Sun Cluster 3.0 12/01 Data Services Developer's Guide

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900 U.S.A.
650-960-1300

Part No. 816-2025
December 2001, Revision A

Contents

Preface

The *Sun Cluster 3.0 12/01 Data Services Developer's Guide* contains information about using the Resource Management API to develop Sun Cluster data services.

This document is intended for experienced developers with extensive knowledge of Sun software and hardware. The information in this book assumes knowledge of the Solaris™ operating environment.

Using UNIX Commands

This document may not contain information on basic UNIX® commands and procedures such as shutting down the system, booting the system, and configuring devices.

See one or more of the following for this information:

- AnswerBook™ online documentation for the Solaris software environment
- Other software documentation that you received with your system
- Solaris operating environment man pages

Typographic Conventions

Typeface or Symbol	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail`.
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`%` **`su`** `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide.* These are called *class* options. You *must* be superuser to do this.
AaBbCc123	Command-line variable; replace with a real name or value	To delete a file, type `rm` *filename*.

Shell Prompts

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

Related Documentation

Application	Title	Part Number
Hardware	*Sun Cluster 3.0 12/01 Hardware Guide*	816-2023
Data Services	*Sun Cluster 3.0 12/01 Data Services Installation and Configuration Guide*	816-2024
API Development	*Sun Cluster 3.0 12/01 Data Services Developer's Guide*	816-2025
Administration	*Sun Cluster 3.0 12/01 System Administration Guide*	816-2026
Concepts	*Sun Cluster 3.0 12/01 Concepts*	816-2027
Error Messages	*Sun Cluster 3.0 12/01 Error Messages Manual*	816-2028
Release Notes	*Sun Cluster 3.0 12/01 Release Notes*	816-2029

Accessing Sun Documentation Online

The docs.sun.com[SM] web site enables you to access Sun technical documentation on the Web. You can browse the docs.sun.com archive or search for a specific book title or subject at:

`http://docs.sun.com`

Ordering Sun Documentation

Fatbrain.com, an Internet professional bookstore, stocks select product documentation from Sun Microsystems, Inc.

For a list of documents and how to order them, visit the Sun Documentation Center on Fatbrain.com at the following Web site.

`http://www1.fatbrain.com/documentation/sun`

Getting Help

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

- Your name and email address (if available)
- Your company name, address, and phone number
- The model number and serial number of your systems
- The release number of the operating environment (for example, Solaris 8)
- The release number of Sun Cluster (for example, Sun Cluster 3.0)

Use the following commands to gather information on your system 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
`prtdiag -v`	Displays system diagnostic information
`/usr/cluster/bin/scinstall -pv`	Displays Sun Cluster release and package version information

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

CHAPTER 1

Resource Management Overview

This book provides guidelines for creating a resource type for a software application such as Oracle, iPlanet™ Web Server, DNS, and so on. As such, this book is targeted at resource type developers. This book uses the second person pronoun, “you”, throughout to address resource type developers.

This chapter provides an overview of the concepts you need to understand in order to develop a data service and contains the following information.

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 cluster facility known as the Resource Group Manager, or RGM, provides the mechanism for high availability and scalability. The elements that form the programming interface to this facility include the following.

- A set of callback methods you write that enable the RGM to control an application on 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 management facilities for monitoring and restarting processes on 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 and adds some additional functionality to ease the writing of callback methods. These functions are implemented in the `libdsdev.so` library.

FIGURE 1-1 shows the interrelationship of these elements.

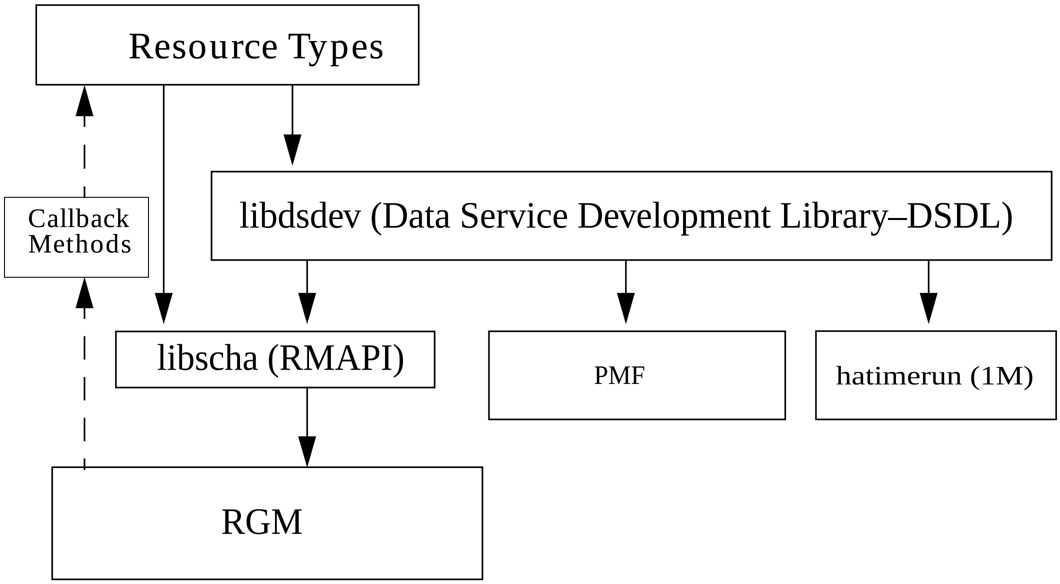


FIGURE 1-1 Programming Architecture

Included in the Sun Cluster package is SunPlex Agent Builder™, a tool that automates the process of creating a data service (see Chapter 8). Agent Builder generates data service code in either C (using DSDL functions to write the callback methods) or in Korn shell (ksh) (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 nodes according to pre-configured policies. The RGM makes a resource highly available in the event of a node failure or reboot by stopping the resource on the affected node and starting it on another. The RGM also automatically starts and stops resource-specific monitors that can detect resource failures and relocate failing resources onto another node or can monitor other aspects of resource performance.

The RGM supports both failover resources, which can be online on at most one node at a time, and scalable resources, which can be online on multiple nodes simultaneously.

RGM Model

This section introduces some fundamental terminology and explain 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, such as the following.

A developer implements 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 they can run on different nodes and fail over independently.

A developer creates 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 so that both resources run on the same node and fail over together.

Resource Types

A resource type consists of a software application to be run on the cluster, control programs used as callback methods by the RGM to manage the application as a cluster resource, and 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.

The resource type developer specifies the properties for the resource type and sets their values in a resource type registration (RTR) file. The RTR file follows a well-defined format described in "Setting Resource and Resource Type Properties" on page 35 and in the `rt_reg(4)` man page. See also "Defining the Resource Type Registration File" on page 77 for a description of a sample resource type registration file.

TABLE A-1 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 into the cluster configuration the information from the resource type registration file. The *Sun Cluster 3.0 12/01 Data Services Installation and Configuration Guide* describes the procedure for registering a data service.

Resources

A resource inherits the properties and values of its resource type. In addition, a developer can declare resource properties in the resource type registration file. See TABLE A-2 for a list of resource properties.

The cluster administrator can change the values of certain properties depending on how they were specified in the resource type registration (RTR) file. For example, property definitions can specify a range of allowable values and specify when the property is tunable, for example, at creation, anytime, or never. Within these specifications, the cluster administrator can make changes to properties using administration commands.

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

Resource Groups

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 invokes callback methods on the individual resources in the group.

The nodes on which 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, set by the cluster administrator, which identifies all *potential primaries* or masters of the resource group.

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 one node only at any time while 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 “Implementing a Failover Resource” on page 53 and “Implementing a Scalable Resource” on page 54 for details on these properties.

See TABLE A-3 for a list of resource group properties.

Resource Group Manager

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

The RGM supports the following functions:

- Whenever a node boots or crashes, the RGM attempts to maintain availability of all managed resource groups by automatically bringing them online on appropriate 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, a resource group, or a resource type

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 invokes a resource-type specific callback method on that resource. Callback methods are described in “Callback Methods” on page 24.

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 callable by the RGM. That is, the data service does not consist of a single executable but rather consists of a number of executable scripts (ksh) or binaries (C), each of which is directly callable by the RGM.

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

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
  - `START` and `STOP` start and stop resources in a group that is being brought online or offline.
  - `INIT`, `FINI`, `BOOT` execute initialization and termination code on resources.
- Administrative support methods
  - `VALIDATE` verifies properties set by administrative action.
  - `UPDATE` updates the property settings of an online resource.
- Net-relative methods
  - `PRENET_START` and `POSTNET_STOP` do special startup or shutdown actions before network addresses in the same resource group are configured up or after they are configured 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 and the `rt_callbacks(1HA)` man page for more information on the callback methods. Also see Chapter 4 and Chapter 7 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 built on top of the base API, and a tool, SunPlex Agent Builder, that automatically generates a data service from some basic inputs you provide.

Resource Management API

The Resource Management API (RMAPI) provides a set of low-level routines that enable a data service to access information about the resources, resource types and resource groups in the system, request a local restart or failover, and 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 `scha_calls(3HA)` and Chapter 3 for more information on the RMAPI routines. Also see Chapter 4 for examples of how to use these routines in sample data service callback methods.

Data Service Development Library (DSDL)

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 DSDL brings together various facilities for data-service development, including:

- `libscha.so`—the low-level resource management APIs
- PMF—the process management facility, which provides a means of monitoring processes and their descendants, and restarting them if they die (see `pmfadm`(1M) and `rpc.pmfd`(1M))
- `hatimerun`—a facility for running programs under a timeout (see `hatimerun`(1M).

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. The `libdsdev.so` library contains the DSDL functions.

See Chapter 5 and the `scds_calls`(3HA) man page for more information about the DSDL.

SunPlex 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, complete with source and executable code (C or Korn shell), 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 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 and a set of commands for administering a cluster.

SunPlex Manager

SunPlex 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 3.0 12/01 Software Installation Guide* for instructions on how to install SunPlex Manager and how to use SunPlex Manager to install cluster software. SunPlex Manager provides online help for most administrative tasks.

Administrative Commands

The Sun Cluster 3.0 commands for administering RGM objects are `scrgadm(1M)`, `scswitch(1M)`, and `scstat(1M) -g`.

The `scrgadm(1M)` command allows viewing, creating, configuring and deleting the resource type, resource group, and resource objects used by the RGM. The command is part of the administrative interface for the cluster, and is not to be used in the same programming context as the application interface described in the rest of this chapter. However, `scrgadm(1M)` is the tool for constructing the cluster configuration in which the API operates. Understanding the administrative interface sets the context for understanding the application interface. Refer to the `scrgadm(1M)` man page for details on the administrative tasks that can be performed by the command.

The `scswitch(1M)` command switches resource groups online and offline on specified nodes and enables or disables a resource or its monitor. See the `scswitch(1M)` man page for details on the administrative tasks that the command can perform.

The `scstat(1M) -g` command shows the current dynamic state of all resource groups and resources.

CHAPTER 2

Developing a Data Service

This chapter provides detailed information about developing a data service.

The following information is in this chapter.

Analyzing the Application for Suitability

The first step in creating a data service is to determine that 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 so.

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, refer to Appendix B.

Note – A scalable service must meet all the following conditions for high availability as well as some additional criteria.

- Both network aware (client-server model) and non network aware (client less) applications are potential candidates for being made highly available or scalable in the Sun Cluster environment. However Sun Cluster cannot provide enhanced availability in time-sharing environments in which applications are run on a server that is accessed through `telnet` or `rlogin`.
- The application must be crash tolerant. That is, it must recover disk data (if necessary) when it is started after an unexpected node death. 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 hostname of the node on which it is running. See "Host Names" on page 268 for additional information.
- The application must operate correctly in environments in which multiple IP addresses are configured up; for example, environments with multihomed hosts, in which the node is on more than one public network, and environments with nodes on which multiple, logical interfaces are configured up on one hardware interface.
- To be highly available, the application data must reside in the cluster file systems—see "Multihosted Data" on page 266.

  If the application uses a hard-wired path name for the location of the data, you could change that path to a symbolic link that points to a location in the cluster file system, without changing application source code. See "Using Symbolic Links for Multihosted Data Placement" on page 267 for additional information.

- Application binaries and libraries can reside locally on each node or on the cluster file system. The advantage of residing on the cluster file system is that a single installation is sufficient. The disadvantage is that rolling upgrade becomes an issue because the binaries are in use while the application is running under control of the RGM.
- The client should have some capacity to retry a query automatically if the first attempt times out. If the application and protocol already handle the case of a single server crashing and rebooting, then they also will handle the case of the containing resource group being failed over or switched over. See "Client Retry" on page 270 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, you should 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 Scalable Resource" on page 54 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 routines (in the `libscha.so` library)
- The Data Service Development Library (DSDL), a set of higher level functions (in the `libdsdev.so` library) that encapsulate the functionality of the RMAPI and provides some additional functionality

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

The recommended approach to developing a data service is:

1. Decide whether to code in C or Korn shell (Ksh). If you decide on Ksh, you cannot use the DSDL, which provides a C interface only.

2. Run Agent Builder, specify the requested inputs, 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 could take numerous approaches to creating a data service. For example, rather than add your own code to specific places in the code generated by Agent Builder, you could replace entirely one of the generated methods or the generated monitor program with a program you write from scratch using DSDL or RMAPI functions. However, regardless of the manner you proceed, in almost every case, starting with Agent Builder makes sense, for the following reasons:

- The code generated by Agent Builder, while generic in nature, has been tested in numerous data services.
- Agent Builder generates an RTR file, a make file, 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 a considerable amount of 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 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 beginning data service development, you must have installed 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, typically, you do development on a separate, non-cluster development machine, not on a cluster node. In this typical case, you must use `pkgadd`(1M) to install the `SUNWscdev` package on your development machine.

When compiling and linking your code, you must set particular options to identify the header and library files. When you have finished development (on a non-cluster node) you can transfer the completed data service to a cluster for running and testing.

Note – Be certain you are using a development version of Solaris 5.8 or higher.

Use the procedures in this section to:

- Install the Sun Cluster development package (`SUNWscdev`) and set the appropriate 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. **Change directory to the** *appropriate CD-ROM directory***.**

```
cd appropriate_CD-ROM_directory
```

2. **Install the SUNWscdev package in the current directory.**

```
pkgadd -d . SUNWscdev
```

3. **In the makefile, specify compiler and linker options to 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 on the cluster.

```
# Makefile for sample data service
...
-I /usr/cluster/include
-L /usr/cluster/lib
-R /usr/cluster/lib
...
```

▼ How to Transfer a Data Service to a Cluster

When you have completed development of a data service on a development machine, you must transfer it to a cluster for testing. To reduce the chance of error, the best way to accomplish this transfer is to package together the data service code and the RTR file and then install the package on all nodes of the cluster.

Note – Whether you use `pkgadd` or some other way to install the data service, you must put the data service on all cluster nodes. Agent Builder automatically packages together the RTR file and data service code.

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, and so on, as well as paths to each of the callback methods. TABLE A-1 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. TABLE A-2 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 the cluster administrator registers the data service with Sun Cluster.

It is recommended that you use Agent Builder to generate the RTR file for your data service because Agent Builder declares the set of properties that are both useful and required for any data service. For example certain properties (such as `Resource_type`) must be declared in the RTR file or registration of the data service fails. Other properties, though not required, will not be available to a system administrator unless you declare them in the RTR file, while some properties are available whether you declare them or not, because the RGM defines them and provides a default value. To avoid this level of complexity, you can simply use Agent Builder to guarantee generation of a proper RTR file. Later on you can edit the RTR file to change specific values if you need to do so.

The rest of this section leads you through a sample RTR file, created by Agent Builder.

Declaring Resource Type Properties

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

Note – One resource type property, `Installed_nodes`, is configurable by a system administrator. In fact, it is only configurable by a system administrator and you cannot declare it in the RTR file.

The syntax for resource type declarations is:

property_name = *value*;

Note – The RGM treats property names as case insensitive. The convention for properties in Sun-supplied RTR files, with the exception of method names, is uppercase for the first letter of the name and lowercase for the rest of the name. Method names—as well as property attributes—contain all uppercase letters.

Following are the 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 will fail.

The first set of resource type declarations provide basic information about the resource type, as follows:

- `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 using the `Vendor_id` as a prefix with a "." separating it from the resource type (`SUNW.smpl`), as in the sample. If you use `Vendor_id`, make it the stock symbol for the company defining the resource type. The resource type name must be unique in the cluster.

Note – By convention, the resource type name (*Resource_typeVendor_id*) is used as the package name. Package names are limited to nine characters, so it is a good idea to limit the total number of characters in these two properties to nine or fewer characters, though the RGM does not enforce this limit. Agent Builder, on the other hand, explicitly generates the package name from the resource type name, so it does enforce the nine character limit.

- `Rt_version`—Identifies the version of the sample data service.
- `API_version`—Identifies the version of the API. `API_version = 2`, indicates that the data service runs under Sun Cluster version 3.0.
- `Failover = TRUE`—Indicates that the data service cannot run in a resource group that can be online on multiple nodes at once, that is, specifies a failover data service. See "Implementing a Failover Resource" on page 53 for more information.
- `START`, `STOP`, `VALIDATE`, and so on—Provide the paths to the respective callback method programs called by the RGM. These paths are relative to the directory specified by `RT_basedir`.

The remaining resource type declarations provide configuration information, as follows:

- 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 specified by RG_PRIMARIES is 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 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, VALIDATE, and so on—Provide the paths to the respective callback method programs called by the RGM. These paths are relative to the directory specified by RT_basedir.

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 curly brackets:

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

For resource properties provided by Sun Cluster, so-called *system-defined* properties, you can change specific attributes in the RTR file. For example, Sun Cluster provides method timeout properties for each of the callback methods, and specifies default values. In the RTR file, you can specify different default values.

