TruCluster Server

Cluster Highly Available Applications

Part Number: AA-RHH0D-TE

June 2001

Product Version: TruCluster Server Version 5.1A

Operating System and Version: Tru64 UNIX Version 5.1A

This manual describes how to make applications highly available on a Compaq Tru64 UNIX TruCluster Server Version 5.1A cluster and describes the application programming interface (API) libraries of the TruCluster Server product.

Compaq Computer Corporation
Houston, Texas

Contents

Figures

Tables

About This Manual

This manual describes how to move applications to the TruCluster™ Server environment, and how to use TruCluster Server functionality to make applications highly available.

This manual also describes how to use TruCluster Server application programming interfaces (APIs) to take advantage of cluster technologies such as the distributed lock manager (DLM), cluster alias, and Memory Channel.

Read the TruCluster Server *Cluster Technical Overview* for an understanding of the TruCluster Server cluster subsystems before moving applications to your TruCluster Server environment.

Audience

This manual is intended for system administrators who want to deploy highly available applications on TruCluster Server or move applications from a TruCluster Available Server or TruCluster Production Server to a TruCluster Server environment. This manual is also intended for developers who want to write distributed applications that need the synchronization services of the DLM, alias functionality, or the high-performance capabilities of the Memory Channel.

New and Changed Features

Section 2.12.2 is a new section; it explains how to create a single-instance, highly available Apache HTTP Server using the cluster application availability (CAA) subsystem.

Section 4.6 is revised to provide more information on process identification (PID) numbers available to cluster members.

Section 7.7 is a new section; it describes how to test the status of a cluster member during a rolling upgrade.

Section 7.8 is a new section; it discusses file access resilience in a cluster.

Organization

This manual is organized as follows:

Part 1	Describes how to get single-instance and multi-instance applications up and running on TruCluster Server.
Chapter 1	Describes the general types of cluster applications.
Chapter 2	Describes how to use the cluster application availability (CAA) facility for single-instance application availability on TruCluster Server.
Chapter 3	Describes how to use the default cluster alias for multi-instance application availability on TruCluster Server.
Part 2	Describes how to move applications to a TruCluster Server environment.
Chapter 4	Discusses general issues to consider before moving applications to TruCluster Server.
Chapter 5	Describes how to move Available Server Environment (ASE)-style applications to TruCluster Server.
Chapter 6	Describes how to move distributed applications to TruCluster Server.
Part 3	Describes how to use APIs to take advantage of TruCluster Server technology.
Chapter 7	Describes how to change an application's source code to allow it to run in a clusterized environment.
Chapter 8	Describes how to use the features of the cluster alias API.
Chapter 9	Describes how to use the features of the DLM.
Chapter 10	Describes how to use the Memory Channel API library.

Related Documents

Consult the following TruCluster Server documentation for assistance in cluster configuration, installation, and administration tasks:

- TruCluster Server *Software Product Description* (SPD) — The comprehensive description of the TruCluster Server Version 5.1A product. You can find the latest version of the SPD at the following URL:

  `http://www.tru64unix.compaq.com/docs/pub_page/spds.html`

- TruCluster Server *Cluster Technical Overview* — Provides an overview of the TruCluster Server technology.
- TruCluster Server *Cluster Release Notes* — Provides important information about TruCluster Server Version 5.1A, including new features, known problems, and workarounds.

- TruCluster Server *Cluster Hardware Configuration* — Describes how to set up the processors that are to become cluster members, and how to configure cluster shared storage.
- TruCluster Server *Cluster Installation* — Describes how to install the TruCluster Server software product.
- TruCluster Server *Cluster Administration* — Describes cluster-specific administration tasks.
- TruCluster Server *Cluster LAN Interconnect* — Describes how to configure and administer a Local Area Network (LAN) as a cluster interconnect.

You can find the latest version of the TruCluster Server documentation at the following URL:

```
http://www.tru64unix.compaq.com/docs/pub_page/cluster_list.html
```

In addition, have available the following manuals from the Compaq Tru64™ UNIX operating system software documentation set:

- Tru64 UNIX *Release Notes*
- Tru64 UNIX *System Administration*
- Tru64 UNIX *Network Administration: Connections*
- Tru64 UNIX *Network Administration: Services*
- Tru64 UNIX *Programmer's Guide*

Icons on Tru64 UNIX Printed Manuals

The printed version of the Tru64 UNIX documentation uses letter icons on the spines of the manuals to help specific audiences quickly find the manuals that meet their needs. (You can order the printed documentation from Compaq.) The following list describes this convention:

G	Manuals for general users
S	Manuals for system and network administrators
P	Manuals for programmers
R	Manuals for reference page users

Some manuals in the documentation help meet the needs of several audiences. For example, the information in some system manuals is also used by programmers. Keep this in mind when searching for information on specific topics.

The *Documentation Overview* provides information on all of the manuals in the Tru64 UNIX documentation set.

Reader’s Comments

Compaq welcomes any comments and suggestions you have on this and other Tru64 UNIX manuals.

You can send your comments in the following ways:

- Fax: 603-884-0120 Attn: UBPG Publications, ZKO3-3/Y32
- Internet electronic mail: `readers_comment@zk3.dec.com`

  A Reader’s Comment form is located on your system in the following location:

  `/usr/doc/readers_comment.txt`

Please include the following information along with your comments:

- The full title of the manual and the order number. (The order number appears on the title page of printed and PDF versions of a manual.)
- The section numbers and page numbers of the information on which you are commenting.
- The version of Tru64 UNIX that you are using.
- If known, the type of processor that is running the Tru64 UNIX software.

The Tru64 UNIX Publications group cannot respond to system problems or technical support inquiries. Please address technical questions to your local system vendor or to the appropriate Compaq technical support office. Information provided with the software media explains how to send problem reports to Compaq.

Conventions

This manual uses the following typographical conventions:

#	A number sign represents the superuser prompt.
`%` **`cat`**	Boldface type in interactive examples indicates typed user input.
`file`	Italic (slanted) type indicates variable values, placeholders, and function argument names.
⋮	A vertical ellipsis indicates that a portion of an example that would normally be present is not shown.
`cat`(1)	A cross-reference to a reference page includes the appropriate section number in parentheses. For example, `cat`(1) indicates that you can find information on the `cat` command in Section 1 of the reference pages.

Part 1

Running Applications on TruCluster Server

1 Cluster Applications

The TruCluster Server Version 5.1A environment allows you to make existing applications more highly available to their clients with minimal effort. You can also develop new applications that take greater advantage of the performance and availability features of a cluster. You can deploy highly available applications having no embedded knowledge that they are executing in a cluster, and that can access their disk data from any member in the cluster.

Using the TruCluster Server cluster application availability (CAA) subsystem, applications can recover from member and resource failures by restarting on another cluster member. In TruCluster Server, CAA replaces the Available Server Environment (ASE), which, in previous TruCluster Software products, provided the ability to make applications highly available. However, unlike the case with ASE, in a TruCluster Server cluster, you do not have to explicitly manage storage resources and mount file systems on behalf of a highly available application. The Cluster File System (CFS) and device request dispatcher make file and disk storage available clusterwide.

TruCluster Server also lets you run components of distributed applications in parallel, providing high availability while taking advantage of cluster-specific synchronization mechanisms and performance optimizations.

This chapter discusses the basic types of cluster applications.

If you are new to TruCluster Server, read the TruCluster Server *Cluster Technical Overview* for an overview of the TruCluster Server product. Also, read the following chapters of this manual to understand the TruCluster Server features that applications can use to become highly available:

- Chapter 2 discusses using the CAA subsystem to make single-instance applications highly available.
- Chapter 3 discusses using the default cluster alias with multi-instance applications for transparent client access.

If you are moving from a TruCluster Available Server or TruCluster Production Server environment, read Part 2, which discusses how to move ASE-style applications to TruCluster Server.

1.1 Application Types

Cluster applications can be divided into the basic types that are listed in Table 1–1.

Table 1–1: TruCluster Server Application Types

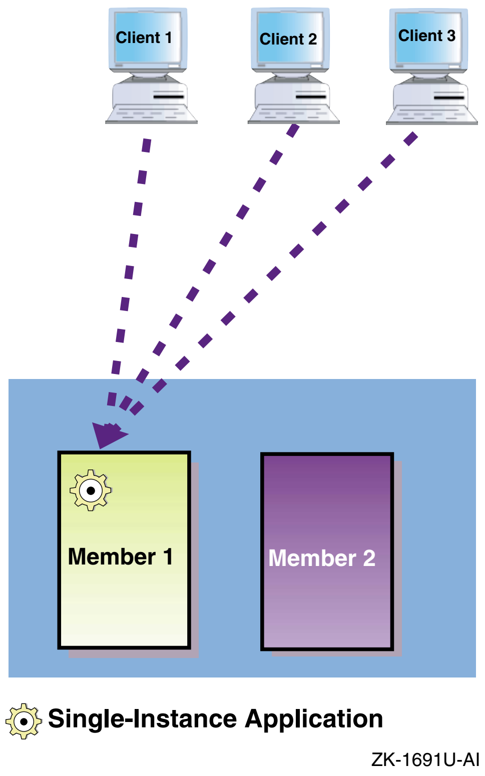
Type	Description	Example
Single-instance	A single-instance application runs on only one member of a cluster at a time.	Single-instance Dynamic Host Configuration Protocol (DHCP) server
Multi-instance	A multi-instance application runs on one or more members of a cluster. It has no restrictions that prevent it from running more than one copy in a cluster.	Multi-instance Netscape FastTrack Server
Distributed	A distributed application runs as independent, cooperating modules that are distributed among cluster members.	Oracle Parallel Server (OPS)

The following sections describe these three application types in more detail.

1.1.1 Single-Instance Applications

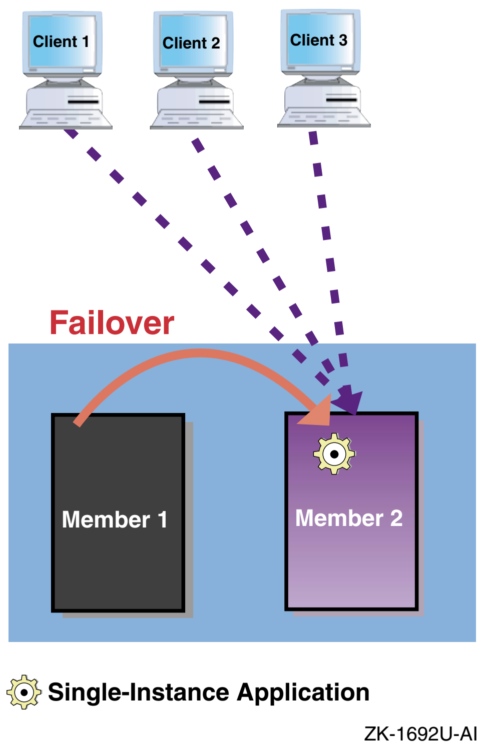
A single-instance application runs on only one cluster member at a time. All clients access the single-instance application on one member as shown in Figure 1–1.

Figure 1–1: Accessing a Single-Instance Application



If the cluster member where the application is installed fails or loses access to a resource, the application can be failed over to another running member. To restart single-instance applications on another cluster member, TruCluster Server provides the CAA system. (See Chapter 2 for information on how to use CAA.) Figure 1–2 shows how the failure of one cluster member results in the failover of an application to the second cluster member.

Figure 1–2: Application Failover Using CAA



Because TruCluster Server is binary compatible with Tru64 UNIX Version 5.1A, any application that runs properly on Tru64 UNIX *and* recognizes the new-style device names (`dsk`) will run on at least one member of a cluster. See Section 4.2 for more information about device naming.

Table 1–2 summarizes single-instance application architectural differences in TruCluster Server Version 5.1A and previous versions of the TruCluster product.

Table 1–2: Single-Instance Application Architectural Differences

TruCluster Server Version 5.0 or Later	TruCluster Available Server and TruCluster Production Server Version 1.6 or Earlier
Application installation and configuration on a single member of the cluster can be seen on all members because of the Cluster File System (CFS). However, the application can only be run on one member at a time because no synchronization has been built into the application.	Application installation and configuration on a single member of the cluster is only visible and usable on that member.
CAA is used to define application start, stop, and network interface dependency needs. Storage is transparently handled by CFS and the device request dispatcher. You can use the cluster alias or an interface alias within a CAA action script to handle IP aliasing needs.	ASE is used to define application start, stop, storage, and IP aliases.

1.1.2 Multi-Instance Applications

A multi-instance application runs multiple instances of itself on a single system or on more than one member of a cluster. If multiple instances of an application can run on more than one system, then the application has been modified to have some cluster awareness, for example, temporary file names have been changed to avoid being overwritten.

A multi-instance application is typically highly available; the failure of one cluster member does not affect the instances of the application running on other members. Multi-instance applications can take advantage of cluster aliasing to distribute client requests among all cluster members as shown in Figure 1–3. In this figure, clients are accessing a multi-instance application through the cluster alias `deli`.

Figure 1–3: Accessing a Multi-Instance Application

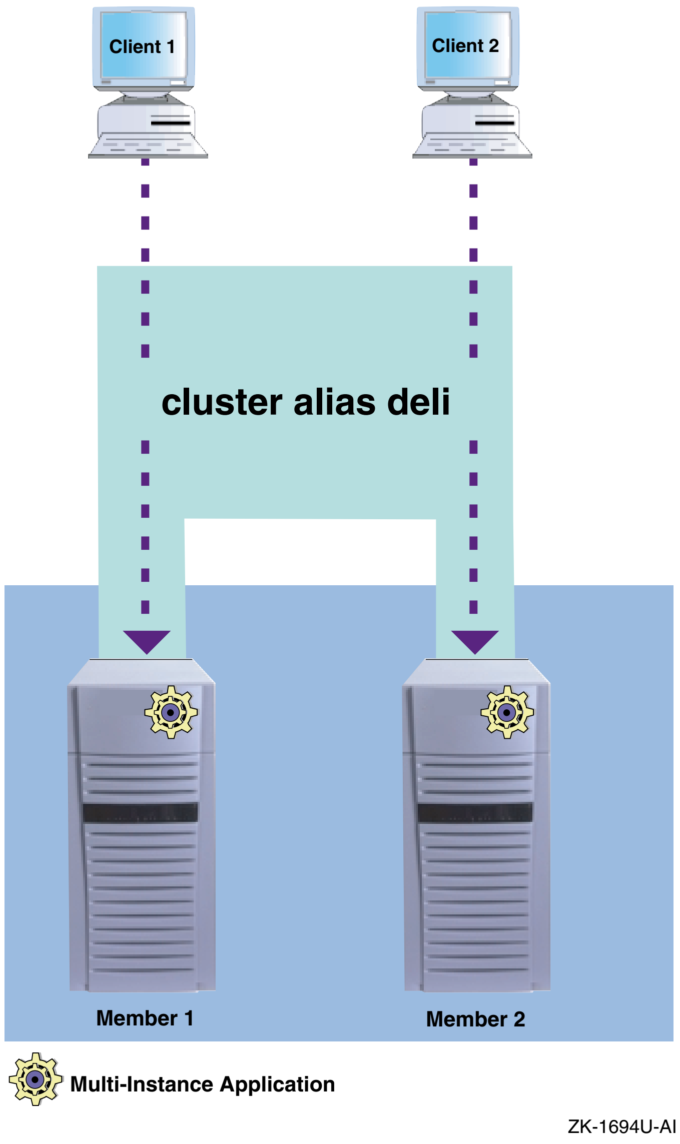


Table 1–3 summarizes multi-instance application architectural differences in TruCluster Server Version 5.1A and previous versions of the TruCluster product.

Table 1–3: Multi-Instance Application Architectural Differences

TruCluster Server Version 5.0 or Later	TruCluster Available Server and TruCluster Production Server Version 1.6 or Earlier
A multi-instance application that is cluster-aware makes explicit use at the source level of the distributed lock manager (DLM) application programming interface (API). Cluster alias automates routing of client requests among instances of the application.	An application can be run as multi-instance by creating multiple ASE services, each with its own interface alias. There is no cluster alias. Load balancing and delivery of client requests to application instances must be designed into the application. The application must be able to access raw distributed raw disk (DRD) devices for storage and must use UNIX file locking semantics, the DLM, or an application-specific locking mechanism.
A multi-instance application that is not cluster-aware must synchronize access to shared data by using standard UNIX file locking or the DLM.	An application that is not cluster-aware must explicitly use the DLM API, because there is no clusterwide implementation of a standard UNIX synchronization API.

For an application to successfully run multiple instances in a cluster, note the following:

- Interprocess communication must be performed through remote procedure calls (RPCs) or socket connections.
- Access to shared files or data must be synchronized. Use `flock`() system calls or the distributed lock manager (DLM) to synchronize access to shared data.
- Use context-dependent symbolic links (CDSLs) or other means to create separate log files for multiple instances of an application.
- If an application has member-specific configuration requirements, you might need to log on to each member where the application will run and configure the application. For example, if an application's installation script is not cluster-aware (that is, the application does not know that it is being installed in a cluster), you may need to tailor the application's configuration after installing it. For more information, see the configuration documentation for the application.
- If you want to use the default cluster alias to distribute network connections among members, define application ports in the `/etc/clua_services` file. Add the `in_multi` alias attribute so that the cluster alias subsystem distributes connection requests directed to the default cluster alias among all members of the alias. See `clua_services`(4) for more information about defining ports, and see Chapter 3 for information on how to use cluster aliasing for transparent client access to multi-instance applications.

See Chapter 4 for more information about these considerations and other general application migration issues.

1.1.3 Distributed Applications

A distributed application is cluster-aware; that is, it knows it is running in a cluster and typically takes advantage of the performance gains of distributed execution. A distributed application uses the cluster application programming interfaces (APIs) such as DLM and Memory Channel. See Chapter 6 for more information about distributed applications.

2 Using CAA for Single-Instance Application Availability

The cluster application availability (CAA) subsystem tracks the state of members and resources in a cluster (such as networks, applications, tape drives, and media changers). CAA monitors the required resources of application resources in a cluster, and ensures that applications run on members that meet their needs.

This chapter covers the following topics:

- When to use CAA (Section 2.1)
- Creating resource profiles (Section 2.2)
- Writing action scripts (Section 2.3)
- Registering resources (Section 2.4)
- Starting application resources (Section 2.5)
- Relocating application resources (Section 2.6)
- Stopping application resources (Section 2.7)
- Unregistering application resources (Section 2.8)
- Displaying CAA status information (Section 2.9)
- Using the graphical user interfaces to manage CAA (Section 2.10)
- Learning CAA — a tutorial (Section 2.11)
- Creating highly available applications — examples (Section 2.12)

2.1 When to Use CAA

CAA is designed to work with applications that run on one cluster member at a time. If the cluster member on which an application is running fails, or if a particular required resource fails, CAA relocates or "fails over" the application to another member that either has the required resources available or on which the required resource can be started.

Multi-instance applications may find it more useful to use a cluster alias to provide transparent application failover. Typically, multi-instance applications achieve high availability to clients by using the cluster alias as discussed in Chapter 3. However, CAA is often useful for multi-instance applications because it allows for simplified, central management (start and stop) of the applications and restarting on application failure. Using CAA gives you the added value of automatic application startup and shutdown at boot time or at shutdown time, without having to add additional `rc3` scripts.

See the TruCluster Server *Cluster Administration* manual for a general discussion of the differences between the cluster alias subsystem and CAA. Also, see Chapter 3 for examples of how to use the default cluster alias with multi-instance applications for high availability.

2.2 Resource Profiles

A resource profile is a file containing attributes that describe how a resource is started, managed, and monitored by CAA. Profiles designate resource dependencies and determine what happens to an application when it loses access to another resource on which it depends.

There are four resource profile types: `application`, `network`, `tape`, and `changer`.

Some of the attributes that you can specify in a resource profile are:

- Resources that are required by the application (`REQUIRED_RESOURCES`). CAA relocates or stops an application if a required resource becomes unavailable.
- Rules for choosing the member on which to start or restart the application (`PLACEMENT`).
- A list of members, in order of preference, to favor when starting or failing over an application (`HOSTING_MEMBERS`). This list is used if the placement policy (`PLACEMENT`) is `favored` or `restricted`.

Complete lists of profile attributes according to resource type are located in Section 2.2.2, Section 2.2.3, Section 2.2.4, and Section 2.2.5.

All resource profiles are located in the clusterwide directory, `/var/cluster/caa/profile`. The file names of resource profiles take the form *`resource_name`* `.cap`. The CAA commands refer to the resources only by the resource name *`resource_name`*.

Each resource type, `application`, `network`, `tape`, and `changer`, has its own kind of resource profile. The examples and tables in the following sections show each type of resource profile and the entries in that profile.

There are required and optional profile attributes for each type of profile. The optional profile attributes may be left unspecified in the profile. Optional profile attributes that have default values are merged at registration time with the values stored in the template for that type and the generic template. Each resource type has a template file that is stored in

/var/cluster/caa/template, named TYPE_*resource_type*.cap, with default values for attributes. A generic template file for values that are used in all types of resources is stored in /var/cluster/caa/template/TYPE_*generic*.cap.

The examples in the following sections show the syntax of a resource profile. Lines starting with a pound sign (#) are treated as comment lines and are not processed as part of the resource profile. A backslash (\) at the end of a line indicates that the next line is a continuation of the previous line. For a more detailed description of profile syntax, see caa(4).

2.2.1 Creating a Resource Profile

The first step to making an application highly available is to create a resource profile. You can use any of the following methods to do this:

- Use the caa_profile command
- Access SysMan (/usr/sbin/sysman caa)
- Copy an existing resource profile in /var/cluster/caa/profile and edit the copy with emacs, vi, or some other text editor

You can use any of these methods together. For example, you can use the caa_profile command to make a resource profile and then use a text editor to manually edit the profile.

You can find several example profiles in the /var/cluster/caa/examples directory.

After you create a resource profile, you must register it with CAA before a resource can be managed or monitored. See Section 2.4 for a description of how to register an application.

2.2.2 Application Resource Profiles

Table 2–1 lists the application profile attributes. For each attribute, the table indicates whether the attribute is required, its default value, and a description.

