Sun Cluster 3.0 Data Services Developers' Guide

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

Part Number 806-1422
November 2000, Revision A

Please Recycle

Contents

Preface

The *Sun Cluster 3.0 Data Services Developers' 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
Release Notes	*Sun Cluster 3.0 Release Notes*	806-1428
Hardware	*Sun Cluster 3.0 Hardware Guide*	806-1420
Installation	Sun Cluster 3.0 Installation Guide	806-1419
Administration	*Sun Cluster 3.0 System Administration Guide*	806-1423
Data Services Registration and Configuration	*Sun Cluster 3.0 Data Services Installation and Configuration Guide*	806-1421

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`

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 and serial numbers of your systems
- The release number of the operating environment (for example, Solaris 7)
- The release number of Sun Cluster (for example, Sun Cluster 3.0)

Use the following commands to gather information about each node 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
`scinstall -pv`	Displays Sun Cluster release and package version information
`scrgadm -pvv`	Displays a detailed listing of the static properties of all existing resource types, resource groups, and resources.
`scstat -g`	Displays dynamic state information for all resources and resource groups.

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

CHAPTER 1

Resource Management API Overview

This book provides guidelines for creating a highly available (HA) data service for a software application such as Oracle, iPlanet™ Web Server, DNS, and so on, using the Resource Management API (RMAPI). As such, this book is targeted at data service developers. This book uses the second person pronoun, “you”, throughout to address data service developers .

This chapter provides an overview of the concepts you need to understand in order to use the API.

The following information is in this chapter.

What Is Sun Cluster?

The Sun Cluster 3.0 system enables applications to be run and administered as highly available and scalable resources (data services). The cluster facility known as the Resource Group Manager, or RGM, provides the mechanism for high availability. The elements that form the programming interface to this facility include the following.

- A set of callback methods the RGM uses to control an application on the cluster

- API commands and functions that callback methods can use to access information about the elements in the cluster
- Process management facilities for monitoring and restarting processes on the cluster

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.

Resource Management Object Model

This section and its subsections introduce some fundamental terminology and explain how the different elements of the API are put together to create a highly available application.

The RGM and its associated API handle 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 could implement a resource type, ha-oracle, which makes an existing Oracle DBMS application highly available. An end user might define separate databases for marketing, engineering, and finance, each of which would be a resource of type ha-oracle. The cluster administrator could place these resources in separate resource groups so they could run on different nodes and fail over independently. Likewise, a developer could create a second resource type, ha-calendar, to implement a highly available calendar server that requires an Oracle database. The cluster administrator could place the resource for the finance calendar into the same resource group as the finance database resource so that both resources would 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.

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 resource type registration file follows a well-defined format described in “Setting Resource and Resource Type Properties” on page 17 and in the `rt_reg(4)` man page. See also “Defining the Resource Type Registration File” on page 56 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 the resource type implementation and underlying application on a cluster and registers it using administrative commands. The registration procedure enters into the cluster configuration the information from the resource type registration file. The Sun Cluster 3.0 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 27 and "Implementing a Scalable Resource" on page 28 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 15.

Resource Group Manager Administrative Interface

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.

Callback Methods

You use the Resource Management API (RMAPI) to implement a resource type. The key elements of a resource type are the callback methods, programs invoked by the RGM to control resources on the cluster. The API defines the arguments and return value of the callback methods.

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 RMAPI provides callback methods in 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 4 and the `rt_callbacks(1HA)` man page for more information on the callback methods. Also see Chapter 5 for examples of how the callback methods are used.

Access Methods

To support implementation of callback methods, the API provides an interface to the RGM in the form of methods to access resource properties and other cluster information. The access methods are provided both in the form of shell commands and in the form of C functions.

The API provides commands and functions to do the following:

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

See Chapter 4 for more information on the access methods. Also see Chapter 5 for examples of how the access methods are used.

CHAPTER 2

Using the Resource Management API

This chapter provides detailed information about using the Resource Management API to implement a resource type.

The following information is in this chapter.

Setting Resource and Resource Type Properties

Sun Cluster provides resource type properties that you can use to define the static configuration of a data service. Resource type properties can specify the type of the resource, its version, the version of the API, and so on, as well as specify paths to each of the callback methods. Table A–1 lists all the resource type properties.

You declare the resource type properties in the resource type registration (RTR) file. The RTR file defines the initial configuration of the data service at the time the cluster administrator registers the data service with Sun Cluster. With the exception of `Installed_nodes`, the cluster administrator cannot configure resource type properties.

