IRIS FailSafe™ Version 2
Programmer's Guide

007–3900–006

CONTRIBUTORS

Written by Lori Johnson

Illustrated by Chrystie Danzer, Dany Galgani, and Chris Wengelski

Edited by Susan Wilkening

Production by Glen Traefald

Engineering contributions by Scott Henry, Vidula Iyer, Herb Lewis, Michael Nishimoto, Kevan Rehm, Hugh Shannon Jr., Bill Sparks, Paddy Sreenivasan, Dan Stekloff, Rebecca Underwood, and Manish Verma

Cover design by Sarah Bolles, Sarah Bolles Design, and Danny Galgani, SGI Technical Publications

New Features in This Guide

This update includes clarifications to terminology and other editorial changes.

Record of Revision

Version	Description
002	December 1999 Published in conjunction with the latest IRIS FailSafe 2.0 rollup patch. It supports IRIX 6.5.9 and later.
003	October 2000 Supports the IRIS FailSafe 2.1 release.
004	April 2001 Supports the IRIS FailSafe 2.1.1 release and IRIX 6.5.12 or later.
005	November 2001 Supports the IRIS FailSafe 2.1.2 release and IRIX 6.5.14 or later.
006	April 2002 Supports the IRIS FailSafe 2.1.3 release and IRIX 6.5.16 or later.

Contents

Figures

Tables

About This Guide

This guide explains how to write your own *plug-in*, the set of scripts that are required to turn an application into a highly available service in conjunction with IRIS FailSafe 2.1.3 software. It also tells you how to create a new resource type and provides instructions for migrating script information from IRIS FailSafe release 1.2 to release 2.*x*.

This guide assumes that the IRIS FailSafe system has been configured as described in the *IRIS FailSafe Version 2 Administrator's Guide*.

This guide supports IRIX 6.5.16 and later.

Audience

This guide is written for system programmers who are writing their own plug-ins for the IRIS FailSafe system. These scripts allow the failover of applications that are not handled by the base and optional plug-ins. Readers 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 Version 2 Administrator's Guide*.

Related Documentation

The following documentation is of interest:

- *IRIS FailSafe Version 2 Administrator's Guide*
- *CXFS Version 2 Software Installation and Administration Guide*

The man pages are as follows:

- `cdbBackup`(1M)
- `cdbRestore`(1M)
- `cmgr`(1M)
- `crsd`(1M)
- `failsafe`(7M)

- `fs2d`(1M)
- `ha_cilog`(1M)
- `ha_cmsd`(1M)
- `ha_cxfs`(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_services`	Cluster 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

Conventions Used in This Guide

This guide uses *FailSafe* as an abbreviation for *IRIS FailSafe*.

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, or highlighting of differences between releases
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)

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. Be sure to include the title and document number of the manual with your comments. (Online, the document number is located in the front matter of the manual. In printed manuals, the document number is located at the bottom of each page.)

You can contact us in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Use the Feedback option on the Technical Publications Library World Wide Web page:

  http://techpubs.sgi.com

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.

- Send mail to the following address:

  Technical Publications
  SGI
  1600 Amphitheatre Pkwy., M/S 535
  Mountain View, California 94043–1351

- Send a fax to the attention of “Technical Publications” at +1 650 932 0801.

We value your comments and will respond to them promptly.

Introduction

IRIS FailSafe 2.1.3 provides highly available services for as many as eight nodes in a cluster. 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 to write your own *plug-in*, the set of scripts that are required to turn an application into a highly available service in conjunction with IRIS FailSafe software.

This chapter contains the following:

For an overview of the components, software layers, communication paths, and order of execution of action and failover scripts, see *IRIS FailSafe Version 2 Administrator's Guide*.

Note: This guide uses *FailSafe* as an abbreviation for *IRIS FailSafe*.

Terminology

This section defines the terminology necessary to configure and monitor highly available services with FailSafe.

Cluster

The *cluster* is the set of systems (nodes) configured to work together as a single computing resource. The cluster is identified by a simple name; this name must be unique within the pool. (For example, you cannot use the same name for the cluster and for a node.)

Node

A *node* is an operating system (OS) image, usually an individual computer. The nodes are connected to a storage area network (SAN) that connects the storage systems to the nodes in the cluster. A node can belong to only one cluster.

This use of the term node does not have the same meaning as a node in an SGI Origin 3000 or SGI 2000 system.

Pool

The *pool* is the set of nodes from which a particular cluster may be formed. Only one cluster may be configured from a given pool, and it need not contain all of the available nodes. (Other pools may exist, but each is disjoint from the other. They share no node or cluster definitions.)

Figure 1-1 shows the concepts of pool and cluster.

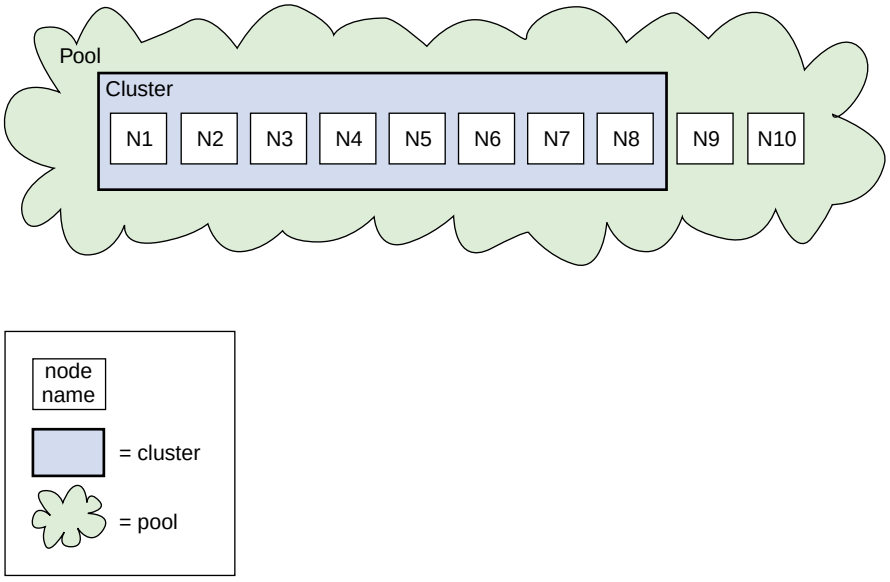

Figure 1-1 Pool and Cluster Concepts

Cluster Database

The *cluster database* (CDB) contains configuration information about all nodes and the cluster. The `fs2d` daemon manages the distribution of the cluster database across the nodes in the pool.

Membership

There are the following types membership:

- *FailSafe membership* is the list of FailSafe nodes in the cluster on which FailSafe can make resource groups online.
- `fs2d` *database membership* (also known as *user-space membership*) is the group of nodes in the **pool** that are accessible to `fs2d`. The `fs2d` daemon manages the distribution of the cluster database across the nodes in the pool. Nodes available to `fs2d` are able to receive cluster database updates, and are therefore part of the `fs2d` database membership; this may be a subset of the nodes defined in the pool.
- *Process membership* is the list of process instances in a cluster that form a cluster process group. There can be multiple process groups per node.

With CXFS coexecution, there is also *CXFS membership*. For more information about CXFS, see *CXFS Version 2 Software Installation and Administration Guide*.

Quorum

The *quorum* is the number of nodes required to form a cluster, which differs according to membership:

- For FailSafe membership, **more than 50%** of the nodes in the cluster must be in a known state (successfully reset or talking to each other using heartbeat control networks) and at least 50% of the nodes can talk to each other.
- For `fs2d` database membership, **50%** of the **nodes in the pool** are required to form and maintain a cluster.

No weighting is used, as opposed to CXFS membership.

Private Network

A *private network* is one that is **dedicated** to cluster communication and is accessible by administrators but not by users:

- The cluster software uses the private network to send the heartbeat/control messages necessary for the cluster configuration to function. If there are delays in receiving heartbeat messages, the cluster software may determine that a node is not responding and remove it from the FailSafe membership. The node will be reset.
- Rebooting network equipment can cause the nodes in a cluster to lose communication; the cluster will move into a degraded state if communication between nodes is lost. Using a private network limits the traffic on the network and therefore will help avoid unnecessary resets or disconnects. Also, because the messaging protocol does not prevent snooping or spoofing, a network with restricted access is safer than one with user access.

Therefore, because the performance and security characteristics of a public network could cause problems in the cluster and because heartbeat is very timing-dependent (even small variations can cause problems), **a private network is required**.

In addition, SGI recommends that all nodes be on the same local network segment.

Note: If there are any network issues on the private network, fix them before trying to use FailSafe.

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.

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.

The 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 and their corresponding resource types for a resource group named `WebGroup.`

Table 1-1 Example Resource Group

Resource	Resource Type
`10.10.48.22`	`IP_address`
`/fs1`	`filesystem`
`vol1`	`volume`
`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.

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

Dependency

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 dependencies are verified when resources are added to a resource group, not when resources are defined.

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. A *resource type dependency list* details the resource types upon which a resource type depends.

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 `WS1` is defined with the `Netscape_web` resource type, then the resource group containing `WS1` must also contain at least one resource of the type `IP_address` and one resource of the type `volume`. This is shown in Figure 1-2.