Table 2–1: Application Profile Attributes

Attribute	Required	Default	Description
`TYPE`	Yes	None	The type of the resource. The type `application` is for application resources.
`NAME`	Yes	None	The name of the resource. The resource name is a string that contains a combination of letters a-z or A-Z, digits 0-9, or the underscore (_) or period (.). The resource name may not start with a period.
`DESCRIPTION`	No	Name of the resource	A description of the resource.
`FAILURE_THRESHOLD`	No	0	The number of failures detected within `FAILURE_INTERVAL` before CAA marks the resource as unavailable and no longer monitors it. If an application's check script fails this number of times, the application resource is stopped and set offline. Tracking of failures is disabled if the value is zero (0).
`FAILURE_INTERVAL`	No	0	The interval, in seconds, during which CAA applies the failure threshold. Tracking of failures is disabled if the value is zero (0).
`REQUIRED_RESOURCES`	No	None	A white-space separated, ordered list of resource names that this resource depends on. Each resource to be used as a required resource in this profile must be registered with CAA or profile registration will fail. For a more detailed explanation, see Section 2.2.2.1.
`OPTIONAL_RESOURCES`	No	None	A white-space separated, ordered list of optional resources that this resource uses during placement decisions. Up to 58 optional resources can be listed. For a more complete explanation, see Section 2.2.2.3.
`PLACEMENT`	No	`balanced`	The placement policy (`balanced`, `favored`, or `restricted`) specifies how CAA chooses the cluster member on which to start the resource.
`HOSTING_MEMBERS`	No	None	An ordered, white-space separated list of cluster members that can host the resource. This attribute is required only if `PLACEMENT` equals `favored` or `restricted`. This attribute must be empty if `PLACEMENT` equals `balanced`.

Table 2–1: Application Profile Attributes (cont.)

Attribute	Required	Default	Description
`RESTART_ATTEMPTS`	No	1	The number of times CAA will attempt to restart the resource on a single cluster member before attempting to relocate the application. A value of 1 means that CAA will only attempt to restart the application once on a member. A second failure will cause an attempt to relocate the application.
`FAILOVER_DELAY`	No	0	The amount of time, in seconds, CAA will wait before attempting to restart or fail over the resource.
`AUTO_START`	No	0	A flag to indicate whether CAA should automatically start the resource after a cluster reboot, regardless of whether the resource was running prior to the cluster reboot. When set to 0, CAA starts the application resource only if it had been running before the reboot. When set to 1, CAA always starts the application after a reboot.
`ACTION_SCRIPT`	Yes	None	The resource-specific script for starting, stopping, and checking a resource. You may specify a full path for the action script file; otherwise, the path `/var/cluster/caa/script` is assumed.
`ACTIVE_PLACEMENT`	No	0	When set to 1, CAA will reevaluate the placement of an application on addition or restart of a cluster member.
`SCRIPT_TIMEOUT`	No	60	The maximum time, in seconds, that an action script may take to complete execution before an error is returned.
`CHECK_INTERVAL`	No	60	The time interval, in seconds, between repeated executions of the check entry point of the resource's action script.

The following example creates an application resource with CAA using `caa_profile`:

```
# /usr/sbin/caa_profile -create clock -t application -B /usr/bin/X11/xclock \
-d "Clock Application" -r network1 -l application2 \
-a clock.scr -o ci=5,ft=2,fi=12,ra=2
```

The contents of the resource profile file that was created by the previous example are as follows:

```
NAME=clock
TYPE=application
ACTION_SCRIPT=clock.scr
ACTIVE_PLACEMENT=0
AUTO_START=0
CHECK_INTERVAL=5
DESCRIPTION=Clock Application
FAILOVER_DELAY=0
FAILURE_INTERVAL=12
FAILURE_THRESHOLD=2
HOSTING_MEMBERS=
OPTIONAL_RESOURCES=application2
PLACEMENT=balanced
REQUIRED_RESOURCES=network1
RESTART_ATTEMPTS=2
SCRIPT_TIMEOUT=60
```

For more information on the application resource profile syntax, see `caa_profile`(8) and `caa`(4).

2.2.2.1 Required Resources

CAA uses the required resources list, in conjunction with the placement policy and hosting members list, to determine which members are eligible to host the application resource. Required resources must be `ONLINE` on any member on which the application is running or started. Only application resources can have required resources, but any type of resource can be defined as a required resource for an application resource.

A failure of a required resource on the hosting member causes CAA to initiate failover of the application or to attempt to restart it on the current member if `RESTART_ATTEMPTS` is not 0. This can cause CAA to fail the application resource over to another member, which provides the required resources, or to stop the application if there is no suitable member. In the latter case, CAA continues to monitor the required resources and restarts the application when the resource is again available on a suitable cluster member.

Required resources lists can also be useful to start, stop, and relocate a group of interdependent application resources when the `caa_start`, `caa_stop`, or `caa_relocate` commands are run with the `-f` option.

2.2.2.2 Application Resource Placement Policies

The placement policy specifies how CAA selects a cluster member on which to start a resource, and on which to relocate it after a failure.

Note

Only cluster members that have all the required resources available (as listed in an application resource's profile) are eligible to be considered in any placement decision involving that application.

The following placement policies are supported:

- `balanced` — CAA favors starting or restarting the application resource on the member currently running the fewest application resources. Placement due to optional resources is considered first. See Section 2.2.2.3. Next, the host with the fewest application resources running is chosen. If no cluster member is favored by these criteria, any available member is chosen.
- `favored` — CAA refers to the list of members in the `HOSTING_MEMBERS` attribute of the resource profile. Only cluster members that are in this list and satisfy the required resources are eligible for placement consideration. Placement due to optional resources is considered first. See Section 2.2.2.3. If no member can be chosen based on optional resources, the order of the hosting members decides which member will run the application resource. If none of the members in the hosting member list are available, CAA favors placing the application resource on any available member. This member may or may not be included in the `HOSTING_MEMBERS` list.
- `restricted` — Like `favored` except that, if none of the members on the hosting list are available, CAA will not start or restart the application resource. A `restricted` placement policy ensures that the resource will never run on a member that is not on the list, even if you manually relocate it to that member.

You must specify hosting members in the `HOSTING_MEMBERS` attribute to use a `favored` or `restricted` placement policy. You must not specify hosting members in the `HOSTING_MEMBERS` attribute with a `balanced` placement policy.

If `ACTIVE_PLACEMENT` is set to 1, the placement of the application resource is reevaluated whenever a cluster member is either added to the cluster or it restarts. This allows applications to be relocated to a preferred member of a cluster after the member recovers from a failure.

2.2.2.3 Optional Resources in Placement Decisions

Optional resources are used to choose a hosting member based on the number of optional resources that are in the `ONLINE` state on each hosting member. If each member has an equal number of optional resources in the `ONLINE` state, CAA considers the order the optional resources as follows.

CAA compares the state of the optional resources on each member starting at the first resource and proceeding successively through the list. For each consecutive resource in the list, if the resource is ONLINE on one member, any member that does not have the resource ONLINE is removed from consideration. Each resource on the list is evaluated in this manner until only one member is available to host the resource. The maximum number of optional resources is 58.

If this algorithm results in multiple favored members, the application is placed on one of these members chosen according to its placement policy.

2.2.3 Network Resource Profiles

Table 2–2 describes the network profile attributes. For each attribute, the table indicates whether the attribute is required, its default value, and a description.

Table 2–2: Network Profile Attributes

Attributes	Required	Default	Description
`TYPE`	Yes	None	The type of the resource. The type `network` is for network resources.
`NAME`	Yes	None	The name of the resource. The resource name is a string that contains a combination of letters a-z or A-Z, digits 0-9, or the underscore (_) or period (.). The resource name may not start with a period.
`DESCRIPTION`	No	None	A description of the resource.
`SUBNET`	Yes	None	The subnet address of the network resource in *nnn.nnn.nnn.nnn* format (for example, 16.140.112.0). The SUBNET value is the bitwise AND of the IP address and the netmask. If you consider an IP address of 16.69.225.12 and a netmask of 255.255.255.0, then the subnet will be 16.69.225.0.
`FAILURE_THRESHOLD`	No	0	The number of failures detected within `FAILURE_INTERVAL` before CAA marks the target value OFFLINE and no longer monitors it. Tracking of failures is disabled if the value is zero (0).
`FAILURE_INTERVAL`	No	0	The interval, in seconds, during which CAA applies the failure threshold. Tracking of failures is disabled if the value is zero (0).

The following example creates a network resource profile:

```
# /usr/sbin/caa_profile -create network1 -t network -s "16.69.244.0" \
-d "Network1"
```

The contents of the profile in file /var/cluster/caa/profile/network1.cap created by the preceding command are as follows:

```
NAME=network1
TYPE=network
DESCRIPTION=Network1
FAILURE_INTERVAL=0
FAILURE_THRESHOLD=0
SUBNET=16.69.244.0
```

For more information on the network resource profile syntax, see caa_profile(8) and caa(4).

Through routing, all members in a cluster can indirectly access any network that is attached to any member. Nevertheless, an application may require the improved performance that comes by running on a member with direct connectivity to a network. For that reason, an application resource may define an optional or required dependency on a network resource. CAA optimizes the placement of that application resource based on the location of the network resource.

When you make a network resource an optional resource (OPTIONAL_RESOURCES) for an application, the application may start on a member that is directly connected to the subnet, depending on the required resources, placement policy, and cluster state. If the network adapter fails, the application may still access the subnet remotely through routing.

If you specify a network resource as a required resource (REQUIRED_RESOURCES) and the network adapter fails, CAA relocates or stops the application. If the network fails on all eligible hosting members, CAA will stop the application.

2.2.4 Tape Resource Profiles

Table 2–3 describes the tape profile attributes. For each attribute, the table indicates whether the attribute is required, its default value, and a description.

Table 2–3: Tape Profile Attributes

Attributes	Required	Default	Description
TYPE	Yes	None	The type of the resource. The type tape is for tape resources.
NAME	Yes	None	The name of the resource. The resource name is a string that contains a combination of letters a-z or A-Z, digits 0-9, or the underscore (_) or period (.). The resource name may not start with a period.
DESCRIPTION	No	None	A description of the resource.

Table 2–3: Tape Profile Attributes (cont.)

Attributes	Required	Default	Description
`DEVICE_NAME`	Yes	None	The device name of the tape resource. Use the full path to the device special file (for example, `/dev/tape/tape1`).
`FAILURE_THRESHOLD`	No	0	The number of failures detected within `FAILURE_INTERVAL` before CAA marks the target value OFFLINE and no longer monitors it. Tracking of failures is disabled if the value is zero (0).
`FAILURE_INTERVAL`	No	0	The interval, in seconds, during which CAA applies the failure threshold. Tracking of failures is disabled if the value is zero (0).

Through the device request dispatcher, all cluster members can indirectly access any tape device that is attached to any cluster member. Nevertheless, an application may require the improved performance that comes from running on a member with direct connectivity to the tape. For that reason, an application resource may define an optional or required dependency on a tape resource. CAA optimizes the placement of that application based on the location of the tape resource.

The following example creates a tape resource profile. After a tape resource has been defined in a resource profile, an application resource profile can designate it as a required or optional resource.

```
# /usr/sbin/caa_profile -create tape1 -t tape -n /dev/tape/tape1 -d "Tape Drive"
```

The contents of the profile that was created in the file `/var/cluster/caa/profile/tape1.cap` by the preceding command are as follows:

```
NAME=tape1
TYPE=tape
DESCRIPTION=Tape Drive
DEVICE_NAME=/dev/tape/tape1
FAILURE_INTERVAL=0
FAILURE_THRESHOLD=0
```

2.2.5 Media Changer Resource Profiles

Table 2–4 describes the media changer profile attributes. For each attribute, the table indicates whether the attribute is required, its default value, and a description.

Table 2–4: Media Changer Attributes

Attributes	Required	Default	Description
`TYPE`	Yes	None	The type of the resource. The type `changer` is for media changer resources.
`NAME`	Yes	None	The name of the resource. The resource name is a string that contains a combination of letters a-z or A-Z, digits 0-9, or the underscore (_) or period (.). The resource name may not start with a period.
`DESCRIPTION`	No	None	A description of the resource.
`DEVICE_NAME`	Yes	None	The device name of the media changer resource. Use the full path to the device special file (for example, `/dev/changer/mc1`).
`FAILURE_THRESHOLD`	No	0	The number of failures detected within `FAILURE_INTERVAL` before CAA marks the target value OFFLINE and no longer monitors it. Tracking of failures is disabled if the value is zero (0).
`FAILURE_INTERVAL`	No	0	The interval, in seconds, during which CAA applies the failure threshold. Tracking of failures is disabled if the value is zero (0).

Through the device request dispatcher, all cluster members can indirectly access any media changer that is attached to any member. Nevertheless, an application may require the improved performance that comes from running on a member with direct connectivity to the media changer. For that reason, an application resource may define an optional or required dependency on a media changer resource. CAA optimizes the placement of that application based on the location of the media changer resource.

The following example creates a media changer resource profile. After a media changer resource has been defined in a resource profile, an application resource profile can designate it as a dependency.

```
# /usr/sbin/caa_profile -create mchanger1 -t changer -n /dev/changer/mc1 \
-d "Media Changer Drive"
```

The contents of the profile that was created in the file `/var/cluster/caa/profile/mchanger1.cap` by the preceding command are as follows:

```
NAME=mchanger1
TYPE=changer
DESCRIPTION=Media Changer Drive
DEVICE_NAME=/dev/changer/mc1
FAILURE_INTERVAL=0
FAILURE_THRESHOLD=0
```

2.3 Writing Action Scripts

Action scripts are necessary for application resources to start, stop, and relocate an application that is managed and monitored by CAA.

You use action scripts to specify the following:

- How to start an application.

  CAA calls the start entry point of the action script to start or restart the application resource. The start entry point executes all commands that are necessary to start the application and must return 0 (zero) for success and a nonzero value for failure.

- How to stop an application and what cleanup occurs before the application is failed over.

  CAA calls the stop entry point of the action script to stop a running application resource. It is not called when stopping an application resource in state `UNKNOWN` (see `caa_stop`(8) for details). The stop entry point executes all commands necessary to stop the application and must return 0 (zero) for success and a nonzero value for failure. If the stop script determines that there is nothing to stop, it should return 0.

- How to determine whether an application is still running.

  CAA calls the `check` entry point of the action script to verify that an application resource is running. The check entry point executes every `CHECK_INTERVAL` seconds and must return 0 (zero) for success and a nonzero value for failure.

Action scripts are located by default in the clusterwide `/var/cluster/caa/script` directory. The file names of action scripts take the form *name*`.scr`.

The easiest way to create an action script is to have the `caa_profile` command automatically create one for you when you create the resource profile. Do this by using the `-B` option. For example:

```
# caa_profile -create resource_name -t application -B application_path
```

Use the `-B` option in the `caa_profile` command to specify the full pathname of an application executable; for example, `/usr/bin/X11/xterm`. When you use the `-B` option, the `caa_profile` command creates an action script named `/var/cluster/caa/script/`*resource_name*`.scr`. To specify a different action script name, use the `-a` option.

Depending on the application, you might need to edit the action script to correctly set up the environment for the application. For example, for an X application like `xclock`, you need to set the `DISPLAY` environment variable on the command line in the action script as appropriate for the current shell. It might look something like:

```
DISPLAY=`hostname`:0
export DISPLAY
```

Because an action script is required for an application resource, when you use the `caa_profile -create` command to create an application resource profile, one of the following conditions must be true:

- You must specify the `caa_profile` option `-B` *`application_executable_pathname`*, so that an action script is automatically created. You may also specify the name of the action script that is created with the `-a` option.
- You must have already created an executable action script in the default directory, `/var/cluster/caa/script/`. The root of the script name must be the same as the name of the resource you create.

  For example, if the action script is named `/var/cluster/caa/script/up-app-1.scr`, then the resource name must be `up-app-1`. Therefore, if you use the `caa_profile` command to create the resource profile, the command line starts as follows:

  ```
  # caa_profile -create up-app-1 -t application
  ```
- You must have already created an executable action script, and you must use the `caa_profile` option `-a` *`action_script_pathname`* to inform CAA where to find the action script. For example:

  ```
  -a /usr/users/smith/caa/scripts/app.scr
  ```

Caution

Make sure that this script is writable only by root, for security reasons.

2.3.1 Guidelines for Writing Application Resource Action Scripts

When writing an action script for an application resource, note the following:

- CAA relies on the exit code from the action script to set the application state to `ONLINE` or `OFFLINE`. Each entry point in the action script must return an exit code of 0 to reflect success or a nonzero exit code to specify failure.
- CAA sets the application state to `UNKNOWN` if your action script's stop entry point fails to exit within the number of seconds in the `SCRIPT_TIMEOUT` value, or returns with a nonzero value. This may happen during a start attempt, a relocation, or a stop attempt. Be sure that the action script stop entry point exits with a 0 value if the application is successfully stopped or if it is not running.

- When a daemon is started, it usually starts instantly as a background process. For an application that does not put itself into the background immediately upon startup, start the application in the background by adding an ampersand (&) to the end of the line where the application is executed. If such an application is used in this way, it will always return success on a start attempt. This means that the default scripts will have no way of detecting failure due to a trivial reason, such as a misspelled command path. When using such commands, we recommend that you execute the commands used in the script interactively to rule out syntax and other trivial errors before using the script with CAA.
- CAA disassociates from `stdin`, `stdout`, and `stderr` before starting an application resource. You must identify an appropriate standard output stream and standard error stream for your application, and use them in the action script when invoking the application. For example:

  ```
  /usr/bin/my_application 1>> /usr/tmp/app.log 2>&1
  ```

For any X-windows applications that you may be running under CAA, you must also consider the following:

- For a graphical application that is served by the cluster and monitored by CAA, you must set the `DISPLAY` environment variable of the client system in the action script. For example:

  ```
  DISPLAY=everest:0.0
  /usr/bin/my_application &
  ```

- On the client system, add the default cluster alias to the list of allowed X server connections. For example:

  ```
  everest#> xhost +my_cluster
  ```

- CAA scripts generated by `caa_profile` or SysMan do not set the `PATH` environment variable. When the scripts are executed, the `PATH` is set to a default value `/sbin:/usr/sbin:/usr/bin`. Therefore, you must explicitly specify most path names that are used in scripts, or you must modify the resulting scripts to explicitly set the `PATH`. Action scripts that were automatically generated with previous releases may have a `PATH` that includes the current directory (`.`). Because this situation may be a potential security issue, modify these scripts to remove the current directory from the path.

2.3.2 Action Script Examples

The action script template is located in `/var/cluster/caa/template/template.scr`. It is the basis for action scripts that are created by the `caa_profile` command, and it is a good example of the elements of an action script.

The following action scripts for application resources can be used as examples and are found in the /var/cluster/caa/script directory:

- cluster_lockd.scr
- dhcp.scr
- named.scr
- autofs.scr

The scripts shown in Section 2.12 are also good examples of action scripts. These example scripts and others can be found in the /var/cluster/caa/examples directory. There are examples of several applications that are commonly administered using CAA. The script sysres_templ.scr that is located in this directory is an example script that contains extra system performance related code that can be used to examine the system load, swap space usage, and disk space available. If you want to use these features in your scripts, set the values for variables that are associated with these features appropriately for your system.

2.4 Registering Resources (caa_register)

Each resource must have a profile. Each resource must be registered with CAA. Use the caa_register command to register your resources. For example, to register the clock application, enter the following command:

```
# /usr/sbin/caa_register clock
```

All resources must be registered in order for them to be managed by CAA. After a resource is registered, the information in the profile is stored in the database /var/cluster/caa/registry/caa.reg. If the profile is modified, you must update the database with caa_register -u.

See caa_register(8) for more information.

2.5 Starting Application Resources (caa_start)

To start an application that is registered with CAA, use the caa_start command. The name of the application resource may or may not be the same as the name of the application. For example:

```
# /usr/sbin/caa_start clock
```

The following text is an example of the command output:

```
Attempting to start 'clock' on member 'polishham'
Start of 'clock' on member 'polishham' succeeded.
```

The application is now running on the system named polishham.

The command will wait up to the `SCRIPT_TIMEOUT` value to receive notification of success or failure from the action script each time that the action script is called.

Application resources can be started and non-application resources can be restarted if they have stopped due to exceeding their failure threshold values. (See the TruCluster Server *Cluster Administration* manual for more information on restarting non-application resources.) You must register a resource (`caa_register`) before you can start it.

Note

Always use `caa_start` and `caa_stop`, or the equivalent SysMan feature, to start and stop resources. Do not start or stop the applications manually at the command line or by executing the action scripts.

See `caa_start`(8) for more information.

When you try to start a resource that has required resources that are `ONLINE` on another cluster member, the start will fail. All required resources must either be `OFFLINE` or `ONLINE` on the member where the resource will be started.

If you use the command `caa_start -f` *`resource_name`* on a resource that has required resources that are `OFFLINE`, the resource starts and all required resources that are not currently `ONLINE` start as well.

Executing the `caa_start` command on an application resource actually only sets the target to `ONLINE`. CAA attempts to change the state to match the target and attempts to start the application by running the action script start entry point. When an application is running, both the target state and current state are `ONLINE`.

The TruCluster Server *Cluster Administration* manual has a more detailed description of how target and state fields describe resources.

2.6 Relocating Application Resources

Use the `caa_relocate` command to relocate application resources. You cannot relocate network, tape, or changer resources.

To relocate an application resource to an available cluster member, or to the cluster member specified, use the `caa_relocate` command. For example, to relocate the `clock` application to member `provolone`, enter the following command:

```
# /usr/sbin/caa_relocate clock -c provolone
```

The following text is an example of the command output:

```
Attempting to stop `clock` on member `polishham`
Stop of `clock` on member `polishham` succeeded.
Attempting to start `clock` on member `provolone`
Start of `clock` on member `provolone` succeeded.
```

To relocate the `clock` application to another member using the placement policy that is defined in the application resource's profile, enter the following command:

```
# /usr/sbin/caa_relocate clock
```

The following text is an example of the command output:

```
Attempting to stop `clock` on member `pepicelli`
Stop of `clock` on member `pepicelli` succeeded.
Attempting to start `clock` on member `polishham`
Start of `clock` on member `polishham` succeeded.
```

The following text is an example of the command output if the application cannot be relocated successfully due to a script returning a nonzero value or a script timeout:

```
Attempting to stop `clock` on member `pepicelli`
Stop of `clock` on member `pepicelli` succeeded.
Attempting to start `clock` on member `provolone`
Start of `clock` on member `provolone` failed.
No more members to consider
Attempting to restart `clock` on member `pepicelli`
Could not relocate resource clock.
```

Each time that the action script is called, the `caa_relocate` command will wait up to the `SCRIPT_TIMEOUT` value to receive notification of success or failure from the action script.

A relocate attempt will fail if:

- The resource has required resources that are `ONLINE`
- Resources that require the specified resource are `ONLINE`

If you use the `caa_relocate -f` *`resource_name`* command on a resource that has required resources that are `ONLINE`, or has resources that require it that are `ONLINE`, the resource is relocated and all resources that require it and are `ONLINE` are relocated. All resources that are required by the resource specified are relocated or started regardless of their state.

See `caa_relocate`(8) for more information.

2.7 Stopping Application Resources (caa_stop)

To stop applications that are running in a cluster environment, use the `caa_stop` command. Immediately after the `caa_stop` command is executed, the target is set to `OFFLINE`. Because CAA always attempts to match a resource's state to its target, the CAA subsystem stops the application. Only application resources can be stopped. Network, tape, and media changer resources cannot be stopped.