Note - Table A–1, which describes the resource type properties, specifies whether each property is optional, required, or conditional. You do not have to declare optional properties in the RTR file because the system supplies a default value if you omit them. This table also lists the default value for each optional property. If you do not declare a required property in the RTR file, registration of the data service fails.If you do not declare a conditional property in the RTR file, the RGM does not create the property and it is not available to the cluster administrator.

The following example shows resource type property entries in an RTR file.

```
# Registration information for example resource type
Resource_type = example_RT;
Vendor_id = SUNW;
Pkglist = SUNWxxx;
RT_Basedir = /opt/SUNWxxx;
START   = bin/service_start;
STOP   = bin/service_stop;
```

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

Sun Cluster also provides resource properties, such as `Failover_mode`, `Thorough_probe_interval`, and method timeouts, that define the static configuration of the resource. Dynamic resource properties such as `Resource_state` and `Status` reflect the active state of a managed resource. In addition to the resource properties, a resource inherits the properties of its resource type. Table A–2 describes the resource properties.

As with resource type properties, you declare resource properties in the RTR file. 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.

By convention, resource property declarations follow the resource type declarations in the RTR file. Entries begin with an open curly bracket and end with a closed curly bracket. The following example shows resource declarations in a sample RTR file.

```
. . .

# 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;
}
# An extension property that can be set at resource creation

{
Property = Log_level;
 Extension;
  enum {OFF, TERSE, VERBOSE};
 DEFAULT = TERSE;
 TUNABLE = AT_CREATION;
```

```
 DESCRIPTION = “Controls the detail of message logging”;

}
```

`Start_timeout` and `Stop_timeout` are system-defined resource properties. Sun Cluster provides a minimum value (1 second) and a default value (3600 seconds) for all timeouts. This sample RTR file changes these values to 60 seconds minimum and 300 seconds default. The cluster administrator can accept the default value or change the value of the timeout to 60 seconds or greater.

Note - You must declare conditional system-defined resource properties in the resource type registration file for them to be available for resources of that type. That is, properties that are not declared cannot be set or queried.

The final point about resource properties is that the cluster administrator can configure them under certain conditions. The following table shows the `TUNABLE` attribute that determines when and if an administrator can configure a resource property.

`NONE` or `FALSE`	Never
`TRUE` or `ANYTIME`	Anytime
`AT_CREATION`	When the data service is added to a cluster
`WHEN_DISABLED`	When the data service is disabled

You can use other attributes to put limits on the configurability of a property. For example, the `Min` and `Max` attributes allow you to set ranges for integer properties. See Table A–4 for a complete list of resource property attributes.

Using Callback Methods

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

Accessing Resource and Resource Group Property Information

The callback methods that enable the RGM to control activation of cluster resources might require access to the properties of that resource. The API provides both shell commands and C functions that you can use in callback methods to access the system-defined and extension properties of resources.

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 the function 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 interface.

The C function for resource property access has a variable argument interface. The API defines string-valued tags that indicate an operation and determine the interpretation of the variable argument list. The “get” access function works in conjunction with “open” and “close” functions that do initialization, finalization, and memory management.

You use three functions together for resource property access:

- `scha_resource_open(3HA)` initializes access to a resource and returns a handle for `scha_resource_get`.
- `scha_resource_get(3HA)` accesses the resource information.
- `scha_resource_close(3HA)` invalidates the handle and frees memory allocated for `scha_resource_get` return values.

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)`.

A command version of `scha_resource_get` is provided for use in shell scripts. The command takes as flagged arguments an operation tag, the resource name, and its group name. Additional unflagged arguments might be available for some operation tags. The `scha_resource_get(1HA)` man page provides details on this access command.

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 should 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 functions or 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.

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

The RGM provides for automatically starting monitors for resources. 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 invoking the scha_control API 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.

When bringing a resource offline, the RGM invokes 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 invokes the MONITOR_START method after the resource itself has been started.

See the Sun Cluster 3.0 Data Services Installation and Configuration Guide for information on fault monitors built into Sun supplied data services.

Resource Group Failover and Restart Control

The scha_control API function allows 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), which determines if the node on which it is run 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 scha_control request. If MONITOR_CHECK fails on all nodes, the failover is canceled.

Resource Properties to Support Monitors

Resource monitors, like the callback methods, need general access to resource properties. Certain system-defined resource properties are specifically for use by monitors, although the resource type implementation determines whether they are used. The monitor-related resource properties are:

- Cheap_probe_interval
- Thorough_probe_interval
- Retry_count
- Retry_interval
- Status
- Status_msg

These properties can be read with the scha_resource_get(1HA)(3HA) access command and function.

