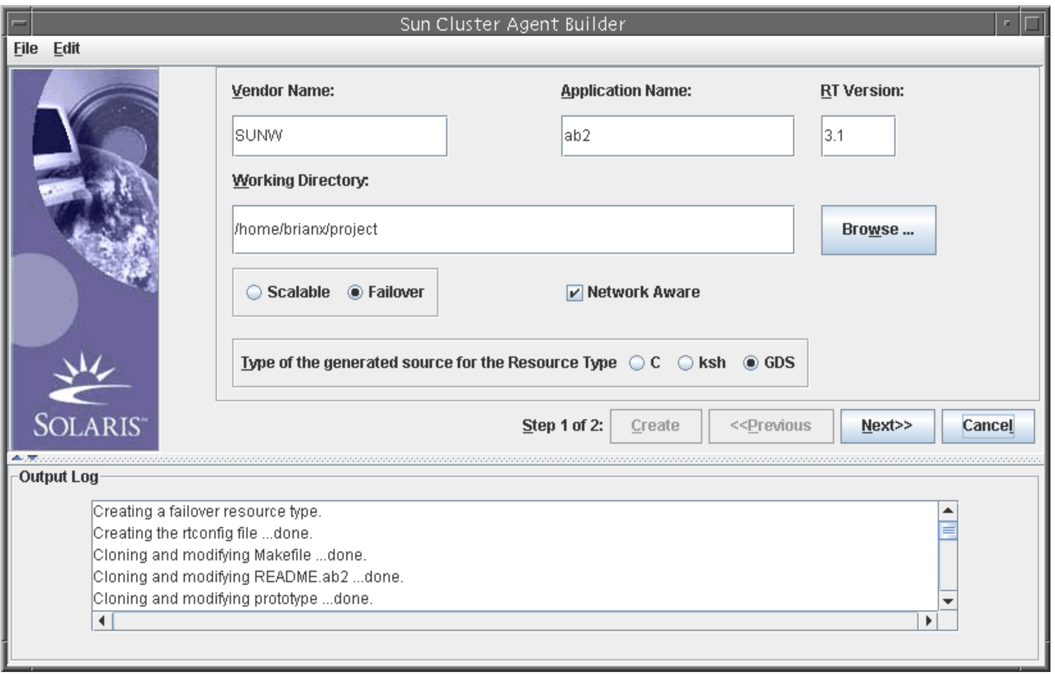


FIGURE 9–2 Agent Builder Create Screen, After You Type Information

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. Starting with the Solaris 9 OS, the combination of Vendor Name and Application Name can exceed nine characters.

- **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 `SUNW` is the vendor name and `ftp` is the application name, Agent Builder names this subdirectory `SUNWftp`.

  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 179.
- **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 nonnetwork 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 177.
- **GDS.** Specify that this service is a generic data service. Chapter 10, "Generic Data Services," 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 164.

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 177.

Using the Configure Screen

The Configure screen, shown in the following figure, 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.

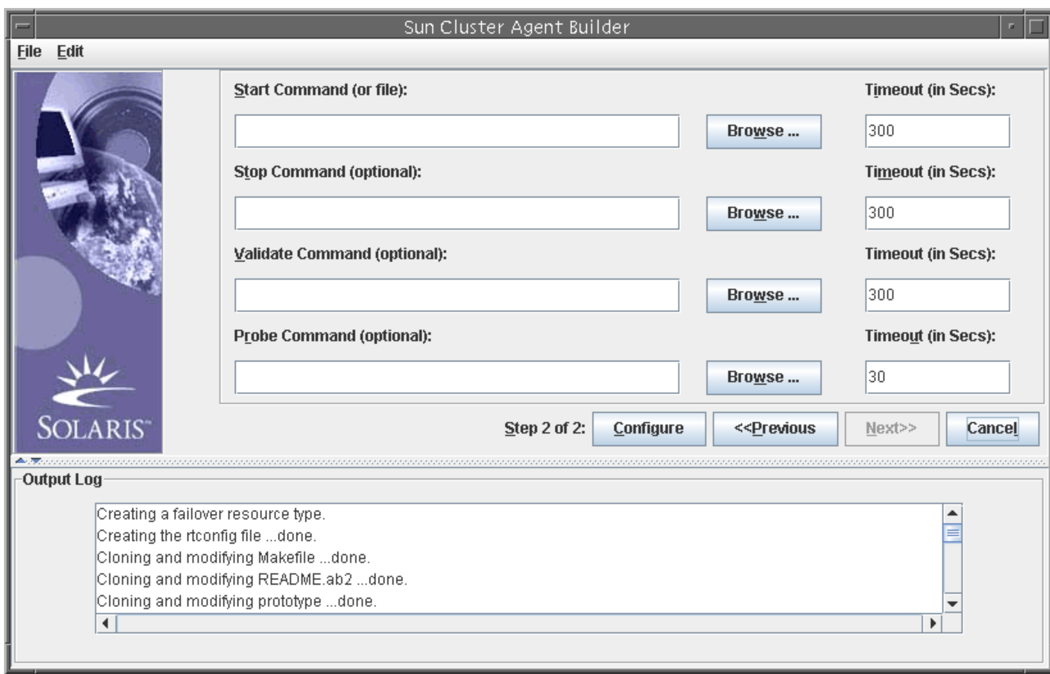


FIGURE 9–3 Configure Screen for Agent Builder

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 174. 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 174.

  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 162. 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 174. 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 174.

  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 174. 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 174.

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 Sun 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 Sun Cluster resource type, resource, and resource group properties are described in Appendix A, "Standard Properties."

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_INTERVAL
- 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 vvvvvvvvvvvvvvv */
/* 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 34 for more information about the RTR file. See Appendix A, "Standard Properties," 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 a Sun 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 174.

3 **Change directories to the `pkg` subdirectory in the working directory.**

4 **Use the `pkgadd` command to install the packages that you created with `scdscreate`.**

```
# pkgadd -d . package-name
```

5 **(Optional) Edit the generated source code.**

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 `SUNW` as the vendor name and create a resource type called `ftp`, Agent Builder creates a directory called `SUNWftp` 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 `SUNW` and the name of the resource type is `ftp`, the name of the RTR file is `SUNW.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*`/SUNWftp/man removeftp`
`pkg`	Contains the final Solaris package that includes the created data service.
`src`	Contains the source files that Agent Builder generates.

Directory Name	Contents
`util`	Contains the `start`, `stop`, and `remove` utility scripts that Agent Builder generates. See "Utility Scripts and Man Pages That Sun Cluster Agent Builder Creates" on page 181. 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 180
- "Utility Scripts and Man Pages That Sun Cluster Agent Builder Creates" on page 181
- "Support Files That Agent Builder Creates" on page 182
- "Package Directory That Agent Builder Creates" on page 183
- "`rtconfig` File" on page 183

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 21, "Callback Methods" on page 23, 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 178.

Utility Scripts and Man Pages That Sun Cluster 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 Sun 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 SUNW is the vendor and ftp is the application name, type the following command to view the startftp(1M) man page:

```
% man -M install-directory/SUNWftp/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/SUNWftp/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 34 for more information. The RTR file is named as *vendor-name*`.`*resource-type-name*, for example, `SUNW.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 a Solaris package. The name of the package is a concatenation of the vendor name and the application name, for example, `SUNWftp`. 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.

When you remove a package from a cluster, the `pkgrm` 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 177.

Cluster Agent Module for Agent Builder

The Cluster Agent module for Agent Builder is a NetBeans™ module. This module enables you to create resource types for the Sun Cluster software through the Sun Java Studio product.

Note – The Sun Java Studio documentation contains information about how to set up, install, and use the Sun Java Studio product. You can find this documentation at the `http://wwws.sun.com/software/sundev/jde/documentation/index.html` web site.

▼ How to Install and Set Up the Cluster Agent Module

The Cluster Agent module is installed when you install the Sun Cluster software. The Sun Cluster installation tool places the Cluster Agent module file `scdsbuilder.jar` in `/usr/cluster/lib/scdsbuilder`. To use the Cluster Agent module with the Sun Java Studio software, you need to create a symbolic link to this file.

Note – The Sun Cluster and Sun Java Studio products and Java 1.4 must be installed and available to the system on which you intend to run the Cluster Agent module.

---

1 Enable all users or only yourself to use the Cluster Agent module.

- To enable all users, become superuser or assume a role that provides `solaris.cluster.modify` RBAC authorization, and create the symbolic link in the global module directory.

  ```
  # cd /opt/s1studio/ee/modules
  # ln -s /usr/cluster/lib/scdsbuilder/scdsbuilder.jar
  ```

  Note – If you installed the Sun Java Studio software in a directory other than `/opt/s1studio/ee`, substitute this directory path with the path that you used.

- To enable only yourself, create the symbolic link in your `modules` subdirectory.

  ```
  % cd ~your-home-dir/ffjuser40ee/modules
  % ln -s /usr/cluster/lib/scdsbuilder/scdsbuilder.jar
  ```

2 Stop and restart the Sun Java Studio software.

▼ How to Start the Cluster Agent Module

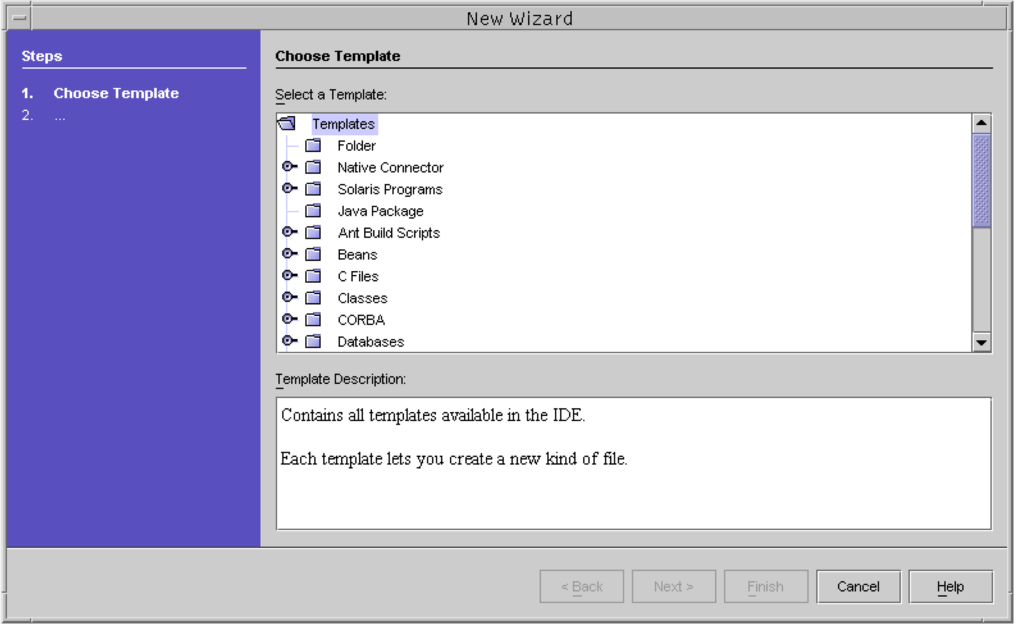
The following steps describe how to start the Cluster Agent module from the Sun Java Studio software.

Note – The Sun Java Studio documentation contains information about how to set up, install, and use the Sun Java Studio product. You can find this documentation at the `http://wwws.sun.com/software/sundev/jde/documentation/index.html` web site.

---

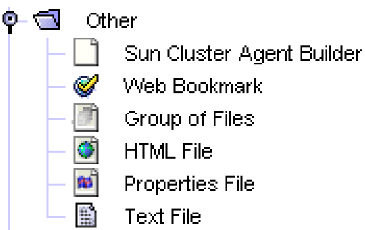
1 From the Sun Java Studio File menu, choose New, or click this icon on the toolbar:

The New Wizard screen appears.



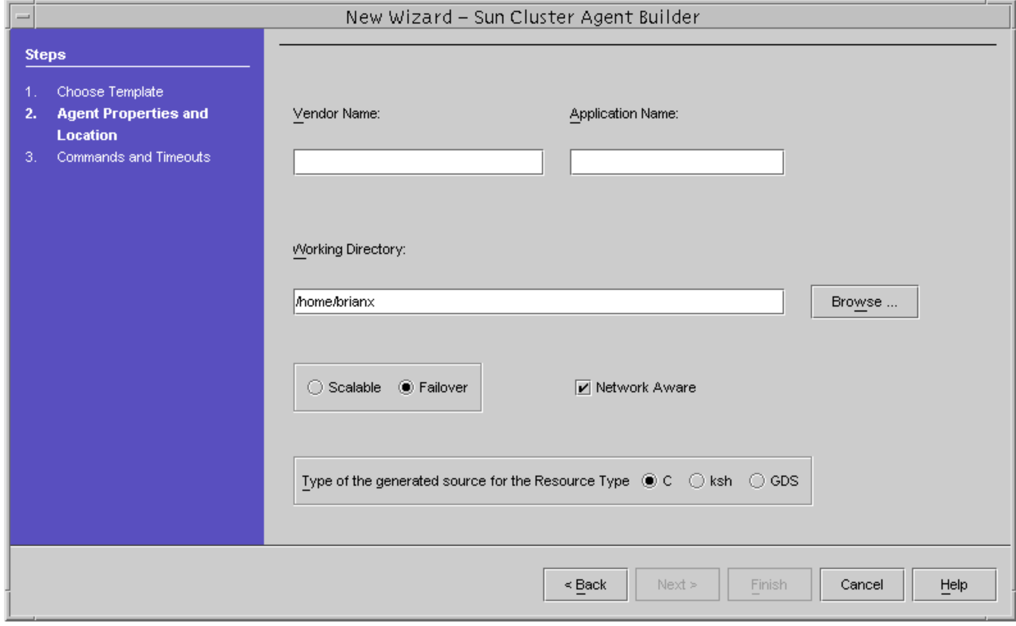
2 In the Select a Template pane, scroll down (if necessary) and click the key next to the Other folder.

The Other folder opens.



3 **From the Other folder, select Sun Cluster Agent Builder and click Next.**

The Cluster Agent module for Sun Java Studio starts. The first New Wizard - Sun Cluster Agent Builder screen appears.



Using the Cluster Agent Module

Use the Cluster Agent module as you would the Agent Builder software. The interfaces are identical. For example, the following figures show that the Create screen in the Agent Builder software and the first New Wizard - Sun Cluster Agent Builder screen in the Cluster Agent module contain the same fields and selections.

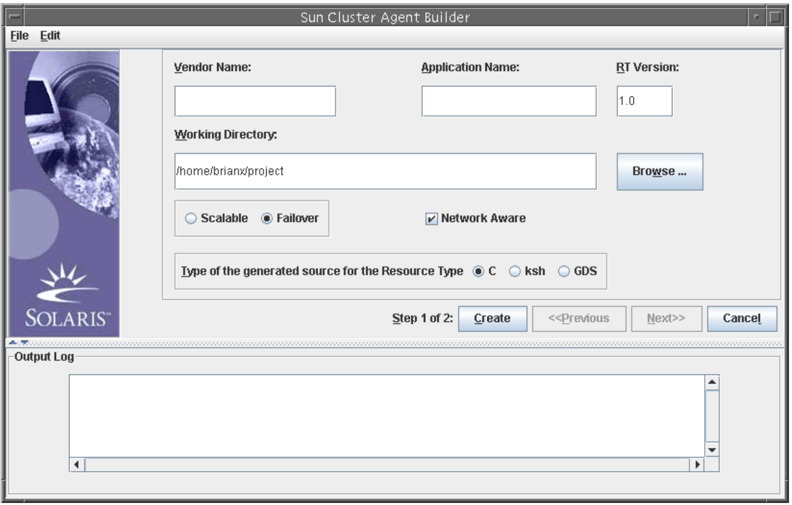


FIGURE 9–4 Create Screen in the Agent Builder Software

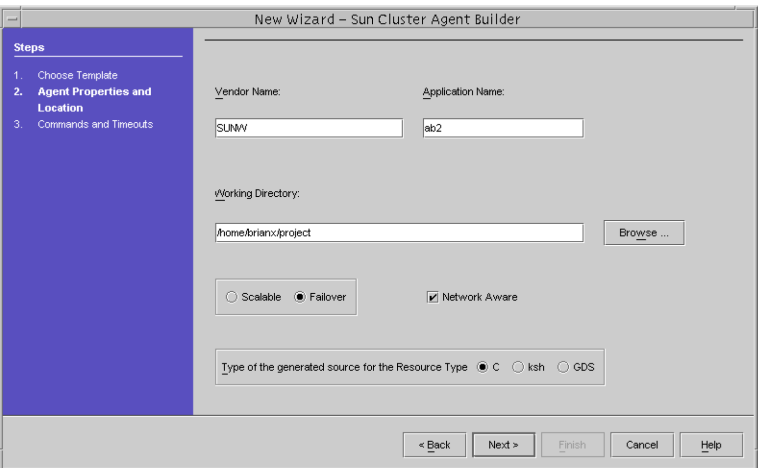


FIGURE 9–5 New Wizard - Sun Cluster Agent Builder Screen in the Cluster Agent Module

Differences Between the Cluster Agent Module and Agent Builder

Despite the similarities between the Cluster Agent module and Agent Builder, minor differences exist:

- In the Cluster Agent module, the resource type is created and configured only after you click Finish on the second New Wizard - Sun Cluster Agent Builder screen. The resource type is *not* created when you click Next on the first New Wizard - Sun Cluster Agent Builder screen.

  In Agent Builder, the resource type is immediately created when you click Create on the Create screen. In addition, the resource type is immediately configured when you click Configure on the Configure screen.
- The information that appears in the Output Log area in Agent Builder appears in a separate window in the Sun Java Studio product.

CHAPTER 10

Generic Data Services

This chapter provides information about the generic data service (GDS) and shows you how to create a service that uses the GDS. You create this service by using either Sun Cluster Agent Builder or Sun Cluster administration commands.

This chapter covers the following topics:

- "Generic Data Services Concepts" on page 189
- "Using Agent Builder to Create a Service That Uses the GDS" on page 196
- "Using Sun Cluster Administration Commands to Create a Service That Uses the GDS" on page 202
- "Command-Line Interface for Agent Builder" on page 204

Generic Data Services Concepts

The GDS is a mechanism for making simple network-aware and nonnetwork-aware applications highly available or scalable by plugging them into the Sun Cluster Resource Group Management (RGM) framework. This mechanism does not require you to code a data service, which you typically must do to make an application highly available or scalable.

You can configure a GDS-based data service to run in non-global zones provided that your associated application is also configured to run in non-global zones.

The GDS is a single, precompiled data service. You cannot modify the precompiled data service and its components, the callback method (`rt_callbacks`) implementations, and the resource type registration file (`rt_reg`).

This section covers the following topics:

- "Precompiled Resource Type" on page 190
- "Advantages and Disadvantages of Using the GDS" on page 190
- "Ways to Create a Service That Uses the GDS" on page 191
- "How the GDS Logs Events" on page 191

- "Required GDS Properties" on page 192
- "Optional GDS Properties" on page 193

Precompiled Resource Type

The generic data service resource type SUNW.gds is included in the SUNWscgds package. The scinstall utility installs this package during cluster installation. See the scinstall(1M) man page. The SUNWscgds package includes the following files:

```
# pkgchk -v SUNWscgds

/opt/SUNWscgds
/opt/SUNWscgds/bin
/opt/SUNWscgds/bin/gds_monitor_check
/opt/SUNWscgds/bin/gds_monitor_start
/opt/SUNWscgds/bin/gds_monitor_stop
/opt/SUNWscgds/bin/gds_probe
/opt/SUNWscgds/bin/gds_svc_start
/opt/SUNWscgds/bin/gds_svc_stop
/opt/SUNWscgds/bin/gds_update
/opt/SUNWscgds/bin/gds_validate
/opt/SUNWscgds/etc
/opt/SUNWscgds/etc/SUNW.gds
```

Advantages and Disadvantages of Using the GDS

Using the GDS has the following advantages over using either the Agent Builder source code (see the scdscreate(1HA) man page) or Sun Cluster administration commands:

- The GDS is easy to use.
- The GDS and its methods are precompiled and therefore cannot be modified.
- You can use Agent Builder to generate scripts for your application. These scripts are put in a Solaris package that can be reused across multiple clusters.

While using the GDS has many advantages, the GDS is *not* the mechanism to use in these instances:

- When more control is required than is available with the precompiled resource type, such as when you need to add extension properties or change default values
- When the source code needs to be modified to add special functions

Ways to Create a Service That Uses the GDS

There are two ways to create a service that uses the GDS:

- Agent Builder
- Sun Cluster administration commands

GDS and Agent Builder

Use Agent Builder and select GDS as the type of generated source code. The user input is used to generate a set of scripts that configure resources for the given application.

GDS and Sun Cluster Administration Commands

This method uses the precompiled data service code in `SUNWscgds`. However, the cluster administrator must use Sun Cluster administration commands to create and configure the resource. See the `clresource`(1CL) man page.

Selecting the Method to Use to Create a GDS-Based Service

A significant amount of typing is required to issue Sun Cluster commands. For example, see "How to Use Sun Cluster Administration Commands to Create a Highly Available Service That Uses the GDS" on page 202 and "How to Use Sun Cluster Administration Commands to Create a Scalable Service That Uses the GDS" on page 203.

Using the GDS with Agent Builder simplifies the process because the GDS generates the scripts that issue the `scrgadm` and `scswitch` commands for you.

How the GDS Logs Events

The GDS enables you to log relevant information that is passed from the GDS to the scripts that the GDS starts. This information includes the status of the start, probe, validate, and stop methods as well as property variables. You can use this information to diagnose problems or errors in your scripts, or apply it to other purposes.

You use the `Log_level` property that is described in "`Log_level` Property" on page 194 to specify the level, or type, of messages that the GDS is to log. You can specify `NONE`, `INFO`, or `ERR`.

GDS Log Files

The following two GDS log files are placed in the directory `/var/cluster/logs/DS/`*resource-group-name*`/`*resource-name*:

- `start_stop_log.txt`, which contains messages that are generated by resource start and stop methods
- `probe_log.txt`, which contains messages that are generated by the resource monitor

The following example shows the types of information that `start_stop_log.txt` contains:

```
06/12/2006 12:38:05 phys-node-1 START-INFO> Start succeeded. [/home/brianx/sc/start_cmd]
06/12/2006 12:42:11 phys-node-1 STOP-INFO> Successfully stopped the application
```

The following example shows the types of information that `probe_log.txt` contains:

```
06/12/2006 12:38:15 phys-node-1 PROBE-INFO> The GDS monitor (gds_probe) has been started
06/12/2006 12:39:15 phys-node-1 PROBE-INFO> The probe result is 0
06/12/2006 12:40:15 phys-node-1 PROBE-INFO> The probe result is 0
06/12/2006 12:41:15 phys-node-1 PROBE-INFO> The probe result is 0
```

Required GDS Properties

This section describes the required GDS properties.

`Port_list` Property

The `Port_list` property identifies the list of ports on which the application listens. You must specify the `Port_list` property in the start script that Agent Builder creates or with the `clresource` command.

Whether you must specify this property depends on whether your application is network aware or not. If you specify that your application is network aware (you set the `Network_aware` property to `TRUE`, the default), you must provide both the `Start_command` extension property and the `Port_list` property. If you specify that your application is nonnetwork aware (you set the `Network_aware` property to `FALSE`), you must provide only the `Start_command` extension property. The `Port_list` property is optional.

`Start_command` Property

The start command, which you specify with the `Start_command` extension property, starts the application. This command must be a UNIX command with arguments that can be passed directly to a shell to start the application.

If your application is network aware, you must provide both the `Start_command` extension property and the `Port_list` property. If your application is nonnetwork aware, you must provide only the `Start_command` extension property.

Optional GDS Properties

Optional GDS properties include both *system-defined properties* and *extension properties*. System-defined properties are a standard set of properties that are provided by Sun Cluster. Properties that are defined in the RTR file are called extension properties.

Here are optional GDS properties:

- `Child_mon_level` extension property (used only with administration commands)
- `Failover_enabled` extension property
- `Log_level` extension property
- `Network_aware` extension property
- `Network_resources_used` property
- `Probe_command` extension property
- `Probe_timeout` extension property
- `Start_timeout` property
- `Stop_command` extension property
- `Stop_signal` extension property
- `Stop_timeout` property
- `Validate_command` extension property
- `Validate_timeout` property

`Child_mon_level` Property

Note – If you use Sun Cluster administration commands, you can use the `Child_mon_level` property. If you use Agent Builder, you cannot use this property.

This property provides control over the processes that are monitored through the Process Monitor Facility (PMF). This property denotes the level up to which the forked children processes are monitored. This property works like the `-C` argument to the `pmfadm` command. See the `pmfadm`(1M) man page.

Omitting this property, or setting it to the default value of `-1`, has the same effect as omitting the `-C` option on the `pmfadm` command. That is, all children and their descendents are monitored.

`Failover_enabled` Property

This property controls the failover behavior of the resource. If this extension property is set to `TRUE`, the application fails over when the number of restarts exceeds the `Retry_count` within the `Retry_interval` number of seconds.

If this property is set to `FALSE`, the application does not restart or fail over to another node when the number of restarts exceeds the `Retry_count` within the `Retry_interval` number of seconds.

You can use this property to prevent the application resource from initiating a failover of the resource group. The default value for this property is TRUE.

Note – In future, use the Failover_mode property in place of the Failover_enabled extension property as Failover_mode better controls failover behavior. For more information, see the descriptions of the LOG_ONLY and RESTART_ONLY values for Failover_mode in the r_properties(5) man page.

Log_level Property

This property specifies the level, or type, of diagnostic messages that are logged by the GDS. You can specify NONE, INFO, or ERR for this property. When you specify NONE, diagnostic messages are not logged by the GDS. When you specify INFO, only informational messages are logged. When you specify ERR, only error messages are logged. By default, the GDS does not log diagnostic messages (NONE).

Network_aware Property

This property specifies whether your application uses the network. By default, the GDS assumes that your application is network aware, that is, uses the network (Network_aware is set to TRUE).

If your application is network aware, you must provide both the Start_command extension property and the Port_list property. If your application is nonnetwork aware, you must provide only the Start_command extension property.

Network_resources_used Property

This property specifies a list of logical host name or shared address network resources that are used by a resource. The default value for this property is the empty list. You must specify this property if the application needs to bind to one or more specific addresses. If you omit this property or you specify Null, the application listens on all addresses.

Before you create the GDS resource, a LogicalHostname or SharedAddress resource must already be configured. See the *Sun Cluster Data Services Planning and Administration Guide for Solaris OS* for information about how to configure a LogicalHostname or SharedAddress resource.

To specify a value, specify one or more resource names. Each resource name can contain one or more LogicalHostname resources or one or more SharedAddress resources. See the r_properties(5) man page for details.

Probe_command Property

This property specifies the probe command that periodically checks the health of a given application. This command must be a UNIX command with arguments that can be passed directly to a shell to probe the application. The probe command returns with an exit status of 0 if the application is running correctly.

The exit status of the probe command is used to determine the severity of the application's failure. This exit status, called the *probe status*, must be an integer between 0 (for success) and 100 (for complete failure). The probe status can also be a special value of 201, which causes the application to immediately fail over unless Failover_enabled is set to FALSE. The GDS probing algorithm uses the probe status to determine whether to restart the application locally or fail it over. See the scds_fm_action(3HA) man page for more information. If the exit status is 201, the application is immediately failed over.

If the probe command is omitted, the GDS provides its own simple probe. This probe connects to the application on the set of IP addresses that is derived from the Network_resources_used property or from the output of the scds_get_netaddr_list() function. See the scds_get_netaddr_list(3HA) man page for more information. If the connect succeeds, the connect disconnects immediately. If both the connect and disconnect succeed, the application is deemed to be running well.

Note – The probe that is provided with the GDS is only intended to be a simple substitute for the fully functioning application-specific probe.

Probe_timeout **Property**

This property specifies the timeout value for the probe command. See "Probe_command Property" on page 194 for additional information. The default for Probe_timeout is 30 seconds.

Start_timeout **Property**

This property specifies the start timeout for the start command. See "Start_command Property" on page 192 for additional information. The default for Start_timeout is 300 seconds.

Stop_command **Property**

This property specifies the command that must stop an application and only return after the application has been completely stopped. This command must be a complete UNIX command that can be passed directly to a shell to stop the application.

If the Stop_command extension property is provided, the GDS stop method starts the stop command with 80 percent of the stop timeout. Regardless of the outcome of starting the stop command, the GDS stop method sends SIGKILL with 15 percent of the stop timeout. The remaining 5 percent of the time is reserved for housekeeping overhead.

If the stop command is omitted, the GDS tries to stop the application by using the signal specified in Stop_signal.

Stop_signal **Property**

This property specifies a value that identifies the signal to stop an application through the PMF. See the signal(3HEAD) man page for a list of the integer values that you can specify. The default value is 15 (SIGTERM).

Stop_timeout Property

This property specifies the timeout for the stop command. See "Stop_command Property" on page 195 for additional information. The default for Stop_timeout is 300 seconds.

Validate_command Property

This property specifies the absolute path to a command to invoke to validate the application. If you do not provide an absolute path, the application is not validated.

Validate_timeout Property

This property specifies the timeout for the validate command. See "Validate_command Property" on page 196 for additional information. The default for Validate_timeout is 300 seconds.

Using Agent Builder to Create a Service That Uses the GDS

You can use Agent Builder to create the service that uses the GDS. Agent Builder is described in more detail in Chapter 9, "Sun Cluster Agent Builder."

Creating and Configuring GDS-Based Scripts

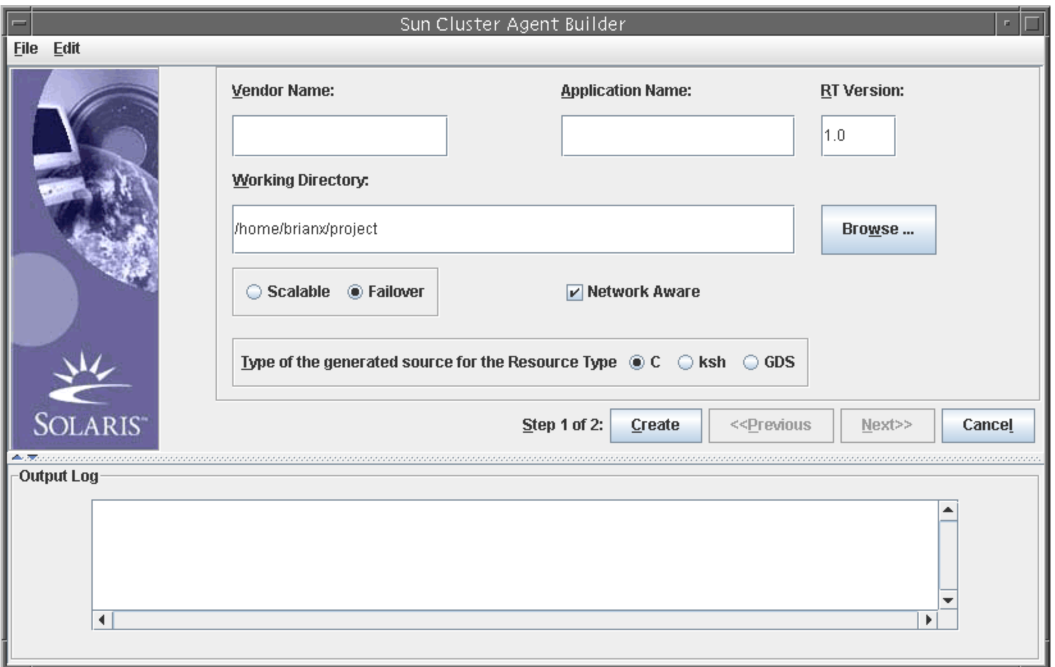
▼ How to Start Agent Builder and Create the Scripts

1. **Become superuser or assume a role that provides** solaris.cluster.modify **RBAC authorization.**

2. **Start Agent Builder.**

```
# /usr/cluster/bin/scdsbuilder
```

3. **The Agent Builder Create screen appears.**



4 Type the vendor name.

5 Type the application name.

Note – Starting with the Solaris 9 OS, the combination of vendor name and application name can exceed nine characters. This combination is used as the name of the package for the scripts.

---

6 Go to the working directory.

You can use the Browse drop-down menu to select the directory rather than typing the path.

7 Select whether the data service is scalable or failover.

You do not need to select Network Aware because that setting is the default when you create the GDS.

8 Select GDS.

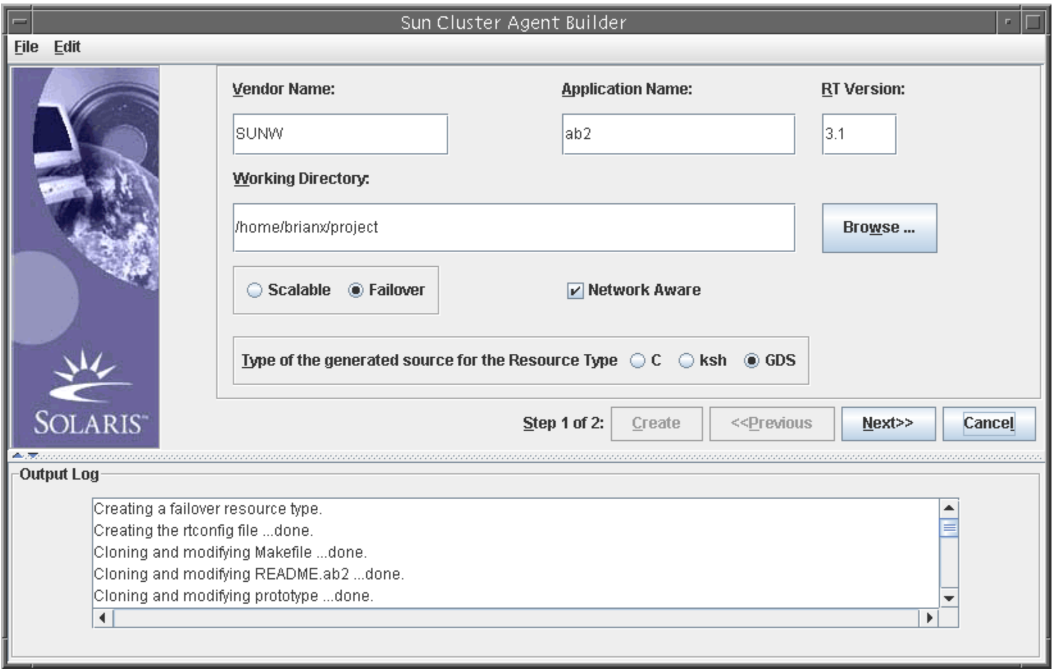
9 (Optional) Change the RT version from the default value that is shown.

Note – You cannot use the following characters in the RT Version field: space, tab, slash (`/`), backslash (`\`), asterisk (`*`), question mark (`?`), comma (`,`), semicolon (`;`), left square bracket (`[`), or right square bracket (`]`).

---

10 Click Create.

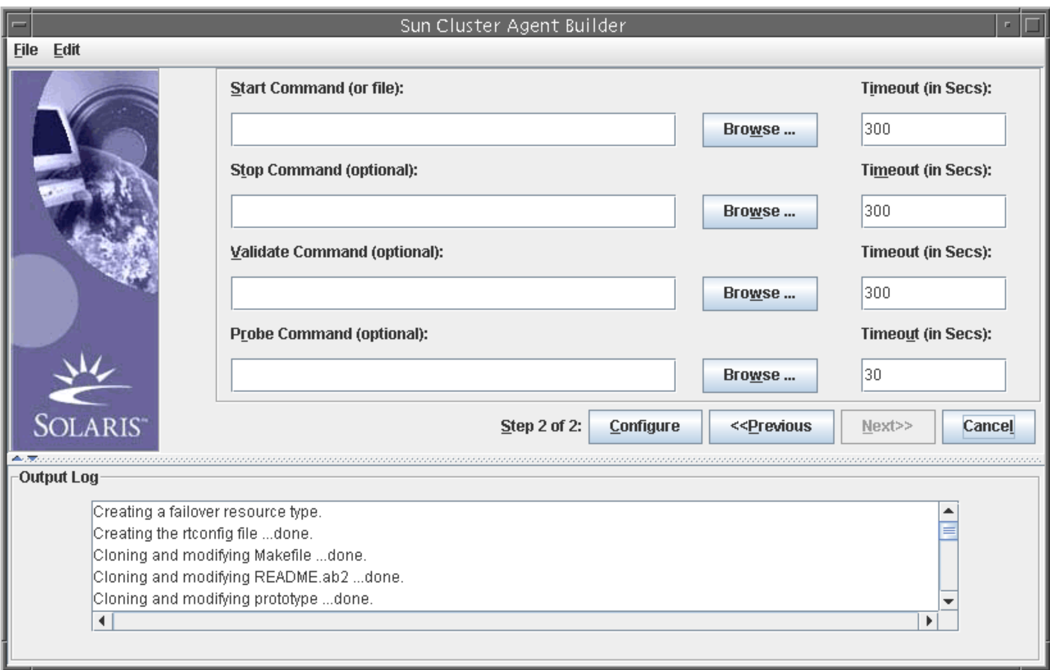
Agent Builder creates the scripts. The results are displayed in the Output Log area.



Note that the Create button is grayed out. You can now configure the scripts.

11 Click Next.

The Configure screen appears.



▼ How to Configure the Scripts

After creating the scripts, you need to configure the new service.

1. **Type the location of the start command, or click Browse to locate the start command.**

   You can specify property variables. Property variables are described in "Using Property Variables" on page 174.

2. **(Optional) Type the location of the stop command, or click Browse to locate the stop command.**

   You can specify property variables. Property variables are described in "Using Property Variables" on page 174.

3. **(Optional) Type the location of the validate command, or click Browse to locate the validate command.**

   You can specify property variables. Property variables are described in "Using Property Variables" on page 174.

4. **(Optional) Type the location of the probe command, or click Browse to locate the probe command.**

   You can specify property variables. Property variables are described in "Using Property Variables" on page 174.

5 (Optional) Specify new timeout values for the start, stop, validate, and probe commands.

6 Click Configure.

Agent Builder configures the scripts.

Note – Agent Builder concatenates the vendor name and the application name to create the package name.

A package for scripts is created and placed in the following directory:

working-dir/*vendor-name-application*/pkg

For example, /export/wdir/NETapp/pkg.

7 On each node of the cluster, become superuser or assume a role that provides solaris.cluster.modify **RBAC authorization.**

8 On each node of the cluster, install the completed package.

```
# cd /export/wdir/NETapp/pkg
# pkgadd -d . NETapp
```

The following files are installed by pkgadd:

```
/opt/NETapp
/opt/NETapp/README.app
/opt/NETapp/man
/opt/NETapp/man/man1m
/opt/NETapp/man/man1m/removeapp.1m
/opt/NETapp/man/man1m/startapp.1m
/opt/NETapp/man/man1m/stopapp.1m
/opt/NETapp/man/man1m/app_config.1m
/opt/NETapp/util
/opt/NETapp/util/removeapp
/opt/NETapp/util/startapp
/opt/NETapp/util/stopapp
/opt/NETapp/util/app_config
```

Note – The man pages and script names correspond to the application name that you typed previously on the Create screen, preceded by the script name (for example, startapp).

---

9 On one node of the cluster, configure the resources and start the application.

```
# /opt/NETapp/util/startapp -h logicalhostname -p port-and-protocol-list
```

The arguments to the startapp script vary according to the type of resource: failover or scalable.

Note – To determine the command line that you need to type, check the customized man page, or run the `startapp` script without any arguments to display a usage statement.

To view the man pages, you need to specify the path to the man page. For example, to view the `startapp`(1M) man page, type:

```
# man -M /opt/NETapp/man startapp
```

To display a usage statement, type:

```
# /opt/NETapp/util/startapp
 The resource name of LogicalHostname or SharedAddress must be
 specified. For failover services:
 Usage: startapp -h logicalhostname
          -p port-and-protocol-list
          [-n ipmpgroup-adapter-list]
 For scalable services:
 Usage: startapp -h shared-address-name
          -p port-and-protocol-list
          [-l load-balancing-policy]
          [-n ipmpgroup/adapter-list]
          [-w load-balancing-weights]
```

Output From Agent Builder

Agent Builder generates three scripts and a configuration file based on input that you provide when you create the package. The configuration file specifies the names of the resource group and the resource type.

The scripts are as follows:

- **Start script.** Configures the resources and starts the application that is under RGM control.
- **Stop script.** Stops the application and takes down resources and resource groups.
- **Remove script.** Removes the resources and resource groups that are created by the start script.

These scripts have the same interface and behavior as the utility scripts that are generated by Agent Builder for non-GDS-based data services. The scripts are put in a Solaris package that you can reuse across multiple clusters.

You can customize the configuration file to provide your own names for resource groups or other arguments that are normally given as arguments to the `scrgadm` and `scswitch` commands. If you do not customize the scripts, Agent Builder provides default values for these arguments.

Using Sun Cluster Administration Commands to Create a Service That Uses the GDS

This section describes how to input arguments to the GDS. You use the existing Sun Cluster administration commands, such as `clresourcetype`, `clresourcegroup`, and `clresource` to maintain and administer the GDS.

If the scripts provide adequate functionality, you do not need to use the lower-level administration commands that are shown in this section. However, you can use the lower-level administration commands if you need to have finer control over the GDS-based resource. These commands are executed by the scripts.

▼ How to Use Sun Cluster Administration Commands to Create a Highly Available Service That Uses the GDS

1. **Become superuser or assume a role that provides** `solaris.cluster.modify` **RBAC authorization.**

2. **Register the resource type** `SUNW.gds`**.**

```
# clresourcetype register SUNW.gds
```

3. **Create the resource group that contains the** `LogicalHostname` **resource and the failover service itself.**

```
# clresourcegroup create haapp_rg
```

4. **Create the resource for the** `LogicalHostname` **resource.**

```
# clreslogicalhostname create -g haapp_rg hhead
```

5. **Create the resource for the failover service itself.**

```
# clresource create -g haapp_rg -t SUNW.gds
        -p Validate_command="/export/app/bin/configtest" \
        -p Scalable=false -p Start_timeout=120 \
        -p Stop_timeout=120 -p Probe_timeout=120 \
        -p Port_list="2222/tcp" \
        -p Start_command="/export/ha/appctl/start" \
        -p Stop_command="/export/ha/appctl/stop" \
        -p Probe_command="/export/app/bin/probe" \
        -p Child_mon_level=0 -p Network_resources_used=hhead \
        -p Failover_enabled=TRUE -p Stop_signal=9 haapp_rs
```

6. **Bring the resource group** `haapp_rg` **online in a managed state.**

```
# clresourcegroup online -M haapp_rg
```

▼ How to Use Sun Cluster Administration Commands to Create a Scalable Service That Uses the GDS

1. **Become superuser or assume a role that provides** `solaris.cluster.modify` **RBAC authorization.**

2. **Register the resource type** `SUNW.gds`**.**

```
# clresourcetype register SUNW.gds
```

3. **Create the resource group for the** `SharedAddress` **resource.**

```
# clresourcegroup create sa_rg
```

4. **Create the** `SharedAddress` **resource** `hhead` **in resource group** `sa_rg`**.**

```
# clressharedaddress create -g sa_rg hhead
```

5. **Create the resource group for the scalable service.**

```
# clresourcegroup create -S -p RG_dependencies=sa_reg app_rg
```

6. **Create the resource for the scalable service.**

```
# clresource create -g app_rg -t SUNW.gds
        -p Validate_command="/export/app/bin/configtest" \
        -p Scalable=TRUE -p Start_timeout=120 \
        -p Stop_timeout=120 -p Probe_timeout=120 \
        -p Port_list="2222/tcp" \
        -p Start_command="/export/app/bin/start" \
        -p Stop_command="/export/app/bin/stop" \
        -p Probe_command="/export/app/bin/probe" \
        -p Child_mon_level=0 -p Network_resource_used=hhead \
        -p Failover_enabled=TRUE -p Stop_signal=9 app_rs
```

7. **Bring the resource group that contains the network resources online.**

```
# clresourcegroup online sa_reg
```

8. **Bring the resource group** `app_rg` **online in a managed state.**

```
# clresourcegroup online -M app_reg
```

Command-Line Interface for Agent Builder

Agent Builder incorporates a command-line interface that provides the same functionality that the GUI provides. This interface consists of the commands `scdscreate` and `scdsconfig`. See the `scdscreate`(1HA) and `scdsconfig`(1HA) man pages.

▼ How to Use the Command-Line Version of Agent Builder to Create a Service That Uses GDS

This section describes how to use the command-line interface to perform the same set of steps shown in "Using Agent Builder to Create a Service That Uses the GDS" on page 196.

1 Become superuser or assume a role that provides `solaris.cluster.modify` RBAC authorization.

2 Create the service.

- For a failover service, type:

```
# scdscreate -g -V NET -T app -d /export/wdir
```

- For a scalable service, type:

```
# scdscreate -g -s -V NET -T app -d /export/wdir
```

Note – The `-d` argument is optional. If you do not specify this argument, the current directory becomes the working directory.

---

3 Configure the service.