In the following example, the `clock` application resource is stopped:

```
# /usr/sbin/caa_stop clock
```

The following text is an example of the command output:

```
Attempting to stop `clock` on member `polishham`
Stop of `clock` on member `polishham` succeeded.
```

When you try to stop a resource that has resources that require it that are `ONLINE`, the stop will fail.

If you use the command `caa_stop -f` *`resource_name`* on a resource that has resources that require it and are `ONLINE`, the resource is stopped and all resources that require it and are `ONLINE` are stopped.

See `caa_stop`(8) for more information.

2.8 Unregistering Application Resources

To unregister an application resource, use the `caa_unregister` command.

In the following example, the `clock` application is unregistered:

```
# /usr/sbin/caa_unregister clock
```

You cannot unregister an application that is `ONLINE` or required by another resource.

See `caa_unregister`(8) for more information.

2.9 CAA Status Information (caa_stat)

This section describes how to display status information on CAA resources.

To display status information on resources on cluster members, use the `caa_stat` command.

In the following example the status information for the `clock` resource is displayed:

```
# /usr/bin/caa_stat clock

NAME=clock
TYPE=application
TARGET=ONLINE
STATE=ONLINE on provolone
```

To view information on all resources, enter the following command:

```
# /usr/bin/caa_stat

NAME=clock
TYPE=application
TARGET=ONLINE
STATE=ONLINE on provolone
NAME=dhcp
TYPE=application
TARGET=ONLINE
STATE=ONLINE on polishham

NAME=named
TYPE=application
TARGET=ONLINE
STATE=ONLINE on polishham

NAME=network1
TYPE=network
TARGET=ONLINE on provolone
TARGET=ONLINE on polishham
STATE=ONLINE on provolone
STATE=ONLINE on polishham
```

To view information on all resources in a tabular form, enter the following command:

```
# /usr/bin/caa_stat -t

Name             Type           Target      State     Host
------------------------------------------------------------------
cluster_lockd    application    ONLINE      ONLINE    provolone
dhcp             application    OFFLINE     OFFLINE
engine_server    application    OFFLINE     OFFLINE
network1         network        ONLINE      ONLINE    provolone
network1         network        ONLINE      ONLINE    polishham
```

To find out how many times a resource has been restarted or has failed within the resource failure interval, the maximum number of times that a resource can be restarted or fail, and the target state of the application, as well as normal status information, enter the following command:

```
# /usr/bin/caa_stat -v

NAME=cluster_lockd
TYPE=application
RESTART_ATTEMPTS=30
RESTART_COUNT=0
FAILURE_THRESHOLD=0
FAILURE_COUNT=0
TARGET=ONLINE
STATE=ONLINE on provolone
```

```
NAME=dhcp
TYPE=application
RESTART_ATTEMPTS=1
RESTART_COUNT=0
FAILURE_THRESHOLD=3
FAILURE_COUNT=1
TARGET=ONLINE
STATE=OFFLINE

NAME=network1
TYPE=network
FAILURE_THRESHOLD=0
FAILURE_COUNT=0 on polishham
FAILURE_COUNT=0 on polishham
TARGET=ONLINE on provolone
TARGET=ONLINE on polishham
STATE=ONLINE on provolone
STATE=OFFLINE on polishham
```

To view verbose content in a tabular form, enter the following command:

```
# /usr/bin/caa_stat -v -t

Name           Type          R/RA   F/FT   Target     State     Host
-------------------------------------------------------------------------
cluster_lockd  application   0/30   0/0    ONLINE    ONLINE
provolone
dhcp           application   0/1    0/0    OFFLINE   OFFLINE
named          application   0/1    0/0    OFFLINE   OFFLINE
network1       network              0/5    ONLINE    ONLINE
provolone
network1       network              0/5    ONLINE    ONLINE
polishham
```

To view the profile information that is stored in the database, enter the following command:

```
# /usr/bin/caa_stat -p

NAME=cluster_lockd
TYPE=application
ACTION_SCRIPT=cluster_lockd.scr
ACTIVE_PLACEMENT=0
AUTO_START=1
CHECK_INTERVAL=5
DESCRIPTION=Cluster lockd/statd
FAILOVER_DELAY=30
FAILURE_INTERVAL=60
FAILURE_THRESHOLD=1
HOSTING_MEMBERS=
OPTIONAL_RESOURCES=
PLACEMENT=balanced
REQUIRED_RESOURCES=
RESTART_ATTEMPTS=2
SCRIPT_TIMEOUT=60

NAME=ln0
TYPE=network
DESCRIPTION=
FAILURE_INTERVAL=0
FAILURE_THRESHOLD=0
SUBNET=16.69.224.0
```

See the TruCluster Server *Cluster Administration* manual and `caa_stat`(1) for more information.

2.10 Graphical User Interfaces

The following sections discuss how to use the SysMan and SysMan Station graphical user interfaces (GUIs) to manage CAA.

2.10.1 Using SysMan Menu to Manage CAA

You can start the SysMan Menu from the command line with `/usr/sbin/sysman`. To access the CAA tools, select the Cluster Application Availablility (CAA) Management task under the TruCluster Specific branch.

```
.
.
.
+ TruCluster Specific
     |Cluster Application Availability (CAA) Management
```

To start only the Cluster Application Availability (CAA) Management task, use `/usr/sbin/sysman caa`.

See the Tru64 UNIX *System Administration* manual for more information on accessing SysMan Menu.

Using the SysMan Menu you can:

- Manage resource profiles
- Monitor CAA resources
- Register resources
- Start resources
- Relocate resources
- Stop resources
- Unregister resources

The CAA GUI provides graphical assistance for cluster administration based on event reports from the Event Manager (EVM) and CAA daemon.

2.10.2 Using SysMan Station to Manage and Monitor CAA

SysMan Station gives users a comprehensive graphical view of their cluster. SysMan Station lets you view the current status of CAA resources on a whole cluster, and manage those resources. SysMan Station also contains the management tool SysMan Menu to manage individual CAA resources. See the Tru64 UNIX *System Administration* manual for further information on accessing the SysMan Station.

To access the CAA SysMan Menu tools in the SysMan Station, follow these steps:

1. Select one of the views under `Views`, for example, `CAA_Applications_(active)` or `CAA_Applications_(all)`.
2. Select the cluster name under the `Views` window, for example, `CAA_Applications_(active) View` or `CAA_Applications_(all) View`.
3. From the Tools menu, select SysMan Menu. The Cluster Application Availablility (CAA) Management task is located under the TruCluster Specific branch.

For more detailed descriptions of the SysMan Menu and SysMan Station, see the online help or the Tru64 UNIX *System Administration* manual.

2.11 CAA Tutorial

This CAA tutorial helps you with the basic instructions necessary to quickly make an application highly available using CAA. For in-depth details on specific commands, you must read all the necessary documentation that pertains to the CAA commands.

- Preconditions (Section 2.11.1)
- Miscellaneous Setup (Section 2.11.2)
- Example of an action script for `dtcalc` (Section 2.11.3)
- Step 1: Creating the application resource profile (Section 2.11.4)
- Step 2: Validating the application resource profile (Section 2.11.5)
- Step 3: Registering the application (Section 2.11.6)
- Step 4: Starting the application (Section 2.11.7)
- Step 5: Relocating the application (Section 2.11.8)
- Step 6: Stopping the application (Section 2.11.9)
- Step 7: Unregistering the application (Section 2.11.10)

2.11.1 Preconditions

You must have root access to a two-member TruCluster Server cluster.

In this tutorial you use CAA to make the Tru64 UNIX application `dtcalc` highly available. Make sure that the test application `/usr/dt/bin/dtcalc` exists.

An X-based application is used only for demonstrative purposes in this example. The X-based application is used to provide immediate viewing of the results of starts, stops, and relocation. You will most likely not find a use for highly available applications of this sort.

2.11.2 Miscellaneous Setup

If you are making an application with a graphical interface highly available using CAA, make sure that you set your `DISPLAY` variable correctly in the *ActionScript*`.scr` file. Modify the `DISPLAY` variable, and copy the file *ActionScript*`.scr` into the scripts directory `/var/cluster/caa/script`.

Verify that the host on which you want to display the application is able to display X applications from the cluster. If you need to modify the access, execute a command similar to following command on the machine that is displaying the application:

```
# xhost + clustername
```

If you are not sure of the actual names of each member, look in the `/etc/hosts` file on your system to get the names of each member. You also can use the `clu_get_info` command to get information on each cluster member, including the host names.

The following command is an example showing the results of the `clu_get_info` command:

```
# clu_get_info


     Cluster information for cluster deli


                Number of members configured in this cluster = 3
                memberid for this member = 3
                Quorum disk = dsk10h
                Quorum disk votes = 1

                   Information on each cluster member

                Cluster memberid = 1
                Hostname = polishham.zk4.com
```

```
Cluster interconnect IP name = polishham=mc0
Member state = UP

Cluster memberid = 2
Hostname = provolone.zk4.com
Cluster interconnect IP name = provolone = mc0
Member state = UP

Cluster memberid = 3
Hostname = pepicelli.zk4.com
Cluster interconnect IP name = pepicelli=mc0
Member state = UP
```

2.11.3 Example of an Action Script for dtcalc

The following example an action script that you can use for the `dtcalc` tutorial, or you can use the more complex action script that is created by the `caa_profile` command:

```
#!/usr/bin/ksh -p
#
# This action script will be used to launch dtcalc.
#
export DISPLAY=`hostname`:0
PATH=/sbin:/usr/sbin:/usr/bin
export PATH
CAATMPDIR=/tmp

CMDPATH=/usr/dt/bin

APPLICATION=${CMDPATH}/dtcalc

CMD=`basename $APPLICATION`

case $1 in

 'start') 1
        if [ -f $APPLICATION ]; then
                $APPLICATION &           exit 0   else
                echo "Found exit1" >/dev/console
                exit 1
        fi
        ;;
 'stop') 2
       PIDLIST=`ps ax | grep $APPLICATION | grep -v 'caa_' \
                       | grep -v 'grep' | awk '{print $1}''
       if [ -n "$PIDLIST" ]; then
               kill -9 $PIDLIST
               exit 0
       fi
       exit 0
       ;;
'check') 3
       PIDLIST='ps ax | grep $CMDPATH | grep -v 'grep' | awk '{print
               $1}''
       if [ -z "$PIDLIST" ]; then
               PIDLIST='ps ax | grep $CMD | grep -v 'grep'
                       | awk '{print $1}''
       fi
```

```
                if [-n "$PIDLIST" ]; then
                        exit 0
                else
                 echo "Error: CAA could not find $CMD." >/dev/console
                 exit 1
                fi
        ;;
esac
```

[1] The start entry point is executed when an application is started.

[2] The stop entry point is executed when an application is stopped.

[3] The check entry point is executed every `CHECK_INTERVAL` seconds.

2.11.4 Step 1: Creating the Application Resource Profile

Create the resource profile for `dtcalc` with the following options to the `caa_profile` command:

```
# /usr/sbin/caa_profile -create dtcalc -t application -B /usr/dt/bin/dtcalc \
-d "dtcalc application" -p balanced
```

When you examine the `dtcalc.cap` file that is located in `/var/cluster/caa/profile/`, you will see the following:

```
# cat dtcalc.cap

NAME=dtcalc
TYPE=application
ACTION_SCRIPT=dtcalc.scr
ACTIVE_PLACEMENT=0
AUTO_START=0
CHECK_INTERVAL=60
DESCRIPTION=dtcalc application
FAILOVER_DELAY=0
FAILURE_INTERVAL=0
FAILURE_THRESHOLD=0
HOSTING_MEMBERS=
OPTIONAL_RESOURCES=
PLACEMENT=balanced
REQUIRED_RESOURCES=
RESTART_ATTEMPTS=1
SCRIPT_TIMEOUT=60
```

2.11.5 Step 2: Validating the Application Resource Profile

To validate the resource profile syntax, enter the following command:

```
# caa_profile -validate dtcalc
```

If there are syntax errors in the profile, `caa_profile` displays messages indicating that the profile did not pass validation.

2.11.6 Step 3: Registering the Application

To register the application, enter the following command:

```
# /usr/sbin/caa_register dtcalc
```

If the profile cannot be registered, messages are displayed explaining why.

To check that the application is registered, enter the following command:

```
# /usr/bin/caa_stat dtcalc

NAME=dtcalc
TYPE=application
TARGET=OFFLINE
STATE=OFFLINE
```

2.11.7 Step 4: Starting the Application

To start the application, enter the following command:

```
# /usr/bin/caa_start dtcalc
```

The following messages are displayed:

```
Attempting to start ‘dtcalc‘ on member ‘provolone‘
Start of ‘dtcalc‘ on member ‘provolone‘ succeeded.
```

You can execute the `/usr/bin/caa_stat dtcalc` command to check that the `dtcalc` action script start entry point executed successfully and `dtcalc` is started. For example:

```
# /usr/bin/caa_stat dtcalc

NAME=dtcalc
TYPE=application
TARGET=ONLINE
STATE=ONLINE on provolone
```

If the `DISPLAY` variable is set correctly in the script, `dtcalc` appears on your display.

2.11.8 Step 5: Relocating the Application

To relocate the application, enter the following command:

```
# /usr/bin/caa_relocate dtcalc -c polishham
```

Execute the command `/usr/bin/caa_stat dtcalc` to verify that `dtcalc` started successfully. An example follows:

```
# /usr/bin/caa_stat dtcalc

NAME=dtcalc
```

```
TYPE=application
TARGET=ONLINE
STATE=ONLINE on polishham
```

The cluster member is listed in the `STATE` attribute.

2.11.9 Step 6: Stopping the Application

To stop the application, enter the following command:

```
# /usr/bin/caa_stop dtcalc
```

The following information is displayed:

```
Attempting to stop `dtcalc` on member `provolone`
Stop of `dtcalc` on member `provolone` succeeded.
```

You can execute the `/usr/bin/caa_stat dtcalc` command to verify that the stop entry point of the `dtcalc` action script executed successfully and `dtcalc` is stopped. For example:

```
# /usr/bin/caa_stat dtcalc

NAME=dtcalc
TYPE=application
TARGET=OFFLINE
STATE=OFFLINE
```

2.11.10 Step 7: Unregistering the Application

To unregister the application, enter the following command:

```
# /usr/sbin/caa_unregister dtcalc
```

2.12 Example Applications Managed by CAA

The following sections contain examples of highly available single-instance applications that are managed by CAA. You can follow the examples to set up the specific applications listed or review them as good examples of the process of setting up any application for use with CAA.

2.12.1 Creating a Single-Instance, Highly Available Netscape FastTrack Server Using CAA

To create a highly available Netscape FastTrack server in a TruCluster Server environment, follow these steps:

1. If the Netscape FastTrack server is not installed, unpack and install the Netscape FastTrack kit.
2. Configure the Netscape server to use the default cluster alias name (`tcralias2` in this example). If you do not want all cluster members

to handle requests for the service, create a new cluster alias. See the TruCluster Server *Cluster Administration* manual for information about creating additional cluster aliases.

3. Add the following entry to the `/etc/clua_services` file:

   ```
   http            80/tcp          in_single
   ```

   Setting the `in_single` attribute means that the cluster alias subsystem will distribute connection requests that are directed to the default cluster alias to one member of the alias. If that member becomes unavailable, the cluster alias subsystem will select another member of the default cluster alias to receive all requests.

4. To reload service definitions, enter the following command on all members:

   ```
   # cluamgr -f
   ```

5. Netscape has separate start and stop scripts. Combine the start and stop scripts into one script that CAA will use to start, stop, and verify the application. The following example shows a combined script:

   ```
   #!/usr/bin/ksh -p
   #
   # TruCluster V5 sample CAA script for Netscape Webserver
   #
   # Some initial setup
   #
   # 8<--------------8<----------- Start Custom variables 8<----------8<----------
   #
   svcName="netscape"                      # Servicename
   CAA_ADMIN="root"                        # Account to receive CAA mail
   CAALOGDIR="/var/cluster/caa/log"        # Directory for logfiles
   ACTION=$1                               # Action (either start or stop)
   LOG="${CAALOGDIR}/${ACTION}_${svcName}.$$" # Destination for script output
   #LOG="/dev/console"
   #
   # Application specific stuff
   #
   PROBE_PROCS="ns-admin ns-httpd"         # Processes to probe
   START_APPCMD="/cludemo/netscape/start-admin" # Application startup cmd
   START_APPCMD2="/cludemo/netscape/https-pingpong/start" # Application startup cmd
   STOP_APPCMD="/cludemo/netscape/https-pingpong/stop"  # Application stop cmd
   STOP_APPCMD2="/cludemo/netscape/stop-admin"   # Application stop cmd
   APPDIR="/cludemo/netscape"              # Application home directory
   ADVFSDIRS=" "                           # Application directories to
   #
   FUSER="/usr/sbin/fuser"                 # Command to use for closing
   EVMPOST="/usr/bin/evmpost -p 650 -a"    # EVM command to post events
   #
   export START_APPCMD START_APPCMD2 STOP_APPCMD STOP_APPCMD2 APPDIR
   export ADVFSDIRS PROBE_PROCS
   #
     :
     :
   # Main section
   #
   # Start section
   #
   case $1 in
   ```

```
'start')
        echo ""                                                 >>  ${LOG}
        echo ""Start action script for service : ${svcName} \
               `/bin/date +"%A %d %B %H:%M:%S"` ""             >>  ${LOG}
#
# Start Netscape
#
        echo "Starting Netscape ... "                           >>  ${LOG}
        echo "Starting Netscape Admin Server ... "              >>  ${LOG}
        cd $APPDIR
        $START_APPCMD                                           >>  ${LOG}
        if [ $? -ne 0 ]; then
          postevent "Netscape Admin Server" start
          exit 2
        fi
        echo "Started Netscape Admin Server"                    >>  ${LOG}
        $START_APPCMD2                                          >>  ${LOG}

        if [ $? -ne 0 ]; then
                postevent "Netscape https Server" start
           exit 2
        fi
        echo "Started Netscape https Server"                    >> ${LOG}
        echo "Started Netscape."                                >> ${LOG}
#
# All done ...
#
        ${EVMPOST} "Start action script for service ${svcName} DONE"
        echo ""Start action script for service ${svcName} DONE, \
                `/bin/date +"%A %d %B %H:%M:%S"` ""             >> ${LOG}
        echo ""                                                 >> ${LOG}
        exit 0
#
;;
#
# Stop section
#
'stop')
        echo ""                                                 >> ${LOG}
        echo ""Stop action script for service : ${svcName} \
                `/bin/date +"%A %d %B %H:%M:%S"` ""             >> ${LOG}
#
# Stop Netscape
#
        echo "Stopping Netscape ... "                           >> ${LOG}
        echo "Stopping Netscape Admin Server ... "              >> ${LOG}
        $STOP_APPCMD                                            >> ${LOG}
        if [ $? -ne 0 ]; then
                postevent "Netscape Admin Server" stop
           exit 2
        fi
        echo "Netscape Admin Server shutdown done ."            >> ${LOG}
        echo "Stopping Netscape https Server ... "              >> ${LOG}
        $STOP_APPCMD2                                           >> ${LOG}
        if [ $? -ne 0 ]; then
                postevent "Netscape https Server" stop
           exit 2
        fi
        echo "Netscape https Server shutdown done ."            >> ${LOG}
        echo "Netscape shutdown done ."                         >> ${LOG}
        ${EVMPOST} "Stop action script for service ${svcName} DONE"
        echo ""Stop action script for service ${svcName} DONE, \
                `/bin/date +"%A %d %B %H:%M:%S"` ""             >> ${LOG}
        echo ""                                                 >> ${LOG}
        exit 0
;;
#
# Probe if application is still alive
#
'check')
        echo ""Probing Netscape daemons at \
                `/bin/date +"%A %d %B %H:%M:%S"`""               >> ${LOG}
        for i in ${PROBE_PROCS}
        do
            probeapp ${i}                                        >> ${LOG}
        done
        echo ""Probing Netscape daemons DONE at \
                `/bin/date +"%A %d %B %H:%M:%S"`""               >> ${LOG}
        exit 0
;;
*)      echo "usage: $0 {start|stop|check}"
        exit 1
;;
esac
```

6. Copy the Netscape CAA script to `/var/cluster/caa/script/netscape.scr`.
7. Create a CAA application resource profile:

   ```
   # caa_profile -create netscape -t application
   ```

   Make sure that your Netscape CAA resource profile looks like the example profile in `/var/cluster/caa/examples/NetScape/ns-httpd.cap`.
8. Register Netscape FastTrack server with CAA:

   ```
   # caa_register netscape
   # caa_stat netscape

   NAME = netscape
   TYPE = application
   STATE = OFFLINE
   ```
9. Start the Netscape FastTrack server:

   ```
   # caa_start netscape
   # caa_stat netscape

   RESOURCE = netscape
   TYPE = application
   STATE = ONLINE on provolone
   ```

2.12.2 Creating a Single-Instance, Highly Available Apache HTTP Server Using CAA

To create a single-instance Apache HTTP server with failover capabilities, follow these steps:

1. Download the latest, standard Apache distribution from the `www.apache.org` Web site to the cluster and follow the site's instructions for building and installing Apache in the `/usr/local/apache` directory.
2. Create a default CAA application resource profile and action script with the following command:

   ```
   # caa_profile -create httpd -t application -B /usr/local/apache/bin/httpd
   ```

   The default profile adopts a failover policy that causes the `httpd` service to fail over to another member when the member on which it is running leaves the cluster. It also allows the `httpd` service to be placed on any active cluster member. You can edit the profile to employ other failover and placement policies and resource dependencies.

   The default action script contains a start entry point that starts the `httpd` service and a stop entry point that stops the `httpd` service.
3. Register the profile with CAA by entering the following command on one member:

   ```
   # caa_register httpd
   ```
4. Start the `httpd` service through CAA by entering the following command on one member:

   ```
   # caa_start httpd
   ```

2.12.3 Creating a Single-Instance Oracle8i Server Using CAA

To create a single-instance Oracle8i Version 8.1.6 or 8.1.7 database server with failover capabilities, follow these steps:

1. Install and configure Oracle8i 8.1.6 or 8.1.7 using the instructions in the Oracle8i documentation.

   Oracle requires that certain kernel attributes be set to specific values, that specific UNIX groups (`dba`, `oinstall`) be created, and that special environment variables be initialized.
2. Before proceeding to set up the CAA service for the Oracle8i single server, you must decide how client applications will reach the service. You can use either the cluster alias feature of the TruCluster Server product or use an interface (IP) alias. If you choose to use a cluster alias, create a new cluster alias for each Oracle8i single server because you can

tune the routing and scheduling attributes of each alias independently. (For information on how to create a cluster alias, see `cluamgr`(8).)

If you want to use a cluster alias, add the IP address and name of the cluster alias to the `/etc/hosts` file.