You can also define new resource properties in the RTR file—so-called *extension* properties—using a set of property attributes provided by Sun Cluster. TABLE A-4 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 defined by the property attributes in the RTR file. For example, the default value for each method timeout in the sample is 300 seconds. An administrator can change this value; however, the minimum allowable value, specified by the `MIN` attribute, is 60 seconds. See TABLE A-4 for a complete 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 = AT_CREATION;
        DEFAULT = ;
}
```

These resource-property declarations add the TUNABLE attribute, which limits the occasions on which the system administrator can change their values. AT_CREATION means the administrator can only specify the value when the resource is created and cannot change it 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 192 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, Retry_interval—Used in the fault monitor. The tunability is anytime, so a system administrator can adjust them if the fault monitor is not functioning optimally.
- Network_resources_used—A list of logical hostname or shared address resources used by the data service. Agent Builder declares this property so a system administrator can specify a list of resources, if there are any, when configuring the data service.
- Scalable—Set to FALSE to indicate this resource does not use the cluster networking (shared address) facility. This setting is consistent with the resource type Failover property set to TRUE to indicate a failover service. See "Implementing a Failover Resource" on page 53 and "Implementing a Scalable Resource" on page 54 for additional information about how to use this property.
- Load_balancing_policy, Load_balancing_weights—Agent Builder automatically declares these properties, however, they have no use in a failover resource type.
- Port_list—Identifies the list of ports on which the server is listening. Agent Builder declares this property so a system administrator can specify a list of ports, when configuring the data service.

Declaring Extension Properties

At the end of the sample RTR file are extension properties, as shown in the following 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, 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 = 30;
        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 some extension properties that are useful for most data services, as follows.

- `Confdir_list`—Specifies the path to the application configuration directory, which is useful information for many applications. The system administrator can provide the location of this directory when configuring the data service.
- `Monitor_retry_count`, `Monitor_retry_interval`, `Probe_timeout`—Control restarts of the fault monitor itself, not of the server daemon.
- `Child_mon_level`—Sets the level of monitoring to be done by PMF. See `pmfadm` (1M) for more information.

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

Implementing Callback Methods

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

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 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 for setting `Status` and `Status_msg`). Rather, you should store dynamic state information in global files.

Note – The cluster administrator can set certain resource properties using the `scrgadm`(1M) command or through an available graphical administrative command or through an available graphical administrative interface. However, do not call `scrgadm` from any callback method because `scrgadm` fails during cluster reconfiguration, that is, when the RGM calls the method.

Idempotency for 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 could 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 invoke its STOP method on it. The same scenarios apply to the MONITOR_START and MONITOR_STOP methods.

For these reasons, you must build idempotency into your STOP and MONITOR_STOP methods, which means that repeated calls of STOP or MONITOR_STOP on the same resource with the same parameters achieve the same results as a single call.

One implication of idempotency is that STOP and MONITOR_STOP must return 0 (success) even if the resource or monitor is already stopped and no work is to done.

Note – The INIT, FINI, BOOT, and UPDATE methods must also be idempotent. A START method need not be idempotent.

Controlling an Application

Callback methods enable the RGM to take control of the underlying resource (application) whenever nodes are in the process of joining or leaving the cluster.

Starting and Stopping a Resource

A resource type implementation requires, at a minimum, a `START` method and a `STOP` method. The RGM calls a resource type's method programs at appropriate times and on the appropriate nodes for bringing resource groups offline and online. For example, after the crash of a cluster node, the RGM moves any resource groups mastered by that node onto a new node. You must implement a `START` method to provide the RGM with 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 requiring a long initialization period have sufficiently long timeouts set on their `START` methods (set default and minimum values for the `Start_timeout` property in the resource type registration 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 on Node1 and back online on Node2. While taking the resource group offline, the RGM calls the `STOP` method on resources in the group to stop all activity on Node1. After the `STOP` methods for all resources have completed on Node1, the RGM brings the resource group back online on Node2.

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 would terminate all processes on the local node related to the resource. Resource types requiring a long time to shut down should have sufficiently long timeouts set on their `STOP` methods. Set the `Stop_timeout` property in the resource type registration file.

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

Deciding on the START and STOP Methods to Use

This section provides some tips about when to use the START and STOP methods versus 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 which methods are appropriate.

Services that use network address resources might require that start or stop steps be done in a certain order relative to the logical hostname address configuration. The optional callback methods PRENET_START and POSTNET_STOP allow a resource type implementation to do special start-up and shutdown actions before and after network addresses in the same resource group are configured up or configured down.

The RGM calls methods that plumb (but do not configure up) the network addresses 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 data service's PRENET_START method (if any).
3. Configure network addresses up.
4. Call data service's START method (if any).

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

1. Call data service's STOP method (if any).
2. Configure network addresses down.
3. Call 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 containing both data service application resources and network address resources, the RGM calls methods to configure up the network addresses before it calls the data service resource START methods. Therefore, if a data service requires network addresses to be configured 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 down the network addresses after it calls the data service resource STOP methods. Therefore, if a data service requires network addresses to be configured 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 invoke 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 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 will respond differently depending on whether the network address or the data service comes online first after a cluster reconfiguration, scha_control giveover, or scswitch switchover. For example, the client implementation might do minimal 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 up when it starts, start it before the network interface is configured up. This ensures that the data service is able to respond immediately to client requests as soon as the network address has been configured up, and 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 down is the POSTNET_STOP method invoked. 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 also is not responding.

The decision to use the START and STOP methods versus the PRENET_START and POSTNET_STOP methods, or to use both, must take the requirements and behavior of both the server and client into account.

Initializing and Terminating a Resource

Three optional methods, `INIT`, `FINI`, and `BOOT`, allow the RGM to execute initialization and termination code on a resource. The RGM invokes the `INIT` method to perform a one-time initialization of the resource when the resource becomes managed—either when the resource group it is in is switched from an unmanaged to a managed state, or when it is created in a resource group that is already managed.

The RGM invokes the `FINI` method to clean up after the resource when the resource becomes unmanaged—either when the resource group it is in is switched to an unmanaged state or when it is deleted from a managed resource group. The clean up must be idempotent, that is, if the clean up has already been done, `FINI` exits 0 (success).

The RGM invokes the `BOOT` method on nodes that have newly joined the cluster, that is, have been booted or rebooted.

The `BOOT` method normally performs the same initialization as `INIT`. This initialization must be idempotent, that is, if the resource has already been initialized on the local node, `BOOT` and `INIT` exit 0 (success).

Monitoring a Resource

Typically, you implement monitors to run periodic fault probes on resources to detect whether the probed resources are functioning correctly. If a fault probe fails, the monitor can attempt to restart locally or request failover of the affected resource group by calling the scha_control(3HA) RMAPI function or the scds_fm_action(3HA) DSDL function.

You can also monitor the performance of a resource and tune or report performance. Writing a resource type-specific fault monitor is completely 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, it 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(3HA) RMAPI function and the scds_fm_action(3HA) DSDL function (which calls scha_control) allow resource monitors to request the failover of a resource group to a different node. As one of its sanity checks, scha_control calls MONITOR_CHECK (if defined), to determine if the requested node is reliable enough to master the resource group containing 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 RMAPI scha_resource_setstatus(1HA), (3HA) command or function, or the DSDL scds_fm_action(3HA) function to set these properties.

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

See "Defining a Fault Monitor" on page 75 for an example of a fault monitor implemented with the RMAPI. See "The SUNW.xfnts Fault Monitor" on page 111 for an example of a fault monitor implemented with the DSDL. See the *Sun Cluster 3.0 12/01 Data Services Installation and Configuration Guide* for information on fault monitors built into Sun supplied data services.

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 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(1HA)(3HA)` interface.

Providing Process Management

The Resource Management API and the DSDL provide process management facilities to implement resource monitors and resource control callbacks. The RMAPI defines the following facilities (see the man pages for details on each of these commands and programs):

- Process Monitor Facility: `pmfadm(1M)` and `rpc.pmfd(1M)` — The Process Monitor Facility (PMF), provides a means of monitoring processes and their descendants, and restarting them if they die. The facility consists of the `pmfadm`(1M) command for starting and controlling monitored processes, and the `rpc.pmfd`(1M) daemon.
- `halockrun`(1M) — A program for running a child program while holding a — A program for running a child program while holding a file lock. This command is convenient for use in shell scripts.
- `hatimerun`(1M) — A program for running a child program under time-out control. This is a convenience command for use in shell scripts.

The DSDL provides the `scds_hatimerun`(3HA) function to implement the `hatimerun` functionality.

The DSDL provides a set of functions (`scds_pmf_`*(3HA)) to implement the PMF functionality. See “PMF Functions” on page 185 for an overview of the DSDL PMF functionality and for a list of the individual functions.

Providing Administrative Support for a Resource

Administrative actions on resources include setting and changing resource properties. The API defines the VALIDATE and UPDATE callback methods so you can hook into these administrative actions.

The RGM calls the optional VALIDATE method when a resource is created and when administrative action 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 indicated by the Init_nodes property of the resource's type (see "Resource Type Properties" on page 188, or the rt_properties(5) man page, for information about Init_nodes. 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 fail.

The RGM calls VALIDATE only when resource or group properties are changed through administrative action, not when the RGM sets properties, or when a monitor sets the resource properties Status and Status_msg.

The RGM calls the optional UPDATE method to notify a running resource that properties have been changed. The RGM invokes UPDATE after an administrative action 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 logical hostname and shared address, 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.

Set the boolean resource type property, `Failover`, 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. This property defaults to `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 allows 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 and restricts the resource group to being mastered by a single node. The RGM does not allow a resource whose `Failover` property is `TRUE` to be created in a resource group whose `RG_mode` is `SCALABLE`.

The `Implicit_network_dependencies` resource group property specifies that the RGM should enforce implicit strong dependencies of non-network-address resources on all network-address resources (logical hostname and shared address) within the group. This means that the non-network address (data service) resources in the group will not have their `START` methods called until the network addresses in the group are configured up. The `Implicit_network_dependencies` property defaults to `TRUE`.

Implementing a Scalable Resource

A scalable resource can be online on more than one node simultaneously. Scalable resources include data services such as Sun Cluster HA for iPlanet Web Server and HA-Apache.

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

Set the boolean resource type property, `Failover`, 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 have a value greater than 1, meaning the 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 in different resource groups. A scalable data service requires that its network address (shared address) resources be configured up before it is started. Therefore, the administrator must set the `RG_dependencies` property (of the resource group containing the scalable service) to include the resource group containing 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 used by this resource. This property defaults to the empty string so the cluster administrator must provide the actual list of shared addresses the scalable service uses when creating the resource. The `scsetup`(1M) command and SunPlex Manager provide features to automatically set up the necessary resources and groups for scalable services.

- `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 creating the resource (unless you set tunability for Load_balancing_policy to `NONE` or `FALSE` in the RTR file). Legal values are:
  - LB_WEIGHTED – 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 (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 it is coming to.

  For a scalable service with `Load_balancing_policy` `LB_STICKY` or `LB_STICKY_WILD`, changing `Load_balancing_weights` while the service is online can cause existing client affinities to be reset. In that 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*, where *weight* is an integer reflecting the relative portion of load distributed to the specified *node*. The fraction of load 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 1/4 of the load and node 2 receives 3/4.
- `Port_list` – identifies the ports on which the server 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 can be configured by the administrator 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 into a scalable resource group. The administrator can enable shared addresses by changing the value of `Scalable` to `TRUE` when creating the resource, and thusly create a scalable service.

On the other hand, even though `Failover` is set to `FALSE`, the administrator can configure the resource into a failover resource group to implement a failover service. The administrator does not change the value of `Scalable`, which is `FALSE`. To support this contingency, 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.

See *Sun Cluster 3.0 12/01 Concepts* for additional information regarding scalable resources.

Validation Checks For Scalable Services

Whenever a resource is created or updated with the scalable property set to `TRUE`, the RGM validates various resource properties. If the properties are not configured correctly, the RGM rejects the attempted update or creation. The RGM performs the following checks:

- The `Network_resources_used` property must be non-empty and contain the names of existing shared address resources. Every node in the `Nodelist` of the resource group containing the scalable resource must appear in either the `NetIfList` property or `AuxNodeList` property of each of the named shared address resources.
- The RG_dependencies property of the resource group that contains the scalable resource must include the resource groups of all shared address resources listed in the scalable resource's `Network_resources_used` property.
- The Port_list property must be non-empty and contain a list of port-protocol pairs such that protocol is either tcp or udp. For example,

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

Writing and Testing Data Services

This section provides some information about writing and testing data services.

Using Keep-Alives

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), eventually the wasted resources grow without bound as clients crash and reboot.

If the client-server communication uses a TCP stream, then 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. 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, and 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, because 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 supports 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 the HA environment. The test cases are suggestions and are not exhaustive. You need access to a test-bed Sun Cluster configuration so the testing work does not impact production machines.

Test that your HA data service behaves properly in all cases where a resource group is moved between physical hosts. These cases include system crashes and the use of the `scswitch(1M)` command. Test that client machines continue to get service after these events.

Test the idempotency 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. Informally, a data service A depends on a 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 `scrgadm(1M)` man page for details.

If resources of your resource type depend on resources of another type, you need to instruct the user to configure the resources and resource groups appropriately, or provide scripts or tools to correctly configure them. If the dependent resource must run on the same node as the depended-on resource, then both resources must be configured in the same resource group.

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

Some data services store no data directly themselves, but instead 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 has 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. Your end user registers the appointment calendar resource type with Sun Cluster.

If the appointment calendar application must run on the same node as the Oracle database, then the end user configures the appointment calendar resource in the same resource group as the HA-ORACLE resource, and makes the appointment calendar resource dependent on the HA-ORACLE resource. This dependency is specified using the `Resource_dependencies` property tag in `scrgadm`(1M).

If the HA-ORACLE resource is able to run on a different node than the appointment calendar resource, the end user configures them into two separate resource groups. The end user might configure a resource group dependency of the calendar resource group on the Oracle resource group. However resource group dependencies are only effective when both resource groups are being started or stopped on the same node at the same time. Therefore, the calendar data service daemon, after it has been started, might poll waiting for the Oracle database to become available. The calendar resource type's START method usually would just return success in this case, because if the START method blocked indefinitely it would put its resource group into a busy state, which would prevent any further state changes (such as edits, failovers, or switchovers) on the group. However, if the calendar resource's START method timed-out or exited non-zero, it might cause the resource group to ping-pong between two or more nodes while the Oracle database remained unavailable.

CHAPTER 3

Resource Management API Reference

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

The information in this chapter includes:

- “RMAPI Access Methods” on page 62 – in the form of shell script commands (1HA) and C functions (3HA)
  - `scha_resource_get(1HA)` (`scha_resource_open_get_close(3HA)`)
  - `scha_resource_setstatus(1HA)(3HA)`
  - `scha_resourcetype_get(1HA)` `scha_resourcetype__open_get_close(3HA)`
  - `scha_resource_resourcegroup_get(1HA)(3HA)` scha_resource_resourcegroup_open_get_close(3HA)
  - `scha_control(1HA)(3HA)`
  - `scha_cluster_get(1HA)` scha_resource_cluster_open_get_close(3HA)
  - scha_cluster_getlogfacility(3HA)
  - scha_cluster_getnodename(3HA)
  - `scha_strerror(3HA)`
- “RMAPI Callback Methods” on page 69 – described in the `rt_callbacks(1HA)` man page.
  - `START`
  - `STOP`
  - `INIT`
  - `FINI`
  - `BOOT`
  - `PRENET_START`
  - `PRENET_STOP`

- MONITOR_START
- MONITOR_STOP
- MONITOR_CHECK
- UPDATE
- VALIDATES

RMAPI Access Methods

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

RMAPI Shell Commands

Shell commands are to be used in shell script implementations of the callback methods for resource types representing services controlled by the cluster's RGM. You can use these commands to:

- Access information about resources, resource types, resource groups, and clusters
- Use with a monitor to 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 individual (1HA) man pages provide the definitive reference for the shell commands. Each command has a man page of the same name 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(1HA) – Accesses information about a resource or resource type under the control of the RGM. It provides the same information as the scha_resource_get(3HA) function.
- scha_resource_setstatus(1HA) – Sets the Status and Status_msg properties of a resource under the control of the RGM. It is used by the resource's monitor to indicate the resource's state as perceived by the monitor. It provides the same functionality as the scha_resource_setstatus(3HA) C function.

Note – Although scha_resource_setstatus is of particular use to a resource monitor, any program can call it.

Resource Type Command

This command accesses information about a resource type registered with the RGM.

- scha_resourcetype_get(1HA) – This command provides the same functionality as the scha_resourcetype_get(3HA) C function.

Resource Group Commands

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

- scha_resourcegroup_get(1HA) – Accesses information about a resource group under the control of the RGM. This command provides the same functionality as the scha_resourcetype_get(1HA) C function.
- scha_control(1HA) – Requests the restart of a resource group under the control of the RGM or its relocation to a different node. This command provides the same functionality as the scha_control(3HA) C function.

Cluster Command

This command accesses information about a cluster, such as node names, IDs, and states, the cluster name, resource groups, and so on.

- scha_cluster_get(1HA) – This command provides the same information as the scha_cluster_get(3HA) C function.

C Functions

C functions are to be used in C program implementations of the callback methods for resource types representing services controlled by the cluster's RGM. You can use these functions to do the following:

- Access information about resources, resource types, resource groups, and clusters
- Use with a monitor to set the `Status` and `Status_msg` properties of a resource
- Request the restart or relocation of a resource group
- Convert an error code to an appropriate error message

Note – Although this section provides brief descriptions of the C functions, the individual `(3HA)` man pages provide the definitive reference for the C functions. Each function has a man page of the same name unless otherwise noted.

See the `scha_calls(3HA)` man page for information on the output arguments and return codes of the C functions.

Resource Functions

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

- scha_resource_open(3HA), scha_resource_get(3HA), and scha_resource_close(3HA) – Together these functions access information on a resource 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 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, in which case the information scha_resource_get obtains through the handle could 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 information about the resource might have changed. This is a non-fatal error message—the function returns successfully. You can choose to ignore the message and accept the returned information, or you can close the current handle and open a new handle to access information about the resource.

  A single 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(3HA) – Sets the Status and Status_msg properties of a resource 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.

Resource Type Functions

Together these functions access information about a resource type registered with the RGM.

- scha_resourcetype_open(3HA), scha_resourcetype_get(3HA), scha_resourcetype_close(3HA)– 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 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, in which case the information scha_resourcetype_get obtains through the handle could 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 information about the resource type might have changed. This is a non-fatal error message—the function returns successfully. You can choose to ignore the message and accept the returned information, or you can close the current handle and open a new handle to access information about the resource type.

  A single 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 or restart a resource group with these functions.

- scha_resourcegroup_open(3HA), scha_resourcegroup_get(3HA), and scha_resourcegroup_close(3HA)–Together these functions access information on a resource group 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 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, in which case the information scha_resourcegroup_get obtains through the handle could 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 information about the resource group might have changed. This is a non-fatal error message—the function returns successfully. You can choose to ignore the message and accept the returned information, or you can close the current handle and open a new handle to access information about the resource group.

  A single 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), or scha_resourcegrooup_close(3HA)

- scha_control(3HA)– Requests the restart of a resource group under the control of the RGM or its relocation to a different node.

Cluster Functions

These functions access or return information about a cluster.

- `scha_cluster_open(3HA)`, `scha_cluster_get(3HA)`, and `scha_cluster_close(3HA)` – Together these functions access information about a cluster, such as node names, IDs, and states, cluster name, resource groups, and so on.

  A single 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)`, or `scha_cluster_close(3HA)`

  A cluster can change—through reconfiguration or administrative action—after `scha_cluster_open` returns the cluster's handle, in which case the information `scha_cluster_get` obtains through the handle could 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 information about the cluster might have changed. This is a non-fatal error message—the function returns successfully. You can choose to ignore the message and accept the returned information, or you can close the current handle and open a new handle to access information about the cluster.
- `scha_cluster_getlogfacility(3HA)` – Returns the number of the system log facility being used as the cluster log. Uses the returned value with the Solaris `syslog(3)` function to record events and status messages to the cluster log.
- `scha_cluster_getnodename(3HA)` – Returns the name of the cluster node on which the function is called.

Utility Function

This function converts an error code to an error message.

- `scha_strerror(3HA)` – Translates an error code—returned by one of the `scha_` functions—to the appropriate error message. Use this function with `logger(1)` to log messages to the system log (`syslog(3)`).

RMAPI Callback Methods

Callback methods are the key elements 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 a node boot or crash.

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

This section describes callback method arguments and exit codes and lists and describes callback methods in the following categories:

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

Note – Although this section provides brief descriptions of the callback methods, including the point at which the method is invoked and the expected effect on the resource, the `rt_callbacks(1HA)` man page is the definitive reference for the callback methods.

Method Arguments

The RGM invokes 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, with `-R` indicating the name of the resource instance, `-T` indicating the type of the resource, and `-G` indicating 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 (the property values of the resource and resource group on which it is called).

See `rt_callbacks(1HA)` for more information.

Exit Codes

All callback methods have the same exit codes defined to specify the effect of the method invocation on the resource state. The `scha_calls(3HA)` man page describes all these exit codes. The exit codes are:

- 0 – Method succeeded
- Any nonzero value – Method failed

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

Method implementations must output failure information using `syslog(3)` on each node. Output written to `stdout` or `stderr` is not guaranteed to be delivered to the user (though it currently is 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 – This required method is invoked on a cluster node when the resource group containing 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 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 time-out value for this method. For example, certain 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 file sets the time-out value for a resource's START method.

- STOP – This required method is invoked on a cluster node when the resource group containing 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 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 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 time-out value for this method. For example, certain resources, such as database daemons, take more time to stop, and thus require that the method have a longer time-out value.

  The way in which the RGM responds to failure of the STOP method depends on the setting of the Failover_mode property (see TABLE A-2).

  The STOP_TIMEOUT property in the resource type registration file sets the time-out value for a resource's STOP method.

- INIT – This optional method is invoked to perform a one-time initialization of the resource when the resource becomes managed—either when the resource group it is in 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 determined by the Init_nodes resource property.

- FINI – This optional method is invoked to clean up after the resource when the resource becomes unmanaged—either when the resource group it is in is switched to an unmanaged state or when the resource is deleted from a managed resource group. The method is called on nodes determined by the Init_nodes resource property.
- BOOT – This optional method, similar to INIT, is invoked to initialize the resource on nodes that join the cluster after the resource group containing the resource has already been put under the management of the RGM. The method is invoked on nodes determined by the Init_nodes resource property. The BOOT method is called when the node joins or rejoins the cluster as the result of being booted or rebooted.

Note – Failure of the INIT, FINI, or BOOT methods causes the syslog(3) function to generate an error message but does not otherwise affect RGM management of the resource.

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 hook into these administrative actions.

- VALIDATE – This optional method is called when a resource is created and when administrative action updates the properties of the resource or its containing resource group. This method is called on the set of cluster nodes indicated by the Init_nodes property of the resource's type. VALIDATE is called 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.

  VALIDATE is called only when resource or resource group properties are changed through administrative action, not when the RGM sets properties, or when a monitor sets the resource properties Status and Status_msg.
- UPDATE – This optional method is called to notify a running resource that properties have been changed. UPDATE is invoked after an administration 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 adjust the running resource accordingly.

  Failure of the UPDATE method causes the syslog(3) function to generate an error message but does not otherwise affect RGM management of the resource.

Net-Relative Callback Methods

Services that use network address resources might require that start or stop steps be done in a certain order relative to the network address configuration. The following optional callback methods, PRENET_START and POSTNET_STOP, enable a resource type implementation to do special startup and shutdown actions before and after a related network address is configured or unconfigured.

- PRENET_START – This optional method is called to do special startup actions before network addresses in the same resource group are configured.
- POSTNET_STOP – This optional method is called to do 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 on resource failure. 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 the node.

CHAPTER 4

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.

The information in this chapter includes:

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 application deaths exceed the failure count within the failure time window, the fault monitor fails the resource group containing the application resource over 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, it tries to correct the situation by restarting the DNS application locally. If this does not improve the situation and the probe repeatedly detects problems with the service, then the probe attempts to fail over the service to another node in the cluster.

Specifically, the sample data service includes:

- A resource type registration file that defines the static properties of the data service.
- A `START` callback method invoked by the RGM to start the `in.named` daemon when the resource group containing the HA-DNS data service is brought online.
- A `STOP` callback method invoked by the RGM to stop the `in.named` daemon when the resource group containing 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 started and stopped by `MONITOR_START` and `MONITOR_STOP` callback methods.
- A `VALIDATE` callback method invoked by the RGM to validate that the configuration directory for the service is accessible.
- An `UPDATE` callback method invoked by the RGM to restart the fault monitor when the system 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 defined in the RTR file.

The information in the RTR file is read by the RGM when the cluster administrator registers the HA-DNS data service.

RTR File Overview

The RTR file follows a well-defined format. It begins with resource type properties, followed by system-defined resource properties, and lastly with extension properties. See the `rt_reg(4)` man page and "Setting Resource and Resource Type Properties" on page 35 for more information.

This section describes the specific properties in the sample RTR file. It provides 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 206.

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.