Setting Status and Status_msg

The Status and Status_msg properties are to be set by the resource monitor to reflect the monitor's view of the resource state. The API provides a function, scha_resource_setstatus, that sets these properties. See the scha_resource_setstatus(3HA) and scha_resource_setstatus(1HA) man pages for details.

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

Resource Group Properties to Support Monitors

Some resource group properties that a monitor might use are: `Nodelist`, `Maximum_primaries`, `Desired_primaries`, `RG_state`, `Resource_list`, and `Global_resources_used`.

Resource group properties can be read with a set of access functions. An open function (`scha_resourcegroup_open(3HA)`) initializes resource group access, a close function (`scha_resourcegroup_close(3HA)`) frees memory allocated by the access function, and operation tag values drive a variable argument function (`scha_resourcegroup_get(3HA)`) that returns property values in client variables that are passed as reference arguments. See Table A–3 for a list of resource group properties.

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_resourcegroup_close(3HA)`.

The sciptable version of this functionality is implemented with a single command, `scha_resourcegroup_get(1HA)`.

No interface can directly change resource group properties, although control requests made using `scha_control` might cause the RGM to change the properties of a resource group. Resource group properties are changed by the RGM or by administrative action.

Resource Type Properties to Support Monitors

Some resource type properties, like `RT_basedir` and `Installed_nodes`, might be of use to a monitor, for example, to specify the location of the program that implements the monitor.

Resource type properties inherited by a particular resource of that type are accessible through the `scha_resource_get` function. An interface is also provided to access the properties of any resource type. All resource type properties are accessible.

The access interface for resource types follows the pattern of the access interface for resources and resource groups. Open and close functions provide initialization and memory management, and a variable argument function provides tag-determined access to properties.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). The sciptable version of this functionality is implemented with a single command, scha_resourcetype_get(1HA.

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

Process management facilities are provided with the Resource Management API to implement resource monitors and resource control callbacks. 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 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.

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. 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.

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.

The boolean resource type property, Failover, if set to TRUE, restricts 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 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 instantiated 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 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. Network-address

resources include the logical hostname and shared address resource types. This property defaults to TRUE.

Implementing a Scalable Resource

A scalable resource is a resource that 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.

The boolean resource property `Scalable` identifies a resource as scalable (TRUE) or not (FALSE). A resource whose `Scalable` property is TRUE is said to be in in *scalable mode*. A resource whose `Scalable` property is FALSE is said to be in *failover* mode.

If 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.
- `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 stick service, is always sent to the same cluster node regardless of the port number it is coming to.

  In case of scalable services, for those 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 run in both scalable and failover mode. To do so, declare the `Scalable` resource property in the data service's RTR file. You can declare it without a value (the default value is `FALSE`), or explicitly set its value to `FALSE`. By default, this resource runs in failover mode. However, the cluster administrator can make the resource run in scalable mode by changing the value of `Scalable` to `TRUE` with an administrative utility.

The cluster administrator creates a scalable resource group to contain scalable service resources. Scalable resources make use of shared address resources, which allow the multiple instances of a scalable service to appear as a single service to the client. The shared address resources upon which a scalable resource depends must reside in a separate failover resource group.

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 to include the resource group containing the shared address resources.

The `RG_mode` property allows 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 does not allow a resource whose `Failover` property is `TRUE` to be instantiated in a resource group whose `RG_mode` is `SCALABLE`.

See Sun Cluster 3.0 Concepts for additional information regarding scalable resources.

Validation Checks For Scalable Services

Whenever a scalable resource is created or updated, the RGM validates various resource properties. If the properites 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.

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.

