IRIS FailSafe™ 2.0 Programmer's Guide

007–3900–002

CONTRIBUTORS

Written by Lori Johnson
Illustrated by Dany Galgani and Chris Wengelski
Edited by Rick Thompson
Production by Susan Gorski
Engineering contributions by Vidula Iyer, Michael Nishimoto, Hugh Shannon Jr., Bill Sparks, Paddy Sreenivasan, Dan Stekloff, Rebecca Underwood, and Manish Verma

© 1999, Silicon Graphics, Inc.— All Rights Reserved

The contents of this document may not be copied or duplicated in any form, in whole or in part, without the prior written permission of Silicon Graphics, Inc.

LIMITED AND RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in the Rights in Data clause at FAR 52.227-14 and/or in similar or successor clauses in the FAR, or in the DOD, DOE or NASA FAR Supplements. Unpublished rights reserved under the Copyright Laws of the United States. Contractor/manufacturer is Silicon Graphics, Inc., 2011 N. Shoreline Blvd., Mountain View, CA 94043-1389.

IRIS, IRIX, and Silicon Graphics are registered trademarks and IRIS FailSafe and the Silicon Graphics logo are trademarks of Silicon Graphics, Inc. INFORMIX is a trademark of Informix Software, Inc. Netscape is a trademark of Netscape Communications Corporation. NFS is a trademark of Sun Microsystems, Inc. Oracle is a trademark of Oracle Corporation.

New Features in This Guide

This update contains the following:

- Revised `ordered` and `round.robin` failover scripts
- Restrictions on resource type names
- Action script API change
- The `ha_get_info()` and `ha_get_info_debug()` functions for reading resource information.
- More information about the `ha_exec2`(8) command
- Information about the execution of action scripts and failover scripts
- Different purposes of the `monitor` and `exclusive` action scripts
- Investigating resource group failures

Record of Revision

Version	Description
002	December 1999 Published in conjunction with the latest IRIS FailSafe 2.0 rollup patch. It supports IRIX 6.5.2 and later.

Contents

Figures

Tables

About This Guide

This guide explains how to write the set of scripts that are required to turn an application into a high-availability service in conjunction with IRIS FailSafe 2.0 software. It also tells you how to create a new resource type and provides instructions for migrating script information from Release 1.2 to Release 2.0 (or later).

This guide assumes that the IRIS FailSafe system has been configured as described in the *IRIS FailSafe 2.0 Administrator's Guide*.

This guide was prepared in conjunction with latest FailSafe 2.0 rollup patch. It supports IRIX 6.5.2 and later.

Audience

This guide is written for system programmers who are developing scripts for the IRIS FailSafe system. These scripts allow the failover of applications that are not handled by the base and optional IRIS FailSafe products. These programmers must be familiar with the operation and administration of nodes running IRIS FailSafe, with the applications that are to be failed over, and with the *IRIS FailSafe 2.0 Administrator's Guide*.

Related Documentation

Besides this guide, other documentation for the IRIS FailSafe system includes

- *IRIS FailSafe 2.0 Administrator's Guide*
- *IRIS FailSafe 2.0 INFORMIX Administrator's Guide*
- *IRIS FailSafe 2.0 NFS Administrator's Guide*
- *IRIS FailSafe 2.0 Oracle Administrator's Guide*
- *IRIS FailSafe 2.0 Netscape Server Administrator's Guide*
- *IRIS FailSafe 2.0 Samba Administrator's Guide*

The IRIS FailSafe reference pages are as follows:

- `cbeutil`(1M)
- `cdbBackup`(1M)
- `cdbRestore`(1M)
- `cdbutil`(1M)
- `cluster_mgr`(1M)
- `crsd`(1M)
- `failsafe`(7M)
- `fs2d`(1M)
- `ha_cilog`(1M)
- `ha_cmsd`(1M)
- `ha_exec2`(1M)
- `ha_fsd`(1M)
- `ha_gcd`(1M)
- `ha_ifd`(1M)
- `ha_ifdadmin`(1M)
- `ha_macconfig2`(1M)
- `ha_srmd`(1M)
- `ha_statd2`(1M)
- `haStatus`(1M)

Release notes are included with each IRIS FailSafe product. The names of the release notes are as follows:

Release Note	Product
`cluster_admin`	Cluster administration services
`cluster_control`	Cluster node control services
`cluster_ha`	Cluster high availability services
`failsafe2`	IRIS 2.0 FailSafe release
`failsafe2_informix`	FailSafe/INFORMIX
`failsafe2_nfs`	FailSafe/NFS
`failsafe2_oracle`	FailSafe/Oracle
`failsafe2_samba`	FailSafe/Samba
`failsafe2_web`	FailSafe/Netscape web
patch_number	FailSafe patch release

Conventions Used in This Guide

These type conventions and symbols are used in this guide:

`Bold`	Function names literal command-line arguments (options/flags)
`Bold fixed-width type`	Commands and text that you are to type literally in response to shell and command prompts
`Italics`	New terms, manual/book titles, commands, variable command-line arguments, filenames, and variables to be supplied by the user in examples, code, and syntax statements
`Fixed-width type`	Code examples, error messages, prompts, and screen text
`#`	IRIX shell prompt for the superuser (`root`)

Introduction to IRIS FailSafe Programming

IRIS FailSafe 2.0 provides highly available services for a cluster that contains up to 8 nodes. These services are monitored by the IRIS FailSafe software. You can create additional services that are highly available by using the instructions in this guide.

This chapter provides an introduction to IRIS FailSafe programming. The sections are as follows:

IRIS FailSafe Concepts

In order to use IRIS FailSafe, you must understand the concepts in this section.

Cluster Node (or Node)

A *cluster node* is a single IRIX image. Usually, a cluster node is an individual computer. The term *node* is also used in this guide for brevity; this use of node does not have the same meaning as a node in an Origin system.

Pool

A *pool* is the entire set of nodes involved with a group of clusters. The group of clusters are usually close together and should always serve a common purpose. A replicated database is stored on each node in the pool.

Cluster

A *cluster* is a collection of one or more nodes coupled to each other by networks or other similar interconnections. A cluster is identified by a simple name; this name

must be unique within the pool. A particular node may be a member of only one cluster. All nodes in a cluster are also in the pool; however, all nodes in the pool are not necessarily in the cluster.

Node Membership

A *node membership* is the list of nodes in a cluster on which IRIS FailSafe can allocate resource groups.

Process Membership

A *process membership* is the list of process instances in a cluster that form a process group. There can be multiple process groups per node.

Resource

A *resource* is a single physical or logical entity that provides a service to clients or other resources. For example, a resource can be a single disk volume, a particular network address, or an application such as a web server. A resource is generally available for use over time on two or more nodes in a cluster, although it can be allocated to only one node at any given time.

Resources are identified by a *resource name* and a *resource type*. One resource can be dependent on one or more other resources; if so, it will not be able to start (that is, be made available for use) unless the dependent resources are also started. Dependent resources must be part of the same *resource group* and are identified in a *resource dependency list*.

Resource Type

A *resource type* is a particular class of resource. All of the resources in a particular resource type can be handled in the same way for the purposes of *failover*. Every resource is an instance of exactly one resource type.

A resource type is identified by a simple name; this name must be unique within the cluster. A resource type can be defined for a specific node, or it can be defined for an entire cluster. A resource type that is defined for a specific node overrides a clusterwide resource type definition with the same name; this allows an individual node to override global settings from a clusterwide resource type definition.

Like resources, a resource type can be dependent on one or more other resource types. If such a dependency exists, at least one instance of each of the dependent resource types must be defined. For example, a resource type named `Netscape_web` might have resource type dependencies on resource types named `IP_address` and `volume`. If a resource named `web1` is defined with the `Netscape_web` resource type, then the resource group containing `web1` must also contain at least one resource of the type `IP_address` and one resource of the type `volume`.

The IRIS FailSafe software includes many predefined resource types. If these types fit the application you want to make highly available, you can reuse them. If none fit, you can create additional resource types by using the instructions in this guide.

Resource Name

A *resource name* identifies a specific instance of a *resource type*. A resource name must be unique for a given resource type.

Resource Group

A *resource group* is a collection of interdependent resources. A resource group is identified by a simple name; this name must be unique within a cluster. Table 1-1 shows an example of the resources for a resource group named `WebGroup`.

Table 1-1 Example Resource Group

Resource	Resource Type
`vol1`	`volume`
`/fs1`	`filesystem`
`10.10.48.22`	`IP_address`
`web1`	`Netscape_web`

If any individual resource in a resource group becomes unavailable for its intended use, then the entire resource group is considered unavailable. Therefore, a resource group is the unit of failover for IRIS FailSafe.

Resource groups cannot overlap; that is, two resource groups cannot contain the same resource.

For information about configuring resource groups, see the *IRIS FailSafe 2.0 Administrator's Guide*.

Resource Dependency List

A *resource dependency list* is a list of resources upon which a resource depends. Each resource instance must have resource dependencies that satisfy its resource type dependencies before it can be added to a resource group.

Resource Type Dependency List

A *resource type dependency list* is a list of resource types upon which a resource type depends. For example, the `filesystem` resource type depends upon the`volume` resource type, and the `Netscape_web` resource type depends upon the `filesystem` and `IP_address` resource types.

For example, suppose a file system instance `fs1` is mounted on volume `vol1` Before `fs1` can be added to a resource group, `fs1` must be defined to depend on `vol1` IRIS FailSafe only knows that a file system instance must have one volume instance in its dependency list. This requirement is inferred from the resource type dependency list.

Failover

A *failover* is the process of allocating a resource group (or application) to another node, according to a *failover policy*. A failover may be triggered by the failure of a resource, a change in the node membership (such as when a node fails or starts), or a manual request by the administrator.

Failover Policy

A *failover policy* is the method used by IRIS FailSafe to determine the destination node of a failover. A failover policy consists of the following:

- *Failover domain*
- *Failover attributes*
- *Failover script*

IRIS FailSafe uses the failover domain output from a failover script along with failover attributes to determine on which node a resource group should reside.

The administrator must configure a failover policy for each resource group. A failover policy name must be unique within the *pool*.

Failover Domain

A *failover domain* is the `ordered` list of nodes on which a given resource group can be allocated. The nodes listed in the failover domain must be within the same cluster; however, the failover domain does not have to include every node in the cluster.

The administrator defines the *initial failover domain* when creating a failover policy. This list is transformed into a *run-time failover domain* by the *failover script*; IRIS FailSafe uses the run-time failover domain along with failover attributes and the node membership to determine the node on which a resource group should reside. IRIS FailSafe stores the run-time failover domain and uses it as input to the next failover script invocation. Depending on the run-time conditions and contents of the failover script, the initial and run-time failover domains may be identical.

In general, IRIS FailSafe allocates a given resource group to the first node listed in the run-time failover domain that is also in the node membership; the point at which this allocation takes place is affected by the failover attributes.

Failover Attribute

A *failover attribute* is a string that affects the allocation of a resource group in a cluster. The administrator must specify system attributes (such as `Auto_Failback` or `Controlled_Failback`), and can optionally supply site-specific attributes.

Failover Scripts

A *failover script* is a shell script that generates a *run-time failover domain* and returns it to the IRIS FailSafe process. The IRIS FailSafe process applies the failover attributes and then selects the first node in the returned failover domain that is also in the current node membership.

The following failover scripts are provided with the IRIS FailSafe release:

- `ordered`, which never changes the initial failover domain. When using this script, the initial and run-time failover domains are equivalent.
- `round-robin`, which selects the resource group owner in a round-robin (circular) fashion. This policy can be used for resource groups that can be run in any node in the cluster.

If these scripts do not meet your needs, you can create a new failover script using the information in this guide.

Action Scripts

The *action scripts* are the set of scripts that determine how a resource is started, monitored, and stopped. There must be a set of action scripts specified for each resource type.

The following is the complete set of action scripts that can be specified for each resource:

- `probe`, which verifies that the resource is configured on a server
- `exclusive`, which verifies that the resource is not already running
- `start`, which starts the resource
- `stop`, which stops the resource
- `monitor`, which monitors the resource
- `restart`, which restarts the resource on the same server after a monitoring failure occurs

The IRIS FailSafe software includes action scripts for predefined resource types. If these scripts fit the resource type that you want to make highly available, you can reuse them by copying them and modifying them as needed. If none fits, you can create additional action scripts by using the instructions in this guide.

Highly Available Services Included with IRIS FailSafe

The base IRIS FailSafe release includes the software required to make the following services highly available:

- IP addresses (the `IP_address` resource type)
- XLV logical volumes (the `volume` resource type)
- XFS file systems (the `filesystem` resource type)
- MAC addresses (the `MAC_address` resource type)

Optional software packages, known as *plug-ins* are available to make additional applications highly available. For example:

- IRIS FailSafe Oracle
- IRIS FailSafe INFORMIX
- IRIS FailSafe Netscape Web
- IRIS FailSafe Samba

Note: IRIS FailSafe NFS is not part of the core IRIS FailSafe software, but it is documented with the base release.

If you want to create new highly available services, or change the functionality of the provided failover scripts and action scripts by writing new scripts, you will use the instructions in this guide. However, not all resources can be made highly available; see "Characteristics that Permit an Application to be Highly Available".

Characteristics that Permit an Application to be Highly Available

The characteristics of an application that can be made highly available are as follows:

- The application can be easily restarted and monitored.

  It should be able to recover from failures as does most client/server software. The failure could be a hardware failure, an operating system failure, or an application failure. If a node crashed and reboots, client/server software should be able to attach again automatically.

- The application must have a start and stop procedure.

  When the resource group fails over, the resources that constitute the resource group are stopped on one node and started on another node, according to the failover script and action scripts.

- The application can be moved from one node to another after failures.

  If the resource has failed, it must still be possible to run the resource stop procedure. In addition, the resource must recover from the failed state when the resource start procedure is executed in another node.

- The application does not depend on knowing the host name; that is, those resources that can be configured to work with an IP address.

- Other resources on which the application depends can be made highly available. If they are not provided by IRIS FailSafe and its optional products (see "Highly Available Services Included with IRIS FailSafe", page 7), you must make these resources highly available, using the information in this guide.

Note: An application itself is not modified to make it highly available.

Overview of the Programming Steps

Note: If you do not want to write the scripts yourself, you can establish a contract with the Silicon Graphics Professional Services group to create customized scripts. See: `http://www.sgi.com/services/index.html`.

To make an application highly available, follow these steps:

1. Understand the application and determine:

   - The configuration required for the application, such as user names, permissions, data location (volumes), and so on. For more information about configuration, see the *IRIS FailSafe 2.0 Administrator's Guide*.

   - The other resources on which the application depends. All interdependent resources must be part of the same resource group.

   - The resource type that applies to this application.

- The number of instances of the resource type that are part of the resource group. (Each instance of a given application, or *resource type*, is a separate *resource*.) For example, a web server may depend upon two filesystem resources.
- The commands and arguments required to start, stop, and monitor the resources for this application (that is, the resources in the resource group).
- The order in which all resources in the resource group must be started and stopped.

2. Determine whether existing action scripts can be reused. If they cannot, write a new set of action scripts, using existing scripts and the templates in `/var/cluster/ha/resource_types/template` as a guide. See Chapter 2, "Writing the Action Scripts and Adding Monitoring Agents", page 21.
3. Determine whether the existing `ordered` or `round-robin` failover scripts can be reused for the resource group. If they cannot, write a new failover script. See Chapter 4, "Defining a New Resource Type", page 65.
4. Determine whether an existing resource type can be reused. If none applies, create a new resource type or modify an existing resource. See Chapter 4, "Defining a New Resource Type".
5. Configure the following in the IRIS FailSafe database (for more information, see the *IRIS FailSafe 2.0 Administrator's Guide*):
   - Resource group
   - Resource type
   - Failover policy
6. Test the action scripts and failover script. See Chapter 5, "Testing Scripts", and "Debugging Notes", page 86.

Note: Do not modify the scripts included with the IRIS FailSafe product. New or customized scripts must have different names from the files included with IRIS FailSafe.

IRIS FailSafe System Software

This section describes the software layers, communication paths, and database.

Layers

An IRIS FailSafe system has the following software layers:

- IRIS FailSafe plug-ins, which create highly available services. Some plug-ins are included with the IRIS FailSafe release, others are available for separate purchase. If the application you want is not available, you can hire the Silicon Graphics Professional Services group to develop the required software, or you can use this guide to write the software yourself.
- IRIS FailSafe base, which includes the ability to define resource groups and failover policies
- High-availability infrastructure that lets you define clusters, resources, and resource types (this consists of the `cluster_ha` installation package)
- Cluster software infrastructure, which lets you do the following:
  - Perform node logging
  - Administer the cluster
  - Define nodes

    The cluster software infrastructure consists of the `cluster_admin` and `cluster_control` subsystems).

Figure 1-1 shows a graphic representation of these layers. Table 1-2 describes the layers, which are located in the `/usr/cluster/bin` directory.

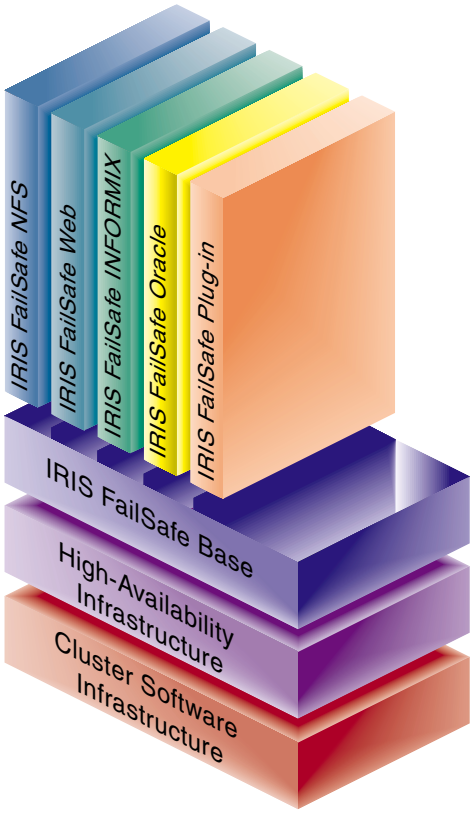

Figure 1-1 Software Layers

Table 1-2 Contents of `/usr/cluster/bin`

Layer	Subsystem	Process	Description
Plug-ins	`failsafe_informix` `failsafe2_oracle`	`ha_ifmx2`	IRIS FailSafe database agents. Each database agent monitors all instances of one type of database.
IRIS FailSafe Base	`failsafe2`	`ha_fsd`	IRIS FailSafe daemon. Provides basic component of the IRIS FailSafe software.

Layer	Subsystem	Process	Description
High-availability infrastructure	`cluster_ha`	`ha_cmsd`	Cluster membership daemon. Provides the list of nodes, called *node membership*, available to the cluster.
		`ha_gcd`	Group membership daemon. Provides group membership and reliable communication services in the presence of failures to IRIS FailSafe processes.
		`ha_srmd`	System resource manager daemon. Manages resources, resource groups, and resource types. Executes action scripts for resources.
		`ha_ifd`	Interface agent daemon. Monitors the local node's network interfaces.
Cluster software infrastructure	`cluster_admin`	`cad`	Cluster administration daemon. Provides administration services.
	`cluster_control`	`crsd`	Node control daemon. Monitors the serial connection to other nodes. Has the ability to reset other nodes.
		`cmond`	Daemon that manages all other daemons. This process starts other processes in all nodes in the cluster and restarts them on failures.
		`fs2d`	Manages the configuration database and keeps each copy in sync on all nodes in the pool

Communication Paths

The following figures show communication paths in IRIS FailSafe.