```
#
# Copyright (c) 1998-2001 by Sun Microsystems, Inc.
# All rights reserved.
#
# Registration information for Domain Name Service (DNS)
#


#pragma ident“@(#)SUNW.sample1.100/05/24 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 will fail.

Note – The RGM treats property names as case insensitive. The convention for properties in Sun-supplied RTR files, with the exception of method names, is uppercase for the first letter of the name and lowercase for the rest of the name. Method names—as well as property attributes—contain all uppercase letters.

Some information about these properties follows.

- The resource type name can be specified by the Resource_type property alone (sample) or using the Vendor_id as a prefix with a "." separating it from the resource type (SUNW.sample).

  If you use Vendor_id, make it the stock symbol for the company 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 runs under Sun Cluster version 3.0.
- Failover = TRUE indicates that the data service cannot run in a resource group that can be online on multiple nodes at once.
- RT_basedir points to /opt/SUNWsample/bin as the directory path to complete relative paths, such as callback method paths.
- START, STOP, VALIDATE, and so on provide the paths to the respective callback method programs invoked by the RGM. These paths are relative to the directory specified by RT_basedir.
- Pkglist identifies SUNWsample as the package that contains the sample data service installation.

Resource type properties not specified in this RTR file, such as Single_instance, Init_nodes, and Installed_nodes, get their default value. See TABLE A-1 for a complete list of the resource type properties, including their default values.

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

Resource Properties in the Sample RTR File

By convention, you declare resource properties following the resource type properties in the RTR file. Resource properties include system-defined properties provided by Sun Cluster and extension properties you define. For either type you can specify a number of property attributes supplied by Sun Cluster, such as minimum, maximum, and default values.

System-Defined Properties in the RTR File

The following listing shows the system-defined properties in the 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 Sun Cluster provides the system-defined properties, you can set different default values using resource property attributes. See “Resource Property Attributes” on page 203 for a complete list of attributes available for applying to resource properties.

Note the following 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) for all timeouts. The sample RTR file changes the minimum 60 and changes the default to 300 seconds. A cluster administrator can accept this default value or change the value of the timeout to something else, (60 or greater). Sun Cluster has no maximum allowable value.
- The properties `Thorough_Probe_Interval`, `Retry_count`, and `Retry_interval`, have the `TUNABLE` attribute set to `ANYTIME`. This settings means 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 “`UPDATE` Method” on page 115.
- Resource properties are classified as
  - *required*—the cluster administrator must specify a value when creating a resource;
  - *optional*—if the 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 the developer needed to declare them in the RTR file. See the r_properties(5) man page or “Resource Properties” on page 192 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 the following 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 = 30;
        TUNABLE = ANYTIME;
        DESCRIPTION = “Time out value for the probe (seconds)”;
}
```

The sample RTR file defines two extension properties, `Confdir` and `Probe_timeout`. `Confdir` 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 services’s `PROBE` method is not a Sun Cluster callback method but a user-defined method. Therefore Sun Cluster doesn’t provide a `Probe_timeout` property for it. The developer has defined an extension property in the RTR file to allow 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 84.
- “Declaring the PMF_TAG and SYSLOG_TAG Variables” on page 85.
- “Parsing the Function Arguments” on page 86.
- “Generating Error Messages” on page 88.
- “Obtaining Property Information” on page 89.

Identifying the Command Interpreter and Exporting the Path

The first line of a shell script must identify the command interpreter. Each of the method scripts 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 rely 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 (with the exception of VALIDATE) use pmfadm(1M) to launch (or stop) either the data service or the monitor, passing the name of the resource. Each script defines a variable, PMF_TAG that can be passed to pmfadm to identify either the data service or the monitor.

Likewise each method script uses the logger(1) command to log messages with 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 group, and resource name 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. 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 invokes all of the callback methods—with the exception of `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 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, with `-R` indicating the name of the resource instance, `-T indicating the type of the resource, and -G` indicating the group into which the resource is configured. See the `rt_callbacks(1HA)` man page for more information on callback methods.

Note – The `VALIDATE` method is called with additional arguments (the property values of the resource and resource group on which it is called). See "Handling Property Updates" on page 109 for more information.

Each callback method needs a function to parse the arguments it 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 shows the `parse_args` function 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 identical to the one used by the other callback methods.

The parse function is called in MAIN as:

```
parse_args "$@"
```

Generating Error Messages

It is recommended that callback methods use the syslog facility 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 facility 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(1) command to log messages in the cluster log. For example, the START method in the sample data service retrieves the syslog facility 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 command for this purpose.

Two kinds of resource properties, system-defined properties and extension properties, are available. System-defined properties are predefined whereas 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 parameter. 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 launch it. The probe program resides in the base directory for the data service, which is pointed to by the RT_BASEDIR property, so 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 specify with the -O parameter that it is an extension property and supply the name of the property as the last parameter. 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 then uses awk(1) 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 on the cluster, and a STOP or POSTNET_STOP method to stop the application daemon on the cluster. The sample data service implements a START and a STOP method. See "Deciding on the START and STOP Methods to Use" on page 47 for information about when you might want to use PRENET_START and POSTNET_STOP instead.

START Method

The RGM invokes the START method on a cluster node when the resource group containing 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 that node.

This section describes the major pieces of the START method for the sample application. It does not describe functionality common to all callback methods, such as the parse_args function and obtaining the syslog facility, which are described in "Providing Common Functionality to All Methods" on page 84.

For the complete listing of the START method, see "START Method Code Listing" on page 209.

START Overview

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

This callback method uses the process monitor facility (pmfadm) to start the DNS daemon (in.named). If DNS crashes or fails to start, the PMF attempts to start it 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 launch 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 configuring the data service.

In the sample data service, the `START` method retrieves the location of the configuration directory using the `scha_resource_get(1HA)` command.

Note – Because `Confdir` is an extension property, `scha_resource_get` returns both the type and value. The `awk(1)` command retrieves just the value and places it 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 then 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 error status. See “`START` Exit Status” on page 94.

```
# 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 it is not present, START logs an error message and exits with 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 launch the application. The pmfadm command allows you to set the number of times to restart the application during a specified time frame. The RTR file contains two properties, Retry_count, which specifies the number of times to attempt restarting an application, and Retry_interval, which specifies the time period over which to do so.

The START method retrieves the values of Retry_count and Retry_interval using the scha_resource_get command and stores their values in shell variables. It then passes these values to pmfadm 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 available, particularly if other data services are dependent on it. One way to verify success is to probe the application to verify 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 perform crash recovery.

Note – Because the application resource, DNS, in the sample data service launches quickly, the sample data service does not poll to verify 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 has overridden the default and specified a different value). In this case, the RGM takes no action other than to set the state of the data service. User intervention is required to restart on the same node or fail over to a different node.

STOP Method

The STOP method is invoked on a cluster node when the resource group containing 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. It does not describe functionality common to all callback methods, such as the parse_args function and obtaining the syslog facility, which are described in “Providing Common Functionality to All Methods” on page 84.

For the complete listing of the STOP method, see “STOP Method Code Listing” on page 212.

STOP Overview

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 this.

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 this is to send a SIGKILL signal through pmfadm.

The STOP method in the sample data service takes both 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 process monitor facility (pmfadm) to stop it.

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 it 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 then allocates 80% of this time to stopping smoothly and 15% to stopping abruptly (5% is reserved), as shown in the following sample.

```
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 it is, 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% of the timeout value has expired STOP sends a SIGKILL signal. If this signal also fails to terminate the process within 15% of the timeout value, the method logs an error message and exits with 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 UNSUCCESFUL"

        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 have dependencies 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 it has the default value NONE (unless the cluster administrator has overridden the default and specified a different value). In this case, the RGM takes no action other than to set the state of the data service to Stop_failed. User intervention is required 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:

- dns_probe, a user-defined program that uses nslookup(1M) to verify that the DNS resource 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 launches 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 the data service over to a new node.

Probe Program

The dns_probe program implements a continuously running process that verifies the DNS resource controlled by the sample data service is running. The dns_probe is launched by the dns_monitor_start method, which is automatically invoked 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 invokes before bringing the sample data service offline.

This section describes the major pieces of the PROBE method for the sample application. It does not describe functionality common to all callback methods, such as the parse_args function and obtaining the syslog facility, which are described in "Providing Common Functionality to All Methods" on page 84.

For the complete listing of the PROBE method, see "PROBE Program Code Listing" on page 215.

Probe Overview

The probe runs in an infinite loop. It uses nslookup(1M) to verify that the proper 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 then 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 needs the values of the following properties:

- Thorough_probe_interval – To set the period during which the probe sleeps
- Probe_timeout – to enforce the time-out 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 containing the PROBE program and the gettime utility

The scha_resource_get command 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, scha_resource_get returns the value only. For extension properties, such as Probe_timeout, scha_resource_get returns the type and value. Use the awk(1) command to obtain the value only.

Checking the Reliability of the Service

The probe itself is an infinite `while` loop of `nslookup(1M)` 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 itself:

- Sets the sleep interval for the probe
- Uses `hatimerun(1M)` to launch `nslookup` passing the `Probe_timeout` value and identifying 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
```

Evaluating Restart Versus Failover

If the *probefail* variable is something other than 0 (success), it means the nslookup command timed out or that 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, then 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. It implements the following conditional code (see the code listin g for decide_restart_or_failover in "PROBE Program Code Listing" on page 215).

- 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, then fail over to another node. If the fail over 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 about this function:

- The gettime utility is used to track the time between restarts. This is a C program residing in the (Rt_basedir) directory.

- The Retry_count and Retry_interval system-defined resource properties determine the number of restart attempts and the interval over which to count. These properties default to 2 attempts in a period of 5 minutes (300 seconds) in the RTR file, though the cluster administrator could change them.
- 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 103, for information about this function.
- The scha_control API command, with the GIVEOVER option, brings the resource group containing 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 does the following.

- It determines if the data service is still registered under PMF. If the service is still registered, the function:
  - Obtains the STOP method name and the Stop_timeout value for the data service.
  - Uses hatimerun to launch 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 launch the START method for the data service, passing the Start_timeout value.
- If the data service is no longer registered under PMF, the implication is that the data service has exceeded the maximum number of allowable retries under PMF, so the scha_control function is called with the GIVEOVER option to fail the data service over 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_NAME`
        START_METHOD=`scha_resource_get -O START \
                -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME`
        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 have failed and the attempt to fail over to a different node has failed as well. It logs the message, "Failover attempt failed".

MONITOR_START Method

The RGM calls the MONITOR_START method to launch 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 common to all callback methods, such as the parse_args function and obtaining the syslog facility, which are described in "Providing Common Functionality to All Methods" on page 84.

For the complete listing of the MONITOR_START method, see "MONITOR_START Method Code Listing" on page 222.

MONITOR_START Overview

This method uses the process monitor facility (pmfadm) to launch 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 launches the probe using the infinite retries option of pmfadm (-n -1, -t -1), which means if the probe fails to start, 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
```

MONITOR_STOP Method

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 common to all callback methods, such as the parse_args function and obtaining the syslog facility, which are described in “Providing Common Functionality to All Methods” on page 84.

For the complete listing of the MONITOR_STOP method, see “MONITOR_STOP Method Code Listing” on page 224.

MONITOR_STOP Overview

This method uses the process monitor facility (pmfadm) to see if 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 idempotency of the method.

```
# 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
```

Caution – Be certain to use the KILL signal with pmfadm to stop the probe and not a maskable signal such as TERM. Otherwise the MONITOR_STOP method can hang indefinitely and eventually time out. The reason for this problem 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 maskable signal, it hangs waiting for scha_control to complete and scha_control hangs waiting for MONITOR_STOP to complete.

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.

MONITOR_CHECK Method

The RGM calls the MONITOR_CHECK method whenever the PROBE method attempts to fail the resource group containing the data service over 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 common to all callback methods, such as the parse_args function and obtaining the syslog facility, which are described in "Providing Common Functionality to All Methods" on page 84.

For the complete listing of the MONITOR_CHECK method, see "MONITOR_CHECK Method Code Listing" on page 226.

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

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

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

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

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

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

See the "VALIDATE Method" on page 109 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 updating of properties by a cluster administrator.

VALIDATE Method

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 group properties are changed through administrative action, 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 the data service over to a new node.

VALIDATE Overview

The RGM calls VALIDATE with additional arguments to those passed to other methods, including the properties and values 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 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 the RGM passes to VALIDATE, the parse_args function retrieves and saves its value. VALIDATE then 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 some data.

If the parse_args function cannot retrieve the value of Confdir from the command-line arguments 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. Then it performs the same checks to verify that the configuration directory is accessible and contains a non-empty named.conf file.

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

VALIDATE Method Parsing Function

The RGM passes the VALIDATE method a different set of parameters than the other callback methods so VALIDATE requires a different function for parsing arguments than the other methods. See the rt_callbacks(1HA) man page for more information on the parameters passed to VALIDATE and the other callback methods. The following 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 passed by the RGM.

The `r` flag (indicating a system-defined property), `g` flag (indicating a resource group property), and the `c` flag (indicating that the validation is occurring during creation of the resource) 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 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 then calls parse_args to parse the arguments passed by the RGM.

```
parse_args “$@”
```

VALIDATE then checks if VALIDATE is being called as the result of an update of properties and if the Confdir extension property was on the command line. VALIDATE then 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 which case it retrieves the existing value of Confdir 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 then uses the value of *CONFDIR* to verify that the directory is accessible. If it 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 it is not, 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 indicating 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 indicating why is sent to the cluster administrator.

UPDATE Method

The RGM calls the UPDATE method to notify a running resource that its properties have been changed. The RGM invokes 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.

UPDATE Overview

The UPDATE method doesn't update properties—that is done by the RGM. Rather, it notifies running processes that an update has occurred. The only process in the sample data service affected by a property update is the fault monitor, so it is this process the UPDATE method stops and restarts.

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

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 administrative user. If the monitor cannot be stopped, UPDATE exits with failure status and sends an error message to the administrative user.

```
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 resides 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 then 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 administrative user. If pmfadm cannot launch 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 facility.

CHAPTER 5

Data Service Development Library (DSDL)

This chapter provides an overview of the application programming interfaces constituting the Data Service Development Library, or DSDL. The DSDL is implemented the `libdsdev.so` library and is included in the Sun Cluster package.

The following information is in this chapter.

DSDL Overview

The DSDL API is layered on top of the RMAPI. As such, it 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 intrinsic to your application, and avoid spending a large amount of time on 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 properies by:

- 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[]`) 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.

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 `scds_initialize` function also initializes the logging environment and validates fault monitor probe settings.

The DSDL provides sets of functions to retrieve resource, resource type, 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 it retrieves.
- Functions do not return errors as the values have been pre-computed 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 is expected to perform the actions required to start a data service on a cluster node. Typically, this includes retrieving the resource properties, locating application-specific executables and configuration files, and launching the application with the appropriate 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 launch.

A START method can call scds_pmf_start to launch an application under control of the Process Monitor Facility (PMF). 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 "The xfnts_start Method" on page 139 for an example of a START method implemented with the DSDL.

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

To avoid putting the resource in 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. It first attempts to stop the resource using SIGTERM signal, and if this fails, uses a SIGKILL signal. See scds_pmf_stop(3HA) for details.

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 launches the fault monitor, under the control of PMF, when the resource starts on a node. The fault monitor runs in 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 determined by PMF during this interval lead to a restart of the resource.
- The probe itself returns a value indicating 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 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, failover 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 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 connection to a service, read and write data to the service, and then 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 “The svc_probe Function” on page 151 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 per resource type implementation per cluster node. A file named */var/cluster/rgm/rt/<rtname>/loglevel* containing only an integer between 1-9 is read by all resource type callback methods. The DSDL routine `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 log no debugging messages.

The `scds_syslog_debug()` function uses the facility returned by the `scha_cluster_getlogfacility(3HA)` function at a priority of `LOG_DEBUG`. You can configure these debug messages in `/etc/syslog.conf`.

You can turn some debugging messages into informational messages for regular operation of the resource type (perhaps at `LOG_INFO` priority) using the `scds_syslog` utility. If you look at the sample DSDL application in Chapter 7 you will notice that it makes liberal use of `scds_syslog_debug` and `scds_syslog` functions.

CHAPTER **6**

Designing Resource Types

This chapter explains the typical usage of the DSDL in designing and implementing resource types. It focuses on designing the resource type to validate the resource configuration, and to start, stop, and monitor the resource. It then describes how to implement the resource type callback methods using the DSDL. Please also refer to the `rt_callbacks(1HA)` man page.

A resource type developer needs access to the resource's property settings to complete these tasks. The DSDL utility, `scds_initialize()`, gives the programmer a uniform way to access the 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()` functions.

The RTR File

The Resource Type Registration (RTR) file is a very important component of a resource type. It specifies the details about the resource type to Sun Cluster. This includes information such as what properties are needed by the implementation, the data types of those properties, the default values of those properties, the file system path for the callback methods for the resource type implementation, and various settings for the system defined properties.

The sample RTR file shipped with the DSDL should suffice for most resource type implementations after editing some basic elements such as the resource type name and the pathname of the resource type callback methods. If a new property is needed to implement the resource type, programmers can declare it as an extension property in the Resource Type Registration (RTR) file of the resource type implementation, and then access the new property using the DSDL `scds_get_ext_property()` utility.

The VALIDATE Method

The VALIDATE method of a resource type implementation is called by the RGM in two scenarios: 1) when a new resource of the resource type is being created, and 2) when 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 (creation) or -u (update) 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 containing 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 rt_reg(4)). 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 value(s) of the properties being updated (in the case of update callback). 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 DSDL function scds_initialize() takes care of these situations in the following manner:

- In the case of resource creation, it parses the proposed resource properties, as passed on the command line. The proposed values of resource properties are thus available to the resource type developer as if the the resource were already created in the system.
- In the case of resource or resource group update, the proposed values of the properties being updated by the administrator are read in from the command line, and the remaining properties (whose values are not being updated) are read in from Sun Cluster using the Resource Management API. A resource type developer using the DSDL need not concern himself with all these housekeeping tasks. The validation of a resource can be done as if all the properties of the resource were available to the developer.

Suppose the function that implements the validation of a resource's properties is called `svc_validate()` which uses the `scds_get_*()` family of functions to look at the property it is interested in validating. 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 the validation function should also log the reason for the failure of the validation of resource. Leaving out that detail (see the next chapter for a more realistic treatment of a validation routine), a simple example `svc_validate()` routine can then be implemented as:

```
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 */
}
```

The resource type developer thus has to concern himself with only the implementation of the `svc_validate()` routine. A typical example for a resource type implementation could be to ensure that an application configuration file named app.conf exists under the `Confdir_list` property. That can be conveniently implemented by a `stat()` system call on the appropriate pathname derived from the `Confdir_list` property.

