Linux FailSafe™ Programmer's Guide

Date: 10 October 2000

Written by Joshua Rodman of SuSE, Inc. and Lori Johnson of SGI

Illustrated by Dany Galgani and Chris Wengelski

Production by Diane Ciardelli and Adrian Daley

Engineering contributions by Rusty Ballinger, Franck Chastagnol, Jeff Hanson, Vidula Iyer, Herbert Lewis, Michael Nishimoto, Hugh Shannon Jr., Bill Sparks, Paddy Sreenivasan, Dan Stekloff, Rebecca Underwood, Mayank Vasa, and Manish Verma

007-4295-001	August 2000	Original publication
007-4295-002	September 2000	Corrections

Contents

About This Guide

This guide explains how to write the set of scripts that are required to turn an application into a highly available service in conjunction with Linux FailSafe software. It also tells you how to create a new resource type.

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

Audience

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

Related Documentation

Besides this guide, other documentation for Linux FailSafe includes the following:

- *Linux FailSafe Administrator's Guide*

System man pages for referenced commands are as follows:

- `cbeutil`
- `cdbBackup`
- `cdbRestore`
- `cdbutil`
- `cluster_mgr`
- `crsd`
- `failsafe`
- `cdbd`
- `ha_cilog`
- `ha_cmsd`
- `ha_exec2`
- `ha_fsd`
- `ha_gcd`
- `ha_ifd`
- `ha_ifdadmin`

- `ha_macconfig2`
- `ha_srmd`
- `ha_statd2`
- `haStatus`

Conventions Used in This Guide

These type conventions and symbols are used in this guide:

`command`

Function names, literal command-line arguments (options/flags)

`filename`

Name of a file or directory

`command -o option`

Commands and text that you are to type literally in response to shell and command prompts

term

New terms

Book Title

Manual or book title

variable

Command-line arguments, filenames, and variables to be supplied by the user in examples, code, and syntax statements

`literal text`

Code examples, error messages, prompts, and screen text

`#`

System shell prompt for the superuser (`root`)

1 Introduction to Writing Application Scripts

Linux FailSafe provides several highly available services for a two–node cluster. These services are monitored by the Linux FailSafe software. You can create additional services that are highly available by using the instructions in this guide.

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

- Section 1.1, *Concepts*
- Section 1.2, *Highly Available Services Included with Linux FailSafe*
- Section 1.3, *Plug-Ins*
- Section 1.5, *Overview of the Programming Steps*
- Section 1.4, *Characteristics that Permit an Application to be Highly Available*
- Section 1.5, *Overview of the Programming Steps*

For an overview of the software layers, communication paths, and cluster configuration database, see the *Linux FailSafe Administrator's Guide*.

1.1 Concepts

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

1.1.1 Cluster Node (or Node)

A **cluster node** is a single Linux execution environment. In other words, a single physical or virtual machine. In current Linux environments this will always be an individual computer. The term **node** is used to indicate this meaning in this guide for brevity, as opposed to any meaning such as a network node.

1.1.2 Pool

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

1.1.3 Cluster

A **cluster** is a collection of one or more nodes coupled to each other by networks or other similar interconnections. A cluster belongs to one pool and only one pool. A cluster is identified by a simple name; this name must be unique within the pool. A particular node may be a member of only one cluster. All nodes in a cluster are also in the pool; however, all nodes in the pool are not necessarily in the cluster.

1.1.4 Node Membership

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

1.1.5 Process Membership

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

1.1.6 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 only be allocated to one node at any given time.

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

1.1.7 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 should 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 definition for a specific node overrides a clusterwide resource type definition with the same name; this allows an individual node to override global settings from a clusterwide resource type definition.

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

The Linux FailSafe software includes some 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.

1.1.8 Resource Name

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

1.1.9 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, *Example Resource Group* 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.

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

1.1.10 Resource Dependency List

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

1.1.11 Resource Type Dependency List

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

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

1.1.12 Failover

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

1.1.13 Failover Policy

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

- Failover domain
- Failover attributes
- Failover script

Linux 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. Linux FailSafe includes predefined failover policies, but youcan define your own failover algorithms as well.

1.1.14 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; Linux FailSafe uses the run-time failover domain along with failover attributes and the node membership to determine the node on which a resource group should reside. Linux 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, Linux FailSafe allocates a given resource group to the first node listed in the run-time failover domain that is also in the node membership; the point at which this allocation takes place is affected by the failover attributes.

1.1.15 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.

1.1.16 Failover Scripts

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

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

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

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

1.1.17 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 in this guide.

1.2 Highly Available Services Included with Linux FailSafe

The base release includes the software required to make IP addresses (the `IP_address` resource type) highly available.

1.3 Plug-Ins

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

The following plug-ins are available for Linux FailSafe:

- Logical volumes (the `volume` resource type) such as provided by `LVM`
- Filesystems such as `reiserfs` and `ext2fs` (the `filesystem` resource type)
- MAC addresses (the `MAC_address` resource type)
- Linux FailSafe Samba
- Linux FailSafe NFS

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

If you want to create new highly available services, or change the functionality of the provided failover scripts and action scripts by writing new scripts, you will use the instructions in this guide. However, not all resources can be made highly available; see Section 1.4, *Characteristics that Permit an Application to be Highly Available*.

1.4 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 the start procedure.

- 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, required resources 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 Linux FailSafe and its optional products (see Section 1.2, *Highly Available Services Included with Linux FailSafe*), you must make these resources highly available, using the information in this guide.

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

1.5 Overview of the Programming Steps

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

To make an application highly available, follow these steps:

1. Understand the application and determine:

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

`/usr/lib/failsafe/resource_types/template` as a guide. See Chapter 2, *Writing the Action Scripts and Adding Monitoring Agents*.

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*.
4. Determine whether an existing resource type can be reused. If none applies, create a new resource type or modify an existing resource type. See Chapter 4, *Defining a New Resource Type*.
5. Configure the following in the cluster configuration database (for more information, see the *Linux FailSafe Administrator's Guide*):
   - Resource group
   - Resource type
   - Failover policy
6. Test the action scripts and failover script. See Chapter 5, *Testing Scripts*, and Section 5.2, *Debugging Notes*.

Do not modify the scripts included with the Linux FailSafe product. New or customized scripts must have different names from the files included with the release.

---

2 Writing the Action Scripts and Adding Monitoring Agents

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

- Section 2.1, *Set of Action Scripts*
- Section 2.2, *Understanding the Execution of Action Scripts*
- Section 2.3, *Preparation*
- Section 2.4, *Script Format*
- Section 2.5, *Steps in Writing a Script*
- Section 2.6, *Examples of Action Scripts*
- Section 2.7, *Monitoring Agents*

2.1 Set of Action Scripts

CAUTION

Multiple instances of scripts may be executed at the same time. For more information, see Section 2.2, *Understanding the Execution of Action Scripts*.

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

- `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.

The `start` and `stop` scripts must be **idempotent**; that is, an action requested multiple times successively should continue to return success, and should have no side-effects. 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.

2.2 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:

- Section 2.2.1, *Multiple Instances of Script Executed at the Same Time*
- Section 2.2.2, *Differences between the `exclusive` and `monitor` Scripts*
- Section 2.2.3, *Successful Execution of Action Scripts*
- Section 2.2.4, *Failure of Action Scripts*
- Section 2.2.5, *Implementing Timeouts and Retrying a Command*
- Section 2.2.6, *Sending UNIX Signals*

2.2.1 Multiple Instances of Script Executed at the Same Time

Multiple instances of the same script may be executed at the same time. To avoid problems this may cause, 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, consider a script which modifies a configuration file to start a new application instance. Multiple instances of the script modifying the file simultaneously could cause file corruption and data loss. The start script for the application should use `ha_execute_lock` when executing the modification script to ensure correct configuration file modification.

Assuming the script is named `modify_configuration_file`, the start script would contain a statement similar to the following:

```
${HA_CMDSPATH}/ha_execute_lock 30
    ${HA_SCRIPTTMPDIR}/lock.volume_assemble \"modify_configuration_file\"
```

The `ha_execute_lock` command takes 3 arguments:

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

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

2.2.2 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 2–1, *Differences Between the `monitor` and `exclusive` Action Scripts* summarizes the differences between the scripts.

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

`exclusive`	`monitor`
Executed in all nodes in the cluster.	Executed only on the node where the resource group (which contains the resource) is online.
Executed before the resource is started in the cluster.	Executed when the resource is online in the cluster. (The `monitor` script could degrade the services provided by the HA server. Therefore, the check performed by the `monitor` script should be lightweight and less time consuming than the check performed by the `exclusive` script))
Executed only once before the resource group is made online in the cluster.	Executed periodically.
Failure will result in resource group not becoming online in the cluster.	Failure will cause a resource group failover to another node or a restart of the resource in the local node. An error will cause false resource group failovers in the cluster.

2.2.3 Successful Execution of Action Scripts

Table 2–2, *Successful Action Script Results* 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 Cluster Manager graphical user interface (GUI) or the `cluster_mgr` command.

Table 2–2 Successful Action Script Results

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

2.2.4 Failure of Action Scripts

Table 2–3, *Failure of an Action Script* shows the state of the resource group and the error state when an action script fails.