Add the following line to the `/etc/clua_services` file to set up the properties of the port that the Oracle8i listener uses:

```
listener   1521/tcp     in_single
```

Setting the `in_single` attribute means that the cluster alias subsystem will distribute connection requests directed to the cluster alias to one member of the alias. If that member becomes unavailable, the cluster alias subsystem will select another member of that cluster alias to receive all requests.

To reload service definitions, enter the following command on all members:

```
# cluamgr -f
```

3. If you choose to use an interface address as the target of client requests to the Oracle8i service, add the IP address and name of the cluster alias to the `/etc/hosts` file.
4. In the `listener.ora` and `tnsnames.ora` files, edit the `HOST` field so that it contains the alias that clients will use to reach the service. For example:

```
   .
   .
   .
   (ADDRESS = (PROTOCOL = TCP) (HOST = alias1) (PORT = 1521))
   .
   .
   .
```

5. An example Oracle CAA script is located in `/var/cluster/caa/examples/DataBase/oracle.scr`. Copy the script to `/var/cluster/caa/script/oracle.scr`, and edit it to meet your environment needs such as e-mail accounts, log file destinations, alias preference, and so on. Do not include any file system references in the script.
6. Perform some initial testing of the scripts by first executing the start and stop entry points outside of CAA. For example:

```
# cd /var/cluster/caa/script
# ./oracle.scr start
```

7. Create a CAA application resource profile using the SysMan Station or by entering the following command:

```
# caa_profile -create oracle -t application \
-d "ORACLE Single-Instance Service" -p restricted -h "provolone polishham"
```

Make sure that your Oracle CAA resource profile looks like the example profile in `/var/cluster/caa/examples/DataBase/oracle.cap`.

8. Register the `oracle` profile with CAA using the SysMan Station or by entering the following command on one member:

   ```
   # caa_register oracle
   ```

9. Start the `oracle` service using the SysMan Station or by entering the following command on one member:

   ```
   # caa_start oracle
   ```

2.12.4 Creating a Single-Instance Informix Server Using CAA

To create a single-instance Informix server with failover capabilities, follow these steps:

1. Install and configure Informix using the instructions in the Informix documentation. Edit the clusterwide `/etc/passwd` and `/etc/group` files to contain entries for `informix` and `dba`, respectively.

2. Before proceeding to set up the CAA service for the Informix single server, you must decide how client applications will reach the service. You can use either the cluster alias feature of the TruCluster Server product or use an interface (IP) alias. If you choose to use a cluster alias, create a new cluster alias for each Informix single server because you can tune the routing and scheduling attributes of each alias independently. (For information on how to create a cluster alias, see `cluamgr`(8).)

   If you want to use a cluster alias, add the IP address and name of the cluster alias to the `/etc/hosts` file.

   Add the following line to the `/etc/clua_services` file to set up the properties of the port that the Informix listener uses:

   ```
   informix   8888/tcp     in_single
   ```

   Setting the `in_single` attribute means that the cluster alias subsystem will distribute connection requests directed to the cluster alias to one member of the alias. If that member becomes unavailable, the cluster alias subsystem will select another member of that cluster alias to receive all requests.

   To reload service definitions, enter the following command on all members:

   ```
   # cluamgr -f
   ```

3. If you choose to use an interface address as the target of client requests to the Informix service, add the IP address and name of the cluster alias to the `/etc/hosts` file.

4. An example Informix CAA script is located in `/var/cluster/caa/examples/DataBase/informix.scr`. Copy the script to `/var/cluster/caa/script/informix.scr`, and edit it to meet your environment needs such as e-mail accounts, log file destinations, and so on. Do not include any file system references in the script.

5. Perform some initial testing of the scripts by first executing the start and stop entry points outside of CAA. For example:

   ```
   # cd
   /var/cluster/caa/script
   # ./informix.scr
   start
   ```

6. Create a CAA application resource profile using the SysMan Station or by entering the following command:

   ```
   # caa_profile -create informix -t application \
   -d "INFORMIX Single-Instance Service" -p restricted -h "provolone polishham"
   ```

   Make sure that your Informix CAA resource profile looks like the example profile in `/var/cluster/caa/examples/DataBase/informix.cap`.

7. Register the `informix` profile with CAA using the SysMan Station or by entering the following command on one member:

   ```
   # caa_register informix
   ```

8. Start the `informix` service using the SysMan Station or by entering the following command on one member:

   ```
   # caa_start informix
   ```

3

Using Cluster Aliasing with Multi-Instance Applications

A cluster alias is an IP address that makes some or all of the systems in a cluster look like a single system to clients rather than individual systems. A cluster can have more than one cluster alias. The default cluster alias includes all members of a cluster, and all members can receive packets addressed to this alias.

This chapter provides examples of multi-instance applications that use the default cluster alias to distribute requests among all cluster members. See the TruCluster Server *Cluster Administration* manual for information on how to modify alias attributes.

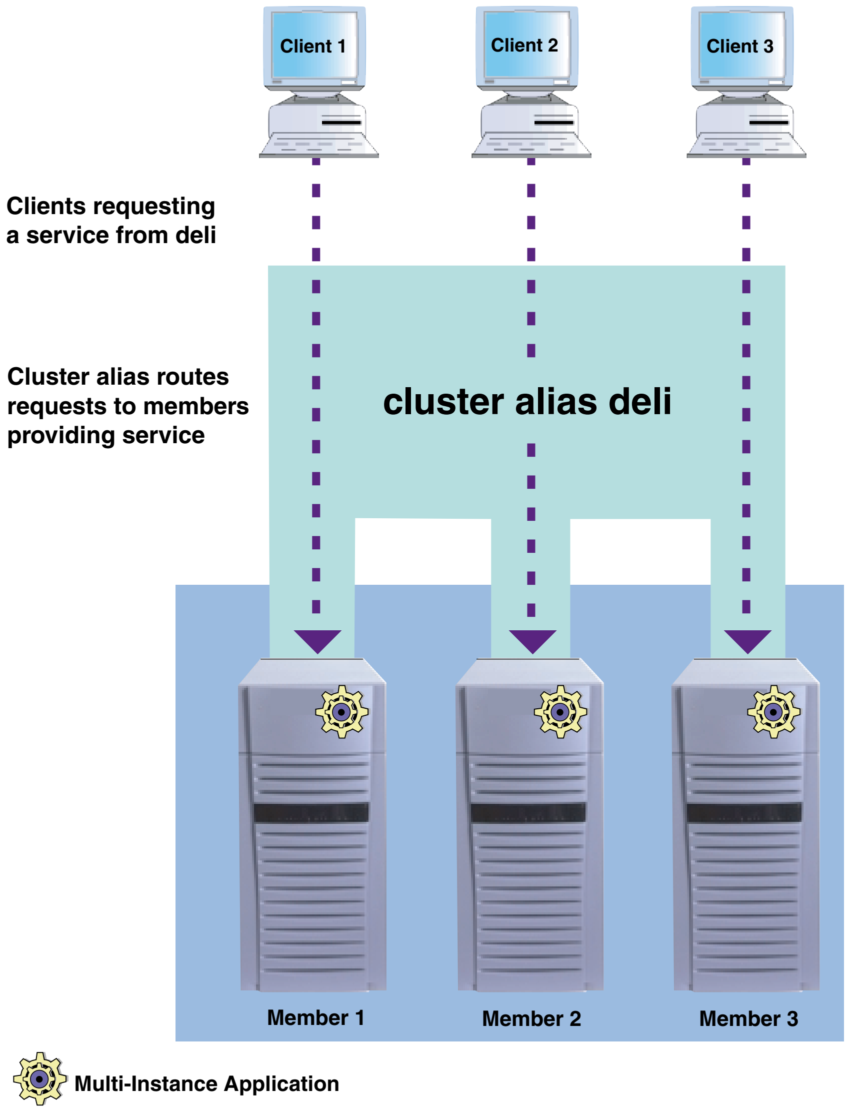
This chapter covers the following topics:

- When to use cluster aliasing (Section 3.1)
- Example applications (Section 3.2 and Section 3.3)

3.1 When to Use Cluster Aliasing

Because a cluster alias sends requests and packets to members of aliases, it is most useful for applications that run on more than one cluster member. Incoming packets or connection requests are distributed among members of a cluster alias. If one member belonging to that alias fails, the cluster alias software transparently routes application-related traffic to the remaining members of the cluster alias. Any new requests will go to the next member. (See the TruCluster Server *Cluster Administration* manual for a summary of the cluster alias features.) Figure 3–1 shows how the cluster alias subsystem distributes client requests.

Figure 3–1: Accessing a Multi-Instance Application Through a Cluster Alias


Client 1
Client 2
Client 3
Clients requesting a service from deli
Cluster alias routes requests to members providing service
cluster alias deli
Member 1
Member 2
Member 3
Multi-Instance Application

ZK-1693U-AI

For single-instance applications, use the cluster application availability (CAA) facility for application control and failover. The cluster alias subsystem still routes packets addressed to an alias, but because CAA ensures that only one member is running the application, the cluster alias will always route requests to that member. See the TruCluster Server *Cluster Administration* manual for a general discussion about the differences between the cluster alias subsystem and CAA.

3.2 Using the Default Cluster Alias to Access a Multi-Instance Apache HTTP Server

To access a highly available, multi-instance Apache HTTP server using the default cluster alias to distribute requests among all cluster members, follow these steps:

1. Download the latest standard Apache distribution from `www.apache.org` and follow the site's instructions for building and installing Apache in the `/usr/local/apache` directory.
2. Edit the `apache/conf/http.conf` configuration file to set the `KeepAlive` parameter to off:

   ```
   KeepAlive Off
   ```

   The cluster alias routes client requests on a connection by connection basis. If you want to load balance requests on a request by request basis using the `selw` and `selp` alias attributes, you should turn off the `KeepAlive` timer that keeps these connections open. If an existing TCP connection is kept open, client requests to the Apache server daemons (`httpds`) running on the cluster will only use the cluster alias to obtain the first connection, and most of the subsequent client requests will go to the member with which that first connection was established.
3. Create member-specific subdirectories for Apache log files in the `/usr/local/apache` directory:

   ```
   # mkdir -p /usr/local/apache/member1/logs
   # mkdir -p /usr/local/apache/member2/logs
   ```
4. Create a context-dependent symbolic link (CDSL) for the log directory:

   ```
   # mkcdsl /usr/local/apache/{memb}/logs /usr/local/apache/logs
   ```

   Note

   If you are using the C shell, you must escape the braces in the `{memb}` string; that is: `\{memb\}`. Otherwise, the braces are stripped by the shell. If you are using the Bourne shell or the Korn shell, you do not have to escape the braces.

5. Add the following entry to the cluster alias services file, `/etc/clua_services`:

   ```
   http  80/tcp  in_multi
   ```

   Setting the `in_multi` attribute for port 80 means that the cluster alias subsystem will distribute connection requests directed to the default cluster alias among all members of the alias.

6. To reload service definitions, enter the following command on all members:

   ```
   # cluamgr -f
   ```

7. On each member, start the Apache server daemon:

   ```
   # /usr/local/apache/bin/httpd
   ```

3.3 Using the Default Cluster Alias to Access a Multi-Instance Netscape FastTrack Server

To access a highly available, multi-instance Netscape FastTrack server using the default cluster alias to distribute requests among all cluster members, follow these steps:

1. If the Netscape FastTrack server is not installed, unpack and install the Netscape FastTrack kit.

   If the Netscape FastTrack server is already installed, locate the server home directory and any Netscape server start and stop scripts.

2. Configure Netscape to use the default cluster alias name. If you do not want to use the default cluster alias, see the TruCluster Server *Cluster Administration* manual for instructions on creating additional aliases.

3. Create member-specific subdirectories for Netscape log files in the `/usr/local/netscape` directory:

   ```
   # mkdir -p /usr/local/netscape/member1/logs
   # mkdir -p /usr/local/netscape/member2/logs
   ```

4. Create a context-dependent symbolic link (CDSL) for the log directory:

   ```
   # mkcdsl /usr/local/netscape/{memb}/logs /usr/local/netscape/logs
   ```

5. Add the following entry to the cluster alias services file, `/etc/clua_services`:

   ```
   http     80/tcp     in_multi
   ```

   Setting the `in_multi` attribute for port 80 means that the cluster alias subsystem will distribute connection requests directed to the default cluster alias among all members of the alias.

6. To reload service definitions, enter the following command on all members:

   ```
   # cluamgr -f
   ```

7. Change your directory to `/usr/opt/netscape/suitespot` and run the setup utility:

   ```
   # cd /usr/opt/netscape/suitespot
   # ns-setup
   ```

This procedure sets up the Netscape Admin server and starts the Admin daemon. When prompted for the name of the server, enter the name of the default cluster alias.

8. Add the default cluster alias name to the list of allowed X server connections and connect to the Netscape Admin server:

   ```
   # xhost +aliasname
   # netscape http://youraliasname.yourdomain:nnnn/
   ```

   The *nnnn* value is the port number that you chose in step 5.

   Optionally, you can allow additional nodes to connect to the Netscape Admin daemon as follows:

   a. Click on the Admin Preference button.
   b. When the screen is displayed, click on Superuser Access Control.
   c. If you want to be able to connect all cluster members to the Netscape Admin server, add all members' IP addresses.
   d. Click on Server Administration to return to the main screen.

   Note

   If you want to start the Netscape Admin daemon at boot time, create a `/sbin/rc3.d` link that points to `/usr/opt/netscape/suitespot/start-admin`. If you do not plan to change your Netscape configuration, you do not need to start the Admin daemon.

9. Create the Netscape FastTrack server as follows:

   a. In the Servers Supporting General Administration section, click on Create New Netscape Fast Track Server 3.01.
   b. Set the default cluster alias as both the Server Name and the Server Identifier. You do not have to make any other changes. If you selected the cluster alias for the Netscape Admin server, all fields are already filled in correctly.
   c. Click OK and then click Return to Server Admin. Note that you can create more than one server on a single system.

10. Start the Netscape FastTrack server.

Part 2

Moving Applications to TruCluster Server

4 General Application Migration Issues

This chapter describes general migration issues that are relevant to all types of applications. Table 4–1 lists each migration issue and the types of applications that might encounter them, as well as where to find more information.

Table 4–1: Application Migration Considerations

Issues	Application Types Affected	For More Information
Clusterwide and member-specific files	Single-instance Multi-instance Distributed	Section 4.1
Device naming	Single-instance Multi-instance Distributed	Section 4.2
Interprocess communication	Multi-instance Distributed	Section 4.3
Synchronized access to shared data	Multi-instance Distributed	Section 4.4
Member-specific resources	Single-instance	Section 4.5
Expanded process IDs (PIDs)	Multi-instance Distributed	Section 4.6
Distributed lock manager (DLM) parameters removed	Multi-instance Distributed	Section 4.7
Licensing	Single-instance Multi-instance Distributed	Section 4.8
Blocking layered products	Single-instance Multi-instance Distributed	Section 4.9
Other migration issues	Single-instance Multi-instance Distributed	Section 4.10

4.1 Clusterwide and Member-Specific Files

In a cluster, there are two sets of configuration data:

- Clusterwide data

  Clusterwide data pertains to files and logs that can be shared by all members of a cluster. For example, when two systems are members of a cluster, they share a common `/etc/passwd` file that contains information about the authorized users for both systems.

  Sharing configuration or management data makes file management easier. For example, Apache and Netscape configuration files can be shared, allowing you to manage the application from any node in the cluster.

- Member-specific data

  Do not allow files that contain member-specific data to be shared by all members of a cluster. Member-specific data may be configuration details that pertain to hardware found only on a specific system, such as a layered product driver for a specific printer connected to one cluster member.

Because the Cluster File System (CFS) makes all files visible to and accessible by all cluster members, those applications that require clusterwide configuration data can easily write to a configuration file that all members can view. However, an application that must use and maintain member-specific configuration information needs to take some additional steps to avoid overwriting files.

To avoid overwriting files, consider using one of the following methods:

Method	**Advantage**	**Disadvantage**
Single file	Easy to manage.	Application must be aware of how to access member-specific data in the single file.
Multiple files	Keeps configuration information in a set of clusterwide files.	Multiple copies of files need to be maintained. Application must be aware of how to access member-specific files.
Context-dependent symbolic links (CDSLs)	Keeps configuration information in member-specific areas. CDSLs are transparent to the application; they look like soft links.	Moving or renaming files will break symbolic links. Application must be aware of how to handle CDSLs. Using CDSLs makes it more difficult for an application to find out about other instances of that application in the cluster.

Consider the alternative that best fits your application's needs. The following sections describe each approach.

4.1.1 Using a Single File

Using a single, uniquely named file keeps application configuration information in one clusterwide file as separate records for each node. The application reads and writes the correct record in the file. Managing a single file is easy because all data is in one central location.

As an example, in a cluster the `/etc/printcap` file contains entries for specific printers. The following parameter can be specified to indicate which nodes in the cluster can run the spooler for the print queue:

```
:on=nodename1,nodename2,nodename3,...:
```

If the first node is up, it will run the spooler. If that node goes down, the next node, if it is up, will run the spooler, and so on.

4.1.2 Using Multiple Files

Using uniquely named multiple files keeps configuration information in a set of clusterwide files. For example, each cluster member has its own member-specific `gated` configuration file in `/etc`. Instead of using a context-dependent symbolic link (CDSL) to reference member-specific files through a common file name, the naming convention for these files takes advantage of member IDs to create a unique name for each member's file. For example:

```
# ls -l /etc/gated.conf.member*
-rw-r--r--   1 root   system    466 Jun 21 17:37 /etc/gated.conf.member1
-rw-r--r--   1 root   system    466 Jun 21 17:37 /etc/gated.conf.member2
-rw-r--r--   1 root   system    466 Jun 21 13:28 /etc/gated.conf.member3
```

This approach requires more work to manage because multiple copies of files need to be maintained. For example, if the member ID of a cluster member changes, you must find and rename all member-specific files belonging to that member. Also, if the application is unaware of how to access member-specific files, you must configure it to do so.

4.1.3 Using CDSLs

Tru64 UNIX Version 5.0 introduced a special form of symbolic link, called a context-dependent symbolic link (CDSL), that TruCluster Server uses to point to the correct file for each member. CDSLs are useful when running multiple instances of an application on different cluster members on different sets of data.

Using a CDSL keeps configuration information in member-specific areas. However, the data can be referenced through the CDSL. Each member reads the common file name, but is transparently linked to its copy of the configuration file. CDSLs are an alternative to maintaining member-specific configuration information when an application cannot be easily changed to use multiple files.

The following example shows the CDSL structure for the file `/etc/rc.config`:

```
/etc/rc.config -> ../cluster/members/{memb}/etc/rc.config
```

For example, where a cluster member has a member ID of 3, the pathname `/cluster/members/{memb}/etc/rc.config` resolves to `/cluster/members/member3/etc/rc.config`.

Tru64 UNIX provides a `mkcdsl` command that lets system administrators create CDSLs and update a CDSL inventory file. For more information on this command, see the TruCluster Server *Cluster Administration* manual and `mkcdsl`(8). For more information on creating CDSLs and hints to avoid overwriting them, see the Tru64 UNIX *System Administration* manual, `hier`(5), `ln`(1), and `symlink`(2).

4.2 Device Naming

Tru64 UNIX Version 5.0 introduced a new device-naming convention that consists of a descriptive name for the device and an instance number. These two elements form the basename of the device. For example:

Location in /dev	Device Name	Instance	Basename
`./disk`	`dsk`	0	`dsk0`
`./disk`	`cdrom`	1	`cdrom1`
`./tape`	`tape`	0	`tape0`

Moving a disk from one physical connection to another does not change the device name for the disk. For a detailed discussion of this device-naming model, see the Tru64 UNIX *System Administration* manual.

Although Tru64 UNIX Version 5.0 recognizes both the old-style (`rz`) and new-style (`dsk`) device names, TruCluster Server Version 5.1 and later recognizes only new-style device names. Applications that depend on old-style device names or the `/dev` directory structure must be modified to use the newer device-naming convention.

You can use the `hwmgr` utility, a generic utility for managing hardware, to help map device names to their bus, target, and LUN position after installing Tru64 UNIX Version 5.1A. For example, enter the following command to view devices:

```
# hwmgr -view devices

  HWID: Device Name         Mfg    Model         Location
  --------------------------------------------------------------------
  40: /dev/disk/dsk2c      DEC      RZ28M    (C) DEC bus-1-targ-1-lun-0
  41: /dev/disk/dsk3c      DEC      RZ28L-AS (C) DEC bus-1-targ-2-lun-0
  42: /dev/disk/dsk4c      DEC      RZ29B    (C) DEC bus-1-targ-3-lun-0
  43: /dev/disk/dsk5c      DEC      RZ28D    (C) DEC bus-1-targ-4-lun-0
  44: /dev/disk/dsk6c      DEC      RZ28L-AS (C) DEC bus-1-targ-5-lun-0
  45: /dev/disk/dsk7c      DEC      RZ1CF-CF (C) DEC bus-1-targ-8-lun-0
  46: /dev/disk/dsk8c      DEC      RZ1CB-CS (C) DEC bus-1-targ-9-lun-0
  47: /dev/disk/dsk9c      DEC      RZ1CF-CF (C) DEC bus-1-targ-10-lun-0
  48: /dev/disk/dsk10c     DEC      RZ1CF-CF (C) DEC bus-1-targ-11-lun-0
  49: /dev/disk/dsk11c     DEC      RZ1CF-CF (C) DEC bus-1-targ-12-lun-0
  50: /dev/disk/dsk12c     DEC      RZ1CF-CF (C) DEC bus-1-targ-13-lun-0
  97: /dev/kevm
 122: /dev/disk/floppy2c            3.5in floppy     fdi0-unit-0
 136: /dev/disk/dsk15c     DEC      RZ28M    (C) DEC bus-0-targ-0-lun-0
 137: /dev/disk/dsk16c     DEC      RZ28L-AS (C) DEC bus-0-targ-1-lun-0
 138: /dev/disk/dsk17c     DEC      RZ28     (C) DEC bus-0-targ-2-lun-0
 139: /dev/disk/dsk18c     DEC      RZ28D    (C) DEC bus-0-targ-3-lun-0
 140: /dev/disk/cdrom2c    DEC      RRD46   (C) DEC  bus-0-targ-6-lun-0
```

Use the following command to view devices clusterwide:

```
# hwmgr -view devices -cluster

HWID: Device Name        Mfg   Model           Hostname    Location
----------------------------------------------------------------------
  4: /dev/kevm                                 provolone
 33: /dev/disk/floppy0c        3.5in floppy    provolone   fdi0-unit-0
 37: /dev/disk/dsk0c     DEC   RZ26L (C) DEC   provolone   bus-0-targ-0-lun-0
 38: /dev/disk/cdrom0c   DEC   RRD46 (C) DEC   provolone   bus-0-targ-4-lun-0
 39: /dev/disk/dsk1c     DEC   RZ1DF-CB (C) DEC provolone  bus-0-targ-8-lun-0
 40: /dev/disk/dsk2c     DEC   RZ28M (C) DEC   provolone   bus-1-targ-1-lun-0
 40: /dev/disk/dsk2c     DEC   RZ28M (C) DEC   pepicelli   bus-1-targ-1-lun-0
 40: /dev/disk/dsk2c     DEC   RZ28M (C) DEC   polishham   bus-1-targ-1-lun-0
 .
 .
 .
```

For more information on using this command, see `hwmgr`(8).