The 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 is expected to perform the actions needed to start up a data service resource on the cluster node. Typically this involves retrieving the resource properties, locating the application specific executables and/or configuration files, and launching the application with appropriate 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 routine `svc_start()`. Another routine, `svc_wait()`, can be called to verify that the application actually starts. The simplified code for the START method becomes:

```
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 global file system needed by the resource may currently not be available on this cluster node. In this case, it is futile to even 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 above assumes `svc_validate()` is sufficiently conservative (so that it checks only for resources on the cluster node that are absolutely needed by the application) or else the resource might fail to start up on all cluster nodes and thus land in START_FAILED state. See scswitch(1M) and the *Sun Cluster 3.0 Data Services Installation and Configuration Guide* for an explanation of this resource state.

The svc_start() routine must return 0 for a successful startup of the resource on the node. If the startup routine encountered a problem, it must return non-zero. Upon failure of this routine, the RGM attempts to start the resource on a different cluster node.

To leverage the DSDL as much as possible, the svc_start() routine can use the scds_pmf_start() utility to start the application under the Process Management Facility (PMF). This utility also leverages the failure callback action feature of PMF (see the -a action flag in pmfadm(1M)) to implement process failure detection.

The 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 demands that

- 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 non-zero) on a cluster node, the resource being stopped would end up in the STOP_FAILED state. Depending upon the Failover_mode setting on the resource, this may lead to a hard rebooting of the cluster node by the RGM. Thus it is important to design the STOP method so that it tries very hard to really stop the application, even by a hard and abrupt killing of the application (for example, using SIGKILL) if the application otherwise fails to terminate. It should also make sure that it does so in a timely fashion, because the framework treats expiry of Stop_timeout as a stop failure, and puts the resource in STOP_FAILED state.

The DSDL utility scds_pmf_stop() should suffice for most applications as it first attempts to softly (via SIGTERM) stop the application (it assumes that it was started under PMF via scds_pmf_start()) followed by a delivering a SIGKILL to the process. See Section “PMF Functions” on page 9-185 for details about this utility.

Following the model of the code we have been using so far, assuming that the application specific routine to stop the application is called svc_stop() (whether the implementation of svc_stop() uses the scds_pmf_stop() is besides the point here, and would depend upon whether or not the application was started under PMF via the START method)) the STOP method can be implemented as

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

The svc_validate() method is not used in the implementation of the STOP method, because even if the system currently has a problem, the STOP method should attempt to STOP the application on this node.

The 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 being managed by the resource. Resource type implementations typically implement a fault monitor as a separate daemon which runs in the background. The MONITOR_START callback method is used to launch this daemon with the appropriate 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 relative pathname (relative to the RT_basedir for the location of the resource type callback method implementations) of the monitor daemon program. It uses the Monitor_retry_interval and Monitor_retry_count extension properties managed by the DSDL to prevent unlimited restarts of the daemon. It 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. It allows the monitor daemon implementation itself to leverage the scds_initialize() utility to set up its own environment. The main effort is in designing the monitor daemon itself.

The MONITOR_STOP Method

The RGM calls the MONITOR_STOP method to stop the fault monitor daemon that was started via 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 robust like 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.

The MONITOR_CHECK Method

The MONITOR_CHECK callback method on a resource is invoked on a node for the specified resource to ascertain whether the cluster node is capable of mastering the resource (that is, can the application(s) being managed by the resource be run successfully on the node?). Typically this involves making sure that all the system resources needed by the application are indeed available on the cluster node. As discussed in Section "The VALIDATE Method" on page 6-124, the routine svc_validate() implemented by the developer is intended to ascertain at least that.

Depending upon the specific application being managed by the resource type implementation, the MONITOR_CHECK method can be written to do some additional tasks. For developers using the DSDL it is recommended that the MONITOR_CHECK method leverage the svc_validate() routine written for the purpose of implementing application specific validation of resource properties.

The UPDATE Method

The RGM calls the UPDATE method of a resource type implementation to apply any changes that were made by the system 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.

A resource type developer needs to cautiously decide which properties are to be dynamically updatable and mark those with the TUNABLE=ANYTIME setting in the RTR file. Typically any property used by the fault monitor daemon of a resource type implementation could be made dynamically updatable provided the UPDATE method implementation at least restarts the monitor daemon.

Possible candidates are

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

These properties affect the way a fault monitor daemon does health checking of the service, how often it does it, what history interval it uses to keep track of the errors, and what are the restart thresholds set on it by PMF. To implement updates of these properties the utility scds_pmf_restart() is provided in the DSDL.

If a resource type developer identifies the need to make a resource property dynamically updatable where modification of that property might have an effect on the running application, the resource type developer needs to implement the appropriate actions so that the updates to that property are correctly applied to any running instances of the application. Currently there is no way to facilitate this via the DSDL. UPDATE is not passed the modified properties on the command line (as is VALIDATE).

The INIT, FINI and BOOT Methods

These are *one time action* methods as defined by the Resource Management API specifications. The sample implementation 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 a resource type developer have a need for 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 which *undoes* the action of the `INIT` or `BOOT` methods.

Designing the Fault Monitor Daemon

Resource type implementations using the DSDL typically have a fault monitor daemon with the following responsibilities.

- Periodically monitoring the health of the application being managed. This particular aspect of a monitor daemon is heavily application dependent and could vary widely from resource type to resource type. The DSDL has some built in utility functions to perform health checks for simple TCP based services. Applications implementing ASCII based protocols such as HTTP, NNTP, IMAP, and POP3 can be implemented using these utilities.
- Keeping track of the problems encountered by the application using the resource properties `Retry_interval` and `Retry_count`. Upon complete failures of the application, deciding whether the PMF action script should restart the service or whether the application failures have accumulated so rapidly that a failover could be considered. The DSDL utilities `scds_fm_action()` and `scds_fm_sleep()` are intended to aid programmers implementing this mechanism.
- Taking appropriate actions (typically either restarting the application or attempting a failover of the containing resource group). The DSDL utility `scds_fm_action()` implements such an algorithm. It computes the current accumulation of probe failures in the past Retry_interval seconds for this purpose.
- Updating the resource state so that application health state is available to the `scstat(1m)` command as well as to the cluster management GUI.

The DSDL utilities are designed so the main loop of the fault monitor daemon can be represented by the following pseudo code.

For fault monitors implemented using the DSDL,

- The detection of application process death by `scds_fm_sleep()` is fairly rapid because the process death notification via PMF is asynchronous. Contrast that with a case where a fault monitor wakes up every so often to check on service health and finds the application dead. The fault detection time is reduced significantly, thereby increasing the availability of the service.
- If the RGM rejects the attempt to fail over the service via the `scha_control(3HA)` API, `scds_fm_action()` *resets* (forgets) its current failure history. The reason is that the failure history is already above `Retry_count`, and if the monitor daemon wakes up in the next iteration and is unable to successfully complete its health check of the daemon, it would again attempt to invoke the `scha_control()` call, which would probably still be rejected, as the situation which 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, via application restart) in the next iteration.

- scds_fm_action() does *not* reset application failure history in case of restart failures, as one would typically like to try scha_control() soon if the situation doesn't correct itself.
- The utility scds_fm_action() updates the resource status to SCHA_RSSTATUS_OK, SCHA_RSSTATUS_DEGRADED or SCHA_RSSTATUS_FAULTED depending upon the failure history. This status is thus available to cluster system management.

In most cases, the application specific health check action can be implemented in a separate stand-alone utility [for example, svc_probe()] and integrated with this 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 7

Sample DSDL Resource Type Implementation

This chapter describes a sample resource type, SUNW.xfnts, implemented with the DSDL. The data service is written in C. The underlying application is the X Font Server, a TCP/IP-based service.

The information in this chapter includes.

About the X Font Server

The X Font Server is a simple 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 `/usr/openwin/bin/xfs`. The daemon is normally started from `inetd`, however, for the current sample, assume that the appropriate entry in the `/etc/inetd.conf` file has been disabled (for example, by the *fsadmin -d* command) so the daemon is under sole control of Sun Cluster.

The next section describes the X Font Server configuration file.

The 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 available to the daemon for serving. The cluster administrator can locate the font directories on the global file system (to optimize the use of the X Font Server on Sun Cluster by maintaining a single copy of the font's database on the system). If so, the administrator must edit `fontserver.cfg` to reflect the new paths for the font directories.

For ease of configuration, the administrator can also place the configuration file itself on the global file system. The `xfs` daemon provides command line arguments to override the default, built-in location of this file. The `SUNW.xfnts` resource type uses the following command to start the daemon under control of Sun Cluster.

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

In the `SUNW.xfnts` resource type implementation, you can use the `Confdir_list` property to manage the location of the `fontserver.cfg` configuration file.

The 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 on the xfs command line enables the system 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 support the use of the -port option on the xfs command line. 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 isn't required to specify a port number—the port defaults to 7100/tcp—but does have the option of specifying a different port if they wish when configuring the resource type, by specifying a different value for the Port_list property.

Naming Conventions

You can identify the various pieces of the sample code by keeping the following conventions in mind.

- RMAPIunctions begin with scha_.
- DSDL functions begin with scds_.
- Callback methods begin with xfnts_.
- User-written functions begin with svc_.

The 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 35.

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 at the time of resource creation. This value cannot be changed later because tunability is limited to AT_CREATION.

The Port_list property identifies the port on which the server daemon listens, as follows.

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

Because the property declares a default value, the cluster administrator has a choice of specifying a new value or accepting the default at the time of resource creation. This value cannot be changed later because tunability is limited to AT_CREATION.

The scds_initialize Call

The DSDL requires that each callback method call the scds_initialize(3HA) function at the beginning of the method. This function does the following.

- 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 do any additional processing of the 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 allocated by scds_initialize.

The xfnts_start Method

The RGM invokes the START method on a cluster node when the resource group containing 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 PMF. 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 *house-keeping* functions (see "The scds_initialize Call" on page 139 and the scds_initialize(3HA) man page for details).

Validating the Service Before Starting

Before it attempts to start the X Font Server, the `xfnts_start` method calls `svc_validate` to verify that a proper configuration is in place to support the `xfs` daemon (see "The `xfnts_validate` Method" on page 154 for details), as follows.

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

Starting the Service

The `xfnts_start` method calls the `svc_start` method, defined in `xfnts.c` to start the `xfs` daemon. This section describes `svc_start`.

The command to launch 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*. When the cluster administrator configures the data service, he provides specific values for these properties.

The `xfnts_start` method declares these properties as string arrays and obtains the values the administrator sets using the `scds_get_ext_confdir_list`(3HA) and `scds_get_port_list`(3HA) functions, as follows.

```
    scha_str_array_t *confdirs;
    scds_port_list_t *portlist;
    scha_err_terr;

    /* 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 as follows.

```
     /* 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 he output is redirected to dev/null to suppress messages generated by the daemon.

The xfnts_start method passes the xfs command line to scds_pmf_start to start the data service under control of PMF, as follows.

```
    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 about the call to scds_pmf_start.

- The SCDS_PMF_TYPE_SVC parameter 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 parameter identifies this as a single-instance resource.
- The cmd parameter is the command line generated previously.
- The final parameter, -1, specifies the child monitoring level. The -1 specifies that PMF monitor all children as well as the original process.

Before returning, svc_pmf_start frees the memory allocated for the portlist structure, as follows.

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

Returning From svc_start

Even when svc_start returns successfully, it is possible the underlying application 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 up. The svc_start method calls svc_wait, which is defined in xfnts.c, to verify the application is running, as follows.

```
    /* 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 method calls scds_get_netaddr_list(3HA) to obtain the network-address resources needed to probe the application, as follows.

```
    /* 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);
    }
```

Then `svc_wait` obtains the `start_timeout` and `stop_timeout` values, as follows.

```
    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 up, `svc_wait` calls `scds_svc_wait` and passes a timeout value equivalent to three percent of the `start_timeout` value. Then `svc_wait` calls `svc_probe` to verify that the application has started. The `svc_probe` method makes a simple socket connection to the server on the specified port. If fails to connect to the port, `svc_probe` returns a value of 100, indicating total failure. If the connection goes through but the disconnect to the port fails, then `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 specified by the `Retry_interval` property of the resource, the `scds_svc_wait` method returns failure. In this case, the `svc_start` method 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 allocated by `scds_initialize`. See "The `scds_initialize` Call" on page 139 and the `scds_close`(3HA) man page for details.

The `xfnts_stop` Method

Because the `xfnts_start` method uses `scds_pmf_start` to start the service under 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 *house-keeping* functions (see "The `scds_initialize` Call" on page 139 and the `scds_initialize`(3HA) man page for details.

The `xfnts_stop` method calls the `svc_stop` method, which is defined in `xfnts.c` 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 about the call in `svc_stop` to the `scds_pmf_stop` function.

- `SCDS_PMF_TYPE_SVC` parameter 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` parameter identifies the signal.
- The `SIGTERM` parameter 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 allocated by `scds_initialize`. See “The `scds_initialize` Call” on page 139 and the `scds_close`(3HA) man page for details.

The `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 PMF.

Note – The first call in `xfnts_monitor_start` is to `scds_initialize`, which performs some necessary *house-keeping* functions (see “The `scds_initialize` Call” on page 139 and the `scds_initialize`(3HA) man page for details.

The `xfnts_monitor_start` method calls the `mon_start` method, which is defined in `xfnts.c` 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 about the call in `svc_mon_start` to the `scds_pmf_start` function.

- `SCDS_PMF_TYPE_MON` parameter 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` parameter identifies this as a single-instance resource.
- The `xfnts_probe` parameter identifies the monitor daemon to start. The assumption is that the monitor daemon is in the same directory as the other callback programs.
- The final parameter, `0`, specifies the child monitoring level—in this case, monitor the monitor daemon only.

Note – Before it exits, the `xfnts_monitor_start` method calls `scds_close` to reclaim resources allocated by `scds_initialize`. See “The `scds_initialize` Call” on page 139 and the `scds_close`(3HA) man page for details.

The `xfnts_monitor_stop` Method

Because the `xfnts_monitor_start` method uses `scds_pmf_start` to start the monitor daemon under 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 *house-keeping* functions (see “The `scds_initialize` Call” on page 139 and the `scds_initialize`(3HA) man page for details.

The `xfnts_monitor_stop` method calls the `mon_stop` method, which is defined in `xfnts.c` 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 about the call in `svc_mon_stop` to the `scds_pmf_stop` function.

- `SCDS_PMF_TYPE_MON` parameter 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` parameter identifies this as a single-instance resource.
- The `SIGKILL` parameter 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 allocated by `scds_initialize`. See “The `scds_initialize` Call” on page 139 and the `scds_close`(3HA) man page for details.

The xfnts_monitor_check Method

The RGM calls the MONITOR_CHECK method whenever the fault monitor attempts to fail the resource group containing the resource over to another node. The xfnts_monitor_check method calls the svc_validate method to verify that a proper configuration is in place to support the xfs daemon (see "The xfnts_validate Method" on page 154 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);
}
```

The 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 PMF. The xfnts_monitor_stop method stops the fault monitor.

The SUNW.xfnts fault monitor does the following.

- It periodically monitors the health of the xfs server daemon using utilities specifically designed to check simple TCP-based services, such as xfs.

- It tracks problems the application encounters within a time window (using the Retry_count and Retry_interval properties) and decides whether to restart or failover the data service in case of complete application failure. The scds_fm_action and scds_fm_sleep functions provide built-in support for this tracking and decision mechanism.
- It implements the failover or restart decision using scds_fm_action.
- It updates the resource state and makes it available to administrative tools and GUIs.

The xfonts_probe Main Loop

The xfonts_probe method implements a loop. Before implementing the loop, xfonts_probe

- 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.

```
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 loop as follows.

```
    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, failover the resource group, or do nothing.

The svc_probe Function

The svc_probe function makes a simple socket connection to the specified port by calling scds_fm_tcp_connect. If the connection fails, svc_probe returns a value of 100 indicating a complete failure. If the connection succeeds, but the disconnect fails, svc_probe returns a value of 50 indicating a partial failure. If the connection and disconnection both succeed, svc_probe returns a value of 0, indicating 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_tt1, t2;
    int sock;
    chartestcmd[2048];
    int time_used, time_remaining;
    time_tconnect_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 success (0), partial failure (50), or complete failure (100) value. 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, failover 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.

The xfnts_validate Method

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 group properties are changed through administrative action, 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 the data service over to a new node.

The RGM calls VALIDATE with additional arguments to those passed to other methods, including the properties and values being updated. The call to scds_initialize at the beginning of xfnts_validate parses all the arguments the RGM passes to xfnts_validate and stores the information in the scds_handle parameter. The subroutines that xfnts_validate calls make use of this information.

The xfnts_validate method calls svc_validate, which verifies the following.

- 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 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 containing 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, as follows.

```
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 allocated by `scds_initialize`. See "The `scds_initialize` Call" on page 139 and the `scds_close`(3HA) man page for details.

The `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 parameter 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 there is only one instance of the fault monitor.

CHAPTER **8**

SunPlex Agent Builder

This chapter describes SunPlex™ Agent Builder, a tool that automates creation of resource types, or data services, to be run under the Resource Group Manager (RGM). A resource type essentially is a wrapper around an application to enable the application to run in a clustered environment under control of the RGM.

Agent Builder provides a screen-based interface for entering simple information about your application and the kind of resource type you want to create. Based on the information you enter, Agent Builder generates the following:

- A set of source files—C or Korn shell (ksh)—for a failover or scalable resource type, corresponding to the resource type's method callbacks
- A customized Resource Type Registration (RTR) file
- Customized utility scripts for starting, stopping, and removing an instance (resource) of the resource type, as well as customized man pages documenting how to use each of these files
- A Solaris package that includes the, binaries (in the case of C source), an RTR file, and the utility scripts

Agent Builder supports network-aware applications—applications that use the network to communicate with clients—as well as non network-aware (or stand-alone) applications. Agent Builder also allows 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 (see "Creating Resource Types With Multiple Independent Process Trees" on page 168.

Using Agent Builder

This section describes how to use Agent Builder, including tasks you must complete before you can use Agent Builder. It also explains ways you can leverage Agent Builder after you have generated your resource type code.

Analyzing The Application

Before using Agent Builder you must determine if your application meets the criteria to be made highly available or scalable. Agent Builder cannot perform this analysis, which is based solely on the runtime characteristics of the application. "Analyzing the Application for Suitability" on page 30 provides detailed information on this topic.

Agent Builder may not always be able to create a complete resource type for your application, though 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, such as code to add validation checks for additional properties or to tune 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 sort of flexibility.

Agent Builder places comments at certain points in the generated source code where you can add your own specific 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 generated by Agent Builder, you can leverage 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—it is included in the `SUNWscdev` package, which is installed by default as part of a standard Sun Cluster software installation (see the **Sun Cluster 3.0 12/01 Software Installation Guide**). Before you use Agent Builder, verify the following:

- Java is included in your `$PATH` variable. Agent Builder depends on Java (Java Development Kit version 1.2.2_05a or higher) and if Java is not in your `$PATH`, `scdsbuilder` returns with an error message.
- You have installed the “Developer System Support” software group of Solaris 8 or higher.
- 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 164.

Note – You can use a different compiler with Agent Builder than the standard `cc` compiler. One way to do this is to create a symbolic link in `$PATH` from `cc` to a different compiler, such as `gcc`. Another way is to change the compiler specification in the makefile (currently, `CC=cc`) to the complete path for a different compiler. For example, in the makefile generated by Agent Builder, change `CC=cc` to `CC=`*pathname*`/gcc`. In this case you cannot run Agent Builder directly but must use the `make` and `make pkg` commands to generate data service code and a package.

Launching Agent Builder

Launch Agent Builder with the command, `scdsbuilder`(1HA):

```
% /usr/cluster/bin/scdsbuilder
```

The initial Agent Builder screen, as shown in FIGURE 8-1, appears.