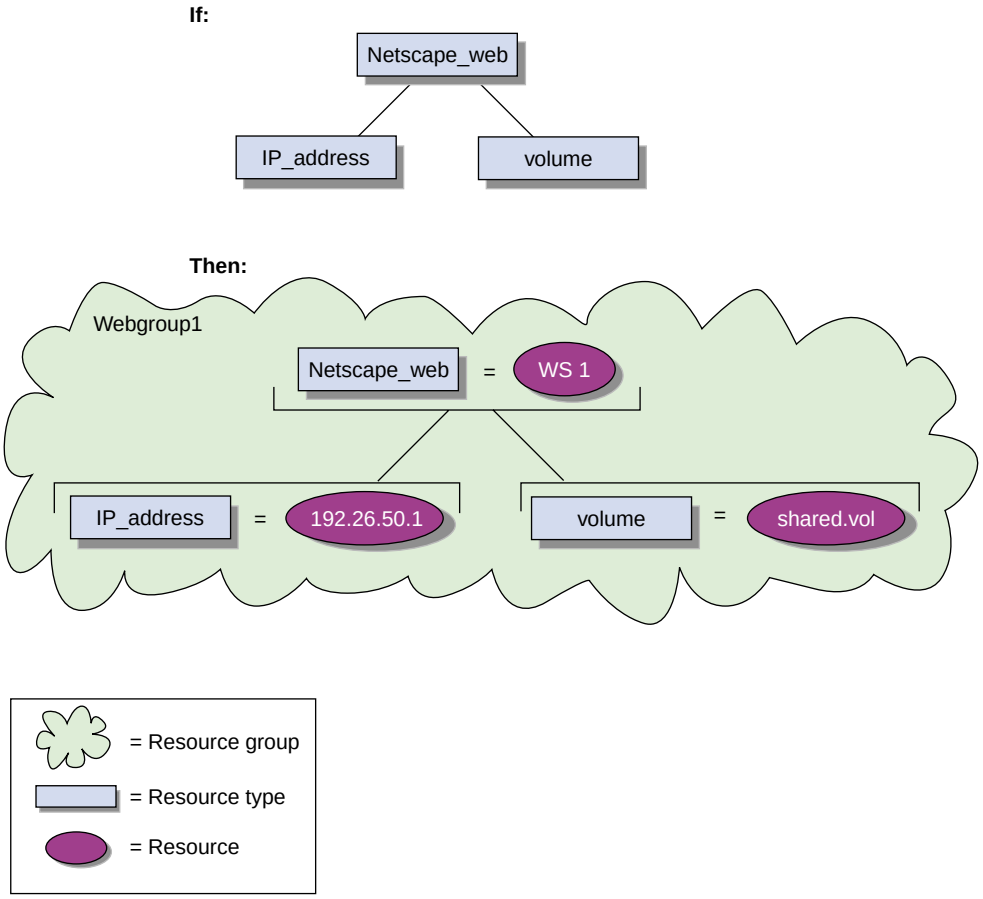

Figure 1-2 Resource Type Dependencies

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 FailSafe 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 FailSafe to determine the destination node of a failover. A failover policy consists of the following:

- Failover domain
- Failover attributes
- Failover script

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; FailSafe uses the run-time failover domain along with failover attributes and the FailSafe membership to determine the node on which a resource group should reside. 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, FailSafe allocates a given resource group to the first node listed in the run-time failover domain that is also in the FailSafe 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 `ha_fsd` process. The (`ha_fsd`) process applies the failover attributes and then selects the first node in the returned failover domain that is also in the current FailSafe membership.

The following failover scripts are provided with the 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 provided in the *IRIS FailSafe Version 2 Programmer's 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 type:

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

The release 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 provided in the *IRIS FailSafe Version 2 Programmer's Guide*.

Plug-In

A *plug-in* is the set of software required to make an application highly available, including a resource type and action scripts. There are plug-ins provided with the base FailSafe release, optional plug-ins available for purchase from SGI, and customized plug-ins you can write using the instructions in this guide. See "Plug-ins", page 10.

Cluster Process Group

A *cluster process group* is a group of application instances in a distributed application that cooperate to provide a service. Each application instance can consist of one or more UNIX processes and spans only one node.

For example, distributed lock manager instances in each node would form a process group. By forming a process group, they can obtain process membership and reliable, ordered, atomic communication services.

Note: There is no relationship between a UNIX process group and a cluster process group.

Plug-ins

There are provided plug-ins, optional plugs-ins available for purchase, and customized plug-ins you create according to the instructions in this book. Table 1-2 shows the provided and optional FailSafe plug-ins and their associated resource types.

Table 1-2 Provided and Optional Plug-Ins

Provided Plug-In	Resource Type	Optional Plug-In	Resource Type
CXFS file system	`CXFS`	FailSafe/DMF	`DMF`
IP addresses	`IP_address`	FailSafe/NFS	`NFS` and `statd_unlimited`
MAC addresses	`MAC_address`	FailSafe/Informix	`INFORMIX_DB`
XFS file systems	`filesystem`	FailSafe/Oracle	`Oracle_DB`

Provided Plug-In	Resource Type	Optional Plug-In	Resource Type
XLV logical volumes	`volume`	FailSafe/Samba	`Samba`
		FailSafe/TMF	`TMF`
		FailSafe/Web (Netscape)	`Netscape_web`

See the release notes for information about the specific releases of these products that are supported.

If you want to create your own plug-in, 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 applications can be made highly available; see "Characteristics that Permit an Application to be Highly Available", page 11.

Note: If you require a customized plug-in but do not want to write it 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`.

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 application fails over, the instances of the application are stopped on one node using the stop procedure and restarted on the other node using start procedure.

  Avoid applications that are started as a daemon from `/etc/inetd.conf` because typically everything in `/etc/inetd.conf` is already running. Trying to automatically edit `/etc/inetd.conf` could cause errors for other daemons started by this file.

Many applications will have a start and stop procedure that belongs in the `/etc/init.d` directory. You can incorporate them into a custom `/var/ha/resources` script to appropriately start and stop the application. If the application also has a `chkconfig`(1m) flag, set it to `off`. The `chkconfig` flag should be set to `on` in the `/var/ha/resources start` script.

- The application does not depend on the `hostname` or any identifier that is specific to a node.
- 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.

  Ensure that there is no affinity for a specific node.
- The application does not depend on knowing the primary host name (as returned by `hostname`); 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 FailSafe and its optional products (see "Plug-ins", page 10), 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

To make an application highly available, follow these steps:

1. Understand the application and determine the following:
   - 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 Version 2 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 best suits this application.

- The number of instances of the resource type that will constitute the application. (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 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 15.
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 61.
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", page 61.
5. Configure the following in the cluster configuration database (for more information, see the *IRIS FailSafe Version 2 Administrator's Guide*):
   - Resource group
   - Resource type
   - Failover policy
6. Test the action scripts and failover script. See Chapter 5, "Testing Scripts", page 85, and "Debugging Notes", page 86.

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

Administrative Commands for Use in Scripts

Table 1-3 shows the administrative commands available with FailSafe for use in scripts.

Table 1-3 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 which allows command execution to be serialized. The lock file prevents multiple instances of the same command from executing at the same time on a single 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 describes how to write the action scripts required for a plug-in 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 "Understanding the Execution of Action Scripts", page 16.

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

- `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 application must have a restart ability on the same node in case of failure. However, the `restart` script may contain only a return-success function.

Understanding the Execution of Action Scripts

Before you can write a new action script, you must understand how action scripts are executed. This section covers the following topics:

When Action Scripts are Executed

Table 2-1 shows when action scripts are executed.

Table 2-1 Execution of Action Scripts

Script	Execution Conditions
`exclusive`	A resource group is made online by the user
	High-availability (HA) processes (`ha_cmsd`, `ha_gcd`, `ha_fsd`, `ha_srmd`, `ha_ifd`) are started
`start`	A resource group is made online by the user
	HA processes are started
	A resource group fails over
`stop`	A resource group is made offline
	HA processes are stopped
	A resource group fails over
	A node is shutdown or rebooted
`monitor`	A resource groups is online
`restart`	The `monitor` script fails

Multiple Instances of a Script Executed at the Same Time

Multiple instances of the same script may be executed at the same time. To avoid this problem, you can use the `ha_filelock` 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 the following 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 the lock on the file.

Differences between the `exclusive` and `monitor` Scripts

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

Table 2-2 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.)
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.

Successful Execution of Action Scripts

Table 2-3 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 FailSafe Manager graphical user interface (GUI) or the `cmgr` command.

Table 2-3 Successful Action Script Results

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

Failure of Action Scripts

Table 2-4 shows the state of the resource group and the error state when an action script fails. (There are no `offline` states with errors.)

Table 2-4 Failure of an Action Script

Failing Script	Resource Group State	Error State
`exclusive`	`online`	`exclusivity`
`monitor`	`online`	`monitoring failure`
`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*

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 twice 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 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 Appendix C, "Using the Script Library", page 103.
- 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:
  - `cmgr`(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 monitoring is required (see "Is Monitoring Necessary?", page 22). However, even if monitoring is not needed, a `monitor` script is still required; in this case, it can contain only a return-success function.
- Determine if a resource type must be added to the cluster database.
- Understand the vendor-supplied startup and shutdown procedures.
- Determine the configuration parameters for the application; these may be used in the action script and should be stored in the cluster database. Action scripts may read from the database.
- Determine whether the resource type can be restarted in the local node and whether this action makes sense.

Is Monitoring Necessary?

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

- Heartbeat monitoring is sufficient; that is, simply verifying that the node is alive (provided automatically by the base 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.
- A resource on which the application 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 base software, additional monitoring is not required.

Caution: Beware that monitoring should be as lightweight as possible so that it does not affect system performance. Also, security issues may make monitoring difficult. If you are unable to provide a monitoring script with appropriate performance and security, consider a monitoring agent; see "Monitoring Agents", page 41.

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 cluster database. Ensure that the monitoring level chosen does not affect system performance. For more information, see the *IRIS FailSafe Version 2 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 monitoring interval time and time-out value for the `monitor` script. The time-out must be long enough to guarantee that occasional anomalies do not cause false failovers. It will be useful for you to determine the peak load that resource may need to sustain.

You must also determine if the `monitor` test should execute multiple times so that an application 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/server resource that follows a well-defined protocol, the `monitor` script can make a simple request and verify that the proper response is received.
- For a web server application, 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`(1m) command to check if a process is running.

  The `ha_exec2` command differs from `killall`(1m) 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`(1) 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 template scripts have the same general format. Following is the type of information in the order in which it appears in the template scripts:

- 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 rest of this chapter discusses Korn shell.

The following sections show an example from the NFS `start` script.

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: 1.20 $"

# 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 cluster 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 Appendix C, "Using the Script Library", page 103.