Table 2–3 Failure of an Action Script

Failing Action Script	Resource Group State	Error State
`exclusive`	`online`	`exclusivity`
`monitor`	`online`	`monitoring failure`
`restart`	`online`	`monitoring failure`
`start`	`online`	`srmd executable error`
`stop`	`online`	`srmd executable error`

2.2.5 Implementing Timeouts and Retrying a Command

You can use the `ha_exec2` 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/lib/failsafe/bin/ha_exec2 timeout_in_seconds number_of_retries command_to_be_executed
```

For example:

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

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

For more information, see the `ha_exec2` man page.

2.2.6 Sending UNIX Signals

You can use the `ha_exec2` 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 "knfsd"
```

The above command sends signal 0 (checks if the process exists) to all processes whose name or arguments match the string `knfsd`. 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 a `ps -ef | grep` command line construction, for performance and speed considerations.

For more information, see the `ha_exec2` man page.

2.3 Preparation

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

- Understand the `scriptlib` functions described in Appendix B, *Using the SRM Script Library*.
- Familiarize yourself with the script templates provided in the following directory: `/usr/lib/failsafe/resource_types/template`
- Read the man pages for the following commands:
  - `cluster_mgr`
  - `cdbd`
  - `ha_cilog`
  - `ha_cmsd`
  - `ha_exec2`
  - `ha_fsd`
  - `ha_gcd`
  - `ha_ifd`
  - `ha_ifdadmin`
  - `ha_macconfig2`
  - `ha_srmd`
  - `ha_statd2`
  - `haStatus`
- Familiarize yourself with the action scripts for other highly available services in `/usr/lib/failsafe/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 Section 2.3.1, *Is Monitoring Necessary?*). 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 configuration 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 CDB.
- Determine whether the resource type can be restarted in its local node, and whether this action makes sense.

2.3.1 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 Linux kernel ipchains filtering 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 Section 2.7, *Monitoring Agents*.

2.3.2 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 configuration database. Ensure that the monitoring level chosen does not affect system performance. For more information, see the *Linux FailSafe Administrator's Guide*.

2.3.3 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

2.3.4 How Often Should Monitoring Occur?

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

2.3.5 Examples of Testing for Monitoring Failure

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

- For a client/server applications 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` command to check for a particular message.
- The following command can be used to determine quickly whether a process exists:

  ```
  /usr/bin/killall -0 process_name
  ```

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

  The `ha_exec2` command differs from `killall` 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/lib/failsafe/bin/ha_exec2 -s 0 -t *process_name*

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

2.4 Script Format

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

```
/usr/lib/failsafe/resource_types/template
```

The template scripts have the same general format. Following is the order in which the information appears in the script:

- 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

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

The following sections show an example from the NFS `start` script. Note that the contents of these examples may not match the latest software.

2.4.1 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:

```
#!/bin/sh

# **************************************************************************
# *                                                                        *
# *                  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: 1.9 $"

# Resource type: NFS
# Start script NFS

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

2.4.2 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
}
```

2.4.3 Read Resource Information

The `get_`*xxx*`_info()` function, such as `get_nfs_info()`, reads the resource information from the cluster configuration 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 B, *Using the SRM Script Library*.

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
}
```

If you wish to get resource dependency information, you can call `ha_get_info` with a third argument of any value. The resource dependency list will be returned in the `HA_STRING` variable.

2.4.4 Exit Status

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

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

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

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. For more information see Appendix B, *Using the SRM Script Library*.

2.4.5 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
}
```

When testing the script, you can obtain debugging information by adding the shell command `set -x` to this section.

For examples of this area, see Section 2.6, *Examples of Action Scripts*.

2.4.6 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=/usr/lib/failsafe
    COMMON_LIB=${HA_DIR}/common_scripts/scriptlib

    # Execute the common library file
    . $COMMON_LIB

    ha_set_global_defs;
}
```

2.4.7 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 following is the section from the NFS start script that calls `ha_check_args`:

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

2.4.8 Read Input File

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

Following is the `ha_read_infile()` function from the common library file `scriptlib`:

```
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
}
```

2.4.9 Complete the Action

Located at the bottom of the script file are the lines which perform the actual work of the requested action using the prior sections and provided tools. The results are written as output to `$HA_OUTFILE`:

```
action_resourcetype;

exit_script $HA_SUCCESS
```

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

```
start_nfs;

exit_script $HA_SUCCESS;
```

2.5 Steps in Writing a Script

CAUTION

Multiple copies of actions scripts can execute at the same time. Therefore, all temporary file names must be unique within the storage space used. Often adding a `script.$$` to the name is sufficient. If multiple nodes share a temporary directory, you will also want to incorporate host identifier to ensure uniqueness. Another method is to 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

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/log/failsafe/script_`*`nodename`* file.

2.6 Examples of Action Scripts

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

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

2.6.1 `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="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"

        # Before we try and export the NFS resource, make sure
        # filesystem is mounted.
        HA_CMD="grep $filesystem /etc/mtab > /dev/null 2>&1";
        ha_execute_cmd "check if the filesystem $filesystem is mounted";
        if [ $? -ne 0 ]; then
            ${HA_LOG} "NFS: filesystem $filesystem not mounted";
            ha_write_status_for_resource ${resource}  ${HA_CMD_FAILED};
            exit_script ${HA_CMD_FAILED};
        fi

        # Now do the job: export the new directory
        # Note: the export_dir command will check wether this directory
```

```
            # is already exported or not.
            HA_CMD="export_dir ${resource} ${export_opts}";
            ha_execute_cmd "export $resource directories to NFS clients";
            if [ $? -ne 0 ]; then
                ${HA_LOG} "NFS: could not export resoure ${resource}"
                ha_write_status_for_resource ${resource} ${HA_CMD_FAILED};
                exit_script ${HA_CMD_FAILED};
            else
                ha_write_status_for_resource ${resource} ${HA_SUCCESS};
            fi

        done
    }
```

2.6.2 `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"


            # Unexport the directory
            HA_CMD="unexport_dir ${resource}"
            ha_execute_cmd "unexport ${resource} directory to NFS clients"
            if [ $? -ne 0 ]; then
                ${HA_LOG} "NFS: Failed to unexport resource ${resource}"
                ha_write_status_for_resource ${resource} ${HA_CMD_FAILED}
            fi

            ha_write_status_for_resource ${resource} ${HA_SUCCESS}
        done
    }
```

2.6.3 `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="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="fs_stat -r ${resource} >/dev/null 2>&1";
            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
            showmount -e | grep "${resource} " >/dev/null 2>&1
            if [ $? -ne 0 ]; then
                ${HA_LOG} "NFS: failed to find ${resource} in exported filesystem list:-"
                ${HA_LOG} "‘showmount -e‘"
                ha_write_status_for_resource ${resource} ${HA_CMD_FAILED}
                exit_script ${HA_CMD_FAILED}
            fi

            # check the NFS daemon is still alive and responding
            exec_rpcinfo;
            if [ $? -ne 0 ]; then
                    ${HA_LOG} "NFS: exec_rpcinfo failed";
                    ha_write_status_for_resource ${resource} ${HA_CMD_FAILED}
                    exit_script $HA_CMD_FAILED
            fi

            # Check the stats ?
            # To Be Done... but there is no nfsstat command
            # for the user space NFS daemon.


            ha_write_status_for_resource $resource $HA_SUCCESS;
        done
    }
```

2.6.4 `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. Linux 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

        # Check if resource is already exported by the NFS server
        showmount -e | grep "${resource} " >/dev/null 2>&1
        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

    done
}
```

2.6.5 `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:

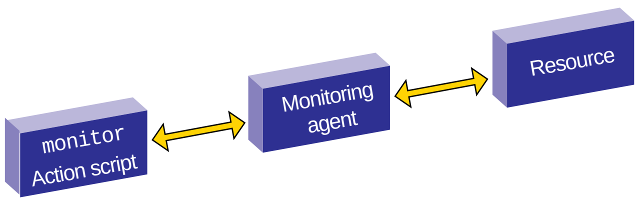
```
# Restart nfs resource
# Return HA_SUCCESS if nfs resource failed over successfully or
# return HA_CMD_FAILED if nfs resource could not be failed over locally.
# The list of nfs resources passed as input is in variable
# $HA_RES_NAMES
#
restart_nfs()
{
    ${HA_DBGLOG} "Entry: restart_nfs()";