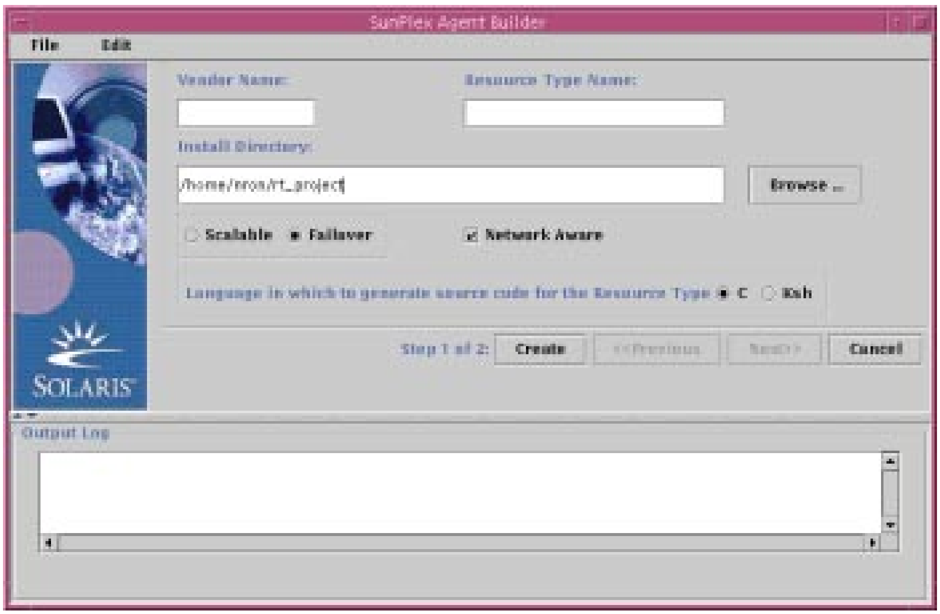


FIGURE 8-1 Initial Screen

Note – You can access Agent Builder through a command-line interface (see "Using the Command Line Version of Agent Builder" on page 163) if the GUI version is not accessible.

Agent Builder provides two screens to guide you through the process of creating a new resource type:

1. **Create—**On this screen you provide basic information about the resource type to create, such as its name and the install directory for the generated files. You also identify 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), and the type of code (C or ksh) to generate. You must complete the information in this screen, and select **Create** to generate the corresponding output, before you can bring up the Configure screen.

2. **Configure—**On this screen, you are required to provide a command to start the application. Optionally, you can provide commands to stop and probe the application. If you do not specify these 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 166). This screen also enables you to change the timeout values for each of these three commands.

Note – If you launch Agent Builder from the install directory for an existing resource type, Agent Builder initializes the Create and Configure screens to the values of the existing resource type.

See "Navigation" on page 176 if you have questions about how to use any of the buttons or menu commands on either of the Agent Builder screens.

Using the Command Line Version of Agent Builder

The command-line version of Agent Builder has the same two-step process as the graphical user interface. However, instead of entering information in the GUI, you pass parameters to two commands, `scdscreate`(1HA) and `scdsconfigure`(1HA). See the man pages for these commands for more information.

Using the Create Screen

The first step in creating a resource type is to fill out the Create screen, which appears when you launch Agent Builder. FIGURE 8-2 shows the Create screen after you enter information in the fields.

FIGURE 8-2 Create Screen

The Create screen contains the following fields, radio buttons, and check box:

- **Vendor Name —** A name to identify the vendor of the resource type. Typically, you specify the stock symbol of the vendor, but any name that uniquely identifies the vendor is valid. Use alphanumeric characters only.
- **Resource Type Name —** The name of the resource type. Use alphanumeric characters only.

Note – Together, the vendor name and resource type name make up the full name of the resource type. The full name must not exceed nine characters.

- **Install Directory —** The directory under which Agent Builder creates a directory structure to contain all the files created for the target resource type. You can create only one resource type in any one Install directory. Agent Builder initializes this field to the path of the directory from which you launched Agent Builder, though you can type a different name or use the **Browse** button to locate a different directory.

Under the install directory, Agent Builder creates a subdirectory with the resource-type name. For example, if `SUNW` is the vendor name and `ftp` is the resource type name, then 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” on page 171).

- **Failover or Scalable —** Specify whether the target resource type will be 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. Check the box to specify network aware; leave it blank to specify non-network aware. Ksh code requires that the application be network aware. Therefore, Agent Builder checks this box, and grays it out if you check the **Ksh** button.
- **C or 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 ksh generated code and then reuse the same information to create C generated code (see “Cloning an Existing Resource Type” on page 169).

Note – If the `cc` compiler is not in your `$PATH`, Agent Builder grays out the C option button and puts a check in the Ksh button. To specify a different compiler, see the note at the end of “Installing and Configuring Agent Builder” on page 161.

After you have entered the required information, click the **Create** button. The Output Log at the bottom of the screen shows the actions that Agent Builder is taking. You can use the **Save Output Log** command in the Edit menu to save the information in the output log.

When finished, Agent Builder displays either a success message or a warning message that it was unable to complete this step, and you should check the output log for details.

If Agent Builder completes successfully, you can Click the **Next** button to bring up the Configure screen, which 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 you have entered or the work that Agent Builder has completed (see “Reusing Completed Work” on page 169).

Using the Configure Screen

The Configure screen, shown in FIGURE 8-3, appears after Agent Builder finishes creating the resource type and you select the **Next** button on the Create screen. You cannot access the Configure screen before the resource type has been created.

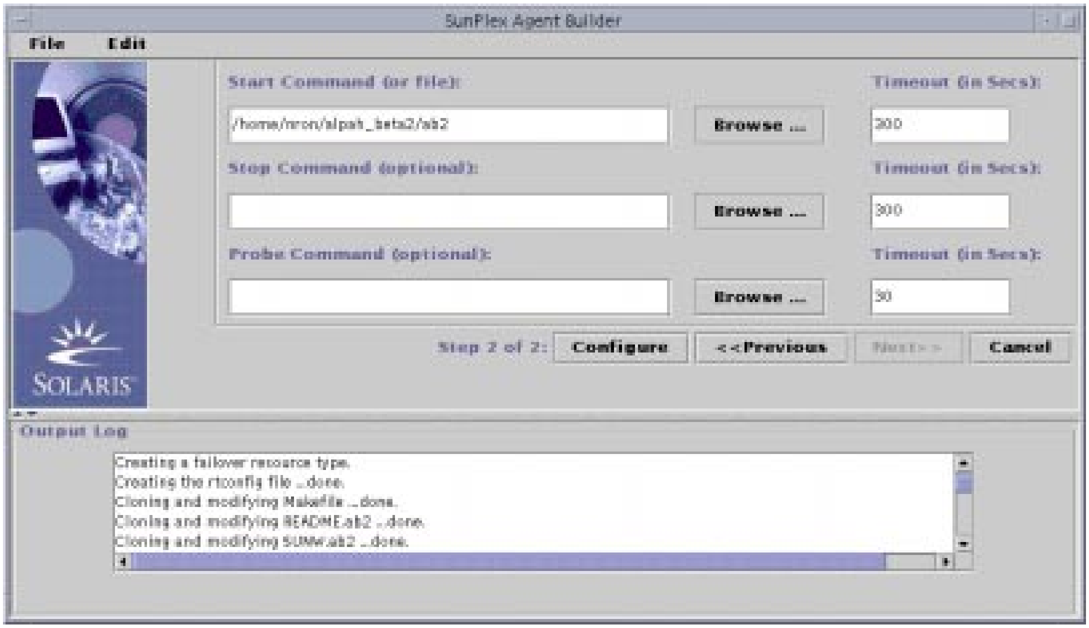


FIGURE 8-3 Configure Screen

The Configure screen contains the following fields:

- **Start Command —** The full command line that can be passed to any UNIX shell to start the base application. It is required that you specify this command. You can type the command in the field provided or use the **Browse** button to locate a file containing the command to start the application.

  The complete command line must include everything necessary to start the application, such as hostnames, port numbers, a path to configuration files, and so on. If your application requires a hostname to be specified on the command line, you can use the `$hostnames` variable that Agent Builder defines (see "Using the Agent Builder `$hostnames` Variable" on page 168).

  Do not enclose the command in double quotes ("").

Note – If the base application has multiple independent process trees, each of which is started with its own tag under PMF control, you cannot specify a single command. Rather, you must create a text file with individual commands to start each process tree, and specify the path to this file in the Start Command text field. See "Creating Resource Types With Multiple Independent Process Trees" on page 168, which lists some special characteristics this file requires to work properly.

- **Stop Command —** The full 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 the Browse button to locate a file containing the command to stop the application. If your application requires a hostname to be specified on the command line, you can use the `$hostnames` variable defined by Agent Builder (see "Using the Agent Builder `$hostnames` Variable" on page 168).

  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% 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% of the timeout value for the application to exit.
  - If `SIGKILL` is unsuccessful the `STOP` method exits unsuccessfully (the remaining 5% 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 appropriate exit status between 0 (success) and 100 (complete failure). This command is optional. You can type the complete path to the command or use the Browse button 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 used by the resource, and if that succeeds, declares the application healthy. You can only use a probe command with network aware applications. Agent Builder always generates a probe command, but disables it for non-network aware applications.

  If your application requires that you specify a hostname on the probe command line, you can use the `$hostnames` variable that Agent Builder defines (see "Using the Agent Builder `$hostnames` Variable" on page 168).

- **Timeout —** (for each command)—A timeout value (in seconds) for each command. You can specify a new value or accept the default value Agent Builder provides (300 seconds for start and stop; 30 seconds for probe).

Using the Agent Builder `$hostnames` Variable

For many applications, specifically network-aware applications, the hostname on which the application listens and services customer requests must be passed to the application on the command line. Therefore, in many cases, hostname is a parameter you must specify for start, stop, and probe commands for the target resource type (on the Configure screen). However, the hostname on which an application listens is cluster specific—it is determined when the resource is run on a cluster and cannot be determined when Agent Builder generates your resource type code.

To solve this problem, Agent Builder provides the `$hostnames` variable that you can specify on the command line for the start, stop, and probe commands. You specify the `$hostnames` variable exactly as you would an actual hostname, 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 hostname configured for that resource (in the `Network_resources_used` resource property of the resource) is substituted for the value of the `$hostnames` variable.

If you configure the `Network_resources_used` property with multiple hostnames, the `$hostnames` variable contains all of them separated by commas.

Creating Resource Types With Multiple Independent Process Trees

Agent Builder can create resource types for applications having more than one independent process tree. These process trees are independent in the sense that PMF monitors and starts them individually. PMF starts each process tree with its own tag.

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 blank lines. You specify this text file in the Start Command text field in the Configure screen.

You must also make certain this text file does not have execute permissions. This enables Agent Builder to distinguish this file, whose purpose is to start multiple process trees, from a simple executable script containing multiple commands. If this text file is given execute permissions, the resources would come up fine on a cluster, but all the commands would be started under one PMF tag, precluding the possibility of monitoring and restarting the process trees individually by PMF.

Reusing Completed Work

Agent Builder enables you leverage completed work in several ways.

- You can clone an existing resource type created with Agent Builder.
- You can edit the source code Agent Builder generates and then recompile the code to create a new package.

Cloning an Existing Resource Type

Follow this procedure to clone an existing resource type generated by Agent Builder.

1. **Load an existing resource type into Agent Builder. You can do this in either of two ways:**

   a. **Launch Agent Builder from the install directory (which contains the `rtconfig` file) for an existing resource type (created with Agent Builder), and Agent Builder loads the values for that resource type in the Create and Configure screens.**

   b. **Use the Load Resource Type command in the File menu.**

2. **Change the install directory on the Create screen.**

   You must use the **Browse** button 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 changes.**

   You might use this procedure to change the type of code generated for the resource type. For example, if you initially create a ksh version of a resource type but find over time that you require a C version, you can load the existing ksh resource type, change the language for the output to C, and then have Agent Builder build a C version of the resource type.

4. **Create the cloned resource type.**

   Select **Create** to create the resource type. Select **Next** to bring up the Configure screen. Select **Configure** to configure the resource type and then **Cancel** to finish.

Editing the Generated Source Code

To keep the process of creating a resource type simple, Agent Builder limits the number of inputs, which necessarily limits the scope of the generated resource type. Therefore, to add more sophisticated features, such as validation checks for additional properties, or to tune parameters Agent Builder does not expose, you need to modify the generated source code or the RTR file.

The source files are in the *install_directory*/*rt_name*/`src` directory. Agent Builder embeds comments in the source code at places you can add code. These comments are of the form (for C code):

```
/* User added code -- BEGIN vvvvvvvvvvvvvvvv */
/* User added code -- END   ^^^^^^^^^^^^^^^^ */
```

Note – These comments are identical in Ksh code, except they use the pound sign (#) to begin the comment line.

For example, *rt_name*`.h` declares all the utility routines that the different programs use. At the end of the list of declarations are comments that enable you to declare additional routines you might have added to any of your code.

Agent Builder also generates the makefile in the *install_directory*/*rt_name*/`src` directory, with appropriate targets. Use the `make` command to recompile the source code, and 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 35 for more information about the RTR file and Appendix A for information about properties).

Directory Structure

Agent Builder creates a directory structure to hold all the files it generates for the target resource type. You specify (on the **Create** screen) the install directory. You must specify separate install directories for any additional resource types you develop. Under the install directory, Agent Builder creates a subdirectory whose name is a concatenation of the vendor name and the resource-type name (from the **Create** screen). 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 install directory.

Under this subdirectory, Agent Builder creates and populates the directories listed in the following table.

TABLE 8-1

Directory Name	Contents
`bin`	For C output, contains the binary files compiled from the source files. For ksh output, contains the same files as the `src` directory.
`etc`	Contains the RTR file. Agent Builder concatenates the vendor name and resource type name, separated by a period (.), to form the RTR filename. 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 (man1m) 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 package.
`src`	Contains the source files that Agent Builder generates.
`util`	Contains the `start`, `stop`, and `remove` utility scripts that Agent Builder generates. See “Utility Scripts and Man Pages” on page 174. Agent Builder appends the resource type name to each of these script names; for example, `startftp`, `stopftp`, `removeftp`.

Outputs

This section describes the outputs that Agent Builder generates.

Source and Binary Files

The Resource Group Manager (RGM)—which manages resource groups and ultimately, resources on a cluster—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 running on the affected node. For example, the RGM calls the `STOP` method to stop a resource running on the affected node and then calls the resource's `START` method to start the resource on a different node. (See "RGM Model" on page 20, "Callback Methods" on page 24 and the `rt_callbacks`(1HA) man page for more information on this model).

To support this model, Agent Builder generates (in the *install_directory*`/`*rt_name*`/bin` directory) eight executable programs (C) or scripts (ksh) that 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` but rather calls *rt_name*`_monitor_start` and *rt_name*`_monitor_stop`, which start and stop the fault monitor by calling *rt_name*`_probe`.

The eight methods that Agent Builder generates are:

- *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`

Refer to the `rt_callbacks`(1HA) man page for specific information on each of these methods.

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`) containing code common to all methods.
- An object file (*rt_name*`.o`) for the common code.
- Source files (`*.c`) for each of the methods.
- Object files (`*.o`) for each of the methods.

Agent Builder links the *rt_name*`.o` file to each of the method `.o` files to create the executables in the *install_directory*`/`*rt_name*`/bin` directory.

For ksh output, the *install_directory*`/`*rt_name*`/bin` and *install_directory*`/`*rt_name*`/src` directories are identical—each contains the eight executable scripts corresponding to the seven callback methods and the `PROBE` method.

Note – The ksh output includes two compiled utility programs (`gettime` and `gethostnames`) that certain of the callback methods require for getting the time and 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 appropriate locations to add code. See “Editing the Generated Source Code” on page 170.

Utility Scripts and Man Pages

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 running on a cluster, generally by using administrative commands or SunPlex Manager. However, as a convenience, Agent Builder generates a customized utility script for this purpose (the start script) as well as scripts for stopping and removing a resource of the target resource type. These three scripts, 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. It also creates the network address resources (LogicalHostname or SharedAddress) that enable the application to communicate with the clients on the network.
- **Stop script**—stops and disables the resource.
- **Remove script**—undoes the work of the start script, that is, it 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 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 resource type name to the script names. For example, if the resource type 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 of the utility scripts. You should read these man pages before you launch these scripts because they document the parameters you need to pass to the script.

To view these man pages, specify the path to this man directory using the `-M` option with the `man` command. For example, if `SUNW` is the vendor and `ftp` is the resource type name, use 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, use the following command to view the `startftp`(1M) man page:

```
man -M /opt/SUNWftp/man startftp
```

Support Files

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, which declares resource and resource type properties 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 35 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.

Packaging Directory

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 resource type name, for example, SUNWftp. The Makefile in the *install_directory*/*rt_name*/src directory supports creation of a new package. For example, if you make changes to the source files and recompile the code, or 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 from more than one node simultaneously. You can solve this problem in one of two ways:

- Run the remove*rt_name* script from one node of the cluster before running pkgrm from any node.
- Run pkgrm from one node of the cluster, which takes care of all necessary clean up, then run pkgrm from the remaining nodes, simultaneously if necessary.

If pkgrm fails because you attempt to run it simultaneously from multiple nodes, run it again from one node then run it from the remaining nodes.

The rtconfig File

In the install directory, Agent Builder generates a configuration file, rtconfig, that contains the inputs you have entered for the Create and Configure screen. If you launch Agent Builder from the install directory for an existing resource type (or load an existing resource type using the File menu **Load Resource Type** command), Agent Builder reads the rtconfig file to initialize the Create and Configure inputs to those of the existing resource type, which is useful if you want to clone an existing resource type (see "Cloning an Existing Resource Type" on page 169.

Navigation

Navigating Agent Builder is simple and intuitive. Agent Builder is a two-step wizard with a corresponding screen for each step (Create and Configure). You enter information in each screen by:

- Typing information in a field.
- Browsing your directory structure and selecting a file or directory.
- Checking one of a set of mutually exclusive radio buttons—for example, **Scalable** or **Failover**.
- Checking an on/off box. For example, checking **Network Aware** identifies the base application as network aware, while leaving this box blank identifies a non-network aware application.

The buttons at the bottom of each screen enable you to complete the task, move to the next or previous screen, or exit Agent Builder. Agent Builder highlights or grays out these buttons as appropriate.

For example, when you have filled in the fields and checked the desired options on the Create screen, click the **Create** button at the bottom of the screen. The **Previous** and **Next** buttons are grayed out because no previous screen exists and you cannot go to the next step before you complete this one.

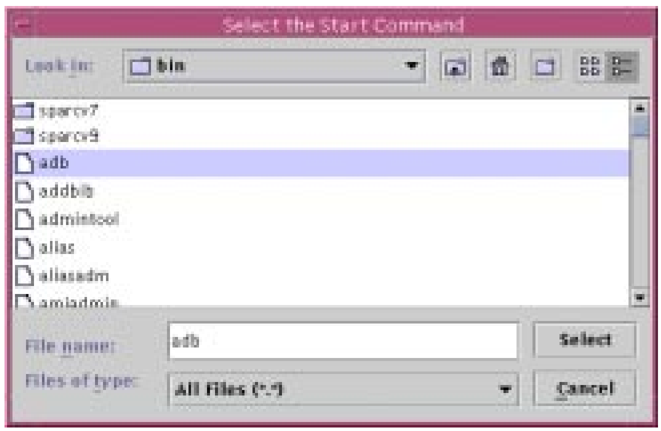
Agent Builder displays progress messages in the output log area at the bottom of the screen. When Agent Builder finishes, it displays a success message, or a warning to look at the output log. The **Next** button is highlighted, or if this is the last screen, only the **Cancel** button is highlighted.

You can select **Cancel** at any time to exit Agent Builder.

Browse Button

Certain Agent Builder fields enable you to type information or to click the **Browse** button to browse your directory structure and select a file or directory.

When you click **Browse**, a screen similar to the following appears:

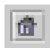


Double click on a folder to open it. When you highlight a file, its name appears in the **File name** box. Click **Select** when you have located and highlighted the appropriate file.

Note – If you are browsing for a directory, highlight it and select the **Open** button. If there are no subdirectories, Agent Builder closes the browse window and places the name of the directory you highlighted in the appropriate field. If this directory has subdirectories, click the close button to close the browse window and return to the previous screen. Agent Builder places the name of the directory you highlighted in the appropriate field.

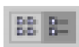
The icons in the upper right corner of the screen do the following:

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.

Menus

Agent Builder provides File and Edit menus.

File Menu

The File men has two commands:

- **Load Resource Type —** Load an existing resource type. Agent Builder provides a browse screen from which you select the install directory for an existing resource type. If a resource type exists in the directory from which you launch Agent Builder, Agent Builder automatically loads the resource type. The **Load Resource Type** command allows you to launch Agent Builder from any directory and select an existing resource type to use as a template for creating a new resource type (see “Cloning an Existing Resource Type” on page 169).
- **Exit —** Exit Agent Builder. You can also exit by clicking **Cancel** on the Create or Configure screen.

Edit Menu

The Edit menu has commands to clear and save the output log:

- **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 engaged in an iterative process of 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 —** Save the log output to a file. Agent Builder provides a browse screen that enables you to choose the directory and specify a filename.