When modifying applications to use the new device-naming convention, look for the following:

- Disks that are included in Advanced File System (AdvFS) domains
- Raw disk devices
- Disks that are encapsulated in Logical Storage Manager (LSM) volumes or that are included in disk groups
- Disk names in scripts
- Disk names in data files (Oracle OPS and Informix XPS)
- SCSI bus renumbering

Note

If you previously renumbered SCSI buses in your ASE, closely verify the mapping from physical device to bus number during an upgrade to TruCluster Server. See the TruCluster Server *Cluster Installation* manual for more information.

4.3 Interprocess Communication

Among the mechanisms that are used by applications to perform interprocess communication (IPC) are shared memory, named pipes, and signals. However, shared memory, named pipes, and signals are not supported clusterwide in TruCluster Server. If an application uses any of these IPC methods, it must be restricted to running as a single-instance application.

To run multiple instances of an application on more than one cluster member, perform all IPC through remote procedure calls (RPCs) or socket connections.

4.4 Synchronized Access to Shared Data

Multiple instances of an application running within a cluster must synchronize with each other for most of the same reasons that multiprocess and multithreaded applications synchronize on a standalone system. However, memory-based synchronization mechanisms (such as critical sections, mutexes, simple locks, and complex locks) work only on the local system and not clusterwide. Shared file data must be synchronized, or files must be used to synchronize the execution of instances across the cluster.

Because the Cluster File System (CFS) is fully POSIX compliant, an application can use `flock()` system calls to synchronize access to shared files among instances. You can also use the distributed lock manager (DLM) API library functions for more sophisticated locking capabilities (such as additional lock modes, lock conversions, and deadlock detection). Because the DLM API library is supplied only in the TruCluster Server product, make sure that code that uses its functions and that is meant also to run on nonclustered systems precedes any DLM function calls with a call to `clu_is_member()`. The `clu_is_member()` function verifies that the system is in fact a cluster member. For more information about this command, see `clu_is_member`(3).

4.5 Member-Specific Resources

If multiple instances of an application are started simultaneously on more than one cluster member, the application may not work properly because it depends on resources that are available only from a specific member, such as large CPU cycles or large physical memory. This may restrict the application to running as a single instance in a cluster. Changing these characteristics in an application may be enough to allow it to run as multiple instances in a cluster.

4.6 Expanded PIDs

In TruCluster Server, process identifiers (PIDs) are expanded to a full 32-bit value. The data type `PID_MAX` is increased to 2147483647 (0x7fffffff); therefore, any applications that test for `PID <= PID_MAX` must be recompiled.

To ensure that PIDs are unique across a cluster, PIDs for each cluster member are based on the member ID and are allocated from a range of numbers unique to that member. The formula for available PIDs in a cluster is:

```
PID = (memberid * (2**19)) + 2
```

Typically, the first two values are reserved for the `kernel idle` process and `/sbin/init`. For example, PIDs 524,288 and 524,289 are assigned to `kernel idle` and `init`, respectively.

Use PIDs to uniquely identify log and temporary files. If an application does store a PID in a file, make sure that that file is member-specific.

4.7 DLM Parameters Removed

Because the distributed lock manager (DLM) persistent resources, resource groups, and transaction IDs are enabled by default in TruCluster Server Version 1.6 and later, the `dlm_disable_rd` and `dlm_disable_grptx` attributes are unneeded and have been removed from the DLM kernel subsystem.

4.8 Licensing

This section discusses licensing constraints and issues.

4.8.1 TruCluster Server Licensing Constraints

TruCluster Server Version 5.1A does not support clusterwide licensing. Each time that you add an additional member to the cluster, you must register all required application licenses on that member for applications that may run on that member.

4.8.2 Layered Product Licensing and Network Adapter Failover

The Redundant Array of Independent Network Adapters (NetRAIN) and the Network Interface Failure Finder (NIFF) provide mechanisms for facilitating network failover and replace the monitored network interface method that was employed in the TruCluster Available Server and Production Server products.

NetRAIN provides transparent network adapter failover for multiple adapter configurations. NetRAIN monitors the status of its network interfaces with NIFF, which detects and reports possible network failures. You can use NIFF to generate events when network devices, including a composite NetRAIN device, fail. You can monitor these events and take appropriate actions when a failure occurs. For more information about NetRAIN and NIFF, see the Tru64 UNIX *Network Administration: Connections* manual.

In a cluster, an application may fail over and restart itself on another member. If it performs a license check when restarting, it may fail because it was looking for a particular member's IP address or its adapter's media access control (MAC) address.

Licensing schemes that use a network adapter's MAC address to uniquely identify a machine can be affected by how NetRAIN changes the MAC address. All network drivers support the SIOCRPHYSADDR `ioctl` that fetches MAC addresses from the interface. This `ioctl` returns two addresses in an array:

- Default hardware address — the permanent address that is taken from the small PROM that each LAN adapter contains.
- Current physical address — the address that the network interface responds to on the wire.

For licensing schemes that are based on MAC addresses, use the default hardware address that is returned by SIOCRPHYSADDR `ioctl`; do not use the current physical address because NetRAIN modifies this address for its own use. See the reference page for your network adapter (for example, `tu`(7)) for a sample program that uses the SIOCRPHYSADDR `ioctl`.

4.9 Blocking Layered Products

A blocking layered product is a product that prevents the `installupdate` command from completing during an update installation of TruCluster Server Version 5.1A. Blocking layered products must be removed from the cluster before starting a rolling upgrade that will include running the `installupdate` command.

Unless a layered product's documentation specifically states that you can install a newer version of the product on the first rolled member, and that the layered product knows what actions to take in a mixed-version cluster, we strongly recommend that you do not install either a new layered product or a new version of a currently installed layered product during a rolling upgrade.

The TruCluster Server *Cluster Installation* manual lists all layered products that are known to break an update installation on TruCluster Server Version 5.1A.

4.10 Other Migration Issues

This section discusses other migration issues to consider before moving applications to TruCluster Server.

4.10.1 UFS Dependencies

TruCluster Server Version 5.1A supports the UNIX File System (UFS) as a read-only file system clusterwide. That is, a UFS file system explicitly mounted read-only is served for clusterwide read-only access by a member selected for its connectivity to the storage containing the file system.

TruCluster Server Version 5.1A includes read/write support for UFS file systems, but the file system is accessible only by that member. No other cluster members can access that file system. There is no failover should that member go down.

Advanced File System (AdvFS) file systems are required to upgrade an existing TruCluster Available Server or Production Server environment to TruCluster Server. If you are using UFS, we recommend that you migrate these file systems to AdvFS before beginning an upgrade to TruCluster Server.

See the TruCluster Server *Cluster Installation* manual for complete TruCluster Server upgrade requirements and the Tru64 UNIX *AdvFS Administration* for information about migrating UFS file systems to AdvFS.

4.10.2 New On-Disk Format for AdvFS Domains

Tru64 UNIX Version 5.0 and later employs a new-style AdvFS format, which provides performance enhancements. Tru64 UNIX Version 5.0 and later recognizes both the new-style and old-style formats. However, file domains created in previous versions of Tru64 UNIX do not support these enhancements.

To convert data to the newer format, back up the data and restore it to a new-style AdvFS domain. There is no conversion utility.

5 Moving ASE Applications to TruCluster Server

This chapter describes how to move Available Server Environment (ASE) applications to TruCluster Server Version 5.1A.

To continue single-instance application availability and failover, TruCluster Server provides the cluster application availability (CAA) subsystem. Before moving ASE services to TruCluster Server, make sure that you are familiar with CAA. See Chapter 2 for detailed information on how to use CAA.

This chapter discusses the following topics:

- Comparing ASE to CAA (Section 5.1)
- Preparing to move ASE services to TruCluster Server (Section 5.2)
- Reviewing ASE scripts (Section 5.3)
- Using an IP alias or networking services (Section 5.4)
- Partitioning file systems (Section 5.5)

5.1 Comparing ASE to CAA

CAA provides resource monitoring and application restart capabilities. It provides the same type of application availability that is provided by user-defined services in the TruCluster Available Server Software and TruCluster Production Server Software products. Table 5–1 compares ASE services with their equivalents in the TruCluster Server product.

Table 5–1: ASE Services and Their TruCluster Server Equivalents

ASE Service	ASE Description	TruCluster Server Equivalent
Disk service (Section 5.1.1)	One or more highly available file systems, Advanced File System (AdvFS) filesets, or Logical Storage Manager (LSM) volumes. Can also include a disk-based application.	Cluster File System (CFS), device request dispatcher, and CAA

Table 5–1: ASE Services and Their TruCluster Server Equivalents (cont.)

ASE Service	ASE Description	TruCluster Server Equivalent
Network File System (NFS) service (Section 5.1.2)	One or more highly available file systems, AdvFS filesets, or LSM volumes that are exported. Can also include highly available applications.	Automatically provided by CFS and the default cluster alias. No service definition required.
User-defined service (Section 5.1.3)	An application that fails over using action scripts.	CAA
Distributed raw disk (DRD) service (Section 5.1.4)	Allows a disk-based, user-level application to run within a cluster by providing clusterwide access to raw physical disks.	Automatically provided by the device request dispatcher. No service definition required.
Tape service (Section 5.1.5)	Depends on a set of one or more tape devices for configuring the NetWorker server and other servers for failover.	CFS, device request dispatcher, and CAA

The following sections describe these ASE services and explain how to handle them in a TruCluster Server environment.

5.1.1 Disk Service

ASE

An ASE disk service includes one or more highly available file systems, Advanced File System (AdvFS) filesets, or Logical Storage Manager (LSM) volumes. Disk services can also include a disk-based application and are managed within the ASE.

TruCluster Server

The Cluster File System (CFS) makes all file storage available to all cluster members, and the device request dispatcher makes disk storage available clusterwide. Because file systems and disks are now available throughout the cluster, you do not need to mount and fail them over explicitly in your action scripts. For more information about using CFS, see the TruCluster Server *Cluster Administration* manual and `cfsmgr`(8).

Use CAA to define a disk service's relocation policies and dependencies. If you are not familiar with CAA, see Chapter 2.

Disk services can be defined to use either a cluster alias or an IP alias for client access.

5.1.2 NFS Service

ASE

An ASE Network File System (NFS) service includes one or more highly available file systems, AdvFS filesets, or LSM volumes that a member system exports to clients making the data highly available. NFS services can also include highly available applications.

TruCluster Server

When configured as an NFS server, a TruCluster Server cluster provides highly available access to the file systems it exports. CFS makes all file storage available to all cluster members. You no longer need to mount any file systems within your action scripts. Define the NFS file system to be served in the `/etc/exports` file, as you would on a standalone server.

Remote clients can mount NFS file systems that are exported from the cluster by using the default cluster alias or by using alternate cluster aliases.

5.1.3 User-Defined Service

ASE

An ASE user-defined service consists only of an application that you want to fail over using your own action scripts. The application in a user-defined service cannot use disks.

TruCluster Server

In ASE you may have created a highly available Internet login service by setting up user-defined start and stop action scripts that invoked `ifconfig`. In TruCluster Server you do not need to create a login service. Clients can log into the cluster by using the default cluster alias.

Use CAA to define a user-defined service's failover and relocation policies and dependencies. If you are not familiar with CAA, see Chapter 2.

5.1.4 DRD Service

ASE

An ASE distributed raw disk (DRD) service provides clusterwide access to raw physical disks. A disk-based, user-level application can run within a cluster, regardless of where in the cluster the physical storage it depends upon is located. A DRD service allows applications, such as database and transaction processing (TP) monitor systems, parallel access to storage media from multiple cluster members. When creating a DRD service, you specify the physical media that the service will provide clusterwide.

TruCluster Server

The device request dispatcher subsystem makes all disk and tape storage available to all cluster members, regardless of where the physical storage is located. You no longer need to explicitly fail over disks when an application fails over to another member.

Prior to Tru64 UNIX Version 5.0, a separate DRD namespace was provided in a TruCluster Production Server environment. As DRD services were added, the `asemgr` utility assigned DRD special file names sequentially in the following form:

```
/dev/rdrd/drd1
/dev/rdrd/drd2
/dev/rdrd/drd3
⋮
```

In a TruCluster Server cluster, you access a raw disk device partition in a TruCluster Server configuration in the same way that you do on a Tru64 UNIX Version 5.0 or later standalone system — by using the device's special file name in the `/dev/rdisk` directory. For example:

```
/dev/rdisk/dsk2c
```

5.1.5 Tape Service

ASE

An ASE tape service depends on a set of one or more tape devices. It may also include media changer devices and file systems. A tape service enables you to configure the Legato NetWorker server and servers for other client/server-based applications for failover. The tape drives, media changers, and file systems all fail over as one unit.

TruCluster Server

CFS makes all file storage available to all cluster members, and the device request dispatcher makes disk and tape storage available clusterwide. Because file systems, disks, and tapes are now available throughout the cluster, you do not need to mount and fail them over explicitly in your action scripts.

Use CAA to define a tape service's failover and relocation policies and dependencies. If you are not familiar with CAA, see Chapter 2.

Tape services can be defined to use either a cluster alias or an IP alias for client access.

5.2 Preparing to Move ASE Services to TruCluster Server

As discussed in the TruCluster Server *Cluster Installation* manual, save both the `var/ase/config/asecdb` database and a text copy of the database before shutting down your TruCluster Available Server or TruCluster Production Server cluster. Having the ASE database content available makes it easier to set up applications on TruCluster Server.

How ASE database content is saved differs between versions of TruCluster Available Server and TruCluster Production Server. The following sections explain how to save ASE database content on different versions of TruCluster Server.

5.2.1 Saving ASE Database Content from TruCluster Available Server and Production Server Version 1.5 or Later

To save the `/var/ase/config/asecdb` database in ASCII format, enter the following `asemgr` command:

```
# asemgr -d -C > asecdb.txt
```

This command saves the ASE database content and outputs it in ASCII format to a file called `asecdb.txt`.

The following information, which is saved from a sample ASE database, is helpful when creating a CAA profile:

```
!! ASE service configuration for netscape

@startService netscape
Service name: netscape
Service type: DISK
Relocate on boot of favored member: no
Placement policy: balanced
.
.
.
```

The following information saved from a sample ASE database is helpful when installing and configuring an application on TruCluster Server:

```
IP address: 16.141.8.239
Device: cludemo#netscape
  cludemo#netscape mount point: /clumig/Netscape
  cludemo#netscape filesystem type: advfs
  cludemo#netscape mount options: rw
  cludemo#netscape mount point group owner: staff
Device: cludemo#cludemo
  cludemo#cludemo mount point: /clumig/cludemo
  cludemo#cludemo filesystem type: advfs
  cludemo#cludemo mount options: rw
  cludemo#cludemo mount point group owner: staff
```

```
AdvFS domain: cludemo
  cludemo volumes: /dev/rz12c
.
.
.
```

5.2.2 Saving ASE Database Content from TruCluster Available Server and Production Server Version 1.4 or Earlier

On a TruCluster Version 1.4 or earlier system, you cannot use the `asemgr` command to save *all* ASE service information. The `asemgr` command does not capture ASE script information. You must use the `asemgr` utility if you want to save all information.

To save script data, follow these steps:

1. Start the `asemgr` utility.
2. From the ASE Main Menu, choose Managing ASE Services.
3. From the Managing ASE Services menu, choose Service Configuration.
4. From the Service Configuration menu, choose Modify a Service.
5. Select a service from the menu.
6. Choose General service information.
7. From the User-defined Service Modification menu, choose User-defined action scripts.
8. Select Start action from the menu. Record the values for script argument and script timeout.
9. From the menu, choose Edit the start action script.

   Write the internal script to a file on permanent storage where it will not be deleted.

Repeat these steps as necessary for all stop, add, and delete scripts. For user-defined services, also save the check script.

To save ASE database content and the rest of your ASE service information (placement policies, service names, and so on), enter the following commands:

```
# asemgr -dv > ase.services.txt
# asemgr -dv {ServiceName}
```

The name of the service, `ServiceName`, is taken from the output that is produced by `asemgr -dv`. Execute `asemgr -dv {ServiceName}` for each service.

5.3 ASE Script Considerations

Review ASE scripts carefully. Consider the following issues for scripts to work properly on TruCluster Server:

- Replace ASE commands with cluster application availability (CAA) commands (see Section 5.3.1).
- Combine separate start and stop scripts (see Section 5.3.2).
- Redirect script output (see Section 5.3.3).
- Replace `nfs_config` with `ifconfig` or create a cluster alias (see Section 5.3.4).
- Handle errors correctly (see Section 5.3.5).
- Remove storage management information from action scripts (see Section 5.3.6).
- Convert device names (see Section 5.3.7).
- Remove references to ASE-specific environment variables (see Section 5.3.8).
- Exit codes (see Section 5.3.9).
- Post events with Event Manager (EVM) (see Section 5.3.10).

5.3.1 Replace ASE Commands with CAA Commands

In TruCluster Server Version 5.1A, the `asemgr` command is replaced by several CAA commands. The following table compares ASE commands with their equivalent CAA commands:

ASE Command	CAA Command	Description
`asemgr -d`	`caa_stat`	Provides status on CAA resources clusterwide
`asemgr -m`	`caa_relocate`	Relocates an application resource from one cluster member to another
`asemgr -s`	`caa_start`	Starts application resources
`asemgr -x`	`caa_stop`	Stops application resources
	`caa_profile`	Creates, validates, deletes, and updates a CAA resource profile
	`caa_register`	Registers a resource with CAA
	`caa_unregister`	Unregisters a resource with CAA

The caa_profile, caa_register, and caa_unregister commands provide functionality that is unique to the TruCluster Server product. For information on how to use any of the CAA commands, see Chapter 2.

5.3.2 Combine Start and Stop Scripts

CAA does not call separate scripts to start and stop an application. If you have separate start and stop scripts for your application, combine them into one script. The following CAA script shows both start and stop entry points:

```
#!/usr/bin/ksh -p
  .
  .
  .

#
# Start section - start the process and report results
#
case $1 in
'start')
        echo ""                                             >> ${LOG}
        echo ""Start action script for service : ${svcName} \
        `/bin/date +"%A %d %B %H:%M:%S"` ""                 >> ${LOG}

        echo "Starting $svcName ... "                       >> ${LOG}
        cd $APPDIR
        if [[ $START_APPCMD != "" ]]
        then
                $START_APPCMD                               >> ${LOG}
                if [ $? -ne 0 ]; then
                        postevent "$svcName" start
                        exit 2
                fi
        fi

        if [[ $START_APPCMD2 != "" ]]
        then
                $START_APPCMD2
                if [$? -ne 0]; then
                        postevent "$svcName" start
                        exit 2
                fi
        fi

        echo "Started $svcName"                             >> ${LOG}
#
# Stop section - stop the process and report results
#
'stop')
        echo ""                                             >> ${LOG}
        echo ""Stop action script for service : ${svcName} \
                `/bin/date +"%A %d %B %H:%M:%S"` ""         >> ${LOG}

        echo "Stopping $svcName ... "                       >> ${LOG}
        cd $APPDIR
        if [[ $STOP_APPCMD != "" ]]
        then
                $STOP_APPCMD                                >> ${LOG}
                if [ $? -ne 0 ]; then
                        postevent "$svcName" stop
                        exit 2
                fi
        fi
        if [[ $STOP_APPCMD2 != "" ]]
        then
                $STOP_APPCMD2
                if [ $? -ne 0 ]; then
                        postevent "$svcName" stop
                        exit 2
                fi
        fi
#
# Check section - check the process and report results
#
'check')
        echo ""Probing $svcName at \
                `/bin/date +"%A %d %B %H:%M:%S"`""            >> ${LOG}
        for i in ${PROBE_PROCS}
        do
                probeapp ${i}                                 >> ${LOG}
        done
        echo ""Probing $svcName DONE at \
                `/bin/date +"%A %d %B %H:%M:%S"`""            >> ${LOG}
;;
*)              echo "usage: $0 {start|stop|check}"
                exit 1
;;
esac
```

5.3.3 Redirect Script Output

CAA scripts run with standard output and standard error streams directed to `/dev/null`. If you want to capture these streams, we recommend that you employ one of the following methods, in the following order of preference:

1. Use the Event Manager (EVM) (as demonstrated in the template script `/var/cluster/caa/template/template.scr`). This is the preferred method because of the output management that EVM provides.
2. Use the `logger` command to direct output to the system log file (`syslog`). See `logger`(1) for more information. This method is not as flexible as using EVM. For example, messages stored in `syslog` are simple text and cannot take advantage of the advanced formatting and searching capabilities of EVM.
3. Direct output to `/dev/console`. This method does not have a persistent record; messages appear only at the console.
4. Direct output to a file. With this method, be aware of log file size, and manage file space appropriately.

Sample CAA scripts that redirect output are available in the `/cluster/caa/examples` directory.

5.3.4 Replace nfs_ifconfig Script

In TruCluster Server there is no longer an `nfs_ifconfig` script as found in the TruCluster ASE. Replace `nfs_ifconfig` scripts with either an `ifconfig alias/-alias` statement in a CAA action script or use a cluster alias.

For more information about using an interface alias in a CAA script, see Section 5.4.1. See the the examples in Section 2.12 for information on using a cluster alias with CAA single-instance applications.

5.3.5 Handling Errors Correctly

Create scripts in TruCluster Server so that they handle errors properly. The "filesystem busy" message is no longer returned. Therefore, an application may be started twice, even if some of its processes are still active on another member.

Make sure that your stop script can stop all processes, or use `fuser`(8) to stop application processes.

5.3.6 Remove Storage Management Information

An ASE service's storage needed to be:

- On a bus that was shared by all cluster members
- Defined as part of the service using the `asemgr` utility
- Managed by service scripts

Because the Cluster File System (CFS) in TruCluster Server makes all file storage available to all cluster members (access to storage is built into the cluster architecture), you no longer need to manage file system mounting and failover within action scripts.

You can remove all storage management information from scripts on TruCluster Server. For example, SAP R/3 scripts may have been set up to mount file systems within their own scripts. You can remove these mount points.

5.3.7 Convert Device Names

As described in Section 4.2, scripts that reference old device names must be modified to use the new device-naming model that was introduced with Tru64 UNIX Version 5.0.

If you used the `ase_fix_config` command to renumber buses, save the output from the command during the upgrade and use it to verify physical devices against bus numbers. See the TruCluster Server *Cluster Installation* manual for information on how to run the `ase_fix_config` command.