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
}
```

Call `ha_get_info` with a third argument of any value to obtain all attributes and dependency information for a resource from the cluster database. Use `ha_get_multi_fields` to retrieve specific dependency information. The resource dependency information is returned in the `$HA_FIELD_VALUE` variable.

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()
{
${HA_DBGLOG} "Exit: 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_template()

# for all template resources passed as parameter
for TEMPLATE in $HA_RES_NAMES
do
    #HA_CMD="command to start $TEMPLATE resource on the local machine";

    #ha_execute_cmd "string to describe the command being executed";

    ha_write_status_for_resource $TEMPLATE $HA_SUCCESS;
done
}
```

Note: When testing the script, you will add the following line to this area to obtain debugging information:

```
set -x
```

For examples of this area, see "Examples of Action Scripts", page 31.

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_args() 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_args() section 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. This function is defined in the scriptlib library. See "Read an Input File", page 110.

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

```
# Read the input file and store the resource names in $HA_RES_NAMES
# variable

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 `script.$$` 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()
{
${HA_DBGLOG} "Entry: start_nfs()";

# for all nfs resources passed as parameter
for resource in ${HA_RES_NAMES}
do
NFSFILEDIR=${HA_SCRIPTTMPDIR}/${LOCAL_TEST_KEY}$resource
HA_CMD="/sbin/mkdir -p $NFSFILEDIR";
ha_execute_cmd "creating nfs status file directory";
```

```
if [ $? -ne 0 ]; then
   ${HA_LOG} "Failed to create ${NFSFILEDIR} directory";
   ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
   exit_script $HA_NOCFGINFO
fi

get_nfs_info $resource
if [ $? -ne 0 ]; then
    ${HA_LOG} "NFS: $resource parameters not present in CDB";
    ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
    exit_script ${HA_NOCFGINFO};
fi

ha_get_field "${HA_STRING}" export-info
if [ $? -ne 0 ]; then
    ${HA_LOG} "NFS: export-info not present in CDB for resource $resource";
    ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
    exit_script ${HA_NOCFGINFO};
fi
export_opts="$HA_FIELD_VALUE"

ha_get_field "${HA_STRING}" filesystem
if [ $? -ne 0 ]; then
    ${HA_LOG} "NFS: filesystem-info not present in CDB for resource
$resource";
    ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
    exit_script ${HA_NOCFGINFO};
fi
filesystem="$HA_FIELD_VALUE"
# Make the script idempotent, check to see if the NFS resource
# is already exported, if so return success. Remember that we
# might not have any export options.
retstat=0;
# Check to see if the NFS resource is already exported
# (without options)
/usr/etc/exportfs | grep "$resource$" >/dev/null 2>&1
retstat=$?
if [ $retstat -eq 1 ]; then
    # Check to see if the NFS resource is already exported
    # with options.
    /usr/etc/exportfs | grep "$resource " | grep "$export_opts$" >/dev/null 2>&1
```

```
    retstat=$?
fi
if [ $retstat -eq 1 ]; then
    # Before we try and export the NFS resource, make sure
    # filesystem is mounted.
    HA_CMD="/sbin/grep $filesystem /etc/mtab > /dev/null 2>&1";
    ha_execute_cmd "check if the filesystem $filesystem is mounted";
    if [ $? -eq 0 ]; then
 HA_CMD="/usr/etc/exportfs -i -o $export_opts $resource";
 ha_execute_cmd "export $resource 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} "NFS: filesystem $filesystem not mounted"
 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()
{
```

```
${HA_DBGLOG} "Entry: 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_LOG} "NFS: $resource parameters not present in CDB";
    ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
    exit_script ${HA_NOCFGINFO};
fi

ha_get_field "${HA_STRING}" export-info
if [ $? -ne 0 ]; then
    ${HA_LOG} "NFS: export-info not present in CDB for resource $resource";
    ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
    exit_script ${HA_NOCFGINFO};
fi
export_opts="$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 "$resource$" >/dev/null 2>&1
retstat=$?
if [ $retstat -eq 1 ]; then
    # Check to see if the filesystem is already exported
    # with options.
    /usr/etc/exportfs | grep "$resource " | 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 $resource /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 $resource";
 ha_execute_cmd "unexport $resource 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} "NFS: filesystem $resource not found in export filesystem list, \
 unexporting anyway";
 HA_CMD="/usr/etc/exportfs -u $resource";
 ha_execute_cmd "unexport $resource 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=${HA_SCRIPTTMPDIR}/${LOCAL_TEST_KEY}$resource
HA_CMD="/sbin/rm -rf $NFSFILEDIR";
ha_execute_cmd "removing nfs status file directory";
if [ $? -ne 0 ]; then
   ${HA_LOG} "Failed to delete ${NFSFILEDIR} directory";
   ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
   exit_script $HA_NOCFGINFO
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()
{
${HA_DBGLOG} "Entry: monitor_nfs()";

for resource in ${HA_RES_NAMES}
do
get_nfs_info $resource
if [ $? -ne 0 ]; then
    # No resource information available.
    ${HA_LOG} "NFS: $resource parameters not present in CDB";
    ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
    exit_script ${HA_NOCFGINFO};
fi

ha_get_field "${HA_STRING}" filesystem
if [ $? -ne 0 ]; then
    # filesystem not available available.
    ${HA_LOG} "NFS: filesystem not present in CDB for resource $resource";
    ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
    exit_script ${HA_NOCFGINFO};
fi
fs="$HA_FIELD_VALUE";


# Check to see if the filesystem is mounted
HA_CMD="/sbin/mount | grep $fs  >> /dev/null 2>&1"
ha_execute_cmd "check to see if $fs is mounted"
if [ $? -ne 0 ]; then
    ${HA_LOG} "NFS: $fs not mounted";
    ha_write_status_for_resource ${resource} ${HA_CMD_FAILED};
    exit_script $HA_CMD_FAILED;
```

```
fi

# stat the filesystem
HA_CMD="/sbin/stat $resource";
ha_execute_cmd "stat mount point $resource"
if [ $? -ne 0 ]; then
    ${HA_LOG} "NFS: cannot stat $resource NFS export point";
    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 $resource"
ha_execute_cmd " check the filesystem $resource is exported"
if [ $? -ne 0 ]; then
    ${HA_LOG} "NFS: failed to find $resource in exported filesystem list:-"
    ${HA_LOG} "'/sbin/cat ${EXPORTFS}'"
    rm -f $EXPORTFS;
    ha_write_status_for_resource ${resource} ${HA_CMD_FAILED};
    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=${HA_SCRIPTTMPDIR}/${LOCAL_TEST_KEY}$resource/.nfsstat
NFS_STAT='/usr/etc/nfsstat -rs | /usr/bin/tail -1 | /usr/bin/awk '{print $1}''
if [ ! -f $NFSFILE ]; then
   ${HA_LOG} "NFS: creating stat file $NFSFILE";
   echo $NFS_STAT > $NFSFILE;
   if [ $NFS_STAT -eq 0 ];then
      # do some rpcinfo's
      exec_rpcinfo;
      if [ $? -ne 0 ]; then
  ${HA_LOG} "NFS: exec_rpcinfo failed (1)";
  ha_write_status_for_resource ${resource} ${HA_CMD_FAILED};
  exit_script $HA_CMD_FAILED;
      fi
```

```
   fi
else
   OLD_STAT=‘/sbin/cat $NFSFILE‘
   if test "X${NFS_STAT}" = "X"; then
 ${HA_LOG} "NFS: NFS_STAT is not set, reset to zero";
 NFS_STAT=0;
   fi
   if test "X${OLD_STAT}" = "X"; then
 ${HA_LOG} "NFS: OLD_STAT is not set, reset to zero";
 OLD_STAT=0;
   fi
   if [ $NFS_STAT -gt $OLD_STAT ]; then
 echo $NFS_STAT > $NFSFILE;
   else
 echo $NFS_STAT > $NFSFILE;
 exec_rpcinfo;
 if [ $? -ne 0 ]; then
    ${HA_LOG} "NFS: exec_rpcinfo failed (2)";
    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()
{

${HA_DBGLOG} "Entry: 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_LOG} "NFS: $resource parameters not present in CDB";
    ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
    exit_script ${HA_NOCFGINFO};
fi

SMFILE=${HA_SCRIPTTMPDIR}/showmount.$$
/etc/showmount -x >> ${SMFILE};
HA_CMD="/sbin/grep $resource ${SMFILE} >> /dev/null 2>&1"
ha_execute_cmd "checking for $resource 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()
{
${HA_DBGLOG} "Entry: 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_LOG} "NFS: $resource parameters not present in CDB";
    ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
    exit_script ${HA_NOCFGINFO};
fi

ha_get_field "${HA_STRING}" export-info
if [ $? -ne 0 ]; then
    ${HA_LOG} "NFS: export-info not present in CDB for resource $resource";
    ha_write_status_for_resource ${resource} ${HA_NOCFGINFO};
    exit_script ${HA_NOCFGINFO};
fi
export_opts="$HA_FIELD_VALUE"

HA_CMD="/usr/etc/exportfs -i -o $export_opts $resource";
ha_execute_cmd "export $resource 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 to the `monitor` action script request.

Monitoring agents are independent processes and can be started by the `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 the 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
```

Note: The ACTIONS line above defines what cmond can do to the PROCS processes. These actions must be the same for every agent. (It does not refer to action scripts.)

If you create a new monitoring agent, you must also 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 to list the executables that constitute your new agent. These executables 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. It describes the following topics:

- "Contents of a Failover Policy"
- "Failover Script Interface", page 54
- "Example Failover Policies", page 55

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 attribute
- 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 also be 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.

- In 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 (for the `ordered` failover script, the order will not change); the initial failover domain is never used again. Depending on the run-time conditions such as load and contents of the failover script, the initial and run-time failover domains may be identical.

For example, suppose that the cluster contains three nodes named `N1`, `N2`, and `N3`; that node failure is not the reason for failover; and that the initial failover domain is as follows:

```
N1 N2 N3
```

The runtime failover domain will vary based on the failover script:

- If `ordered`:

  ```
  N1 N2 N3
  ```

- If `round-robin`:

  ```
  N2 N3 N1
  ```

- If a customized failover script, the order could be any permutation, based on the contents of the script:

  ```
  N1 N2 N3                N1 N3 N2
  N2 N1 N3                N2 N3 N1
  N3 N1 N2                N3 N2 N1
  ```

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.