    # 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"


            # Note: the export_dir command will check wether this directory
            # is already exported or not.
            HA_CMD="export_dir ${resource} ${export_opts}";
            ha_execute_cmd "export $resource directories to NFS clients";
            if [ $? -ne 0 ]; then
                ${HA_LOG} "NFS: could not export resoure ${resource}"
                ha_write_status_for_resource ${resource} ${HA_CMD_FAILED};
                exit_script ${HA_CMD_FAILED};
            else
                ha_write_status_for_resource ${resource} ${HA_SUCCESS};
            fi

        done
    }
```

2.7 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, *Monitoring Process* 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 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 /etc/failsafe/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 /etc/failsafe/cmon_process_groups/ip_addresses file contains information about the ha_ifd process that monitors network interfaces. It contains the following, where ACTIONS represents what cmond can perform on the agents (which will be the same for all scripts):

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

If you create a new monitoring agent, you must also create a corresponding file in the /etc/failsafe/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 processes must be located in the /usr/lib/failsafe/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 /etc/failsafe/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/lib/failsafe/bin/ha_x and /usr/lib/failsafe/bin/ha_y) to be present.

3 Creating a Failover Policy

This chapter tells you how to create a failover policy. It describes the following topics:

- Section 3.1, *Contents of a Failover Policy*
- Section 3.2, *Failover Script Interface*
- Section 3.3, *Example Failover Policies for Linux FailSafe*

3.1 Contents of a Failover Policy

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

- Failover domain
- Failover attributes
- Failover scripts

Linux 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.

3.1.1 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; 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.

For example, suppose the initial failover domain is:

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

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

3.1.2 Failover Attributes

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

Table 3–1, *Required Failover Attributes (mutually exclusive)* shows the required failover attributes. You must specify one and only one of these attributes. Note that the starting conditions for the attributes differs: for the required attributes, the starting condition is that a node joins the cluster membership when the cluster is already providing HA services; for the optional attributes, the starting condition is that HA services are started and the resource group is running in only one node in the cluster

Table 3–1 Required Failover Attributes (mutually exclusive)

Name	Description
`Auto_Failback`	Specifies that the resource group is made online based on the failover policy when a 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.

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

Table 3–2 Optional Failover Attributes (mutually exclusive)

Name	Description
`Auto_Recovery`	Specifies that the 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 `In-place_Recovery` attribute. If you specify neither of these attributes, Linux 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 the default and is mutually exclusive with the `Auto_Recovery` attribute. If you specify neither of these attributes, Linux FailSafe will use this attribute by default if you have specified the `Controlled_Failback` attribute.

3.1.3 Failover Scripts

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

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

Failover scripts are stored in the `/usr/lib/failsafe/policies` directory.

3.1.3.1 The `ordered` Failover Script

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. (Line breaks added for readability.)

```
#!/bin/sh
#
# Copyright (c) 2000 Silicon Graphics, Inc.  All Rights Reserved.
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of version 2 of the GNU General Public License as
# published by the Free Software Foundation.
#
# This program is distributed in the hope that it would be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
#
# Further, this software is distributed without any warranty that it is
# free of the rightful claim of any third person regarding infringement
# or the like.  Any license provided herein, whether implied or
# otherwise, applies only to this software file.  Patent licenses, if
# any, provided herein do not apply to combinations of this program with
# other software, or any other product whatsoever.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write the Free Software Foundation,
# Inc., 59 Temple Place - Suite 330, Boston MA 02111-1307, USA.
#
# Contact information: Silicon Graphics, Inc., 1600 Amphitheatre Pkwy,
# Mountain View, CA 94043, or:
#
# http://www.sgi.com
#
# For further information regarding this notice, see:
#
# http://oss.sgi.com/projects/GenInfo/NoticeExplan

#
# $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/lib/failsafe/bin
LOG="${DIR}/ha_cilog -g ha_script -s script"
FILE=/usr/lib/failsafe/policies/ordered

input=$1
output=$2

{
  read version
  read name
  read owner
  read attr
  read mem1 mem2 mem3 mem4 mem5 mem6 mem7 mem8
  read afd1 afd2 afd3 afd4 afd5 afd6 afd7 afd8
} < ${input}
```

```
${LOG} -l 1 "${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
```

3.1.3.2 The `round-robin` Failover Script

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

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

```
#!/bin/sh
#
# Copyright (c) 2000 Silicon Graphics, Inc.  All Rights Reserved.
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of version 2 of the GNU General Public License as
# published by the Free Software Foundation.
#
# This program is distributed in the hope that it would be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
#
# Further, this software is distributed without any warranty that it is
# free of the rightful claim of any third person regarding infringement
# or the like.  Any license provided herein, whether implied or
# otherwise, applies only to this software file.  Patent licenses, if
# any, provided herein do not apply to combinations of this program with
# other software, or any other product whatsoever.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write the Free Software Foundation,
# Inc., 59 Temple Place - Suite 330, Boston MA 02111-1307, USA.
#
# Contact information: Silicon Graphics, Inc., 1600 Amphitheatre Pkwy,
# Mountain View, CA 94043, or:
#
# http://www.sgi.com
#
# For further information regarding this notice, see:
#
# http://oss.sgi.com/projects/GenInfo/NoticeExplan


#
# $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/lib/failsafe/bin
LOG="${DIR}/ha_cilog -g ha_script -s script"
```

```
FILE=/usr/lib/failsafe/policies/round-robin

# Read input file
input=$1
output=$2

{
  read version
  read name
  read owner
  read attr
  read mem1 mem2 mem3 mem4 mem5 mem6 mem7 mem8
  read afd1 afd2 afd3 afd4 afd5 afd6 afd7 afd8
} < ${input}