```
# scdsconfig -s "/export/app/bin/start" \
-e "/export/app/bin/configtest" \
-t "/export/app/bin/stop" \
-m "/export/app/bin/probe" -d /export/wdir
```

You can specify property variables. Property variables are described in "Using Property Variables" on page 174.

Note – Only the start command (`scdsconfig -s`) is required. All other options and arguments are optional.

---

4 On each node of the cluster, install the completed package.

```
# cd /export/wdir/NETapp/pkg
# pkgadd -d . NETapp
```

The following files are installed by pkgadd:

```
/opt/NETapp
/opt/NETapp/README.app
/opt/NETapp/man
/opt/NETapp/man/man1m
/opt/NETapp/man/man1m/removeapp.1m
/opt/NETapp/man/man1m/startapp.1m
/opt/NETapp/man/man1m/stopapp.1m
/opt/NETapp/man/man1m/app_config.1m
/opt/NETapp/util
/opt/NETapp/util/removeapp
/opt/NETapp/util/startapp
/opt/NETapp/util/stopapp
/opt/NETapp/util/app_config
```

Note – The man pages and script names correspond to the application name that you typed previously on the Create screen, preceded by the script name (for example, startapp).

---

5 **On one node of the cluster, configure the resources and start the application.**

\# **/opt/NETapp/util/startapp -h** *logicalhostname* **-p** *port-and-protocol-list*

The arguments to the startapp script vary according to the type of resource: failover or scalable.

Note – To determine the command line that you need to type, check the customized man page or run the `startapp` script without any arguments to display a usage statement.

To view the man pages, you need to specify the path to the man page. For example, to view the `startapp`(1M) man page, type:

```
# man -M /opt/NETapp/man startapp
```

To display a usage statement, type:

```
# /opt/NETapp/util/startapp
 The resource name of LogicalHostname or SharedAddress must be specified.
 For failover services:
 Usage: startapp -h logicalhostname
          -p port-and-protocol-list
          [-n ipmpgroup/adapter-list]
 For scalable services:
 Usage: startapp -h shared-address-name
          -p port-and-protocol-list
          [-l load-balancing-policy]
          [-n ipmpgroup/adapter-list]
          [-w load-balancing-weights]
```

CHAPTER 11

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 207
- "Property Functions" on page 209
- "Network Resource Access Functions" on page 209
- "PMF Functions" on page 211
- "Fault Monitor Functions" on page 212
- "Utility Functions" on page 212

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_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.HAStoragePlus` resources that are used by a resource:

`scds_hasp_check(3HA)` – Retrieves status information about `SUNW.HAStoragePlus` resources that are used by a resource. This information is obtained from the state (online or otherwise) of all `SUNW.HAStoragePlus` resources on which the resource depends by using the `Resource_dependencies` or `Resource_dependencies_weak` system properties that are defined for the resource. See the `SUNW.HAStoragePlus(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)` and `scds_error_string_i18n(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 *caches* 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` 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_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) and 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.

CHAPTER 12

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, Sun 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 Sun 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 213
- "How a Client Registers With the Server" on page 217
- "How the Server Replies to a Client" on page 219
- "How the Server Delivers Events to a Client" on page 221
- "How the CRNP Authenticates Clients and the Server" on page 223
- "Example of Creating a Java Application That Uses the CRNP" on page 224

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 voting 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 Sun 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.

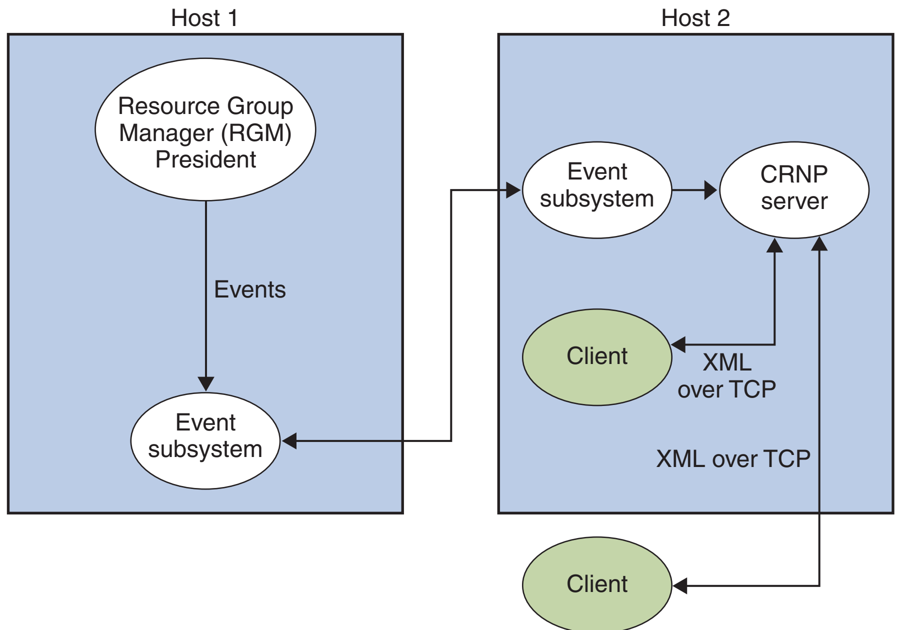


FIGURE 12–1 Flow of Events Between CRNP Components

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.

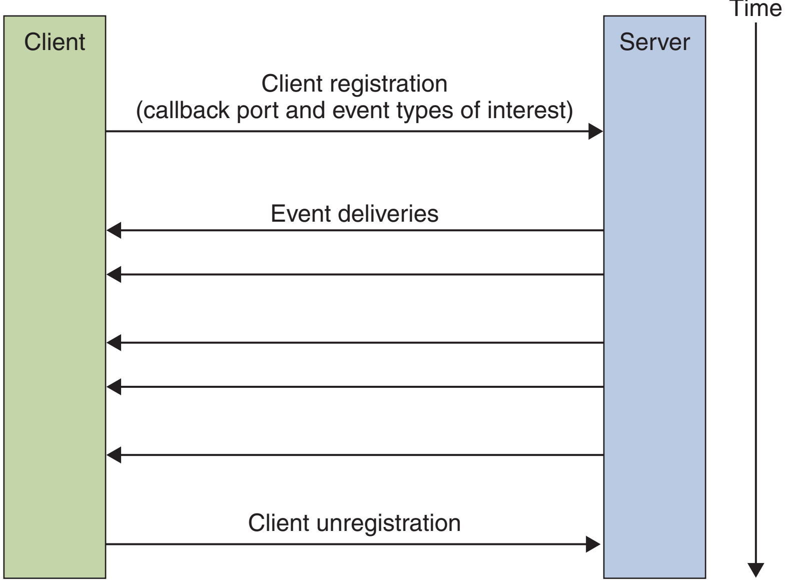


FIGURE 12–2 Flow of Communication Between a Client and a Server

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`	This message takes four forms: `ADD_CLIENT`, `REMOVE_CLIENT`, `ADD_EVENTS`, and `REMOVE_EVENTS`. Each of these forms contains the following information: ■ Protocol version ■ Callback port in ASCII format (not binary format) `ADD_CLIENT`, `ADD_EVENTS`, and `REMOVE_EVENTS` also contain an unbounded list of event types. Each of these forms includes the following information: ■ Event class ■ Event subclass (optional) ■ List of the name and value pairs (optional) 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 F, "Document Type Definitions for the CRNP."
`SC_REPLY`	This message contains the following information: ■ Protocol version ■ Error code ■ Error message The DTD from which the classes of `SC_REPLY` are generated is `SC_REPLY`. This DTD is described in more detail in Appendix F, "Document Type Definitions for the CRNP."

`SC_EVENT`	This message contains the following information:
	■ Protocol version
	■ Event class
	■ Event subclass
	■ Vendor
	■ Publisher
	■ Name and value pairs list (0 or more name and value pair data structures)
	  ■ Name (string)
	  ■ Value (string or string array)
	
	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 F, "Document Type Definitions for the CRNP."

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
EC_Cluster ESC_cluster_membership	Required: none Optional: none	Registers for all cluster membership change events (node death or join cluster)
EC_Cluster ESC_cluster_rg_state	One required, as follows: rg_*name* Value type: string Optional: none	Registers for all state change events for resource group *name*
EC_Cluster ESC_cluster_r_state	One required, as follows: r_*name* Value type: string Optional: none	Registers for all state change events for resource *name*
EC_Cluster None	Required: none Optional: none	Registers for all Sun Cluster events

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 358 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 ESC_cluster_membership	Publisher: rgm Vendor: SUNW	Name: node_list Value type: string array Name: state_list The state_list 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 –1, 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. Value type: string array

Class and Subclass	Publisher and Vendor	Name and Value Pairs
EC_Cluster ESC_cluster_rg_state	Publisher: rgm Vendor: SUNW	Name: rg_name Value type: string Name: node_list Value type: string array Name: state_list The state_list contains string representations of the state of the resource group. Valid values are those values that you can retrieve with the scha_cmds(1HA) commands. Value type: string array
EC_Cluster ESC_cluster_r_state	Publisher: rgm Vendor: SUNW	Name: r_name Value type: string Name: node_list Value type: string array Name: state_list The state_list contains string representations of the state of the resource. Valid values are those values that you can retrieve with the scha_cmds(1HA) commands. Value type: string array

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 `http://java.sun.com/webservices/jaxp/`.
- This example presents pieces of an application, which can be found in its entirety in Appendix G, "`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 G, "`CrnpClient.java` Application."
- For brevity, comments are excluded from the sample code in this chapter. The complete application in Appendix G, "`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 `http://java.sun.com/webservices/jaxp/`.

   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-rootjaxp-api. \
jar:jaxp-rootsax.jar:jaxp-rootxalan.jar:jaxp-root/xercesImpl \
.jar:jaxp-root/xsltc.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-rootjaxp-api. \
jar:jaxp-rootsax.jar:jaxp-rootxalan.jar:jaxp-root/xercesImpl \
.jar:jaxp-root/xsltc.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 G, "`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.localPort, 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 `http://java.sun.com/webservices/jaxp/` 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 228.

`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 G, "`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 http://java.sun.com/webservices/jaxp/.

- **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 http://java.sun.com/webservices/jaxp/ 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 http://java.sun.com/webservices/jaxp/ 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 229 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 236.

```
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
                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
                        nl = elem.getElementsByTagName("VALUE");
                        if (nl.getLength() != 1) {
                           System.out.println("Error in parsing: can't find "
                               + "VALUE 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
                        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 227.

```
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. **Become superuser or assume a 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 G, "`CrnpClient.java` Application."

A P P E N D I X A

Standard Properties

This appendix describes the standard resource type, resource, and resource group properties. This appendix also describes the resource property attributes that are available for changing system-defined properties and creating extension properties.

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

This appendix covers the following topics:

- "Resource Type Properties" on page 243
- "Resource Properties" on page 253
- "Resource Group Properties" on page 273
- "Resource Property Attributes" on page 287

Resource Type Properties

The following information describes the resource type properties that are defined by the Sun Cluster software.

The property values are categorized as follows:

- **Required**. The property requires an explicit value in the Resource Type Registration (RTR) file. Otherwise, the object to which the property belongs cannot be created. A space or the empty string is not allowed as a value.
- **Conditional**. To exist, the property must be declared in the RTR file. Otherwise, the RGM does not create the property and the property is not available to administrative utilities. A space or the empty string is allowed. If the property is declared in the RTR file but no value is specified, the RGM supplies a default value.

- **Conditional or Explicit**. To exist, the property must be declared in the RTR file with an explicit value. Otherwise, the RGM does not create the property and the property is not available to administrative utilities. A space or the empty string is not allowed.
- **Optional**. The property can be declared in the RTR file. If the property is not declared in the RTR file, the RGM creates it and supplies a default value. If the property is declared in the RTR file but no value is specified, the RGM supplies the same default value as if the property was not declared in the RTR file.
- **Query-only** – Cannot be set directly by an administrative tool.

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.

Property names are shown first, followed by a description.

Note – Resource type property names, such as `API_version` and `Boot`, are *not* case sensitive. You can use any combination of uppercase and lowercase letters when you specify property names.

`API_version` (`integer`)

The minimum version of the resource management API that is required to support this resource type implementation.

The following information summarizes the maximum `API_version` that is supported by each release of Sun Cluster.

Before and up to 3.1	2
3.1 10/03	3
3.1 4/04	4
3.1 9/04	5
3.1 8/05	6
3.2	7
3.2 2/08	8
3.2 1/09	9

Declaring a value for `API_version` that is greater than `2` in the RTR file prevents that resource type from being installed on a version of Sun Cluster that supports a lower maximum version. For example, if you declare `API_version=7` for a resource type, that resource type cannot be installed on any version of Sun Cluster that was released before 3.2.

Note – If you do not declare this property or set this property to the default value (2), the data service can be installed on any version of Sun Cluster starting with Sun Cluster 3.0.

Category: Optional

Default: 2

Tunable: NONE

Boot (string)

An optional callback method that specifies the path to the Boot method program. The RGM runs the Boot method for each managed resource of this type, on a node that joins or rejoins the cluster.

The set of nodes on which Boot, Init, Fini, or Validate methods are run is determined by the setting of the resource types Init_nodes property. You can set the Init_nodes property to RG_PRIMARIES, which indicates the nodes that are specified in the resource type's Installed_nodes property.

Category: Conditional or Explicit

Default: No default

Tunable: NONE

Failover (boolean)

If you set this property to TRUE, resources of this type cannot be configured in any group that can be online on multiple nodes at the same time.

You use this resource-type property in combination with the Scalable resource property, as follows:

If the value of the Failover resource type is	If the value of the Scalable resource is	Description
TRUE	TRUE	Do not specify this illogical combination.
TRUE	FALSE	Specify this combination for a failover service.

If the value of the `Failover` resource type is	If the value of the `Scalable` resource is	Description
`FALSE`	`TRUE`	Specify this combination for a scalable service that uses a `SharedAddress` resource for network load balancing. The *Sun Cluster Concepts Guide for Solaris OS* describes `SharedAddress` in more detail. You can configure a scalable resource to run in a global-cluster non-voting node. But, do not configure a scalable resource to run in multiple global-cluster non-voting nodes on the same Solaris host.
`FALSE`	`FALSE`	Use this combination to select a multi-master service that does not use network load balancing. You can use a scalable service of this type in zones.

The description of `Scalable` in the `r_properties(5)` man page and Chapter 3, "Key Concepts for System Administrators and Application Developers," in *Sun Cluster Concepts Guide for Solaris OS* contain additional information.

Category: Optional

Default: `FALSE`

Tunable: `NONE`

`Fini` (`string`)

An optional callback method that specifies the path to the `Fini` method program. The RGM runs the `Fini` method when a resource of this type is no longer managed by the RGM.

The `Fini` method usually undoes any initializations that were performed by the `Init` method.

The set of nodes on which `Boot`, `Init`, `Fini`, or `Validate` methods are run is determined by the setting of the resource types `Init_nodes` property. You can set the `Init_nodes` property to `RG_PRIMARIES`, which indicates the nodes that are specified in the resource type's `Installed_nodes` property.

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_primaries` or `RT_installed_nodes`. For most resource types, `Init_nodes` is set to `RG_primaries`, 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.

Category: Conditional or Explicit

Default: No default

Tunable: `NONE`

`Global_zone` (`boolean`)

A Boolean value that, if declared in the RTR file, indicates whether the methods of this resource type execute in the global zone, that is, either a zone-cluster node or a global-cluster non-voting node. If this property is set to `TRUE`, methods execute in the global zone even if the resource group that contains the resource runs in a non-global zone. Set this property to `TRUE` only for services that can be managed only from the global zone, such as network addresses and file systems.

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.

Category: Optional

Default: `FALSE`

Tunable: `ANYTIME`

`Init` (`string`)

An optional callback method that specifies the path to the `Init` method program. The RGM runs the `Init` method when a resource of this type becomes managed by the RGM.

The set of nodes on which Boot, Init, Fini, or Validate methods are run is determined by the setting of the resource types Init_nodes property. You can set the Init_nodes property to RG_PRIMARIES, which indicates the nodes that are specified in the resource type's Installed_nodes property.

Category: Conditional or Explicit

Default: No default

Tunable: NONE

Init_nodes (enum)

Indicates the nodes on which the RGM is to call the Init, Fini, Boot, and Validate methods. You can set this property to RG_PRIMARIES (just the nodes that can master the resource) or RT_INSTALLED_NODES (all nodes on which the resource type is installed).

Category: Optional

Default: RG_PRIMARIES

Tunable: NONE

Installed_nodes (string_array)

A list of the cluster node names on which the resource type can be run. Specify an asterisk (*) to explicitly include all cluster nodes, which is the default.

Category: The cluster administrator can configure this property

Default: All cluster nodes

Tunable: ANYTIME

Is_logical_hostname (boolean

TRUE indicates that this resource type is some version of the LogicalHostname resource type that manages failover Internet Protocol (IP) addresses.

Category: Query-only

Default: No default

Tunable: NONE

Is_shared_address (boolean)

TRUE indicates that this resource type is some version of the SharedAddress resource type that manages shared Internet Protocol (IP) addresses.

Category: Query-only

Default: No default

Tunable: NONE

Monitor_check (string)
An optional callback method: the path to the program that the RGM runs before performing a monitor-requested failover of a resource of this type. If the monitor-check program exits with nonzero on a node, any attempt to fail over to that node as a result of calling scha_control with the GIVEOVER tag is prevented.

Category: Conditional or Explicit

Default: No default

Tunable: NONE

Monitor_start (string)
An optional callback method: the path to the program that the RGM runs to start a fault monitor for a resource of this type.

Category: Conditional or Explicit

Default: No default

Tunable: NONE

Monitor_stop (string)
A callback method that is required if Monitor_start is set: the path to the program that the RGM runs to stop a fault monitor for a resource of this type.

Category: Conditional or Explicit

Default: No default

Tunable: NONE

Pkglist (string_array)
An optional list of packages that are included in the resource type installation.

Category: Conditional or Explicit

Default: No default

Tunable: NONE

Postnet_stop (string)
An optional callback method: the path to the program that the RGM runs after calling the Stop method of any network-address resources on which a resource of this type depends. After the network interfaces are configured down, this method must perform Stop actions.

Category: Conditional or Explicit

Default: No default

Tunable: NONE

`Prenet_start` (`string`)
An optional callback method: the path to the program that the RGM runs before the RGM calls the `Start` method of any network-address resources on which a resource of this type depends. This method performs `Start` actions that must be performed before network interfaces are configured.

Category: Conditional or Explicit

Default: No default

Tunable: `NONE`

`Proxy` (`boolean`)
A Boolean value that indicates whether a resource of this type is a proxy resource.

A *proxy resource* is a Sun Cluster resource that imports the state of a resource from another cluster framework such as Oracle Cluster Ready Services (CRS). Oracle CRS, which is now known as Oracle clusterware CRS, is a platform-independent set of system services for cluster environments.

A proxy resource type uses the `Prenet_start` method to start a daemon that monitors the state of the external (proxied) resource. The `Postnet_stop` method stops the monitoring daemon. The monitoring daemon issues the `scha_control` command with the `CHANGE_STATE_ONLINE` or the `CHANGE_STATE_OFFLINE` tag to set the proxy resource's state to `Online` or to `Offline`, respectively. The `scha_control()` function similarly uses the `SCHA_CHANGE_STATE_ONLINE` and `SCHA_CHANGE_STATE_OFFLINE` tags. See the `scha_control`(1HA) and `scha_control`(3HA) man pages for more information.

If set to `TRUE`, the resource is a proxy resource.

Category: Optional

Default: `FALSE`

Tunable: `NEVER`

`Resource_list` (`string_array`)
The list of all resources of the resource type. The cluster administrator does not set this property directly. Rather, the RGM updates this property when the cluster administrator adds or removes a resource of this type to or from any resource group.

Category: Query-only

Default: Empty list

Tunable: `NONE`

`Resource_type` (`string`)
The name of the resource type. To view the names of the currently registered resource types, use:

```
resourcetype show +
```

In Sun Cluster 3.1 and Sun Cluster 3.2, a resource type name includes the version, which is mandatory:

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

The three components of the resource type name are properties that are specified in the RTR file as *vendor-id*, *resource-type*, and *rt-version*. The `resourcetype` command inserts the period (.) and colon (:) delimiters. The *rt-version* suffix of the resource type name is the same value as the `RT_version` property. To ensure that the *vendor-id* is unique, use the stock symbol of the company that is creating the resource type. Resource type names that were created before Sun Cluster 3.1 continue to use the syntax:

vendor-id`.`*resource-type*

Category: Required

Default: Empty string

Tunable: `NONE`

`RT_basedir` (`string`)
The directory path that is used to complete relative paths for callback methods. This path must be set to the directory in which the resource type packages are installed. The path must be a complete path, that is, it must start with a forward slash (`/`).

Category: Required unless all method path names are absolute

Default: No default

Tunable: `NONE`

`RT_description` (`string`)
A brief description of the resource type.

Category: Conditional

Default: Empty string

Tunable: `NONE`

`RT_system` (`boolean`)
If the `RT_system` property is `TRUE` for a resource type, you cannot delete the resource type (**`resourcetype unregister`** *resource-type-name*) . This property prevents the accidental deletion of resource types, such as `LogicalHostname`, that are used to support the cluster infrastructure. However, you can apply the `RT_system` property to any resource type.

To delete a resource type whose `RT_system` property is set to `TRUE`, you must first set the property to `FALSE`. Use care when you delete a resource type whose resources support cluster services.

Category: Optional

Default: `FALSE`

Tunable: ANYTIME

RT_version (string)
Starting with the Sun Cluster 3.1 release, a mandatory version string that identifies this resource type implementation. This property was optional in Sun Cluster 3.0. The RT_version is the suffix component of the full resource type name.

Category: Conditional/Explicit or Required

Default: No default

Tunable: NONE

Single_instance (boolean)
If TRUE, indicates that only one resource of this type can exist in the cluster.

Category: Optional

Default: FALSE

Tunable: NONE

Start (string)
A callback method: the path to the program that the RGM runs to start a resource of this type.

Category: Required unless the RTR file declares a Prenet_start method

Default: No default

Tunable: NONE

Stop (string)
A callback method: the path to the program that the RGM runs to stop a resource of this type.

Category: Required unless the RTR file declares a Postnet_stop method

Default: No default

Tunable: NONE

Update (string)
An optional callback method: the path to the program that the RGM runs when properties of a running resource of this type are changed.

Category: Conditional or Explicit

Default: No default

Tunable: NONE

`Validate` (`string`)
An optional callback method that specifies the path to the `Validate` method program. The RGM runs the `Validate` method to check values for properties of resources of this type.

The set of nodes on which `Boot`, `Init`, `Fini`, or `Validate` methods are run is determined by the setting of the resource types `Init_nodes` property. You can set the `Init_nodes` property to `RG_PRIMARIES`, which indicates the nodes that are specified in the resource type's `Installed_nodes` property.

Category: Conditional or Explicit

Default: No default

Tunable: `NONE`

`Vendor_ID` (`string`)
See the `Resource_type` property.

Category: Conditional

Default: No default

Tunable: `NONE`

Resource Properties

This section describes the resource properties that are defined by the Sun Cluster software.

The property values are categorized as follows:

- **Required**. The cluster administrator must specify a value when he or she creates a resource with an administrative utility.
- **Optional**. If the cluster administrator does not specify a value when he or she creates a resource group, the system supplies a default value.
- **Conditional**. The RGM creates the property only if the property is declared in the RTR file. Otherwise, the property does not exist and is not available to cluster administrators. A conditional property that is declared in the RTR file is optional or required, depending on whether a default value is specified in the RTR file. For details, see the description of each conditional property.
- **Query-only**. Cannot be set directly by an administrative tool.

The `Tunable` attribute, which is described in "Resource Property Attributes" on page 287, lists whether and when you can update resource properties, as follows:

`FALSE` or `NONE` Never

`TRUE` or `ANYTIME` Any time

AT_CREATION	When the resource is added to a cluster
WHEN_DISABLED	When the resource is disabled

Property names are shown first, followed by a description.

Affinity_timeout (integer)
Length of time in seconds during which connections from a given client IP address for any service in the resource are sent to the same server node.

This property is relevant only when Load_balancing_policy is either Lb_sticky or Lb_sticky_wild. In addition, Weak_affinity must be set to FALSE.

This property is used only for scalable services.

Category: Optional

Default: No default

Tunable: ANYTIME

Boot_timeout for each callback method in the Type (integer)
A time lapse, in seconds, after which the RGM concludes that an invocation of this method has failed. For a given resource type, timeout properties are defined only for those methods that are declared in the RTR file.

Category: Conditional or Optional

Default: 3600 (one hour), if the method itself is declared in the RTR file

Tunable: ANYTIME

Cheap_probe_interval (integer)
The number of seconds between invocations of a quick fault probe of the resource. This property is created by the RGM and is available to the cluster administrator only if it is declared in the RTR file. This property is optional if a default value is specified in the RTR file.

If the Tunable attribute is not specified in the RTR file, the Tunable value for the property is WHEN_DISABLED.

Category: Conditional

Default: No default

Tunable: WHEN_DISABLED

Extension properties
Extension properties as declared in the RTR file of the resource's type. The implementation of the resource type defines these properties. "Resource Property Attributes" on page 287 contains information about the individual attributes that you can set for extension properties.

Category: Conditional

Default: No default

Tunable: Depends on the specific property

Failover_mode (enum)

Modifies the recovery actions that the RGM takes when a resource fails to start or to stop successfully, or when a resource monitor finds a resource to be unhealthy and consequently requests a restart or failover.

NONE, SOFT, or HARD (method failures)

These settings affect only failover behavior when a start or stop method (Prenet_start, Start, Monitor_stop, Stop, Postnet_stop) fails. The RESTART_ONLY and LOG_ONLY settings can also affect whether the resource monitor can initiate the execution of the scha_control command or the scha_control() function. See the scha_control(1HA) and the scha_control(3HA) man pages. NONE indicates that the RGM is not to take any recovery action when one of the previously listed start or stop methods fails. SOFT or HARD indicates that if a Start or Prenet_start method fails, the RGM is to relocate the resource's group to a different node. For Start or Prenet_start failures, SOFT and HARD are the same.

For failure of a stop method (Monitor_stop, Stop, or Postnet_stop), SOFT is the same as NONE. If Failover_mode is set to HARD when one of these stop methods fails, the RGM reboots the node to force the resource group offline. The RGM might then attempt to start the group on another node.

RESTART_ONLY or LOG_ONLY

Unlike NONE, SOFT, and HARD, which affect failover behavior when a start or stop method fails, RESTART_ONLY and LOG_ONLY affect all failover behavior. Failover behavior includes monitor-initiated (scha_control) restarts of resources and resource groups, and giveovers that are initiated by the resource monitor (scha_control). RESTART_ONLY indicates that the monitor can run scha_control to restart a resource or a resource group. The RGM allows Retry_count restarts within Retry_interval. If Retry_count is exceeded, no further restarts are permitted.

Note – A negative value of Retry_count, which is permitted by some but not all resource types, specifies an unlimited number of resource restarts. A more dependable way to specify unlimited restarts is to do the following:

- Set Retry_interval to a small value such as 1 or 0.
- Set Retry_count to a large value such as 1000.

If the resource type does not declare the Retry_count and Retry_interval properties, an unlimited number of resource restarts is permitted.

If Failover_mode is set to LOG_ONLY, no resource restarts or giveovers are permitted. Setting Failover_mode to LOG_ONLY is the same as setting Failover_mode to RESTART_ONLY with Retry_count set to zero.

RESTART_ONLY or LOG_ONLY (method failures)

If a Prenet_start, Start, Monitor_stop, Stop, or Postnet_stop method fails, RESTART_ONLY and LOG_ONLY are the same as NONE. That is, the node is neither failed over nor rebooted.

Effect of Failover_mode settings on a data service

The effect that each setting for Failover_mode has on a data service depends on whether the data service is monitored or unmonitored and whether it is based on the Data Services Development Library (DSDL).

- A data service is monitored if it implements a Monitor_start method and monitoring of the resource is enabled. The RGM starts a resource monitor by executing the Monitor_start method after starting the resource itself. The resource monitor probes the health of the resource. If the probes fail, the resource monitor might request a restart or a failover by calling the scha_control() function. For DSDL-based resources, probes might reveal partial failure (degradation) or a complete failure of the data service. Repeated partial failures accumulate to a complete failure.
- A data service is unmonitored if it does not provide a Monitor_start method or monitoring of the resource has been disabled.
- DSDL-based data services include those that are developed with Agent Builder, through the GDS, or by using the DSDL directly. Some data services, HA Oracle for example, were developed without using the DSDL.

NONE, SOFT, or HARD (probe failures)

If you set Failover_mode to NONE, SOFT, or HARD and the data service is a monitored DSDL-based service, and if the probe fails completely, the monitor calls the scha_control() function to request a restart of the resource. If probes continue to fail, the resource is restarted up to a maximum of Retry_count number of times within Retry_interval. If the probes fail again after the Retry_count number of restarts is reached, the monitor requests a failover of the resource's group to another node.

If you set Failover_mode to NONE, SOFT, or HARD and the data service is an unmonitored DSDL-based service, the only failure that is detected is the death of the resource's process tree. If the resource's process tree dies, the resource is restarted.

If the data service is a not a DSDL-based service, the restart or failover behavior depends on how the resource monitor is coded. For example, the Oracle resource monitor recovers by restarting the resource or the resource group, or by failing over the resource group.

RESTART_ONLY (probe failures)

If you set Failover_mode to RESTART_ONLY and the data service is a monitored DSDL-based service, and if the probe fails completely, the resource is restarted Retry_count times within Retry_interval. However, if Retry_count is exceeded, the resource monitor exits, sets the resource status to FAULTED, and generates the status message "Application faulted, but not restarted. Probe quitting." At this point, although monitoring is still enabled, the resource is effectively unmonitored until it is repaired and restarted by the cluster administrator.

If you set Failover_mode to RESTART_ONLY and the data service is an unmonitored DSDL-based service, and if the process tree dies, the resource is *not* restarted.

If a monitored data service is not DSDL-based, the recovery behavior depends on how the resource monitor is coded. If you set Failover_mode to RESTART_ONLY, the resource or resource group can be restarted by a call to the scha_control() function Retry_count times within Retry_interval. If the resource monitor exceeds Retry_count, the attempt to restart fails. If the monitor calls the scha_control() function to request a failover, that request fails as well.

LOG_ONLY (probe failures)

If you set Failover_mode to LOG_ONLY for any data service, all scha_control() requests either to restart the resource or resource group or to fail over the group are precluded. If the data service is DSDL-based, a message is logged when a probe completely fails, but the resource is not restarted. If a probe fails completely more than Retry_count times within Retry_interval, the resource monitor exits, sets the resource status to FAULTED, and generates the status message "Application faulted, but not restarted. Probe quitting." At this point, although monitoring is still enabled, the resource is effectively unmonitored until it is repaired and restarted by the cluster administrator.

If you set Failover_mode to LOG_ONLY and the data service is an unmonitored DSDL-based service, and if the process tree dies, a message is logged but the resource is not restarted.

If a monitored data service is not DSDL-based, the recovery behavior depends on how the resource monitor is coded. If you set Failover_mode to LOG_ONLY, all scha_control() requests either to restart the resource or resource group or to fail over the group fail.

Category: Optional

Default: NONE

Tunable: ANYTIME

Fini_timeout for each callback method in the Type (integer)
A time lapse, in seconds, after which the RGM concludes that an invocation of this method has failed. For a given resource type, timeout properties are defined only for those methods that are declared in the RTR file.

Category: Conditional or Optional

Default: 3600 (one hour), if the method itself is declared in the RTR file

Tunable: ANYTIME

Global_zone_override (boolean)

This property is allowed only for resource types that set the Global_zone=TRUE property in the RTR file. The setting of the Global_zone_override property overrides the value of the resource type property Global_zone for the particular resource. See the rt_properties(5) man page for more information.

When the Global_zone property is set to TRUE the resource methods always execute in the global-cluster voting node.

Setting the Global_zone_override property to FALSE forces the resource methods to execute on a non-global zone, that is, either a zone-cluster node or a global-cluster non-voting node in which the resource group is configured, rather than always executing in the global zone as they usually would when the Global_zone property is set to TRUE.

This property is optional if a default value is specified in the RTR file.

If the Tunable attribute is not specified in the RTR file, the Tunable value for the property is AT_CREATION. You can set the Tunable attribute in the RTR file to AT_CREATION, WHEN_DISABLED, or ANYTIME.

Use caution when you set the Tunable attribute to Anytime in the RTR file. Changes to the Global_zone_override property take effect immediately, even if the resource is online. For example, suppose that the Global_zone_override tunability is set to ANYTIME and the Global_zone_override property is currently set to FALSE on a resource that is configured in a non-global zone. When the resource is switched online, the starting methods are executed in the non-global zone. If the Global_zone_override property is then set to TRUE and the resource is switched offline, the stopping methods are executed in the global zone. Your method code must be able to handle this possibility. If it cannot, then you must set the Tunable attribute to WHEN_DISABLED or AT_CREATION instead.

Category: Conditional or Optional

Default: TRUE

Tunable: AT_CREATION

Init_timeout for each callback method in the Type (integer)

A time lapse, in seconds, after which the RGM concludes that an invocation of this method has failed. For a given resource type, timeout properties are defined only for those methods that are declared in the RTR file.

Category: Conditional or Optional

Default: 3600 (one hour), if the method itself is declared in the RTR file

Tunable: ANYTIME

`Load_balancing_policy` (`string`)

A string that defines the load-balancing policy in use. This property is used only for scalable services. The RGM automatically creates this property if the `Scalable` property is declared in the RTR file. `Load_balancing_policy` can take the following values:

`Lb_weighted` (the default). The load is distributed among various nodes according to the weights 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'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 the IP address is coming.

Category: Conditional or Optional

Default: `Lb_weighted`

Tunable: `AT_CREATION`

`Load_balancing_weights` (`string_array`)

For scalable resources only. The RGM automatically creates this property if the `Scalable` property is declared in the RTR file. The format is *weight@node,weight@node*, where *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. 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. The empty string (“”), the default, sets a uniform distribution. Any node that is not assigned an explicit weight receives a default weight of 1.

If the `Tunable` attribute is not specified in the RTR file, the `Tunable` value for the property is `ANYTIME`. Changing this property revises the distribution for new connections only.

Category: Conditional or Optional

Default: The empty string (“”)

Tunable: `ANYTIME`

`Monitor_check_timeout` for each callback method in the Type (`integer`)

A time lapse, in seconds, after which the RGM concludes that an invocation of this method has failed. For a given resource type, timeout properties are defined only for those methods that are declared in the RTR file.

Category: Conditional or Optional

Default: `3600` (one hour), if the method itself is declared in the RTR file

Tunable: `ANYTIME`

`Monitor_start_timeout` for each callback method in the Type (`integer`)
A time lapse, in seconds, after which the RGM concludes that an invocation of this method has failed. For a given resource type, timeout properties are defined only for those methods that are declared in the RTR file.

Category: Conditional or Optional

Default: `3600` (one hour), if the method itself is declared in the RTR file

Tunable: `ANYTIME`

`Monitor_stop_timeout` for each callback method in the Type (`integer`)
A time lapse, in seconds, after which the RGM concludes that an invocation of this method has failed. For a given resource type, timeout properties are defined only for those methods that are declared in the RTR file.

Category: Conditional or Optional

Default: `3600` (one hour), if the method itself is declared in the RTR file

Tunable: `ANYTIME`

`Monitored_switch` (`enum`)
Set to `Enabled` or `Disabled` by the RGM if the cluster administrator enables or disables the monitor with an administrative utility. If `Disabled`, monitoring on the resource is stopped, although the resource itself remains online. The `Monitor_start` method is not called until monitoring is re-enabled. If the resource does not have a monitor callback method, this property does not exist.

Category: Query-only

Default: No default

Tunable: `NONE`

`Network_resources_used` (`string_array`)
A list of logical-hostname or shared-address network resources on which the 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 `Scalable` is not declared in the RTR file, `Network_resources_used` is unavailable unless it is explicitly declared in the RTR file.

This property is updated automatically by the RGM, based on the setting of the resource-dependencies properties. You do not need to set this property directly. However, if you add a resource name to this property, the resource name is automatically added to the `Resource_dependencies` property. In addition, if you delete a resource name from this property, the resource name is automatically deleted from any resource-dependencies property in which the resource also appears.

Category: Conditional or Optional

Default: The empty list

Tunable: ANYTIME

Num_resource_restarts on each cluster node (integer)

The number of restart requests that have occurred on this resource within the past *n* seconds, where *n* is the value of the Retry_interval property.

A restart request is any of the following calls:

- The scha_control(1HA) command with the RESOURCE_RESTART argument.
- The scha_control(3HA) function with the SCHA_RESOURCE_RESTART argument.
- The scha_control command with the RESOURCE_IS_RESTARTED argument.
- The scha_control() function with the SCHA_RESOURCE_IS_RESTARTED argument.

The RGM resets the restart counter to zero for a given resource on a given node whenever that resource executes one of the following:

- The scha_control command with the GIVEOVER argument.
- The scha_control() function with the SCHA_GIVEOVER argument.

The counter is reset whether the giveover attempt succeeds or fails.

If a resource type does not declare the Retry_interval property, the Num_resource_restarts property is not available for resources of that type.

Category: Query-only

Default: No default

Tunable: See description

Num_rg_restarts on each cluster node (integer)

The number of resource group restart requests that have occurred for this resource within the past *n* seconds, where *n* is the value of the Retry_interval property.

A resource group restart request is either of the following calls:

- The scha_control(1HA) command with the RESTART argument.
- The scha_control(3HA) function with the SCHA_RESTART argument.

If a resource type does not declare the Retry_interval property, the Num_rg_restarts property is not available for resources of that type.

Category: Query-only

Default: No default

Tunable: See description

`On_off_switch` (`enum`)
Set to `Enabled` or `Disabled` by the RGM if the cluster administrator enables or disables the resource with an administrative utility. If disabled, a resource is brought offline and has no callbacks run until it is re-enabled.

Category: Query-only

Default: No default

Tunable: `NONE`

`Port_list` (`string_array`)
A list of port numbers on which the server is listening. Appended to each port number is a slash (`/`) followed by the protocol that is being used by that port, for example, `Port_list=80/tcp` or `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

If the `Scalable` property is declared in the RTR file, the RGM automatically creates `Port_list`. Otherwise, this property is unavailable unless it is explicitly declared in the RTR file.

Setting up this property for Apache is described in the *Sun Cluster Data Service for Apache Guide for Solaris OS*.

Category: Conditional or Required

Default: No default

Tunable: `ANYTIME`

`Postnet_stop_timeout` for each callback method in the Type (`integer`)
A time lapse, in seconds, after which the RGM concludes that an invocation of this method has failed. For a given resource type, timeout properties are defined only for those methods that are declared in the RTR file.

Category: Conditional or Optional

Default: `3600` (one hour), if the method itself is declared in the RTR file

Tunable: `ANYTIME`

`Prenet_start_timeout` for each callback method in the Type (`integer`)
A time lapse, in seconds, after which the RGM concludes that an invocation of this method has failed. For a given resource type, timeout properties are defined only for those methods that are declared in the RTR file.

Category: Conditional or Optional

Default: 3600 (one hour), if the method itself is declared in the RTR file

Tunable: ANYTIME

R_description (string)
A brief description of the resource.

Category: Optional

Default: The empty string

Tunable: ANYTIME

Resource_dependencies (string_array)
A list of resources on which the resource has a strong dependency. A strong dependency determines the order of method calls.

A resource with resource dependencies, referred to as the *dependent resource*, cannot be started if any resource in the list, referred to as the *depended-on resource*, is not online. If the dependent resource and one of the depended-on resources in the list start at the same time, the RGM waits to start the dependent resource until the depended-on resource in the list starts. If the depended-on resource does not start, the dependent resource remains offline. The depended-on resource might not start because the resource group for the depended-on resource in the list remains offline or is in a Start_failed state. If the dependent resource remains offline because of a dependency on a depended-on resource in a different resource group that fails to start or is disabled or offline, the dependent resource's group enters a Pending_online_blocked state. If the dependent resource has a dependency on a depended-on resource in the same resource group that fails to start or is disabled or offline, the resource group does not enter a Pending_online_blocked state.

By default in a resource group, application resources have an implicit strong resource dependency on network address resources. Implicit_network_dependencies in "Resource Group Properties" on page 273 contains more information.

Within a resource group, Prenet_start methods are run in dependency order before Start methods. Postnet_stop methods are run in dependency order after Stop methods. In different resource groups, the dependent resource waits for the depended-on resource to finish Prenet_start and Start before it runs Prenet_start. The depended-on resource waits for the dependent resource to finish Stop and Postnet_stop before it runs Stop.

To specify the scope of a dependency, append the following qualifiers, including the braces ({}), to the resource name when you specify this property.

{LOCAL_NODE} Limits the specified dependency to a per-host basis. The behavior of the dependent is affected by the depended-on resource only on the same host. The dependent resource waits for the depended-on resource to start on the same host. The situation is similar for stopping and restarting.

`{ANY_NODE}`	Extends the specified dependency to any node. The behavior of the dependent is affected by the depended-on resource on any node. The dependent resource waits for the depended-on resource to start on any primary node before it starts itself. The situation is similar for stopping and restarting.
`{FROM_RG_AFFINITIES}`	Specifies that the scope of the resource dependency is derived from the `RG_affinities` relationship of the resource groups to which the resources belong. If the dependent resource's group has a positive affinity for the depended-on resource's resource group, and they are starting or stopping on the same node, the dependency is `{LOCAL_NODE}`. If no such positive affinity exists, or if the groups are starting on different nodes, the dependency is `{ANY_NODE}`.

Resource dependencies between two resources that are located in the same resource group are always `{LOCAL_NODE}`.

If you do not specify a qualifier, `FROM_RG_AFFINITIES` is used by default.

Category: Optional

Default: The empty list

Tunable: `ANYTIME`

`Resource_dependencies_offline_restart` (`string_array`)
A list of resources in the same or in different groups on which the `Resource_dependencies_offline_restart` resource has an offline-restart dependency.

This property works just as `Resource_dependencies` does, except that, if any resource in the offline-restart dependency list is stopped, this resource is stopped. If that resource in the offline-restart dependency list is subsequently restarted, this resource is restarted.

This resource cannot be started if the start of any resource in the list fails. If this resource and one of the resources in the list start at the same time, the RGM waits until the resource in the list starts before the RGM starts this resource. If the resource in this resource's `Resource_dependencies` list does not start (for example, if the resource group for the resource in the list remains offline or if the resource in the list is in a `Start_failed` state), this resource also remains offline. If this resource remains offline because of a dependency on a resource in a different resource group that fails to start, this resource's group enters a `Pending_online_blocked` state.

If this resource is brought offline at the same time as those in the list, this resource stops before those in the list. However, if this resource remains online or fails to stop, a resource in the list stops anyway.

If a fault occurs on a "depended-on" resource on a node, and the resource cannot recover, the RGM brings that resource on that node offline. The RGM also brings all of the depended-on resource's offline-restart dependents offline by triggering a restart on them. When the cluster administrator resolves the fault and reenables the depended-on resource, the RGM brings the depended-on resource's offline-restart dependents back online as well.

To specify the scope of a dependency, append the following qualifiers, including the braces ({ }), to the resource name when you specify this property.

`{LOCAL_NODE}` Limits the specified dependency to a per-host basis. The behavior of the dependent is affected by the depended-on resource only on the same host. The dependent resource waits for the depended-on resource to start on the same host. The situation is similar for stopping and restarting.

`{ANY_NODE}` Extends the specified dependency to any node. The behavior of the dependent is affected by the depended-on resource on any node. The dependent resource waits for the depended-on resource to start on any primary node before it starts itself. The situation is similar for stopping and restarting.

`{FROM_RG_AFFINITIES}` Specifies that the scope of the resource dependency is derived from the `RG_affinities` relationship of the resource groups to which the resources belong. If the dependent resource's group has a positive affinity for the depended-on resource's resource group, and they are starting or stopping on the same node, the dependency is `{LOCAL_NODE}`. If no such positive affinity exists, or if the groups are starting on different nodes, the dependency is `{ANY_NODE}`.

Resource dependencies between two resources that are located in the same resource group are always `{LOCAL_NODE}`.

If you do not specify a qualifier, `FROM_RG_AFFINITIES` is used by default.

Category: Optional

Default: The empty list

Tunable: `ANYTIME`

`Resource_dependencies_restart` (`string_array`)

A list of resources on which the resource has a restart dependency. A restart dependency determines the order of method calls.

This property works as `Resource_dependencies` does, with one addition. If any resource in the restart dependency list, referred to as a *depended-on resource*, is restarted, the resource with resource dependencies, referred to as the *dependent resource*, is restarted. After the

depended-on resource in the list comes back online, the RGM stops and restarts the dependent resource. This restart behavior occurs when the resource groups that contain the dependent and depended-on resources remain online.

A resource with resource dependencies, referred to as the *dependent resource*, cannot be started if any resource in the list, referred to as the *depended-on resource*, is not online. If the dependent resource and one of the depended-on resources in the list start at the same time, the RGM waits to start the dependent resource until the depended-on resource in the list starts. If the depended-on resource does not start, the dependent resource remains offline. The depended-on resource might not start because the resource group for the depended-on resource in the list remains offline or is in a `Start_failed` state. If the dependent resource remains offline because of a dependency on a depended-on resource in a different resource group that fails to start or is disabled or offline, the dependent resource's group enters a `Pending_online_blocked` state. If the dependent resource has a dependency on a depended-on resource in the same resource group that fails to start or is disabled or offline, the resource group does not enter a `Pending_online_blocked` state.

To specify the scope of a dependency, append the following qualifiers, including the braces ({}), to the resource name when you specify this property.

`{LOCAL_NODE}` — Limits the specified dependency to a per-host basis. The behavior of the dependent is affected by the depended-on resource only on the same host. The dependent resource waits for the depended-on resource to start on the same host. The situation is similar for stopping and restarting.

`{ANY_NODE}` — Extends the specified dependency to any node. The behavior of the dependent is affected by the depended-on resource on any node. The dependent resource waits for the depended-on resource to start on any primary node before it starts itself. The situation is similar for stopping and restarting.

`{FROM_RG_AFFINITIES}` — Specifies that the scope of the resource dependency is derived from the `RG_affinities` relationship of the resource groups to which the resources belong. If the dependent resource's group has a positive affinity for the depended-on resource's resource group, and they are starting or stopping on the same node, the dependency is `{LOCAL_NODE}`. If no such positive affinity exists, or if the groups are starting on different nodes, the dependency is `{ANY_NODE}`.

Resource dependencies between two resources that are located in the same resource group are always `{LOCAL_NODE}`.

If you do not specify a qualifier, `FROM_RG_AFFINITIES` is used by default.

Category: Optional

Default: The empty list

Tunable: ANYTIME

Resource_dependencies_weak (string_array)

A list of resources on which the resource has a weak dependency. A weak dependency determines the order of method calls.

The RGM calls the Start methods of the resources in this list, referred to as the *depended-on resources*, before the Start method of the resource with resource dependencies, referred to as the *dependent resource*. The RGM calls the Stop methods of the dependent resource before the Stop methods of the depended-on resources. The dependent resource can still start if the depended-on resources fail to start or remain offline.

If the dependent resource and a depended-on resource in its Resource_dependencies_weak list start concurrently, the RGM waits to start the dependent resource until the depended-on resource in the list starts. If the depended-on resource in the list does not start, for example, if the resource group for the depended-on resource in the list remains offline or the depended-on resource in the list is in a Start_failed state, the dependent resource starts. The dependent resource's resource group might enter a Pending_online_blocked state temporarily as resources in the dependent resource's Resource_dependencies_weak list start. When all depended-on resources in the list have started or failed to start, the dependent resource starts and its group reenters the Pending_online state.

Within a resource group, Prenet_start methods are run in dependency order before Start methods. Postnet_stop methods are run in dependency order after Stop methods. In different resource groups, the dependent resource waits for the depended-on resource to finish Prenet_start and Start before it runs Prenet_start. The depended-on resource waits for the dependent resource to finish Stop and Postnet_stop before it runs Stop.

To specify the scope of a dependency, append the following qualifiers, including the braces ({}), to the resource name when you specify this property.

{LOCAL_NODE} — Limits the specified dependency to a per-host basis. The behavior of the dependent is affected by the depended-on resource only on the same host. The dependent resource waits for the depended-on resource to start on the same host. The situation is similar for stopping and restarting.

{ANY_NODE} — Extends the specified dependency to any node. The behavior of the dependent is affected by the depended-on resource on any node. The dependent resource waits for the depended-on resource to start on any primary node before it starts itself. The situation is similar for stopping and restarting.

{FROM_RG_AFFINITIES} — Specifies that the scope of the resource dependency is derived from the RG_affinities relationship of the resource groups to which the resources belong. If the dependent resource's group

has a positive affinity for the depended-on resource's resource group, and they are starting or stopping on the same node, the dependency is `{LOCAL_NODE}`. If no such positive affinity exists, or if the groups are starting on different nodes, the dependency is `{ANY_NODE}`.

Resource dependencies between two resources that are located in the same resource group are always `LOCAL_NODE`.

If you do not specify a qualifier, `FROM_RG_AFFINITIES` is used by default.

Category: Optional

Default: The empty list

Tunable: `ANYTIME`

`Resource_name` (`string`)

The name of the resource instance. This name must be unique within the cluster configuration and cannot be changed after a resource has been created.

Category: Required

Default: No default

Tunable: `NONE`

`Resource_project_name` (`string`)

The Solaris project name that is associated with the resource. Use this property to apply Solaris resource management features, such as CPU shares and resource pools, to cluster data services. When the RGM brings resources online, it starts the related processes under this project name. If this property is not specified, the project name is taken from the `RG_project_name` property of the resource group that contains the resource (see the `rg_properties(5)` man page). If neither property is specified, the RGM uses the predefined project name `default`. The specified project name must exist in the projects database(see the `projects(1)` man page and *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*).

This property is supported starting with the Solaris 9 OS.

Note – Changes to this property take effect the next time that the resource is started.

Category: Optional

Default: Null

Tunable: `ANYTIME`

`Resource_state` on each cluster node (`enum`)
The RGM-determined state of the resource on each cluster node. Possible states are `Online`, `Offline`, `Start_failed`, `Stop_failed`, `Monitor_failed`, `Online_not_monitored`, `Starting`, and `Stopping`.

You cannot configure this property.

Category: Query-only

Default: No default

Tunable: `NONE`

`Retry_count` (`integer`)
The number of times that a monitor attempts to restart a resource if it fails.

If the `Retry_count` is exceeded, depending on the particular data service and the setting of the `Failover_mode` property, the monitor might perform one of the following actions:

- Allow the resource group to remain on the current primary primary node, even though the resource is in a faulted state
- Request a failover of the resource group onto a different node

This property is created by the RGM and is made available to the cluster administrator only if this property is declared in the RTR file. This property is optional if a default value is specified in the RTR file.