You can specify the following classes of failover attributes:

- Required attributes: either `Auto_Failback` or `Controlled_Failback` (mutually exclusive)
- Optional attributes:
  - `Auto_Recovery` or `InPlace_Recovery` (mutually exclusive)
  - `Critical_RG`
  - `Node_Failures_Only`

Note: The starting conditions for the attributes differs by class:

- For required attributes, a node joins the FailSafe membership when the cluster is already providing highly available services.
- For optional attributes, highly available services are started and the resource group is running in only one node in the cluster.

Table 3-1 describes each attribute.

Table 3-1 Failover Attributes

Class	Name	Description
Required	`Auto_Failback`	Specifies that the resource group is made online based on the failover policy when the node joins the cluster. 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 remains on the same node when a node joins the cluster. 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.
Optional	`Auto_Recovery`	Specifies that the resource group is made online based on the failover policy even when an exclusivity check shows that the resource group is running on a node. 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 is made online on the same node where the resource group is running. 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.

Class	Name	Description
	`Critical_RG`	Allows monitor failure recovery to succeed even when there are resource group release failures. When resource monitoring fails, FailSafe attempts to move the resource group to another node in the application failover domain. If FailSafe fails to release the resources in the resource group, FailSafe puts the Resource group into `srmd executable error` status. If the `Critical_RG` failover attribute is specified in the failover policy of the resource group, FailSafe will reset the node where the release operation failed and move the resource group to another node based on failover policy.
	`Node_Failures_Only`	Allows failover only when there are node failures. This attribute does not have an impact on resource restarts in the local node. The failover does not occur when there is a resource monitoring failure in the resource group. This attribute is useful for customers who are using a hierarchical storage management system such as DMF; in this situation, a customer may want to have resource monitoring failures reported without automatic recovery, allowing operators to perform the recovery action manually if necessary.

Failover Scripts

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

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

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

`ordered`

The `ordered` failover script is provided with the 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
```

round-robin

The `round-robin` failover 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 cluster 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 2.1.*x* 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 (or had) the resource group online.
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 FailSafe 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 FailSafe, which then chooses the node on which the resource group should be allocated by applying the failover attributes and FailSafe membership to the run-time failover domain.

Example Failover Policies

There are two general types of configuration, each of which can have from two through eight 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 three primary nodes and one backup node.

This section shows examples of failover policies for the following types of configuration, each of which can have from two through eight 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.

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.

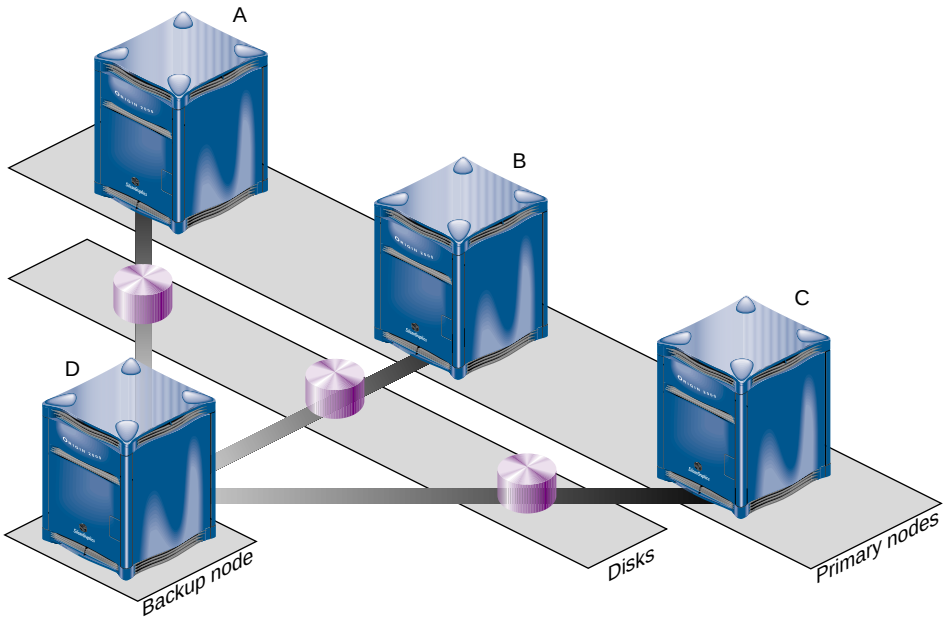

Figure 3-1 *N*+1 Configuration Concept

You could configure the following failover policies for load balancing:

- Failover policy for `RG1`:
  - Initial failover domain = `A`, `D`
  - Failover attribute = `Auto_Failback`, `Critical_RG`
  - 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.

Suppose resource group `RG1` is online on node `A` in the cluster. When the monitor of one of the resources in `RG1` fails, FailSafe attempts to move the resource group to node `D`. If the release of `RG1` from node `A` fails, FailSafe will reset node `A` and allocate the resource group on node `D`. If `Critical_RG` failover attribute was not specified, `RG1` will have an `srmd executable` error.

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.

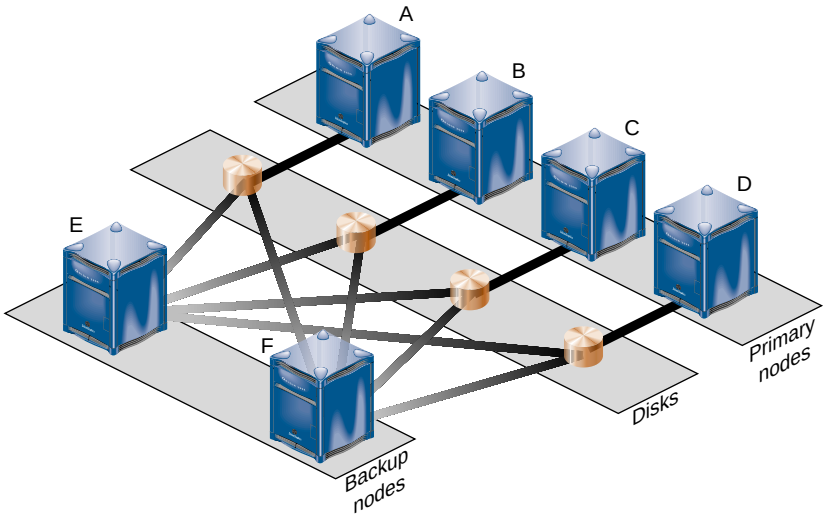

Figure 3-2 *N*+2 Configuration Concept

You could configure the following failover policy for 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.

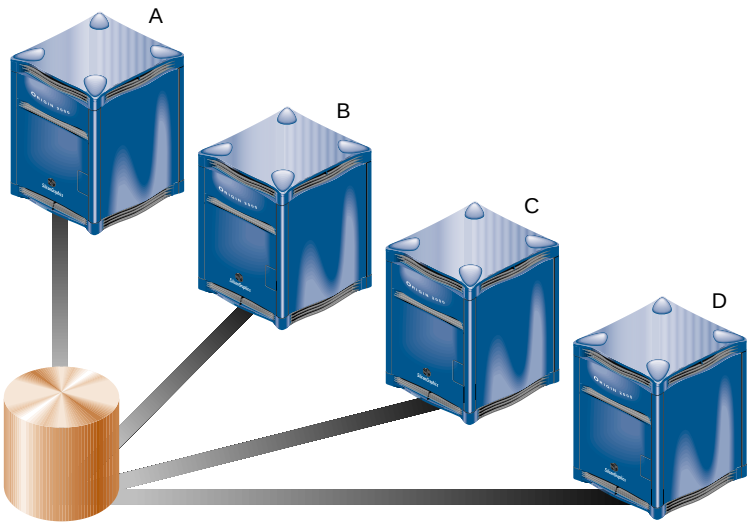

Figure 3-3 *N+M* Configuration Concept

You could configure the following failover policy for 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 describes how to define a new resource type. The following are examples of candidates for resource types:

- Databases that support transactions
- Web servers
- Applications that use user datagram protocol (UDP) for communication with clients

See also "Characteristics that Permit an Application to be Highly Available", page 11.

You will want to create a new resource type when creating something entirely new or when you want to have multiple resource types that are similar except for one or two attributes. For example, if you want to enable local restart for most IP addresses but not for some, you could create a new resource type called `IP_address2` using all of the same information as for the default `IP_address` except changing the value of the restart mode to `1` rather than the default `0`.

This chapter contains the following sections:

Information You Must Gather

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 after 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`
502	`Samba`
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.
- 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
    ```

Copying an Existing Resource Type to Create a New One

Caution: Only `root` can make changes to the cluster database. However, any user can use the GUI to view database information; therefore, you should not include any sensitive information in the cluster database. Users should keep this mind when deciding the list of resource attributes for the resource type.

If an existing resource type is similar to the type you want to create, you can use the following procedure:

1. Log in as `root`.
2. Copy the directory for the existing resource type and give the new directory an appropriate name. For example, to use the `NFS` resource type as the basis for a new resource type named `NFS_CXFS`, do the following:

```
# cp -r /var/cluster/ha/resource_types/NFS /var/cluster/ha/resource_types/NFS_CXFS
```

3. Modify each script in the new `NFS_CXFS` directory so that it uses the name of the new resource type. You must make this modification for the `LOCAL_TEST_KEY=` variable definition; modifying log messages and comments is optional but recommended.

   For example, you would change the following line in the `/var/cluster/ha/resource_types/NFS_CXFS/start` script:

   - From:

     ```
     LOCAL_TEST_KEY=NFS
     ```

   - To:

     ```
     LOCAL_TEST_KEY=NFS_CXFS
     ```

4. Eliminate any unneeded dependencies for the new resource type, using either the GUI or the `cmgr` command.

   For example, you would eliminate the `filesystem` dependency from the new `NFS_CXFS` as follows:

   ```
   # cmgr
   Welcome to SGI Cluster Manager Command-Line Interface
   cmgr> modify resource_type NFS_CXFS in cluster "testcluster"
   Enter commands, when finished enter either "done" or "cancel"
   ```