# 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;
```

3.1.3.3 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 `/usr/lib/failsafe/policies` directory. You can then configure the cluster configuration database to use your new failover script for the required resource groups.

3.2 Failover Script Interface

The following is passed to the failover script:

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

version

Linux FailSafe version. The Linux FailSafe release uses version number 1.

name

Name of the failover script (used for error validations and logging purposes).

owner

Logical name of the node that has the resource group allocated.

attributes

Failover attributes (`Auto_Failback` or `Controlled_Failback` must be included)

possibleowners

List of possible owners for the resource group. This list can be subset of the current node membership.

domain

Ordered list of nodes used at the last failover. (At the first failover, the initial failover domain is used.)

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

3.3 Example Failover Policies for Linux FailSafe

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

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

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

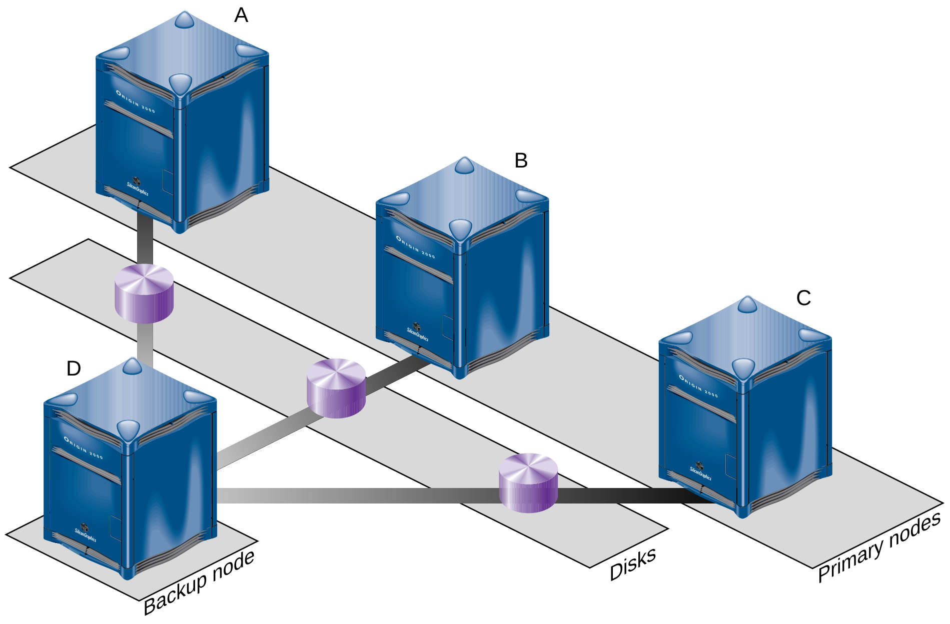
- *N* primary nodes and one backup node (*N*+1)
- *N* primary nodes and two backup nodes (*N*+2)
- *N* primary nodes and *M* backup nodes (*N*+*M*)

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

3.3.1 N+1 Configuration for Linux FailSafe

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

Figure 3–1 ***N+*1 Configuration Concept**



You could configure the following failover policies for load balancing:

- Failover policy for RG1:
  - Initial failover domain = A, D
  - Failover attribute = `Auto_Failback`
  - Failover script = `ordered`
- Failover policy for RG2:
  - Initial failover domain = B, D
  - Failover attribute = `Auto_Failback`
  - Failover script = `ordered`
- Failover policy for RG3:
  - Initial failover domain = C, D
  - Failover attribute = `Auto_Failback`
  - Failover script = `ordered`

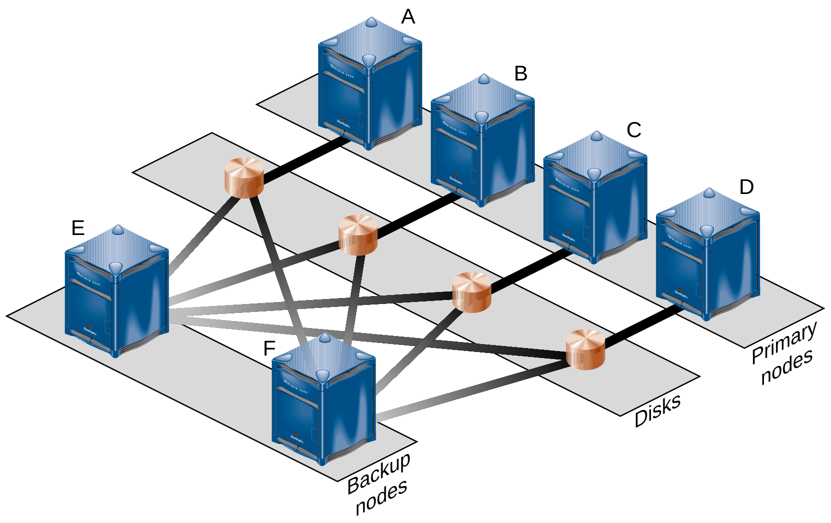
If node A fails, RG1 will fail over to node D. As soon as node A reboots, RG1 will be moved back to node A.

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

3.3.2 N+2 Configuration

Figure 3–2, *N+2 Configuration Concept* 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`

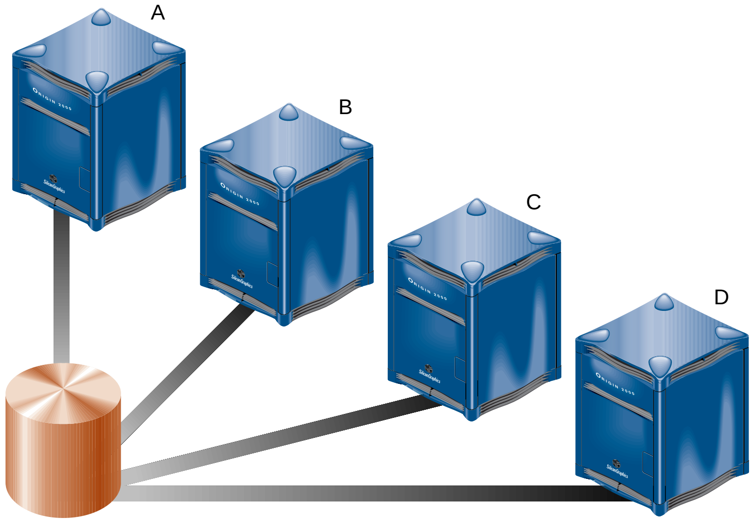
If node A fails, RG7 will fail over to node E. If node E also fails, RG7 will fail over to node F. If A is rebooted, RG7 will remain on node F.

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

3.3.3 N+M Configuration for Linux FailSafe

Figure 3–3, *N+M Configuration Concept* 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.

4 Defining a New Resource Type

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

- Section 4.1, *Using the GUI*