If the `Tunable` attribute is not specified in the RTR file, the `Tunable` value for the property is `WHEN_DISABLED`.

Note – If you specify a negative value for this property, the monitor attempts to restart the resource an unlimited number of times.

However, some resource types do not allow you to set `Retry_count` to a negative value. A more dependable way to specify unlimited restarts is to do the following:

- Set `Retry_interval` to a small value such as 1 or 0.
- Set `Retry_count` to a large value such as 1000.

Category: Conditional

Default: See above

Tunable: `WHEN_DISABLED`

`Retry_interval` (`integer`)
The number of seconds over which to count attempts to restart a failed resource. The resource monitor uses this property in conjunction with `Retry_count`. This property is

created by the RGM and is available to the cluster administrator only if it is declared in the RTR file. This property is optional if a default value is specified in the RTR file.

If the `Tunable` attribute is not specified in the RTR file, the `Tunable` value for the property is `WHEN_DISABLED`.

Category: Conditional

Default: No default (see above)

Tunable: `WHEN_DISABLED`

`Scalable` (`boolean`)

Indicates whether the resource is scalable, that is, whether the resource uses the networking load-balancing features of the Sun Cluster software.

Note – You can configure a scalable resource group (which uses network load-balancing) to run in a global-cluster non-voting node. However, you can run such a scalable resource group in only one node per Solaris host.

If this property is declared in the RTR file, the RGM automatically creates the following scalable service properties for resources of that type: `Affinity_timeout`, `Load_balancing_policy`, `Load_balancing_weights`, `Network_resources_used`, `Port_list`, `UDP_affinity`, and `Weak_affinity`. These properties have their default values unless they are explicitly declared in the RTR file. The default for `Scalable`, when it is declared in the RTR file, is `TRUE`.

If this property is declared in the RTR file, it cannot be assigned a `Tunable` attribute other than `AT_CREATION`.

If this property is not declared in the RTR file, the resource is not scalable, you cannot tune this property, and no scalable service properties are set by the RGM. However, you can explicitly declare the `Network_resources_used` and `Port_list` properties in the RTR file. These properties can be useful in a nonscalable service as well as in a scalable service.

Using this resource property in combination with the `Failover` resource type property is described in more detail in the `r_properties(5)` man page.

Category: Optional

Default: No default

Tunable: `AT_CREATION`

`Start_timeout` for each callback method in the Type (`integer`)

A time lapse, in seconds, after which the RGM concludes that an invocation of this method has failed. For a given resource type, timeout properties are defined only for those methods that are declared in the RTR file.

Category: Conditional or Optional

Default: 3600 (one hour), if the method itself is declared in the RTR file

Tunable: ANYTIME

Status on each cluster node (enum)
Set by the resource monitor with the scha_resource_setstatus command or the scha_resource_setstatus() or scha_resource_setstatus_zone() functions. Possible values are OK, DEGRADED, FAULTED, UNKNOWN, and OFFLINE. When a resource is brought online or offline, the RGM automatically sets the Status value if the Status value is not set by the resource's monitor or methods.

Category: Query-only

Default: No default

Tunable: NONE

Status_msg on each cluster node (string)
Set by the resource monitor at the same time as the Status property. When a resource is brought online or offline, the RGM automatically resets this property to the empty string if this property is not set by the resource's methods.

Category: Query-only

Default: No default

Tunable: NONE

Stop_timeout for each callback method in the Type (integer)
A time lapse, in seconds, after which the RGM concludes that an invocation of this method has failed. For a given resource type, timeout properties are defined only for those methods that are declared in the RTR file.

Category: Conditional or Optional

Default: 3600 (one hour), if the method itself is declared in the RTR file

Tunable: ANYTIME

Thorough_probe_interval (integer)
The number of seconds between invocations of a high-overhead fault probe of the resource. This property is created by the RGM and is available to the cluster administrator only if it is declared in the RTR file. This property is optional if a default value is specified in the RTR file.

If the Tunable attribute is not specified in the RTR file, the Tunable value for the property is WHEN_DISABLED.

Category: Conditional

Default: No default

Tunable: WHEN_DISABLED

Type (string)
The resource type of which this resource is an instance.

Category: Required

Default: No default

Tunable: NONE

Type_version (string)
Specifies which version of the resource type is currently associated with this resource. The RGM automatically creates this property, which cannot be declared in the RTR file. The value of this property is equal to the RT_version property of the resource's type. When a resource is created, the Type_version property is not specified explicitly, though it might appear as a suffix of the resource type name. When a resource is edited, the Type_version property can be changed to a new value.

The tunability of this property is derived from the following sources:

- The current version of the resource type
- The #$upgrade_from directive in the RTR file

Category: See description

Default: No default

Tunable: See description

UDP_affinity (boolean)
If this property is set to TRUE, sends all UDP traffic from a given client to the same server node that currently handles all TCP traffic for the client.

This property is relevant only when Load_balancing_policy is either Lb_sticky or Lb_sticky_wild. In addition, Weak_affinity must be set to FALSE.

This property is only used for scalable services.

Category: Optional

Default: No default

Tunable: WHEN_DISABLED

Update_timeout for each callback method in the Type (integer)
A time lapse, in seconds, after which the RGM concludes that an invocation of this method has failed. For a given resource type, timeout properties are defined only for those methods that are declared in the RTR file.

Category: Conditional or Optional

Default: 3600 (one hour), if the method itself is declared in the RTR file

Tunable: ANYTIME

`Validate_timeout` for each callback method in the Type (`integer`)

A time lapse, in seconds, after which the RGM concludes that an invocation of this method has failed. For a given resource type, timeout properties are defined only for those methods that are declared in the RTR file.

Category: Conditional or Optional

Default: `3600` (one hour), if the method itself is declared in the RTR file

Tunable: ANYTIME

`Weak_affinity` (`boolean`)

If this property is set to `TRUE`, this property enables the weak form of the client affinity.

The weak form of the client affinity allows connections from a given client to be sent to the same server node except when the following conditions occur:

- A server listener starts in response to, for example, a fault monitor's restarting, a resource's failing over or switching over, or a node's rejoining a cluster after failing
- `Load_balancing_weights` for the scalable resource changes because the cluster administrator performed an administrative action

Weak affinity provides a low-overhead alternative to the default form, both in terms of memory consumption and processor cycles.

This property is relevant only when `Load_balancing_policy` is either `Lb_sticky` or `Lb_sticky_wild`.

This property is only used for scalable services.

Category: Optional

Default: No default

Tunable: WHEN_DISABLED

Resource Group Properties

The following information describes the resource group properties that are defined by the Sun Cluster software.

The property values are categorized as follows:

- **Required.** The cluster administrator must specify a value when creating a resource group with an administrative utility.
- **Optional.** If the cluster administrator does not specify a value when creating a resource group, the system supplies a default value.

- **Query-only.** Cannot be set directly by an administrative tool.

Property names are shown first, followed by a description.

`Auto_start_on_new_cluster` (`boolean`)

This property controls whether the Resource Group Manager (RGM) starts the resource group automatically when a new cluster is forming. The default is `TRUE`.

If set to `TRUE`, the RGM attempts to start the resource group automatically to achieve `Desired_primaries` when all the nodes of the cluster are simultaneously rebooted.

If set to `FALSE`, the resource group does not start automatically when the cluster is rebooted. The resource group remains offline until the first time that the resource group is manually switched online by using the `clresourcegroup online` command or the equivalent GUI instruction. After that, the resource group resumes normal failover behavior.

Category: Optional

Default: `TRUE`

Tunable: `ANYTIME`

`Desired_primaries` (`integer`)

The preferred number of nodes that the group can run on simultaneously.

The default is 1. The value of the `Desired_primaries` property must be less than or equal to the value of the `Maximum_primaries` property.

Category: Optional

Default: `1`

Tunable: `ANYTIME`

`Failback` (`boolean`)

A Boolean value that indicates whether to recalculate the set of nodes on which the group is online when a node joins the cluster. A recalculation can cause the RGM to bring the group offline on less preferred nodes and online on more preferred nodes.

Category: Optional

Default: `FALSE`

Tunable: `ANYTIME`

`Global_resources_used` (`string_array`)

Indicates whether cluster file systems are used by any resource in this resource group. Legal values that the cluster administrator can specify are an asterisk (*) to indicate all global resources, and the empty string (“”) to indicate no global resources.

Category: Optional

Default: All global resources

Tunable: ANYTIME

Implicit_network_dependencies (boolean)

A Boolean value that indicates, when TRUE, that the RGM should enforce implicit strong dependencies of nonnetwork address resources on network address resources within the group. This means that the RGM starts all network address resources before all other resources and stops network address resources after all other resources within the group. Network address resources include the logical host name and shared address resource types.

In a scalable resource group, this property has no effect because a scalable resource group does not contain any network address resources.

Category: Optional

Default: TRUE

Tunable: ANYTIME

Maximum_primaries (integer)

The maximum number of nodes on which the group might be online at the same time.

If the RG_mode property is Failover, the value of this property must be no greater than 1. If the RG_mode property is Scalable, a value greater than 1 is allowed.

Category: Optional

Default: 1

Tunable: ANYTIME

Nodelist (string_array)

A list of cluster nodes on which a resource group can be brought online in order of preference. These nodes are known as the potential primaries or masters of the resource group.

Category: Optional

Default: The list of all voting nodes in the cluster in arbitrary order

Tunable: ANYTIME

Pathprefix (string)

A directory in the cluster file system in which resources in the group can write essential administrative files. Some resources might require this property. Make Pathprefix unique for each resource group.

Category: Optional

Default: The empty string

Tunable: ANYTIME

Pingpong_interval (integer)

A nonnegative integer value (in seconds) that is used by the RGM to determine where to bring the resource group online in these instances:

- In the event of a reconfiguration.
- As the result of the execution of a scha_control command with the GIVEOVER argument or the scha_control() function with the SCHA_GIVEOVER argument.

In the event of a reconfiguration, the resource group might fail more than once to come online within the past Pingpong_interval seconds on a particular node. This failure occurs because the resource's Start or Prenet_start method exited with a nonzero status or timed out. As a result, that node is considered ineligible to host the resource group, and the RGM looks for another master.

If a scha_control command or scha_control -O GIVEOVER command is executed on a given node by a resource, thereby causing its resource group to fail over to another node, the first node (on which scha_control was run) cannot be the destination of another scha_control -O GIVEOVER by the same resource until Pingpong_interval seconds have elapsed.

Category: Optional

Default: 3600 (one hour)

Tunable: ANYTIME

Resource_list (string_array)

The list of resources that are contained in the group. The cluster administrator does not set this property directly. Rather, the RGM updates this property as the cluster administrator adds or removes resources from the resource group.

Category: Query-only

Default: No default

Tunable: NONE

RG_affinities (string)

The RGM is to try to locate a resource group on a host that is a current master of another given resource group (positive affinity) or that is not a current master of a given resource group (negative affinity).

You can set RG_affinities to the following strings:

- ++, or strong positive affinity
- +, or weak positive affinity
- -, or weak negative affinity
- --, or strong negative affinity
- +++, or strong positive affinity with failover delegation

For example, RG_affinities=+RG2,--RG3 indicates that this resource group has a weak positive affinity for RG2 and a strong negative affinity for RG3.

Using the RG_affinities property is described in Chapter 2, "Administering Data Service Resources," in *Sun Cluster Data Services Planning and Administration Guide for Solaris OS*.

Category: Optional

Default: The empty string

Tunable: ANYTIME

RG_dependencies (string_array)

Optional list of resource groups that indicates a preferred ordering for bringing other groups online or offline on the same node. The graph of all strong RG_affinities (positive and negative) together with RG_dependencies is not allowed to contain cycles.

For example, suppose that resource group RG2 is listed in the RG_dependencies list of resource group RG1, that is, RG1 has a resource group dependency on RG2.

The following list summarizes the effects of this resource group dependency:

- When a node joins the cluster, Boot methods on that node are not run on resources in RG1 until all Boot methods on that node have completed on resources in RG2.
- If RG1 and RG2 are both in the PENDING_ONLINE state on the same node at the same time, the starting methods (Prenet_start or Start) are not run on any resources in RG1 until all the resources in RG2 have completed their starting methods.
- If RG1 and RG2 are both in the PENDING_OFFLINE state on the same node at the same time, the stopping methods (Stop or Postnet_stop) are not run on any resources in RG2 until all the resources in RG1 have completed their stopping methods.
- An attempt to switch the primaries of RG1 or RG2 fails if switching the primaries would leave RG1 online on any node and RG2 offline on all nodes. The clresourcegroup(1CL) and clsetup(1CL) man pages contain more information.
- Setting the Desired_primaries property to a value that is greater than zero on RG1 is not permitted if Desired_primaries is set to zero on RG2.
- Setting the Auto_start_on_new_cluster property to TRUE on RG1 is not permitted if Auto_start_on_new_cluster is set to FALSE on RG2.

Category: Optional

Default: The empty list

Tunable: ANYTIME

RG_description (string)

A brief description of the resource group.

Category: Optional

Default: The empty string

Tunable: ANYTIME

RG_is_frozen (boolean)
Indicates whether a global device on which a resource group depends is being switched over. If this property is set to TRUE, the global device is being switched over. If this property is set to FALSE, no global device is being switched over. A resource group depends on global devices as indicated by its Global_resources_used property.

You do not set the RG_is_frozen property directly. The RGM updates the RG_is_frozen property when the status of the global devices changes.

Category: Optional

Default: No default

Tunable: NONE

RG_mode (enum)
Indicates whether the resource group is a failover or a scalable group. If the value is Failover, the RGM sets the Maximum_primaries property of the group to 1 and restricts the resource group to being mastered by a single node.

If the value of this property is Scalable, the RGM allows the Maximum_primaries property to be set to a value that is greater than 1. As a result, the group can be mastered by multiple nodes simultaneously. The RGM does not allow a resource whose Failover property is TRUE to be added to a resource group whose RG_mode is Scalable.

If Maximum_primaries is 1, the default is Failover. If Maximum_primaries is greater than 1, the default is Scalable.

Category: Optional

Default: Depends on the value of Maximum_primaries

Tunable: NONE

RG_name (string)
The name of the resource group. This property is required and must be unique within the cluster.

Category: Required

Default: No default

Tunable: NONE

RG_project_name (string)
The Solaris project name (see the projects(1) man page) that is associated with the resource group. Use this property to apply Solaris resource management features, such as CPU shares and resource pools, to cluster data services. When the RGM brings resource groups online, it

starts the related processes under this project name for resources that do not have the Resource_project_name property set (see the r_properties(5) man page). The specified project name must exist in the projects database (see the projects(1) man page and *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*).

This property is supported starting with the Solaris 9 OS.

Note – Changes to this property take affect the next time that the resource is started.

Category: Optional

Default: The text string "default"

Tunable: ANYTIME

RG_slm_cpu (decimal number)
If the RG_slm_type property is set to AUTOMATED, this number is the basis for the calculation of the number of CPU shares and the size of the processor set.

Note – You can only use the RG_slm_cpu property if RG_slm_type is set to AUTOMATED. For more information, see the RG_slm_type property.

The maximum value for the RG_slm_cpu property is 655. You can include two digits after the decimal point. Do not specify 0 for the RG_slm_cpu property. If you set a share value to 0, a resource might not be scheduled by the Fair Share Scheduler (FFS) when the CPU is heavily loaded.

Changes that you make to the RG_slm_cpu property while the resource group is online are taken into account dynamically.

Because the RG_slm_type property is set to AUTOMATED, Sun Cluster creates a project named SCSLM_*resourcegroupname*. *resourcegroupname* represents the actual name that you assign to the resource group. Each method of a resource that belongs to the resource group is executed in this project. Starting with Solaris 10 OS, these projects are created in the resource group's node, whether it is a global-cluster voting node or a global-cluster non-voting node. See the project(4) man page.

The project SCSLM_*resourcegroupname* has a project.cpu-shares value of 100 times the RG_slm_cpu property value. If the RG_slm_cpu property is not set, this project is created with a project.cpu-shares value of 1. The default value for the RG_slm_cpu property is 0.01.

Starting with the Solaris 10 OS, if the RG_slm_pset_type property is set to DEDICATED_STRONG or to DEDICATED_WEAK, the RG_slm_cpu property is used to calculate the size of processor sets. The RG_slm_cpu property is also used to calculate the value of zone.cpu-shares.

For information about processor sets, see the *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*.

Note – You can use the RG_slm_cpu property only on a global cluster. You cannot use this property on a zone cluster.

Category: Optional

Default: 0.01

Tunable: ANYTIME

RG_slm_cpu_min (decimal number)
Determines the minimum number of processors on which an application can run.

You can only use this property if all of the following factors are true:

- The RG_slm_type property is set to AUTOMATED
- The RG_slm_pset_type property is set to DEDICATED_STRONG or to DEDICATED_WEAK
- The RG_slm_cpu property is set to a value that is greater than or equal to the value set for the RG_slm_cpu_min property
- You are using the Solaris 10 OS

The maximum value for the RG_slm_cpu_min property is 655. You can include two digits after the decimal point. Do not specify 0 for the RG_slm_cpu_min property. The RG_slm_cpu_min and RG_slm_cpu properties determine the values of pset.min and pset.max, respectively, for the processor set that Sun Cluster generates.

Changes that you make to the RG_slm_cpu and the RG_slm_cpu_min properties while the resource group is online are taken into account dynamically. If the RG_slm_pset_type property is set to DEDICATED_STRONG, and not enough CPUs are available, the change that you request for the RG_slm_cpu_min property is ignored. In this case, a warning message is generated. On next switchover, if not enough CPUs are available for the RG_slm_cpu_min property, errors due to the lack of CPUs can occur.

For information about processor sets, see the *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*.

Note – You can use the RG_slm_cpu_min property only on a global cluster. You cannot use this property on a zone cluster.

Category: Optional

Default: 0.01

Tunable: ANYTIME

RG_slm_type (string)

Enables you to control system resource usage and automate some steps to configure the Solaris operating system for system resource management. Possible values for RG_SLM_type are AUTOMATED and MANUAL.

If you set the RG_slm_type property to AUTOMATED, the resource group is started with control of the CPU usage.

As a result, Sun Cluster does the following:

- Creates a project named SCSLM_*resourcegroupname*. All methods in the resources in this resource group execute in this project. This project is created the first time a method of a resource in this resource group is executed on the node.
- Sets the value of project.cpu_shares that is associated with the project to the value of the RG_slm_cpu property times 100. By default, the value for project.cpu_shares is 1.
- Starting with the Solaris 10 OS, sets zone.cpu_shares to 100 times the sum of the RG_slm_cpu property in all the online resource groups. This property also sets RG_slm_type to AUTOMATED in this node. The node can be a global-cluster voting node or a global-cluster non-voting node. The global-cluster non-voting node is bound to a Sun Cluster generated pool. Optionally, if the RG_slm_pset_type property is set to DEDICATED_WEAK or to DEDICATED_STRONG, this Sun Cluster generated pool is associated with a Sun Cluster generated processor set. For information about dedicated processor sets, see the description of the RG_slm_pset_type property. When you set the RG_slm_type property to AUTOMATED, all operations that are performed are logged.

If you set the RG_slm_type property to MANUAL, the resource group executes in the project that is specified by the RG_project_name property.

For information about resource pools and processor sets, see the *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*.

Note –

- Do not specify resource group names that exceed 58 characters. If a resource group name contains more than 58 characters, you cannot configure CPU control, that is, you cannot set the RG_slm_type property to AUTOMATED.
- Refrain from including dashes (-) in resource group names. The Sun Cluster software replaces all dashes in resource group names with underscores (_) when it creates a project. For example, Sun Cluster creates the project named SCSLM_rg_dev for a resource group named rg-dev. If a resource group named rg_dev already exists, a conflict arises when Sun Cluster attempts to create the project for the resource group rg-dev.

Note – You can use the RG_slm_type property only on a global cluster. You cannot use this property on a zone cluster.

Category: Optional

Default: manual

Tunable: ANYTIME

RG_slm_pset_type (string)
Enables the creation of a dedicated processor set.

You can only use this property if all of the following factors are true:

- The RG_slm_type property is set to AUTOMATED
- You are using the Solaris 10 OS
- The resource group executes in a global-cluster non-voting node

Possible values for RG_slm_pset_type are DEFAULT, DEDICATED_STRONG, and DEDICATED_WEAK.

For a resource group to execute as DEDICATED_STRONG or DEDICATED_WEAK, the resource group must be configured so there are only global-cluster non-voting nodes in its node list.

The global-cluster non-voting node must not be configured for a pool other than the default pool (POOL_DEFAULT). For information about zone configuration, see the zonecfg(1M) man page. This global-cluster non-voting node must not be dynamically bound to a pool other than the default pool. For more information on pool binding, see the poolbind(1M) man page. These two pool conditions are verified only when the methods of the resources in the resource group are launched.

The values DEDICATED_STRONG and DEDICATED_WEAK are mutually exclusive for resource groups that have the same node in their node list. You cannot configure resource groups in the same node so that some have RG_slm_pset_type set to DEDICATED_STRONG and others set to DEDICATED_WEAK.

If you set the RG_slm_pset_type property to DEDICATED_STRONG, Sun Cluster does the following in addition to the actions performed by the RG_slm_type property when it is set to AUTOMATED:

- Creates and dynamically binds a pool to the global-cluster non-voting node in which the resource group starts for either or both the PRENET_START and START methods.
- Creates a processor set with a size between the following sums
  - The sum of the RG_slm_cpu_min property in all the resource groups that are online on the node in which this resource group starts.
  - The sum of the RG_slm_cpu property in the resource groups that are running on that node.

When either the STOP or POSTNET_STOP methods execute, the Sun Cluster generated processor set is destroyed. If resource groups are no longer online on the node, the pool is destroyed, and the global-cluster non-voting node is bound to the default pool (POOL_DEFAULT).

- Associates the processor set to the pool.
- Sets zone.cpu_shares to 100 times the sum of the RG_slm_cpu property in all the resource groups that are running the node.

If you set the RG_slm_pset_type property to DEDICATED_WEAK, the resource group behaves the same as if RG_slm_pset_type was set to DEDICATED_STRONG. However, if enough processors are not available to create the processor set, the pool is associated to the default processor set.

If you set the RG_slm_pset_typeproperty to DEDICATED_STRONG and not enough processors are available to create the processor set, an error is generated. As a result, the resource group is not started on that node.

When CPUs are allocated, the DEFAULTPSETMIN minimum size has priority over DEDICATED_STRONG, which has priority over DEDICATED_WEAK. However, when you use the clnode command to increase the size of the default processor, and not enough processors are available, this priority is ignored. For information about the DEFAULTPSETMIN property, see the clnode(1CL) man page.

The clnode command assigns a minimum of CPUs to the default processor set dynamically. If the number of CPUs that you specify is not available, Sun Cluster periodically retries to assign this number of CPUs. Failing that, Sun Cluster tries to assign smaller numbers of CPUs to the default processor set until the minimum number of CPUs are assigned. This action might destroy some DEDICATED_WEAK processor sets, but does not destroy DEDICATED_STRONG processor sets.

When you start a resource group for which you've set the RG_slm_pset_type property to DEDICATED_STRONG, it might destroy the processor sets that are associated with the DEDICATED_WEAK processor sets. This resource group might do so if not enough CPUs are available on the node for both processor sets. In this case, the processes of the resource group that are running in the DEDICATED_WEAK processor sets are associated with the default processor set.

To swap the value of the RG_slm_pset_type property between DEDICATED_STRONG or DEDICATED_WEAK, you must first set it to the default.

If resource groups that are configured for CPU control are not online in a global-cluster non-voting node, the CPU share value is set to zone.cpu-shares for that node. By default, zone.cpu-shares is set to 1. For more information about zone configuration, see the zonecfg(1M) man page.

If you set the RG_slm_pset_type property to DEFAULT, Sun Cluster creates a pool named SCSLM_pool_*zonename*, but does not create a processor set. In this case, SCSLM_pool_*zonename* is associated with the default processor set. The shares that are assigned to the node equal the sum of the values for RG_slm_cpu for all the resource groups in the node.

For information about resource pools and processor sets, see the *System Administration Guide: Solaris Containers-Resource Management and Solaris Zones*.

Note – You can use the RG_slm_pset_type property only on a global cluster. You cannot use this property on a zone cluster.

Category: Optional

Default: default

Tunable: ANYTIME

RG_state on each cluster node (enum)
Set by the RGM to UNMANAGED, ONLINE, OFFLINE, PENDING_ONLINE, PENDING_OFFLINE, ERROR_STOP_FAILED, ONLINE_FAULTED, or PENDING_ONLINE_BLOCKED to describe the state of the group on each cluster node.

You cannot configure this property. However, you can indirectly set this property by running the clresourcegroup command or by using the equivalent clsetup or Sun Cluster Manager commands. A group can exist in an UNMANAGED state when that group is not under the control of the RGM.

The following descriptions summarize each state.

Note – States apply to individual nodes only, except the UNMANAGED state, which applies across all nodes. For example, a resource group might be OFFLINE on node A, but PENDING_ONLINE on node B.

UNMANAGED — The initial state of a newly created resource group, or the state of a previously managed resource group. Either Init methods have not yet been run on resources in the group, or Fini methods have been run on resources in the group.

The group is not managed by the RGM.

ONLINE — The resource group has been started on the node. In other words, the starting methods

	Prenet_start, Start, and Monitor_start, as applicable to each resource, have executed successfully on all enabled resources in the group.
OFFLINE	The resource group has been stopped on the node. In other words, the stopping methods Monitor_stop, Stop, and Postnet_stop, as applicable to each resource, have executed successfully on all enabled resources in the group. This state also applies before a resource group has started for the first time on the node.
PENDING_ONLINE	The resource group is starting on the node. The starting methods Prenet_start, Start, and Monitor_start, as applicable to each resource, are being executed on enabled resources in the group.
PENDING_OFFLINE	The resource group is stopping on the node. The stopping methods Monitor_stop, Stop, and Postnet_stop, as applicable to each resource, are being executed on enabled resources in the group.
ERROR_STOP_FAILED	One or more resources within the resource group failed to stop successfully and are in the Stop_failed state. Other resources in the group might remain online or offline. This resource group is not permitted to start on any node until the ERROR_STOP_FAILED state is cleared. You must use an administrative command, such as clresource clear, to manually kill the Stop_failed resource and reset its state to OFFLINE.
ONLINE_FAULTED	The resource group was PENDING_ONLINE and has finished starting on this node. However, one or more resources ended up in the START_FAILED state or with FAULTED status.
PENDING_ONLINE_BLOCKED	The resource group failed to start fully because one or more resources within that resource group have an unsatisfied strong resource dependency on a resource in a different

resource group. Such resources remain OFFLINE. When the resource dependencies are satisfied, the resource group automatically moves back to the PENDING_ONLINE state.

Category: Query-only

Default: No default

Tunable: NONE

Suspend_automatic_recovery (boolean)

A Boolean value that indicates whether the automatic recovery of a resource group is suspended. A suspended resource group is *not* automatically restarted or failed over until the cluster administrator explicitly issues the command that resumes automatic recovery. Whether online or offline, suspended data services remain in their current state. You can still manually switch the resource group to a different state on specified nodes. You can also still enable and disable individual resources in the resource group.

If the Suspend_automatic_recovery property is set to TRUE, automatic recovery of the resource group is suspended. If this property is set to FALSE, automatic recovery of the resource group is resumed and active.

You do not set this property directly. The RGM changes the value of the Suspend_automatic_recovery property when the cluster administrator suspends or resumes automatic recovery of the resource group. The cluster administrator suspends automatic recovery with the clresourcegroup suspend command. The cluster administrator resumes automatic recovery with the clresourcegroup resume command. The resource group can be suspended or resumed regardless of the setting of its RG_system property.

Category: Query-only

Default: FALSE

Tunable: NONE

RG_system (boolean)

If the RG_system property is TRUE for a resource group, particular operations are restricted for the resource group and for the resources that the resource group contains. This restriction is intended to help prevent accidental modification or deletion of critical resource groups and resources. Only the clresourcegroupcommand is affected by this property. Operations for scha_control(1HA) and scha_control(3HA) are not affected.

Before performing a restricted operation on a resource group (or a resource group's resources), you must first set the RG_system property of the resource group to FALSE. Use care when you modify or delete a resource group that supports cluster services, or when you modify or delete the resources that such a resource group contains.

Operation	Example
Delete a resource group	**clresourcegroup delete RG1**
Edit a resource group property (except for RG_system)	**clresourcegroup set -p RG_desription=... +**
Add a resource to a resource group	**clresource create -g RG1 -t SUNW.nfs R1** The resource is created in the enabled state and with resource monitoring turned on.
Delete a resource from a resource group	**clresource delete R1**
Edit a property of a resource that belongs to a resource group	**clresource set -g RG1 -t SUNW.nfs -p r_description="HA-NFS res" R1**
Switch a resource group offline	**clresourcegroup offline RG1**
Manage a resource group	**clresourcegroup manage RG1**
Unmanage a resource group	**clresourcegroup unmanage RG1**
Enable a resource in a resource group	**clresource enable R1**
Enable monitoring for a resource in a resource group	**clresource monitor R1**
Disable a resource in a resource group	**clresource disable R1**
Disable monitoring for a resource	**clresource unmonitor R1**

If the RG_system property is TRUE for a resource group, the only property of the resource group that you can edit is the RG_system property itself. In other words, editing the RG_system property is never restricted.

Category: Optional

Default: FALSE

Tunable: ANYTIME

Resource Property Attributes

This section describes the resource property attributes that you can use to change system-defined properties or to create extension properties.

Caution – You cannot specify Null or the empty string (“”) as the default value for boolean, enum, or int types.

Property names are shown first, followed by a description.

`Array_maxsize`
For a `stringarray` type, the maximum number of array elements that are permitted.

`Array_minsize`
For a `stringarray` type, the minimum number of array elements that are permitted.

`Default`
Indicates a default value for the property.

`Description`
A string annotation that is intended to be a brief description of the property. The `Description` attribute cannot be set in the RTR file for system-defined properties.

`Enumlist`
For an `enum` type, a set of string values that are permitted for the property.

`Extension`
If used, indicates that the RTR file entry declares an extension property that is defined by the resource type implementation. Otherwise, the entry is a system-defined property.

`Max`
For an `int` type, the maximum value that is permitted for the property.

`Maxlength`
For `string` and `stringarray` types, the maximum string length that is permitted.

`Min`
For an `int` type, the minimal value that is permitted for the property.

`Minlength`
For `string` and `stringarray` types, the minimum string length that is permitted.

`Per_node`
If used, indicates that the extension property can be set on a per-node basis.

If you specify the `Per_node` property attribute in a type definition, you must specify a default value with the `Default` property attribute as well. Specifying a default value ensures that a value is returned when a user requests a per-node property value on a node to which an explicit value has not been assigned.

You cannot specify the `Per_node` property attribute for a property of type `stringarray`.

`Property`
The name of the resource property.

`Tunable`
Indicates when the cluster administrator can set the value of this property in a resource. Set to `NONE` or `FALSE` to prevent the cluster administrator from setting the property. Values that enable a cluster administrator to tune a property are `TRUE` or `ANYTIME` (at any time),

AT_CREATION (only when the resource is created), or WHEN_DISABLED (when the resource is disabled). To establish other conditions, such as “when monitoring is disabled” or “when offline”, set this attribute to ANYTIME and validate the state of the resource in the Validate method.

The default differs for each standard resource property, as shown in the following entry. The default setting for tuning an extension property, if not otherwise specified in the RTR file, is TRUE (ANYTIME).

Type of the property
Allowable types are string, boolean, integer, enum, and stringarray. You cannot set the type attribute in an RTR file entry for system-defined properties. The type determines acceptable property values and the type-specific attributes that are allowed in the RTR file entry. An enum type is a set of string values.

APPENDIX B

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 291
- "Start Method Code Listing" on page 295
- "Stop Method Code Listing" on page 298
- "gettime Utility Code Listing" on page 300
- "PROBE Program Code Listing" on page 301
- "Monitor_start Method Code Listing" on page 307
- "Monitor_stop Method Code Listing" on page 309
- "Monitor_check Method Code Listing" on page 311
- "Validate Method Code Listing" on page 313
- "Update Method Code Listing" on page 317

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 B–1 SUNW.Sample RTR File