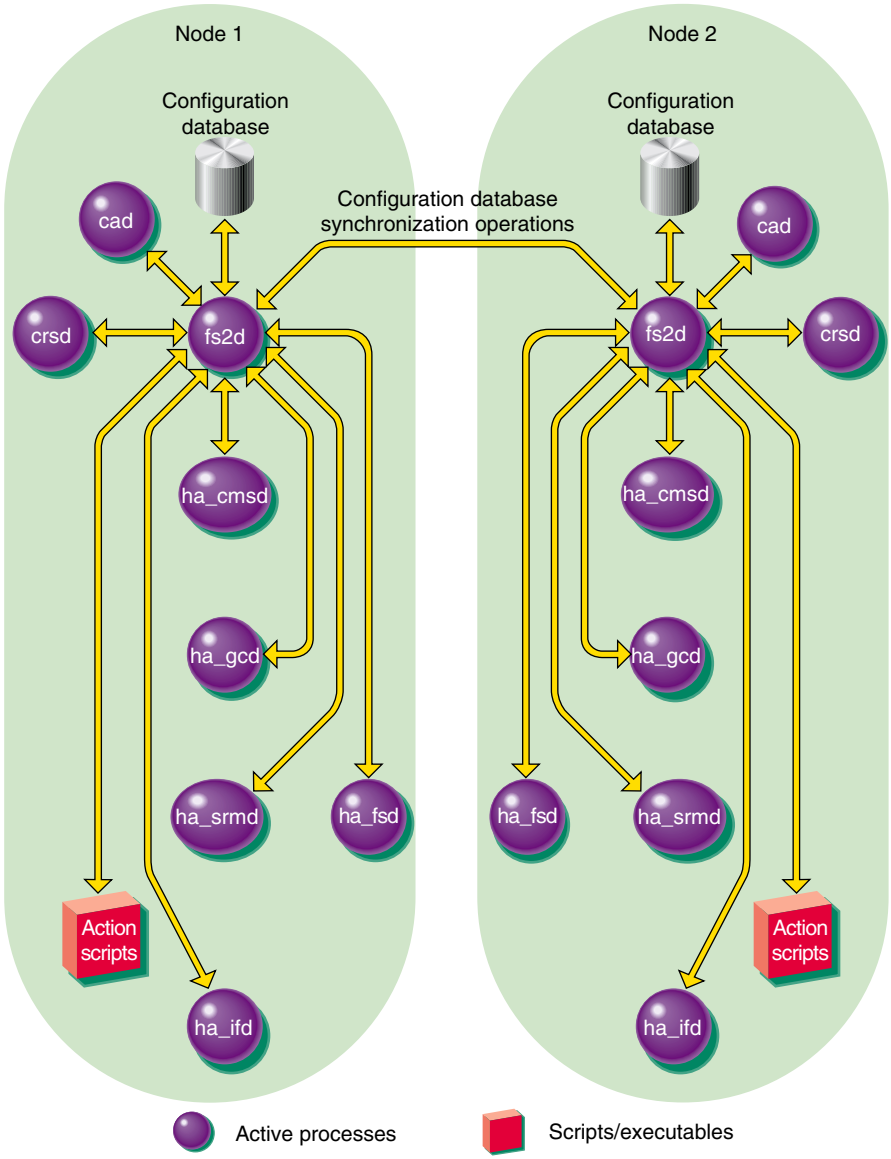

Figure 1-2 Read/Write Actions to the Configuration Database

Figure 1-3 shows the communication path for a node that is in the pool but not in a cluster.

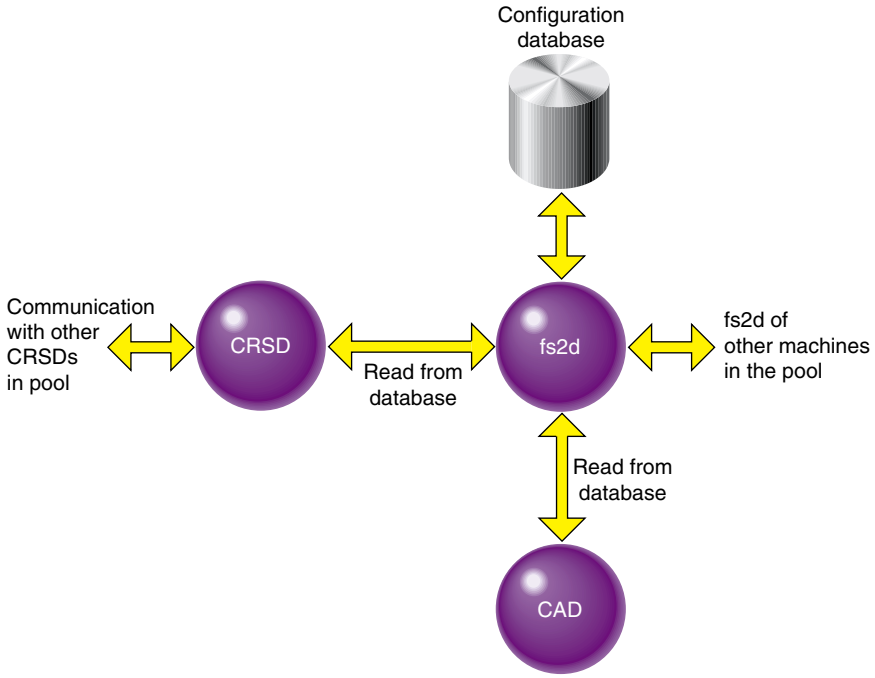

Figure 1-3 Communication Path for a Node that is Not in a Cluster

When Does FailSafe Execute Action and Failover Scripts

The order of execution is as follows:

1. IRIS FailSafe starts up and reads the resource group information from the configuration database.
2. IRIS FailSafe asks the system resource manager (SRM) to run `exclusive` scripts for the all resource groups that are in the `Online ready` state.

3. SRM returns one of the following each resource group:
   - `running`
   - `partially running`
   - `not running`
4. If a resource group has a state of `not running` in a node where HA services have been started, the following occurs:
   a. IRIS FailSafe runs the failover policy script associated with the resource group. The failover policy scripts takes the list of nodes that are capable of running the resource group (the *failover domain*) as a parameter.
   b. The failover policy script returns an ordered list of nodes in descending order of priority (the *run-time failover domain*) where the resource group can be placed.
   c. IRIS FailSafe sends a request to SRM to move the resource group to the first node in the run-time failover domain.
   d. SRM executes the `start` action script for all resources in the resource group:
      - If the `start` script fails, the resource group is marked `online` on that node with `srmd executable error` error.
      - If the `start` script is successful, SRM automatically starts monitoring those resources. After the specified start monitoring time passes, SRM executes the `monitor` action script for the resource in the resource group.
5. If state of the resource group is `running` or `partially running` on only one node in the cluster, IRIS FailSafe runs the associated failover policy script:
   - If the highest priority node is the same node where the resource group is partially running or running, the resource group is made online on the same node. In the `partially running` case, IRIS FailSafe asks SRM to execute `start` scripts for resources in the resource group that are not running.
   - If the highest priority node is a another node in the cluster, IRIS FailSafe asks SRM to execute `stop` action scripts for resources in the resource group. IRIS FailSafe makes the resource group online in the highest priority node in the cluster.

6. If the state of the resource group is `running` or `partially running` in multiple nodes in the cluster, the resource group is marked with an `error exclusivity` error. These resource groups will require operator intervention to become online in the cluster.

Figure 1-4 shows the message paths for action scripts and failover policy scripts.

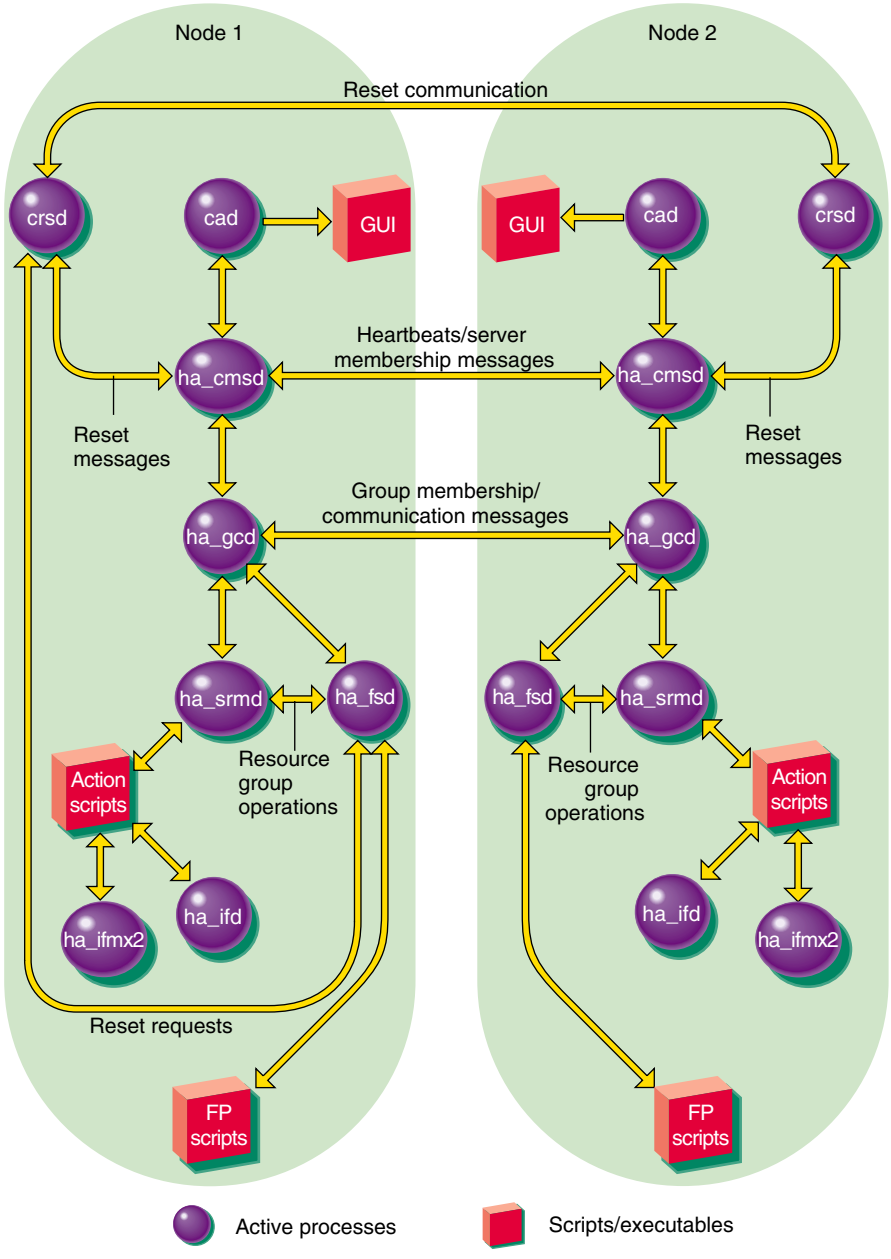

Figure 1-4 Message Paths for Action Scripts and Failover Policy Scripts

Components

The IRIS FailSafe database is a key component of IRIS FailSafe software. It contains all information about the following:

- Resources
- Resource types
- Resource groups
- Failover policies
- Nodes
- Clusters

The configuration database daemon (`fs2d`) maintains identical databases on each node in the cluster.

Table 1-3 shows the contents of the `/var/cluster/ha` directory. Table 1-4 shows the administrative commands available with IRIS FailSafe for use in scripts.

Table 1-3 Contents of `/var/cluster/ha` directory

Directory or File	Contents
`comm/`	Contains files that communicate between various daemons.
`common_scripts/`	Contains the script library (the common functions that may be used in action scripts).
`log/`	Contains the logs of all scripts and daemons executed by IRIS FailSafe. The outputs and errors from the commands within the scripts are logged in the `script_`*nodename* file.
`policies/`	Contains the failover scripts used for resource groups.
`resource_types/template`	Contains the template action scripts.
`resource_types/`*rt*	Contains the action scripts for the *rt* resource type.

Directory or File	Contents
`resource_types/`*rt* `/exclusive`	Verifies that the resource type is not already running.
`resource_types/`*rt* `/monitor`	Monitors the resource type.
`resource_types/`*rt*`/probe`	Verifies that the resource type is configured on the node.
`resource_types/`*rt*`/restart`	Restarts the resource type on the same node on a monitoring failure.
`resource_types/`*rt*`/start`	Starts the resource type.
`resource_types/`*rt*`/stop`	Stops the resource type.

Table 1-4 IRIS FailSafe Administrative Commands for Use in Scripts

Command	Purpose
`ha_cilog`	Logs messages to the `script_` *nodename* log files.
`ha_execute_lock`	Executes a command with a file lock. this is used to prevent multiple instances of the same command from executing in the node.
`ha_exec2`	Executes a command and retries the command on failure or timeout.)
`ha_filelock`	Locks a file.
`ha_fileunlock`	Unlocks a file.
`ha_ifdadmin`	Communicates with the `ha_ifd` network interface agent daemon.
`ha_http_ping2`	Checks if a web server is running.
`ha_macconfig2`	Displays or modifies MAC addresses of a network interface.

Writing the Action Scripts and Adding Monitoring Agents

This chapter provides information about writing the action scripts required to make an application highly available and how to add monitoring agents. It discusses the following topics:

Set of Action Scripts

Caution: Multiple instances of scripts may be executed at the same time. For more information, see "Execution of Action Scripts", page 22.

The following set of action scripts can be provided for each resource:

- `probe`, which verifies that the resource is configured on a node
- `exclusive`, which verifies that the resource is not already running
- `start`, which starts the resource
- `stop`, which stops the resource
- `monitor`, which monitors the resource
- `restart`, which restarts the resource on the same node when a monitoring failure occurs

The `start`, `stop`, and `exclusive` scripts are required for every resource type.

Note: The `start` and `stop` scripts must be *idempotent*; that is, they have the appearance of being run once but can in fact be run multiple times. For example, if the `start` script is run for a resource that is already started, the script must not return an error.

A `monitor` script is required, but if you wish it may contain only a return-success function. A `restart` script is required if the restart mode is set to 1; however, this script may contain only a return-success function. The `probe` script is optional.

Execution of Action Scripts

This section covers the following topics:

Multiple Instances of Script Executed at the Same Time

Multiple instances of scripts may be executed at the same time. To avoid this problem, you can use the `ha_file_lock` and `ha_execute_lock` commands to achieve sequential execution of commands in different instances of the same script.

For example, multiple instances of `xlv_assemble` should not be executed in a node at the same time. Therefore, the `start` script for `volumes` should execute `xlv_assemble` under the control of `ha_execute_lock` as follows:

```
${HA_CMDSPATH}/ha_execute_lock  30
${HA_SCRIPTTMPDIR}/lock.volume_assemble \"/sbin/xlv_assemble -l
-s${VOLUME_NAME} \"
```

The `ha_execute_lock` command takes 3 arguments:

- Number of seconds before the command times out waiting for the file lock
- File to be used for locking
- Command to be executed

The `ha_execute_lock` command tries to obtain a lock on the file every second for *timeout* seconds. After obtaining a lock on the file, it executes the command argument. On command completion, it releases lock on the file.

Differences between the `exclusive` and `monitor` Scripts

Although same check can be used in `monitor` and `exclusive` action scripts, they are used for different purposes. Table 2-1 summarizes the differences between the scripts.

Table 2-1 Differences Between the `monitor` and `exclusive` Action Scripts

`exclusive`	`monitor`
Executed in all nodes in the cluster.	Executed only on the node where the resource group (which contains the resource) is online.
Executed before the resource is started in the cluster.	Executed when the resource is online in the cluster. (The `monitor` script could degrade the services provided by the HA server. Therefore, the check performed by the `monitor` script should be lightweight and less time consuming than the check performed by the `exclusive` script))

exclusive	monitor
Executed only once before the resource group is made online in the cluster.	Executed periodically.
Failure will result in resource group not becoming online in the cluster.	Failure will cause a resource group failover to another node or a restart of the resource in the local node. An error will cause false resource group failovers in the cluster.

Successful Execution of Action Scripts

Table 2-2 shows the state of a resource group after the successful execution of an action script for every resource within a resource group. To view the state of a resource group, use the IRIS FailSafe Cluster Manager graphical user interface (GUI) or the `cluster_mgr` command.

Table 2-2 Successful Action Script Results

Event	Action Script to Execute	Resource Group State
Resource group is made online on a node	`start`	`online`
Resource group is made offline on a node	`stop`	`offline`
Online status of the resource group	`exclusive`	(No effect)
Normal monitoring of online resource group	`monitor`	`online`
Resource group monitoring failure	`restart`	`online`
Configuration verification	`probe`	(No effect)

Failure of Action Scripts

Table 2-3 shows the state of the resource group and the error state when an action script fails.

Table 2-3 Failure of an Action Script

Failing Action Script	Resource Group State	Error State
`exclusive`	`online`	`exclusivity`
`monitor`	`online`	`monitoring failure`
`probe`	(No effect)	(No effect)
`restart`	`online`	`monitoring failure`
`start`	`online`	`srmd executable error`
`stop`	`online`	`srmd executable error`

Implementing Timeouts and Retrying a Command

You can use the `ha_exec2`(1m) command to execute action scripts using timeouts. This allows the action script to be completed within the specified time, and permits proper error messages to be logged on failure or timeout. The *retry* variable is especially useful in `monitor` and `exclusive` action scripts.

To retry a command, use the following syntax:

`/usr/cluster/bin/ha_exec2` *timeout_in_seconds number_of_retries command_to_be_executed*

For example:

```
${HA_CMDSPATH}/ha_exec2 30 2 "umount /fs"
```

The above `ha_exec2` command executes the`umount /fs` command line. If the command does not complete within 30 seconds, it kills the `umount`(1m) command and retries the command. The `ha_exec2` command retries the `umount` command 2 times if it times out or fails.

For more information, see the `ha_exec2`(1m) man page.

Sending UNIX Signals

You can use the ha_exec2(1m) command to send UNIX signals to specific process. A process is identified by its the name or its arguments.

For example:

```
${HA_CMDSPATH}/ha_exec2 -s 0 -t "SYBASE_DBSERVER"
```

The above command sends signal 0 (checks if the process exists) to all processes whose name or arguments match the SYBASE_DBSERVER string. The command returns 0 if it is a success.

You should use the ha_exec2 command to check for server processes in the monitor script instead of using the ps -ef | grep command line.

For more information, see the ha_exec2(1m) man page.

Preparation

Before you can write the action scripts, you must do the following:

- Understand the scriptlib functions described in "Using the Script Library", page 101.
- Familiarize yourself with the script templates provided in the following directory: /var/cluster/ha/resource_types/template
- Read the man pages for the following commands:
  - cluster_mgr(1M)
  - fs2d(1M)
  - ha_cilog(1M)
  - ha_cmsd(1M)
  - ha_exec2(1M)
  - ha_fsd(1M)
  - ha_gcd(1M)
  - ha_ifd(1M)

  - `ha_ifdadmin`(1M)
  - `ha_macconfig2`(1M)
  - `ha_srmd`(1M)
  - `ha_statd2`(1M)
  - `haStatus`(1M)
- Familiarize yourself with the action scripts for other highly available services in `/var/cluster/ha/resource_types` that are similar to the scripts you wish to create.
- Understand how to do the following actions for your application:
  - Verify that the resource is running
  - Verify that the resource can be run
  - Start the resource
  - Stop the resource
  - Check for the server processes
  - Do a simple query as a client and understand the expected response
  - Check for configuration file or directory existence (as needed)
- Determine whether or not a monitoring script is required (see "Is Monitoring Necessary?"). If it is not, a `monitor` script is still required, but it can contain only a return-success function.
- Determine if a resource type must be added to the IRIS FailSafe database.
- Understand the vendor-supplied startup and shutdown procedures.
- Be aware of the configuration parameters for the application.
- Determine whether the resource type can be restarted in the local node. Does it make sense to do a restart of the resource type in the local node?

Is Monitoring Necessary?

In the following situations, you may not need to perform monitoring:

- Heartbeat monitoring is sufficient; that is, simply verifying that the node is alive (provided automatically by IRIS FailSafe software) determines the health of the highly available service.
- There is no process or resource that can be monitored. For example, the SGI Gauntlet Internet Firewall software performs IP filtering on firewall nodes. Because the filtering is done in the kernel, there is no process or resource to monitor.
- The resource on which the resource depends is already monitored. For example, monitoring some client-node resources might best be done by monitoring the file systems, volumes, and network interfaces they use. Because this is already done by the IRIS FailSafe base software, additional monitoring is not required.

Caution: Beware that monitoring should not be so expensive that it affects system performance. If this appears to be the case, consider a monitoring agent; see "Monitoring Agents", page 45.

Also, security issues may make monitoring difficult.

Types of Monitoring

There are two types of monitoring that may be accomplished in a `monitor` script:

- Is the resource present?
- Is the resource responding?