CHAPTER 9

DSDL Reference

This chapter lists and briefly describes the DSDL API functions. See the individual 3HA man pages for a complete description of each DSDL function. The DSDL defines a C interface only. There is no scriptable DSDL interface.

The DSDL provides functions in the following categories.

DSDL Functions

The following subsections provide a brief overview to each category of DSDL functions. However, the individual 3HA man pages are the definitive reference for DSDL functions.

General Purpose Functions

The functions in this section provide a broad range of functionality. These functions enable you to

- Initialize the DSDL environment
- Retrieve resource, resource type, and resource group names, and extension property values
- Failover and restart a resource group and restart a resource
- Convert error strings to error messages
- Execute a command under a timeout

The following functions initialize the calling method.

- `scds_initialize`(3HA) allocate resources and initialize the DSDL environment.
- `scds_close`(3HA) – free resources allocated by `scds_initialize`(3HA.

The following functions retrieve information about resources, resource types, resource groups, and extension properties.

- `scds_get_resource_name`(3HA) – retrieve the name of the resource for the calling program.
- `scds_get_resource_type_name`(3HA) – retrieve the name of the resource type for the calling program.
- `scds_get_resource_group_name`(3HA) – retrieve the name of the resource group for the calling program.
- `scds_get_ext_property`(3HA) – retrieve the value of the specified extension property.
- `scds_free_ext_property`(3HA) – free the memory allocated by `scds_get_ext_property`(3HA).

The following functions fail over or restart a resource or resource group.

- `scds_failover_rg`(3HA) – fail over a resource group.
- `scds_restart_rg`(3HA) – restart a resource group.
- `scds_restart_resource`(3HA) – restart a resource.

The following two functions execute a command under a timeout and convert an error code to an error message.

- `scds_timerun`(3HA) – execute a command under a timeout value.
- `scds_error_string`(3HA) – translate an error code to an error string.

Property Functions

These functions provide convenience APIs for accessing specific properties of the relevant resource, resource group and resource type, including some commonly-used extension properties. The DSDL provides the `scds_initialize`(3HA) function to parse the command line arguments. The library then *caches* the various properties of the relevant resource, resource group and resource type.

A single man page, scds_property_functions(3HA) describes all of these functions. This section contains the following functions

- `scds_get_rs_`*property_name*
- `scds_get_rg_`*property_name*
- `scds_get_rt_`*property_name*
- `scds_get_ext_`*property_name*

Network Resource-Access Functions

The functions listed in this section retrieve, print, and free network resources used by resources and resource groups. The `scds_get_*` functions in this section provide a convenient way of retrieving network resources without querying specific properties such as `Network_resources_used`, and `Port_list` using the RMAPI functions. The `scds_print_*` functions print values from the data structures returned by the `scds_get_*` functions. The `scds_free_*` functions free the memory allocated by the `scds_get_*` functions.

The following functions are concerned with hostnames.

- `scds_get_rg_hostnames`(3HA) – retrieve a list of hostnames used by the network resources in a resource group.
- `scds_get_rs_hostnames`(3HA) – retrieve a list of hostnames used by the resource.
- `scds_print_net_list`(3HA) – print the contents of the list of hostnames returned by `scds_get_rg_hostnames`(3HA) or `scds_get_rs_hostnames`(3HA).
- `scds_free_net_list`(3HA) – free the memory allocated by `scds_get_rg_hostnames`(3HA) or `scds_get_rs_hostnames`(3HA).

The following functions are concerned with port lists.

- `scds_get_port_list`(3HA) – retrieve a list of port-protocol pairs used by a resource.
- `scds_print_port_list`(3HA) – print the contents of the list of port-protocol pairs returned by `scds_get_port_list`.
- `scds_free_port_list`(3HA) – free the memory allocated by `scds_get_port_list`.

The following functions are concerned with network addresses.

- `scds_get_netaddr_list`(3HA) – retrieve a list of network addresses used by a resource.
- `scds_print_netaddr_list`(3HA) – print the contents of the list of network addresses returned by `scds_get_netaddr_list`.
- `scds_free_netaddr_list`(3HA) – free the memory allocated by `scds_get_netaddr_list`.

Fault Monitoring Using TCP Connections

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 then disconnect from the service.

This section contains the following functions.

- `scds_tcp_connect`(3HA) – establish a TCP connection to a process.
- `scds_tcp_read`(3HA) – use a TCP connection to read data from the process being monitored.
- `scds_tcp_write`(3HA) – use a TCP connection to write data to a process being monitored.
- `scds_simple_probe`(3HA) – probe a process by establishing and terminating a TCP connection to the process.
- `scds_tcp_disconnect`(3HA) – terminate the connection to a process being monitored.

PMF Functions

These functions encapsulate the PMF functionality. The DSDL model for monitoring through PMF creates and uses implicit *tag* values for `pmfadm`(1M). The PMF facility also uses implicit values for the `Restart_interval`, `Retry_count` and `action_script` (the `-t`, `-n` and `-a` options to `pmfadm`(1M)). Most importantly, the DSDL ties the process death history, as discovered by PMF, into the application failure history as detected by the fault monitor to compute the restart or failover decision.

This section contains the following functions.

- `scds_pmf_get_status`(3HA) – determine if the specified instance is being monitored under PMF control.
- `scds_pmf_restart_fm`(3HA) – restarts the fault monitor using PMF.
- `scds_pmf_signal`(3HA) – send the specified signal to a process tree running under PMF control.
- `scds_pmf_start`(3HA) – execute a specified program (including a fault monitor) under PMF control.
- `scds_pmf_stop`(3HA) – terminate a process running under PMF control.
- `scds_stop_monitoring`(3HA) — stop monitoring a process running under PMF 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 section contains the following functions.

- `scds_fm_sleep`(3HA) – wait for a message on a fault monitor control socket.
- `scds_fm_action`(3HA) – take action after completion of a probe.
- `scds_fm_print_probes`(3HA) – write probe status information to the system log.

Utility Functions

The functions in this section enable you to write messages and debugging messages to the system log. This section contains the following two functions.

- `scds_syslog`(3HA) – write messages to the system log.
- `scds_syslog_debug`(3HA) – write a debugging message to the system log.

APPENDIX **A**

Standard Properties

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

The following is a list of the information in this appendix:

Note – The property values, such as `True` and `False`, are *not* case sensitive.

Resource Type Properties

TABLE A-1 describes the resource type properties defined by Sun Cluster. The property values are categorized as follows (in the Category column):

- **Required** — The property requires an explicit value in the Resource Type Registration (RTR) file or the object to which it belongs cannot be created. A blank 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 it and it is not available to administrative utilities. A blank 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/Explicit** — To exist, the property must be declared in the RTR file with an explicit value; otherwise, the RGM does not create it and it is not available to administrative utilities. A blank or the empty string is not allowed.
- **Optional** — The property can be declared in the RTR file,; if it isn't, 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 were not declared in the RTR file.

Resource type properties are not updatable by administrative utilities with the exception of `Installed_nodes`, which cannot be declared in the RTR file and must be set by the administrator.

TABLE A-1 Resource Type Properties *(1 of 3)*

Property Name	Description	Updatable	Category
API_version (integer)	The version of the resource management API used by this resource type implementation. The default for SC 3.0 is 2.	N	Optional
BOOT (string)	An optional callback method: the path to the program that the RGM invokes on a node, which joins or rejoins the cluster when a resource of this type is already managed. This method is expected to do initialization actions for resources of this type similar to the INIT method.	N	Conditional/Explicit
Failover (Boolean)	True indicates that resources of this type cannot be configured in any group that can be online on multiple nodes at once. The default is False.	N	Optional
FINI (string)	An optional callback method: the path to the program that the RGM invokes when a resource of this type is removed from RGM management.	N	Conditional/Explicit
INIT (string)	An optional callback method: the path to the program that the RGM invokes when a resource of this type becomes managed by the RGM.	N	Conditional/Explicit
Init_nodes (enum)	The values can be RG_primaries (just the nodes that can master the resource) or RT_installed_nodes (all nodes on which the resource type is installed). Indicates the nodes on which the RGM is to call the INIT, FINI, BOOT and VALIDATE methods. The default value is RG_primaries.	N	Optional
Installed_nodes (string array)	A list of the cluster node names on which the resource type is allowed to be run. The RGM automatically creates this property. The cluster administrator can set the value. You cannot declare this property in the RTR file. The default is all cluster nodes.	Y	Configurable by cluster administrator
Monitor_check (string)	An optional callback method: the path to the program that the RGM invokes before doing a monitor-requested failover of a resource of this type.	N	Conditional/Explicit
Monitor_start (string)	An optional callback method: the path to the program that the RGM invokes to start a fault monitor for a resource of this type.	N	Conditional/Explicit

TABLE A-1 Resource Type Properties *(2 of 3)*

Property Name	Description	Updatable	Category
Monitor_stop (string)	A callback method that is required if Monitor_start is set: the path to the program that the RGM invokes to stop a fault monitor for a resource of this type.	N	Conditional/Explicit
Pkglist (string array)	An optional list of packages that are included in the resource type installation.	N	Conditional/Explicit
Postnet_stop (string)	An optional callback method: the path to the program that the RGM invokes after calling the STOP method of any network-address resources (Network_resources_used) that a resource of this type is dependent on. This method is expected to do STOP actions that must be done after the network interfaces are configured down.	N	Conditional/Explicit
Prenet_start (string)	An optional callback method: the path to the program that the RGM invokes before calling the START method of any network-address resources (Network_resources_used) that a resource of this type is dependent on. This method is expected to do START actions that must be done before network interfaces are configured up.	N	Conditional/Explicit
RT_basedir (string)	The directory path that is used to complete relative paths for callback methods. This path is expected to be set to the installation location for the resource type packages. It must be a complete path, that is, it must start with a forward slash (/). This property is not required if all the method path names are absolute.	N	Required (unless all method path names are absolute)
RT_description (string)	A brief description of the resource type. The default is the empty string.	N	Conditional

TABLE A-1 Resource Type Properties *(3 of 3)*

Property Name	Description	Updatable	Category
`Resource_type` (string)	The name of the resource type. Must be unique in the cluster installation. You must declare this property as the first entry in the RTR file; otherwise, registration of the resource type fails. In addition, you can specify `Vendor_id` to identify the resource type. `Vendor_id` serves as a prefix that is separated from a resource type name by a ".", for example, `SUNW.http`. You can completely identify the resource type with `Resource_type` and Vendor_id or omit `Vendor_id`. For example, both `SUNW.http` and `http` are valid. If you specify the `Vendor_id`, use the stock symbol for the company that defines the resource type. If two resource-types in the cluster differ only in the `Vendor_id` prefix, the use of the abbreviated name fails. The default is the empty string.	N	Required
`RT_version` (string)	An optional version string of this resource type implementation.	N	Conditional/Explicit
`Single_instance` (Boolean)	If `True`, indicates that only one resource of this type can exist in the cluster. Hence, the RGM allows only one resource of this type to run cluster-wide at one time. The default value is `False`.	N	Optional
`START` (string)	A callback method: the path to the program that the RGM invokes to start a resource of this type.	N	Required (unless the RTR file declares a `PRENET_START` method)
`STOP` (string)	A callback method: the path to the program that the RGM invokes to stop a resource of this type.	N	Required (unless the RTR file declares a `POSTNET_STOP` method)
`UPDATE` (string)	An optional callback method: the path to the program that the RGM invokes when properties of a running resource of this type are changed.	N	Conditional/Explicit
`VALIDATE` (string)	An optional callback method: the path to the program that will be invoked to check values for properties of resources of this type.	N	Conditional/Explicit
`Vendor_ID` (string)	See the `Resource_type` property.	N	Conditional

Resource Properties

TABLE A-2 describes the resource properties defined by Sun Cluster. The property values are categorized as follows (in the Category column):

- **Required** — The administrator must specify a value when creating a resource with an administrative utility.
- **Optional** — If the administrator does not specify a value when creating 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 system administrators. A conditional property 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.

TABLE A-2 also lists whether and when resource properties are updatable (in the Updatable column), as follows

`None` or `False`	Never.
`True` or `Anytime`	Any time.
`At_creation`	When the resource is added to a cluster.
`When_disabled`	When the resource is disabled.

TABLE A-2 Resource Properties *(1 of 6)*

Property Name	Description	Updatable	Category
Cheap_probe_interval (integer)	The number of seconds between invocations of a quick fault probe of the resource. This property is only created by the RGM and available to the administrator 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 resource type file, the Tunable value for the property is When_disabled. This property is required if the Default attribute is not specified in the property declaration in the RTR file.	When disabled	Conditional
Extension properties	Extension properties as declared in the RTR file of the resource's type. The implementation of the resource type defines these properties. For information on the individual attributes you can set for extension properties. see TABLE A-4.	Depends on the specific property	Conditional
Failover_mode (enum)	Controls whether the RGM relocates a resource group or aborts a node in response to a failure of a START or STOP method call on the resource. None indicates that the RGM should just set the resource state on method failure and wait for operator intervention. Soft indicates that failure of a START method should cause the RGM to relocate the resource's group to a different node while failure of a STOP method should cause the RGM to set the resource state and wait for operator intervention. Hard indicates that failure of a START method should cause the relocation of the group and failure of a STOP method should cause the forcible stop of the resource by aborting the cluster node. The default is None.	Any time	Optional

TABLE A-2 Resource Properties *(2 of 6)*

Property Name	Description	Updatable	Category
`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 (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 it is coming to. The default value is `Lb_weighted`.	At creation	Conditional Optional
`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 distributed to the specified *node*. The fraction of load 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 1/4 of the load and node 2 receives 3/4. 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 resource type file, the `Tunable` value for the property is `Anytime`. Changing this property revises the distribution for new connections only. The default value is the empty string ("").	Any time	Conditional Optional
method`_timeout` for each callback method in the Type. (integer)	A time lapse, in seconds, after which the RGM concludes that an invocation of the method has failed. The default is 3,600 (one hour) if the method itself is declared in the RTR file.	Any time	Conditional Optional

TABLE A-2 Resource Properties *(3 of 6)*

Property Name	Description	Updatable	Category
`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`, the monitor does not have its `START` method called until it is enabled again. If the resource does not have a monitor callback method, this property does not exist. The default is `Enabled`.	Never	Query-only
`Network_resources_used` (string array)	A list of logical host name or shared address network resources used by the resource. For scalable services, this property must refer to shared address resources that exist in a separate resource group. For failover services, this property refers to logical host name or shared address resources that exist in the same resource group. 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. If the `Tunable` attribute is not specified in the resource type file, the `Tunable` value for the property is `At_creation`.	At creation	Conditional Required
`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 has no callbacks invoked until it is enabled again. The default is `Disabled`.	Never	Query-only
`Port_list` (string array)	A comma-separated list of port numbers on which the server is listening. Appended to each port number is the protocol being used by that port, for example, `Port_list=80/tcp`. 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. For specifics on setting up this property for Apache, see the Apache chapter in the *Sun Cluster 3.0 12/01 Data Services Installation and Configuration Guide.*	At creation	Conditional Required
`R_description` (string)	A brief description of the resource. The default is the empty string.	Any time	Optional

TABLE A-2 Resource Properties *(4 of 6)*

Property Name	Description	Updatable	Category
Resource_dependencies (string array)	A list of resources in the same group that must be online in order for this resource to be online. This resource cannot be started if the start of any resource in the list fails. When bringing the group offline, this resource is stopped before those in the list. Resources in the list are not allowed to be disabled unless this resource is disabled first. The default is the empty list.	Any time	Optional
Resource_dependencies_ weak (string array)	A list of resources in the same group that determines the order of method calls within the group. The RGM calls the START methods of the resources in this list before the START method of this resource and the STOP methods of this resource before the STOP methods of those in the list. The resource can still be online if those in the list fail to start or are disabled. The default is the empty list.	Any time	Optional
Resource_name (string)	The name of the resource instance. Must be unique within the cluster configuration and cannot be changed after a resource has been created.	Never	Required
Resource_state: on each cluster node (enum)	The RGM-determined state of the resource on each cluster node. Possible states are: Online, Offline, Stop_failed, Start_failed, Monitor_failed, and Online_not_monitored. This property is not user configurable.	Never	Query-only
Retry_count (integer)	The number of times a monitor attempts to restart a resource if it fails. This property is created by the RGM only and available to the administrator if it is declared in the RTR file. It is optional if a default value is specified in the RTR file. If the Tunable attribute is not specified in the resource type file, the Tunable value for the property is When_disabled. This property is required if the Default attribute is not specified in the property declaration in the RTR file.	When disabled	Conditional

TABLE A-2 Resource Properties *(5 of 6)*

Property Name	**Description**	**Updatable**	**Category**
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 only and available to the administrator if it is declared in the RTR file. It is optional if a default value is specified in the RTR file. If the Tunable attribute is not specified in the resource type file, the Tunable value for the property is When_disabled. This property is required if the Default attribute is not specified in the property declaration in the RTR file.	When disabled	Conditional
Scalable (Boolean)	Indicates whether the resource is scalable. If this property is declared in the RTR file, the RGM automatically creates the following scalable service properties for resources of that type: Network_resources_used, Port_list, Load_balancing_policy, and Load_balancing_weights. 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. When this property is declared in RTR file, the Tunable attribute must be set to At_creation or resource creation fails. If this property is not declared in the RTR file, the resource is not scalable, the cluster administrator 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, if desired, because they can be useful in a non-scalable service as well as in a scalable service.	At creation	Optional
Status: on each cluster node (enum)	Set by the resource monitor. Possible values are: OK, degraded, faulted, unknown, and offline. The RGM sets the value to unknown when the resource is brought online and to Offline when it is brought offline.	Never	Query-only

TABLE A-2 Resource Properties *(6 of 6)*

Property Name	**Description**	**Updatable**	**Category**
`Status_msg`: on each cluster node (string)	Set by the resource monitor at the same time as the `Status` property. This property is settable per resource per node. The RGM sets it to the empty string when the resource is brought offline.	Never	Query-only
`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 only and available to the administrator if it is declared in the RTR file. It is optional if a default value is specified in the RTR file. If the `Tunable` attribute is not specified in the resource type file, the `Tunable` value for the property is `When_disabled`. This property is required if the `Default` attribute is not specified in the property declaration in the RTR file.	When disabled	Conditional
`Type` (string)	The resource type of which this resource is an instance.	Never	Required

Resource Group Properties

TABLE A-3 describes the resource group properties defined by Sun Cluster. The property values are categorized as follows (in the Category column):

- **Required** — The administrator must specify a value when creating a resource group with an administrative utility.
- **Optional** — If the 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.

The Updatable column shows whether the property is updatable (Y) or not (N) after it is initially set.

TABLE A-3 Resource Group Properties

Property Name	Description	Updatable	Category
`Desired_primaries` (integer)	The number of nodes where the group is desired to be online at once. The default is 1. 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.	Y	Optional
`Failback` (Boolean)	A Boolean value that indicates whether to recalculate the set of nodes where the group is online when the cluster membership changes. A recalculation can cause the RGM to bring the group offline on less preferred nodes and online on more preferred nodes. The default is `False`.	Y	Optional
`Global_resources_used` (string array)	Indicates whether cluster file systems are used by any resource in this resource group. Legal values that the administrator can specify are an asterisk (*) to indicate all global resources, and the empty string (`""`) to indicate no global resources. The default is all global resources.	Y	Optional

TABLE A-3 Resource Group Properties *(Continued)*

Property Name	Description	Updatable	Category
`Implicit_network_dependencies` (Boolean)	A Boolean value that indicates, when `True`, that the RGM should enforce implicit strong dependencies of non-network-address resources on network-address 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. The default is `True`.	Y	Optional
`Maximum_primaries` (integer)	The maximum number of nodes where the group might be online at once. The default is 1. 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.	Y	Optional
`Nodelist` (string array)	A list of cluster nodes where the group can be brought online in order of preference. These nodes are known as the potential primaries or masters of the resource group. The default is the list of all cluster nodes.	Y	Optional
`Pathprefix` (string)	A directory in the cluster file system in which resources in the group can write can write essential administrative files. Some resources might require this property. Make `Pathprefix` unique for each resource group. The default is the empty string.	Y	Optional

TABLE A-3 Resource Group Properties *(Continued)*

Property Name	Description	Updatable	Category
`Pingpong_interval` (integer)	A non-negative integer value (in seconds) used by the RGM to determine where to bring the resource group online in the event of a reconfiguration or as the result of an `scha_control giveover` command or function being executed. In the event of a reconfiguration, if the resource group fails to come online more than once within the past `Pingpong_interval` seconds on a particular node (because the resource's `START` or `PRENET_START` method exited non-zero or timed out), that node is considered ineligible to host the resource group and the RGM looks for another master. If a call to a resource's `scha_control`(1ha)(3ha) command or function causes the resource group to be brought offline on a particular node within the past `Pingpong_interval` seconds, that node is ineligible to host the resource group as the result of a subsequent call to `scha_control` originating from another node. The default value is 3,600 (one hour).	Y	Optional
`Resource_list` (string array)	The list of resources that are contained in the group. The administrator does not set this property directly. Rather, the RGM updates this property as the administrator adds or removes resources from the resource group. The default is the empty list.	N	Query-only
`RG_dependencies` (string array)	Optional list of resource groups indicating a preferred ordering for bringing other groups online or offline on the same node. Has no effect if the groups are brought online on different nodes. The default is the empty list.	Y	Optional
`RG_description` (string)	A brief description of the resource group. The default is the empty string.	Y	Optional

TABLE A-3 Resource Group Properties *(Continued)*

Property Name	Description	Updatable	Category
RG_mode (enum)	Indicates whether the resource group is a failover or 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 have a value greater than 1, meaning 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. The default is Failover if Maximum_primaries is 1 and Scalable if Maximum_primaries is greater than 1.	N	Optional
RG_name (string)	The name of the resource group. Must be unique within the cluster.	N	Required
RG_state: on each cluster node (enum)	Set by the RGM to Online, Offline, Pending_online, Pending_offline or Error_stop_failed to describe the state of the group on each cluster node. A group can also exist in an unmanaged state when it is not under the control of the RGM. This property is not user configurable. The default is Offline.	N	Query-only

Resource Property Attributes

TABLE A-4 describes the resource property attributes that can be used to change system-defined properties or create extension properties.