```
resource_type NFS_CXFS ? remove dependency filesystem
resource_type NFS_CXFS ? done
Successfully modified resource_type NFS_CXFS
```

5. Modify the `monitor` script for the new resource type as needed.

   For example, the difference between the standard `NFS` `monitor` script and the new `NFS_CXFS` `monitor` script is that when you export CXFS filesystems, you do not want FailSafe to check if the filesystem is mounted and to exit with `HA_CMD_FAILED` if it is not. The `NFS_CXFS` `monitor` script itself will determine what action should take place if the filesystem becomes unmounted. To accomplish this, you would modify the `/var/cluster/ha/resource_types/NFS_CXFS/monitor` script to comment out the `exit_script` line in the following section (line modified shown here in bold)

```
# Check to see if the filesystem is mounted
HA_CMD="/sbin/mount | grep $fs >> /dev/null 2>&1"
ha_execute_cmd "check to see if $fs is mounted"
if [ $? -ne 0 ]; then
    ${HA_LOG} "NFS: $fs not mounted";
    ha_write_status_for_resource ${resource} ${HA_CMD_FAILED};
#   exit_script $HA_CMD_FAILED;
fi
```

   The result of this change is that the status of the commands will be written to the log, but the script will not exit.

Creating a New Resource Type from Scratch

Caution: Only `root` can make changes to the database. However, any user can use the GUI to view database information; therefore, you should not include any sensitive information in the cluster database. Users should keep this mind when deciding the list of resource attributes for the resource type.

If none of the existing resource types are similar to the type you want to create, you can create a resource type from scratch using the following methods:

- "Using the FailSafe Manager GUI ", page 67

- "Using `cmgr` Interactively ", page 72
- "Using `cmgr` With a Script", page 77

Using the FailSafe Manager 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 Version 2 Administrator's Guide* and Appendix B, "Starting the FailSafe Manager", page 101.

Define a New Resource Type

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

Tasks
> 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`.

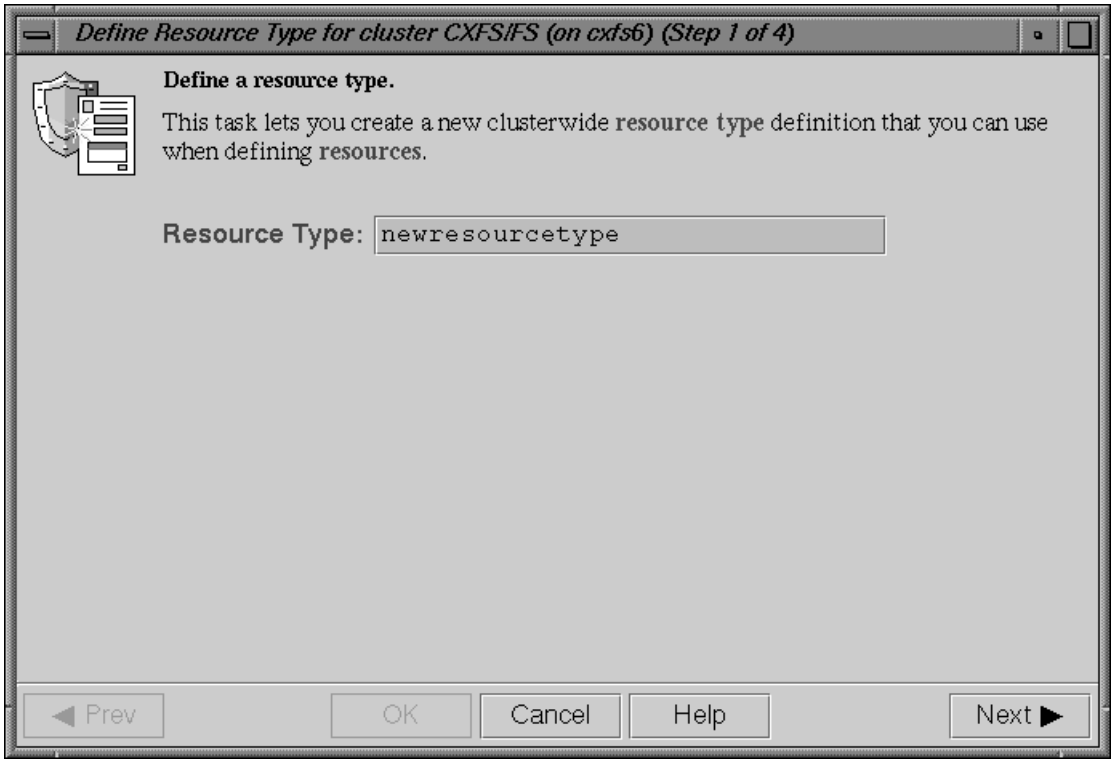

Figure 4-1 Specify the Name of the New Resource Type

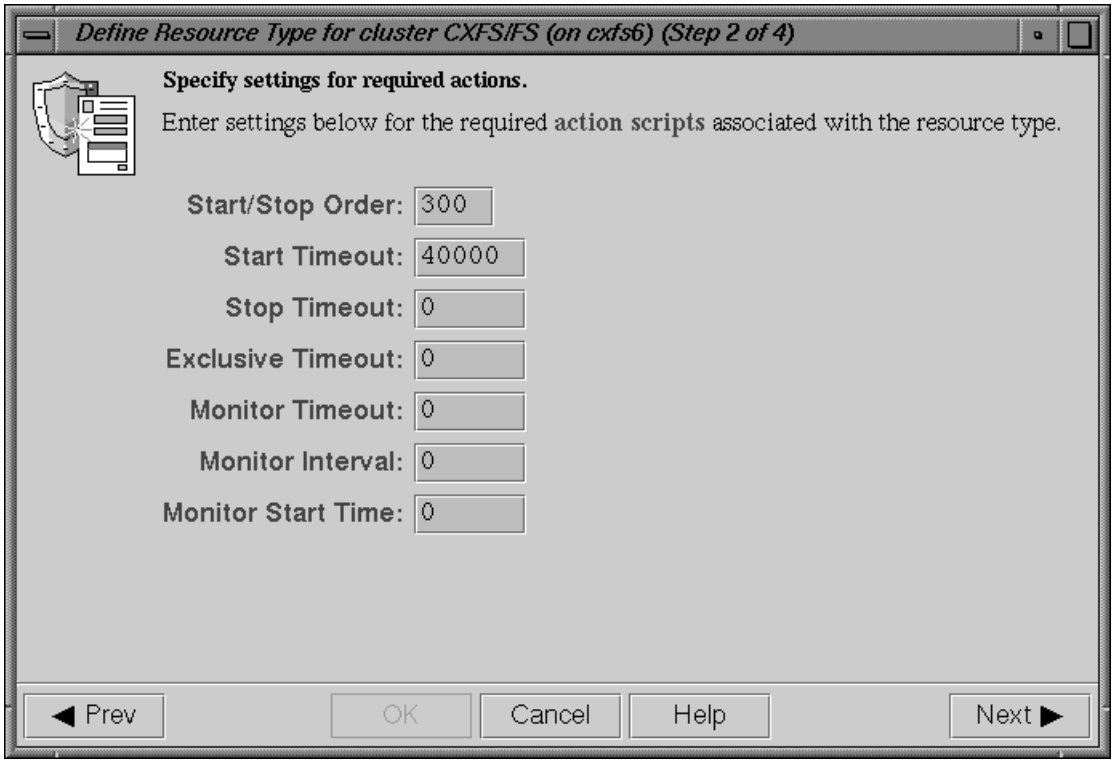

Figure 4-2 Specify Settings for Required Actions

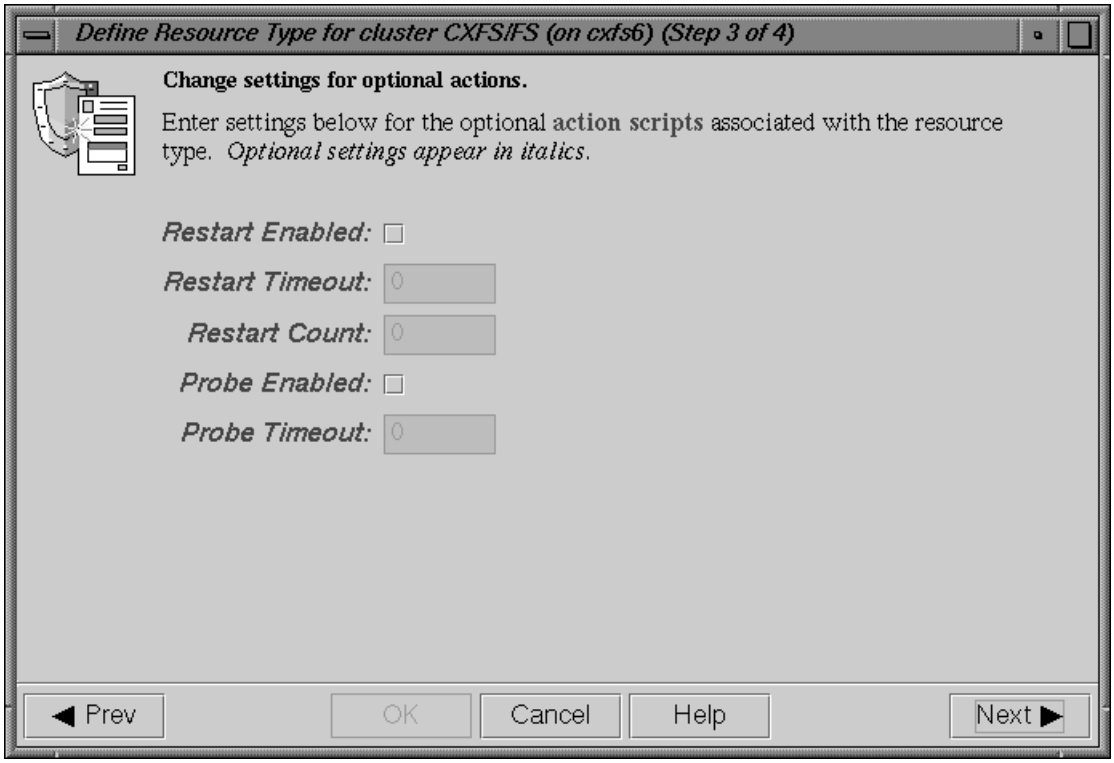

Figure 4-3 Change Settings for Optional Actions

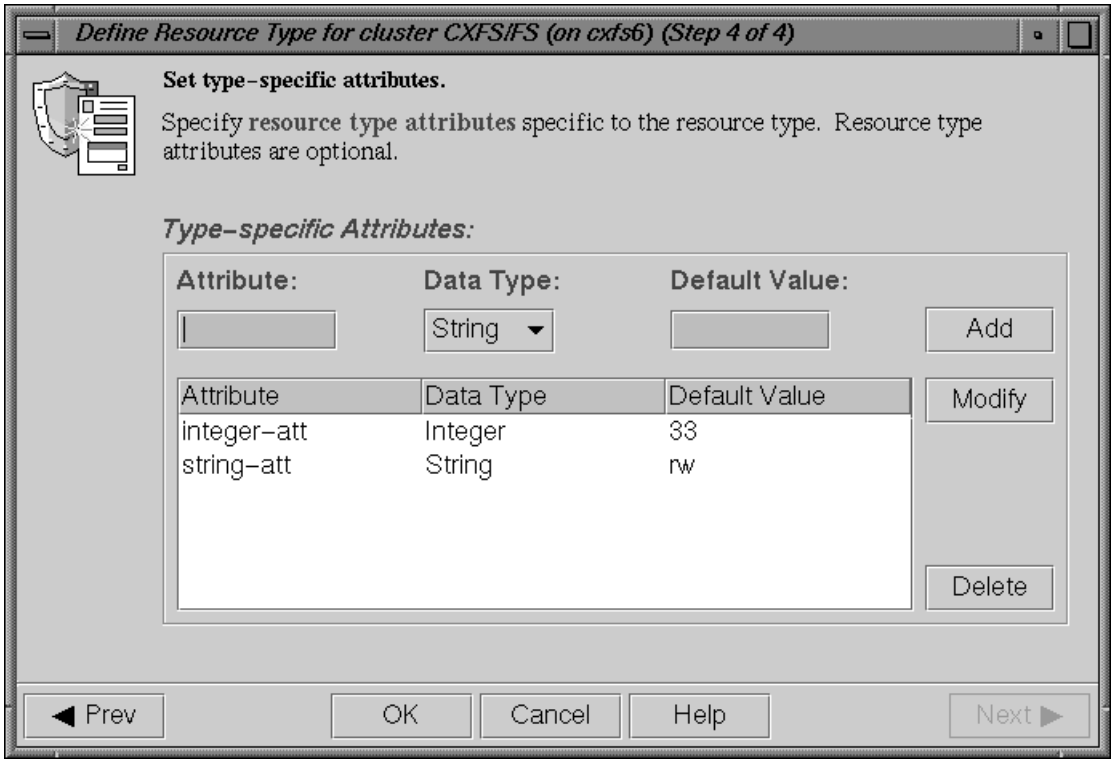

Figure 4-4 Set Type-specific Attributes

Define Dependencies

To define the dependencies for a given type, select the following menu:

Tasks
> Resource Types
> Add/Remove Dependencies for a Resource Type

Figure 4-5 shows an example of adding a dependency (`filesystem`) to the `newresourcetype` resource type.

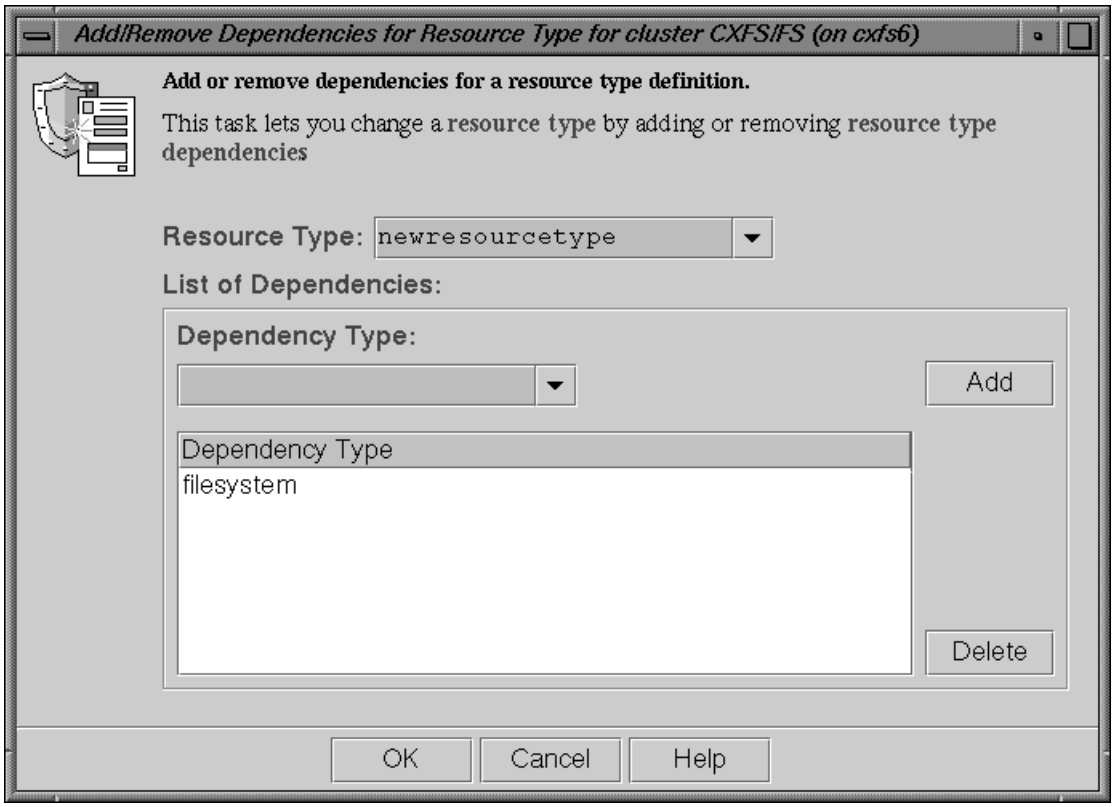

Figure 4-5 Add Dependencies

Using `cmgr` Interactively

The following steps show the use of `cmgr` (which is the same command as `cluster_mgr`(1m)) 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 `cmgr` command. You can use the `-p` option to prompt you for information (the command name can be abbreviated to `cmgr`):