You can define multiple levels of monitoring within the monitor script, and the administrator can choose the desired level by configuring the resource definition in the IRIS FailSafe database. Ensure that the monitoring level chosen does not affect system performance. For more information, see the *IRIS FailSafe 2.0 Administrator's Guide*.

What are the Symptoms of Monitoring Failure?

Possible symptoms of failure include the following:

- The resource returns an error code
- The resource returns the wrong result
- The resource does not return quickly enough

How Often Should Monitoring Occur?

You must determine the probe time and time-out values for the `monitor` script. The time-out must be long enough to guarantee that occasional anomalies do not cause false failovers.

You must also determine if the `monitor` test should execute multiple times so that a node is not declared dead after a single failure. In general, testing more than once before declaring failure is a good idea.

Examples of Testing for Monitoring Failure

The test should be simple and complete quickly, whether it succeeds or fails. Some examples of tests are as follows:

- For a client-node resource that follows a protocol, the `monitor` script can make a simple request and verify that the proper response is received.
- For a web node, the `monitor` script can request a home page, verify that the connection was made, and ignore the resulting home page.
- For a database, a simple request such as querying a table can be made.
- For NFS, more complicated end-to-end monitoring is required. The test might consist of mounting an exported file system, checking access to the file system with a `stat()` system call to the root of the file system, and undoing the mount.
- For a resource that writes to a log file, check that the size of the log file is increasing or use the `grep`(1) command to check for a particular message.
- The following command can be used to determine quickly whether a process exists:

```
/sbin/killall -0 process_name
```

You can also use the `ha_exec2` command to check if a process is running.

The `ha_exec2`(8) command differs from `killall`(8) in that it performs a more exhaustive check on the process name as well as process arguments. `killall` searches for the process using the process name only. The command line is as follows:

`/usr/cluster/bin/ha_exec2 -s 0 -t` *process_name*

Note: Do not use the `ps` command to check on a particular process because its execution can be too slow.

Script Format

Templates for the action scripts are provided in the following directory:

`/var/cluster/ha/resource_types/template`

The scripts have the same general format:

- Header information
- Set local variables
- Read resource information
- Exit status
- Perform the basic action of the script, which is the customized area you must provide
- Set global variables
- Verify arguments
- Read input file

Note: Action "scripts" can be of any form – such as Bourne shell script, perl script, or C language program.

The following sections show an example from the NFS `start` script. The contents of these examples may not match the released system.

Header Information

The header information contains comments about the resource type, script type, and resource configuration format. You must modify the code as needed.

Following is the header for the NFS `start` script:

```
#!/sbin/ksh

# ****************************************************************************
# *                                                                          *
# *                  Copyright (C) 1998 Silicon Graphics, Inc.               *
# *                                                                          *
# *  These coded instructions, statements, and computer programs  contain  *
# *  unpublished  proprietary  information of Silicon Graphics, Inc., and  *
# *  are protected by Federal copyright law.  They  may  not be disclosed  *
# *  to  third  parties  or copied or duplicated in any form, in whole or  *
# *  in part, without the prior written consent of Silicon Graphics, Inc.  *
# *                                                                          *
# ****************************************************************************

#ident "$Revision: 1.4 $"

# Resource type: NFS
# Start script NFS

#
# Test resource configuration information is present in the database in
# the following format
#
# resource-type.NFS
```

Set Local Variables

The `set_local_variables()` section of the script defines all of the variables that are local to the script, such as temporary file names or database keys. All local variables should use the `LOCAL_` prefix. You must modify the code as needed.

Following is the `set_local_variables()` section from the NFS `start` script:

```
set_local_variables()
{
```

```
    LOCAL_TEST_KEY=NFS
}
```

Read Resource Information

The `get_xxx_info()` function, such as `get_nfs_info()`, reads the resource information from the database. `$1` is the test resource name. If the operation is successful, a value of 0 is returned; if the operation fails, 1 is returned.

The information is returned in the `HA_STRING` variable. For more information about `HA_STRING`, see "Using the Script Library", page 101.

Following is the `get_nfs_info()` section from the NFS `start` script

```
get_nfs_info ()
{
    ha_get_info ${LOCAL_TEST_KEY} $1
    if [ $? -ne 0 ]; then
        return 1;
    else
        return 0;
    fi
}
```

Exit Status

In the `exit_script()` function, `$1` contains the `exit_status` value. If cleanup actions are required, such as the removal of temporary files that were created as part of the process, place them before the `exit` line.

Following is the `exit_script()` section from the NFS `start` script

```
exit_script()
{
    exit $1;
}
```

Note: If you call the `exit_script` function prior to normal termination, it should be preceded by the `ha_write_status_for_resource` function and you should use the same return code that is logged to the output file.

Basic Action

This area of the script is the portion you must customize. The templates provide a minimal framework.

Following is the framework for the basic action from the `start` template:

```
start_test()
{
    # for all test resources passed as parameter
    for TEST in $HA_RES_NAMES
    do
        # HA_CMD="<command to start $TEST resource on the local machine>";
        # ha_execute_cmd "<string to describe the command being executed>";

        ha_write_status_for_resource $TEST $HA_SUCCESS;
    done
}
```

Note: When testing the script, you will add the `set -x` line to this area to obtain debugging information.

For examples of this area, see "Examples of Action Scripts", page 35.

Set Global Variables

The following lines set all of the global and local variables and store the resource names in `$HA_RES_NAMES`.

Following is the `set_global_variables()` function from the NFS `start` script:

```
set_global_variables()
{
    HA_DIR=/var/cluster/ha
    COMMON_LIB=${HA_DIR}/common_scripts/scriptlib

    # Execute the common library file
    . $COMMON_LIB

    ha_set_global_defs;
}
```

Verify Arguments

The `ha_check_arg()` function verifies the arguments and stores them in the `$HA_INFILE` and `$HA_OUTFILE` variables. It returns 1 on error and 0 on success.

Following is the `ha_check_arg ()` function from the NFS `start` script:

```
ha_check_args $*;

if [ $? -ne 0 ]; then
    exit $HA_INVAL_ARGS;
fi
```

Read Input File

The `ha_read_infile()` function reads the input file and stores the resource names in the `$HA_RES_NAMES` variable.

Following is the `ha_read_infile()` function from the NFS `start` script:

```
ha_read_infile;
```

Complete the Action

Each action script ends with the following, which performs the action and writes the output status to the `$HA_OUTFILE`:

action_resourcetype`;`

```
exit_script $HA_SUCCESS
```

Following is the completion from the NFS `start` script:

```
start_nfs;

exit_script $HA_SUCCESS;
```

Steps in Writing a Script

Caution: Multiple copies of actions scripts can execute at the same time. Therefore, all temporary file names used by the scripts can be suffixed by `PIDscript.$$` in order to make them unique, or you can use the resource name because it must be unique to the cluster.

For each script, you must do the following:

- Get the required variables
- Check the variables
- Perform the action
- Check the action

Note: The `start` and `stop` scripts are required to be *idempotent*; that is, they have the appearance of being run once but can in fact be run multiple times. For example, if the `start` script is run for a resource that is already started, the script must not return an error.

All action scripts must return the status to the `/var/cluster/ha/log/script_`*nodename* file.

Examples of Action Scripts

The following sections use portions of the NFS scripts as examples.

Note: The examples in this guide may not exactly match the released system.

`start` Script

The NFS `start` script does the following:

1. Creates a resource-specific NFS status directory.
2. Exports the specified export-point with the specified export-options.

Following is a section from the NFS `start` script:

```
# Start the resource on the local machine.
# Return HA_SUCCESS if the resource has been successfully started on the local
# machine and HA_CMD_FAILED otherwise.
#
start_nfs()
{
    # for all nfs resources passed as parameter
    for resource in ${HA_RES_NAMES}
    do
        NFSFILEDIR=${HA_SCRIPTTMPDIR}$resource
        HA_CMD="/sbin/mkdir -p $NFSFILEDIR";
        ha_execute_cmd "creating nfs status file directory";

        get_nfs_info $resource
        if [ $? -ne 0 ]; then
            ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
            exit_script $HA_CMD_FAILED;
        fi
        ha_get_field "${HA_STRING}" export-point
        if [ $? -ne 0 ]; then
            ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
            exit_script $HA_CMD_FAILED;
        fi
        export_pt="$HA_FIELD_VALUE"
        ha_get_field "${HA_STRING}" export-info
        if [ $? -ne 0 ]; then
            ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
            exit_script $HA_CMD_FAILED;
        fi
        export_opts="$HA_FIELD_VALUE"

        # Make the script idempotent, check to see if the filesystem
        # is already exported, if so return success. Remember that we
        # might not have any export options.
        retstat=0;
        # Check to see if the filesystem is already exported
        # (without options)
        /usr/etc/exportfs | grep "$export_pt$" >/dev/null 2>&1
        retstat=$?
        if [ $retstat -eq 1 ]; then
```

```
            # Check to see if the filesystem is already exported
            # with options.
            /usr/etc/exportfs | grep "$export_pt " | grep "$export_opts$"
>/dev/null
2>&1
            retstat=$?
        fi
        if [ $retstat -eq 1 ]; then
            # Before we try and export the file system, make sure
            # it exists.
            HA_CMD="/sbin/grep $export_pt /etc/mtab > /dev/null 2>&1";
            ha_execute_cmd "check if the export-point exists";
            if [ $? -eq 0 ]; then
                HA_CMD="/usr/etc/exportfs -i -o $export_opts $export_pt";
                ha_execute_cmd "export $export_pt directories to NFS clients";
                if [ $? -ne 0 ]; then
                    ha_write_status_for_resource ${resource} ${HA_CMD_FAILED};
                else
                    ha_write_status_for_resource ${resource} ${HA_SUCCESS};
                fi
            else
                ${HA_LOG} "Failed to find filesystem $export_pt"
                ha_write_status_for_resource ${resource} ${HA_CMD_FAILED};
            fi
        else
            ha_write_status_for_resource ${resource} ${HA_SUCCESS};
        fi
    done
}
```

`stop` Script

The NFS `stop` script does the following:

1. Unexports the specified export-point.
2. Removes the NFS status directory.

Following is an example from the NFS `stop` script:

```
# Stop the nfs resource on the local machine.
# Return HA_SUCCESS if the resource has been successfully stopped on the local
```

```
# machine and HA_CMD_FAILED otherwise.
#
stop_nfs()
{
    # for all nfs resources passed as parameter
    for resource in ${HA_RES_NAMES}
    do
        get_nfs_info $resource
        if [ $? -ne 0 ]; then
            # NFS resource information not available.
            ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
            exit_script $HA_CMD_FAILED;
        fi
        ha_get_field "${HA_STRING}" export-point
        if [ $? -ne 0 ]; then
            # NFS export-point not available.
            ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
            exit_script $HA_CMD_FAILED;
        fi
        export_pt="$HA_FIELD_VALUE"

        # Make the script idempotent, check to see if the filesystem
        # is already exported, if so return success. Remember that we
        # might not have any export options.
        retstat=0;
        # Check to see if the filesystem is already exported
        # (without options)
        /usr/etc/exportfs | grep "$export_pt$" >/dev/null 2>&1
        retstat=$?
        if [ $retstat -eq 1 ]; then
            # Check to see if the filesystem is already exported
            # with options.
            /usr/etc/exportfs | grep "$export_pt " | grep "$export_opts$"
>/dev/null
2>&1
            retstat=$?
        fi
        if [ $retstat -eq 0 ]; then
            # Before we unexport the filesystem, check that it exists
            HA_CMD="/sbin/grep $export_pt /etc/mtab > /dev/null 2>&1";
            ha_execute_cmd "check if the export-point exists";
```

```
            if [ $? -eq 0 ]; then
                HA_CMD="/usr/etc/exportfs -u $export_pt";
                ha_execute_cmd "unexport $export_pt directories to NFS clients";
                if [ $? -ne 0 ]; then
                    ha_write_status_for_resource ${resource} ${HA_CMD_FAILED};
                else
                    ha_write_status_for_resource ${resource} ${HA_SUCCESS};
                fi
            else
                ${HA_LOG} "filesystem $export_pt not found in export filesystem
list, unexporting anyway";
                HA_CMD="/usr/etc/exportfs -u $export_pt";
                ha_execute_cmd "unexport $export_pt directories to NFS clients";
                ha_write_status_for_resource ${resource} ${HA_SUCCESS};
            fi
        else
            ha_write_status_for_resource ${resource} ${HA_SUCCESS};
        fi
        # remove the monitor nfs status file
        NFSFILEDIR=${HA_SCRIPTTMPDIR}$resource
        HA_CMD="/sbin/rm -rf $NFSFILEDIR";
        ha_execute_cmd "removing nfs status file directory";
    done
}
```

`probe` Script

The NFS `probe` script does the following:

1. Verifies that the NFS daemons are running.
2. Verifies that the file system is present.

Following is an example from the NFS `probe` script:

```
# Check if the nfs resource can be online on this node. Verify if the
# database configuration is correct for the resource.
# Return HA_SUCCESS if the resource can be online on the local node
# and HA_CMD_FAILED otherwise.
#
probe_nfs()
{
```

```
    # for all nfs resources passed as parameter
    for resource in ${HA_RES_NAMES}
    do
        HA_CMD="/sbin/killall -0 nfsd"
        ha_execute_cmd "checking for nsfd processes"
        if [ $? -ne 0 ]; then
            ha_write_status_for_resource ${resource} ${HA_CMD_FAILED};
            exit_script $HA_CMD_FAILED;
        fi
        get_nfs_info $resource
        if [ $? -ne 0 ]; then
            ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
            exit_script $HA_CMD_FAILED;
        fi
        ha_get_field "${HA_STRING}" filesystem
        if [ $? -ne 0 ]; then
            ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
            exit_script $HA_CMD_FAILED;
        fi
        fs="$HA_FIELD_VALUE"
        # Check if the file system is present
        if [ -b $fs ]; then
            ha_write_status_for_resource ${resource} ${HA_SUCCESS};
        else
            ha_write_status_for_resource ${resource} ${HA_CMD_FAILED};
        fi
    done
}
```

`monitor` Script

The NFS `monitor` script does the following:

1. Verifies that the file system is mounted at the correct mount point.
2. Requests the status of the exported file system.
3. Checks the export-point.
4. Requests NFS statistics and (based on the results) make a Remote Procedure Call (RPC) to NFS as needed.

Following is an example from the NFS `monitor` script:

```
# Check if the nfs resource is allocated in the local node
# This check must be light weight and less intrusive compared to
# exclusive check. This check is done when the resource has been
# allocated in the local node.
# Return HA_SUCCESS if the resource is running in the local node
# and HA_CMD_FAILED if the resource is not running in the local node
# The list of the resources passed as input is in variable
# $HA_RES_NAMES
#
monitor_nfs()
{
    for resource in ${HA_RES_NAMES}
    do
        get_nfs_info $resource
        if [ $? -ne 0 ]; then
            # No resource information available.
            ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
            exit_script $HA_CMD_FAILED;
        fi
        ha_get_field "${HA_STRING}" export-point
        if [ $? -ne 0 ]; then
            # NFS export-point not available available.
            ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
            exit_script $HA_CMD_FAILED;
        fi
        export_pt="$HA_FIELD_VALUE";
        ha_get_field "${HA_STRING}" filesystem
        if [ $? -ne 0 ]; then
            # filesystem not available available.
            ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
            exit_script $HA_CMD_FAILED;
        fi
        fs="$HA_FIELD_VALUE";
        # Check to see if the filesystem is mounted
        HA_CMD="/sbin/mount | grep $fs | grep $export_pt >> /dev/null 2>&1"
        ha_execute_cmd "check to see if $export_pt is mounted on $fs"
        if [ $? -ne 0 ]; then
            ha_write_status_for_resource ${resource} ${HA_CMD_FAILED};
            exit_script $HA_CMD_FAILED;
        fi
```

```
# stat the filesystem
HA_CMD="/sbin/stat $export_pt";
ha_execute_cmd "stat mount point $export_pt"
if [ $? -ne 0 ]; then
    ha_write_status_for_resource ${resource} ${HA_CMD_FAILED};
    exit_script $HA_CMD_FAILED;
fi

# check the filesystem is exported
EXPORTFS="${HA_SCRIPTTMPDIR}/exportfs.$$"
/usr/etc/exportfs > $EXPORTFS 2>&1
HA_CMD="awk '{print \$1}' $EXPORTFS | grep $export_pt"
ha_execute_cmd " check the filesystem $export_pt is exported"
if [ $? -ne 0 ]; then
    ${HA_LOG} "failed to find $export_pt in exported filesystem list:-"
    ${HA_LOG} "/sbin/cat ${EXPORTFS}"
    ha_write_status_for_resource ${resource} ${HA_CMD_FAILED};
    rm -f $EXPORTFS
    exit_script $HA_CMD_FAILED;
fi
rm -f $EXPORTFS
# create a file to hold the nfs stats. This will will be
# deleted in the stop script.
NFSFILE=${HA_SCRIPTTMPDIR}$resource/.nfsstat
NFS_STAT=nfsstat -rs | tail -1 | awk '{print $1}'
if [ ! -f $NFSFILE ]; then
   echo $NFS_STAT > $NFSFILE;
   if [ $NFS_STAT -eq 0 ];then
      # do some rpcinfo's
      exec_rpcinfo;
      if [ $? -ne 0 ]; then
         ha_write_status_for_resource ${resource} ${HA_CMD_FAILED};
         exit_script $HA_CMD_FAILED;
      fi
   fi
else
   OLD_STAT=/sbin/cat $NFSFILE
   if [ $NFS_STAT -gt $OLD_STAT ]; then
        echo $NFS_STAT > $NFSFILE;
   else
        echo $NFS_STAT > $NFSFILE;
```

```
                exec_rpcinfo;
                if [ $? -ne 0 ]; then
                   ha_write_status_for_resource $resource ${HA_CMD_FAILED};
                   exit_script $HA_CMD_FAILED;
                fi
            fi
        fi
        ha_write_status_for_resource $resource $HA_SUCCESS;
    done
}
```

`exclusive` Script

The NFS `exclusive` script determines whether the file system is already exported. The check made by an exclusive script can be more expensive than a monitor check. IRIS FailSafe uses this script to determine if resources are running on a node in the cluster, and to thereby prevent starting resources on multiple nodes in the cluster.

Following is an example from the NFS `exclusive` script:

```
# Check if the nfs resource is running in the local node. This check can
# more intrusive than the monitor check. This check is used to determine
# if the resource has to be started on a machine in the cluster.
# Return HA_NOT_RUNNING if the resource is not running in the local node
# and HA_RUNNING if the  resource is running in the local node
# The list of nfs resources passed as input is in variable
# $HA_RES_NAMES
#
exclusive_nfs()
{
    # for all resources passed as parameter
    for resource in ${HA_RES_NAMES}
    do
        get_nfs_info $resource
        if [ $? -ne 0 ]; then
            # No resource information available
            ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
            exit_script $HA_CMD_FAILED;
        fi
        ha_get_field "${HA_STRING}" export-point
        if [ $? -ne 0 ]; then
```

```
            ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
            exit_script $HA_CMD_FAILED;
        fi
        export_pt="$HA_FIELD_VALUE";
        SMFILE=${HA_SCRIPTTMPDIR}/showmount.$$
        /etc/showmount -x >> ${SMFILE};
        HA_CMD="/sbin/grep $export_pt ${SMFILE} >> /dev/null 2>&1"
        ha_execute_cmd "checking for $export_pt exported directory"
        if [ $? -eq 0 ];then
            ha_write_status_for_resource ${resource} ${HA_RUNNING};
            ha_print_exclusive_status ${resource} ${HA_RUNNING};
        else
            ha_write_status_for_resource ${resource} ${HA_NOT_RUNNING};
            ha_print_exclusive_status ${resource} ${HA_NOT_RUNNING};
        fi
        rm -f ${SMFILE}
    done
}
```

`restart` Script

The NFS `restart` script exports the specified export-point with the specified export-options.