5.3.8 Replace or Remove ASE Variables

ASE scripts may contain the following ASE environment variables:

- `MEMBER_STATE`

  In ASE the `MEMBER_STATE` variable is placed in a stop script to determine whether or not the script is executing on a running system or on a system that is booting. The `MEMBER_STATE` variable has one of the following variables:

  – `RUNNING`

  – `BOOTING`

  During system startup, TruCluster Server does not provide the option to run the stop section of an action script. To perform file cleanup of references, log files, and so on, move these cleanup actions to the start section of your action script.

- `ASEROUTING`

  The `ASEROUTING` variable is replaced by the TruCluster Server cluster alias subsystem functionality. Remove this variable from TruCluster Server application action scripts.

- `ASE_PARTIAL_MIRRORING`

  The `ASE_PARTIAL_MIRRORING` variable does not exist in TruCluster Server. Remove this variable from TruCluster Server application action scripts.

5.3.9 Exit Codes

ASE exit codes for all scripts return 0 for success; anything else equals failure.

Each entry point of a CAA script returns an exit code of 0 for success and a nonzero value for failure. (Scripts that are generated by the `caa_profile` command from the script template return a 2 for failure.) For the `check` section of a CAA script, an exit code of 0 means that the application is running.

5.3.10 Posting Events

The Event Manager (EVM) provides a single point of focus for the multiple channels (such as log files) through which system components report event and status information. EVM combines these events into a single event stream, which the system administrator can monitor in real time or view as historical events retrieved from storage. Use the `evmpost(8)` command to post events into EVM from action scripts.

Note

The CAA sample scripts that are located in the `/var/cluster/caa/examples` directory use EVM to post events.

5.4 Networking Considerations

The following sections discuss networking issues to consider when moving ASE services to TruCluster Server:

- Using an alias (Section 5.4.1)
- Networking services (Section 5.4.2)

5.4.1 Using an Alias

If an application requires the use of an alias, you can use either a cluster alias or an interface alias.

Using a cluster alias is most appropriate when:

- Multiple member systems must appear as a single system to clients of Transport Control Protocol (TCP)-based or User Datagram Protocol (UDP)-based applications. Often multiple instances of a given application may be active across cluster members, and cluster aliases provide a simple, reliable, and transparent mechanism for establishing client connections to those members that are hosting the target application.
- You want to take advantage of the cluster alias's ability to handle client network availability transparently.

Using an interface alias is the preferred mechanism when:

- You are running a single-instance service and one cluster member satisfies all client requests at any given time.
- Performance is critical; you want all clients to reach the one member that is providing the service, and you can never afford to take an extra routing hop.
- You are able to provide for client network availability on each cluster member that can host the service, by using a Redundant Array of Independent Network Adapters (NetRAIN) interface and by setting up a dependency on a client network resource in the application's CAA profile.

5.4.1.1 Cluster Alias

You must use cluster aliasing to provide client access to multi-instance network services. Because the TruCluster Server cluster alias subsystem creates and manages aliases on a clusterwide basis, you do not have to explicitly establish and remove interface aliases with the `ifconfig` command. See Chapter 3 for information about using cluster aliasing with multi-instance applications. See the TruCluster Server *Cluster Administration* manual for more information about how to use cluster aliasing in general.

You can use a cluster alias to direct client traffic to a single cluster member that is hosting a single-instance application, like the Oracle8i single server. When you configure a service under a cluster alias and set the `in_single` cluster alias attribute, the alias subsystem ensures that all client requests that are directed at that alias are routed to a cluster member that is running the requested service as long as it is available. However, for single-instance applications, consider using CAA for more control and flexibility in managing application failover. See Chapter 2 for information about using CAA.

Although you do not need to define NFS services in a TruCluster Server cluster to allow clients to access NFS file systems exported by the cluster, you may need to deal with the fact that clients know these services by the IP addresses that are provided by the ASE environment. Clients must access NFS file systems that are served by the cluster by using the default cluster alias.

5.4.1.2 Interface Alias

If a single-instance application cannot easily be converted to use a cluster alias, you can continue to use an interface alias by either modifying an existing `nfs_ifconfig` entry to use `ifconfig`(8), or add a call to `ifconfig` in a CAA action script.

When modifying a CAA action script, call `ifconfig alias` to assign an alias to an interface. Use the following command prior to starting up the application; otherwise, the application might not be able to bind to the alias address:

```
ifconfig interface_id alias alias_address netmask mask
```

To deassign an alias from an interface, call `ifconfig -alias` after all applications and processes have been stopped; otherwise, an application or process might not be able to continue to communicate with the interface alias.

The following example contains a section from an Apache CAA script:

```
.
.
.
# Assign an IP alias to a given interface
IFCNFG_ALIAS_ADD="/sbin/ifconfig tu0 alias 16.141.8.118 netmask 255.255.255.0"
#
#Deassign an IP alias to an interface
IFCNFG_ALIAS_DEL="/sbin/ifconfig tu0 -alias 16.141.8.118 netmask 255.255.255.0"
.
.
.

# Start section
#
case $1 in
'start')
        echo ""                                                >> ${LOG}
        echo ""Start action script for service : ${svcName} \
                `/bin/date +"%A %d %B %H:%M:%S"` ""            >> ${LOG}
#
# Start Apache
#
        echo "Starting Apache ... "                            >> ${LOG}
        cd $APPDIR
#
        $IFCNFG_ALIAS_ADD                                      >> ${LOG}
        if [ $? -ne 0 ]; then
                postevent "Apache IP Alias" start
           exit 2
        fi
#
        $START_APPCMD                                          >> ${LOG}
        if [ $? -ne 0 ]; then
                postevent "Apache Server" start
           exit 2
        fi
        echo "Started Apache Server"                           >> ${LOG}
;;
#
# Stop section
#
'stop')
        echo ""                                                >> ${LOG}
        echo ""Stop action script for service : ${svcName} \
                `/bin/date +"%A %d %B %H:%M:%S"` ""            >> ${LOG}
#
# Stop Apache
#
        echo "Stopping Apache ... "                            >> ${LOG}
        $STOP_APPCMD                                           >> ${LOG}
        if [ $? -ne 0 ]; then
                postevent "Apache Server" stop
           exit 2
        fi
#
        $IFCNFG_ALIAS_DEL                                      >> ${LOG}
        if [ $? -ne 0 ]; then
                postevent "Apache IP Alias" stop
           exit 2
        fi
;;
.
.
.

esac
```

5.4.2 Networking Services

In the TruCluster Available Server Software and TruCluster Production Server Software products, the `asemgr` utility provided a mechanism to monitor client networks.

In Tru64 UNIX Version 5.0 or later, client network monitoring is a feature of the base operating system. The NetRAIN interface provides protection against certain kinds of network connectivity failures. The Network Interface Failure Finder (NIFF) is an additional feature that monitors the status of its network interfaces and reports indications of network failures. Applications that are running in a TruCluster Server cluster can use the NetRAIN and NIFF features in conjunction with the Tru64 UNIX Event Manager (EVM) to monitor the health of client networks.

For more information about NetRAIN and NIFF, see the Tru64 UNIX *Network Administration: Connections* manual, `niffd`(8), `niff`(7), and `nr`(7).

5.5 File System Partitioning

CFS makes all files accessible to all cluster members. Each cluster member has the same access to a file, whether the file is stored on a device that is connected to all cluster members or on a device that is private to a single member. However, CFS does allow you to mount an AdvFS file system so that it is accessible to only a single cluster member. This is called **file system partitioning**.

ASE offered functionality like that of file system partitioning. File system partitioning is provided in TruCluster Server Version 5.1A to ease migration from ASE. See the TruCluster Server *Cluster Administration* manual for information on how to mount partitioned file systems and any known restrictions.

6 Moving Distributed Applications to TruCluster Server

A distributed application is an application that has already been modified for use in a cluster. It is cluster-aware; that is, it knows it is running in a cluster. Just running an application in a cluster does not make that application cluster-aware. Components of distributed applications often use the application programming interface (API) libraries that ship with the TruCluster Server product to communicate across member systems and coordinate their access to shared data.

The following subsystem APIs are fully compatible with those that were provided in earlier TruCluster products:

- Cluster alias (`clua_*.3`)
- Distributed lock manager (DLM) (`dlm_*.3`)
- Memory Channel (`imc_*.3`)

For more information about using the cluster alias, DLM, and Memory Channel APIs, see Chapter 8, Chapter 9, and Chapter 10, respectively.

This chapter discusses the following topics:

- Preparing to move distributed applications to TruCluster Server (Section 6.1)
- Creating Oracle Parallel Server on TruCluster Server (Section 6.2)
- Moving Oracle Parallel Server to TruCluster Server (Section 6.3)

6.1 Preparing to Move Distributed Applications to TruCluster Server

When preparing to move distributed applications to TruCluster Server, note the following:

- Device names may be hard coded in data files. You can rename the data files, create symbolic links, or re-create the database. Symbolic links are easiest to manage. Create soft links that point to the raw devices and update the permissions.

Note

While you can rename OPS data files, the renaming of Informix XPS data files is not supported; use symbolic links instead. See the *Informix Installation Guide* for more information.

- Do not edit control files. Save your original control files.
- Relink the Oracle binary code after you have created a single-member TruCluster Server cluster. This is due to the new entry points in the connection manager library for TruCluster Server.

6.2 Running Oracle Parallel Server on TruCluster Server

This section explains how to get the Oracle Parallel Server (OPS) option of Oracle 8i Release 3 (8.1.7) up and running in a TruCluster Server Version 5.1A cluster. Oracle8i 8.1.7 takes advantage of the direct I/O feature introduced in TruCluster Server Version 5.1 to allow you to configure OPS on a TruCluster Server Version 5.1A or later Cluster File System. Previous versions of Oracle require the use of raw disk partitions or volumes in a cluster. See the TruCluster Server *Cluster Administration* manual for a discussion of direct I/O in a cluster.

To run OPS on TruCluster Server, follow these steps:

1. Install and configure Oracle8i Release 3 (8.1.7) using the instructions in the Oracle8i Release 3 (8.1.7) documentation. You only need to install Oracle8i on one cluster member.

   Oracle has special requirements, including that certain kernel attributes be set to specific values, that specific UNIX groups (`dba`, `oinstall`) be created, and that special environment variables be initialized.

   Configure the Oracle Parallel Server option, using the instructions in the Oracle8i Release 3 (8.1.7) documentation.

2. On the Tru64 UNIX Version 5.1A system, enter the following `hwmgr` command to locate the physical device of each distributed raw disk (DRD) service:

   ```
   # hwmgr -view devices -member provolone
   ```

   For example, entering the `hwmgr` command on the `provolone` cluster member produces the following:

   ```
   HWID:   DSF Name              Model          Location
   ----------------------------------------------------------------------
   36:  /dev/disk/dsk3c    RZ28     bus-1-targ-1-lun-0  provolone.zk4.com
   37:  /dev/disk/dsk4c    RZ28D    bus-1-targ-2-lun-0  provolone.zk4.com
   38:  /dev/disk/dsk5c    RZ29B    bus-1-targ-3-lun-0  provolone.zk4.com
   39:  /dev/disk/dsk6c    RZ28D    bus-1-targ-4-lun-0  provolone.zk4.com
   ```

```
40:  /dev/disk/dsk7c    RZ28L-AS bus-1-targ-5-lun-0  provolone.zk4.com
63:  /dev/disk/floppy1c 3.5in    fdi0-unit-0         provolone.zk4.com
73:  /dev/disk/dsk8c    RZ1CB-CS bus-0-targ-0-lun-0  provolone.zk4.com
74:  /dev/disk/cdrom1c  RRD45    bus-0-targ-4-lun-0  provolone.zk4.com
  .
  .
  .
```

3. Create soft links that point to the raw devices and update the permissions. Do *not* edit the control file to remove DRD names.

4. Configure the Net8 listener to use a cluster alias for load balancing of client requests. You can also use the multi-threaded server (MTS) capability of OPS to provide load balancing of client requests. See the Oracle8i documentation.

   To use a cluster alias, add the following line to the `/etc/clua_services` file to set up the properties of the port that the Oracle8i listener uses:

   ```
   listener            1521/tcp            in_multi
   ```

   Setting the `in_multi` attribute for port 1521 means that the cluster alias subsystem will distribute connection requests directed to a cluster alias among all members of the alias.

5. Reload the cluster alias service definitions by entering the following command on each cluster member:

   ```
   # cluamgr -f
   ```

6. After you have set up OPS within the cluster, and have verified that it can be accessed by both local and remote clients, you must ensure that each member, when it is booted, starts its database instance and, when it is shut down, stops its database instance. The recommended method involves placing a script in the `/sbin/init.d`

   You can also create a cluster application availability (CAA) action script to automatically start and stop the database instance, but you will have to restrict OPS to one member of the cluster. For more information about starting and stopping OPS, see the Oracle8i Release 3 (8.1.7) documentation.

6.3 Moving Oracle Parallel Server to TruCluster Server

To move OPS to TruCluster Server, note the following:

- If you are moving from TruCluster Software Products Version 1.6 or earlier, pay careful attention to device names. There are no special device names for DRD-managed storage. If your database references `/dev/drd` or `/dev/rdrd`, create symbolic links, as described in step 3 of Section 6.2, to point to the new devices.

- OPS allows renaming of datafiles. If you prefer to rename datafiles instead of creating symbolic links, follow these steps:

  1. Save your original control files. Do not edit control files.
  2. Start up the database using the `nomount` option.
  3. Rename the datafile to a generic file name using the following command:

     ```
     SVRMGR> alter database rename file ...
     ```

  4. After renaming the datafile, use the following command to validate the database. This guarantees that your database is consistent and detects any incorrect symbolic links.

     ```
     SVRMGR> analyze table validate structure cascade
     ```

Part 3

Writing Cluster-Aware Applications

7 Programming Considerations

This chapter describes programming changes you can make to an application's source code to allow it to run in a cluster. You must have access to the application's source files to make the required changes.

This chapter discusses the following topics:

- Modifications that are required for remote procedure call (RPC) programs (Section 7.1)
- Portable applications — developing applications that run in a cluster or on a standalone system (Section 7.2)
- Support for the Cluster Logical Storage Manager (CLSM) (Section 7.3)
- Diagnostic utility support (Section 7.4)
- Compact Disc-Read Only Memory File System (CDFS) file system restrictions (Section 7.5)
- Scripts called from the `/cluster/admin/run` directory (Section 7.6)
- Cluster member status during a rolling upgrade (Section 7.7)
- File access resilience in a cluster (Section 7.8)

7.1 Modifications Required for RPC Programs

Make the following modifications to existing, nonclusterized remote procedure call (RPC) programs to allow them to run in a clustered environment:

- Conditionalize the code to replace calls to `bind()` with calls to `clusvc_getcommport()` or `clusvc_getresvcommport()` when the code is executed in a cluster environment. Use these functions only if you want to run an RPC application on multiple cluster members, making them accessible via a cluster alias. In addition to ensuring that each instance of an RPC application uses the same common port, these functions also inform the portmapper that the application is a multi-instance, alias application. See `clusvc_getcommport(3)` and `clusvc_getresvcommport(3)` for more information.
- Services that are not calling `svc_register()` must call `clua_registerservice()` to allow the service to accept incoming cluster alias connections. See `clua_registerservice(3)` for more information.

(Note that `clusvc_getcommport()` and `clusvc_getresvcommport()` automatically call `clua_registerservice()`.)

7.2 Portable Applications: Standalone and Cluster

Tru64 UNIX Version 5.0 or later provides the following built-in features that make it easier to develop applications that run either in a cluster or on a standalone system:

- Stub libraries in Tru64 UNIX Version 5.0 or later let you build applications that call functions in the `libclu.so` API library that ships with TruCluster Server.
- The `clu_is_member()` function, which is provided in `libc`, determines whether the local system is a cluster member. If the local system is a cluster member, the function returns TRUE; otherwise, it returns FALSE.
- The `clu_is_ready()` function, which is also provided in `libc`, determines whether the local system has been configured to run in a cluster (that is, TruCluster Server Software is installed and the system is running a clusterized kernel). If the local system is configured to run in a cluster, the function returns TRUE; otherwise, it returns FALSE. The `clu_is_ready()` function is most useful in code that runs in the boot path before the connection manager establishes cluster membership.
- The `clu_info()` function and the `clu_get_info()` command return information about the configuration or a value indicating that the system is not configured to be in a cluster.

For more information, see `clu_info`(3), `clu_is_member`(3), and `clu_get_info`(8).

7.3 CLSM Support

The Cluster Logical Storage Manager (CLSM) does not provide interlocking support for normal I/O on mirrored volumes between different nodes. CLSM assumes any application that simultaneously opens the same volume from different nodes already performs the necessary locking to prevent two nodes from writing to the same block at the same time. In other words, if a cluster-aware application is not well-behaved and issues simultaneous writes to the same block from different nodes on a CLSM mirrored volume, data integrity will be compromised.

This is not an issue with Oracle Parallel Server (OPS) because OPS uses the distributed lock manager (DLM) to prevent this situation. Also, it is not an issue with the Cluster File System (CFS), because only one node can have a file system mounted at a time.

While steady-state I/O to mirrored volumes is not interlocked by CLSM between different nodes, CLSM provides interlocking between nodes to accomplish mirror recovery and plex attach operations.

7.4 Diagnostic Utility Support

If you have, or want to write, a diagnostic utility for an application or subsystem, the TruCluster Server `clu_check_config` command calls diagnostic utilities, provides an execution environment, and maintains log files.

See `clu_check_config`(8) for a description of how to add a diagnostic utility to the cluster environment and have it called by the `clu_check_config` command.

7.5 CDFS File System Restrictions

In TruCluster Server Version 5.1A, there are restrictions on managing Compact Disc-Read Only Memory File System (CDFS) file systems in a cluster. Some commands and library functions behave differently on a cluster than on a standalone system.

Table 7–1 lists the CDFS library functions and their expected behavior in a TruCluster Server environment.

Table 7–1: CDFS Library Functions

Library Function	Expected Result on Server	Expected Result on Client
`cd_drec`	Success	Not supported
`cd_ptrec`	Success	Not supported
`cd_pvd`	Success	Not supported
`cd_suf`	Success	Not supported
`cd_type`	Success	Not supported
`cd_xar`	Success	Not supported
`cd_nmconv` `CD_GETNMCONV`	Success	Success
`cd_nmconv` `CD_SETNMCONV`	Success	Success
`cd_getdevmap`	No map	Not supported
`cd_setdevmap`	Not supported	Not supported

Table 7–1: CDFS Library Functions (cont.)

Library Function	Expected Result on Server	Expected Result on Client
`cd_idmap` `CD_GETUMAP` `CD_GETGMAP`	Success	Not supported
`cd_idmap` `CD_SETUMAP` `CD_SETGMAP`	Success	Success
`cd_defs` `CD_GETDEFS`	Success	Success
`cd_defs` `CD_SETDEFS`	Success	Success

For information about managing CDFS file systems in a cluster, see the TruCluster Server *Cluster Administration* manual.

7.6 Scripts Called from the /cluster/admin/run Directory

An application that needs to have specific actions taken on its behalf when a cluster is created, or when members are added or deleted, can place a script in the `/cluster/admin/run` directory. These scripts are called during the first boot of the initial cluster member following the running of `clu_create`, each cluster member (including the newest one) following `clu_add_member`, and all remaining cluster members following `clu_delete_member`.

The scripts in `/cluster/admin/run` must use the following entry points:

- `-c`

  For actions to take when `clu_create` runs.
- `-a`

  For actions to take when `clu_add_member` runs.
- `-d` *memberid*

  For actions to take when `clu_delete_member` runs.

Place only files or symbolic links to files that are executable by `root` in the `/cluster/admin/run` directory. We recommend that you adhere to the following file-naming convention:

- Begin the executable file name with an uppercase letter `C`.
- Make the next two characters a sequential number as used in `/sbin/rc3.d` for your area.
- For the remaining characters, use a name that is associated with your script.

The following file name is an example of this naming convention:

```
/cluster/admin/run/C40sendmail
```

The `clu_create`, `clu_add_member`, and `clu_delete_member` commands create the required `it`(8) files and links to ensure that the scripts are run at the correct time.

7.7 Testing the Status of a Cluster Member During a Rolling Upgrade

The following example program shows one way to determine whether the cluster is in the middle of a rolling upgrade, and whether this cluster member has rolled:

```
#include <stdio.h>
#include <sys/clu.h> /* compile with -lclu */

#define DEBUG 1

main()
{
  struct clu_gen_info  *clu_gen_ptr = NULL;
  char cmd[256];

  if (clu_is_member()) {
    if (clu_get_info(&clu_gen_ptr) == 0) {
      if(system("/usr/sbin/clu_upgrade -q status") == 0) {
      sprintf(cmd, "/usr/sbin/clu_upgrade -q check roll %d",
            clu_gen_ptr->my_memberid);
      if (system(cmd) == 0) {
        if (DEBUG) printf("member has rolled\n");
      }
      else if (DEBUG) printf("member has not rolled\n");
      }
      else if (DEBUG) printf("no rolling upgrade in progress\n");
   }
   else if (DEBUG) printf("nonzero return from clu_get_info(\n");
 }
 else if (DEBUG) printf("not a member of a cluster\n");
}
```

7.8 File Access Resilience in a Cluster

While a cluster application is transferring files, read and write operations may fail if the member running the application is shut down or fails. Typically, a client of the application will see the connection to the server as lost. Be aware of how your application handles lost connections. Some ways applications handle lost connections are:

- The client application simply fails (for example, an error is written and the application exits).
- The client application sees a problem with the connection and automatically retries its read or write operation.

- The client application sees a problem with the connection and displays a window that allows the user to abort, retry, or cancel the operation.

If your client application fails when it loses its connection to the server application (regardless of whether it is running on a single system or a cluster), consider implementing the following:

- When updating files, first write the update to a new temporary file. If the write operation is successful, copy the temporary file over the original file. If the write operation encountered a problem, you have not destroyed the original file. You need only to clean up your temporary files and start again.
- When reading files, make sure that your application is set up to deal with read operation errors and recover from them (for example, retry the operation).