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 identify the static library files, and the –R option to identify the dynamic library files.

```
# 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 it on all cluster nodes.

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 methods (if any).
3. Configure network addresses up.
4. Call data service's START methods (if any).

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

1. Call data service's STOP methods (if any).
2. Configure network addresses down.
3. Call data service's POSTNET_STOP methods (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.

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 those 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 would not necessarily learn of the connection break if the connection happens to be quiescent at the time.

For example, consider the case in which the client is waiting for a response from the server to a long-running request. In this scenario, the client's request message has already arrived at the server and has been acknowledged at the TCP layer, so the client's TCP module has no need to keep retransmitting it. The client application is now 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

Data Service Requirements

An ordinary, non-cluster-aware application must meet the requirements set out in this chapter to be a candidate for high availability (HA).

A data service is made highly available by configuring its resources into resource groups. The data service'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 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.

Client-Server Environment

Sun Cluster is designed for client-server networking environments. 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`. Such models typically have no inherent ability to recover from a server crash.

Crash Tolerance

The data service must be crash tolerant. That is, it must crash-recover disk data (if necessary) when it is started as the result of a cluster reconfiguration,

`scha_control` giveover, or `scswitch` switchover. Crash tolerance is a prerequisite for making a data service highly available because crash recovery (the ability to crash-recover the disk and restart the data service) is a data integrity issue.

Note - The data service is not required to be able to recover connections.

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 a data service 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, which 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.

A data service might have command-line switches or configuration files pointing to the location of the data files. If the data service uses hard-wired path names, you might change the path name to a symbolic link that points to a file in a global file system, without changing the data service code. See "Using Symbolic Links for Multihosted Data Placement" on page 41 for a more detailed discussion about using symbolic links.

In the worst case, the data service's source code must be modified to provide some mechanism for pointing to the actual data location. You might 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 the system administrator installs and configures Sun Cluster, he or she 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.

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 32.

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 irrelevant.

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.

Using Symbolic Links for Multihosted Data Placement

This section describes how to use symbolic links to avoid having to modify data service code. Occasionally an existing data service has the path names of its data files hard-wired, with no mechanism for overriding the hard-wired path names. To avoid modifying the data service's code, you can sometimes use symbolic links.

For example, suppose the data service 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 potential problem can occur with this use of symbolic links. That is, sometimes the data service, or one of its administrative procedures, modifies the data file name as well as its contents. For example, suppose that the data service 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.

```
rename("/etc/mydatafile.new", "/etc/mydatafile");
```

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 data service 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 data service 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.

CHAPTER 4

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 44 – 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 49 – 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 (see Table A–2).

  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 5

Sample Application

This chapter describes a Sun Cluster Data Services sample application, `in.named`. The `in.named` daemon is the Solaris implementation of the Domain Name Service (DNS). The sample application demonstrates how to make a data service 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 Application

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 SC 3.0 Process Monitor Facility (PMF). If application deaths exceed the failure count within the failure time window, the

resource group containing the application resource is automatically failed 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 data service is healthy. If the probe detects a hung DNS data 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 data service, then the probe attempts to fail over the data service to another node in the cluster.

Specifically, the sample application 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 or when the HA-DNS resource is enabled.
- A `STOP` callback method invoked by the RGM to stop the `in.named` daemon when the resource group containing HA-DNS goes offline or the resource is disabled.
- A fault monitor to check the reliability of the data 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 data 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 17 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 113.

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


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

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

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

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

START      =    dns_svc_start;
STOP      =    dns_svc_stop;

VALIDATE       =   dns_validate;
UPDATE       =   dns_update;

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

Tip - You must declare the `Resource_type` property as the first entry in the RTR file. Otherwise, registration of the resource type 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. 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 111 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 leaves the minimum of 1 (except for the `Stop_timeout`, which is 10) 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, (1 or greater and 10 or greater for the `Stop_timeout`. 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 89.

- 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.

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.

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.

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

Providing Common Functionality to All Methods

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

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
```

(Continuation)

```
##############################################################################

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 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=$RESOURCE_NAME.named
PMF_TAG=$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_stop`, `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 83 for more information.

Each 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 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 methods.

The parse function is called in `MAIN` as:

```
parse_args ''$@''
```

Generating Error Messages

It is recommended that methods use the `syslog` facility to output error messages to end users. All 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 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 `PRENET_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 32 for information about when you might want to use `PRENET_START` and `PRENET_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 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 methods, such as the `parse_args` function and obtaining the syslog facility, which are described in “Providing Common Functionality to All Methods” on page 62.

For the complete listing of the `START` method, see “`START` Method Code Listing” on page 116.

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 method uses the process monitor facility (pmfadm) to start the DNS daemon (in.named). If DNS crashes or fails to start, the method 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.

This START method is guaranteed to be idempotent. Although the RGM should not call a START method twice without first stopping the data service with a call to its STOP method, this START method exits with success even if DNS is already running.

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 70.