Following is an example from the `restart` script for NFS:

```
# Restart nfs resource
# Return HA_SUCCESS if nfs resource failed over successfully or
# return HA_CMD_FAILED if nfs resource could not be failed over locally.
# Return HA_NOT_SUPPORTED if local restart is not supported for nfs
# resource type.
# The list of nfs resources passed as input is in variable
# $HA_RES_NAMES
#
restart_nfs()
{
    # for all nfs resources passed as parameter
    for resource in ${HA_RES_NAMES}
    do
        get_nfs_info $resource
        if [ $? -ne 0 ]; then
```

```
            ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
            exit_script $HA_CMD_FAILED
        fi
        ha_get_field "${HA_STRING}" export-point
        if [ $? -ne 0 ]; then
            ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
            exit_script $HA_CMD_FAILED
        fi
        export_pt="$HA_FIELD_VALUE"
        ha_get_field "${HA_STRING}" export-info
        if [ $? -ne 0 ]; then
            ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
            exit_script $HA_CMD_FAILED
        fi
        export_opts="$HA_FIELD_VALUE"

        HA_CMD="/usr/etc/exportfs -i -o $export_opts $export_pt";
        ha_execute_cmd "export $export_pt directories to NFS clients";
        if [ $? -ne 0 ]; then
           ha_write_status_for_resource ${resource} ${HA_CMD_FAILED};
        else
           ha_write_status_for_resource ${resource} ${HA_SUCCESS};
        fi
    done
}
```

Monitoring Agents

If resources cannot be monitored using a lightweight check, you should use a *monitoring agent*. The `monitor` action script contacts the monitoring agent to determine the status of the resource in the node. The monitoring agent in turn periodically monitors the resource. Figure 2-1 shows the monitoring process.

Figure 2-1 Monitoring Process

Monitoring agents are useful for monitoring database resources. In databases, creating the database connection is costly and time consuming. The monitoring agent maintains connections to the database and it queries the database using the connection in response the `monitor` action script request.

Monitoring agents are independent processes and can be started by `cmond` process, although this is not required. For example, if a monitoring agent must be started when activating highly available services on a node, information about that agent can be added to the `cmond` configuration on that node. The `cmond` configuration is located in the `/var/cluster/cmon/process_groups` directory. Information about different agents should go into different files. The name of the file is not relevant to the activate/deactivate procedure.

If a monitoring agent exits or aborts, `cmond` will automatically restart monitoring agent. This prevents `monitor` action script failures due to monitoring agent failures.

For example, the `/var/cluster/cmon/process_groups/ip_addresses` file contains information about the `ha_ifd` process that monitors network interfaces. It contains the following:

```
TYPE = cluster_agent
PROCS = ha_ifd
ACTIONS = start stop restart attach detach
AUTOACTION = attach
```

If you create a new monitoring agent, you must also a create a corresponding file in the `/var/cluster/cmon/process_groups` directory that contains similar information about the new agent. To do this, you can copy the `ip_addresses` file and modify the `PROCS` line so that it lists the processes that constitute your new agent. These processes must be located in the `/usr/cluster/bin` directory. You should not modify the other configuration lines (`TYPE`, `ACTIONS`, and `AUTOACTION`).

Suppose you need to add a new agent called `newagent` that consists of processes `ha_x` and `ha_y`. The configuration information for this agent will be located in the `/var/cluster/cmon/process_groups/newagent` file, which will contain the following:

```
TYPE = cluster_agent
PROCS = ha_x ha_y
ACTIONS = start stop restart attach detach
AUTOACTION = attach
```

In this case, the software will expect two executables (`/usr/cluster/bin/ha_x` and `/usr/cluster/bin/ha_y`) to be present.

Creating a Failover Policy

This chapter tells you how to create a failover policy.

Contents of a Failover Policy

A *failover policy* is the method by which a resource group is failed over from one node to another. A failover policy consists of the following:

- *Failover domain*
- *Failover attributes*
- *Failover scripts*

IRIS FailSafe uses the failover domain output from a failover script along with failover attributes to determine on which node a resource group should reside.

The administrator must configure a failover policy for each resource group. The name of the failover policy must be unique within the *pool*.

Failover Domain

A *failover domain* is the `ordered` list of nodes on which a given *resource group* can be allocated. The nodes listed in the failover domain `must` be within the same cluster; however, the failover domain does not have to include every node in the cluster. The failover domain can be also used to statically load balance the resource groups in a cluster.

Examples:

- In a four–node cluster, a set of two nodes that have access to a particular XLV volume may be the failover domain of the resource group containing that XLV volume.
- If you have a cluster of nodes named venus, mercury, and pluto, you could configure the following initial failover domains for resource groups RG1 and RG2:
  - mercury, venus, pluto for RG1
  - pluto, mercury for RG2

The administrator defines the *initial failover domain* when configuring a failover policy. The initial failover domain is used when a cluster is first booted. The ordered list specified by the initial failover domain is transformed into a *run-time failover domain* by the *failover script*. With each failure, the failover script takes the current run-time failover domain and potentially modifies it; the initial failover domain is never used again. Depending on the run-time conditions and contents of the failover script, the initial and run-time failover domains may be identical.

IRIS FailSafe stores the run-time failover domain and uses it as input to the next failover script invocation.

Failover Attributes

A *failover attribute* is a value that is passed to the *failover script* and used by IRIS FailSafe for the purpose of modifying the *run-time failover domain* used for a specific resource group. There are required and optional failover attributes, and you can also specify your own strings as attributes.

Table 3-1 shows the required failover attributes.

Table 3-1 Required Failover Attributes (mutually exclusive)

Name	Description
`Auto_Failback`	Specifies that the resource group will be run on the first available node in the run-time failover domain. If the first node fails, the next available node will be used; when the first node reboots, the resource group will return to it. This attribute is best used when some type of load balancing is required. You must specify either this attribute or the `Controlled_Failback` attribute.
`Controlled_Failback`	Specifies that the resource group will be run on the first available node in the run-time failover domain, and will remain running on that node until it fails. If the first node fails, the next available node will be used; the resource group will remain on this new node even after the first node reboots. This attribute is best used when client/server applications have expensive recovery mechanisms, such as databases or any application that uses `tcp` to communicate. You must specify either this attribute or the `Auto_Failback` attribute.

When defining a failover policy, you can choose one of the recovery attributes shown in Table 3-2. The recovery attribute determines the node on which a resource group will be allocated when its state changes to online and a member of the group is already allocated (such as when volumes are present).

Table 3-2 Optional Failover Attributes (mutually exclusive)

Name	Description
`Auto_Recovery`	Specifies that the failover policy will be used to allocate the resource group. This attribute is optional and is mutually exclusive with the `InPlace_Recovery` attribute. If you specify neither of these attributes, IRIS FailSafe will use this attribute by default if you have specified the `Auto_Failback` attribute.
`InPlace_Recovery`	Specifies that the resource group will be allocated on the node that already contains part of the resource group. This attribute is optional and is mutually exclusive with the `Auto_Recovery` attribute. If you specify neither of these attributes, IRIS FailSafe will use this attribute by default if you have specified the `Controlled_Failback` attribute.

Failover Scripts

A *failover script* generates the run-time failover domain and returns it to the IRIS FailSafe process. The IRIS FailSafe process applies the failover attributes and then selects the first node in the returned failover domain that is also in the current node membership.

Note: The run-time of the failover script must be capped to a system-definable maximum. Hence, any external calls must be guaranteed to return quickly. If the failover script takes too long to return, IRIS FailSafe will kill the script process and use the previous run-time failover domain.

Failover scripts are stored in the `/var/clusters/ha/policies` directory.

The `ordered` Failover Script

The `ordered` failover script is provided with the IRIS FailSafe release. The `ordered` script never changes the initial domain; when using this script, the initial and run-time domains are equivalent. The script reads six lines from the input file and in case of errors logs the input parameters and/or the error to the script log.

The following example shows the contents of the `ordered` failover script.

```
#!/sbin/ksh
#
# $1 - input file
# $2 - output file
#
# line 1 input file - version
# line 2 input file - name
# line 3 input file - owner field
# line 4 input file - attributes
# line 5 input file - list of possible owners
# line 6 input file - application failover domain


DIR=/usr/cluster/bin
LOG=${DIR}/ha_cilog -g ha_script -s script
FILE=/var/cluster/ha/policies/ordered

input=$1
output=$2
cat ${input} | read version
head -2 ${input} | tail -1 | read name
head -3 ${input} | tail -1 | read owner
head -4 ${input} | tail -1 | read attr
head -5 ${input} | tail -1 | read mem1 mem2 mem3 mem4 mem5 mem6 mem7 mem8
head -6 ${input} | tail -1 | read afd1 afd2 afd3 afd4 afd5 afd6 afd7 afd8



${LOG} -l 1 "${FILE}:" `/bin/cat ${input}`

if [ "${version}" -ne 1 ] ; then
    ${LOG} -l 1 "ERROR: ${FILE}: Different version no. Should be (1) rather than
(${version})" ;
    exit 1;
elif [ -z "${name}" ]; then
    ${LOG} -l 1 "ERROR: ${FILE}: Failover script not defined";
    exit 1;
elif [ -z "${attr}" ]; then
    ${LOG} -l 1 "ERROR: ${FILE}: Attributes not defined";
    exit 1;
```

```
elif [ -z "${mem1}" ]; then
    ${LOG} -l 1 "ERROR: ${FILE}: No node membership defined";
    exit 1;
elif [ -z "${afd1}" ]; then
    ${LOG} -l 1 "ERROR: ${FILE}: No failover domain defined";
    exit 1;
fi


found=0
for i in $afd1 $afd2 $afd3 $afd4 $afd5 $afd6 $afd7 $afd8; do
    for j in $mem1 $mem2 $mem3 $mem4 $mem5 $mem6 $mem7 $mem8; do
        if [ "X${j}" = "X${i}" ]; then
            found=1;
            break;
        fi
    done
done


if [ ${found} -eq 0 ]; then
    mem="("$mem1")"" ""("$mem2")"" ""("$mem3")"" ""("$mem4")"" ""("$mem5")""
""("$mem6")"" ""("$mem7")"" ""("$mem8")";
    afd="("$afd1")"" ""("$afd2")"" ""("$afd3")"" ""("$afd4")"" ""("$afd5")""
""("$afd6")"" ""("$afd7")"" ""("$afd8")";
    ${LOG} -l 1 "ERROR: ${FILE}: Policy script failed"
    ${LOG} -l 1 "ERROR: ${FILE}: " `/bin/cat ${input}`
    ${LOG} -l 1 "ERROR: ${FILE}: Nodes defined in membership do not match the
ones in failure domain"
    ${LOG} -l 1 "ERROR: ${FILE}: Parameters read from input file: version =
$version, name = $name, owner = $owner,  attribute = $attr, nodes = $mem, afd = $afd"
    exit 1;
fi


if [ ${found} -eq 1 ]; then
    rm -f ${output}
    echo $afd1 $afd2 $afd3 $afd4 $afd5 $afd6 $afd7 $afd8 > ${output}
    exit 0
fi
exit 1
```

The `round-robin` Failover Script

The `round-robin` script selects the resource group owner in a round-robin (circular) fashion. This policy can be used for resource groups that can be run in any node in the cluster.

The following example shows the contents of the `round-robin` failover script.

```
#!/sbin/ksh
#
# $1 - input file
# $2 - output file
#
# line 1 input file - version
# line 2 input file - name
# line 3 input file - owner field
# line 4 input file - attributes
# line 5 input file - Possible list of owners
# line 6 input file - application failover domain


DIR=/usr/cluster/bin
LOG=${DIR}/ha_cilog -g ha_script -s script
FILE=/var/cluster/ha/policies/round-robin

# Read input file
input=$1
output=$2
cat ${input} | read version
head -2 ${input} | tail -1 | read name
head -3 ${input} | tail -1 | read owner
head -4 ${input} | tail -1 | read attr
head -5 ${input} | tail -1 | read mem1 mem2 mem3 mem4 mem5 mem6 mem7 mem8
head -6 ${input} | tail -1 | read afd1 afd2 afd3 afd4 afd5 afd6 afd7 afd8

# Validate input file
${LOG} -l 1 "${FILE}:" `/bin/cat ${input}`

if [ "${version}" -ne 1 ] ; then
    ${LOG} -l 1 "ERROR: ${FILE}: Different version no. Should be (1) rather than
(${version})" ;
    exit 1;
```

```
elif [ -z "${name}" ]; then
    ${LOG} -l 1 "ERROR: ${FILE}: Failover script not defined";
    exit 1;
elif [ -z "${attr}" ]; then
    ${LOG} -l 1 "ERROR: ${FILE}: Attributes not defined";
    exit 1;
elif [ -z "${mem1}" ]; then
    ${LOG} -l 1 "ERROR: ${FILE}: No node membership defined";
    exit 1;
elif [ -z "${afd1}" ]; then
    ${LOG} -l 1 "ERROR: ${FILE}: No failover domain defined";
    exit 1;
fi


# Return 0 if $1 is in the membership and return 1 otherwise.
check_in_mem()
{
    for j in $mem1 $mem2 $mem3 $mem4 $mem5 $mem6 $mem7 $mem8; do
        if [ "X${j}" = "X$1" ]; then
            return 0;
        fi
    done
    return 1;
}

# Check if owner has to be changed. There is no need to change owner if
# owner node is in the possible list of owners.
check_in_mem ${owner}
if [ $? -eq 0 ]; then
    nextowner=${owner};
fi

# Search for the next owner
if [ "X${nextowner}" = "X" ]; then
    next=0;
    for i in $afd1 $afd2 $afd3 $afd4 $afd5 $afd6 $afd7 $afd8; do
        if [ "X${i}" = "X${owner}" ]; then
            next=1;
            continue;
        fi
```

```
        if [ "X${owner}" = "XNO ONE" ]; then
            next=1;
        fi

        if [ ${next} -eq 1 ]; then
            # Check if ${i} is in membership
            check_in_mem ${i};
            if [ $? -eq 0 ]; then
                # found next owner
                nextowner=${i};
                next=0;
                break;
            fi
        fi
    done
fi

if [ "X${nextowner}" = "X" ]; then
    # wrap round the afd list.
    for i in $afd1 $afd2 $afd3 $afd4 $afd5 $afd6 $afd7 $afd8; do
        if [ "X${i}" = "X${owner}" ]; then
            # Search for next owner complete
            break;
        fi

        # Previous loop should have found new owner
        if [ "X${owner}" = "XNO ONE" ]; then
            break;
        fi

        if [ ${next} -eq 1 ]; then
            check_in_mem ${i};
            if [ $? -eq 0 ]; then
                # found next owner
                nextowner=${i};
                next=0;
                break;
            fi
        fi
    done
```

```
fi

if [ "X${nextowner}" = "X" ]; then
    ${LOG} -l 1 "ERROR: ${FILE}: Policy script failed"
    ${LOG} -l 1 "ERROR: ${FILE}: " `/bin/cat ${input}`
    ${LOG} -l 1 "ERROR: ${FILE}: Could not find new owner"
    exit 1;
fi


# nextowner is the new owner
print=0;
rm -f ${output};

# Print the new afd to the output file
echo -n "${nextowner} " > ${output};
for i in $afd1 $afd2 $afd3 $afd4 $afd5 $afd6 $afd7 $afd8;
do
    if [ "X${nextowner}" = "X${i}" ]; then
        print=1;
    elif [ ${print} -eq 1 ]; then
        echo -n "${i} " >> ${output}
    fi
done

print=1;
for i in $afd1 $afd2 $afd3 $afd4 $afd5 $afd6 $afd7 $afd8; do
    if [ "X${nextowner}" = "X${i}" ]; then
        print=0;
    elif [ ${print} -eq 1 ]; then
        echo -n "${i} " >> ${output}
    fi
done

echo >> ${output};
exit 0;
```

Creating a New Failover Script

If the `ordered` or `round-robin` scripts do not meet your needs, you can create a new failover script and place it in the `/var/clusters/ha/policies` directory. You can then configure the IRIS FailSafe database to use your new failover script for the required resource groups.

Failover Script Interface

The following is passed to the failover script:

`function(`*version*`,` *name*`,` *owner*`,` *attributes*`,` *possibleowners*`,` *domain*`)`

version	IRIS FailSafe version. The IRIS FailSafe 2.0 release uses version number 1.
name	Name of the failover script (used for error validations and logging purposes).
owner	Logical name of the node that has the resource group allocated.
attributes	Failover attributes (`Auto_Failback` or `Controlled_Failback` must be included)
possibleowners	List of possible owners for the resource group. This list can be subset of the current node membership.
domain	Ordered list of nodes used at the last failover. (At the first failover, the initial failover domain is used.)

The failover script returns the newly generated run-time failover domain to IRIS FailSafe, which then chooses the node on which the resource group should be allocated by applying the failover attributes and node membership to the run-time failover domain.

Example Failover Policies

There are two general types of configuration, each of which can have from 2 through 8 nodes:

- *N* nodes that can potentially failover their applications to any of the other nodes in the cluster.
- *N* primary nodes that can failover to *M* backup nodes. For example, you could have 3 primary nodes and 1 backup node.

This section shows examples of failover policies for the following types of configuration, each of which can have from 2 through 8 nodes:

- *N* primary nodes and one backup node (*N*+1)
- *N* primary nodes and two backup nodes (*N*+2)
- *N* primary nodes and *M* backup nodes (*N*+*M*)

Note: The diagrams in the following sections illustrate the configuration concepts discussed here, but they do not address all required or supported elements, such as reset hubs. For configuration details, see the *IRIS FailSafe 2.0 Installation and Maintenance Instructions*.

N+1 Configuration

Figure 3-1 shows a specific instance of an *N*+1 configuration in which there are three primary nodes and one backup node. (This is also known as a *star configuration*.) The disks shown could each be disk farms.

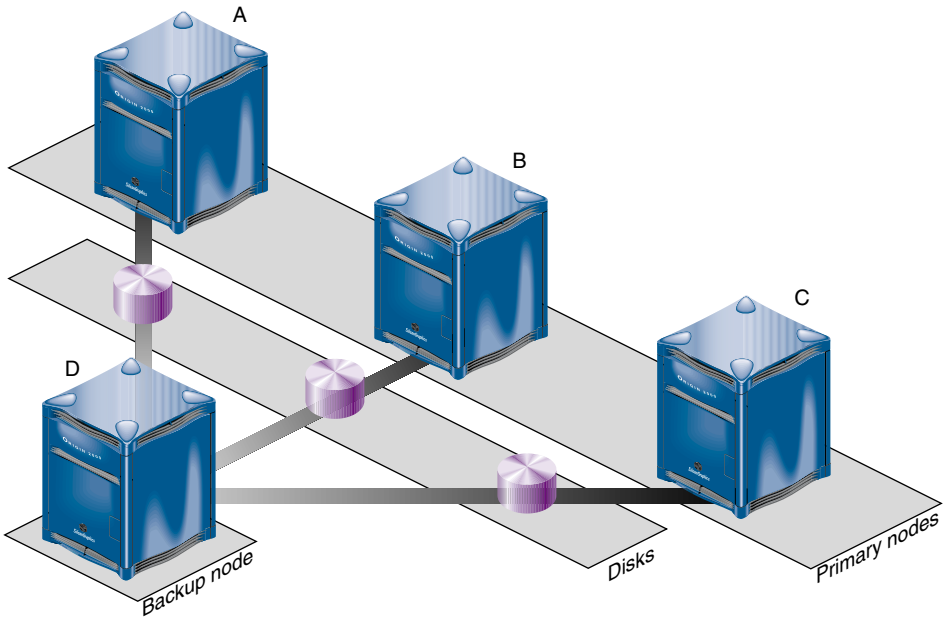

Figure 3-1 *N*+1 Configuration Concept

You could configure the following failover policies for load balancing:

- Failover policy for RG1:
  - Initial failover domain = A, D
  - Failover attribute = `Auto_Failback`
  - Failover script = `ordered`
- Failover policy for RG2:
  - Initial failover domain = B, D
  - Failover attribute = `Auto_Failback`
  - Failover script = `ordered`

- Failover policy for RG3:
  - Initial failover domain = C, D
  - Failover attribute = `Auto_Failback`
  - Failover script = `ordered`

If node A fails, RG1 will fail over to node D. As soon as node A reboots, RG1 will be moved back to node A.

If you change the failover attribute to `Controlled_Failback` for RG1 and node A fails, RG1 will fail over to node D and will remain running on node D even if node A reboots.

N+2 Configuration

Figure 3-2 shows a specific instance of an *N*+2 configuration in which there are four primary nodes and two backup nodes. The disks shown could each be disk farms.