8 Cluster Alias Application Programming Interface

This chapter discusses the following topics:

- Cluster alias port terminology (Section 8.1)
- Cluster alias functions (Section 8.2)
- Cluster port space (Section 8.3)
- Information for multi-instance services that bind to reserved ports (Section 8.4)
- Cluster alias `setsockopt()` options (Section 8.5)
- Port attributes: the relationship between the port attributes set by `/etc/clua_services`, `clua_registerservice()`, and the `setsockopt()` cluster alias socket options (Section 8.6)
- UDP applications and source addresses (Section 8.7)

8.1 Cluster Alias Port Terminology

The following port-related terms are used in this chapter:

well-known port	A port whose number is known by both the client and server. For example, those ports whose numbers are listed in `/etc/services`. The server explicitly binds to and listens on its well-known port because clients know the port number and expect to connect to the server via its well-known port.
dynamic port	The inverse of a well-known port. The application does not explicitly specify a port number; it is assigned the first open port that is found by the operating system.
ephemeral port	A dynamic port within the ephemeral port space (ports with numbers greater than 1024, or more explicitly, between `IPPORT_RESERVED` and `IPPORT_USERRESERVED`).

locked port	A port that is dedicated to an individual cluster member and is not available clusterwide. When a port is locked, an attempt by an application to bind to the port fails with `EADDRINUSE`, unless the application attempting to bind sets the `SO_REUSEALIASPORT` option on the socket.
reserved port	In the cluster alias subsystem, a port whose number is greater than 512 and less than `IPPORT_RESERVED` (1024). The default cluster behavior is that a bind to a reserved port automatically locks the port. (Ports with numbers less than or equal to 512 are never locked.)

8.2 Cluster Alias Functions

The cluster alias library, `libclua.a` and `libclua.so`, provides functions for getting and setting cluster alias subsystem attributes. Table 8–1 lists the functions that the cluster alias library provides. See the specific section 3 reference pages for more information.

Table 8–1: Cluster Alias Functions

Function	Description
`clua_error`	Map a cluster alias message ID to a printable character string, returning the string to the caller.
`clua_getaliasaddress`	Get the IP address of one cluster alias known to the local node.
`clua_getaliasinfo`	Get information about one cluster alias and its members.
`clua_getdefaultalias`	Get the IP address of the default cluster alias.
`clua_isalias`	Determine whether an IP address is that of a cluster alias.
`clua_registerservice`	Register a dynamic port as eligible to receive incoming connections.
`clua_unregisterservice`	Release a port.
`clusvc_getcommport`	Bind to a common port within a cluster.
`clusvc_getresvcommport`	Bind to a reserved port within a cluster.
`print_clua_liberror`	Map a cluster alias message ID to a printable character string, returning the string to `stderr`.

Use the following `#include` files and libraries when writing and compiling programs that use cluster alias functions:

- Programs that use the clua functions #include <clua/clua.h> and are compiled with -lclua.
- Programs that use the clua_getaliasaddress(), clua_getaliasinfo(), clua_getdefaultalias(), clua_isalias(), clua_registerservice(), or clua_unregisterservice() functions are compiled with -lclua -lcfg.
- Programs that use the clusvc functions #include <netinet/in.h>, and are compiled with -lclu.

The following list describes the cluster alias functions in more detail:

clua_error() and print_clua_liberror()

The clua_error() and print_clua_liberror() functions map a cluster alias message ID to a printable character string. The clua_error() function returns the string to the caller. The print_clua_liberror() function prints the string to stderr. See clua_error(3) for more information.

clua_getaliasaddress() and clua_getaliasinfo()

A call to the clua_getaliasaddress function gets the IP address of one cluster alias known to the local node. Each subsequent call returns another alias IP address. When the list of aliases that are known to the node is exhausted, the function returns CLUA_NOMOREALIASES.

When it is passed the address of an alias in a sockaddr structure, clua_getaliasinfo() populates a clu_info structure with information about that alias.

Programs usually make iterative calls to clua_getaliasaddress and pass each alias address to clua_getaliasinfo to get information about that alias. The following example shows this iterative loop (wrapped inside a short main() program with some calls to printf()):

```
/*  compile with -lclua -lcfg */
#include <sys/socket.h> /* AF_INET */
#include <clua/clua.h>  /* includes <netinet/in.h> */
#include <netdb.h>      /* gethostbyaddr() */
#include <arpa/inet.h>  /* inet_ntoa() */

main ()
{
  int context = 0;
  struct sockaddr addr;
  struct clua_info outbuf, *pout;
  clua_status_t result1, result2;
  struct hostent *hp;
  pout=&outbuf;

  while ((result1=clua_getaliasaddress(&addr,
                                       &context)) == CLUA_SUCCESS)
   {
     if ((result2=clua_getaliasinfo(&addr, pout)) == CLUA_SUCCESS) {
      hp = gethostbyaddr((const void *)&pout->addr,
                            sizeof (struct in_addr), AF_INET);
        printf ("\nCluster alias name:\t\t %s\n", hp->h_name);
        printf ("Cluster alias IP address:\t %s\n",
inet_ntoa(pout->addr)));
        printf ("Cluster alias ID (aliasid):\t %d\n", pout->aliasid);
        printf ("Connections rcvd from net:\t %d\n",
pout->count_cnx_rcv_net);
        printf ("Connections forwarded:\t\t %d\n",
pout->count_cnx_fwd_clu);
        printf ("Connections rcvd within cluster: %d\n",
                pout->count_cnx_rcv_clu);
     } else {
        print_clua_liberror(result2);
        break;
     }
   }
  if (result1 != CLUA_SUCCESS && result1 != CLUA_NOMOREALIASES)
   print_clua_liberror(result1);
}
```

See `clua_getaliasaddress`(3) and `clua_getaliasinfo`(3) for more information.

`clua_getdefaultalias()`

Where the `clua_getaliasaddress()` function can iteratively return the IP addresses of all cluster aliases that are known to the local node, the `clua_getdefaultalias()` function returns only the address of the default cluster alias. See `clua_getdefaultalias`(3) for more information.

`clua_isalias()`

When it is passed an IP address, the `clua_isalias()` function determines whether the address is that of a cluster alias. See `clua_isalias`(3) for more information.

`clua_registerservice()` and `clua_unregisterservice()`

The `clua_registerservice()` function registers a dynamic port as eligible to receive incoming connections. For ports in the range 512-1024, use the `CLUSRV_STATIC` option. Otherwise, the port is reserved clusterwide by the first node to bind to the port and the remaining cluster members will not be able to bind to the port.

The `clua_unregisterservice()` function releases a port.

See `clua_registerservice`(3) for more information.

`clusvc_getcommport()` and `clusvc_getresvcommport()`

Programs that use RPC can call the `clusvc_getcommport()` and `clusvc_getresvcommport()` functions to bind to a common port within a cluster. Use `clusvc_getresvcommport()` when binding to a reserved (privileged) port, a port number in the range 0-1024. See Section 8.4 for information on binding to reserved ports. The following example shows a typical calling sequence (wrapped inside a short `main()` program with some calls to `printf()`):

```
/* compile with -lclu */
#include <rpc/rpc.h>   /* includes <netinet/in.h> */
#include <syslog.h>    /* LOG_ERR */
#include <unistd.h>    /* gethostname() */
#include <sys/param.h> /* MAXHOSTLEN */

main () {
  int s, i, namelen;
  int cluster = 0;
  uint prog = 100999;  /* replace with real program number */
  struct sockaddr_in addr;
  int len = sizeof(struct sockaddr_in);
  char local_host[MAXHOSTNAMELEN +1];

  gethostname (local_host, sizeof (local_host) - 1);
  cluster = clu_is_member();
  printf ("\nSystem %s %s a cluster member\n",
          local_host, cluster?"is":"is not");

  if ((s = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0) {
    syslog(LOG_ERR, "socket: %m");
    exit(1);
  }

  bzero(&addr, sizeof(addr));
  addr.sin_family = AF_INET;
  addr.sin_addr.s_addr = INADDR_ANY;

  if (cluster) {
    if (clusvc_getcommport(s, prog, IPPROTO_UDP, &addr) < 0) {
      syslog(LOG_ERR, "clusvc_getcommport: %m");
      exit(1);
    }
  } else {
    addr.sin_port = 0;
    if (bind(s, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
      syslog(LOG_ERR, "bind: %m");
      exit(1);
    }
    if (getsockname(s, (struct sockaddr *)&addr, &len) != 0) {
      syslog(LOG_ERR, "getsockname: %m");
      (void) close(s);
      exit(1);
    }
  }
  printf ("  addr.sin_family: %d\n", addr.sin_family);
  printf ("  addr.sin_port:   %u\n", addr.sin_port);
  printf ("  addr.sin_addr:   %x\n", addr.sin_addr);
  printf ("  addr.sin_zero:   ");
  for (i = 0; i<8;i++)
    printf("%d ", (int)addr.sin_zero[i]);
  putchar('\n');
}
```

8.3 Cluster Port Space

The cluster emulates single-system port semantics by using the same port space across the cluster. The following list provides a brief overview of how the cluster alias subsystem handles port space.

- Ports less than 512 (0 < (`IPPORT_RESERVED`[1] / 2): These well-known ports are never locked. In a single system, more than one process cannot bind to the same port without using the `SO_REUSEPORT` socket option. In a cluster, a port whose number is less than 512 is never locked in order to allow one process on each cluster member to bind to the port.
- Ports from 512 through 1024 ((`IPPORT_RESERVED` / 2) to `IPPORT_RESERVED`): If there is an explicit bind to the port, a reserved port is locked unless the port is marked as `static`. See Section 8.4 for information on binding to reserved ports.
- Ports greater than 1024 (`IPPORT_RESERVED` to `IPPORT_USERRESERVED`): An ephemeral port is locked if `sin_port=0`. An ephemeral port is not locked if there is an explicit bind to the port.

The port space has the following implications for single-instance and multi-instance applications. In general:

- Make sure that single-instance applications use a locked port because only one instance of the application should be handling network requests. Therefore, you can use one of the following approaches:
  - Explicitly bind to a reserved port (512 through 1024).
  - Use an ephemeral port (`sin.port=0`). (This locks the port, but has the problem that the client must somehow be notified of the port number.)
  - Use CAA to make sure that only one instance of the application is running in the cluster.
- A multi-instance application does not want to restrict access to the first instance of the application to bind to a port. Therefore, you can use one of the following approaches:
  - Use a port that is less than 512.
  - Use a reserved port (512 through 1024) and set `SO_REUSEALIASPORT`.
  - Do an explicit bind to an ephemeral port.
  - Use an ephemeral port (`sin_port=0`) and set `SO_REUSEALIASPORT`. (This keeps the port from being locked, but still has the problem that the client must somehow be notified of the port number.)

[1] `IPPORT_RESERVED` and `IPPORT_USERRESERVED` are defined in `<netinet/in.h>`.

8.4 Binding to Reserved Ports (512 Through 1024)

The following information is useful background for multi-instance services binding to reserved ports.

Reserved ports receive special treatment because they can be used as either well-known ports or dynamic ports. By default, if a process explicitly binds to a port in the range 512-1024, the port is assumed to be dynamic: it is reserved clusterwide (locked), and binds on other members will fail. This is primarily done because many programs call `bind()` looking for free ports in this range, expecting to fail when a port is in use.

The cluster alias subsystem treats reserved ports differently for the following reasons:

- There is no way to differentiate between an application that is searching for a dynamic reserved port and one that knows which well-known port it wants.
- Dynamic ports are sometimes used as the local port for an outbound connection, which are identified only by port/address pairs. If a dynamic port is used for an outbound connection and if the address is that of a cluster alias, the alias subsystem cannot allow multiple nodes to have the same port. If that were allowed, the alias subsystem would not be able to distinguish between, for example, connection 1000/alias from different nodes in the cluster. Therefore, if the alias subsystem does not know that a reserved port will be `static` (available clusterwide), it must lock it down.

Ports above 1024 do not encounter this because there is a standard way to allocate a dynamic port: bind to port 0 (`sin_port=0`) and the system picks a port from the ephemeral port space. Therefore, the cluster alias subsystem assumes that a process that is explicitly binding to a port below 512 or above 1024 knows what it is doing and is getting a well-known port.

Make sure that multi-instance services that use well-known ports in the 512-1024 range for inbound connections register that port as `static` with the cluster alias subsystem. To register a port as static, either put an entry for the port in `/etc/clua_services` or modify the program to call `clua_registerservice` with the `CLUASRV_STATIC` option. See `clua_registerservice`(3) and `clua_services`(4) for more information about these functions. (The `static` option also serves another purpose for ports greater than 1024: it directs the system to leave a port designated as `static` out of the ephemeral port space.)

Note

Only specify the `static` attribute in `/etc/clua_services` for multi-instance services that are started every time a cluster member boots. For example, services with entries in `/etc/inetd.conf`. Otherwise, different applications on different cluster members that are looking for a dynamic port might bind to the same port.

8.5 setsockopt() Options

The `setsockopt`() and `getsockopt`() system calls support the following cluster alias socket options:

`SO_CLUA_DEFAULT_SRC`

If the local address has not already been set through a call to `bind`(), the socket uses the default cluster alias as its source address.

`SO_CLUA_IN_NOALIAS`

Attempts to bind the socket to a cluster alias address will fail. Use this option for services that you do not want to be accessed using a cluster alias.

A bind to an dynamic port (greater than or equal to `IPPORT_RESERVED` and less than `IPPORT_USERRESERVED`) does not result in that port being locked.

A bind to a reserved port with a wildcard address (`INADDR_ANY` or `in6addr_any`) does not result in that port being locked.

The source address for outgoing UDP sends or TCP connection requests is a local host address (never a cluster alias address).

The `SO_CLUA_IN_NOLOCAL` and `SO_CLUA_IN_NOALIAS` options are mutually exclusive.

`SO_CLUA_IN_NOLOCAL`

The socket must receive packets that are addressed to a cluster alias and will drop any packets that are not addressed to a cluster alias. Use this option for services that you want only to be accessed using a cluster alias.

The `SO_CLUA_IN_NOLOCAL` and `SO_CLUA_IN_NOALIAS` options are mutually exclusive.

`SO_RESVPORT`

An attempt to bind the socket to a port in the reserved range (512-1024) will fail if the port is marked static, either by a `static` entry for the port in `/etc/clua_services` or through a call to `clua_registerservice`() with the `CLUASRV_STATIC` option. The call to `bind`() returns `EADDRINUSE`.

SO_REUSEALIASPORT

The socket can reuse a locked cluster alias port. When this option is set, a `bind()` is distributed clusterwide. (A distributed application can use this side effect to determine whether or not a port is in use.)

8.6 Port Attributes: /etc/clua_services, clua_registerservice(), and setsockopt()

The strings that are used in the `/etc/clua_services` file have a one-to-one mapping with the `CLUASRV_*` options that are used by `clua_registerservice()`. Some of these strings/options also have a relationship with the cluster alias `setsockopt()` options. Table 8–2 lists these relationships.

Table 8–2: Relationship Among Cluster Alias Port Attributes

clua_services	clua_registerservice()	setsockopt()
in_multi	CLUASRV_MULTI	
in_single	CLUASRV_SINGLE	
in_noalias	CLUASRV_IN_NOALIAS	SO_CLUA_IN_NOALIAS
out_alias	CLUASRV_OUT[a]	SO_CLUA_DEFAULT_SRC [b]
in_nolocal	CLUASRV_IN_NOLOCAL	SO_CLUA_IN_NOLOCAL
static	CLUASRV_STATIC	
		SO_REUSEALIASPORT
		SO_RESVPORT

[a] The `CLUASRV_OUT` option forces the default cluster alias as the source address only if the destination port for the service has the `out_alias` attribute set in the `clua_services` file.
[b] The `SO_CLUA_DEFAULT_SRC` option does not examine the attributes that are associated with the destination port. (Neither `CLUASRV_OUT` nor `SO_CLUA_DEFAULT_SRC` override the manual setting of an address. These two options set the source address to the default cluster alias only if the address is not yet set.)

8.7 UDP Applications and Source Addresses

Some User Datagram Protocol (UDP)-based applications require that messages from the server to the client come from the same server address that is used by the client when sending the UDP request. Because systems in a cluster do not normally use a cluster alias as the source address for outbound UDP messages, these applications may have problems using cluster aliases.

This section describes how to make a UDP-based server application in a cluster respond with the address that was used to reach it (whether the address is a cluster alias or a local address) as follows:

- The preferred approach is to bind a socket to each local IP address, plus an additional socket to the default cluster alias address. (Use `clua_getdefaultalias` to get the alias's address.) Respond to requests using the same socket that received the request; the address will automatically be the one used by the client. This approach replaces using one `listen` on the wildcard.
- A somewhat simpler approach is to use two sockets: one listening on the wildcard and one on the default cluster alias. (As in the previous example, reuse the incoming socket for outbound packets.) If there are multiple local addresses in the same subnet, this approach does not always supply the correct local address, but handles the default cluster alias address properly. If there are multiple cluster aliases and you want the application to respond with the right one (the one used for input), use the multiple sockets approach, with a separate `listen()` on each cluster alias. Use `clua_getaliasaddress()` to create a list of defined cluster aliases.

Note

If you want the application to be accessed only using a cluster alias and you want the application to always respond using the default cluster alias, listen on the wildcard with a single socket but set the socket with the `SO_CLUA_DEFAULT_SRC` option. The application will always use the default cluster alias address for outbound traffic.

9 Distributed Lock Manager

The Cluster File System (CFS) is fully POSIX compliant, so an application can use `flock` system calls to synchronize access to shared files among instances. Alternatively, applications can use the functions that are supplied by the cluster distributed lock manager (DLM).

The DLM provides features beyond those supplied by traditional POSIX file locks. These include:

- A larger set of lock modes, beyond exclusive and shared locks.
- Asynchronous notification and callback to a lock holder when the holder is blocking the granting of the lock to another process.
- Asynchronous notification and callback to a lock waiter when the requested lock is granted.
- The ability to request and wait for conversion of existing locks to higher or lower lock modes.
- Independent namespaces in which locks exist, and various security mechanisms protecting access to these namespaces.
- A value block within the lock structure that processes sharing a resource can use to communicate and coordinate their actions.
- Sophisticated deadlock detection mechanisms.

Note

The DLM applications programming interface (API) library is supplied with the TruCluster Server cluster software and not the Tru64 UNIX base operating system. For this reason, make sure that code intended to run on both standalone systems and in a cluster, and calls DLM functions, uses `clu_is_member` to determine whether the application is running in a cluster.

This chapter describes how to use DLM to synchronize access to shared resources in a cluster. It discusses the following topics:

- How the DLM synchronizes the accesses of multiple processes to a specific resource (Section 9.1).

- The concepts of resources, resource granularity, namespaces, resource names, and lock groups (Section 9.2).
- The concepts of locks, lock modes, lock compatibility, lock management queues, lock conversions, and deadlock detection (Section 9.3).
- How to use the `dlm_unlock` function to dequeue lock requests (Section 9.4).
- How to use the `dlm_cancel` function to cancel lock conversion requests (Section 9.5).
- Specialized locking techniques, such as lock request completion, the expediting of lock requests, blocking notifications, lock conversions, parent locks and sublocks, and lock value blocks (Section 9.6).
- How applications can perform local buffer caching (Section 9.7).
- A code example showing the basic DLM operations (Section 9.8).

9.1 DLM Overview

The distributed lock manager (DLM) provides functions that allow cooperating processes in a cluster to synchronize access to a shared resource, such as a raw disk device, a file, or a program. For the DLM to effectively synchronize access to a shared resource, all processes in the cluster that share the resource must use DLM functions to control access to the resource.

DLM functions allow callers to:

- Request a new lock on a resource
- Release a lock or group of locks
- Convert the mode of an existing lock
- Cancel a lock conversion request
- Wait for a lock request to be granted, or continue operation and be notified asynchronously of the request's completion
- Receive asynchronous notification when a lock granted to the caller is blocking another lock request

Table 9–1 lists the functions the DLM provides. These functions are available in the `libdlm` library for use by applications.

Table 9–1: Distributed Lock Manager Functions

Function	Description
`dlm_cancel`	Cancels a lock conversion request
`dlm_cvt`	Synchronously converts an existing lock to a new mode

Table 9–1: Distributed Lock Manager Functions (cont.)

Function	Description
`dlm_detach`	Detaches a process from all namespaces
`dlm_get_lkinfo`	Obtains information about a lock request associated with a given process
`dlm_get_rsbinfo`	Obtains locking information about resources managed by the DLM
`dlm_glc_attach`	Attaches to an existing process lock group
`dlm_glc_create`	Creates a group lock container
`dlm_glc_destroy`	Destroys a group lock container
`dlm_glc_detach`	Detaches from a process lock group
`dlm_lock`	Synchronously requests a lock on a named resource
`dlm_locktp`	Synchronously requests a lock on a named resource, using group locks or transaction IDs
`dlm_notify`	Polls for outstanding completion and blocking notifications
`dlm_nsjoin`	Joins the specified namespace
`dlm_nsleave`	Leaves the specified namespace
`dlm_perrno`	Prints the message text associated with a given DLM message ID
`dlm_perror`	Prints the message text associated with a given DLM message ID, plus a caller-specified message string
`dlm_quecvt`	Asynchronously converts an existing lock to a new mode
`dlm_quelock`	Asynchronously requests a lock on a named resource
`dlm_quelocktp`	Asynchronously requests a lock on a named resource, using group locks or transaction IDs
`dlm_rd_attach`	Attaches a process or process lock group to a recovery domain
`dlm_rd_collect`	Initiates the recovery procedure for a specified recovery domain by collecting those locks on resources in the domain that have invalid lock value blocks
`dlm_rd_detach`	Detaches a process or process lock group from a recovery domain
`dlm_rd_validate`	Completes the recovery procedure for a specified recovery domain by validating the resources in the specified recovery domain collection
`dlm_set_signal`	Specifies the signal to be used for completion and blocking notifications

Table 9–1: Distributed Lock Manager Functions (cont.)

Function	Description
`dlm_sperrno`	Obtains the character string associated with a given DLM message ID and stores it in a variable
`dlm_unlock`	Releases a lock

The DLM itself does not ensure proper access to a resource. Rather, the processes that are accessing a resource agree to access the resource cooperatively, use DLM functions when doing so, and respect the rules for using the lock manager. These rules are as follows:

- All processes must always refer to the resource by the same name. The name must be unique within a given namespace.
- For user ID and group ID qualified namespaces, the protections and ownership (that is, the user IDs and group IDs) that are employed within the namespace must be consistent throughout the cluster. A public namespace has no such access restrictions. (See Section 9.2.2 for more information.)
- Before accessing a resource, all processes must acquire a lock on the resource by queuing a lock request. Use the `dlm_lock`, `dlm_locktp`, `dlm_quelock`, and `dlm_quelocktp` functions for this purpose.

Because locks are owned by processes, applications that use the DLM must take into account the following points:

- Because the DLM delivers signals, completion notifications, and blocking notifications to the process, avoid using the DLM API functions in a threaded application. Most signals are delivered to an arbitrary thread within a multithreaded process. Any signal that is defined as the DLM notification signal (by `dlm_set_signal`) can be delivered to a thread that is not the one waiting for the lock grant or blocking notification (and thus never be delivered to the one that is waiting).
- When a process forks, the child process does not inherit its parent's lock ownership or namespace attachment. Before accessing a shared resource, the child process must attach to the namespace that includes the resource and acquire any needed locks.
- Because the DLM maintains process-specific information (such as the process-space addresses of the blocking and completion routines), a call to the `exec` routine invalidates this information and results in unpredictable behavior. Before issuing a call to the `exec` routine, a process must release its locks using the `dlm_unlock` and `dlm_detach` functions. If the process does not call these functions, the DLM causes the call to the `exec` routine to fail.