- Section 4.2, *Using `cluster_mgr` Interactively*
- Section 4.3, *Using `cluster_mgr` With a Script*

It also tells you how to test the results in Section 4.4, *Testing a New Resource Type*.

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, *Order Ranges* 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`

Range	Reservation
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, *Resource Type Order Numbers* shows the order numbers of the resource types provided with the release or available as plug-ins from SGI.

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`
502	`Samba`

- 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 `/usr/lib/failsafe/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, Linux 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-info` which takes a value that defines the export options for the file system. These options are used in the `kexportfs` 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` command. For example:

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

4.1 Using the GUI

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

4.1.1 Define a New Resource Type

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

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

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

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

Figure 4–1 Select Define a New Resource

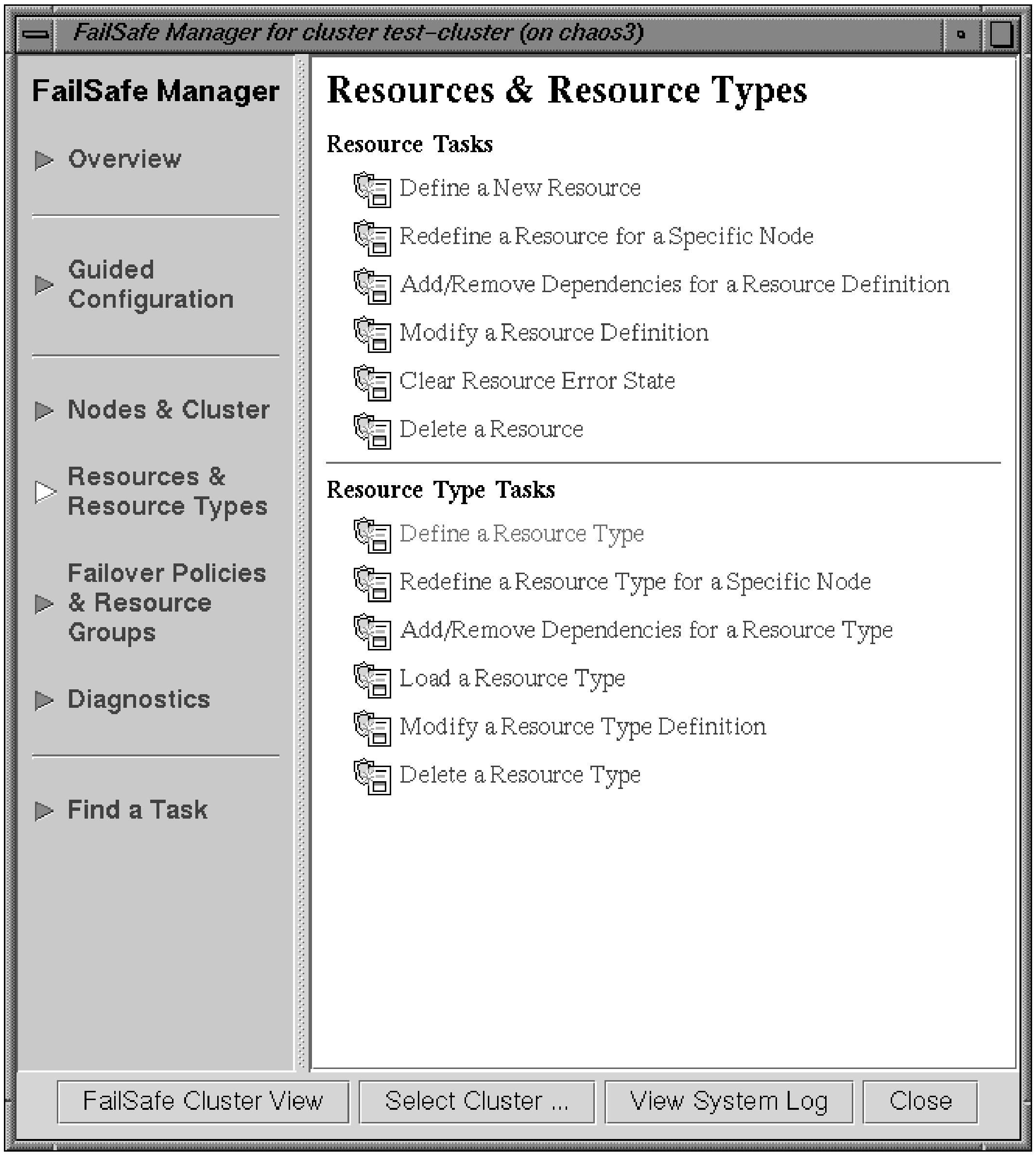
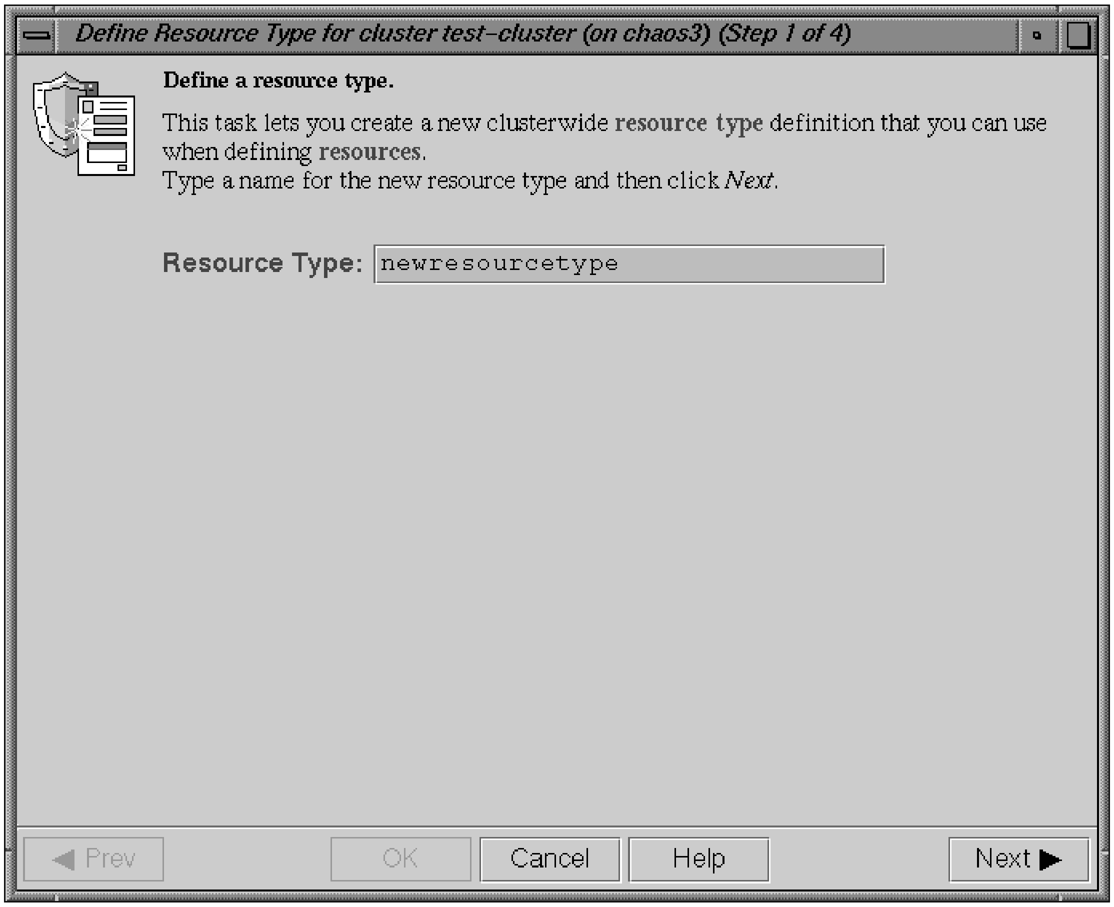

FailSafe Manager for cluster test-cluster (on chaos3)
FailSafe Manager
Overview
Guided Configuration
Nodes & Cluster
Resources & Resource Types
Failover Policies & Resource Groups
Diagnostics
Find a Task
Resources & Resource Types
Resource Tasks
Define a New Resource
Redefine a Resource for a Specific Node
Add/Remove Dependencies for a Resource Definition
Modify a Resource Definition
Clear Resource Error State
Delete a Resource
Resource Type Tasks
Define a Resource Type
Redefine a Resource Type for a Specific Node
Add/Remove Dependencies for a Resource Type
Load a Resource Type
Modify a Resource Type Definition
Delete a Resource Type
FailSafe Cluster View
Select Cluster ...
View System Log
Close

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



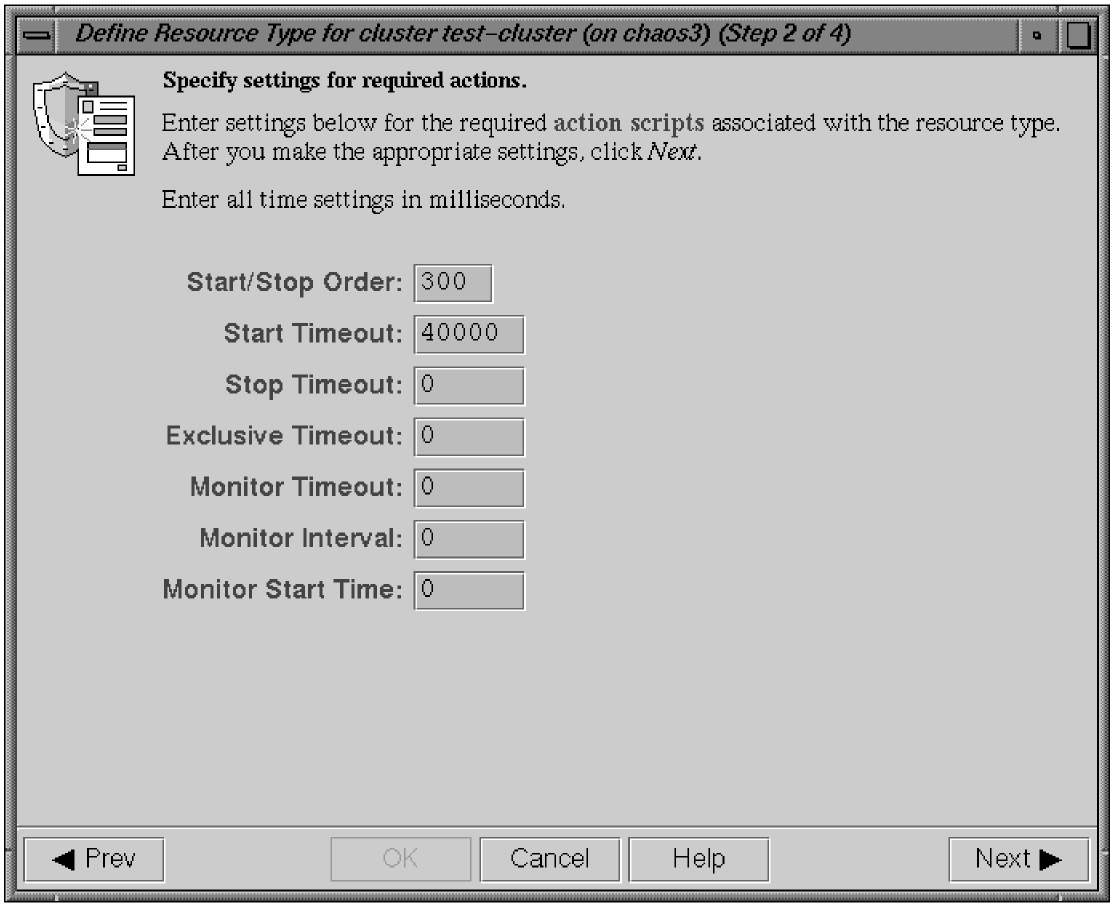
Define Resource Type for cluster test-cluster (on chaos3) (Step 1 of 4)

Define a resource type.

This task lets you create a new clusterwide **resource type** definition that you can use when defining **resources**.
Type a name for the new resource type and then click *Next*.

Resource Type: newresourcetype

Figure 4–3 Specify Settings for Required Actions



Define Resource Type for cluster test-cluster (on chaos3) (Step 2 of 4)

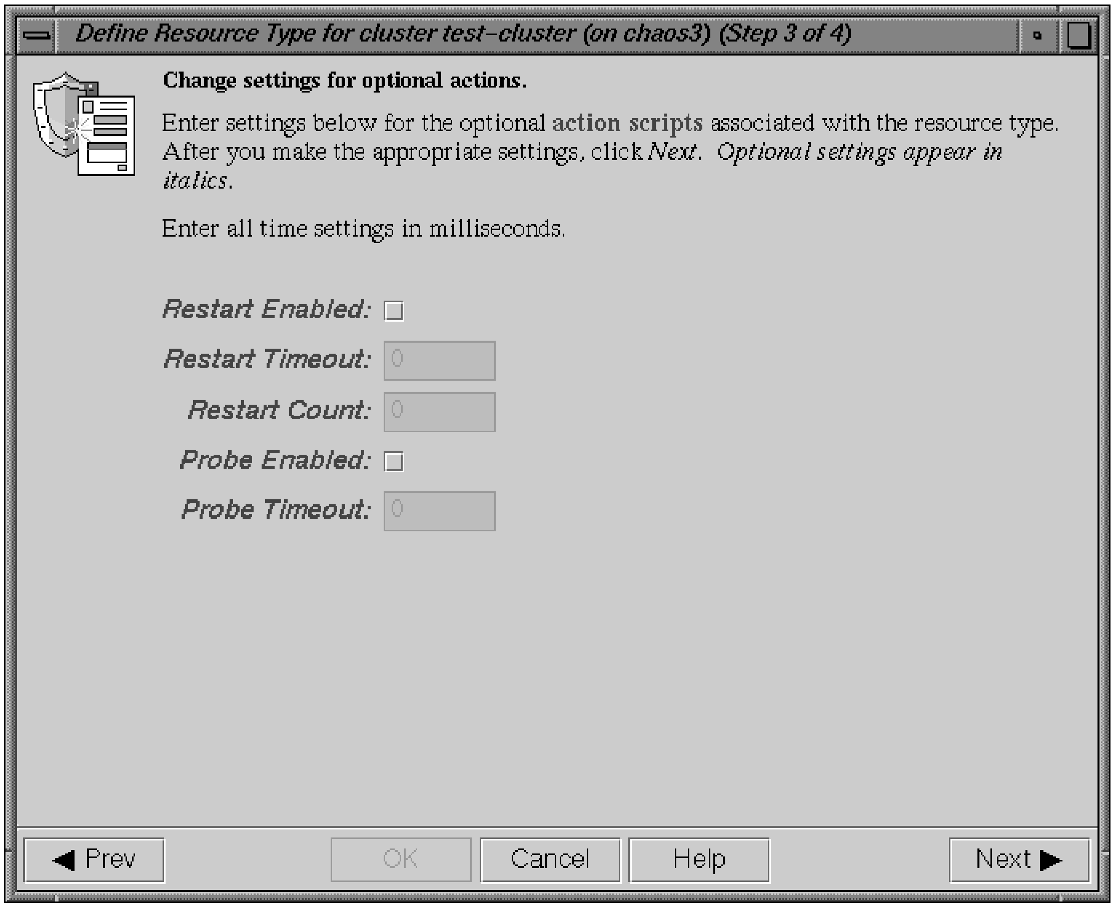
Specify settings for required actions.

Enter settings below for the required action scripts associated with the resource type. After you make the appropriate settings, click *Next*.