```
# Check if $CONFIG_DIR is accessible.
if [ ! -d $CONFIG_DIR ]; then
 logger -p ${SYSLOG_FACILITY}.err \
   -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME] \
   "${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 ${SYSLOG_FACILITY}.err \
   -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME] \
   "${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, if it crashes during startup. 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))
```

(Continuation)

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

# Log a message indicating that HA-DNS has been started.
if [ $? -eq 0 ]; then
 logger -p ${SYSLOG_FACILITY}.err \
   -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME] \
   "${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 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 methods, such as the parse_args function and obtaining the syslog facility, which are described in “Providing Common Functionality to All Methods” on page 62.

For the complete listing of the STOP method, see “STOP Method Code Listing” on page 119.

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 $RESOURCE_NAME.named; 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 $RESOURCE_NAME.named -w $HARD_TIMEOUT KILL
  if [ $? -ne 0 ]; then
      logger -p ${SYSLOG_FACILITY}.err \
      -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME]
      ''${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 ${SYSLOG_FACILITY}.err \
         -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME]
\
         ''HA-DNS is not started''

 # Even if HA-DNS is not running, exit success to avoid putting
```

(continued)

(Continuation)

```
 # 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 methods, such as the `parse_args` function and obtaining the syslog facility, which are described in "Providing Common Functionality to All Methods" on page 62.

For the complete listing of the `PROBE` method, see "`PROBE` Program Code Listing" on page 123.

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 server 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.c` 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
```

(continued)

(Continuation)

```
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
```

(continued)

(Continuation)

```
        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 ${SYSLOG_FACILITY}.err\
  -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME]\
  "${ARGV0} Probe for resource HA-DNS successful"
fi
```

The `decide_restart_or_failover` 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.

- 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 78, 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
{
```

(continued)

(Continuation)

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

                if [[ $? -ne 0 ]]; then
                        logger-p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG] \
                                ''${ARGV0} Stop method failed.''
                        return 1
                fi

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

                if [[ $? -ne 0 ]]; then
                        logger-p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG] \
                                ''${ARGV0} Start method failed.''
                        return 1
                fi


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

        return 0
```

(continued)

```
}
```

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 methods, such as the parse_args function and obtaining the syslog facility, which are described in "Providing Common Functionality to All Methods" on page 62.

For the complete listing of the MONITOR_START method, see "MONITOR_START Method Code Listing" on page 129.

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, MONITOR_START 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 `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 methods, such as the `parse_args` function and obtaining the syslog facility, which are described in “Providing Common Functionality to All Methods” on page 62.

For the complete listing of the `MONITOR_STOP` method, see “`MONITOR_STOP` Method Code Listing” on page 131.

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 $RESOURCE_NAME.monitor; then
 pmfadm -s $RESOURCE_NAME.monitor KILL
 if [ $? -ne 0 ]; then
   logger -p ${SYSLOG_FACILITY}.err \
    -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME] \
```

(continued)

(Continuation)

```
    "${ARGV0} Could not stop monitor for resource " \
    $RESOURCE_NAME
         exit 1
 else
   # could successfully stop the monitor. Log a message.
   logger -p ${SYSLOG_FACILITY}.err \
    -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME] \
    "${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 methods, such as the `parse_args` function and obtaining the syslog facility, which are described in “Providing Common Functionality to All Methods” on page 62.

For the complete listing of the MONITOR_CHECK method, see “MONITOR_CHECK Method Code Listing” on page 133.

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

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

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

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

# 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 84 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 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 # [arg..]
{

 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.
```

(continued)

(Continuation)

```
        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 ${SYSLOG_FACILITY}.err \
        -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME] \
        "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 ${SYSLOG_FACILITY}.err \
   -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME] \
   "${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 ${SYSLOG_FACILITY}.err \
    -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME] \
    "${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 ${SYSLOG_FACILITY}.err \
 -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME] \
 "${ARGV0} Validate method for resource "$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 then 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 $RESOURCE_NAME.monitor TERM
   if [ $? -ne 0 ]; then
      logger -p ${SYSLOG_FACILITY}.err \
        -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME] \
        "${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.

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

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

TABLE A–1 Resource Type Properties *(continued)*

Property Name	Description	Updatable	Category
`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
`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

TABLE A–1 Resource Type Properties *(continued)*

Property Name	Description	Updatable	Category
`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 *(continued)*

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)

TABLE A–1 Resource Type Properties *(continued)*

Property Name	Description	Updatable	Category
`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

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

TABLE A–2 Resource Properties *(continued)*

Property Name	Description	Updatable	Category
`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 *(continued)*

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

TABLE A–2 Resource Properties *(continued)*

Property Name	Description	Updatable	Category
`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 *(continued)*

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

TABLE A–2 Resource Properties *(continued)*

Property Name	Description	Updatable	Category
`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 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
`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

TABLE A–2 Resource Properties *(continued)*

Property Name	Description	Updatable	Category
`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 *(continued)*

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

TABLE A–2 Resource Properties *(continued)*

Property Name	Description	Updatable	Category
`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
`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

TABLE A–2 Resource Properties *(continued)*

Property Name	Description	Updatable	Category
`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
`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

TABLE A–3 Resource Group Properties *(continued)*

Property Name	Description	Updatable	Category
`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

TABLE A–3 Resource Group Properties *(continued)*

Property Name	Description	Updatable	Category
`RG_description` (string)	A brief description of the resource group. The default is the empty string.	Y	Optional
`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.