```
#
# Copyright (c) 1998-2006 by Sun Microsystems, Inc.
# All rights reserved.
#
# Registration information for Domain Name Service (DNS)
#
```

EXAMPLE B–1 `SUNW.Sample` RTR File *(Continued)*

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

Resource_type = "sample";
Vendor_id = SUNW;
RT_description = "Domain Name Service on Sun Cluster";

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

RT_basedir=/opt/SUNWsample/bin;
Pkglist = SUNWsample;

Start             = dns_svc_start;
Stop              = dns_svc_stop;

Validate          = dns_validate;
Update            = dns_update;

Monitor_start     = dns_monitor_start;
Monitor_stop      = dns_monitor_stop;
Monitor_check     = dns_monitor_check;

# 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;
```

EXAMPLE B–1 `SUNW.Sample` RTR File *(Continued)*

```
        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;
```

EXAMPLE B–1 `SUNW.Sample` RTR File *(Continued)*

```
        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 B–2 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   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
                        ;;
```

EXAMPLE B–2 dns_svc_start Method *(Continued)*

```
                *)
                    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 theresource.
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
```

EXAMPLE B–2 dns_svc_start Method *(Continued)*

```
# 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_NAMÈ

# 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_NAMÈ 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 $RETRY_INTERVAL; if it crashes
# more often than that, PMF will cease trying to restart it. If there is a
# process already registered under the tag <$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 B–3 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   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
                     ;;
```

EXAMPLE B–3 dns_svc_stop Method *(Continued)*

```
                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

# Obtain the Stop_timeout value from the RTR file.
STOP_TIMEOUT=`scha_resource_get -O STOP_TIMEOUT -R $RESOURCE_NAME -G \
$RESOURCEGROUP_NAMÈ

# 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
```

EXAMPLE B–3 dns_svc_stop Method *(Continued)*

```
      # 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 B–4 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   00/05/24 SMI”

#include <stdio.h>
#include <sys/types.h>
#include <time.h>
```

EXAMPLE B–4 gettime.c Utility Program (Continued)

```
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 B–5 dns_probe Program

```
#!/bin/ksh
#pragma ident   “@(#)dns_probe   1.1   00/04/19 SMI”
#
# 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.
```

EXAMPLE B–5 `dns_probe` Program *(Continued)*

```
                        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_NAMÈ
                STOP_METHOD=`scha_resource_get -O STOP \
                        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAMÈ
                hatimerun -t $STOP_TIMEOUT $RT_BASEDIR/$STOP_METHOD \
                        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME \
                        -T $RESOURCETYPE_NAME

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

EXAMPLE B–5 `dns_probe` Program *(Continued)*

```
                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_NAMÈ
                START_METHOD=`scha_resource_get -O START \
                        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAMÈ
                hatimerun -t $START_TIMEOUT $RT_BASEDIR/$START_METHOD \
                        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME \
                        -T $RESOURCETYPE_NAME

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

        else
                # 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/gettimè
```

EXAMPLE B–5 dns_probe Program *(Continued)*

```
        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/gettimè
     time_diff=`expr $current_time - $start_timè
     if [ $time_diff -ge $RETRY_INTERVAL ]; then
        # This failure happened after the time window
        # elapsed, so reset the retries counter,
        # slide the window, and do a retry.
        retries=1
        start_time=$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] \
```

EXAMPLE B–5 dns_probe Program *(Continued)*

```
                “${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_NAMÈ

# 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_NAMÈ

# 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_NAMÈ

# Get the retry interval value from the system defined property
Retry_interval
```

EXAMPLE B–5 dns_probe Program *(Continued)*

```
RETRY_INTERVAL=scha_resource_get -O RETRY_INTERVAL -R $RESOURCE_NAME \
-G $RESOURCEGROUP_NAMÈ

# 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_NAMÈ

# 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.
   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
```

EXAMPLE B–5 dns_probe Program *(Continued)*

```
        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 B–6 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   00/05/24 SMI"

###############################################################################
# Parse program arguments.
#
function parse_args # [args ...]
{
        typeset opt

        while getopts 'R:G:T:' opt
        do
```

EXAMPLE B–6 dns_monitor_start Method *(Continued)*

```
                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

# 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_NAMÈ
```

EXAMPLE B–6 dns_monitor_start Method *(Continued)*

```
# 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 $RESOURCETYPE_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 B–7 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   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
```

EXAMPLE B–7 dns_monitor_stop Method *(Continued)*

```
                ;;
        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
```

EXAMPLE B–7 dns_monitor_stop Method *(Continued)*

```
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 B–8 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   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
      ;;
```

EXAMPLE B–8 dns_monitor_check Method *(Continued)*

```
      *)
      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

# 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_NAMÈ

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

# 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}'`
```

EXAMPLE B–8 dns_monitor_check Method *(Continued)*

```
# 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 B–9 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>
```

EXAMPLE B–9 dns_validate Method *(Continued)*

```
#   OR
# dns_validate -u -R <..> -G <...> -T <..> -x <extn-prop_being_updated=value>

#pragma ident   "@(#)dns_validate   1.1   00/05/24 SMI"

###############################################################################
# Parse program arguments.
#
function parse_args # [args ...]
{
   typeset opt

   while getopts 'cur:x:g:R:T:G:' 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
                        ;;
                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.
```

EXAMPLE B–9 dns_validate Method *(Continued)*

```
                        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
```

EXAMPLE B–9 dns_validate Method *(Continued)*

```
# 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 B–10 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   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
}
###############################################################################
# MAIN
```

EXAMPLE B–10 dns_update Method *(Continued)*

```
###############################################################################
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

# 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_NAMÈ

# 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
```

APPENDIX C

DSDL Sample Resource Type Code Listings

This appendix lists the complete code for each method in the SUNW.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 SUNW.xfnts in more detail.