```
# /usr/cluster/bin/cmgr -p
Welcome to SGI Cluster Manager Command-Line Interface

cmgr>
```

3. Use the `set` subcommand to specify the default cluster used for `cmgr` 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

        0) Modify Action Script.
        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 timeout (in milliseconds) ? 40000

        0) Modify Action Script.
        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:
        start

Action name stop
Executable timemout?  (in milliseconds) 40000

        0) Modify Action Script.
        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

        0) Modify Action Script.
        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: integer-att

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

        0) Modify Action Script.
        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

        0) Modify Action Script.
        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

        0) Modify Action Script.
        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

template
MAC_address
newresourcetype
IP_address
filesystem
volume

cmgr> exit
#
```

Using cmgr With a Script

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

`cmgr -f` *scriptname*

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

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

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

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

Note: If you include `-i` when using a `cmgr` 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 `cmgr` line, a comment line (starting with #), or a blank line. 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:

```
# 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
done

add action stop
set exec_time to 40000
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 `cmgr -f` command line with this script, you will see the following output:

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

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

```
# /usr/cluster/bin/cmgr -c "show resource_types in cluster TEST"

template
MAC_address
newresourcetype
IP_address
filesystem
volume
```

Server-side Properties File

Each resource type can have an optional properties file containing a formatted label for each plug-in attribute and strings of help text that will be displayed in the GUI. The file has the following name:

`/var/cluster/ha/resource_types/`*resource_type*`/`*resource_type*

For example, the properties file for the `IP_Address` resource type would be as follows:

```
/var/cluster/ha/resource_types/IP_Address/IP_Address
```

The contents of this file is not propagated by the cluster database; therefore, it should be installed on each node in the cluster along with the resource type's scripts. (If the

properties file is not installed on a given node and that node is used as the GUI server, the help text will not be displayed.)

Property Formats

In each resource type's properties file, you can have the following property:

resource_type`.properFormat =` *introductory text*

For each type-specific attribute, you can have the following properties:

- Label that will be displayed in the GUI:

  resource_type`.`*Attribute*`.label =` *GUI_label*

- Help (glossary) text that will be linked to from each attribute's label:

  resource_type`.`*Attribute*`.glossary =` *glossary_key*

- Information describing what the resource type is for and how it should be configured:

  glossary_key `=` *help text*

Example Properties File

Following is an example properties file for the `IP_Address` resource type.

```
# IP_address

IP_address.properFormat = \
An IP address resource that belongs to a resource group can be \
used by clients to access the highly available resource group. \
As with any other type of resource, an IP address resource will \
be moved from one node to another when \
FailSafe detects a failure. \
The resource name for an IP address must follow standard dot \
notation.  It should not be configured on any network interface. \
IP addresses that require name resolution are not valid IP_address \
resource names. \
For example, "192.0.2.22" could be the name of an IP_address \
resource.
```

```
IP_address.NetworkMask.label = \
        Network Mask
IP_address.NetworkMask.glossary = \
        glossary.IP_address.NetworkMask
glossary.IP_address.NetworkMask: \
        <B>IP address network mask</B><P>\
        The network mask of the IP address \
        (for example, "0xffffff00"). \
        See the <B>ifconfig(1M)</B> reference page for more \
        information.

IP_address.interfaces.label = \
        Interfaces
IP_address.interfaces.glossary = \
        glossary.IP_address.interfaces
glossary.IP_address.interfaces: \
        <B>IP address interfaces</B><P>\
        A comma-separated list of interfaces on which the \
        IP address can be configured \
        (for example, "ec0,et0,ef0" or "hip0" or "lb0"). \
        See the <B>ifconfig(1M)</B> reference page for more \
        information.