Enter all time settings in milliseconds.

Start/Stop Order:	300
Start Timeout:	40000
Stop Timeout:	0
Exclusive Timeout:	0
Monitor Timeout:	0
Monitor Interval:	0
Monitor Start Time:	0

Figure 4–4   Change Settings for Optional Actions


Define Resource Type for cluster test-cluster (on chaos3) (Step 3 of 4)

Change settings for optional actions.

Enter settings below for the optional action scripts associated with the resource type. After you make the appropriate settings, click *Next*. *Optional settings appear in italics.*

Enter all time settings in milliseconds.

Restart Enabled: ☐

Restart Timeout: 0

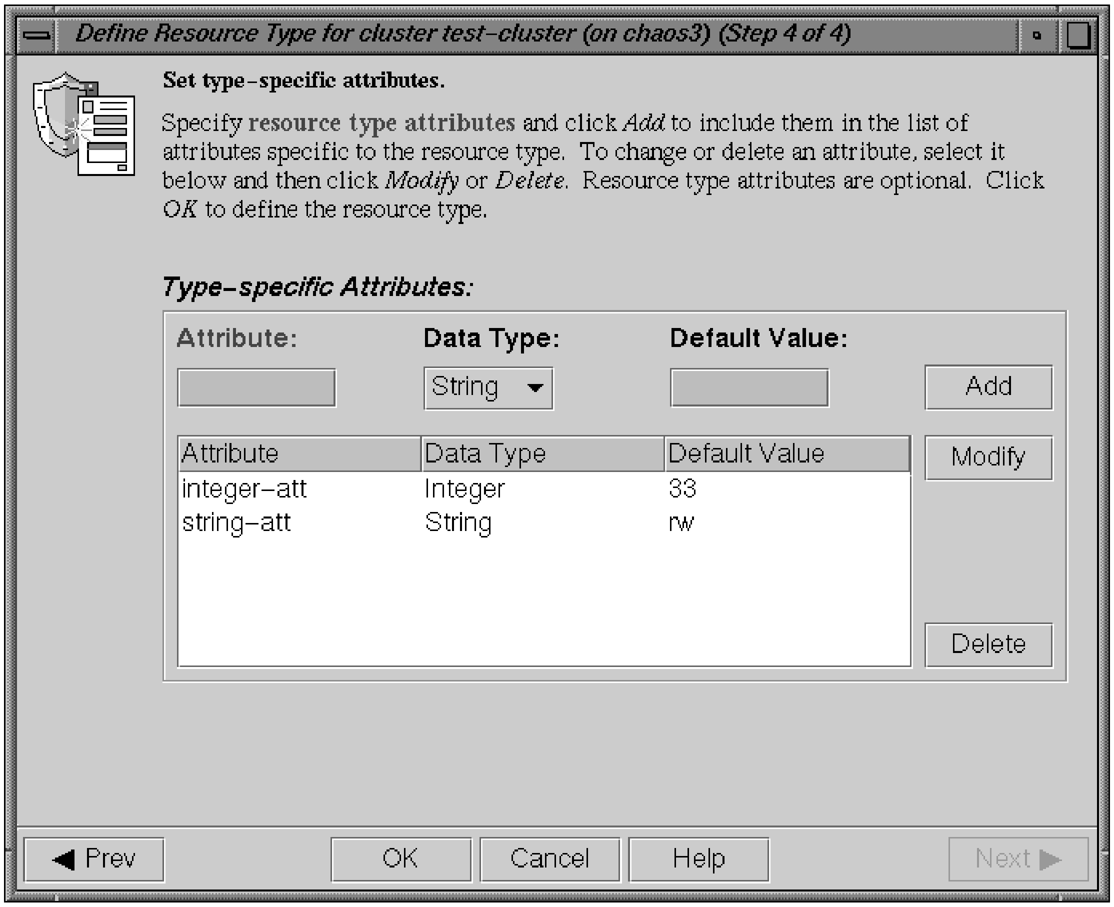
Restart Count: 0

Probe Enabled: ☐

Probe Timeout: 0

◀ Prev
OK
Cancel
Help
Next ▶

Figure 4–5 Set Type-specific Attributes


Define Resource Type for cluster test-cluster (on chaos3) (Step 4 of 4)

Set type-specific attributes.

Specify **resource type attributes** and click *Add* to include them in the list of attributes specific to the resource type. To change or delete an attribute, select it below and then click *Modify* or *Delete*. Resource type attributes are optional. Click *OK* to define the resource type.

Type-specific Attributes:

Attribute: (empty) Data Type: String Default Value: (empty)

Attribute	Data Type	Default Value
integer–att	Integer	33
string–att	String	rw

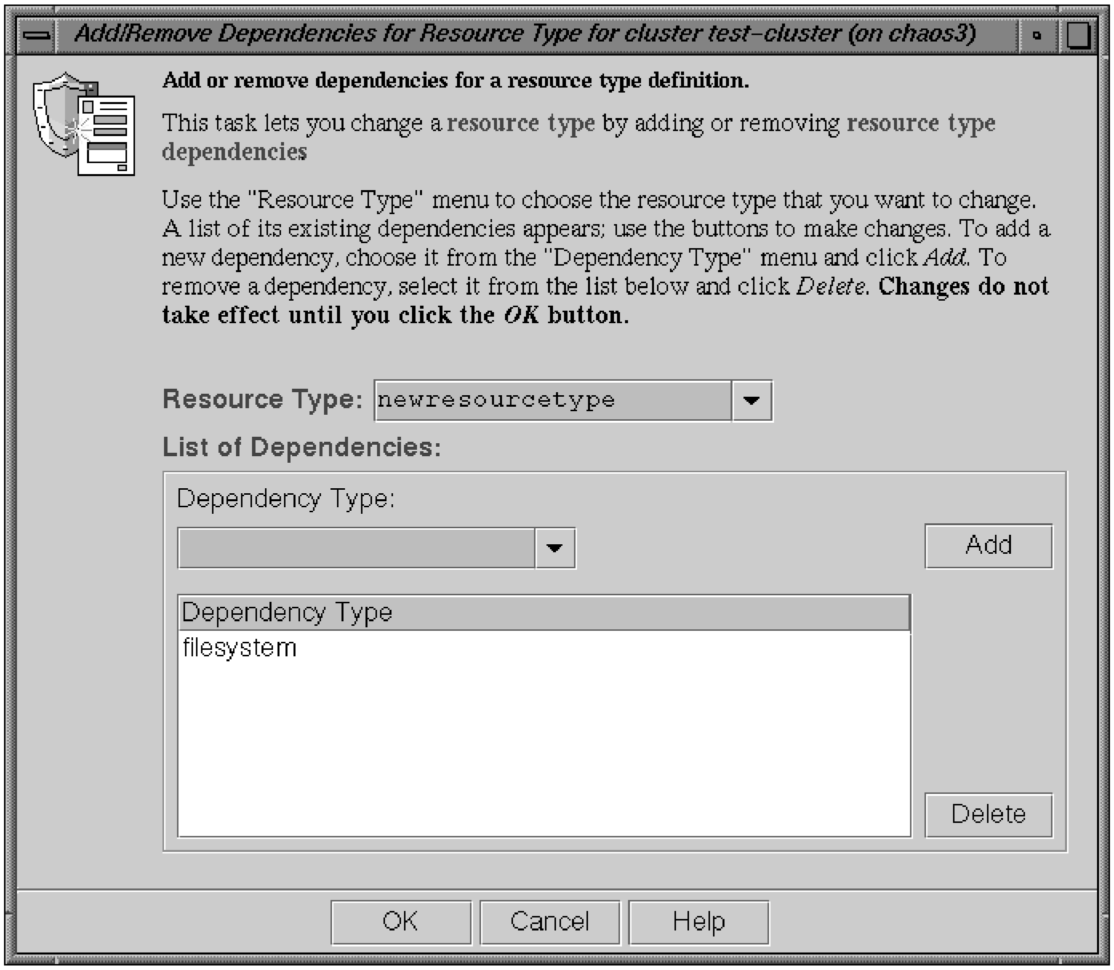
4.1.2 Define Dependencies

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

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

Figure 4–6, *Add Dependencies* shows an example of adding two dependencies (`filesystem` and `NFS`) to the `newresourcetype` resource type.

Figure 4–6 Add Dependencies



Add/Remove Dependencies for Resource Type for cluster test-cluster (on chaos3)

Add or remove dependencies for a resource type definition.

This task lets you change a **resource type** by adding or removing **resource type dependencies**

Use the "Resource Type" menu to choose the resource type that you want to change. A list of its existing dependencies appears; use the buttons to make changes. To add a new dependency, choose it from the "Dependency Type" menu and click *Add*. To remove a dependency, select it from the list below and click *Delete*. **Changes do not take effect until you click the *OK* button.**

Resource Type: newresourcetype

List of Dependencies:

Dependency Type:

Dependency Type
filesystem

4.2 Using `cluster_mgr` Interactively

The following steps show the use of `cluster_mgr` interactively to define a resource type called `newresourcetype`. Note that you can have multiple resource types. For example, if you want to have some IP addresses that allow local restart (restart mode = 0) and some that do not (restart mode = 1), you can copy the `IP_address` type to a new type named `IP_address2` and change just that value in the `IP_address2`.

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