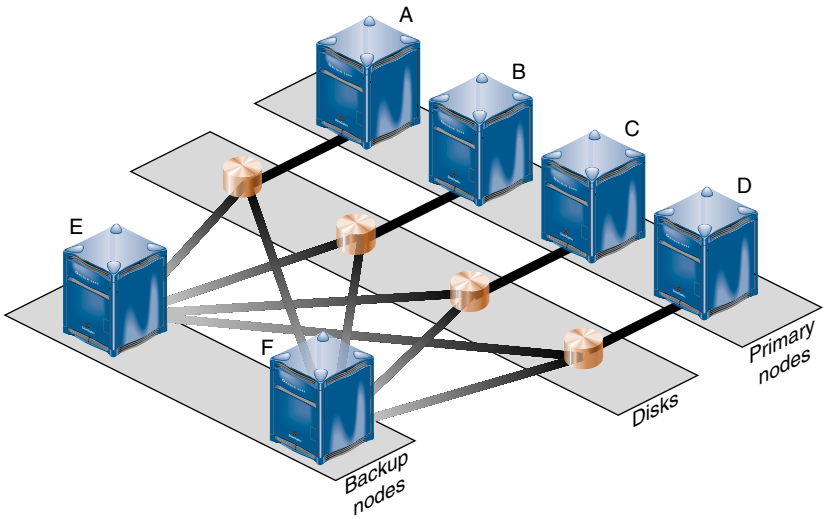

Figure 3-2 *N*+2 Configuration Concept

You could configure the following failover policy for a database resource groups RG7 and RG8:

- Failover policy for RG7:
  - Initial failover domain = A, E, F
  - Failover attribute = `Controlled_Failback`
  - Failover script = `ordered`
- Failover policy for RG8:
  - Initial failover domain = B, F, E
  - Failover attribute = `Auto_Failback`
  - Failover script = `ordered`

If node A fails, RG7 will fail over to node E. If node E also fails, RG7 will fail over to node F. If A is rebooted, RG7 will remain on node F.

If node B fails, RG8 will fail over to node F. If B is rebooted, RG8 will return to node B.

N+M Configuration

Figure 3-3 shows a specific instance of an *N+M* configuration in which there are four primary nodes and each can serve as a backup node. The disk shown could be a disk farm.

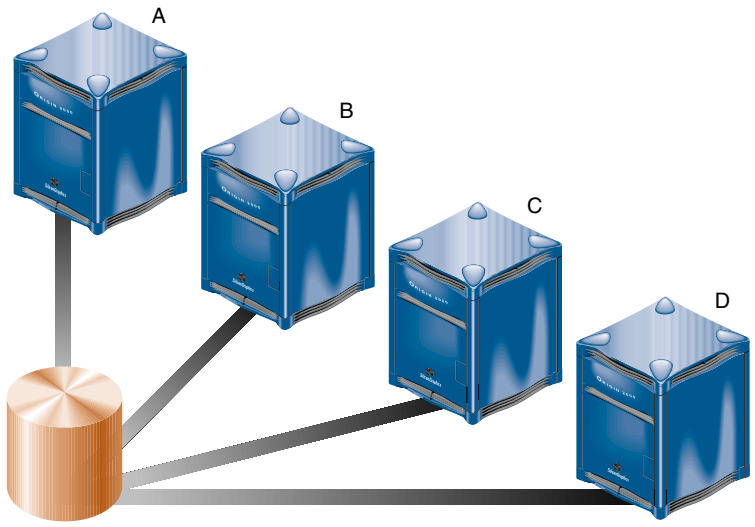

Figure 3-3 *N+M* Configuration Concept

You could configure the following failover policy for a database resource groups RG5 and RG6:

- Failover policy for RG5:
  - Initial failover domain = A, B, C, D
  - Failover attribute = `Controlled_Failback`
  - Failover script = `ordered`
- Failover policy for RG6:
  - Initial failover domain = C, A, D
  - Failover attribute = `Controlled_Failback`
  - Failover script = `ordered`

If node C fails, RG6 will fail over to node A. When node C reboots, RG6 will remain running on node A. If node A then fails, RG6 will return to node C and RG5 will move to node B. If node B then fails, RG5 moves to node C.

Defining a New Resource Type

This chapter tells you how to define a new resource type:

- "Using the GUI", page 68
- "Using `cluster_mgr` Interactively", page 75
- "Using `cluster_mgr` With a Script", page 79

It also tells you how to test the results in "Testing a New Resource Type", page 82.

To define a new resource type, you must have the following information:

- Name of the resource type. The name can consist of alphanumeric characters and any of the following:

  ```
  - (hyphen)
  _ (underscore)
  /
  .
  :
  "
  =
  @
  ,
  ```

  The name cannot contain a space, an unprintable character, or any of the following characters:

  ```
  *
  ?
  \
  #
  ```

- Name of the cluster to which the resource type will apply.
- If the resource type is to be restricted to a specific node, you must know the node name.
- Order of performing the action scripts for resources of this type in relation to resources of other types:
  - Resources are started in the increasing order of this value

– Resources are stopped in the decreasing order of this value

  Ensure that the number you choose for a new resource type permits the resource types on which it depends to be started before it is started, or stopped before it is stopped, as appropriate.

  Table 4-1 shows the conventions used for order ranges. The values available for customer use are 201-400 and 701-999.

Table 4-1 Order Ranges

Range	Reservation
1-100	SGI-provided basic system resource types, such as `MAC_address`
101-200	SGI-provided system plug-ins that can be started before `IP_address`
201-400	User-defined resource types that can be started before `IP_address`
401-500	SGI-provided basic system resource types, such as `IP_address`
501-700	SGI-provided system plug-ins that must be started after `IP_address`
701-999	User-defined resource types that must be started after `IP_address`

  Table 4-2 shows the order numbers of the resource types provided with the release.

Table 4-2 Resource Type Order Numbers

Order Number	Resource Type
10	`MAC_address`
20	`volume`
30	`filesystem`
201	`NFS`
401	`IP_address`
411	`statd`
501	`Netscape_web`

Order Number	Resource Type
511	`Oracle_DB`
521	`INFORMIX_DB`

- Restart mode, which can be one of the following values:
  - 0 = Do not restart on monitoring failures
  - 1 = Restart a fixed number of times
- Number of local restarts (when restart mode is 1).
- Location of the executable script. This is always `/var/cluster/ha/resource_types/`*resource_type_tname*.
- Monitoring interval, which is the time period (in milliseconds) between successive executions of the `monitor` action script; this is only valid for the `monitor` action script.
- Starting time for monitoring. When the resource group is made online in a cluster node, IRIS FailSafe will start monitoring the resources after the specified time period (in milliseconds).
- Action scripts to be defined for this resource type. You must specify scripts for `start`, `stop`, `exclusive`, and `monitor`, although the `monitor` script may contain only a return-success function if you wish. If you specify 1 for the restart mode, you must specify a `restart` script. The `probe` script is optional.
- Type-specific attributes to be defined for this resource type. The action scripts use this information to start, stop, and monitor a resource of this resource type. For example, NFS requires the following resource keys:
  - `export-point` which takes a value that defines the export disk name. This name is used as input to the `exportfs`(1M) command. For example:

    ```
    export-point = /this_disk
    ```

  - `export-info` which takes a value that defines the export options for the file system. These options are used in the `exportfs`(1M) command. For example:

    ```
    export-info = rw,wsync,anon=root
    ```

– `filesystem` which takes a value that defines the raw file system. This name is used as input to the `mount`(1M) command. For example:

```
filesystem = /dev/xlv/xlv_object
```

Using the GUI

You can use the FailSafe Manager graphical user interface (GUI) to define a new resource type and to define the dependencies for a given type. For details about the GUI, see the *IRIS FailSafe 2.0 Administrator's Guide*. For convenience, "Starting the FailSafe Manager", page 99, contains information about starting the GUI.

Define a New Resource Type

To define a new resource type using the GUI, select the following task:

```
Resources & Resource Types => Define a Resource Type
```

The GUI will prompt you for required and optional information. Online help is provided for each item.

The following figures show this process for a new resource type called `newresourcetype`.