IP_address.BroadcastAddress.label = \
        Broadcast Address
IP_address.BroadcastAddress.glossary = \
        glossary.IP_address.BroadcastAddress
glossary.IP_address.BroadcastAddress: \
        <B>IP address broadcast address</B><P>\
        The broadcast address for the IP address \
        (for example, "192.0.2.255"). \
        See the <B>ifconfig(1M)</B> reference page for more \
        information.
```

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 `cmgr` or the GUI.

   For example, using `cmgr`:

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

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

   For example:

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

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
  ```

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, add 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:
  - Enter the following command on that node:

    ```
    ps -ef | grep application
    ```

    where *application* is the name (or a portion of the name) of the executable for the application.
  - Use appropriate commands provided by the application. For example, the FailSafe INFORMIX option uses the INFORMIX command `onstat`.
- To show the status of a resource, use the following `cmgr` command:

```
show status of resource resourcename of resource_type typename [in cluster clustername]
```

For example:

```
cmgr> show status of resource /hafs1/subdir of resource_type NFS in cluster nfs-cluster

State: Online
Error: None
Owner: hans2
Flags: Resource is monitored locally
```

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

  `show status of node` *nodename*

  For example:

  ```
  cmgr> show status of node hans2

  FailSafe status of node is UP.


  Machine (hans2) is not configured for CXFS.
  ```

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

`show status of resource_group` *RG_name* `in cluster` *clustername*

For example:

```
cmgr> show status of resource_group nfs-group1 in cluster nfs-cluster

State: Online
Error: No error
Owner: hans2
```

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 error on the primary node. The nature of the 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.

  `HA_LOG` logs messages at log level 1 and `HA_DBGLOG` uses log level 11.

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.

4. Change the log level from `HA_NORMLVL` to `HA_DBGLVL` to allow messages written with `HA_DBGLOG` to be printed by adding the following line after the `set_global_variables` statement in your script:

   ```
   HA_CURRENT_LOGLEVEL=$HA_DBGLVL
   ```

The output file will contain one of the following return values for the `start`, `stop`, `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`. The syntax for the input and output files would be as follows:

- Input file: <*resourcename*>
- Output file: <*resourcename*> <*status*>

The following example shows:

- The exit status of the action script is `1`
- 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 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`(1)

- 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 From 1.2 to 2.1.*x*

This chapter provides guidelines for migrating your IRIS FailSafe 1.2 resources and monitor script information to IRIS FailSafe 2.1.*x* action scripts. It assumes you are already familiar with the migration information provided in the *IRIS FailSafe Version 2 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 a Script Executed at the Same Time", page 17.

The software for 2.1.*x* and 1.2 can coexist in the same node. However, 2.1.*x* and 1.2 cannot run at the same time.

There is no configuration checksum verification in scripts.

Resource Types

In 2.1.*x*, the `ha.conf` configuration file has been replaced by the cluster database. The cluster database is automatically copied to all nodes in the pool. See the *IRIS FailSafe Version 2 Administrator's Guide* for information about configuring a 2.1*x* system.

If you require new resource types, you will create them using either the FailSafe Cluster Manager GUI (graphical user interface) or the `cmgr`(1M) 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 2.1.*x* resource types:

- `start-monitor-time`
- `lmon-probe-time` (equivalent in 2.1 to the `monitor` script's interval parameter)
- `lmon-timeout`

Note: All 2.1.*x* 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 2.1.*x*.

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 2.1*x,* 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 2.1*x*, 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 2.1*x*, 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 Appendix C, "Using the Script Library", page 103.

Parameter Parsing

In 2.1*x*, 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 1.2 and 2.1.*x* Scripts

IRIS FailSafe 1.2	IRIS FailSafe 2.1.*x*
`giveaway, giveback`	`stop`
`takeover, takeback`	`start`
`check`	`monitor`
(no equivalent)	`exclusive`, `restart`

In 2.1.*x*, 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`, `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 2.1.*x* equivalents:

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

Note: There are no 1.2 equivalents for the 2.1.*x* `exclusive` and `restart` scripts.

In the following examples, only the relevant portions of the scripts are shown. Areas in common between 1.2 and 2.1.*x* are in bold.

1.2 giveback / 2.1.*x* 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 2.1.*x*, 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 / 2.1.*x* 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 2.1.*x*, 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/ 2.1.*x* 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 2.1.*x*, 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 2.1.*x*, 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 FailSafe brings the group online and will be stopped when 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 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:

- Enter the following command line:

  ```
  # /usr/sbin/fstask
  ```

- Choose `FailSafe Manager` from the FailSafe toolchest.

  You must 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.

- 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 102, shows an example of the FailSafe Manager. For more information, see *IRIS FailSafe Version 2 Administrator's Guide*.

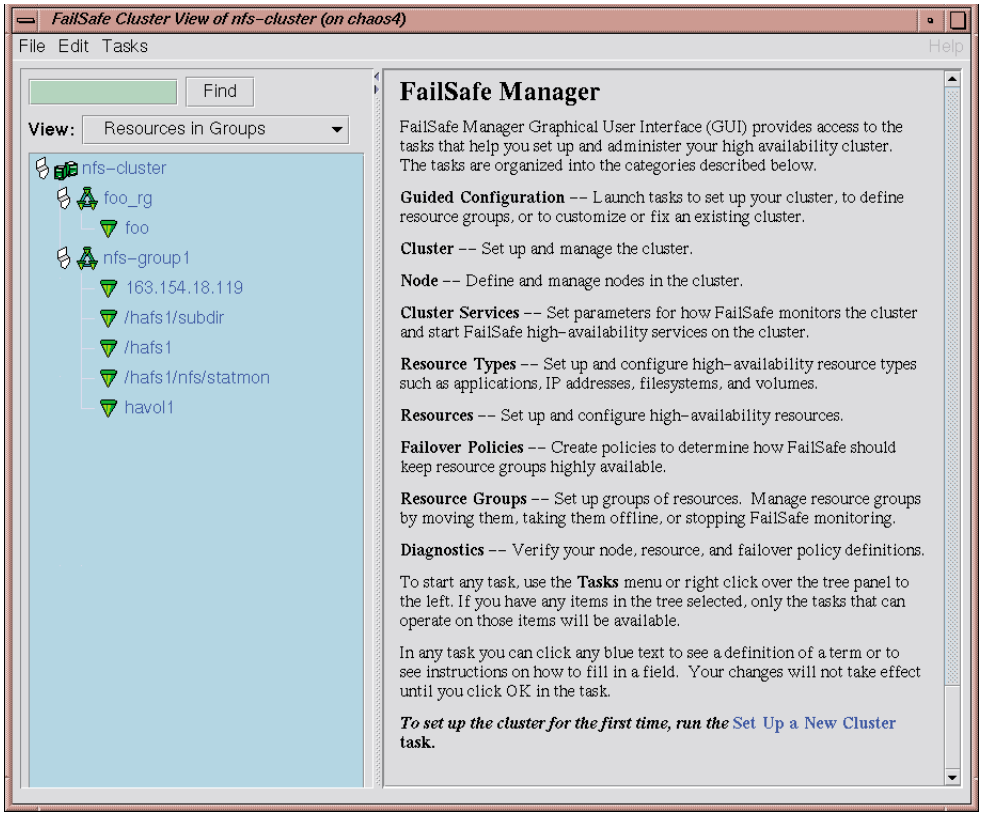

Figure B-1 FailSafe Manager

Using the Script Library

The purpose of the script library (`scriptlib`) is to simplify the IRIS FailSafe application interface so that users can use scripts and need not be aware of input and output file format. However, file format is described in "File Formats".

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

File Formats

There are three file formats:

- *Input file*, which contains the list of resources that must be acted on by the executable; each resource must be specified on a separate line in the file. The FailSafe application interface can also pass action flags for each resource in the input file.

The format of a line in the input file is as follows:

resource_name action_flags

The *resource_name* and *action_flags* fields are separated by whitespace.

- (Optional) *Output file*, in which the executable writes the return the status of each resource on a separate line, using the following format:

  resource_name resource_status

  There are corresponding lines for each line in the input file. The *resource_name* and *resource_status* fields are separated by whitespace. The resource status may be one of the following:

  - `HA_SUCCESS`
  - `HA_RUNNING`
  - `HA_NOT_RUNNING`
  - `HA_INVAL_ARGS`
  - `HA_CMD_FAILED`
  - `HA_NOTSUPPORTED`
  - `HA_NOCFGINFO` (no configuration information)

  If information about a resource is not present in the output file, SRMD assumes that the action on the resource has timed out. A nonzero value for the *resource_status* field is considered an error.

  If the executable requires more information to perform the action on the resource, the information must be stored in the cluster database (CDB) in the local machine. The executables can use cluster database commands to extract information about the resource.

- *Input parameter file*, which contains the cluster name in the following format:

  `ClusterName` *clustername*

Set Global Definitions

The `ha_set_global_defs()` function sets the global definitions for the environment variables shown in the following subsections.

The `HA_INFILE` and `HA_OUTFILE` variables set the input and output files for a script. These variables do not have global definitions, and are not set by the `ha_set_global_defs()` function.

Global Variable

`HA_HOSTNAME`

The output of the `uname` command with the `-n` option, which is the host name or node name. The node name is the name by which the system is known to communications networks.

Default: `'uname -n'`

Command Location Variables

`HA_CMDSPATH`

Path to user commands.

Default: `/usr/cluster/bin`

`HA_PRIVCMDSPATH`

Path to privileged commands (those that can only be run by `root`).

Default: `/usr/sysadm/privbin`

`HA_LOGCMD`

Command used to log information.

Default: `ha_cilog`

HA_RESOURCEQUERYCMD

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.

Default: `resourceQuery`

HA_SCRIPTTMPDIR

Location of the script temporary directory.

Default: `/tmp`

Database Location Variables

HA_CDB

Location of the cluster database.

Default: `/var/cluster/cdb/cdb.db`

Script Log Level Variables

HA_NORMLVL

Normal level of script logs.

Default: `0`

HA_DBGLVL

Debug level of script logs.

Default: `10`

Script Log Variables

HA_SCRIPTGROUP

Log for the script group.

Default: `script`

HA_SCRIPTSUBSYS

Log for the script subsystem.

Default: `script`

Script Logging Command Variables

HA_LOGQUERY_OUTPUT

Determine the current logging level for scripts.