Caution – You cannot specify `NULL` or the empty string ("") as the default value for `boolean`, `enum`, or `int` types.

TABLE A-4 Resource Property Attributes

Property	Description
`Property`	The name of the resource property.
`Extension`	If used, indicates that the RTR file entry declares an extension property defined by the resource type implementation. Otherwise, the entry is a system-defined property.
`Description`	A string annotation intended to be a brief description of the property. The description attribute cannot be set in the RTR file for system-defined properties.
Type of the property	Allowable types are: `string`, `boolean`, `int`, `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.
`Default`	Indicates a default value for the property.
`Tunable`	Indicates when the cluster administrator can set the value of this property in a resource. Can be set to `None` or `False` to prevent the administrator from setting the property. Values that allow administrator tuning are: `True` or `Anytime` (at any time), `At_creation` (only when the resource is created), or `When_disabled` (when the resource is offline). The default is `True` (`Anytime`).
`Enumlist`	For an `enum` type, a set of string values permitted for the property.
`Min`	For an `int` type, the minimal value permitted for the property.
`Max`	For an `int` type, the maximum value permitted for the property.
`Minlength`	For `string` and `stringarray` types, the minimum string length permitted.
`Maxlength`	For `string` and `stringarray` types, the maximum string length permitted.
`Array_minsize`	For `stringarray` type, the minimum number of array elements permitted.
`Array_maxsize`	For `stringarray` type, the maximum number of array elements permitted.

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 file.

This appendix includes the following code listings.

Resource Type Registration File Listing

The resource type registration (RTR) file contains resource and resource type property declarations that define the initial configuration of the data service at the time the cluster administrator registers the data service.

CODE EXAMPLE B-1 SUNW.Sample RTR File

```
#
# Copyright (c) 1998-2000 by Sun Microsystems, Inc.
# All rights reserved.
#
# Registration information for Domain Name Service (DNS)
#


#pragma ident“@(#)SUNW.sample1.100/05/24 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;
         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;
}
```

CODE EXAMPLE B-1 SUNW.Sample RTR File *(Continued)*

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

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

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

#
# Extension Properties
#

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

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

CODE EXAMPLE B-1 SUNW.Sample RTR File *(Continued)*

```
        PROPERTY = Probe_timeout;
        EXTENSION;
        INT;
        DEFAULT = 30;
        TUNABLE = ANYTIME;
        DESCRIPTION = "Time out value for the probe (seconds)";
}
```

START Method Code Listing

The RGM invokes the START method on a cluster node when the resource group containing the data service resource is brought online on that node or when the resource is enabled. In the sample application, the START method activates the in.named (DNS) daemon on that node.

CODE 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_start1.100/05/24 SMI"

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

        while getopts 'R:G:T:' opt
        do
                case "$opt" in
```

CODE EXAMPLE B-2 dns_svc_start Method *(Continued)*

```
                R)
                        # Name of the DNS resource.
                        RESOURCE_NAME=$OPTARG
                        ;;
                G)
                        # Name of the resource group in which the resource is
                        # configured.
                        RESOURCEGROUP_NAME=$OPTARG
                        ;;
                T)
                        # Name of the resource type.
                        RESOURCETYPE_NAME=$OPTARG
                        ;;

                *)
                    logger -p ${SYSLOG_FACILITY}.err \
                  -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME] \
                    "ERROR: Option $OPTARG unknown"
                     exit 1
                     ;;

                esac
        done

}



###############################################################################
#
# MAIN
#
###############################################################################

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

# Obtain the syslog facility to use to log messages.
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# Parse the arguments that have been passed to this method
parse_args "$@"

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

# Get the value of the Confdir property of the resource in order to start
```

CODE EXAMPLE B-2 `dns_svc_start` Method *(Continued)*

```
# DNS. Using the resource name and the resource group entered, find the value of
# Confdir value set by the cluster administrator when adding the resource.
config_info=`scha_resource_get -O Extension -R $RESOURCE_NAME -G
$RESOURCEGROUP_NAME Confdir`
# scha_resource_get returns the “type” as well as the “value” for the extension
# properties. Get only the value of the extension property.
CONFIG_DIR=`echo $config_info | awk ‘{print $2}’`

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

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

# Check that the named.conf file is present in the $CONFIG_DIR directory.
if [ ! -s named.conf ]; then
    logger -p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG] \
        “${ARGV0} File $CONFIG_DIR/named.conf is missing or empty”
    exit 1
fi

# Get the value for Retry_count from the RTR file.
RETRY_CNT=`scha_resource_get -O Retry_Count -R $RESOURCE_NAME -G \
$RESOURCEGROUP_NAME`

# Get the value for Retry_interval from the RTR file. Convert this value, which
is in
# seconds, to minutes for passing to pmfadm. Note that this is a
# conversion with round-up, for example, 50 seconds rounds up to one minute.
((RETRY_INTRVAL = `scha_resource_get -O Retry_Interval -R $RESOURCE_NAME -G
$RESOURCEGROUP_NAME` / 60))

# Start the in.named daemon under the control of PMF. Let it crash and restart
# up to $RETRY_COUNT times in a period of $RETRY_INTERVAL; if it crashes
# more often than that, PMF will cease trying to restart it. If there is a
# process already registered under the tag <$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.
```

CODE EXAMPLE B-2 `dns_svc_start` Method *(Continued)*

```
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 `STOP` method is invoked on a cluster node when the resource group containing the HA-DNS resource is brought offline on that node or the resource is disabled. This method stops the `in.named` (DNS) daemon on that node.

CODE 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_stop1.100/05/24 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
```

CODE EXAMPLE B-3 `dns_svc_stop` Method *(Continued)*

```
                        # configured.
                        RESOURCEGROUP_NAME=$OPTARG
                        ;;
                T)
                        # Name of the resource type.
                        RESOURCETYPE_NAME=$OPTARG
                        ;;

                *)
                    logger -p ${SYSLOG_FACILITY}.err \
                  -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME] \
                    "ERROR: Option $OPTARG unknown"
                     exit 1
                     ;;

                esac
        done

}


###############################################################################
#
# MAIN
#
###############################################################################

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

# Obtain the syslog facility to use to log messages.
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# Parse the arguments that have been passed to this method
parse_args "$@"

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

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

# Attempt to stop the data service in an orderly manner using a SIGTERM
# signal through PMF. Wait for up to 80% of the Stop_timeout value to
# see if SIGTERM is successful in stopping the data service. If not, send SIGKILL
# to stop the data service. Use up to 15% of the Stop_timeout value to see
# if SIGKILL is successful. If not, there is a failure and the method exits with
```

CODE EXAMPLE B-3 dns_svc_stop Method *(Continued)*

```
# non-zero status. The remaining 5% of the Stop_timeout is for other uses.
((SMOOTH_TIMEOUT=$STOP_TIMEOUT * 80/100))

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

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

    # Since the data service did not stop with a SIGTERM signal, use
    # SIGKILL now and wait for another 15% of the total timeout value.
    pmfadm -s $PMF_TAG.named -w $HARD_TIMEOUT KILL
    if [ $? -ne 0 ]; then
        logger -p ${SYSLOG_FACILITY}.err -t [SYSLOG_TAG] \
        "${ARGV0} Failed to stop HA-DNS; Exiting UNSUCCESFUL"

        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 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.

CODE 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.c1.100/05/24 SMI”


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

main()
{
    printf(“%d\n”, time(0));
    exit(0);
}
```

PROBE Program Code Listing

The PROBE program checks the availability of the data service using nslookup(1M) commands. The MONITOR_START callback method launches this program and the MONITOR_START callback method stops it.

CODE EXAMPLE B-5 dns_probe Program

```
#!/bin/ksh
#pragma ident“@(#)dns_probe1.100/04/19 SMI”
#
# Probe method for HA-DNS.
#
```

CODE EXAMPLE B-5 dns_probe Program *(Continued)*

```
# 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_probe1.100/05/24 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

}
```

CODE EXAMPLE B-5 `dns_probe` Program *(Continued)*

```
###############################################################################
#
# restart_service ()
#
# This function tries to restart the data service by calling the STOP method
# followed by the START method of the dataservice. If the dataservice has
# already died and no tag is registered for the dataservice under PMF,
# then this function fails the service over to another node in the cluster.
#
function restart_service
{

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

                # Obtain the STOP method name and the STOP_TIMEOUT value for
                # this resource.
                STOP_TIMEOUT=`scha_resource_get -O STOP_TIMEOUT \
                        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME`
                STOP_METHOD=`scha_resource_get -O STOP \
                        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME`
                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_NAME`
                START_METHOD=`scha_resource_get -O START \
                        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME`
                hatimerun -t $START_TIMEOUT $RT_BASEDIR/$START_METHOD \
                        -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME \
                        -T $RESOURCETYPE_NAME

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

CODE EXAMPLE B-5 `dns_probe` Program *(Continued)*

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


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

        return 0
}




###############################################################################
#
# decide_restart_or_failover ()
#
# This function decides the action to be taken upon the failure of a probe:
# restart the data service locally or fail over to another node in the cluster.
#
function decide_restart_or_failover
{

    # Check if this is the first restart attempt.
    if [ $retries -eq 0 ]; then
        # This is the first failure. Note the time of
        # this first attempt.
        start_time=`$RT_BASEDIR/gettime`
        retries=`expr $retries + 1`
        # Because this is the first failure, attempt to restart
        # the data service.
        restart_service
        if [ $? -ne 0 ]; then
            logger -p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG] \
                "${ARGV0} Failed to restart data service."
            exit 1
        fi
```

CODE EXAMPLE B-5 `dns_probe` Program *(Continued)*

```
    else
    # This is not the first failure
    current_time=`$RT_BASEDIR/gettime`
    time_diff=`expr $current_time - $start_time`
    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] \
                "${ARGV0} Failed to restart HA-DNS."
            exit 1
        fi
    fi
fi
}
```

CODE EXAMPLE B-5 `dns_probe` Program *(Continued)*

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

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

# Obtain the syslog facility to use to log messages.
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# Parse the arguments that have been passed to this method
parse_args "$@"

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

# The interval at which probing is to be done is set in the system defined
# property THOROUGH_PROBE_INTERVAL. Obtain the value of this property with
# scha_resource_get
PROBE_INTERVAL=`scha_resource_get -O THOROUGH_PROBE_INTERVAL -R $RESOURCE_NAME
-G $RESOURCEGROUP_NAME`

# Obtain the timeout value allowed for the probe, which is set in the
# PROBE_TIMEOUT extension property in the RTR file. The default timeout for
# nslookup is 1.5 minutes.
probe_timeout_info=`scha_resource_get -O Extension -R $RESOURCE_NAME -G \
$RESOURCEGROUP_NAME Probe_timeout`
PROBE_TIMEOUT=`echo $probe_timeout_info | awk '{print $2}'`

# Identify the server on which DNS is serving by obtaining the value
# of the NETWORK_RESOURCES_USED property of the resource.
DNS_HOST=`scha_resource_get -O NETWORK_RESOURCES_USED -R $RESOURCE_NAME -G \
$RESOURCEGROUP_NAME`

# Get the retry count value from the system defined property Retry_count
RETRY_COUNT=`scha_resource_get -O RETRY_COUNT -R $RESOURCE_NAME -G \
$RESOURCEGROUP_NAME`

# Get the retry interval value from the system defined property Retry_interval
RETRY_INTERVAL=`scha_resource_get -O RETRY_INTERVAL -R $RESOURCE_NAME -G \
$RESOURCEGROUP_NAME`

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

CODE EXAMPLE B-5 `dns_probe` Program *(Continued)*

```
# 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
         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.
```

CODE EXAMPLE B-5 `dns_probe` Program *(Continued)*

```
    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.

CODE 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_start1.100/05/24 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)
```

CODE EXAMPLE B-6 `dns_monitor_start` Method *(Continued)*

```
                        # 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_NAME`

# Start the probe for the data service under PMF. Use the infinite retries
# option to start the probe. Pass the resource name, group, and type to the
# probe method.
```

CODE EXAMPLE B-6 `dns_monitor_start` Method *(Continued)*

```
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.

CODE 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_stop1.100/05/24 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
                        ;;
```

CODE EXAMPLE B-7 `dns_monitor_stop` Method *(Continued)*

```
                G)
                        # Name of the resource group in which the resource is
                        # configured.
                        RESOURCEGROUP_NAME=$OPTARG
                        ;;
                T)
                        # Name of the resource type.
                        RESOURCETYPE_NAME=$OPTARG
                        ;;

                *)
                    logger -p ${SYSLOG_FACILITY}.err \
                  -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME] \
                    "ERROR: Option $OPTARG unknown"
                     exit 1
                     ;;

                esac
        done

}


###############################################################################
#
# MAIN
#
###############################################################################

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

# Obtain the syslog facility to use to log messages.
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# Parse the arguments that have been passed to this method
parse_args "$@"

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

# See if the monitor is running, and if so, kill it.
if pmfadm -q $PMF_TAG.monitor; then
    pmfadm -s $PMF_TAG.monitor KILL
    if [ $? -ne 0 ]; then
    logger -p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG] \
        "${ARGV0} Could not stop monitor for resource " \
        $RESOURCE_NAME
```

CODE EXAMPLE B-7 `dns_monitor_stop` Method *(Continued)*

```
        exit 1
    else
    # Could successfully stop the monitor. Log a message.
    logger -p ${SYSLOG_FACILITY}.info -t [$SYSLOG_TAG]\
        "${ARGV0} Monitor for resource " $RESOURCE_NAME \
        " successfully stopped"
    fi
fi

exit 0
```

`MONITOR_CHECK` Method Code Listing

This method verifies the existence of the directory pointed to by the `Confdir` property. The RGM calls `MONITOR_CHECK` whenever the `PROBE` method fails the data service over to a new node and also to check nodes that are potential masters.

CODE 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 1.100/05/24 SMI"

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

    while getopts 'R:G:T:' opt
    do
    case "$opt" in

    R)
```

CODE EXAMPLE B-8 dns_monitor_check Method *(Continued)*

```
    # Name of the DNS resource.
    RESOURCE_NAME=$OPTARG
    ;;

    G)
    # Name of the resource group in which the resource is
    # configured.
    RESOURCEGROUP_NAME=$OPTARG
    ;;

    T)
    # Name of the resource type.
    RESOURCETYPE_NAME=$OPTARG
    ;;

    *)
    logger -p ${SYSLOG_FACILITY}.err \
    -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME] \
    “ERROR: Option $OPTARG unknown”
    exit 1
    ;;

    esac
    done

}

###############################################################################
#
# MAIN
###############################################################################

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

# Obtain the syslog facility to use to log messages.
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# Parse the arguments that have been passed to this method.
parse_args “$@”

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

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

CODE EXAMPLE B-8 dns_monitor_check Method *(Continued)*

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

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

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

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

# Log a message indicating that monitor check was successful.
if [ $? -eq 0 ]; then
    logger -p ${SYSLOG_FACILITY}.info -t [$SYSLOG_TAG] \
    "${ARGV0} Monitor check for DNS successful."
    exit 0
else
    logger -p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG] \
    "${ARGV0} Monitor check for DNS not successful."
    exit 1
fi
```

VALIDATE Method Code Listing

This method verifies the existence of the directory pointed to by the `Confdir` 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 the data service over to a new node.

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


#pragma ident“@(#)dns_validate1.100/05/24 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.
```

CODE EXAMPLE B-9 dns_validate Method *(Continued)*

```
                        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 \
```

CODE EXAMPLE B-9 `dns_validate` Method *(Continued)*

```
                    -t [$SYSLOG_TAG] \
                    "ERROR: Option $OPTARG unknown"
                     exit 1
                     ;;

               esac
    done
}

##############################################################################
#
# MAIN
#
##############################################################################

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

# Obtain the syslog facility to use to log messages.
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`

# Set the Value of CONFDIR to null. Later, this method retrieves the value
# of the Confdir property from the command line or using scha_resource_get.
CONFDIR=""
UPDATE_PROPERTY=0
CONFDIR_FOUND=0

# Parse the arguments that have been passed to this method.
parse_args "$@"

# If the validate method is being called due to the updating of properties
# try to retrieve the value of the Confdir extension property from the command
# line. Otherwise, obtain the value of Confdir using scha_resource_get.
if ( (( $UPDATE_PROPERTY == 1 )) &&  (( CONFDIR_FOUND == 0 )) ); then
    config_info=`scha_resource_get -O Extension -R $RESOURCE_NAME \
        -G $RESOURCEGROUP_NAME Confdir`
    CONFDIR=`echo $config_info | awk '{print $2}'`
fi

# Verify that the Confdir property has a value. If not there is a failure
# and exit with status 1.
if [[ -z $CONFDIR ]]; then
    logger -p ${SYSLOG_FACILITY}.err \
        "${ARGV0} Validate method for resource "$RESOURCE_NAME " failed"
    exit 1
fi

# Now validate the actual Confdir property value.
```

CODE EXAMPLE B-9 `dns_validate` Method *(Continued)*

```
# 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.