This appendix covers the following topics:

- "xfnts.c File Listing" on page 319
- "xfnts_monitor_check Method Code Listing" on page 333
- "xfnts_monitor_start Method Code Listing" on page 334
- "xfnts_monitor_stop Method Code Listing" on page 335
- "xfnts_probe Method Code Listing" on page 336
- "xfnts_start Method Code Listing" on page 339
- "xfnts_stop Method Code Listing" on page 340
- "xfnts_update Method Code Listing" on page 341
- "xfnts_validate Method Code Listing" on page 343

xfnts.c File Listing

This file implements the subroutines that are called by the SUNW.xfnts methods.

EXAMPLE C–1 xfnts.c

```
/*
 * Copyright (c) 1998-2006 by Sun Microsystems, Inc.
 * 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
```

EXAMPLE C–1 xfnts.c *(Continued)*

```
 * 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 01/01/18 SMI"

#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <netinet/in.h>
#include <scha.h>
#include <rgm/libdsdev.h>
#include <errno.h>
#include "xfnts.h"

/*
 * 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
```

EXAMPLE C–1 xfnts.c *(Continued)*

```
 * 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 */
   }
```

EXAMPLE C–1 xfnts.c *(Continued)*

```
   /*
    * 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/openwin/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,
```

EXAMPLE C–1 xfnts.c *(Continued)*

```
        “Property Port_list must have only one value.”);
      scds_free_port_list(portlist);
      return (1); /* Validation Failure */
   }
#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:
```

EXAMPLE C–1 xfnts.c *(Continued)*

```
   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/openwin/bin/xfs -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,resourcename,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/openwin/bin/xfs notice: terminating”
```

EXAMPLE C–1 xfnts.c *(Continued)*

```
     * In order to suppress the daemon message,
     * the output is redirected to /dev/null.
     */
    (void) sprintf(cmd,
        "/usr/openwin/bin/xfs -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;

    /*
```

EXAMPLE C–1 xfnts.c *(Continued)*

```
    * 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);
   }
```

EXAMPLE C–1 xfnts.c *(Continued)*

```
/*
 * 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) {
```

EXAMPLE C–1 xfnts.c *(Continued)*

```
         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.");
```

EXAMPLE C–1 xfnts.c *(Continued)*

```
   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.
 */
```

EXAMPLE C–1 xfnts.c *(Continued)*

```
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
```

EXAMPLE C–1 xfnts.c *(Continued)*

```
     * 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);
```

EXAMPLE C–1 xfnts.c *(Continued)*

```
   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 C–2 xfnts_monitor_check.c

```
/*
 * Copyright (c) 1998-2006 by Sun Microsystems, Inc.
 * All rights reserved.
 *
 * xfnts_monitor_check.c - Monitor Check method for HA-XFS
 */

#pragma ident "@(#)xfnts_monitor_check.c 1.11 01/01/18
SMI"

#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 C–3 xfnts_monitor_start.c

```
/*
 * Copyright (c) 1998-2006 by Sun Microsystems, Inc.
 * All rights reserved.
 *
 * xfnts_monitor_start.c - Monitor Start method for HA-XFS
 */

#pragma ident "@(#)xfnts_monitor_start.c 1.10 01/01/18
SMI"

#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 C–4 xfnts_monitor_stop.c

```
/*
 * Copyright (c) 1998-2006 by Sun Microsystems, Inc.
 * All rights reserved.
 *
 * xfnts_monitor_stop.c - Monitor Stop method for HA-XFS
 */

#pragma ident "@(#)xfnts_monitor_stop.c 1.9 01/01/18 SMI"

#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 C–5 xfnts_probe.c

```
/*
 * Copyright (c) 1998-2006 by Sun Microsystems, Inc.
 * 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;
```

EXAMPLE C–5 xfnts_probe.c *(Continued)*

```
   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.
```

EXAMPLE C–5 xfnts_probe.c *(Continued)*

```
         */
        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 xfs daemon on that node.

EXAMPLE C–6 xfnts_start.c

```
/*
 * Copyright (c) 1998-2006 by Sun Microsystems, Inc.
 * 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.”);
```

EXAMPLE C–6 `xfnts_start.c` *(Continued)*

```
      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 C–7 `xfnts_stop.c`

```
/*
 * Copyright (c) 1998-2006 by Sun Microsystems, Inc.
 * All rights reserved.
 *
```

EXAMPLE C–7 xfnts_stop.c *(Continued)*

```
 * 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 C–8 xfnts_update.c

```
#pragma ident "@(#)xfnts_update.c  1.10     01/01/18 SMI"
```

EXAMPLE C–8 xfnts_update.c *(Continued)*

```
/*
 * Copyright (c) 1998-2006 by Sun Microsystems, Inc.
 * 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);
```

EXAMPLE C–8 xfnts_update.c *(Continued)*

```
   }

   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 C–9 xfnts_validate.c

```
/*
 * Copyright (c) 1998-2006 by Sun Microsystems, Inc.
 * 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)
```

EXAMPLE C–9 xfnts_validate.c *(Continued)*

```
{
      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);

}
```

APPENDIX D

Legal RGM Names and Values

This appendix lists the requirements for legal characters for Resource Group Manager (RGM) names and values.

This appendix covers the following topics:

- "RGM Legal Names" on page 345
- "RGM Values" on page 347

RGM Legal Names

RGM names fall into the following categories:

- Resource group names
- Resource type names
- Resource names
- Property names
- Enumeration literal names

Rules for Names Except Resource Type Names

Except for resource type names, all names must comply with these rules:

- Names must be in ASCII.
- Names must start with a letter.
- Names can contain uppercase and lowercase letters, digits, dashes (-), and underscores (_).
- The maximum number of characters that you can use in a name is 255.

Format of Resource Type Names

The format of the complete name of a resource type depends on the resource type, as follows:

- If the resource type's resource type registration (RTR) file contains the `#$upgrade` directive, the format is as follows:

  vendor-id.*base-rt-name*:*rt-version*

- If the resource type's RTR file does *not* contain the `#$upgrade` directive, the format is as follows:

  vendor-id.*base-rt-name*

A period separates *vendor-id* and *base-rt-name*. A colon separates *base-rt-name* and *rt-version*.

The variable elements in this format are as follows:

vendor-id — Specifies the vendor ID prefix, which is the value of the `Vendor_id` resource type property in the RTR file. If you are developing a resource type, choose a vendor ID prefix that uniquely identifies the vendor, such as your company's stock ticker symbol. For example, the vendor ID prefix of resource types that are developed by Sun Microsystems, Inc. is `SUNW`.

base-rt-name — Specifies the base resource type name, which is the value of the `Resource_type` resource type property in the RTR file.

rt-version — Specifies the version suffix, which is the value of the `RT_version` resource type property in the RTR file. The version suffix is *only* part of the complete resource type name if the RTR file contains the `#$upgrade` directive. The `#$upgrade` directive was introduced in Release 3.1 of the Sun Cluster product.

Note – If only one version of a base resource type name is registered, you do not have to use the complete name in administrative commands. You can omit the vendor ID prefix, the version number suffix, or both.

For more information, see "Resource Type Properties" on page 243.

EXAMPLE D–1 Complete Name of a Resource Type With the `#$upgrade` Directive

This example shows the complete name of a resource type for which properties in the RTR file are set, as follows:

- `Vendor_id=SUNW`
- `Resource_type=sample`
- `RT_version=2.0`

EXAMPLE D–1 Complete Name of a Resource Type With the `#$upgrade` Directive *(Continued)*

The complete name of the resource type that is defined by this RTR file is as follows:

```
SUNW.sample:2.0
```

EXAMPLE D–2 Complete Name of a Resource Type Without the `#$upgrade` Directive

This example shows the complete name of a resource type for which properties in the RTR file are set, as follows:

- `Vendor_id=SUNW`
- `Resource_type=nfs`

The complete name of the resource type that is defined by this RTR file is as follows:

```
SUNW.nfs
```

RGM Values

RGM values fall into two categories: property values and description values. Both categories share the same rules:

- Values must be in ASCII.
- The maximum length of a value is 4 megabytes minus 1, that is, 4,194,303 bytes.
- Values cannot contain the following characters:
  - Null
  - Newline
  - Semicolon (;)

APPENDIX E

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 29 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 *Sun Cluster Concepts Guide for Solaris OS*.

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 349
- "Host Names" on page 351
- "Multihomed Hosts" on page 351
- "Binding to `INADDR_ANY` as Opposed to Binding to Specific IP Addresses" on page 352
- "Client Retry" on page 353

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 Sun Cluster include the UNIX File System (UFS), Quick File System (QFS), Veritas File System (VxFS), and Solaris ZFS (Zettabyte File System).

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 Oracle software.

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 350 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 Sun Cluster software supports the use of UNIX UFS file systems and HA raw devices that are configured in a volume manager. When installing and configuring the Sun Cluster software, the cluster administrator must specify which disk resources to use for UFS 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. Sun 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 Sun 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 Sun 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 Solaris Operating System already handle multihomed hosts correctly. Many data services do all their network communication by binding to the 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 Sun Cluster logical network addresses.

Binding to INADDR_ANY as Opposed to Binding to Specific IP Addresses

Even when non-multihomed hosts are used, the Sun 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) 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 a Sun Cluster environment if they bind to INADDR_ANY. These data services must dynamically change the set of IP addresses to which they are bound as the resource group is mastered or unmastered. One strategy for accomplishing the rebinding is to have the starting and stopping methods for these data services kill and restart the data service's daemons.

The Network_resources_used resource property permits the end user to configure a specific set of network address resources to which the application resource should bind. For resource types that require this feature, the Network_resources_used property must be declared in the RTR file for the resource type.

When the RGM brings the resource group online or offline, the RGM follows a specific order for plumbing, unplumbing, and configuring network addresses up or down in relation to when the RGM calls call data service resource methods. See "Deciding Which Start and Stop Methods to Use" on page 47.

By the time the data service's Stop method returns, the data service must have stopped by using the resource group's network addresses. Similarly, by the time the Start method returns, the data service must have started to use the network addresses.

If the data service binds to INADDR_ANY rather than to individual IP addresses, the order in which data service resource methods are called and network address methods are called is not relevant.

If the data service's stop and start methods accomplish their work by killing and restarting the data service's daemons, the data service stops and starts using the network addresses at the correct times.

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.

APPENDIX F

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 355
- "NVPAIR XML DTD" on page 357
- "SC_REPLY XML DTD" on page 358
- "SC_EVENT XML DTD" on page 359

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 2001-2006 Sun Microsystems, Inc. All rights reserved.
        Use is subject to license terms.

        Intended Use:

A client of the Cluster Reconfiguration Notification Protocol should use this xml format
to register initially with the service, to subsequently register for more events, to
subsequently remove registration of some events, or to remove itself from the service
entirely.

A client is uniquely identified by its callback IP and port.  The port is defined in the
SC_CALLBACK_REG element, and the IP is taken as the source IP of the registration
connection.  The final attribute of the root SC_CALLBACK_REG element is either an
ADD_CLIENT, ADD_EVENTS, REMOVE_CLIENT, or REMOVE_EVENTS, depending on which form of the
message the client is using.
```

```
The SC_CALLBACK_REG contains 0 or more SC_EVENT_REG sub-elements.

One SC_EVENT_REG is the specification for one event type.  A client may specify only the
CLASS (an attribute of the SC_EVENT_REG element), or may specify a SUBCLASS (an optional
attribute) for further granularity.  Also, the SC_EVENT_REG has as subelements 0 or more
NVPAIRs, which can be used to further specify the event.

Thus, the client can specify events to whatever granularity it wants.  Note that a client
cannot both register for and unregister for events in the same message.  However a client
can subscribe to the service and sign up for events in the same message.

Note on versioning: the VERSION attribute of each root element is marked "fixed", which
means that all message adhering to these DTDs must have the version value specified.  If a
new version of the protocol is created, the revised DTDs will have a new value for this
fixed" VERSION attribute, such that all message adhering to the new version must have the
new version number.
–>
<!– SC_CALLBACK_REG definition

The root element of the XML document is a registration message.  A registration message
consists of the callback port and the protocol version as attributes, and either an
ADD_CLIENT, ADD_EVENTS, REMOVE_CLIENT, or REMOVE_EVENTS attribute, specifying the
registration type.  The ADD_CLIENT, ADD_EVENTS, and REMOVE_EVENTS types should have one or
more SC_EVENT_REG subelements.  The REMOVE_CLIENT should not specify an SC_EVENT_REG
subelement.
        ATTRIBUTES:
               VERSION        The CRNP protocol version of the message.
               PORT           The callback port.
               REG_TYPE       The type of registration.  One of:
                       ADD_CLIENT, ADD_EVENTS, REMOVE_CLIENT, REMOVE_EVENTS

        CONTENTS:
SUBELEMENTS: SC_EVENT_REG (0 or more)
–>
<!ELEMENT SC_CALLBACK_REG (SC_EVENT_REG*)>
<!ATTLIST SC_CALLBACK_REG
       VERSION        NMTOKEN                                                #FIXED
       PORT           NMTOKEN                                                #REQUIRED
       REG_TYPE       (ADD_CLIENT|ADD_EVENTS|REMOVE_CLIENT|REMOVE_EVENTS)    #REQUIRED

>
<!– SC_EVENT_REG definition

The SC_EVENT_REG defines an event for which the client is either registering or
unregistering interest in receiving event notifications.  The registration can be for any
level of granularity, from only event class down to specific name/value pairs that must be
present.  Thus, the only required attribute is the CLASS.  The SUBCLASS attribute, and the
NVPAIRS sub-elements are optional, for higher granularity.
```

```
Registrations that specify name/value pairs are registering interest in notification of
messages from the class/subclass specified with ALL name/value pairs present.
Unregistrations that specify name/value pairs are unregistering interest in notifications
that have EXACTLY those name/value pairs in granularity previously specified.
Unregistrations that do not specify name/value pairs unregister interest in ALL event
notifications of the specified class/subclass.

        ATTRIBUTES:
               CLASS:        The event class for which this element is registering
                             or unregistering interest.
               SUBCLASS:     The subclass of the event (optional).

        CONTENTS:
               SUBELEMENTS: 0 or more NVPAIRs.
—>
<!ELEMENT SC_EVENT_REG (NVPAIR*)>
<!ATTLIST SC_EVENT_REG
       CLASS          CDATA                #REQUIRED
       SUBCLASS       CDATA                #IMPLIED
>
```

NVPAIR XML DTD

```
<!— NVPAIR XML format specification

        Copyright 2001-2006 Sun Microsystems, Inc. All rights reserved.
        Use is subject to license terms.

        Intended Use:
              An nvpair element is meant to be used in an SC_EVENT or SC_CALLBACK_REG
              element.
—>
<!— NVPAIR definition

       The NVPAIR is a name/value pair to represent arbitrary name/value combinations.
       It is intended to be a direct, generic, translation of the Solaris nvpair_t
       structure used by the sysevent framework.  However, there is no type information
       associated with the name or the value (they are both arbitrary text) in this xml
       element.

       The NVPAIR consists simply of one NAME element and one or more VALUE elements.
       One VALUE element represents a scalar value, while multiple represent an array
       VALUE.

       ATTRIBUTES:
```

```
      CONTENTS:
              SUBELEMENTS: NAME(1), VALUE(1 or more)
-->
<!ELEMENT NVPAIR (NAME,VALUE+)>
<!-- NAME definition

        The NAME is simply an arbitrary length string.

        ATTRIBUTES:

        CONTENTS:
                Arbitrary text data.  Should be wrapped with <![CDATA[...]]> to prevent XML
                parsing inside.
-->
<!ELEMENT NAME (#PCDATA)>

<!-- VALUE definition
        The VALUE is simply an arbitrary length string.

        ATTRIBUTES:

        CONTENTS:
                Arbitrary text data.  Should be wrapped with <![CDATA[...]]> to prevent XML
                parsing inside.
-->
<!ELEMENT VALUE (#PCDATA)>
```

SC_REPLY **XML DTD**

```
<!-- SC_REPLY XML format specification

        Copyright 2001-2006 Sun Microsystems, Inc. All rights reserved.
        Use is subject to license terms.
-->
<!-- SC_REPLY definition

        The root element of the XML document represents a reply to a message.  The reply
        contains a status code and a status message.

        ATTRIBUTES:
                VERSION:        The CRNP protocol version of the message.
                STATUS_CODE:    The return code for the message.  One of the
                                following: OK, RETRY, LOW_RESOURCES, SYSTEM_ERROR, FAIL,
                                MALFORMED, INVALID_XML, VERSION_TOO_HIGH, or
                                VERSION_TOO_LOW.

                CONTENTS:
```

```
                        SUBELEMENTS: SC_STATUS_MSG(1)
—>
<!ELEMENT SC_REPLY (SC_STATUS_MSG)>
<!ATTLIST SC_REPLY
        VERSION        NMTOKEN                                        #FIXED   "1.0"
        STATUS_CODE    OK|RETRY|LOW_RESOURCE|SYSTEM_ERROR|FAIL|MALFORMED|INVALID,\
                       VERSION_TOO_HIGH, VERSION_TOO_LOW) #REQUIRED
>
<!– SC_STATUS_MSG definition
        The SC_STATUS_MSG is simply an arbitrary text string elaborating on the status
        code.  Should be wrapped with <![CDATA[...]]> to prevent XML parsing inside.

        ATTRIBUTES:

        CONTENTS:
                Arbitrary string.
—>
<!ELEMENT SC_STATUS_MSG (#PCDATA)>
```

SC_EVENT **XML DTD**

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 2001-2006 Sun Microsystems, Inc. All rights reserved.
        Use is subject to license terms.

        The root element of the XML document is intended to be a direct, generic,
        translation of the Solaris 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
>
```

A P P E N D I X G

CrnpClient.java Application

This appendix shows the complete CrnpClient.java application that is discussed in more detail in Chapter 12, "Cluster Reconfiguration Notification Protocol."