Default:

```
'${HA_PRIVCMDSPATH}/loggroupQuery _NUM_LOG_GROUPS=1 \
_LOG_GROUP_0=ha_script'
```

HA_DBGLOG

Command used to log debug messages from the scripts.

Default: `ha_dbglog`

HA_CURRENT_LOGLEVEL

Display the current log level. The default will be 0 (no script logging) if the `loggroupQuery` command fails or does not find configuration information.

Default: `'echo ${HA_LOGQUERY_OUTPUT} | /usr/bin/awk '{print $2}''`

HA_LOG

Command used to log the scripts.

Default: `ha_log`

Script Error Value Variables

HA_SUCCESS

Successful execution of the script. This variable is used by the `start`, `stop`, `restart`, and `monitor` scripts.

Default: `0`

HA_NOT_RUNNING

The script is not running. This variable is used by `exclusive` scripts.

Default: `0`

HA_INVAL_ARGS

An invalid argument was entered. This is used by all scripts.

Default: `1`

HA_CMD_FAILED

A command called by the script has failed. This variable is used by the `start`, `stop`, `restart`, and `monitor`, scripts.

Default: `2`

HA_RUNNING

The script is running. This variable is used by `exclusive` scripts.

Default: `2`

HA_NOTSUPPORTED

The specific action is not supported for this resource type. This is used by all scripts.

Default: 3

HA_NOCFGINFO

No configuration information was found. This is used by all scripts.

Default: 4

Check Arguments

An action script can have an input file (`$1 HA_INFILE`), an output file (`$2 HA_OUTFILE`), and a parameter file (`$3 HA_PARAMFILE`). The parameter file is optional.

The `ha_check_args()` function checks the arguments specified for a script and sets the `$HA_INFILE` and `$HA_OUTFILE` variables accordingly.

If a parameter file exists, the `ha_check_args()` function reads the list of parameters from the file and sets the `$HA_CLUSTERNAME` variable.

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`, which is set in the action script. `$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:

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

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

    return 1;
}
```

Get the Value for Multiple Fields

The `ha_get_multi_fields()` function obtains the field values from a string, where `$1` is the string and `$2` is the field name. The string format is a series of name-value field pairs, where a name field is followed by the value of the name, separated by whitespace.

This function is typically used to extract dependency information. There may be multiple fields with the same name, so the string returned in `HA_FIELD_VALUE` may contain multiple values separated by white space. This appears as follows:

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

    for ha_i in $HA_STR
    do
        if [ $ha_field -eq 1 ]; then
            ha_field=0;
            if [ $ha_i = $HA_FIELD_NAME ]; then
                ha_found=1;
            fi
        else
            ha_field=1;
            if [ $ha_found -eq 1 ]; then
                if [ -z "$HA_FIELD_VALUE" ]; then
                    HA_FIELD_VALUE=$ha_i;
                else
                    HA_FIELD_VALUE="$HA_FIELD_VALUE $ha_i";
                fi;
                ha_found=0;
            fi
        fi
    done

    if [ -z "$HA_FIELD_VALUE" ]; then
        return 1;
    else
        return 0;
    fi
}
```

Get Resource Information

The `ha_get_info()` and `ha_get_info_debug()`functions read resource information. `$1` is the resource type, `$2` is the resource name, and `$3` is an optional parameter of any value that specifies a request for resource dependency information. Resource information is stored in the `HA_STRING` variable. 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 $3
        return;
    fi

    if [ -n "$3" ]; then
        ha_doall="_ALL=true"
    else
        ha_doall=""
    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 $HA_RETRY_CMD_MAX ];
    do
        if [ -n "${HA_CLUSTERNAME}" ]; then
            HA_STRING='${HA_PRIVCMDSPATH}/${HA_RESOURCEQUERYCMD} \
                        _CDB_DB=$HA_CDB _RESOURCE=$2 _RESOURCE_TYPE=$1  \
                        $ha_doall _NO_LOGGING=true _CLUSTER=${HA_CLUSTERNAME}'
        else
            HA_STRING='${HA_PRIVCMDSPATH}/${HA_RESOURCEQUERYCMD} \
                        _CDB_DB=$HA_CDB _RESOURCE=$2 _RESOURCE_TYPE=$1  \
                        $ha_doall _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

    if [ -n "$ha_doall" ]; then
        echo $HA_STRING  \
                | grep "No resource dependencies" > /dev/null 2>&1
        if [ $? -eq 0 ]; then
            HA_STRING=
        else
            HA_STRING=‘echo $HA_STRING  | /bin/sed -e "s/^.*Resource dependencies //"‘
        fi
    fi

    return ${ha_exit_code};
}
```

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

```
ha_get_info_debug()
{
    if [ -n "$3" ]; then
        ha_doall="_ALL=true"
    else
        ha_doall=""
    fi

    if [ -n "${HA_CLUSTERNAME}" ]; then
        HA_STRING=‘${HA_PRIVCMDSPATH}/${HA_RESOURCEQUERYCMD} \
                    _CDB_DB=$HA_CDB _RESOURCE=$2 _RESOURCE_TYPE=$1      \
                    $ha_doall _CLUSTER=${HA_CLUSTERNAME}‘
    else
        HA_STRING=‘${HA_PRIVCMDSPATH}/${HA_RESOURCEQUERYCMD} \
                    _CDB_DB=$HA_CDB _RESOURCE=$2 _RESOURCE_TYPE=$1 $ha_doall‘
    fi
    ha_exit_code=$?

    if [ $? -ne 0 ]; then
        ${HA_LOG} "${HA_RESOURCEQUERYCMD}: resource name $2 resource type $1"
```

```
        ${HA_LOG} "Failed with error: ${HA_STRING}";
    fi

    if [ -n "$ha_doall" ]; then
        echo $HA_STRING  \
                | grep "No resource dependencies" > /dev/null 2>&1
        if [ $? -eq 0 ]; then
            HA_STRING=
        else
            HA_STRING=‘echo $HA_STRING  | /bin/sed -e "s/^.*Resource dependencies //"‘
        fi
    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: `exclusive`, `start`, `stop`, `monitor`, and `restart`.

cluster

The set of nodes in the pool that have been defined as a cluster. A cluster is identified by a simple name; this name must be unique within the pool. All nodes in the cluster are also in the pool. However, all nodes in the pool are not necessarily in the cluster; that is, the cluster may consist of a subset of the nodes in the pool. There is only one cluster per pool.

cluster administrator

The person responsible for managing and maintaining a cluster.

cluster database

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

cluster node

See *node*.

cluster process group

A group of application instances in a distributed application that cooperate to provide a service.

For example, distributed lock manager instances in each node would form a process group. By forming a process group, they can obtain membership and reliable, ordered, atomic communication services. There is no relationship between a UNIX process group and a cluster process group.

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 that 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.

control network

The network that connects nodes through their network interfaces (typically Ethernet) such that FailSafe can maintain a cluster's high availability by sending heartbeat messages and control messages through the network to the attached nodes. 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.

database

See *cluster database*.

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

failover policy

The method used by 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 FailSafe process. The process applies the failover attributes and then selects the first node in the returned failover domain that is also in the current FailSafe membership.

FailSafe membership

The list of FailSafe nodes in a cluster on which FailSafe can make resource groups online. It differs from `fs2d` database membership and CXFS membership. For more information about CXFS, see *CXFS Version 2 Software Installation and Administration Guide*.

FailSafe database

See *cluster database*.

`fs2d` database membership

The group of nodes in the pool that are accessible to `fs2d` and therefore can receive cluster database updates; this may be a subset of the nodes defined in the pool. (Also known as *user-space* membership).

heartbeat messages

Messages that cluster software sends between the nodes to indicate that 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 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 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. (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 FailSafe processes that use the same log configuration. A log group usually corresponds to one daemon, such as `gcd`.

log level

A number controlling the number of log messages that 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

A single IRIX kernel image. Usually, a node is an individual computer. The term *node* does not have the same meaning as a node in an Origin system.

node ID

A 16-bit positive integer that uniquely defines a cluster node. During node definition, 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

See *FailSafe membership*.

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 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, FailSafe stops the group (if needed) and stops monitoring the group. An offline resource group can be running on a node, yet not under FailSafe control. If the cluster administrator specifies the detach-only option while taking the group offline, then 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 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, 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 FailSafe will not start the group but will begin monitoring the group.

owner host

A system that can control a node remotely, such as power-cycling the node. At run time, the owner host must be defined as a node in the 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 node with the system controller port, so the node can be controlled remotely by the owner host.

plug-ins

Software used to make applications highly available. There are provided, optional, and customized plug-ins.

pool

The entire set of nodes that are coupled to each other by networks and are defined as nodes in FailSafe. The nodes are usually close together and should always serve a common purpose. A replicated cluster database is stored on each node in the pool.

All nodes that can be added to a cluster are part of the pool, but not all nodes in the pool must be part of the cluster. There is only one pool. (Other pools may exist, but each is disjoint from the other. They share no node or cluster definitions.)

plug-in

The set of software required to make an application highly available, including a resource type and action scripts. There are plug-ins provided with the base FailSafe release, optional plug-ins available for purchase from SGI, and customized plug-ins you can write using the instructions in this guide.

port password

The system controller password for privileged commands, 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, 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, FailSafe uses heuristics to try to estimate whether the machine has been powered down. If the heuristic algorithm returns with success, FailSafe assumes the remote machine has been reset successfully. When powerfail mode is turned off, the heuristics are not used and 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.

properties file

An optional file that contains a formatted label for each plug-in attribute and strings of help text that will be displayed in the GUI. There can be a properties file for each resource type.

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.

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 clusterwide resource type definition with the same name; this allows an individual node to override global settings from a clusterwide 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 `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 FailSafe brings the group online and will be stopped when 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 FailSafe brings a

resource group online, all volume resources in the group will be started before all `filesystem` resources in the group.

system controller port

A port located on a node that provides a way to power-cycle the node remotely. Enabling or disabling a system controller port in the cluster database tells 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 FailSafe to use in the process of computing FailSafe cluster 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, 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

E

F

G

H

V

W

X