TABLE A–4 Resource Property Attributes *(continued)*

Property	Description
`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 with Sun Cluster.

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.sample 1.1 00/05/24 SMI’’

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

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

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

START      =  dns_svc_start;
STOP      =  dns_svc_stop;

VALIDATE        = dns_validate;
UPDATE        = dns_update;

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

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

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

# The MIN value for all method timeouts is set to 60 seconds. This
# prevents administrators from setting shorter timeouts, which do
not
# improve switchover/failover performance, and can lead to undesired
# RGM actions (false failovers, node reboot, or moving the resource
group
# to ERROR_STOP_FAILED state, requiring operator intervention).
Setting
# too-short method timeouts leads to a *decrease* in overall availability
# of the data service.
{
 PROPERTY = Start_timeout;
 MIN=60;
 DEFAULT=300;
```

(continued)

(Continuation)

```
}

{
       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;
```

(continued)

(Continuation)

```
        MIN=60;
        MAX=3600;
        DEFAULT=300;
        TUNABLE = ANYTIME;
}

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

#
# Extension Properties
#

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

# Time out value in seconds before declaring the probe as failed.
{
        PROPERTY = Probe_timeout;
        EXTENSION;
        INT;
        DEFAULT = 30;
        TUNABLE = ANYTIME;
        DESCRIPTION = ‘‘Time out value for the probe (seconds)’’;
}
```

START Method Code Listing

The RGM 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_start 1.1 00/05/24 SMI’’

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

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

                *)
                    logger -p ${SYSLOG_FACILITY}.err \
                    -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME]
\
                    ‘‘ERROR: Option $OPTARG unknown’’
```

(continued)

(Continuation)

```
                    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

# Get the value of the Confdir property of the resource in order
to start
# DNS. Using the resource name and the resource group entered, find
the value of
# Confdir value set by the cluster administrator 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}'

# 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
```

(continued)

(Continuation)

```
empty''
 exit 1
fi

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

# Get the value for Retry_interval from the RTR file. Convert this
value, which is in
# seconds, to minutes for passing to pmfadm. Note that this is necessarily
# a conversion with round-down, that is 59 seconds or less converts
to zero minutes.
((RETRY_INTRVAL =`scha_resource_get -O Retry_Interval
-R $RESOURCE_NAME -G $RESOURCEGROUP_NAMÈ / 60))

# Start the in.named daemon under the control of PMF. Let it crash
and restart
# up to $RETRY_COUNT times in a period of $RETRY_INTERVAL; if it
crashes
# more often than that, PMF will cease trying to restart it. If
there is a
# process already registered under the tag <$RESOURCE_NAME.named>,
then,
# PMF sends out an alert message that the process is already running.
echo ''Retry interval is ''$RETRY_INTRVAL
pmfadm -c $PMF_TAG.named -n $RETRY_CNT -t $RETRY_INTRVAL \
    /usr/sbin/in.named -c named.conf

# Log a message indicating that HA-DNS has been started.
if [ $? -eq 0 ]; then
 logger -p ${SYSLOG_FACILITY}.info -t [$SYSLOG_TAG]\
        ''${ARGV0} HA-DNS successfully started''
fi
exit 0
```

STOP Method Code Listing

The 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_stop 1.1 00/05/24 SMI’’

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

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

                *)
                    logger -p ${SYSLOG_FACILITY}.err \
                    -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME]
\
                    ‘‘ERROR: Option $OPTARG unknown’’
                     exit 1
                     ;;

                esac
        done

}


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

(continued)