CODE 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_update1.100/05/24 SMI"

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

CODE EXAMPLE B-10 `dns_update` Method *(Continued)*

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

        while getopts 'R:G:T:' opt
        do
                case "$opt" in
                R)
                        # Name of the DNS resource.
                        RESOURCE_NAME=$OPTARG
                        ;;
                G)
                        # Name of the resource group in which the resource is
                        # configured.
                        RESOURCEGROUP_NAME=$OPTARG
                        ;;
                T)
                        # Name of the resource type.
                        RESOURCETYPE_NAME=$OPTARG
                        ;;

                *)
                    logger -p ${SYSLOG_FACILITY}.err \
                  -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME] \
                    "ERROR: Option $OPTARG unknown"
                     exit 1
                     ;;

                esac
        done

}



###############################################################################
#
# MAIN
#
###############################################################################

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

# Obtain the syslog facility to use to log messages.
SYSLOG_FACILITY=`scha_cluster_get -O SYSLOG_FACILITY`
```

CODE EXAMPLE B-10 dns_update Method *(Continued)*

```
# 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_NAME`

# When the Update method is called, the RGM updates the value of the property
# being updated. This method must check if the fault monitor (probe)
# is running, and if so, kill it and then restart it.
if pmfadm -q $PMF_TAG.monitor; then

    # Kill the monitor that is running already
        pmfadm -s $PMF_TAG.monitor TERM
        if [ $? -ne 0 ]; then
                logger -p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG] \
                    “${ARGV0} Could not stop the monitor”
                exit 1
        else
                # Could successfully stop DNS. Log a message.
                logger -p ${SYSLOG_FACILITY}.info -t [$SYSLOG_TAG] \
                    “Monitor for HA-DNS successfully stopped”
        fi

    # Restart the monitor.
    pmfadm -c $PMF_TAG.monitor -n -1 -t -1 $RT_BASEDIR/dns_probe \
       -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME -T $RESOURCETYPE_NAME
    if [ $? -ne 0 ]; then
        logger -p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG]\
            “${ARGV0} Could not restart monitor for HA-DNS “
    exit 1
    else
    logger -p ${SYSLOG_FACILITY}.info -t [$SYSLOG_TAG]\
                    “Monitor for HA-DNS successfully restarted”

    fi
fi
exit 0
```

APPENDIX **C**

DSDL Sample Resource Type Code Listing

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 called by the callback methods. The code listings in this appendix are as follows.

xfnts.c

This file implements the subroutines called by the SUNW.xfnts methods.

CODE EXAMPLE C-1 xfnts.c

```
/*
 * Copyright (c) 1998-2001 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
 * 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.
 */
#defineSVC_WAIT_PCT3

/*
 * We need to use 95% of probe_timeout to connect to the port and the
```

CODE EXAMPLE C-1 `xfnts.c`

```
 * remaining time is used to disconnect from port in the svc_probe function.
 */
#defineSVC_CONNECT_TIMEOUT_PCT95

/*
 * 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.
 */

#defineSVC_WAIT_TIME5

/*
 * This value will be used as disconnect timeout, if there is no
 * time left from the probe_timeout.
 */

#defineSVC_DISCONNECT_TIMEOUT_SECONDS2


/*
 * 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)
{
    charxfnts_conf[SCDS_ARRAY_SIZE];
    scha_str_array_t *confdirs;
    scds_net_resource_list_t *snrlp;
    int rc;
    struct stat statbuf;
```

CODE EXAMPLE C-1 xfnts.c

```
	scds_port_list_t*portlist;
	scha_err_terr;

	/*
	 * Get the configuration directory for the XFS dataservice from the
	 * confdir_list extension property.
	 */
	confdirs = scds_get_ext_confdir_list(scds_handle);

	/* Return an error if there is no confdir_list extension property */
	if (confdirs == NULL || confdirs->array_cnt != 1) {
	scds_syslog(LOG_ERR,
	    "Property Confdir_list is not set properly.");
	return (1); /* Validation failure */
	}

	/*
	 * Construct the path to the configuration file from the extension
	 * property confdir_list. Since HA-XFS has only one configuration
	 * we will need to use the first entry of the confdir_list property.
	 */
	(void) sprintf(xfnts_conf, "%s/fontserver.cfg", confdirs->str_array[0]);

	/*
	 * Check to see if the HA-XFS configuration file is in the right place.
	 * Try to access the HA-XFS configuration file and make sure the
	 * permissions are set properly
	 */
	if (stat(xfnts_conf, &statbuf) != 0) {
	/*
	 * suppress lint error because errno.h prototype
	 * is missing void arg
	 */
	scds_syslog(LOG_ERR,
	    "Failed to access file <%s> : <%s>",
	    xfnts_conf, strerror(errno));/*lint !e746 */
	return (1);
	}

	/*
	 * Make sure that xfs binary exists and that the permissions
	 * are correct. The XFS binary are assumed to be on the local
	 * File system and not on the Global File System
	 */
	if (stat("/usr/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,
        “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;
```

CODE EXAMPLE C-1 xfnts.c

```
    }
    if (scds_get_ext_monitor_retry_interval(scds_handle) <= 0) {
    scds_syslog(LOG_ERR,
        “Property Monitor_retry_interval is not set.”);
    rc = 1; /* Validation Failure */
    goto finished;
    }

    /* All validation checks were successful */
    scds_syslog(LOG_INFO, “Successful validation.”);
    rc = 0;

finished:
    scds_free_net_list(snrlp);
    scds_free_port_list(portlist);

    return (rc); /* return result of validation */
}


/*
 * svc_start():
 *
 * Start up the X font server
 * Return 0 on success, > 0 on failures.
 *
 * The XFS service will be started by running the command
 * /usr/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];
    charcmd[SCDS_ARRAY_SIZE];
    scha_str_array_t *confdirs;
    scds_port_list_t *portlist;
    scha_err_terr;

    /* 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]);
```

CODE EXAMPLE C-1 `xfnts.c`

```
    /* 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”
     * 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():
 *
```

CODE EXAMPLE C-1 xfnts.c

```
 * 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_terr;

    /*
     * 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,
```

CODE EXAMPLE C-1 xfnts.c

```
        “No network address resources found in resource group.”);
    return (1);
    }

    /* Return an error if there are no network resources */
    if (netaddr == NULL || netaddr->num_netaddrs == 0) {
    scds_syslog(LOG_ERR,
        “No network address resource in resource group.”);
    return (1);
    }

    /*
     * Get the Start method timeout, port number on which to probe,
     * the Probe timeout value
     */
    svc_start_timeout = scds_get_rs_start_timeout(scds_handle);
    probe_timeout = scds_get_ext_probe_timeout(scds_handle);

    /*
     * sleep for SVC_WAIT_PCT percentage of start_timeout time
     * before actually probing the dataservice. This is to allow
     * the dataservice to be fully up in order to reply to the
     * probe. NOTE: the value for SVC_WAIT_PCT could be different
     * for different data services.
     * Instead of calling sleep(),
     * call scds_svc_wait() so that if service fails too
     * many times, we give up and return early.
     */
    if (scds_svc_wait(scds_handle, (svc_start_timeout * SVC_WAIT_PCT)/100)
    != SCHA_ERR_NOERR) {

    scds_syslog(LOG_ERR, “Service failed to start.”);
    return (1);
    }

    do {
    /*
     * probe the data service on the IP address of the
     * network resource and the portname
     */
    rc = svc_probe(scds_handle,
        netaddr->netaddrs[0].hostname,
        netaddr->netaddrs[0].port_proto.port, probe_timeout);
    if (rc == SCHA_ERR_NOERR) {
        /* Success. Free up resources and return */
        scds_free_netaddr_list(netaddr);
        return (0);
```

CODE EXAMPLE C-1 xfnts.c

```
    }

    /*
     * Dataservice is still trying to come up. Sleep for a while
     * before probing again. Instead of calling sleep(),
     * call scds_svc_wait() so that if service fails too
     * many times, we give up and return early.
     */
    if (scds_svc_wait(scds_handle, SVC_WAIT_TIME)
        != SCHA_ERR_NOERR) {
        scds_syslog(LOG_ERR, "Service failed to start.");
        return (1);
    }

    /* We rely on RGM to timeout and terminate the program */
    } while (1);

}

/*
 * This function starts the fault monitor for a HA-XFS resource.
 * This is done by starting the probe under PMF. The PMF tag
 * is derived as <RG-name,RS-name,instance_number.mon>. The restart option
 * of PMF is used but not the "infinite restart". Instead
 * interval/retry_time is obtained from the RTR file.
 */

int
mon_start(scds_handle_t scds_handle)
{
    scha_err_terr;

    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);
```

CODE EXAMPLE C-1 xfnts.c

```
    if (err != SCHA_ERR_NOERR) {
    scds_syslog(LOG_ERR,
        “Failed to start fault monitor.”);
    return (1);
    }

    scds_syslog(LOG_INFO,
        “Started the fault monitor.”);

    return (SCHA_ERR_NOERR); /* Successfully started Monitor */
}


/*
 * This function stops the fault monitor for a HA-XFS resource.
 * This is done via PMF. The PMF tag for the fault monitor is
 * constructed based on <RG-name_RS-name,instance_number.mon>.
 */

int
mon_stop(scds_handle_t scds_handle)
{

    scha_err_terr;

    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).
```

CODE EXAMPLE C-1 xfnts.c

```
 *
 * The probe does a simple socket connection to the XFS server on the specified
 * port which is configured as the resource extension property (Port_list) and
 * pings the dataservice. If the probe fails to connect to the port, we return
 * a value of 100 indicating that there is a total failure. If the connection
 * goes through and the disconnect to the port fails, then a value of 50 is
 * returned indicating a partial failure.
 *
 */
int
svc_probe(scds_handle_t scds_handle, char *hostname, int port, int timeout)
{
    int  rc;
    hrtime_tt1, t2;
    int sock;
    chartestcmd[2048];
    int time_used, time_remaining;
    time_tconnect_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 */
```

CODE EXAMPLE C-1 `xfnts.c`

```
    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
```

CODE EXAMPLE C-1 xfnts.c

```
     * 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);
```

CODE EXAMPLE C-1 `xfnts.c`

```
    return (SCDS_PROBE_COMPLETE_FAILURE/2);
    }
    return (0);
}
```

`xfnts_monitor_check` Method

This method verifies that the basic resource type configuration is valid.

CODE EXAMPLE C-2 `xfnts_monitor_check.c`

```
/*
 * Copyright (c) 1998-2001 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;
    intrc;

    /* 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);
```

CODE EXAMPLE C-2 xfnts_monitor_check.c

```
    /* 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

This method starts the xfnts_probe method.

CODE EXAMPLE C-3 xfnts_monitor_start.c

```
/*
 * Copyright (c) 1998-2001 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;
    intrc;

    /* 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.");
```

CODE EXAMPLE C-3 `xfnts_monitor_start.c`

```
    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

This method stops the `xfnts_probe` method.

CODE EXAMPLE C-4 `xfnts_monitor_stop.c`

```
/*
 * Copyright (c) 2001 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;
```

CODE EXAMPLE C-4 `xfnts_monitor_stop.c`

```
/*
    /* 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

The `xfnts_probe` method checks the availability of the application and decides whether to failover or restart the data service. The `xfnts_monitor_start` callback method launches this program and the `xfnts_monitor_stop` callback method stops it.

CODE EXAMPLE C-5 `xfnts_probe.c+`

```
/*
 * Copyright (c) 1998-2001 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"
```

CODE EXAMPLE C-5 `xfnts_probe.c+`

```
/*
 * 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_tscds_handle;

    hrtime_tht1, ht2;
    unsigned longdt;

    scds_netaddr_list_t *netaddr;
    char*hostname;

    if (scds_initialize(&scds_handle, argc, argv) != SCHA_ERR_NOERR) {
    scds_syslog(LOG_ERR, "Failed to initialize the handle.");
    return (1);
    }


    /* Get the ip addresses available for this resource */
    if (scds_get_netaddr_list(scds_handle, &netaddr)) {
    scds_syslog(LOG_ERR,
        "No network address resource in resource group.");
    scds_close(&scds_handle);
    return (1);
    }

    /* Return an error if there are no network resources */
    if (netaddr == NULL || netaddr->num_netaddrs == 0) {
    scds_syslog(LOG_ERR,
        "No network address resource in resource group.");
    return (1);
    }


    /*
     * Set the timeout from the X props. This means that each probe
```

CODE EXAMPLE C-5 xfnts_probe.c+

```
     * iteration will get a full timeout on each network resource
     * without chopping up the timeout between all of the network
     * resources configured for this resource.
     */
    timeout = scds_get_ext_probe_timeout(scds_handle);

    for (;;) {

    /*
     * sleep for a duration of thorough_probe_interval between
     *  successive probes.
     */
    (void) scds_fm_sleep(scds_handle,
        scds_get_rs_thorough_probe_interval(scds_handle));

    /*
     * Now probe all ipaddress we use. Loop over
     * 1. All net resources we use.
     * 2. All ipaddresses in a given resource.
     * For each of the ipaddress that is probed,
     * compute the failure history.
     */
    probe_result = 0;
    /*
     * Iterate through the all resources to get each
     * IP address to use for calling svc_probe()
     */
    for (ip = 0; ip < netaddr->num_netaddrs; ip++) {
        /*
         * Grab the hostname and port on which the
         * health has to be monitored.
         */
        hostname = netaddr->netaddrs[ip].hostname;
        port = netaddr->netaddrs[ip].port_proto.port;
        /*
         * HA-XFS supports only one port and
         * hence obtain the port value from the
         * first entry in the array of ports.
         */
        ht1 = gethrtime(); /* Latch probe start time */
        scds_syslog(LOG_INFO, “Probing the service on “
            “port: %d.”, port);

        probe_result =
        svc_probe(scds_handle, hostname, port, timeout);

        /*
```

CODE EXAMPLE C-5 `xfnts_probe.c+`

```
         * 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

The RGM invokes the `START` method on a cluster node when the resource group containing the data service resource is brought online on that node or when the resource is enabled. The `xfnts_start` method activates the `xfs` daemon on that node.

CODE EXAMPLE C-6 `xfnts_start.c`

```
/*
 * Copyright (c) 1998-2001 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
```

CODE EXAMPLE C-6 `xfnts_start.c` *(Continued)*

```
 * an action script.
 */

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

    /*
     * Process all the arguments that have been passed to us from RGM
     * and do some initialization for syslog
     */

    if (scds_initialize(&scds_handle, argc, argv) != SCHA_ERR_NOERR) {
    scds_syslog(LOG_ERR, “Failed to initialize the handle.”);
    return (1);
    }

    /* Validate the configuration and if there is an error return back */
    rc = svc_validate(scds_handle);
    if (rc != 0) {
    scds_syslog(LOG_ERR,
        “Failed to validate configuration.”);
    return (rc);
    }

    /* Start the data service, if it fails return with an error */
    rc = svc_start(scds_handle);
    if (rc != 0) {
    goto finished;
    }

    /* Wait for the service to start up fully */
    scds_syslog_debug(DBG_LEVEL_HIGH,
        “Calling svc_wait to verify that service has started.”);

    rc = svc_wait(scds_handle);

    scds_syslog_debug(DBG_LEVEL_HIGH,
        “Returned from svc_wait”);

    if (rc == 0) {
    scds_syslog(LOG_INFO, “Successfully started the service.”);
    } else {
    scds_syslog(LOG_ERR, “Failed to start the service.”);
    }
```

CODE EXAMPLE C-6 `xfnts_start.c` *(Continued)*

```
finished:
    /* Free up the Environment resources that were allocated */
    scds_close(&scds_handle);

    return (rc);
}
```

The `xfnts_stop` Method Code Listing

The RGM invokes the `STOP` method on a cluster node when the resource group containing the HA-XFS resource is brought offline on that node or the resource is disabled. This method stops the `xfs` daemon on that node.

CODE EXAMPLE C-7 `xfnts_stop.c`

```
/*
 * Copyright (c) 1998-2001 by Sun Microsystems, Inc.
 * All rights reserved.
 *
 * xfnts_svc_stop.c - Stop method for HA-XFS
 */

#pragma ident “@(#)xfnts_svc_stop.c 1.10 01/01/18 SMI”

#include <rgm/libdsdev.h>
#include “xfnts.h”

/*
 * Stops the HA-XFS process using PMF
 */

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

    scds_handle_tscds_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.”);
```

CODE EXAMPLE C-7 `xfnts_stop.c` *(Continued)*

```
    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);
}
```

The `xfnts_update` Method Code Listing

The RGM calls the UPDATE method to notify a running resource that its properties have been changed. The RGM invokes UPDATE after an administrative action succeeds in setting properties of a resource or its group.

CODE EXAMPLE C-8 `xfnts_update.c`

```
#pragma ident “@(#)xfnts_update.c  1.10     01/01/18 SMI”

/*
 * Copyright (c) 1998-2001 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
```

CODE EXAMPLE C-8 `xfnts_update.c` *(Continued)*

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

    /* Process the arguments passed by RGM and initialize syslog */
    if (scds_initialize(&scds_handle, argc, argv) != SCHA_ERR_NOERR) {
    scds_syslog(LOG_ERR, “Failed to initialize the handle.”);
    return (1);
    }

    /*
     * check if the Fault monitor is already running and if so stop and
     * restart it. The second parameter to scds_pmf_restart_fm() uniquely
     * identifies the instance of the fault monitor that needs to be
     * restarted.
     */

    scds_syslog(LOG_INFO, “Restarting the fault monitor.”);
    result = scds_pmf_restart_fm(scds_handle, 0);
    if (result != SCHA_ERR_NOERR) {
    scds_syslog(LOG_ERR,
        “Failed to restart fault monitor.”);
    /* Free up all the memory allocated by scds_initialize */
    scds_close(&scds_handle);
    return (1);
    }

    scds_syslog(LOG_INFO,
        “Completed successfully.”);

    /* Free up all the memory allocated by scds_initialize */
    scds_close(&scds_handle);

    return (0);
}
```

The `xfnts_validate` Method Code Listing

This method verifies the existence of the directory 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 the data service over to a new node.

CODE EXAMPLE C-9 `xfnts_validate.c`

```
/*
 * Copyright (c) 1998-2001 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;
    intrc;

    /* Process arguments passed by RGM and initialize syslog */
    if (scds_initialize(&scds_handle, argc, argv) != SCHA_ERR_NOERR) {
    scds_syslog(LOG_ERR, “Failed to initialize the handle.”);
    return (1);
    }
    rc = svc_validate(scds_handle);

    /* Free up all the memory allocated by scds_initialize */
    scds_close(&scds_handle);

    /* Return the result of validate method */
```

CODE EXAMPLE C-9 `xfnts_validate.c` *(Continued)*

```
    return (rc);

}
```

APPENDIX **D**

Legal RGM Names and Values

This appendix lists the requirements for legal characters for RGM names and values.

RGM Legal Names

RGM names fall into five categories:

- Resource group names
- Resource type names
- Resource names
- Property names
- Enumeration literal names

Except for resource type names, all names must comply with the following rules:

- Must be in ASCII.
- Must start with a letter.
- Can contain upper and lowercase letters, digits, dashes (-), and underscores (_).
- Must not exceed 255 characters.

A resource type name can be a simple name (specified by the `Resource_type` property in the RTR file) or a complete name (specified by the `Vendor_id` and `Resource_type` properties in the RTR file). When you specify both these properties, the RGM inserts a period between the `Vendor_id` and `Resource_type` to form the complete name. For example, if `Vendor_id=SUNW` and `Resource_type=sample`, the complete name is `SUNW.sample`. This is the only case where a period is a legal character in an RGM name.

RGM Values

RGM values fall into two categories: property values and description values, both of which share the same rules, as follows:

- 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 any of the following characters: null, newline, comma, or semicolon.

APPENDIX **E**

Requirements for Non-Cluster Aware Applications

An ordinary, non-cluster-aware application must meet certain requirements to be a candidate for high availability (HA). The section, “Analyzing the Application for Suitability” on page 30, lists these requirements. This appendix provides additional details for certain of the 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 global file system, making the data accessible by a surviving server in the event that one server fails. See information regarding cluster file systems in Sun Cluster 3.0 12/01 Concepts.

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.

Multihosted Data

The highly available global file systems' disksets are multihosted so that when a physical host crashes, one of the surviving hosts can access the disk. For an application to be highly available, its data must be highly available, and thus its data must reside in the global HA file systems.

The global file system is mounted on disk groups that are created as independent entities. The user can choose to use some disk groups as mounted global file systems and others as raw devices for use with a data service, such as HA Oracle.

An application might have command-line switches or configuration files pointing to the location of the data files. If the application uses hard-wired pathnames, you could change the pathnames to symbolic links that point to a files in a global file system, without changing the application code. See "Using Symbolic Links for Multihosted Data Placement" on page 267 for a more detailed discussion about using symbolic links.

In the worst case, the application's source code must be modified to provide some mechanism for pointing to the actual data location. You could do this by implementing additional command-line switches.

Sun Cluster supports the use of UNIX UFS file systems and HA raw devices configured in a volume manager. When installing and configuring Sun Cluster, the system administrator must specify which disk resources to use for UFS file systems and which for raw devices. Typically, raw devices are used only by database servers and multimedia servers.

Using Symbolic Links for Multihosted Data Placement

Occasionally an application has the path names of its data files 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 it 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 it renames the temporary file to have the real file name by using the `rename(2)` system call (or the `mv(1)` program). By creating the temporary file and then renaming it to the real file, the data service is attempting to ensure that its data file contents are always well formed.

Unfortunately, the `rename(2)` 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 global file system. Because the `/etc` file system is private to each host, the data is not available after a takeover or switchover.

The underlying problem in this situation is that the existing application is not aware of the symbolic link and was not written with symbolic links considered. 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 on the cluster's global 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 (that is, a host name configured into a logical host name resource that resides in the same resource group as the application resource), rather than that of the physical host.

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 takeover 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 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, 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 system 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 environment already handle multihomed hosts properly. 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 currently configured on the machine. A data service daemon that uses `INADDR_ANY` generally does not have to be changed to handle the Sun Cluster logical network addresses.

Binding to INADDR_ANY Versus Binding to Specific IP Addresses

Even in the non-multihomed case, 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 properly 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, it follows a specific order for plumbing, unplumbing and configuring network address up or down in relation to when it calls call data service resource methods. See "Deciding on the 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 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 stopping and starting methods accomplish their work by killing and restarting data service's daemons, then the data service stops and starts using the network addresses at the appropriate times.

Client Retry

To a network client, a takeover 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, then they also will 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.