```
*
?
\
#
```

---

1. Log in as `root`.

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

```
# /usr/lib/failsafe/bin/cluster_mgr -p
Welcome to Linux FailSafe Cluster Manager Command-Line Interface

cmgr>
```

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

```
cmgr> set cluster test
```

> 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.

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

```
cmgr> define resource_type newresourcetype

(Enter "cancel" at any time to abort)

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

DEFINE RESOURCE TYPE OPTIONS

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

Enter option:1

No current resource type actions

Action name ? start
```

```
Executable Time? **40000**
Monitoring Interval? **0**
Start Monitoring Time? **0**

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

Enter option:1

Current resource type actions:
        Action - 1: start

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

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

Enter option:3

No current type specific attributes

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

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

Enter option:3
```

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

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

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

Enter option:5

No current resource type dependencies

Dependency name ? filesystem

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

Enter option:7

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

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

No current resource type dependencies

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

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

Enter option:9
Successfully defined resource_type newresourcetype

cmgr> show resource_types in cluster test

NFS
template
newresourcetype
statd
MAC_address
IP_address
filesystem
volume

cmgr> exit
#
```

4.3 Using `cluster_mgr` With a Script

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

cluster_mgr -f *scriptname*

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

```
#!/usr/lib/failsafe/bin/cluster_mgr -f
```

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

```
#!/usr/lib/failsafe/bin/cluster_mgr -if
```

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

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

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

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

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

set cluster test
define resource_type newresourcetype
set order to 300
set restart_mode to 0
add action start
set exec_time to 40000
set monitor_interval to 0
set monitor_time to 0
done
add action stop
set exec_time to 40000
set monitor_interval to 0
set monitor_time to 0
done
add type_attribute integer-att
set data_type to integer
set default_value to 33
done
add type_attribute string-att
set data_type to string
set default_value to rw
done
add dependency filesystem
done
quit
```

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

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

#
```

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

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

NFS
template
newresourcetype
statd
```

```
MAC_address
IP_address
filesystem
volume
```

4.4 Testing a New Resource Type

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

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

   For example, using `cluster_mgr`:

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

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

   For example:

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

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

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

5. If there are resource group errors, do the following:
   - Check the `srmd` logs (`/var/log/failsafe/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/log/failsafe/script_`*`nodename`* on the same node) for `IP_address exclusive` script errors.
   - After the fixing the problems in the action script, perform an `offline_force` operation to clear the error. For example:

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

5 Testing Scripts

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

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.

5.1 General Testing and Debugging Techniques

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

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

  ```
  set -x
  ```

- To check that an application is running on a node, you may be able to use a command provided by the application.
- Another way to check that an application is running on a node, is to enter this command on that node:

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

  application is the name (or a portion of the name) of the executable for the application.
- To show the status of a resource, use the following `cluster_mgr` command:

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

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

  ```
  cmgr> show status of node nodename
  ```

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

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

5.2 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/log/failsafe/script_`*`nodename`* file in each node in the cluster.

5.3 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
   ```

   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
```

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

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

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

The following example shows:

- The exit status of the action script is 1
- The exit status of the resource is 2

The use of `anonymous` indicates that the script was run manually. When the script is run by Linux 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/log/failsafe/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:
/bin/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:
/bin/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/run/failasafe/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/log/failsafe/script_`*`nodename`*.

5.4 Special Testing Considerations for the `monitor` Script

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

Use this checklist for testing a `monitor` script:

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

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

A Starting the FailSafe Manager

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

- On a system with `sysadm_failsafe2-client` and KDE or GNOME installed, choose "FailSafe Manager" from the Applications menu. (After installing `sysadm_failsafe2-client`, you may need to restart the desktop's launch bar before the Failsafe Manager will show up in the Applications menu. (To do this in KDE, right-click on the panel and select Restart from the menu.)
- On a system with `sysadm_failsafe2-client` installed, enter the following command line:

  ```
  $ /usr/bin/fstask
  ```

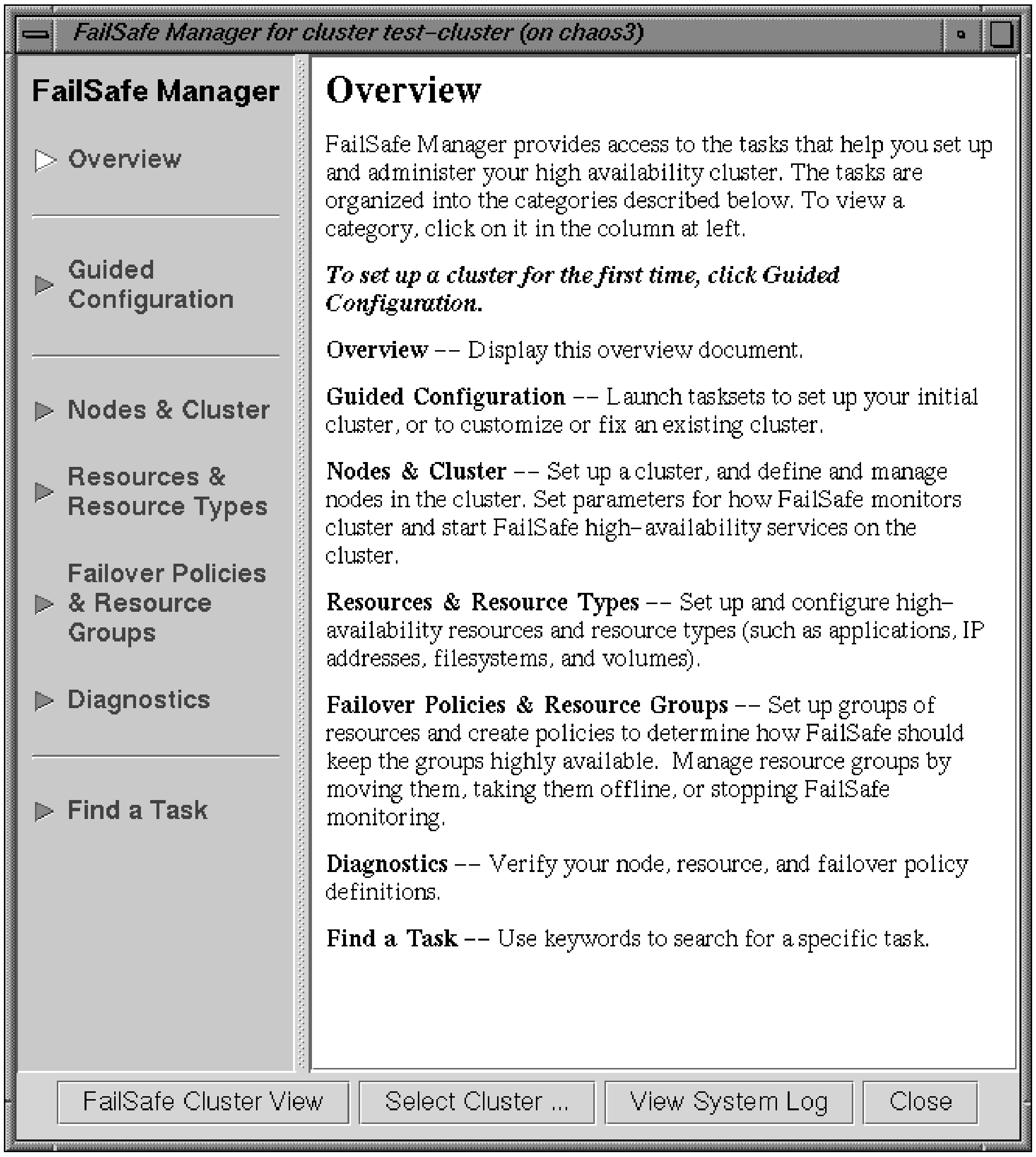
- In your web browser, enter the following, where *server* is the name of the node in the pool or cluster that you want to administer:

  ```
  http://server/FailSafeManager/
  ```

  At the resulting Web page, click on the icon. The server must have `sysadm_failsafe2-web` installed.

Figure A–1, *FailSafe Manager*, shows the FailSafe Manager.

Figure A–1 FailSafe Manager



Overview

FailSafe Manager provides access to the tasks that help you set up and administer your high availability cluster. The tasks are organized into the categories described below. To view a category, click on it in the column at left.

To set up a cluster for the first time, click Guided Configuration.

Overview –– Display this overview document.

Guided Configuration –– Launch tasksets to set up your initial cluster, or to customize or fix an existing cluster.

Nodes & Cluster –– Set up a cluster, and define and manage nodes in the cluster. Set parameters for how FailSafe monitors cluster and start FailSafe high–availability services on the cluster.

Resources & Resource Types –– Set up and configure high–availability resources and resource types (such as applications, IP addresses, filesystems, and volumes).

Failover Policies & Resource Groups –– Set up groups of resources and create policies to determine how FailSafe should keep the groups highly available. Manage resource groups by moving them, taking them offline, or stopping FailSafe monitoring.

Diagnostics –– Verify your node, resource, and failover policy definitions.

Find a Task –– Use keywords to search for a specific task.

B Using the SRM Script Library

The `/usr/lib/failsafe/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.

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

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

- Linux FailSafe application interfaces
- 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

B.1 Linux FailSafe application interfaces

The Linux FailSafe application interface identifies resources by two strings:

- Resource name
- Resource type

For example, a resource named `vol1` of resource type `volumes` is identified by the following: `vol1 volumes`.

Using the script library simplifies interaction with the interface. If you do not use the script library, you must understand the following file formats used by action scripts:

- 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. SRMD can also pass action flags for each resource in the input file. The format of a line in the input file is as follows (fields separated by white space):

  resource_name action_flags

- Output file, in which the executable writes the return the status of the each resource on a separate line using the following format (fields separated by white space):

  resource_name resource_status

- (optional) Input parameters file, which contains the name of the cluster:

ClusterName *clustername*

The following codes are defined in `/usr/lib/failsafe/common_scripts/scriptlib`:

- `HA_SUCCESS`
- `HA_NOT_RUNNING`
- `HA_INVAL_ARGS`
- `HA_CMD_FAILED`
- `HA_RUNNING`
- `HA_NOTSUPPORTED`
- `HA_NOCFGINFO`

B.2 Set Global Definitions

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

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.

B.2.1 Global Variable

B.2.1.1 HA_HOSTNAME

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

Default: `‘uname -n‘`

B.2.2 Command Location Variables

B.2.2.1 HA_CMDSPATH

Path to user commands.

Default: `/usr/lib/failsafe/bin`

B.2.2.2 HA_PRIVCMDSPATH

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

Default: `/usr/lib/sysadm/privbin`

B.2.2.3 HA_LOGCMD

Command used to log information.

Default: `ha_cilog`

B.2.2.4 `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`

B.2.2.5 `HA_SCRIPTTMPDIR`

Location of the script temporary directory.

Default: `/var/run/failsafe/tmp`

B.2.3 Database Location Variables

B.2.3.1 `HA_CDB`

Location of the cluster configuration database.

Default: `/var/lib/failsafe/cdb/cdb.db`

B.2.4 Script Log Level Variables

B.2.4.1 `HA_NORMLVL`

Normal level of script logs.

Default: 1

B.2.4.2 `HA_DBGLVL`

Debug level of script logs.

Default: 10

B.2.5 Script Log Variables

B.2.5.1 `HA_SCRIPTGROUP`

Log for the script group.

Default: `ha_script`

B.2.5.2 `HA_SCRIPTSUBSYS`

Log for the script subsystem.

Default:`script`

B.2.6 Script Logging Command Variables

B.2.6.1 `HA_DBGLOG`

Command used to log debug messages from the scripts.

Default: `ha_dbglog`

B.2.6.2 `HA_CURRENT_LOGLEVEL`

The value of the current logging level. `ha_log` will only output messages if this value is greater than or equal to `HA_NORMLVL`. `ha_dbglog` will only output messages if this value is greater than or equal to `HA_DBGLVL`.

Default: 2

B.2.6.3 `HA_LOG`

Command used to log the scripts.

Default: `ha_log`

B.2.7 Script Error Value Variables

B.2.7.1 `HA_SUCCESS`

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

Default: 0

B.2.7.2 `HA_NOT_RUNNING`

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

Default: 0

B.2.7.3 `HA_INVAL_ARGS`

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

Default: 1

B.2.7.4 `HA_CMD_FAILED`

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

Default: 2

B.2.7.5 `HA_RUNNING`

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

Default: 2

B.2.7.6 `HA_NOTSUPPORTED`

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

Default: 3

B.2.7.7 `HA_NOCFGINFO`

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

Default: 4

B.3 Check Arguments

An action script has the following arguments: an input file, `HA_INFILE`, an output file `HA_OUTFILE`, and an optional parameter file `HA_PARAMFILE`. These become the positional arguments to the script, `$1`, `$2` and `$3` 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;
}
```

B.4 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 ${HA_INFILE}`
    do
        HA_RES_NAMES="${HA_RES_NAMES} ${HA_RESOURCE}";
    done
}
```

B.5 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_LOG} "$1 failed"
        ${HA_LOG} ‘cat ${HA_SCRIPTTMPDIR}/script.$$‘
    fi

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

    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 string `$HA_CMD_OUTPUT`.

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

    HA_CMD_OUTPUT=‘${HA_CMD}‘;
```

```
        ha_exit_code=$?;

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

        return $ha_exit_code;
    }
```

B.6 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
    }
```

B.7 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 a series of name-value field pairs, where a name field is followed by the value of the name, separated by whitespace. This appears 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;
    }
```

B.8 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
}
```

B.9 Get Resource Information

The `ha_get_info()` and `ha_get_info_debug()`functions read resource information. `$1` is the resource type and `$2` is the resource name, 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. The return value of the query is passed back to the caller, 0 indicates success. All errors are logged. 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.

The function ha_get_info_debug() differs from ha_get_info in that it will attempt the resource query a single time, instead of retrying as ha_get_info() would do. This is likely to be useful for testing due to faster returns and less complexity. You can call this function directly, or you can create the file /var/run/failsafe/resourceQuery.debug which will cause all invocations of ha_get_info() in all scripts on the node to be diverted to ha_get_info_debug() until the file is removed.

```
ha_get_info()
{
    if [ -f /var/run/failsafe/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};
}


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};
}
```

B.10 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

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

cluster administrator

The person responsible for managing and maintaining a cluster.

cluster configuration database

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

cluster node

A single Linux execution environment. In other words, a single physical machine or single running Linux kernel. In current Linux environments this will be an individual computer. The term **node** is used within this guide to indicate this meaning, as opposed to any alternate meaning such as a network node.

control messages

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

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

control network

The network that connects nodes through their network interfaces (typically Ethernet) such that Linux FailSafe can maintain a cluster's high availability by sending heartbeat messages and control messages through the network to the attached nodes. Linux 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 configuration 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 node membership (such as when a node fails or starts), or a manual request by the administrator.

failover attribute

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

failover domain

The ordered list of nodes on which a particular resource group can be allocated. The nodes listed in the failover domain must be within the same cluster; however, the failover domain does not have to include every node in the cluster.The administrator defines the initial failover domain when creating a failover policy. This list is transformed into the run-time failover domain by the failover script the run-time failover domain is what is actually used to select the failover node. Linux 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, Linux FailSafe allocates a given resource group to the first node listed in the run-time failover domain that is also in the node membership; the point at which this allocation takes place is affected by the failover attributes.

failover policy

The method used by Linux 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 Linux 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 node membership.

Failsafe database

See **cluster configuration database**

heartbeat messages

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

heartbeat interval

Interval between heartbeat messages. The node timeout value must be at least 10 times the heartbeat interval for proper Linux 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/log/failsafe/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 Linux FailSafe notifications for a particular log group. A log file is part of the log configuration for a log group. By default, log files reside in the `/var/log/failsafe` directory, but the cluster administrator can customize this. Note: Linux FailSafe logs both normal operations and critical errors to `/var/log/failsafe`, as well as to individual logs for specific log groups.

log group

A set of one or more Linux 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 Linux FailSafe will write into an associated log group’s log file. A log level is part of the log configuration for a log group.

node

See **cluster node**

node ID

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

node membership

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

node timeout

If no heartbeat is received from a node in this period of time, the node is considered to be dead. The node timeout value must be at least 10 times the heartbeat interval for proper Linux 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, Linux FailSafe stops the group (if needed) and stops monitoring the group. An offline resource group can be running on a node, yet not under Linux FailSafe control. If the cluster administrator specifies the `detach only` option while taking the group offline, then Linux 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 Linux 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, Linux 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 Linux FailSafe will not start the group but will begin monitoring the group.

owner host

A system that can control a node remotely, for example 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.

pool

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

port password

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

powerfail mode

When powerfail mode is turned `on`, Linux 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, Linux FailSafe uses heuristics to try to estimate whether the machine has been powered down. If the heuristic algorithm returns with success, Linux FailSafe assumes the remote machine has been reset successfully. When powerfail mode is turned `off`, the heuristics are not used and Linux FailSafe may not be able to detect node power failures.

process group

A group of application instances. Each application instance can consist of one or more UNIX processes and spans only one node.

process membership

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

resource

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

resource dependency

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

resource dependency list

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

resource group

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

resource keys

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

resource name

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

resource type

A particular class of resource. All of the resources in a particular resource type can be handled in the same way for the purposes of failover. Every resource is an instance of exactly one resource type. A resource type is identified by a simple name; this name must be unique within a cluster. A resource type can be defined for a specific node or for an entire cluster. A resource type that is defined for a node overrides a cluster-wide resource type definition with the same name; this allows an individual node to override global settings from a cluster-wide resource type definition.

resource type dependency

A set of resource types upon which a resource type depends. For example, the `filesystem` resource type depends upon the `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 non–negative integer. In a resource group, the start/stop orders of the resource types determine the order in which the resources will be started when Linux FailSafe brings the group online and will be stopped when Linux 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 Linux FailSafe brings a resource group online, all volume resources in the group will be started before all file system resources in the group.

system controller port

A port located on a node that provides a way to power-cycle the node remotely. One example of this in the x86-based hardware arena is the Intel EMP (Emergency Management Port) supplied on some Intel motherboards. Enabling or disabling a system controller port in the cluster configuration database (CDB) tells Linux 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 Linux FailSafe to use in the process of computing node membership for the cluster, when exactly half the nodes in the cluster are up and can communicate with each other. If a tie-breaker node is not specified, Linux 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