(Continuation)

```
#############################################################################

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

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

# Attempt to stop the data service in an orderly manner using a
SIGTERM
# signal through PMF. Wait for up to 80% of the Stop_timeout value
to
# see if SIGTERM is successful in stopping the data service. If
not, send SIGKILL
# to stop the data service. Use up to 15% of the Stop_timeout value
to see
# if SIGKILL is successful. If not, there is a failure and the method
exits with
# non-zero status. The remaining 5% of the Stop_timeout is for other
uses.
((SMOOTH_TIMEOUT=$STOP_TIMEOUT * 80/100))

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

# See if in.named is running, and if so, kill it.
if pmfadm -q $PMF_TAG.named; then
 # Send a SIGTERM signal to the data service and wait for 80% of
the
 # total timeout value.
 pmfadm -s $PMF_TAG.named -w $SMOOTH_TIMEOUT TERM
 if [ $? -ne 0 ]; then
  logger -p ${SYSLOG_FACILITY}.info -t [SYSLOG_TAG] \
      ''${ARGV0} Failed to stop HA-DNS with SIGTERM; Retry
with \
       SIGKILL''

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

      exit 1
  fi
 fi
```

(continued)

(Continuation)

```
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.c 1.1 00/05/24 SMI''
```

(continued)

(Continuation)

```
#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_probe 1.1 00/04/19 SMI’’
#
# Probe method for HA-DNS.
#
# This program checks the availability of the data service using
nslookup, which
# queries the DNS server to look for the DNS server itself. If the
server
# does not respond or if the query is replied to by some other server,
# other server, then the probe concludes that there is some problem
with the
# the probe concludes that there is a problem with the data service
and either
# another node in the cluster. Probing is done at a specific interval

# set by the THOROUGH_PROBE_INTERVAL property in the RTR file.


#pragma ident ‘‘@(#)dns_probe 1.1 00/05/24 SMI’’


###############################################################################
```

(continued)

(Continuation)

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

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

                *)
                    logger -p ${SYSLOG_FACILITY}.err \
                    -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME]
\
                    ''ERROR: Option $OPTARG unknown''
                     exit 1
                     ;;

                esac
        done

}


################################################################################
# restart_service ()
#
# This function tries to restart the dataservice 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
```

(continued)

(Continuation)

```
        # 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
                        logger-p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG]
\
                                ''${ARGV0} Start method
failed.''
                        return 1
                fi
```

(continued)

(Continuation)

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

        return 0
}



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

 # Check if this is the first restart attempt.
 if [ $retries -eq 0 ]; then
  # This is the first failure. Note the time of
  # this first attempt.
  start_time=$RT_BASEDIR/gettimè
  retries=expr $retries + 1
  # Because this is the first failure, attempt to restart
 # the data service.
 restart_service
  if [ $? -ne 0 ]; then
   logger -p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG] \
       ''${ARGV0} Failed to restart data service.''
   exit 1
  fi
 else
  # This is not the first failure
  current_time=$RT_BASEDIR/gettimè
  time_diff=expr $current_time - $start_timè
  if [ $time_diff -ge $RETRY_INTERVAL ]; then
   # This failure happened after the time window
   # elapsed, so reset the retries counter,
   # slide the window, and do a retry.
   retries=1
    start_time=$current_time
   # Because the previous failure occurred more than
```

(continued)

(Continuation)

```
   # Retry_interval ago, attempt to restart the data service.
   restart_service
   if [ $? -ne 0 ]; then
    logger -p ${SYSLOG_FACILITY}.err \
        -t [$SYSLOG_TAG
        ‘‘${ARGV0} Failed to restart HA-DNS.’’
    exit 1
   fi
  elif [ $retries -ge $RETRY_COUNT ]; then
   # Still within the time window,
   # and the retry counter expired, so fail over.
   retries=0
   scha_control -O GIVEOVER -G $RESOURCEGROUP_NAME \
       -R $RESOURCE_NAME
   if [ $? -ne 0 ]; then
    logger -p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG] \
        ‘‘${ARGV0} Failover attempt failed.’’
    exit 1
   fi
  else
   # Still within the time window,
   # and the retry counter has not expired,
   # so do another retry.
   retries=expr $retries + 1
   restart_service
   if [ $? -ne 0 ]; then
    logger -p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG] \
        ‘‘${ARGV0} Failed to restart HA-DNS.’’
    exit 1
   fi
  fi
 fi
}


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

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

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

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

PMF_TAG=$RESOURCE_NAME.named

# 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
```

(continued)

(Continuation)

```
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

# 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
```

(continued)

(Continuation)

```
                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 the decide_restart_or_failover function,
 # which evaluates whether to restart the data service or to fail
it over
 # to another node.

 if [ $probefail -ne 0 ]; then
  decide_restart_or_failover
 else
  logger -p ${SYSLOG_FACILITY}.info -t [$SYSLOG_TAG]\
      ''${ARGV0} Probe for resource HA-DNS successful''
 fi
done
```

MONITOR_START Method Code Listing

This method starts the PROBE program for the data service.