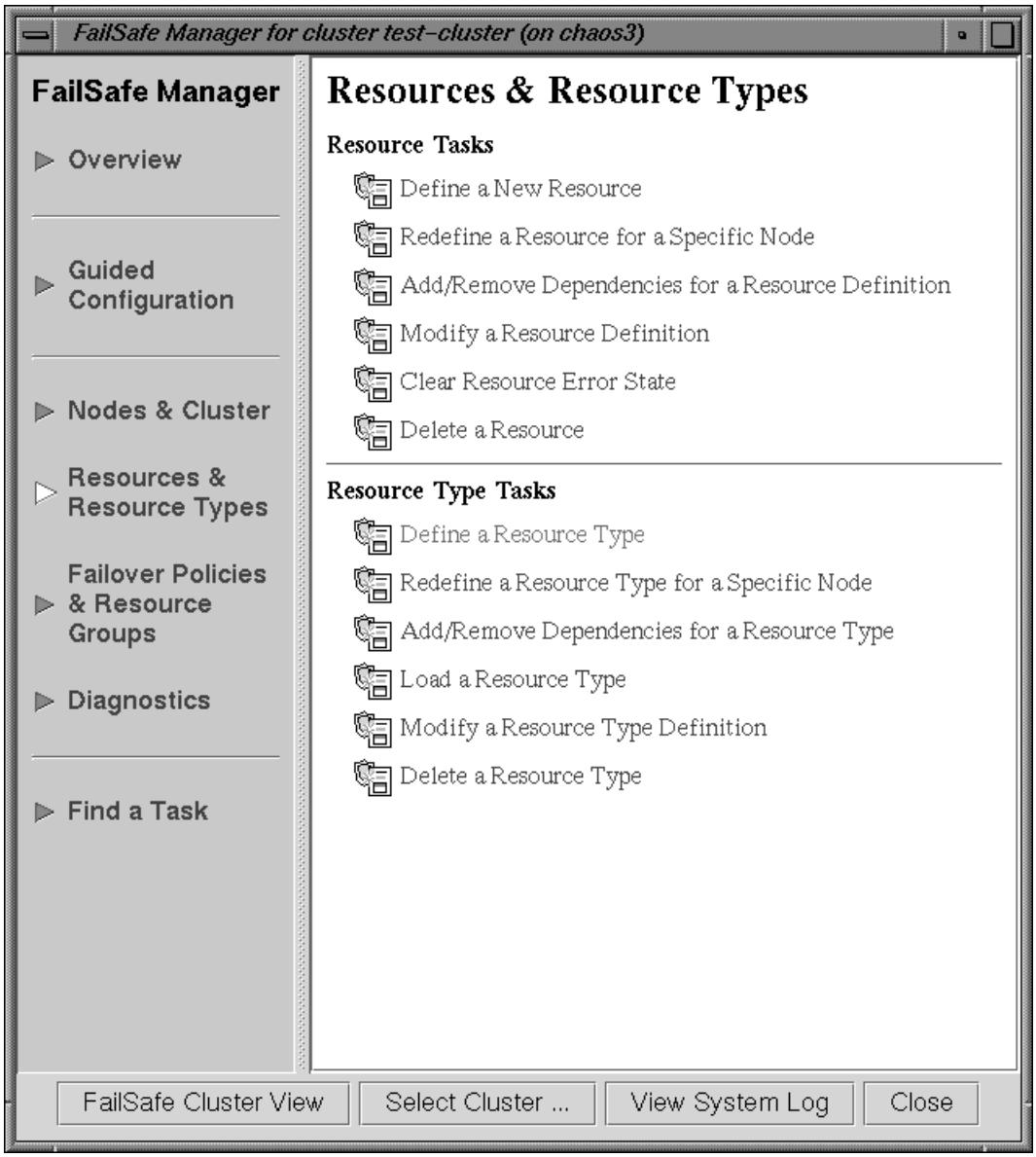

Figure 4-1 Select **Define a New Resource**

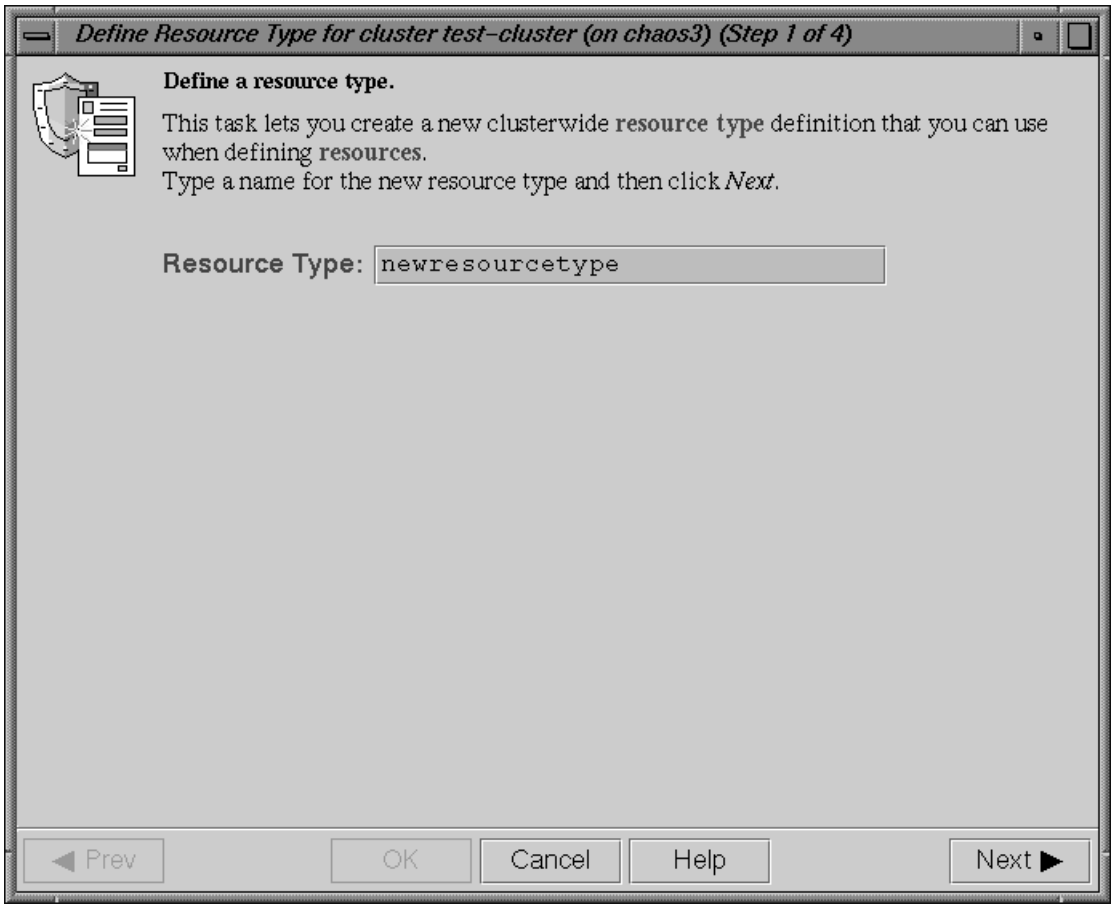

Figure 4-2 Specify the Name of the New Resource Type

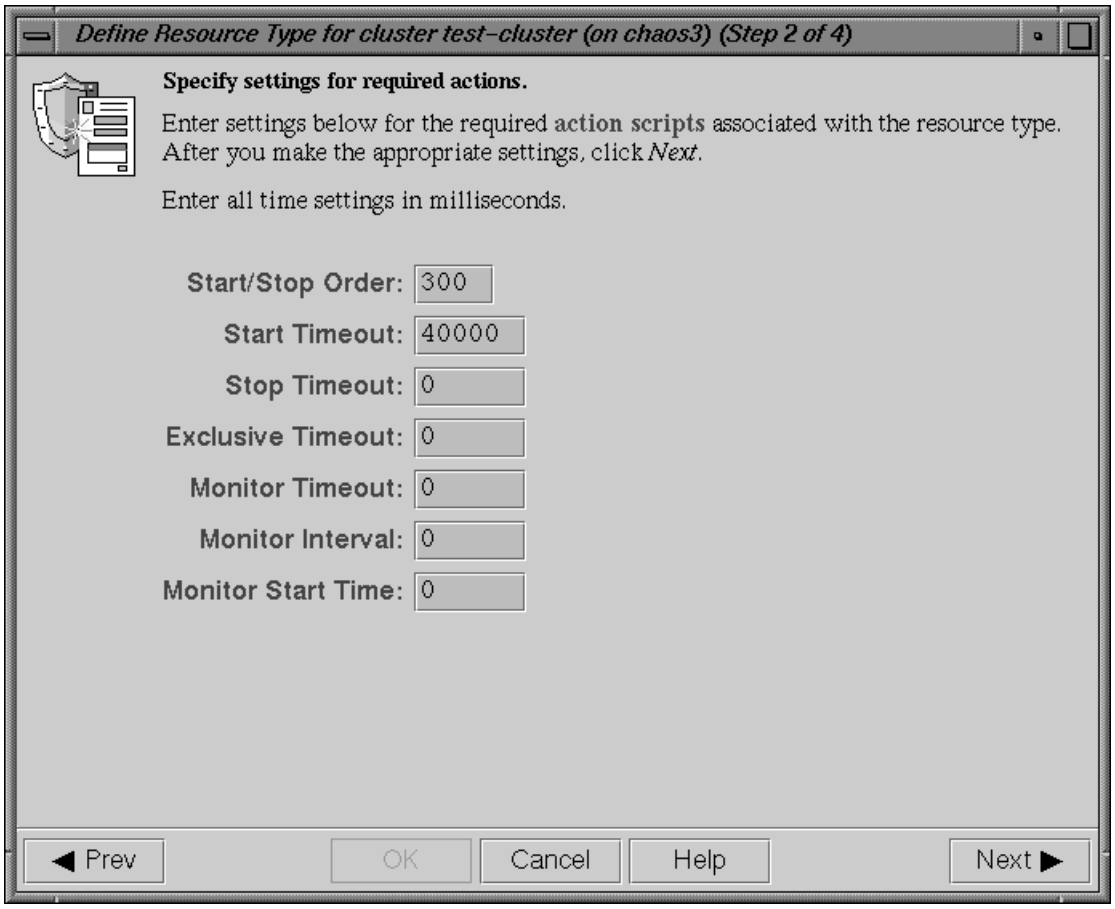

Figure 4-3 Specify Settings for Required Actions

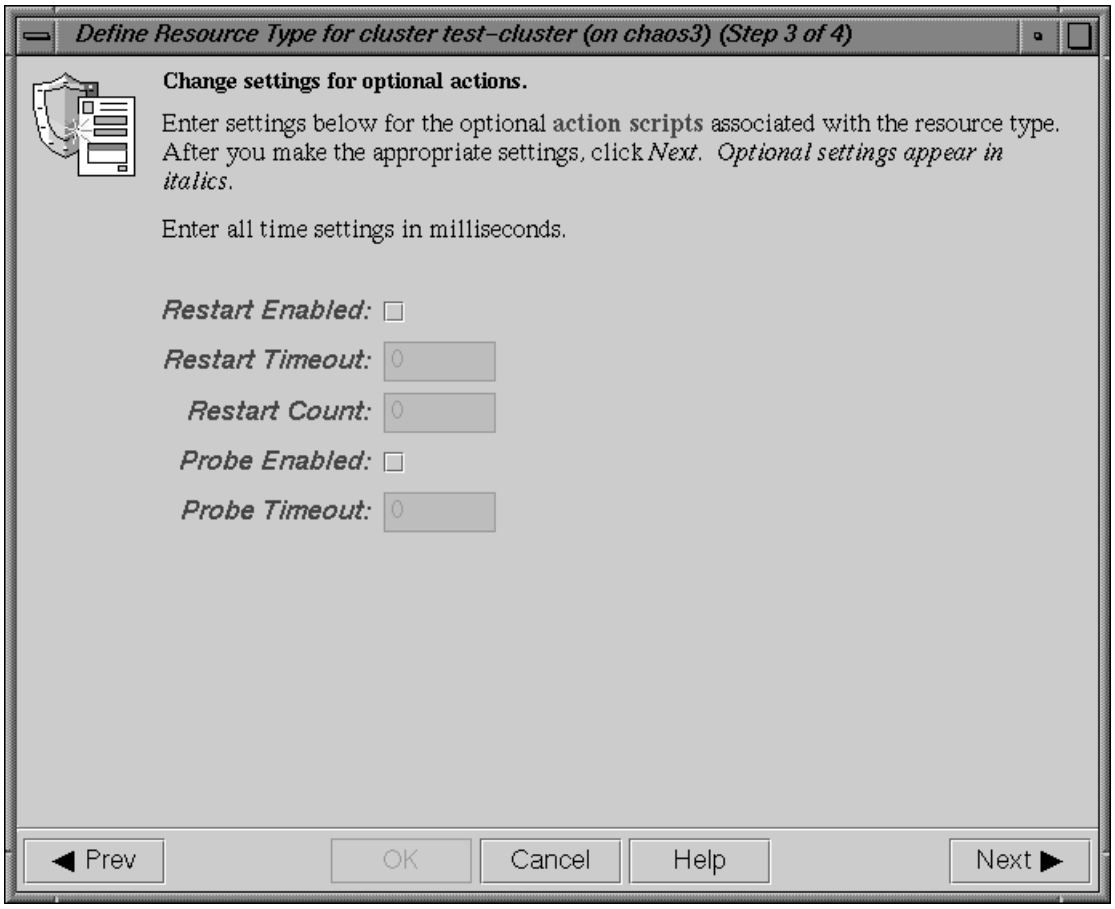

Figure 4-4 Change Settings for Optional Actions

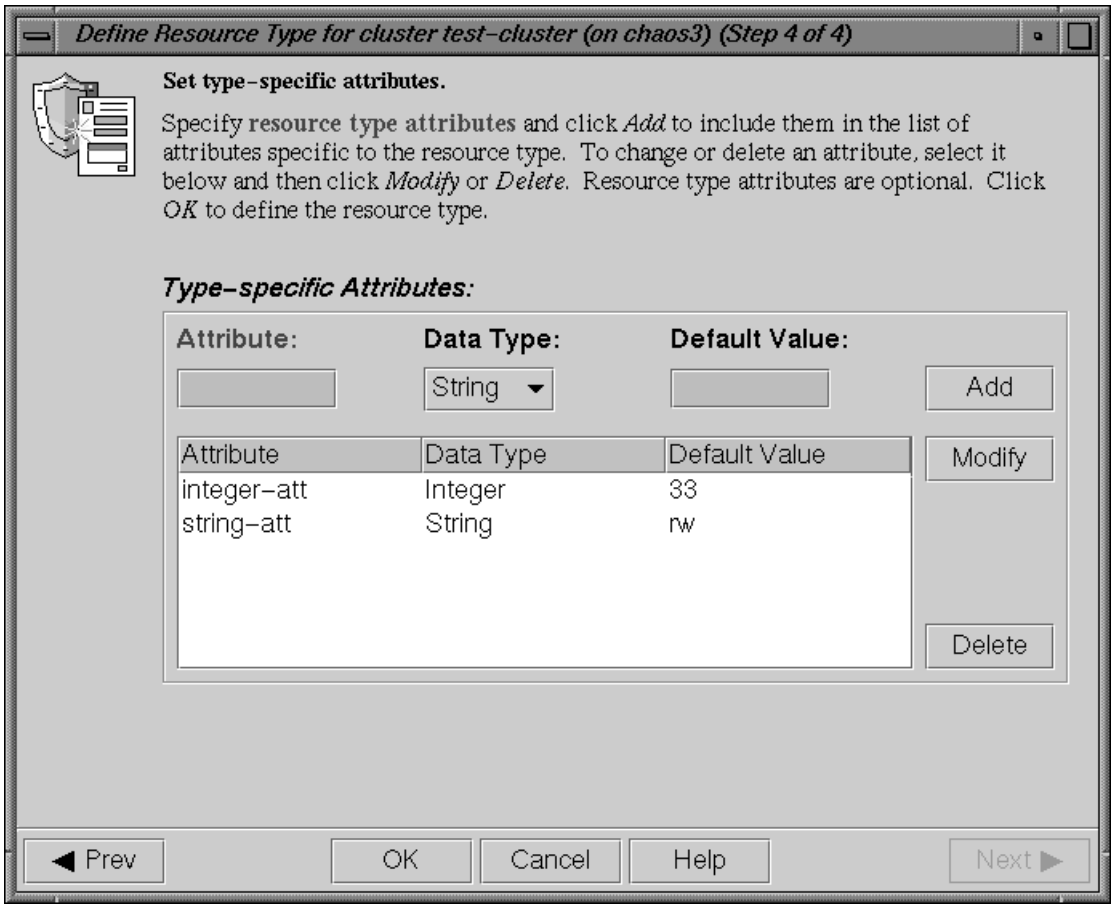

Figure 4-5 Set Type-specific Attributes

Define Dependencies

To define the dependencies for a given type use the following task:

```
Add/Remove Dependencies for a Resource Type
```

Figure 4-6 shows an example of adding two dependencies (`filesystem` and `NFS`) to the `newresourcetype` resource type.

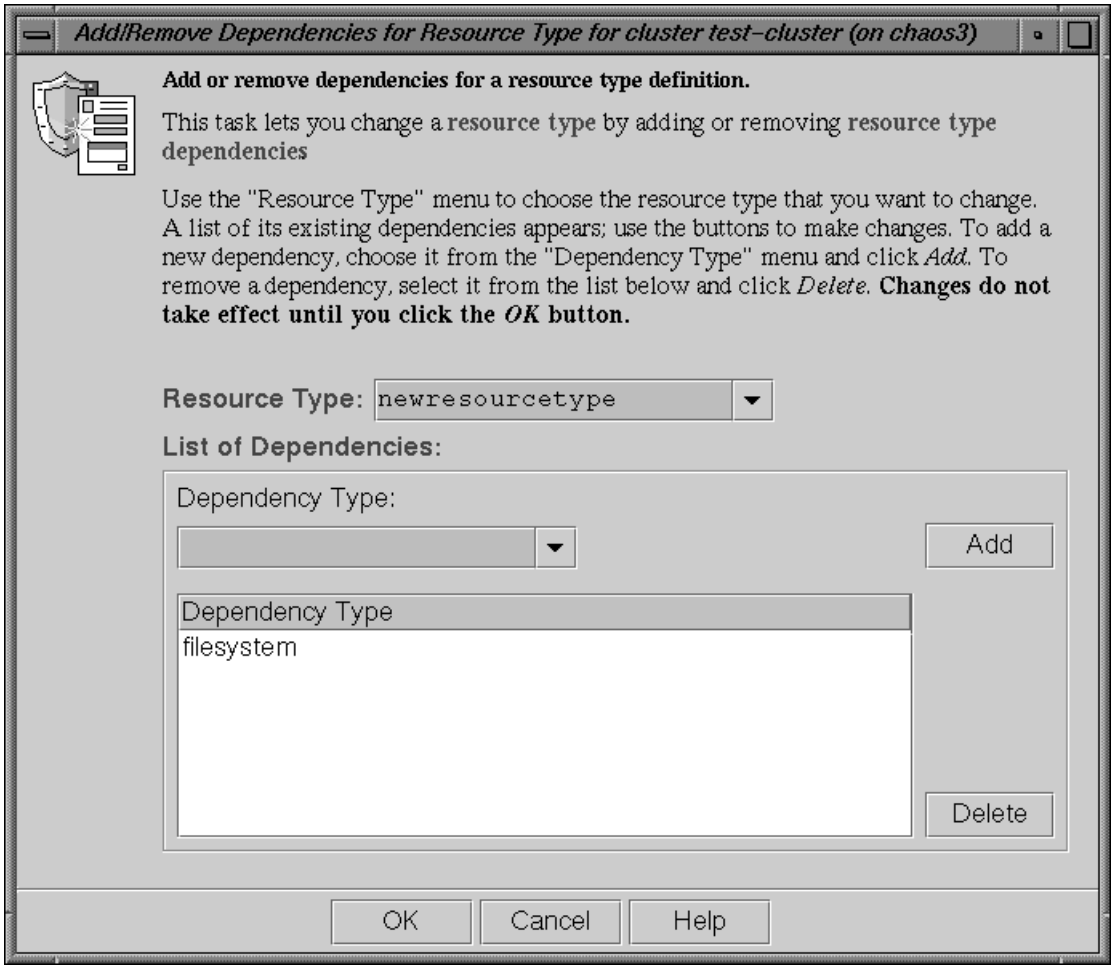

Figure 4-6 Add Dependencies

Using `cluster_mgr` Interactively

The following steps show the use of `cluster_mgr` interactively to define a resource type called `newresourcetype`.

Note: A resource type name cannot contain a space, an unprintable character, or any of the following characters:

```
*
?
\
#
```

---

1. Log in as `root`.
2. Execute the `cluster_mgr` command using the `-p` option to prompt you for information (the command name can be abbreviated to `cmgr`):

   ```
   # /usr/cluster/bin/cluster_mgr -p
   Welcome to IRIS FailSafe Cluster Manager Command-Line Interface

   cmgr>
   ```

3. Use the `set` subcommand to specify the default cluster used for `cluster_mgr` operations. In this example, we use a cluster named `test`:

   ```
   cmgr> set cluster test
   ```

   Note: If you prefer, you can specify the cluster name as needed with each subcommand.

   ---

4. Use the `define resource_type` subcommand. By default, the resource type will apply across the cluster; if you wish to limit the resource type to a specific node, enter the node name when prompted. If you wish to enable restart mode, enter 1 when prompted.

   Note: The following example only shows the prompts and answers for two action scripts (`start` and `stop`) for a new resource type named`newresourcetype`.

```
cmgr> define resource_type newresourcetype

(Enter "cancel" at any time to abort)

Node[optional]?
Order ? 300
Restart Mode ? (0)

DEFINE RESOURCE TYPE OPTIONS

        1) Add Action Script.
        2) Remove Action Script.
        3) Add Type Specific Attribute.
        4) Remove Type Specific Attribute.
        5) Add Dependency.
        6) Remove Dependency.
        7) Show Current Information.
        8) Cancel. (Aborts command)
        9) Done. (Exits and runs command)

Enter option:1

No current resource type actions

Action name ? start
Executable Time? 40000
Monitoring Interval? 0
Start Monitoring Time? 0

        1) Add Action Script.
        2) Remove Action Script.
        3) Add Type Specific Attribute.
        4) Remove Type Specific Attribute.
        5) Add Dependency.
        6) Remove Dependency.
        7) Show Current Information.
        8) Cancel. (Aborts command)
        9) Done. (Exits and runs command)

Enter option:1
```

```
Current resource type actions:
        Action - 1: start

Action name stop
Executable Time? 40000
Monitoring Interval? 0
Start Monitoring Time? 0

        1) Add Action Script.
        2) Remove Action Script.
        3) Add Type Specific Attribute.
        4) Remove Type Specific Attribute.
        5) Add Dependency.
        6) Remove Dependency.
        7) Show Current Information.
        8) Cancel. (Aborts command)
        9) Done. (Exits and runs command)

Enter option:3

No current type specific attributes

Type Specific Attribute ? integer-att
Datatype ? integer
Default value[optional] ? 33

        1) Add Action Script.
        2) Remove Action Script.
        3) Add Type Specific Attribute.
        4) Remove Type Specific Attribute.
        5) Add Dependency.
        6) Remove Dependency.
        7) Show Current Information.
        8) Cancel. (Aborts command)
        9) Done. (Exits and runs command)

Enter option:3

Current type specific attributes:
        Type Specific Attribute - 1: export-point
```

```
Type Specific Attribute ? string-att
Datatype ? string
Default value[optional] ? rw

        1) Add Action Script.
        2) Remove Action Script.
        3) Add Type Specific Attribute.
        4) Remove Type Specific Attribute.
        5) Add Dependency.
        6) Remove Dependency.
        7) Show Current Information.
        8) Cancel. (Aborts command)
        9) Done. (Exits and runs command)

Enter option:5

No current resource type dependencies

Dependency name ? filesystem

        1) Add Action Script.
        2) Remove Action Script.
        3) Add Type Specific Attribute.
        4) Remove Type Specific Attribute.
        5) Add Dependency.
        6) Remove Dependency.
        7) Show Current Information.
        8) Cancel. (Aborts command)
        9) Done. (Exits and runs command)

Enter option:7

Current resource type actions:
        Action - 1: start
        Action - 2: stop

Current type specific attributes:
        Type Specific Attribute - 1: integer-att
        Type Specific Attribute - 2: string-att

No current resource type dependencies
```

```
Resource dependencies to be added:
        Resource dependency - 1: filesystem

        1) Add Action Script.
        2) Remove Action Script.
        3) Add Type Specific Attribute.
        4) Remove Type Specific Attribute.
        5) Add Dependency.
        6) Remove Dependency.
        7) Show Current Information.
        8) Cancel. (Aborts command)
        9) Done. (Exits and runs command)

Enter option:9
Successfully defined resource_type newresourcetype

cmgr> show resource_types in cluster test

NFS
template
Netscape_web
newresourcetype
statd
Oracle_DB
MAC_address
IP_address
INFORMIX_DB
filesystem
volume

cmgr> exit
#
```

Using `cluster_mgr` With a Script

You can write a script that contains all of the information required to define a resource type and supply it to `cluster_mgr` by using the `-f` option:

`cluster_mgr -f` *scriptname*

Or, you could include the following as the first line of the script and then execute the script itself:

```
#!/usr/cluster/bin/cluster_mgr -f
```

If any line of the script fails, `cluster_mgr` will exit. You can choose to ignore the failure and continue the process by using the -i option, as follows:

```
#!/usr/cluster/bin/cluster_mgr -if
```

Note: If you include `-i` when using a cluster_mgr command line as the first line of the script, you must use this exact syntax (that is, `-if`).

A template script for creating a new resource type is located in `/var/cluster/cmgr-templates/cmgr-create-resource_type`. Each line of the script must be a valid `cluster_mgr` line, a comment line (starting with #), or a blank line.

Note: You must include a `done` command line to finish a multi-level command. If you concatenate information from multiple template scripts to prepare your cluster configuration, you must remove the `quit` at the end of each template script.

For example, you could use the following script to define the same `newresourcetype` resource type defined interactively in the previous section:

```
# newresourcetype.script: Script to define the "newresourcetype" resource type

set cluster test
define resource_type newresourcetype
set order to 300
set restart_mode to 0
add action start
set exec_time to 40000
set monitor_interval to 0
set monitor_time to 0
done
add action stop
set exec_time to 40000
set monitor_interval to 0
set monitor_time to 0
done
```

```
add type_attribute integer-att
set data_type to integer
set default_value to 33
done
add type_attribute string-att
set data_type to string
set default_value to rw
done
add dependency filesystem
done
quit
```

When you execute the `cluster_mgr -f` command line with this script, you will see the following output:

```
# /usr/cluster/bin/cluster_mgr -f newresourcetype.script
Successfully defined resource_type newresourcetype

#
```

To verify that the resource type was defined, enter the following:

```
# /usr/cluster/bin/cluster_mgr -c "show resource_types in cluster test"

NFS
template
Netscape_web
newresourcetype
statd
Oracle_DB
MAC_address
IP_address
INFORMIX_DB
filesystem
volume
```

Testing a New Resource Type

After adding a new resource type, you should test it as follows:

1. Define a resource group that contains resources of the new type. Ensure that the group contains all of the resources on which the new resource type depends.
2. Bring the resource group online in the cluster using `cluster_mgr` or the GUI.

   For example, using `cluster_mgr`:

   ```
   cmgr> admin online resource_group new_rg in cluster test_cluster
   ```

3. Check the status of the resource group using `cluster_mgr` or GUI after few minutes.

   For example:

   ```
   cmgr> show status of resource_group new_rg in cluster test_cluster
   ```

4. If the resource group has been made online successfully, you will see output similar to the following:

   ```
   State: Online
   Error: No error
   Owner: node1
   ```

5. If there are resource group errors, do the following:

   - Check the `srmd` logs (`/var/cluster/ha/log/srmd_`*nodename*) on the node on which the resource group is online
   - Search for the string `ERROR` in the log file. There should be an error message about a resource in the resource group. The message also provides information about the action script that failed. For example:

```
Wed Nov 3 04:20:10.135 <E ha_srmd srm 12127:1 sa_process_tasks.c:627>
CI_FAILURE, ERROR: Action (exclusive) for resource (10.0.2.45) of type
(IP_address) failed with status (failed)
exclusive script failed for the resource 10.0.2.45 of resource type
IP_address. The status "failed"
indicates that the script returned an error.
```

   - Check the script logs (`/var/cluster/ha/log/script_`*nodename* on the same node) for `IP_address exclusive` script errors.

- After the fixing the problems in the action script, perform an `offline_force` operation to clear the error. For example:

```
cmgr> admin offline_force resource_group new_rg in cluster test_cluster
```

Testing Scripts

This chapter describes how to test action scripts without running IRIS FailSafe. It also provides tips on how to debug problems that you may encounter.

Note: Parameters are passed to the action scripts as both input files and output files. Each line of the input file contains the resource name; the output file contains the resource name and the script exit status.

General Testing and Debugging Techniques

Some general testing and debugging techniques you can use during testing are as follows:

- To get debugging information, adding the following line to each of your scripts in the main function of the script:

  ```
  set -x
  ```

- To check that an application is running on a node, you may be able to use a command provided by the application. For example, the IRIS FailSafe INFORMIX option uses the INFORMIX command `onstat`.
- Another way to check that an application is running on a node, is to enter this command on that node:

  # **ps -ef** | **grep** *application*

  application is the name (or a portion of the name) of the executable for the application.

- To show the status of a resource, use the following `cluster_mgr` command:

  cmgr> **set cluster** *clustername*
  cmgr> **show status of resource** *resourcename* **of resource_type** *typename*

- To show the status of a node, use the following `cluster_mgr` command:

  cmgr> **show status of node** *nodename*

- To show the status of a resource group, use the following `cluster_mgr` command:

  ```
  cmgr> show status of resource_group rgname in cluster cname
  ```

Debugging Notes

- The `exclusive` script returns an error when the resource is running in the local node. If the resource is actually running in the node, there is no `exclusive` action script bug.
- If the resource group does not become online on the primary node, it can be because of a `start` script error on the primary node or a `monitor` script on the primary node. The nature of failure can be seen in the `srmd` logs of the primary node.
- If the action script failure status is `timeout`, resource type timeouts for the action should be increased. In the case of the `monitor` script, the check can be made more lightweight.
- The resource type action script timeouts are for a resource. So, if an action is performed on two resources, the script timeout is twice the configured resource type action timeout.
- If the resource group has a configuration error, check the `srmd` logs on the primary node for errors.
- The action scripts that use `${HA_LOG}` and `${HA_DBGLOG}` macros to log messages can find the messages in `/var/cluster/ha/log/script_`*nodename* file in each node in the cluster.

Testing an Action Script

To test an action script, do the following:

1. Create an input file, such as `/tmp/input`, that contains expected resource names. For example, to create a file that contains the resource named `disk1` do the following:

   ```
   # echo "/disk1" > /tmp/input
   ```

2. Create an input parameter file, such as `/tmp/ipparamfile`, as follows:

   ```
   # echo "ClusterName web-cluster" > /tmp/ipparamfile
   ```

3. Execute the action script as follows:

   ```
   # start /tmp/input /tmp/output /tmp/ipparamfile
   ```

   Note: The use of the input parameter file is optional.

The output file will contain one of the following return values for the `start`, `stop`, `probe`, `monitor`, and `restart` scripts:

```
HA_SUCCESS=0
HA_INVAL_ARGS=1
HA_CMD_FAILED=2
HA_NOTSUPPORTED=3
HA_NOCFGINFO=4
```

The output file will contain one of the following return values for the `exclusive` script:

```
HA_NOT_RUNNING=0
HA_RUNNING=2
```

Note: If you call the `exit_script` function prior to normal termination, it should be preceded by the `ha_write_status_for_resource` function and you should use the same return code that is logged to the output file.