Contents of CrnpClient.java

```
/*
 * CrnpClient.java
 * ================
 *
 * Note regarding XML parsing:
 *
 * This program uses the Sun Java Architecture for XML Processing (JAXP) API.
 * See http://java.sun.com/webservices/jaxp/ 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 registartion 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 obect, 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 registartion 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_memberhip.
 */
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
  * rgname 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
{
 /*
  * Crate 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
        //
        nl = elem.getElementsByTagName("VALUE");
        if (nl.getLength() != 1) {
         System.out.println("Error in parsing: can't find "
             + "VALUE 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
        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, 79
  ANYTIME, 77
  AT_CREATION, 78
  tunability values, 77
  WHEN_DISABLED, 78
  WHEN_OFFLINE, 78
  WHEN_UNMANAGED, 78
  WHEN_UNMONITORED, 77
(Resource Type Registration)
  file
    upgrading, 76

A

accessing network address, with DSDL, 124
administration commands, using to create a service that uses GDS, 202
administrative interface, RGM (Resource Group Manager), 26
Affinity_timeout, resource property, 254
Agent Builder
  analyzing the application, 163
  binary files, 180
  cloning existing resource type, 177
  Cluster Agent module, 184
    differences, 188
  command-line version, 178
  Configure screen, 171
  configuring, 164
  Create screen, 169
  creating a service that uses GDS with command-line version of, 204
  description, 20, 25
  directory structure, 179
  editing generated source code, 178
  installing, 164
  man pages, 181
  navigating in, 166
    Browse, 167
    Edit menu, 169
    File menu, 168
    menus, 168
  output, 201
  package directory, 183
  reusing code, 177
  rtconfig file, 183
  scripts, 181
  source files, 180
  starting, 165, 196
  support files, 182
  using, 163
  using to create a service that uses GDS, 196
  using to create GDS, 191
ANYTIME, #$upgrade_from directive, 77
API, Resource Management, *See* RMAPI
API_version, resource type property, 244
application environment, Sun Cluster, 19
arguments, RMAPI method, 69
Array_maxsize, resource property attribute, 288
Array_minsize, resource property attribute, 288

arraymax, resource type upgrade, 75
arraymin, resource type upgrade, 75
AT_CREATION, #$upgrade_from directive, 78
attributes, resource property, 287
Auto_start_on_new_cluster, resource group property, 274

B

binary files, Agent Builder, 180
Boot, resource type property, 245
Boot method, using, 50, 72
Boot_timeout, resource property, 254
Browse, Agent Builder, 167

C

C program functions, RMAPI, 65
callback method, overview, 20
callback methods
- control, 70
- description, 23
- initialization, 70
- Monitor_check, 74
- Monitor_start, 74
- Monitor_stop, 74
- naming conventions, 141
- Postnet_start, 73
- Prenet_start, 73
- RMAPI, 68
- Update, 73
- using, 54
- Validate, 73

Cheap_probe_interval, resource property, 254
checks, validating for scalable services, 58
client, CRNP, 217
cloning existing resource type, Agent Builder, 177
clsetup, description, 26
Cluster Agent module
- Agent Builder differences, 188
- description, 184
- installing, 184
- setting up, 184

Cluster Agent module *(Continued)*
- starting, 185
- using, 187

cluster commands, RMAPI, 65
cluster functions, RMAPI, 67
Cluster Reconfiguration Notification Protocol, *See* CRNP
code
- changing method, 82
- changing monitor, 81

codes, RMAPI exit, 69
command line
- Agent Builder, 178
- commands on, 27

commands
- clsetup, 26
- halockrun, 54
- hatimerun, 54
- RMAPI resource type, 64
- Sun Cluster, 27
- using to create a service that uses GDS, 202
- using to create GDS, 191

components, RMAPI, 24
concepts, CRNP, 213
Configure screen, Agent Builder, 171
configuring, Agent Builder, 164
conventions
- callback method names, 141
- function names, 141

Create screen, Agent Builder, 169
CRNP (Cluster Reconfiguration Notification Protocol)
- authentication, 223
- client, 217
- client identification process, 217
- communication, 215
- concepts, 213
- description, 214
- error conditions, 220
- example Java application, 224
- function of, 214
- message types, 215
- registration of client and server, 217
- SC_CALLBACK_REG messages, 217
- SC_EVENT, 221, 222

CRNP (Cluster Reconfiguration Notification Protocol) *(Continued)*
 SC_REPLY, 219
 semantics of protocol, 215
 server, 217
 server event delivery, 221
 server reply, 219

D

daemon, designing the fault monitor, 135
data service
 creating
  analyzing suitability, 29
  determining the interface, 31
 sample, 87
  common functionality, 94-98
  controlling the data service, 99
  defining a fault monitor, 104
  extension properties in RTR file, 93
  generating error messages, 97
  handling property updates, 114
  Monitor_check method, 113
  Monitor_start method, 110
  Monitor_stop method, 111
  obtaining property information, 98
  probe program, 105
  resource properties in RTR file, 90
  RTR file, 89
  Start method, 99
  Stop method, 102
  Update method, 118
  Validate method, 114
 setting up development environment, 32
 transferring to cluster for testing, 34
Data Service Development Library, *See* DSDL
data services
 testing, 59
 testing HA, 60
 writing, 59
debugging resource types with DSDL, 124
Default, resource property attribute, 288
default property values
 new value for upgrade, 79

default property values *(Continued)*
 Sun Cluster 3.0, 80
 when inherited, 79
dependencies, coordinating between resources, 60
Description, resource property attribute, 288
description values, rules, 347
Desired_primaries, resource group property, 274
directive
 #$upgrade, 77, 346
 #$upgrade_from, 77, 79
 default tunability, 78
 placement in RTR file, 77
 RT_version, 77
 tunability constraints, 77
directories, Agent Builder, 183
directory structure, Agent Builder, 179
distinguishing between multiple registered versions, *rt-version*, 76
distinguishing between vendors, *vendor-id*, 76
documentation requirements
 for upgrade, 83-85
 tunability constraints, 83
DSDL (Data Service Development Library)
 accessing network address, 124
 components, 25
 debugging resource types, 124
 description, 121, 122
 enabling HA local file systems, 125
 fault monitor functions, 212
 fault monitoring, 210
 general purpose functions, 207
 implementing a fault monitor, 123
 libdsdev.so, 20
 network resource access functions, 209
 overview, 20
 Process Monitor Facility (PMF) functions, 211
 property functions, 209
 sample resource type implementation
  determining the fault monitor action, 156
  returning from svc_start(), 144
  scds_initialize() function, 141
  starting the service, 142
  SUNW.xfnts fault monitor, 150
  SUNW.xfnts RTR file, 140

DSDL (Data Service Development Library), sample resource type implementation *(Continued)*
svc_probe() function, 152
TCP port number, 140
validating the service, 142
X font server, 139
X font server configuration file, 140
xfnts_monitor_check method, 149
xfnts_monitor_start method, 147
xfnts_monitor_stop method, 148
xfnts_probe main loop, 151
xfnts_start method, 142
xfnts_stop method, 146
xfnts_update method, 159
xfnts_validate method, 156
starting a data service, 123
stopping a data service, 123
utility functions, 212
where implemented, 20

E
editing generated Agent Builder source code, 178
enabling HA local file systems with DSDL, 125
enumeration literal names, rules, 345
Enumlist, resource property attribute, 288
error conditions, CRNP, 220
events, guaranteed delivery, 221
examples
data service, 87
Java application that uses CRNP, 224
exit codes, RMAPI, 69
extension properties
declaring, 42
resource property attribute, 288
resource type, 254

F
Failback, resource group property, 274
Failover, resource type property, 245
Failover_mode, resource property, 255
failover resource, implementing, 55
fault monitor
daemon
designing the, 135
functions, DSDL, 212
SUNW.xfnts, 150
files
binary in Agent Builder, 180
rtconfig, 183
source in Agent Builder, 180
support in Agent Builder, 182
Fini, resource type property, 246
Fini method, guidelines for implementing, 49
Fini method, using, 49-50, 71
Fini_timeout, resource property, 257
format, resource type names, 346
fully qualified resource type name, how obtained, 76
functions
DSDL fault monitor, 212
DSDL network resource access, 209
DSDL Process Monitor Facility (PMF), 211
DSDL property, 209
DSDL utility, 212
general purpose DSDL, 207
naming conventions, 141
RMAPI C program, 65
RMAPI cluster, 67
RMAPI resource, 65
RMAPI resource group, 67
RMAPI resource type, 66
RMAPI utility, 68
scds_initialize(), 141
svc_probe(), 152

G
GDS (generic data service)
Child_mon_level property, 193
creating a service with command-line version of Agent Builder, 204
definition, 45
description, 189
Failover_enabled property, 193
Log_level property, 194
Network_aware property, 194

GDS (generic data service) *(Continued)*
- Network_resources_used property, 194
- Port_list property, 192
- Probe_command property, 194
- Probe_timeout property, 195
- required properties, 192
- Start_command extension property, 192
- Start_timeout property, 195
- Stop_command property, 195
- Stop_signal property, 195
- Stop_timeout property, 196
- SUNW.gds resource type, 190
- using commands to create service that uses, 202
- using Sun Cluster Agent Builder to create service that uses, 196
- using with Sun Cluster administration commands, 191
- using with Sun Cluster Agent Builder, 191
- Validate_command property, 196
- Validate_timeout property, 196
- ways to use, 191
- when to use, 190
- why use, 190

generic data service
- *See* GDS

Global_resources_used, resource group property, 274

Global_zone, resource type property, 247

Global_zone_override, resource property, 258

H

HA data services, testing, 60

halockrun, description, 54

hatimerun, description, 54

I

idempotence, methods, 44

implementing
- fault monitor with DSDL, 123
- resource type monitor, 80
- resource type names, 80
- RMAPI, 20

Implicit_network_dependencies, resource group property, 275

Init, resource type property, 247

Init method, using, 49, 71

Init_nodes, resource type property, 248

Init_timeout, resource property, 258

installation requirements, resource type packages, 80

Installed_nodes, resource type property, 248

installing Agent Builder, 164

interface, RGM (Resource Group Manager), 26

interfaces
- command-line, 27
- programming, 24

Is_logical_hostname, resource type property, 248

Is_shared_address, resource type property, 248

J

Java, sample application that uses CRNP, 224

K

keep-alives, using, 59

L

legal names, Resource Group Manager (RGM), 345

libdsdev.so, DSDL, 20

libscha.so, RMAPI, 20

Load_balancing_policy, resource property, 258

Load_balancing_weights, resource property, 259

logging, adding to a resource, 53

M

man pages, Agent Builder, 181

master, description, 22

Max, resource property attribute, 288

max, resource type upgrade, 75

Maximum_primaries, resource group property, 275
Maxlength, resource property attribute, 288
menus
  Agent Builder, 168
  Agent Builder Edit, 169
  Agent Builder File, 168
message logging, adding to a resource, 53
messages
  SC_CALLBACK_REG CRNP, 217, 218-219
  SC_EVENT CRNP, 221, 222
  SC_REPLY CRNP, 219
method arguments, RMAPI, 69
method code, changing, 82
methods
  Boot, 50, 72, 134
  callback, 54
    control, 70
    initialization, 70
  Fini, 49-50, 71, 134
  Fini, guidelines for implementing, 49
  idempotence, 44
  Init, 49, 71, 134
  Monitor_check, 74, 133
  Monitor_check callback, 74
  Monitor_start, 74, 132
  Monitor_start callback, 74
  Monitor_stop, 74, 133
  Monitor_stop callback, 74
  Postnet_start, 73
  Postnet_start callback, 73
  Prenet_start, 73
  Prenet_start callback, 73
  Start, 47, 70, 130
  Stop, 47, 70, 131
  Update, 54, 73, 133
  Update callback, 73
  Validate, 54, 73, 128
  Validate callback, 73
  xfnts_monitor_check, 149
  xfnts_monitor_start, 147
  xfnts_monitor_stop, 148
  xfnts_start, 142
  xfnts_stop, 146
  xfnts_update, 159
methods *(Continued)*
  xfnts_validate, 156
Min, resource property attribute, 288
min, resource type upgrade, 75
Minlength, resource property attribute, 288
modifying resource types, 75
Monitor_check, resource type property, 248
Monitor_check method
  compatibility, 78
  using, 74
Monitor_check_timeout, resource property, 259
monitor code, changing, 81
Monitor_start, resource type property, 249
Monitor_start method, using, 74
Monitor_start_timeout, resource property, 259
Monitor_stop, resource type property, 249
Monitor_stop method, using, 74
Monitor_stop_timeout, resource property, 260
Monitored_switch, resource property, 260

N

naming conventions
  callback methods, 141
  functions, 141
navigating Agent Builder, 166
network resource access functions, DSDL, 209
Network_resources_used, resource property, 260
Nodelist, resource group property, 275
Num_resource_restarts, resource property, 261
Num_rg_restarts, resource property, 261

O

On_off_switch, resource property, 261
options, tunability, 77

P

package directory, Agent Builder, 183
Pathprefix, resource group property, 275
Per_node, resource property attributes, 288

Pingpong_interval, resource group property, 275
Pkglist, resource type property, 249
PMF (Process Monitor Facility)
  functions, DSDL, 211
  overview, 20
  purpose, 53
Port_list, resource property, 262
Postnet_start method, using, 73
Postnet_stop
  compatibility, 78
  resource type property, 249
Postnet_stop_timeout, resource property, 262
Prenet_start, resource type property, 249
Prenet_start method, using, 73
Prenet_start_timeout, resource property, 262
primary nodes, 22
process management, 53
Process Monitor Facility, *See* PMF
programming architecture, 20
programming interfaces, 24
properties
  changing resource, 54
  Child_mon_level, 193
  declaring extension, 42
  declaring resource, 37
  declaring resource type, 34
  Failover_enabled, 193
  GDS, required, 193
  Log_level, 194
  Network_aware, 194
  Network_resources_used, 194
  Port_list, 192
  Probe_command, 194
  Probe_timeout, 195
  resource, 253
  resource group, 273
  setting resource, 34, 54
  setting resource type, 34
  Start_command extension, 192
  Start_timeout, 195
  Stop_command, 195
  Stop_signal, 195
  Stop_timeout, 196
  Validate_command, 196
properties *(Continued)*
  Validate_timeout, 196
Property, resource property attribute, 288
property attributes, resource, 287
property functions, DSDL, 20
property names, rules, 345
property values
  default, 79
  rules, 347
property variables, 174
  how Agent Builder substitutes types of, 176
  list of, 175
  list of resource, 175
  list of resource group, 175
  list of resource type, 175
  syntax of, 176
Proxy, resource type property, 250

R

R_description, resource property, 263
registering CRNP clients and servers, 217
resource
  adding message logging to a, 53
  implementing a failover, 55
  implementing a scalable, 55
  monitoring, 50
  starting, 45
  stopping, 45
resource commands, RMAPI, 64
resource dependencies, coordinating, 60
Resource_dependencies, resource property, 263
Resource_dependencies_offline_restart, resource property, 264
Resource_dependencies_restart, resource property, 265
Resource_dependencies_weak, resource property, 267
resource functions, RMAPI, 65
resource group commands, RMAPI, 64
resource group functions, RMAPI, 67
Resource Group Manager, *See* RGM
Resource Group Manager (RGM)
  legal names, 345

Resource Group Manager (RGM) *(Continued)*
  values, 347
resource group names, rules, 345
resource group properties, 273
  accessing information about, 44
  Auto_start_on_new_cluster, 274
  Desired_primaries, 274
  Failback, 274
  Global_resources_used, 274
  Implicit_network_dependencies, 275
  Maximum_primaries, 275
  Nodelist, 275
  Pathprefix, 275
  Pingpong_interval, 275
  Resource_list, 276
  RG_affinities, 276
  RG_dependencies, 277
  RG_description, 277
  RG_is_frozen, 278
  RG_mode, 278
  RG_name, 278
  RG_project_name, 278
  RG_slm_cpu, 279
  RG_slm_cpu_min, 280
  RG_slm_pset_type, 282
  RG_slm_type, 281
  RG_state, 284
  RG_system, 286
  Suspend_automatic_recovery, 286
resource groups
  description, 22
  failover, 22
  properties, 22
  scalable, 22
Resource_list
  resource group property, 276
  resource type property, 250
Resource Management API, *See* RMAPI
Resource_name, resource property, 268
resource names, rules, 345
Resource_project_name, resource property, 268
resource properties, 253
  accessing information about, 44
  Affinity_timeout, 254

resource properties *(Continued)*
  Boot_timeout, 254
  changing, 54
  Cheap_probe_interval, 254
  declaring, 37
  extension, 254
  Failover_mode, 255
  Fini_timeout, 257
  Global_zone_override, 258
  Init_timeout, 258
  Load_balancing_policy, 258
  Load_balancing_weights, 259
  Monitor_check_timeout, 259
  Monitor_start_timeout, 259
  Monitor_stop_timeout, 260
  Monitored_switch, 260
  Network_resources_used, 260
  Num_resource_restarts, 261
  Num_rg_restarts, 261
  On_off_switch, 261
  Port_list, 262
  Postnet_stop_timeout, 262
  Prenet_start_timeout, 262
  R_description, 263
  Resource_dependencies, 263
  Resource_dependencies_offline_restart, 264
  Resource_dependencies_restart, 265
  Resource_dependencies_weak, 267
  Resource_name, 268
  Resource_project_name, 268
  Resource_state, 268
  Retry_count, 269
  Retry_interval, 269
  Scalable, 270
  setting, 34, 54
  Start_timeout, 270
  Status, 271
  Status_msg, 271
  Stop_timeout, 271
  Thorough_probe_interval, 271
  Type, 272
  Type_version, 272
  UDP_affinity, 272
  Update_timeout, 272

resource properties *(Continued)*
- Validate_timeout, 273
- Weak_affinity, 273

resource property attributes, 287
- Array_maxsize, 288
- Array_minsize, 288
- Default, 288
- Description, 288
- Enumlist, 288
- Extension, 288
- Max, 288
- Maxlength, 288
- Min, 288
- Minlength, 288
- Per_node, 288
- Property, 288
- Tunable, 288
- type, 289

Resource_state, resource property, 268
Resource_type, resource type property, 250
resource-type, upgrading, 76
resource type, what happens when upgrading, 79
resource type monitor, implementing, 80
resource type names
- implementing, 80
- obtaining fully qualified, 76
- restrictions, 78, 170
- rules, 346
- Sun Cluster 3.0, 79
- version suffix, 76
- without version suffix, 79

resource type packages, installation requirements, 80
resource type properties
- API_version, 244
- Boot, 245
- declaring, 34
- Failover, 245
- Fini, 246
- Global_zone, 247
- Init, 247
- Init_nodes, 248
- Installed_nodes, 248
- Is_logical_hostname, 248
- Is_shared_address, 248

resource type properties *(Continued)*
- Monitor_check, 248
- Monitor_start, 249
- Monitor_stop, 249
- Pkglist, 249
- Postnet_stop, 249
- Prenet_start, 249
- Proxy, 250
- Resource_list, 250
- Resource_type, 250
- RT_basedir, 251
- RT_description, 251
- RT_system, 251
- RT_version, 252
- setting, 34
- Single_instance, 252
- Start, 252
- Stop, 252
- Update, 252
- Validate, 252
- Vendor_ID, 253

resource type registration, *See* RTR
resource types
- commands
  - RMAPI, 64
- debugging with DSDL, 124
- description, 21
- functions
  - RMAPI, 66
- modifying, 75
- multiple versions, 75
- upgrading requirements, 75

resources
- coordinating dependencies between, 60
- description, 22

Retry_count, resource property, 269
Retry_interval, resource property, 269
reusing code, Agent Builder, 177
RG_affinities, resource group property, 276
RG_dependencies, resource group property, 277
RG_description, resource group property, 277
RG_is_frozen, resource group property, 278
RG_mode, resource group property, 278
RG_name, resource group property, 278

RG_project_name, resource group property, 278
RG_slm_cpu, resource group property, 279
RG_slm_cpu_min, resource group property, 280
RG_slm_pset_type, resource group property, 282
RG_slm_type, resource group property, 281
RG_state, resource group property, 284
RG_system, resource group property, 286
RGM (Resource Group Manager)
  administrative interface, 26
  description, 23
  handling of resource groups, 21
  handling of resource types, 21
  handling of resources, 21
  purpose, 20
RMAPI (Resource Management API), 20
  C program functions, 65
  callback methods, 68
  cluster commands, 65
  cluster functions, 67
  components, 24
  exit codes, 69
  libscha.so, 20
  method arguments, 69
  resource commands, 64
  resource functions, 65
  resource group commands, 64
  resource group functions, 67
  resource type commands, 64
  resource type functions, 66
  shell commands, 63
  utility functions, 68
  where implemented, 20
rt-version, upgrading, 76
RT_basedir, resource type property, 251
RT_description, resource type property, 251
RT_system, resource type property, 251
RT_version
  purpose, 78
  resource type property, 252
  when to change, 78
rtconfig file, 183
RTR (Resource Type Registration)
  description, 23
RTR (Resource Type Registration) *(Continued)*
  file
    changing, 81
    description, 128
    SUNW.xfnts, 140
rules
  description values, 347
  enumeration literal names, 345
  property names, 345
  property values, 347
  resource group names, 345
  resource names, 345

S

sample data service
  common functionality, 94-98
  controlling the data service, 99
  defining a fault monitor, 104
  extension properties in RTR file, 93
  generating error messages, 97
  handling property updates, 114
  Monitor_check method, 113
  Monitor_start method, 110
  Monitor_stop method, 111
  obtaining property information, 98
  probe program, 105
  RTR file, 89
  sample properties in RTR file, 90
  Start method, 99
  Stop method, 102
  Update method, 118
  Validate method, 114
sample DSDL code
  determining the fault monitor action, 156
  returning from svc_start(), 144
  scds_initialize() function, 141
  starting the service, 142
  SUNW.xfnts fault monitor, 150
  SUNW.xfnts RTR file, 140
  svc_probe() function, 152
  TCP port number, 140
  validating the service, 142
  X font server, 139

sample DSDL code *(Continued)*
- X font server configuration file, 140
- xfnts_monitor_check method, 149
- xfnts_monitor_start method, 147
- xfnts_monitor_stop method, 148
- xfnts_probe main loop, 151
- xfnts_start method, 142
- xfnts_stop method, 146
- xfnts_update method, 159
- xfnts_validate method, 156

SC_CALLBACK_REG, contents, 218-219
SC_EVENT, contents, 222
SC_REPLY, contents, 219
Scalable, resource property, 270
scalable resource, implementing, 55
scalable services, validating, 58
scds_initialize() function, 141
screens
- Configure, 171
- Create, 169

scripts
- Agent Builder, 181
- configuring, 199
- creating, 196

server
- CRNP, 217
- X font
  - configuration file, 140
  - definition, 139
- xfs
  - port number, 140

shell commands, RMAPI, 63
Single_instance, resource type property, 252
source code, editing generated Agent Builder, 177
source files, Agent Builder, 180
Start, resource type property, 252
Start method, using, 47, 70
Start_timeout, resource property, 270
starting a data service with DSDL, 123
Status, resource property, 271
Status_msg, resource property, 271
Stop, resource type property, 252
Stop method
- compatibility, 78

Stop method *(Continued)*
- using, 47, 70

Stop_timeout, resource property, 271
stopping a data service with DSDL, 123
Sun Cluster
- application environment, 19
- commands, 27
- using with GDS, 190

Sun Cluster Agent Builder, *See* Agent Builder
Sun Cluster Manager, description, 26
SUNW.xfnts
- fault monitor, 150
- RTR file, 140

support files, Agent Builder, 182
Suspend_automatic_recovery, resource group property, 286
svc_probe() function, 152
syntax
- description values, 347
- enumeration literal names, 345
- property names, 345
- property values, 347
- resource group names, 345
- resource names, 345
- resource type names, 346

T

TCP connections, using DSDL fault monitoring, 210
testing
- data services, 59
- HA data services, 60

Thorough_probe_interval, resource property, 271
tunability constraints, documentation requirements, 83
tunability options, 77
- ANYTIME, 77
- AT_CREATION, 78
- WHEN_DISABLED, 78
- WHEN_OFFLINE, 78
- WHEN_UNMANAGED, 78
- WHEN_UNMONITORED, 77

Tunable, resource property attribute, 288
Type, resource property, 272

type, resource property attributes, 289
Type_version, resource property, 272

U
UDP_affinity, resource property, 272
Update, resource type property, 252
Update method
    compatibility, 78
    using, 54, 73
Update_timeout, resource property, 272
upgrade aware, defined, 76
upgrade directive, 346
upgrades, documentation requirements, 83-85
upgrading resource types, 75
utility functions
    DSDL, 212
    RMAPI, 68

V
Validate, resource type property, 252
Validate method
    using, 54, 73
Validate_timeout, resource property, 273
validation checks, scalable services, 58
values
    default property, 79
    Resource Group Manager (RGM), 347
variables
    how Agent Builder substitutes types of property, 176
    list of property, 175
    list of resource group property, 175
    list of resource property, 175
    list of resource type property, 175
    property, 174
    syntax of property, 176
vendor-id
    distinguishing between, 76
    upgrading, 76
Vendor_ID, resource type property, 253

W
Weak_affinity, resource property, 273
WHEN_DISABLED, #$upgrade_from directive, 78
WHEN_OFFLINE, #$upgrade_from directive, 78
WHEN_UNMANAGED, #$upgrade_from directive, 78
WHEN_UNMONITORED, #$upgrade_from directive, 77
writing data services, 59

X
X font server
    configuration file, 140
    definition, 139
xfnts_monitor_check, 149
xfnts_monitor_start, 147
xfnts_monitor_stop, 148
xfnts_start, 142
xfnts_stop, 146
xfnts_update, 159
xfnts_validate, 156
xfs server, port number, 140