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_start 1.1 00/05/24 SMI’’

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

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

                *)
      logger -p ${SYSLOG_FACILITY}.err \
                 -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME]
\
                    ‘‘ERROR: Option $OPTARG unknown’’
                     exit 1
                     ;;
                esac
        done

}
```

(continued)

(Continuation)

```
#############################################################################
# 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

# Find where the probe method resides by obtaining the value of
the
# RT_BASEDIR property of the data service.
RT_BASEDIR=scha_resource_get -O RT_BASEDIR -R $RESOURCE_NAME
-G $RESOURCEGROUP_NAME

# Start the probe for the data service under PMF. Use the infinite
retries
# option to start the probe. Pass the resource name, group, and
type to the
# probe method.
pmfadm -c $PMF_TAG.monitor -n -1 -t -1 \
    $RT_BASEDIR/dns_probe -R $RESOURCE_NAME -G $RESOURCEGROUP_NAME
\
    -T $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_stop 1.1 00/05/24 SMI’’


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

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

                *)
                    logger -p ${SYSLOG_FACILITY}.err \
                    -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME]
\
                    ‘‘ERROR: Option $OPTARG unknown’’
                     exit 1
                     ;;

                esac
        done

}


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

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

(continued)

(Continuation)

```
# 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

# See if the monitor is running, and if so, kill it.
if pmfadm -q $PMF_TAG.monitor; then
 pmfadm -s $PMF_TAG.monitor KILL
 if [ $? -ne 0 ]; then
  logger -p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG] \
      ''${ARGV0} Could not stop monitor for resource '' \
      $RESOURCE_NAME
         exit 1
 else
  # Could successfully stop the monitor. Log a message.
  logger -p ${SYSLOG_FACILITY}.info -t [$SYSLOG_TAG]\
      ''${ARGV0} Monitor for resource '' $RESOURCE_NAME
\
      '' successfully stopped''
 fi
fi

exit 0
```

MONITOR_CHECK Method Code Listing

This method verifies the existence of the directory pointed to by the `Confdir` property. The RGM calls `MONITOR_CHECK` whenever the `PROBE` method fails the data service over to a new node.

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
```

(Continuation)

```
verifiy
# that the configuration directory and files are available on the
new node.


#pragma ident ‘‘%Z%%M% %I% %E% SMI’’

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

 while getopts ‘R:G:T:’ opt
 do
  case ‘‘$opt’’ in

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

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

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

  *)
  logger -p ${SYSLOG_FACILITY}.err \
  -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME]
\
  ‘‘ERROR: Option $OPTARG unknown’’
  exit 1
  ;;

  esac
 done

}

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

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

# Obtain the syslog facility to use to log messages.
```

(continued)

(Continuation)

```
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

# 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_validate 1.1 00/05/24 SMI’’

################################################################################
# Parse program arguments.
#
function parse_args # [args ...]
```

(continued)

(Continuation)

```
{
 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
```

(continued)

(Continuation)

```
    CONFDIR=$VAL
    CONFDIR_FOUND=1
   fi
                        ;;

                *)
                    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

# 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
```

(continued)

(Continuation)

```
 logger -p ${SYSLOG_FACILITY}.err \
     ‘‘${ARGV0} Validate method for resource ‘‘$RESOURCE_NAME ‘‘ failed’’
 exit 1
fi

# Now validate the actual Confdir property value.

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

# Check that the named.conf file is present in the Confdir directory.
if [ ! -s $CONFDIR/named.conf ]; then
        logger -p ${SYSLOG_FACILITY}.err -t [$SYSLOG_TAG] \
            ‘‘${ARGV0} File $CONFDIR/named.conf is missing
or empty’’
        exit 1
fi

# Log a message indicating that the Validate method was successful.
logger -p ${SYSLOG_FACILITY}.info -t [$SYSLOG_TAG] \
   ‘‘${ARGV0} Validate method for resource ‘‘$RESOURCE_NAME
\
   ‘‘ completed successfully’’

exit 0
```

UPDATE Method Code Listing

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

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_update 1.1 00/05/24 SMI’’

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

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

                *)
                    logger -p ${SYSLOG_FACILITY}.err \
                    -t [$RESOURCETYPE_NAME,$RESOURCEGROUP_NAME,$RESOURCE_NAME]
\
                    ‘‘ERROR: Option $OPTARG unknown’’
                     exit 1
                     ;;

                esac
        done

}




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

(continued)

(Continuation)

```
#
###############################################################################

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

# 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
```

(continued)

(Continuation)

APPENDIX **C**

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.