Suppose you have a resource named `/disk1` and the following files:

- The syntax for the input file is: <*resourcename*>
- The syntax for the output file is: <*resourcename*> <*status*>

The following example shows:

- The exit status of the action script is 2
- The exit status of the resource is 2

Note: The use of anonymous indicates that the script was run manually. When the script is run by IRIS FailSafe, the full path to the script name is displayed.

```
# echo "/disk1" > /tmp/ipfile
# ./monitor  /tmp/ipfile /tmp/opfile /tmp/ipparamfile
# echo $?
2
# cat /tmp/opfile
/disk1 2
# tail /var/cluster/ha/log/script_heb1
Tue Aug 25 11:32:57.437 <anonymous script 23787:0 Unknown:0> ./monitor:
./monitor called with /tmp/ipfile and /tmp/opfile
Tue Aug 25 11:32:58.118 <anonymous script 24556:0 Unknown:0> ./monitor:
check to see if /disk1 is mounted on /disk1
Tue Aug 25 11:32:58.433 <anonymous script 23811:0 Unknown:0> ./monitor:
/sbin/mount | grep /disk1 | grep /disk1 >> /dev/null 2>&1 exited with
status 0
Tue Aug 25 11:32:58.665 <anonymous script 24124:0 Unknown:0> ./monitor:
stat mount point /disk1
Tue Aug 25 11:32:58.969 <anonymous script 23525:0 Unknown:0> ./monitor:
/sbin/stat /disk1 exited with status 0
Tue Aug 25 11:32:59.258 <anonymous script 24431:0 Unknown:0> ./monitor:
check the filesystem /disk1 is exported
Tue Aug 25 11:32:59.610 <anonymous script 6982:0 Unknown:0> ./monitor:
Tue Aug 25 11:32:59.917 <anonymous script 24040:0 Unknown:0> ./monitor:
awk '{print \$1}' /var/cluster/ha/tmp/exportfs.23762 | grep /disk1 exited
with status 1
Tue Aug 25 11:33:00.131 <anonymous script 24418:0 Unknown:0> ./monitor:
echo failed to find /disk1 in exported filesystem list:-
Tue Aug 25 11:33:00.340 <anonymous script 24236:0 Unknown:0> ./monitor:
echo /disk2
```

For additional information about a script's processing, see the /var/cluster/ha/log/script_*nodename*.

Special Testing Considerations for the `monitor` Script

The `monitor` script tests the liveliness of applications and resources. The best way to test it is to induce a failure, run the script, and check if this failure is detected by the script; then repeat the process for another failure.

Use this checklist for testing a `monitor` script:

- Verify that the script detects failure of the application successfully.
- Verify that the script always exits with a return value.
- Verify that the script does not contain commands that can hang (such as using DNS for name resolution) or those that continue forever, such as `ping`.
- Verify that the script completes before the time-out value specified in the configuration file.
- Verify that the script's return codes are correct.

During testing, measure the time it takes for a script to complete and adjust the monitoring times in your script accordingly. To get a good estimate of the time required for the script to execute, run it under different system load conditions.

Migrating to IRIS FailSafe 2.0

This chapter provides guidelines for migrating your IRIS FailSafe 1.2 resources and monitor script information to IRIS FailSafe 2.0 action scripts. It assumes you are already familiar with the migration information provided in the *IRIS FailSafe 2.0 Administrator's Guide*.

Cautions

Multiple instances of action scripts may be executed at the same time. To avoid this, you can use the `ha_execute_lock` command. For more information, see "Multiple Instances of Script Executed at the Same Time", page 22.

The software for IRIS FailSafe 2.0 and IRIS FailSafe 1.2 can coexist in the same node. However, IRIS FailSafe 2.0 and IRIS FailSafe 1.2 cannot run at the same time.

There are no configuration checksum verification in scripts.

Resource Types

In 2.0, the `ha.conf` configuration file has been replaced by the configuration database. The configuration database is automatically copied to all nodes in the pool. See the *IRIS FailSafe 2.0 Administrator's Guide* for information about configuring a 2.0 system.

If you require new resource types, you will create them using either the IRIS FailSafe Cluster Manager GUI (graphical user interface) or the `cluster_mgr` command. See Chapter 4, "Defining a New Resource Type".

You may be able to reuse the following monitoring information from the 1.2 `ha.conf` file with regard to 2.0 resource types:

- `start-monitor-time`
- `lmon-probe-time`
- `lmon-timeout`

Note: All IRIS FailSafe 2.0 time-outs are in milliseconds.

The following examples show information (in bold) that is used in the 1.2 `ha.conf` file and reused when creating a new resource type in 2.0.

Suppose a portion of the 1.2 `ha.conf` file had this:

```
action apache
{
        local-monitor = /var/ha/actions/ha_apache_lmon
}

action-timer apache
{
        start-monitor-time = 120
        lmon-probe-time = 120
        lmon-timeout = 60
}
```

You would reuse the information when creating a resource type in 2.0, as follows:

```
cmgr> create resource_type apache in cluster apache-cluster
Enter commands, when finished enter either "done", "cancel", "check"
Resource Type Name [apache]? apache
Cluster? apache-cluster
Node? node1
Order [0]? 500
Restart Mode [0]?0
Restart Count [0]?0
Number of Actions [0]? 4
Action? start
Executable? /var/cluster/ha/resource_types/apache/start
Executable Time? 20000
Monitoring Interval? 0
Start Monitoring Time? 0
Action? stop
Executable? /var/cluster/ha/resource_types/apache/stop
Executable Time? 20000
Monitoring Interval? 0
Start Monitoring Time? 0
Action? monitor
Executable?  /var/cluster/ha/resource_types/apache/monitor
Executable Time? 60000
Monitoring Interval? 120000
```

```
Start Monitoring Time? 120000
Action? exclusive
Executable? /var/cluster/ha/resource_types/apache/exclusive
Executable Time? 60000
Monitoring Interval? 0
Start Monitoring Time? 0?0
Number of Resource Keys [0]? 1
Name of resource key? search-string
Datatype? string
Default Key? httpd
Enter dependency commands,when finished enter either "done" or "cancel"

resource_type apache? add type IP_address
resource_type apache? done
```

Reading Information

In 2.0, configuration information is read using the `ha_get_info()` and `ha_get_field()` shell functions. These functions are equivalent to the 1.2 `ha_cfginfo` command.

In 2.0, all common functions and variables are kept in `/var/cluster/ha/common_scripts/scriptlib` file. This file is equivalent to the 1.2 `/var/ha/actions/common.vars` file.

For more information, see "Using the Script Library", page 101.

Parameter Parsing

In 2.0, action script parameters are passed in a file and information is also returned in a file. The script takes a list of resource names as parameters.

Action Scripts

Table A-1, page 94, summarizes the differences in scripts between the releases.

Table A-1 Differences between IRIS FailSafe 1.2 and 2.0 Scripts

IRIS FailSafe 1.2	IRIS FailSafe 2.0
`giveaway, giveback`	`stop`
`takeover, takeback`	`start`
`check`	`monitor`
(no equivalent)	`exclusive`, `probe`, `restart`

In 2.0, the action scripts are installed as `/var/cluster/ha/Resource_Type_Name/Action_Name` directory, where *Resource_Type_Name* is the name of the resource type (such as `NFS` and *Action_Name* is the name of the action script (such as `start`).

Templates of the action scripts (`start`, `stop`, `probe`, `monitor`, `exclusive`, `restart`) are provided in the `/var/cluster/ha/resource_types/template` directory. For more information about action scripts, see Chapter 2, "Writing the Action Scripts and Adding Monitoring Agents".

The following sections provide example portions of 1.2 scripts and their 2.0 equivalents:

- `giveback` and `stop`
- `takeover` and `start`
- `monitor` and `monitor`

Note: There are no 1.2 equivalents for the 2.0 `probe`, `exclusive`, and `restart` scripts.

In the following examples, only the relevant portions of the scripts are shown. Areas in common between 1.2 and 2.0 are in bold.

1.2 giveback / 2.0 stop

For example, suppose you had the following in the `giveback` script in 1.2:

```
giveback()
{
```

```
    for i in`$CFG_INFO ${T_APACHE}`
    do
        SEARCH="$CFG_INFO ${T_APACHE}${CFG_SEP}${i}${CFG_SEP}${T_BACKUP}"
        BACKUP=`$SEARCH`
        if [ $? -eq 1 ]; then
            ${LOGGER} "$0: Trouble finding backup-node for apache ($SEARCH)"
            exit $INCORRECT_CONF_FILE;
        fi
        # If I am the backup
        if [ ${BACKUP} = ${HOST} ]; then
            ${LOGGER} "$0: Stopping apache for backup server."
            killall -9 /apache-fs/usr/local/apache_1.2.0/src/httpd
            if [ $? -ne "0" ]; then
                ${LOGGER} "$0: halt of apache on backup server failed."
            fi
        fi

        exit $SUCCESS
    done
}
```

In 2.0, you would have the following in the `stop` script:

```
stop_apache()
{
    for server in $HA_RES_NAMES
    do
        ${HA_DBGLOG} "Stopping apache server $server"
        killall -9 /apache-fs/usr/local/apache_1.2.0/src/httpd
        if [ $? -ne "0" ]; then
            ${HA_LOG} "halt of apache server $server failed."
            ha_write_status_for_resource $server $HA_CMD_FAILED;
        else
            ${HA_DBGLOG} "halt of apache server $server successful"
            ha_write_status_for_resource $server $HA_SUCCESS;
        fi
    done
}
```

1.2 takeover / 2.0 start

For example, suppose you had the following in the `takeover` script in 1.2:

```
takeover()
{
    for i in`$CFG_INFO ${T_APACHE}`
    do
        SEARCH="$CFG_INFO ${T_APACHE}${CFG_SEP}${i}${CFG_SEP}${T_BACKUP}"
        BACKUP=`$SEARCH`
        if [ $? -eq 1 ]; then
            ${LOGGER} "$0: Trouble finding backup-node for apache ($SEARCH)"
            exit $INCORRECT_CONF_FILE;
        fi
        # If I am the backup
        if [ ${BACKUP} = ${HOST} ]; then
            ${LOGGER} "$0: Starting apache for backup server."
            /apache-fs/usr/local/apache_1.2.0/src/httpd -d \
/apache-fs/usr/local/apache_1.2.0
           if [ $? -ne "0" ]; then
                ${LOGGER} "$0: start of apache on backup server failed."
                exit $FAILED
            fi
        fi
        exit $SUCCESS
    done
}
```

In 2.0, you would have the following in the `start` script:

```
start_apache()
{
    for server in $HA_RES_NAMES
    do
        ${HA_DBGLOG} "Starting apache server $server"
        /apache-fs/usr/local/apache_1.2.0/src/httpd -d \
/apache-fs/usr/local/apache_1.2.0
        if [ $? -ne "0" ]; then
            ${HA_LOG} "start of apache server $server failed."
            ha_write_status_for_resource $server $HA_CMD_FAILED;
        else
            ${HA_DBGLOG} "start of apache server $server successful"
```

```
            ha_write_status_for_resource $server $HA_SUCCESS;
        fi
    done
}
```

1.2 monitor/ 2.0 monitor

For example, suppose you had the following in the `monitor` script in 1.2:

```
monitor()
{

    # Read the search string entry
    for i in`$CFG_INFO ${T_APACHE}
    do
        SEARCH="$CFG_INFO ${T_APACHE}${CFG_SEP}${i}${CFG_SEP}${T_SEARCH_STR}"
        SEARCH_STR=$SEARCH
        ${SEARCH_STR:=httpd};
    done

    EXEC="${KILLALL} -0 ${SEARCH_STR}";
    execute_cmd "check if apache server processes are running"

}
```

In 2.0, you would have the following in the `monitor` script:

```
monitor_apache()
{
    for server in $HA_RES_NAMES
    do
        get_apache_info $server
        if [ $? -eq 0 ]; then
            APACHE_FIELDS=${HA_STRING
            ha_get_field "${APACHE_FIELDS}" search-string;
            if [ $? -eq 0 ]; then
                SEARCH_STR=${HA_FIELD_VALUE};
            fi
        fi
        ${SEARCH_STR:=httpd};
        HA_CMD=${KILLALL} -0 ${SEARCH_STR}";
```

```
        ha_execute_cmd "check if server $server processes are running"
        if [ $? -ne 0 ]; then
            ${HA_LOG} "monitor of apache server $server failed."
            ha_write_status_for_resource $server $HA_CMD_FAILED;
        else
            ${HA_DBGLOG} "monitor of apache server $server successful"
            ha_write_status_for_resource $server $HA_SUCCESS;
        fi
    done
}
```

Ordering Script Actions

In 2.0, each resource type has a start/stop order, which is a nonnegative integer. In a resource group, the start/stop orders of the component resource types determine the order in which the resources will be started when IRIS FailSafe brings the group online and will be stopped when IRIS FailSafe takes the group offline. The group's resources are started in increasing order, and stopped in decreasing order.

Note: Resources of the same type are started and stopped in indeterminate order.

For example, if resource type `volume` has order 10 and resource type `filesystem` has order 20, then when IRIS FailSafe brings a resource group online, all volume resources in the group will be started before all file system resources in the group.

There is no need to create software links similar to those used in 1.2.

Starting the FailSafe Manager

To start the FailSafe Manager, use one of these methods:

- Choose `FailSafe Manager` from the FailSafe toolchest.

  You will need to restart the toolchest after installing FailSafe to see the FailSafe entry on the toolchest display. Enter the following commands to restart the toolchest:

  ```
  % killall toolchest
  % /usr/bin/X11/toolchest &
  ```

  In order for this to take effect, `sysadm_failsafe2.sw.desktop` must be installed on the client system, as described in the *IRIS FailSafe 2.0 Installation and Maintenance Instructions*.

- Enter the following command line:

  ```
  % /usr/sbin/fstask
  ```

- In your Web browser, enter the following, where *server* is the name of the node in the pool or cluster that you want to administer:

  ```
  http://server/FailSafe Manager/
  ```

  At the resulting Web page, click on the icon.

Figure B-1, page 100, shows the FailSafe Manager.

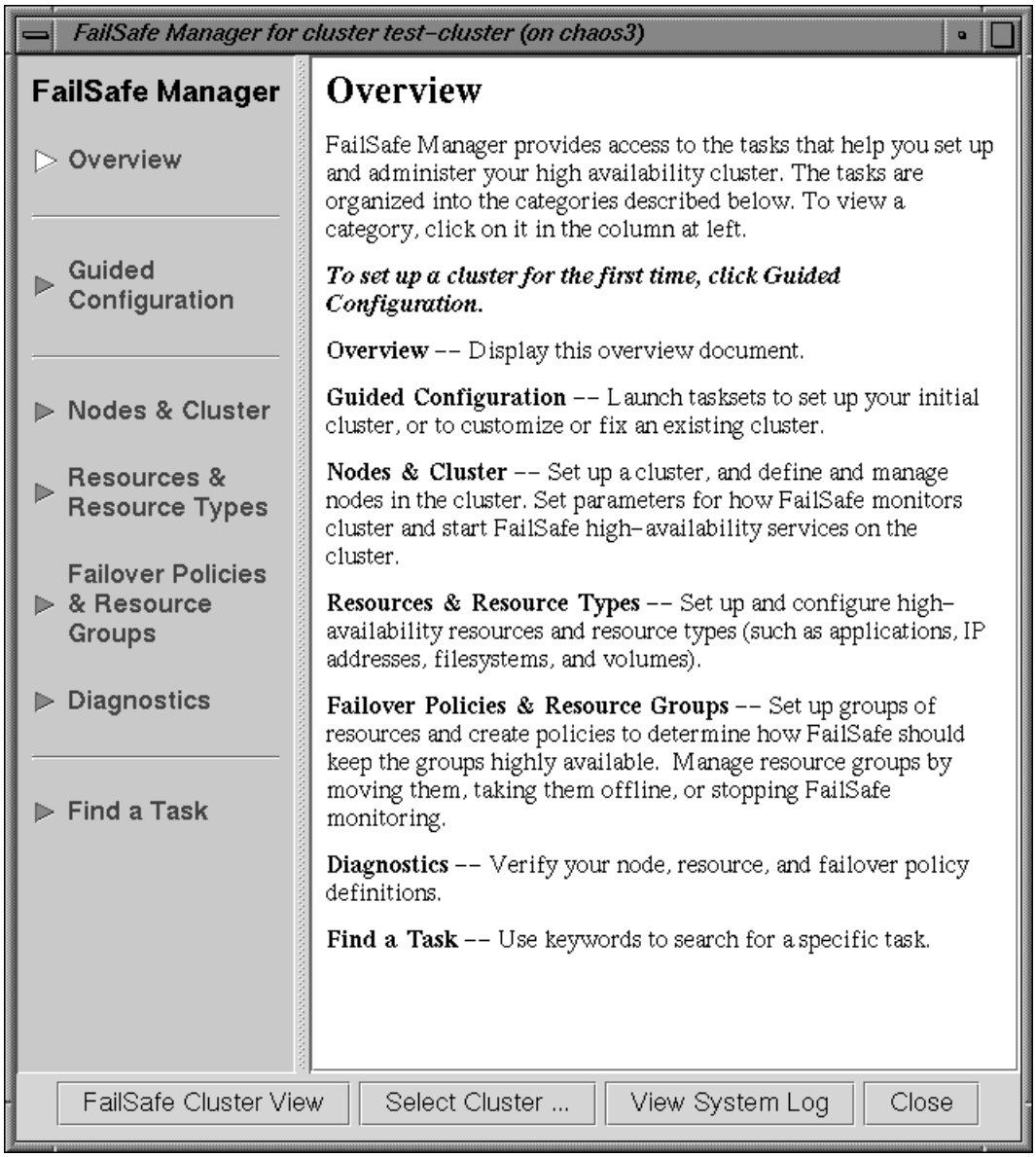

Figure B-1 FailSafe Manager

Using the Script Library

The `/var/cluster/ha/common_scripts/scriptlib` file contains the library of environment variables (beginning with uppercase `HA_`) and functions (beginning with lowercase `ha_`) available for use in your action scripts.

Note: Do not change the contents of the `scriptlib` file.

This chapter describes functions that perform the following tasks, using samples from the `scriptlib` file:

- Set global definitions
- Check arguments
- Read an input file
- Execute a command
- Write status for a resource
- Get the value for a field
- Get resource information
- Print exclusivity check messages

Set Global Definitions

The `ha_set_global_defs()` function sets the global definitions for the environment variables shown in Table C-1, page 102.

Table C-1 Global Environment Variables

Variable Type	Variable Name	Default	Description
Global variable	HA_HOSTNAME	`uname -n`	The output of the uname command with the -n option, which is the host name or nodename. The nodename is the name by which the system is known to communications networks.
Command location	HA_CMDSPATH	/usr/cluster/bin	Path to user commands.
	HA_PRIVCMDSPATH	/usr/sysadm/privbin	Path to privileged commands (those that can only be run by root).
	HA_LOGCMD	ha_cilog	Command used to log into the IRIS FailSafe logs.
	HA_RESOURCEQUERYCMD	resourceQuery	Resource query command. This is an internal command that is not meant for direct use in scripts; use the ha_get_info() function of scriptlib instead.
	HA_SCRIPTTMPDIR	/tmp	Location of the script temporary directory.
Database location	HA_CDB	/var/cluster/cdb/cdb.db	Location of the IRIS FailSafe database.
Script log variables	HA_SCRIPTGROUP	script	Log for the script group.
	HA_SCRIPTSUBSYS	script	Log for the script subsystem.
Script log levels	HA_NORMLVL	0	Normal level of script logs.
	HA_DBGLVL	10	Debug level of script logs.

Variable Type	Variable Name	Default	Description
Script logging commands	HA_LOGQUERY_OUTPUT	`'${HA_PRIVCMDSPATH}/loggroupQuery _NUM_LOG_GROUPS=1 \ _LOG_GROUP_0=ha_script'`	Determine the current logging level for scripts.
	HA_DBGLOG	ha_dbglog	Command used to log debug messages from the scripts.
	HA_CURRENT_LOGLEVEL	`'echo ${HA_LOGQUERY_OUTPUT} | /usr/bin/awk '{print $2}''`	Display the current log level. The default will be 0 (no script logging) if the loggroupQuery command fails or does not find configuration information.
	HA_LOG	ha_log	Command used to log the scripts.
Script error values	HA_SUCCESS	0	Successful execution of the script. This variable is used by the start, stop, restart, monitor, and probe scripts.
	HA_NOT_RUNNING	0	The script is not running. This variable is used by exclusive scripts.
	HA_INVAL_ARGS	1	An invalid argument was entered. This is used by all scripts.
	HA_CMD_FAILED	2	A command called by the script has failed. his variable is used by the start, stop, restart, monitor, and probe scripts.

Variable Type	Variable Name	Default	Description
	HA_RUNNING	2	The script is running. This variable is used by exclusive scripts.
	HA_NOTSUPPORTED	3	The specific action is not supported for this resource type. This is used by all scripts.
	HA_NOCFGINFO	4	No configuration information was found. This is used by all scripts.

Check Arguments

The ha_check_args() function checks the arguments specified for the script and sets the $HA_INFILE and $HA_OUTFILE variables, which specify the input and output files, respectively. It also reads the list of parameters from the input parameters file (if it exists) and sets the $HA_CLUSTERNAME variable. The input parameters file contains the list of parameters that can be used by the executable. The format of input parameters file is as follows, where *key* is ClusterName:

key value

Note: The input parameter file is optional.

In the following, long lines use the continuation character (\) for readability.

```
ha_check_args()
{
    ${HA_DBGLOG} "$HA_SCRIPTNAME called with $1, $2 and $3"

    if  ! [ $# -eq 2 -o $# -eq 3 ]; then
        ${HA_LOG} "Incorrect number of arguments"
        return 1;
    fi

    if [ ! -r $1 ]; then
        ${HA_LOG} "file $1 is not readable or does not exist"
        return 1;
    fi

    if [ ! -s $1 ]; then
        ${HA_LOG} "file $1 is empty"
        return 1;
    fi

    if [ $# -eq 3 ]; then
        HA_PARAMFILE=$3

        if [ ! -r $3 ]; then
            ${HA_LOG} "file $3 is not readable or does not exist"
            return 1;
        fi

        HA_CLUSTERNAME=`/usr/bin/awk '{ if ( $1 == "ClusterName" ) \
        print $2 }' ${HA_PARAMFILE}`
    fi

    HA_INFILE=$1
    HA_OUTFILE=$2

    return 0;
}
```

Read an Input File

The ha_read_infile() function reads the $HA_INFILE input file into the $HA_RES_NAMES variable, which specifies the list of resource names.

```
ha_read_infile()
{
    HA_RES_NAMES="";

    for HA_RESOURCE in`cat ${HA_INFILE}
    do
        HA_TMP="${HA_RES_NAMES} ${HA_RESOURCE}";
        HA_RES_NAMES=${HA_TMP};
    done
}
```

Execute a Command

The ha_execute_cmd() function executes the command specified by $HA_CMD. $1 is the string to be logged. The function returns 1 on error and 0 on success. On errors, the standard output and standard error of the command is redirected to the log file.

```
ha_execute_cmd()
{
    OUTFILE=${HA_SCRIPTTMPDIR}/script.$$

    ${HA_DBGLOG} $1

    eval ${HA_CMD} > ${OUTFILE} 2>&1;

    ha_exit_code=$?;

    if [ $ha_exit_code -ne 0 ]; then
        ${HA_DBGLOG}`cat ${HA_SCRIPTTMPDIR}/script.$$
    fi

    ${HA_DBGLOG} "${HA_CMD} exited with status $ha_exit_code";

    /sbin/rm ${OUTFILE}

    return $ha_exit_code;
```

```
}
```

The ha_execute_cmd_ret() function is similar to ha_execute_cmd, except that it places the command output in the location specified by $HA_CMD_OUTPUT.

```
ha_execute_cmd_ret()
{
    ${HA_DBGLOG} $1

    # REVISIT: Is it possible to redirect the output to a log
    HA_CMD_OUTPUT=`${HA_CMD}`;

    ha_exit_code=$?;

${HA_DBGLOG} "${HA_CMD} exited with status $ha_exit_code";

    return $ha_exit_code;
}
```

Write Status for a Resource

The ha_write_status_for_resource() function writes the status for a resource to the $HA_OUTFILE output file. $1 is the resource name, and $2 is the resource status.

```
ha_write_status_for_resource()
{
    echo $1 $2 >> $HA_OUTFILE;
}
```

Similarly, the ha_write_status_for_all_resources() function writes the status for all resources. $HA_RES_NAMES is the list of resource names.

```
ha_write_status_for_all_resources()
{
    for HA_RES in $HA_RES_NAMES
    do
        echo $HA_RES $1 >> $HA_OUTFILE;
    done
}
```

Get the Value for a Field

The ha_get_field() function obtains the field value from a string, where $1 is the string and $2 is the field name. The string format is as follows:

name value name value . . .

```
ha_get_field()
{
    HA_STR=$1
    HA_FIELD_NAME=$2
    ha_found=0;

    for ha_i in $HA_STR
    do
        if [ $ha_i = $HA_FIELD_NAME ]; then
            ha_found=1;
            continue;
        fi
        if [ $ha_found -eq 1 ]; then
            HA_FIELD_VALUE=$ha_i
            return 0;
        fi
    done

    return 1;
}
```

Get Resource Information

The ha_get_info() and ha_get_info_debug()functions read resource information. $1 is the resource type and $2 is the resource name. Resource information is stored in the HA_STRING variable. All query errors are ignored; the return value is always 0. If the resourceQuery command fails, the HA_STRING is set to an invalid string, and future calls to ha_get_info() or ha_get_info_debug() return errors.

You can use ha_get_info_debug() while developing scripts.

```
ha_get_info()
{
    if [ -f /var/cluster/ha/resourceQuery.debug ]; then
```

```
        ha_get_info_debug $1 $2
        return;
    fi

    # Retry resourceQuery command $HA_RETRY_CMD_MAX times if $HA_RETRY_CMD_ERR
    # is returned.
    ha_retry_count=1

    while [ $ha_retry_count -le $HA_RETRY_CMD_MAX ];
    do
        if [ "X${HA_CLUSTERNAME}" != "X" ]; then
            HA_STRING=`${HA_PRIVCMDSPATH}/${HA_RESOURCEQUERYCMD} \
                        _CDB_DB=$HA_CDB _RESOURCE=$2 _RESOURCE_TYPE=$1  \
                        _NO_LOGGING=true _CLUSTER=${HA_CLUSTERNAME}`
        else
            HA_STRING=`${HA_PRIVCMDSPATH}/${HA_RESOURCEQUERYCMD} \
                        _CDB_DB=$HA_CDB _RESOURCE=$2 _RESOURCE_TYPE=$1  \
                        _NO_LOGGING=true`
        fi

        ha_exit_code=$?

        if [ $ha_exit_code -ne 0 ]; then
            ${HA_LOG} "${HA_RESOURCEQUERYCMD}: resource name $2 resource type $1" \
            ${HA_LOG} "Failed with error: ${HA_STRING}";
        fi

        if [ $ha_exit_code -ne $HA_RETRY_CMD_ERR ]; then
            break;
        fi

        ha_retry_count=`expr $ha_retry_count + 1`

    done

    return ${ha_exit_code};
}
```

The `ha_get_info` is a faster version of `ha_get_info_debug()`.

```
ha_get_info_debug()
{
```

```
    if [ "X${HA_CLUSTERNAME}" != "X" ]; then
        HA_STRING=`${HA_PRIVCMDSPATH}/${HA_RESOURCEQUERYCMD} \
                    _CDB_DB=$HA_CDB _RESOURCE=$2 _RESOURCE_TYPE=$1      \
                    _CLUSTER=${HA_CLUSTERNAME}`
    else
        HA_STRING=`${HA_PRIVCMDSPATH}/${HA_RESOURCEQUERYCMD} \
                    _CDB_DB=$HA_CDB _RESOURCE=$2 _RESOURCE_TYPE=$1`
    fi
    ha_exit_code=$?

    if [ $? -ne 0 ]; then
        ${HA_LOG} "${HA_RESOURCEQUERYCMD}: resource name $2 resource type $1"
        ${HA_LOG} "Failed with error: ${HA_STRING}";
    fi

    return ${ha_exit_code};
}
```

Print Exclusivity Check Messages

The `ha_print_exclusive_status()` function prints exclusivity check messages to the log file. `$1` is the resource name and `$2` is the exit status.

```
ha_print_exclusive_status()
{
    if [ $? -eq $HA_NOT_RUNNING ]; then
        ${HA_LOG} "resource $1 exclusive status: NOT RUNNING"
    else
        ${HA_LOG} "resource $1 exclusive status: RUNNING"
    fi
}
```

The `ha_print_exclusive_status_all_resources()` function is similar, but it prints exclusivity check messages for all resources. `$HA_RES_NAMES` is the list of resource names.

```
ha_print_exclusive_status_all_resources()
{
    for HA_RES in $HA_RES_NAMES
    do
        ha_print_exclusive_status ${HA_RES} $1
    done
}
```

Glossary

action scripts

The set of scripts that determine how a resource is started, monitored, and stopped. There must be a set of action scripts specified for each resource type. The possible set of action scripts is: `probe`, `exclusive`, `start`, `stop`, `monitor`, and `restart`.

cluster

A collection of one or more *cluster nodes* coupled to each other by networks or other similar interconnections. A cluster is identified by a simple name; this name must be unique within the *pool* A particular node may be a member of only one cluster.

cluster administrator

The person responsible for managing and maintaining an IRIS FailSafe cluster.

cluster configuration database

Contains configuration information about all resources, resource types, resource groups, failover policies, nodes, and clusters.

cluster node

A single IRIX image. Usually, a cluster node is an individual computer. The term *node* is also used in this guide for brevity; this use of node does not have the same meaning as a node in an Origin system.

control messages

Messages that cluster software sends between the cluster nodes to request operations on or distribute information about cluster nodes and resource groups. IRIS FailSafe sends control messages for the purpose of ensuring nodes and groups remain highly available. Control messages and heartbeat messages are sent through a node's network interfaces that have been attached to a control network. A node can be attached to multiple control networks.

A node's control networks should not be set to accept control messages if the node is not a dedicated IRIS FailSafe node. Otherwise, end users who run non-IRIS FailSafe jobs on the machine can have their jobs killed unexpectedly when IRIS FailSafe resets the node.

control network

The network that connects nodes through their network interfaces (typically Ethernet) such that IRIS FailSafe can maintain a cluster's high availability by sending heartbeat messages and control messages through the network to the attached nodes. IRIS FailSafe uses the highest priority network interface on the control network; it uses a network interface with lower priority when all higher-priority network interfaces on the control network fail.

A node must have at least one control network interface for heartbeat messages and one for control messages (both heartbeat and control messages can be configured to use the same interface). A node can have no more than eight control network interfaces.

dependency list

See *resource dependency* or *resource type dependency*.

failover

The process of allocating a *resource group* to another *node* according to a *failover policy* A failover may be triggered by the failure of a resource, a change in the node membership (such as when a node fails or starts), or a manual request by the administrator.

failover attribute

A string that affects the allocation of a resource group in a cluster. The administrator must specify system-defined attributes (such as `Auto_Failback` or `Controlled_Failback`), and can optionally supply site-specific attributes.

failover domain

The ordered list of nodes on which a particular *resource group* can be allocated. The nodes listed in the failover domain must be within the same cluster; however, the failover domain does not have to include every node in the cluster.The administrator defines the *initial failover domain* when creating a failover policy. This list is transformed into the *run-time failover domain* by the *failover script* the run-time failover domain is what is actually used to select the failover node. IRIS FailSafe stores the run-time failover domain and uses it as input to the next failover script invocation. The initial and run-time failover domains may be identical, depending upon the contents of the failover script. In general, IRIS FailSafe allocates a given resource group to the first node listed in the run-time failover domain that is also in the node

membership; the point at which this allocation takes place is affected by the *failover attributes*.

failover policy

The method used by IRIS FailSafe to determine the destination node of a failover. A failover policy consists of a *failover domain*, *failover attributes*, and a *failover script*. A failover policy name must be unique within the *pool*.

failover script

A failover policy component that generates a *run-time failover domain* and returns it to the IRIS FailSafe process. The IRIS FailSafe process applies the failover attributes and then selects the first node in the returned failover domain that is also in the current node membership.

heartbeat messages

Messages that cluster software sends between the nodes that indicate a node is up and running. Heartbeat messages and *control messages* are sent through a node's network interfaces that have been attached to a control network. A node can be attached to multiple control networks.

heartbeat interval

Interval between heartbeat messages. The node timeout value must be at least 10 times the heartbeat interval for proper IRIS FailSafe operation (otherwise false failovers may be triggered). The higher the number of heartbeats (smaller heartbeat interval), the greater the potential for slowing down the network. Conversely, the fewer the number of heartbeats (larger heartbeat interval), the greater the potential for reducing availability of resources.

initial failover domain

The ordered list of nodes, defined by the administrator when a failover policy is first created, that is used the first time a cluster is booted.The ordered list specified by the initial failover domain is transformed into a *run-time failover domain* by the *failover script*; the run-time failover domain is used along with failover attributes to determine the node on which a resource group should reside. With each failure, the failover script takes the current run-time failover domain and potentially modifies it; the initial failover domain is never used again. Depending on the run-time conditions

and contents of the failover script, the initial and run-time failover domains may be identical. See also *run-time failover domain*.

key/value attribute

A set of information that must be defined for a particular resource type. For example, for the resource type `filesystem` one key/value pair might be *mount_point=/fs1* where *mount_point* is the key and *fs1* is the value specific to the particular resource being defined. Depending on the value, you specify either a `string` or `integer` data type. In the previous example, you would specify `string` as the data type for the value *fs1*.

log configuration

A log configuration has two parts: a *log level* and a *log file*, both associated with a *log group*. The cluster administrator can customize the location and amount of log output, and can specify a log configuration for all nodes or for only one node. For example, the `crsd` log group can be configured to log detailed level-10 messages to the `/var/cluster/ha/log/crsd-foo` log only on the node *foo* and to write only minimal level-1 messages to the `crsd` log on all other nodes.

log file

A file containing IRIS FailSafe notifications for a particular *log group*. A log file is part of the *log configuration* for a log group. By default, log files reside in the `/var/cluster/ha/log` directory, but the cluster administrator can customize this. Note: IRIS FailSafe logs both normal operations and critical errors to `/var/adm/SYSLOG`, as well as to individual logs for specific log groups.

log group

A set of one or more IRIS FailSafe processes that use the same log configuration. A log group usually corresponds to one IRIS FailSafe daemon, such as `gcd`.

log level

A number controlling the number of log messages that IRIS FailSafe will write into an associated log group's log file. A log level is part of the log configuration for a log group.

node

See *cluster node*

node ID

A 16-bit positive integer that uniquely defines a cluster node. During node definition, IRIS FailSafe will assign a node ID if one has not been assigned by the cluster administrator. Once assigned, the node ID cannot be modified.

node membership

The list of nodes in a cluster on which IRIS FailSafe can allocate resource groups.

node timeout

If no heartbeat is received from a node in this period of time, the node is considered to be dead. The node timeout value must be at least 10 times the heartbeat interval for proper IRIS FailSafe operation (otherwise false failovers may be triggered).

notification command

The command used to notify the cluster administrator of changes or failures in the cluster, nodes, and resource groups. The command must exist on every node in the cluster.

offline resource group

A resource group that is not highly available in the cluster. To put a resource group in offline state, IRIS FailSafe stops the group (if needed) and stops monitoring the group. An offline resource group can be running on a node, yet not under IRIS FailSafe control. If the cluster administrator specifies the *detach only* option while taking the group offline, then IRIS FailSafe will not stop the group but will stop monitoring the group.

online resource group

A resource group that is highly available in the cluster. When IRIS FailSafe detects a failure that degrades the resource group availability, it moves the resource group to another node in the cluster. To put a resource group in online state, IRIS FailSafe starts the group (if needed) and begins monitoring the group. If the cluster administrator specifies the *attach only* option while bringing the group online, then IRIS FailSafe will not start the group but will begin monitoring the group.

owner host

A system that can control an IRIS FailSafe node remotely, such as power-cycling the node. Serial cables must physically connect the two systems through the node's system controller port. At run time, the owner host must be defined as a node in the IRIS FailSafe pool.

owner TTY name

The device file name of the terminal port (TTY) on the *owner host* to which the system controller serial cable is connected. The other end of the cable connects to the IRIS FailSafe node with the system controller port, so the node can be controlled remotely by the owner host.

pool

The entire set of nodes involved with a group of clusters. The group of clusters are usually close together and should always serve a common purpose. A replicated database is stored on each node in the pool.

port password

The password for the system controller port, usually set once in firmware or by setting jumper wires. (This is not the same as the node's root password.)

powerfail mode

When powerfail mode is turned `on` IRIS FailSafe tracks the response from a node's system controller as it makes reset requests to a cluster node. When these requests fail to reset the node successfully, IRIS FailSafe uses heuristics to try to estimate whether the machine has been powered down. If the heuristic algorithm returns with success, IRIS FailSafe assumes the remote machine has been reset successfully. When powerfail mode is turned `offq` the heuristics are not used and IRIS FailSafe may not be able to detect node power failures.

process membership

A list of process instances in a cluster that form a process group. There can multiple process groups per node.

resource

A single physical or logical entity that provides a service to clients or other resources. For example, a resource can be a single disk volume, a particular network address, or an application such as a web server. A resource is generally available for use over time on two or more nodes in a *cluster*, although it can be allocated to only one node at any given time. Resources are identified by a *resource name* and a *resource type*. Dependent resources must be part of the same *resource group* and are identified in a *resource dependency list*

resource dependency

The condition in which a resource requires the existence of other resources.

resource dependency list

A list of resources upon which a resource depends. Each resource instance must have resource dependencies that satisfy its resource type dependencies before it can be added to a resource group.

resource group

A collection of resources. A resource group is identified by a simple name; this name must be unique within a cluster. Resource groups cannot overlap; that is, two resource groups cannot contain the same resource. All interdependent resources must be part of the same resource group. If any individual resource in a resource group becomes unavailable for its intended use, then the entire resource group is considered unavailable. Therefore, a resource group is the unit of failover for IRIS FailSafe.

resource keys

Variables that define a resource of a given resource type. The action scripts use this information to start, stop, and monitor a resource of this resource type.

resource name

The simple name that identifies a specific instance of a *resource type*. A resource name must be unique within a given resource type.

resource type

A particular class of *resource*. All of the resources in a particular resource type can be handled in the same way for the purposes of *failover*. Every resource is an instance of

exactly one resource type. A resource type is identified by a simple name; this name must be unique within a cluster. A resource type can be defined for a specific node or for an entire cluster. A resource type that is defined for a node overrides a cluster-wide resource type definition with the same name; this allows an individual node to override global settings from a cluster-wide resource type definition.

resource type dependency

A set of resource types upon which a resource type depends. For example, the *filesystem* resource type depends upon the*ensp;volume* resource type, and the *Netscape_web* resource type depends upon the *filesystem* and *IP_address* resource types.

resource type dependency list

A list of resource types upon which a resource type depends.

run-time failover domain

The ordered set of nodes on which the resource group can execute upon failures, as modified by the *failover script*. The run-time failover domain is used along with failover attributes to determine the node on which a resource group should reside. See also *initial failover domain*.

start/stop order

Each resource type has a start/stop order, which is a nonnegative integer. In a resource group, the start/stop orders of the resource types determine the order in which the resources will be started when IRIS FailSafe brings the group online and will be stopped when IRIS FailSafe takes the group offline. The group's resources are started in increasing order, and stopped in decreasing order; resources of the same type are started and stopped in indeterminate order. For example, if resource type *volume* has order 10 and resource type *filesystem* has order 20, then when IRIS FailSafe brings a resource group online, all volume resources in the group will be started before all file system resources in the group.

system controller port

A port sitting on a node that provides a way to power-cycle the node remotely. Enabling or disabling a system controller port in the cluster configuration database (CDB) tells IRIS FailSafe whether it can perform operations on the system controller port. (When the port is enabled, serial cables must attach the port to another node, the owner host.) System controller port information is optional for a node in the pool,

but is required if the node will be added to a cluster; otherwise resources running on that node never will be highly available.

tie-breaker node

A node identified as a tie-breaker for IRIS FailSafe to use in the process of computing node membership for the cluster, when exactly half the nodes in the cluster are up and can communicate with each other. If a tie-breaker node is not specified, IRIS FailSafe will use the node with the lowest node ID in the cluster as the tie-breaker node.

type-specific attribute

Required information used to define a resource of a particular resource type. For example, for a resource of type *filesystem* you must enter attributes for the resource's volume name (where the file system is located) and specify options for how to mount the file system (for example, as readable and writable).

Index