9.2 Resources

A resource can be any entity in a cluster (for example, a file, a data structure, a raw disk device, a database, or an executable program). When two or more processes access the same resource concurrently, they must often synchronize their access to the resource to obtain correct results.

The lock management functions allow processes to associate a name or binary data with a resource and to synchronize access to that resource. Without synchronization, if one process is reading the resource while another is writing new data, the writer can quickly invalidate anything that is being read by the reader.

From the viewpoint of the DLM, a resource is created when a process (or a process on behalf of a DLM process group) first requests a lock on the resource's name. At that point, the DLM creates the structure that contains, among other things, the resource's lock queues and its lock value block.

As long as at least one process owns a lock on the resource, the resource continues to exist. After the last lock on the resource is dequeued, the DLM can delete the resource. Normally, a lock is dequeued by a call to the `dlm_unlock` function, but a lock (and potentially a resource as well) can be freed abnormally if the process exits unexpectedly.

9.2.1 Resource Granularity

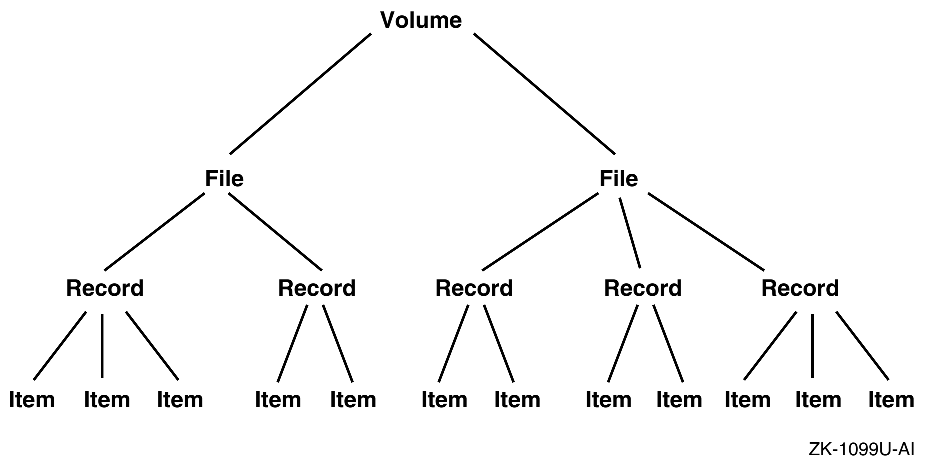
Many resources can be divided into smaller parts. As long as a part of a resource can be identified by a resource name, the part can be locked.

Figure 9–1 shows a model of a database. The database is divided into volumes, which in turn are subdivided into files. Files are further divided into records, and the records are further divided into items.

The processes that request locks on the database that are shown in Figure 9–1 can lock the whole database, a volume in the database, a file, a record, or a single item. Locking the entire database is considered locking at a coarse granularity; locking a single item is considered locking at a fine granularity.

Parent locks and sublocks are the mechanism by which the DLM allows processes to achieve locking at various degrees of granularity. See Section 9.6.5 for more information about parent locks and sublocks.

Figure 9–1: Model Database



9.2.2 Namespaces

You can view a namespace as a container for resource names. Multiple namespaces exist to provide separation of unrelated applications for reasons of security and modularity.

A namespace can be qualified by effective user ID or effective group ID, or it can be a public namespace that is accessible by all processes, regardless of their user or group ID.

Access to a namespace that is based on a user ID is limited to holders of that user ID. Access to a namespace that is based on a group ID is limited to members of that group. Any process can join a public namespace and manipulate locks in that namespace.

For namespaces that are qualified by user ID or group ID, security is based by determining a process's right to access the namespace, as evidenced by its holding the effective user ID or effective group ID. As a result, the user and group ID namespaces must be consistent across the cluster. After access to the namespace has been granted to a process, its individual locking operations within that namespace are unrestricted.

Applications using a public namespace must provide their own security and access mechanisms. One safe way to control access to locks in a public namespace is for the lock owner to hold onto a lock in Protected Read (PR) mode or a higher mode and not respond to any blocking notification. This prevents any other process from changing the lock value block for as long as the lock owner is executing.

Cooperating processes must use the same namespace to coordinate locks for a given resource. A process must join a namespace before attempting to call the `dlm_lock`, `dlm_locktp`, `dlm_quelock`, or `dlm_quelocktp` function to acquire a lock on a resource in that namespace. When a process calls the `dlm_nsjoin` function for a user ID or group ID qualified namespace, the DLM verifies that it is permitted to access a namespace by verifying that the process holds the group or user ID appropriate to that namespace. If the process passes this check, the DLM returns a handle to the namespace. When a process calls the `dlm_nsjoin` function for a public namespace, the DLM returns a handle to the namespace.

The process must present this handle on subsequent calls to DLM functions to acquire root locks (that is, the base parent lock for a given resource in a namespace). You can add sublocks under root locks without further namespace access checks.

A process can be a member of up to `DLM_NSPROCMAX` namespaces.

9.2.3 Uniquely Identifying Resources

The DLM distinguishes resources by using the following attributes:

- A namespace (`nsp`) — Use the `dlm_nsjoin` function to obtain a namespace handle before issuing a call to the `dlm_lock`, `dlm_locktp`, `dlm_quelock`, or `dlm_quelocktp` function to obtain a top-level (root) lock in a namespace. A root lock has no parent.
- The resource name that is specified by the process (`resnam`) — The name that is specified by the process represents the resource being locked. Other processes that need to access the resource must refer to it using the same name. The correlation between the name and the resource is a convention that is agreed upon by the cooperating processes.
- The resource name length (`resnlen`).
- The identification of the lock's parent (`parid`), if specified in a request — If a lock request is queued that specifies a parent lock ID of zero (0), the lock manager considers it to be a request for a root lock on a resource. If the lock request specifies a nonzero parent lock ID, it is considered to be a request for a sublock on the resource. In this case, the DLM accepts the request only if the root lock has been granted. This mechanism enables a process to lock a resource at different degrees of granularity and build lock trees.

For example, the following two sets of attributes identify the same resource:

Attribute	nsp	resnam	resnlen	parid
Resource 1	14	disk1	5	80
Resource 1	14	disk1	5	80

The following two sets of attributes also identify the same resource:

Attribute	nsp	resnam	resnlen	parid
Resource 1	14	disk1	5	40
Resource 1	14	disk12345	5	40

The following two sets of attributes identify different resources:

Attribute	nsp	resnam	resnlen	parid
Resource 1	0	disk1	5	80
Resource 2	0	disk1	5	40

9.3 Using Locks

To use distributed lock manager (DLM) functions, a process must request access to a resource (request a lock) using the `dlm_lock`, `dlm_locktp`, `dlm_quelock`, or `dlm_quelocktp` function. The request specifies the following parameters:

- A namespace handle that is obtained from a prior call to the `dlm_nsjoin` function — For those namespaces that are qualified by effective user ID or group ID, the DLM verifies a process's right to access a namespace before allowing it to obtain and manipulate locks on resources in that namespace. All processes can access public namespaces. See Section 9.2.2 for more information on namespaces.
- The resource name that represents the resource — The meaning of a resource name is defined by the application program. The DLM uses the resource name as a mechanism for matching lock requests that are issued by multiple processes. Resource names exist within a namespace. The same resource name in different namespaces is considered by the DLM to be a different name.
- The length of the resource name — A resource name can be from 1 through `DLM_RESNAMELEN` bytes in length.

- The identification of the lock's parent — You can specify as a parent ID either zero (0) to request a root lock, or a nonzero parent ID to request a sublock of that parent. See Section 9.2.2 for more information.
- The address of a location to which the DLM returns a lock ID — The `dlm_lock`, `dlm_locktp`, `dlm_quelock`, and `dlm_quelocktp` functions return a lock ID when the request has been accepted. The application then uses this lock ID to refer to the lock on subsequent operations, such as calls to the `dlm_cvt`, `dlm_quecvt`, and `dlm_unlock` functions.
- A lock request mode — The DLM functions compare the lock mode of the newly requested lock to the lock modes of other locks with the same resource name. See Section 9.3.1 for more information about lock modes.

Null mode locks (see Section 9.3.1) are compatible with all other lock modes and are always granted immediately.

New locks are granted immediately in the following instances:

- If no other process has a lock on the resource.
- If another process has a lock on the resource, the mode of the new request is compatible with the existing lock, and no locks are waiting in the CONVERTING or WAITING queue. See Section 9.3.2 for more information about lock mode compatibility.

New locks are not granted in the following instance:

- If another process already has a lock on the resource and the mode of the new request is not compatible with the lock mode of the existing lock, the new request is placed in a first-in first-out (FIFO) queue, where the lock waits until the resource's currently granted lock mode (resource group grant mode) becomes compatible with the lock request.

Processes can also use the `dlm_cvt` and `dlm_quecvt` functions to change the lock mode of a lock. This is called a lock conversion. See Section 9.3.4 for more information.

9.3.1 Lock Modes

The mode of a lock determines whether or not the resource can be shared with other lock requests. Table 9–2 describes the six lock modes.

Table 9–2: Lock Modes

Mode	Description
Null (DLM_NLMODE)	Grants no access to the resource; the Null mode is used as a placeholder for future lock conversions, or as a means of preserving a resource and its context when no other locks on it exist.
Concurrent Read (DLM_CRMODE)	Grants read access to the resource and allows it to be shared with other readers and writers. The Concurrent Read mode is generally used when additional locking is being performed at a finer granularity with sublocks, or to read data from a resource in an unprotected fashion (that is, while allowing simultaneous writes to the resource).
Concurrent Write (DLM_CWMODE)	Grants write access to the resource and allows it to be shared with other writers. The Concurrent Write mode is typically used to perform additional locking at a finer granularity, or to write in an unprotected fashion.
Protected Read (DLM_PRMODE)	Grants read access to the resource and allows it to be shared with other readers. No writers are allowed access to the resource. This is the traditional share lock.
Protected Write (DLM_PWMODE)	Grants write access to the resource and allows it to be shared with Concurrent Read mode readers. No other writers are allowed access to the resource. This is the traditional update lock.
Exclusive (DLM_EXMODE)	Grants write access to the resource and prevents it from being shared with any other readers or writers. This is the traditional Exclusive lock.

9.3.2 Levels of Locking and Compatibility

Locks that allow the process to share a resource are called low-level locks; locks that allow the process almost exclusive access to a resource are called high-level locks. Null and Concurrent Read mode locks are considered low-level locks; Protected Write and Exclusive mode locks are considered high-level locks. The lock modes from lowest to highest level access modes are as follows:

1. Null (NL)
2. Concurrent Read (CR)
3. Concurrent Write (CW) and Protected Read (PR)
4. Protected Write (PW)
5. Exclusive (EX)

The Concurrent Write (CW) and Protected Read (PR) modes are considered to be of equal level.

Locks that can be shared with other granted locks on a resource (that is, the resource's group grant mode) are said to have compatible lock modes. Higher-level lock modes are less compatible with other lock modes than are lower-level lock modes.

Table 9–3 lists the compatibility of the lock modes.

Table 9–3: Compatibility of Lock Modes

Mode of Requested Lock	Resource Group Grant Mode					
	NL	**CR**	**CW**	**PR**	**PW**	**EX**
Null (NL)	Yes	Yes	Yes	Yes	Yes	Yes
Concurrent Read (CR)	Yes	Yes	Yes	Yes	Yes	No
Concurrent Write (CW)	Yes	Yes	Yes	No	No	No
Protected Read (PR)	Yes	Yes	No	Yes	No	No
Protected Write (PW)	Yes	Yes	No	No	No	No
Exclusive (EX)	Yes	No	No	No	No	No

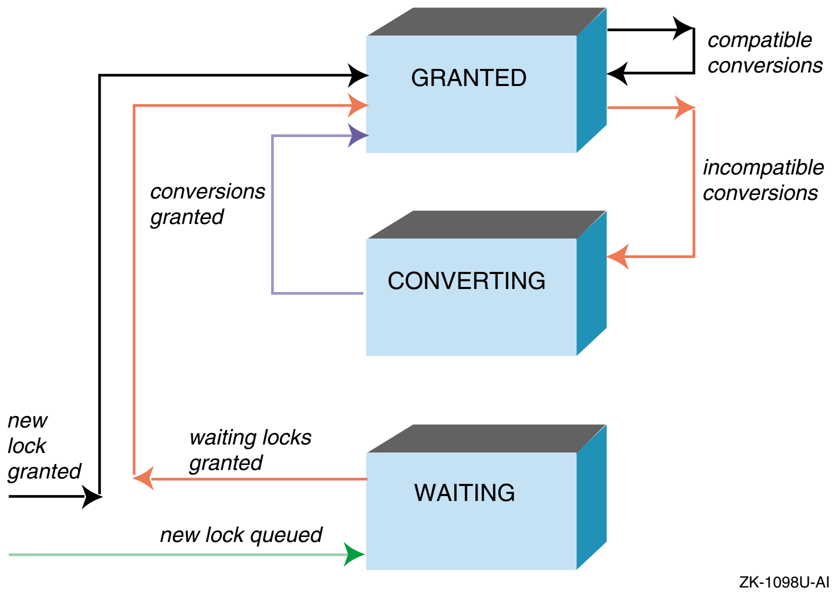
9.3.3 Lock Management Queues

A lock on a resource can be in one of the following three states:

- GRANTED — The lock request has been granted.
- CONVERTING — The lock is granted at one mode and a convert request is waiting to be granted at a mode that is compatible with the current resource group grant mode.
- WAITING — The new lock request is waiting to be granted.

A queue is associated with each of the three states, as shown in Figure 9–2.

Figure 9–2: Three Lock Queues



When you request a new lock on an existing resource, the DLM determines if any other locks are waiting in either the CONVERTING or WAITING queue, as follows:

- If other locks are waiting in either queue, the new lock request is placed at the end of the WAITING queue, except if the requested lock is a Null mode lock, in which case it is granted immediately.
- If both the CONVERTING and WAITING queues are empty, the lock manager determines if the new lock is compatible with the other granted locks. If the lock request is compatible, the lock is granted. If the lock request is not compatible, it is placed on the WAITING queue. (You can specify the `DLM_NOQUEUE` flag to the `dlm_lock`, `dlm_locktp`, `dlm_quelock`, `dlm_quelocktp`, `dlm_cvt`, or `dlm_quecvt` call to direct the DLM not to queue a lock request if it cannot be granted immediately. In this case, the lock request is granted if it is compatible with the resource's group grant mode, or is rejected with a `DLM_NOTQUEUED` error if it is not.)

9.3.4 Lock Conversions

Lock conversions allow processes to change the mode of locks. For example, a process can maintain a low-level lock on a resource until it decides to limit access to the resource by requesting a lock conversion.

You specify lock conversions by using either the `dlm_cvt` or the `dlm_quecvt` function with the lock ID of a previously granted lock that you want to convert. If the requested lock mode is compatible with the currently granted locks, the conversion request is granted immediately. If the requested lock mode is incompatible with the existing locks in the GRANTED queue, the request is placed at the end of the CONVERTING queue. The lock retains its granted mode until the conversion request is granted.

After the DLM grants the conversion request, it grants any compatible requests that are immediately following it on the CONVERTING queue. The DLM continues to grant requests until the CONVERTING queue is empty or it encounters an incompatible lock.

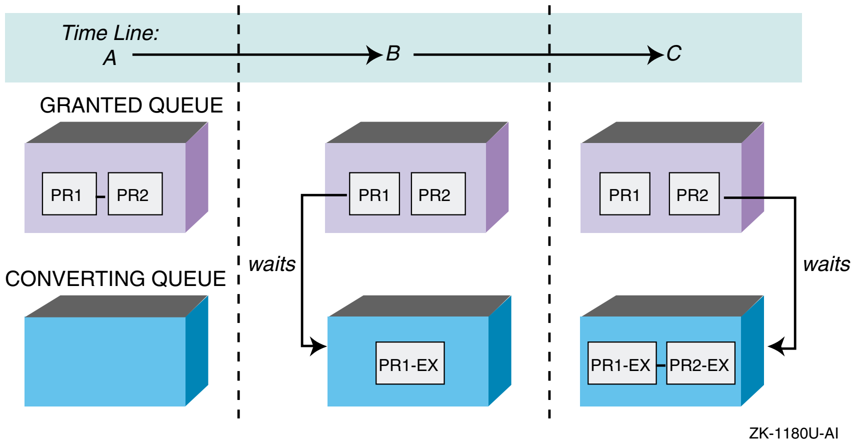
When the CONVERTING queue is empty, the DLM examines the WAITING queue. It grants the first lock request on the WAITING queue if it is compatible with the locks currently granted. The DLM continues to grant requests until the WAITING queue is empty or it encounters an incompatible lock.

9.3.5 Deadlock Detection

The DLM can detect two forms of deadlock:

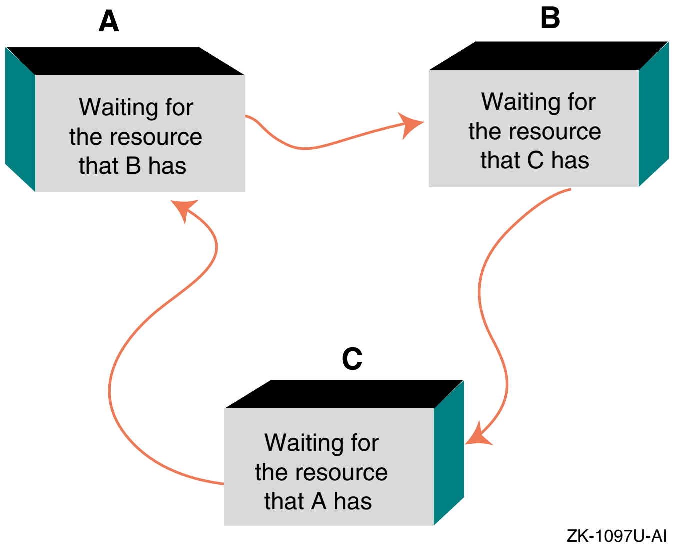
- Conversion deadlock — A conversion deadlock occurs when a conversion request has a granted mode that is incompatible with the requested mode of another conversion request that is ahead of it in the CONVERTING queue. For example, in Figure 9–3, there are two granted PR mode locks on a resource (that is, the resource grant mode is PR). One PR mode lock tries to convert to EX mode and, as a result, must wait in the CONVERTING queue. Then, the second PR mode lock also tries to convert to EX mode. It, too, must wait, behind the first lock's request, in the CONVERTING queue. However, the first lock's request can never be granted, because its requested mode (EX) is incompatible with the second lock's granted mode (PR). The second lock's request can never be granted because it is waiting behind the first lock's request in the CONVERTING queue.

Figure 9–3: Conversion Deadlock



- Multiple resource deadlock — A multiple resource deadlock occurs when a list of processes are each waiting for each other in a circular fashion. For example, in Figure 9–4, three processes have queued requests for resources that cannot be accessed until the current locks that are held are dequeued (or converted to a lower lock mode). Each process is waiting for another process to dequeue its lock request.

Figure 9–4: Multiple Resource Deadlock



A
Waiting for the resource that B has
B
Waiting for the resource that C has
C
Waiting for the resource that A has
ZK-1097U-AI

If the DLM determines that either a conversion deadlock or a multiple resource deadlock exists, it chooses a lock to use as a victim to break the deadlock. Although the victim is arbitrarily selected, it is guaranteed to be either on the CONVERTING or WAITING queue (that is, it is not in the GRANTED queue).

The DLM returns a `DLM_DEADLOCK` final completion status code to the process that issued this `dlm_lock`, `dlm_locktp`, or `dlm_cvt` function call (or provides this status in the *`completion_status`* parameter to the completion routine that is specified in the call to the `dlm_quelock`, `dlm_quelocktp`, or `dlm_quecvt` function). Granted locks are never revoked; only converting and waiting lock requests can receive the `DLM_DEADLOCK` status code.

Note

Do not make assumptions about which lock the DLM will choose to break a deadlock. Also, it is possible to have undetectable deadlocks when other services such as semaphores or file locks are used in conjunction with the DLM. The DLM detects only those deadlocks that involve its own locks.

9.4 Dequeuing Locks

When a process no longer needs a lock on a resource, it can release the lock by calling the `dlm_unlock` function.

When a lock is released, the specified lock request is removed from whatever queue it is in. Locks are dequeued from any queue: GRANTED, WAITING, or CONVERTING. When the last lock on a resource is dequeued, the resource is deleted from the distributed lock manager (DLM) database.

The `dlm_unlock` function can write or invalidate the resource's lock value block if it specifies the *`valb_p`* parameter and the `DLM_VALB` flag. If the lock to be dequeued has a granted mode of PW or EX, the contents of the process's value block are stored in the resource value block. If the lock that is being dequeued is in any other mode, the lock value block is not used. If the `DLM_INVVALBLK` flag is specified, the resource's lock value block is marked invalid.

The `dlm_unlock` function uses the following flags:

- The `DLM_DEQALL` flag indicates that all locks that are held by the process are to be dequeued or that a subtree of locks are to be dequeued, depending on the value of the *`lkid_p`* parameter, as listed in Table 9–4.

Table 9–4: Using the DLM_DEQALL Flag in a dlm_unlock Function Call

lkid_p	DLM_DEQALL	Result
≠ 0	Clear	Only the lock specified by *`lkid_p`* is released.
≠ 0	Set	All sublocks of the indicated lock are released. The lock specified by *`lkid_p`* is not released.
= 0	Clear	Returns the invalid lock ID condition value (`DLM_IVLOCKID`).
= 0	Set	All locks held by the process are released.

- The `DLM_INVVALBLK` flag causes the DLM to invalidate the resource lock value block of granted or converting PW or EX mode locks. The resource lock value block remains marked as invalid until it is again written. See Section 9.6.6 for more information about lock value blocks.
- The `DLM_VALB` flag causes the DLM to write the resource lock value block for granted or converting PW or EX mode locks.

You cannot specify both the `DLM_VALB` and `DLM_INVVALBLK` flags in the same request.

9.5 Canceling a Conversion Request

The `dlm_cancel` function cancels a lock conversion. A process can cancel a lock conversion only if the lock request has not yet been granted, in which case the request is in the CONVERTING queue. Cancellation causes a lock in the CONVERTING queue to revert to the granted lock mode it had before the conversion request. The *`blkrtn`* and *`notprm`* values of the lock also revert to the old values. The DLM calls any completion routine that is specified in the conversion request to indicate that the request has been canceled. The returned status is `DLM_CANCEL`.

9.6 Advanced Locking Techniques

The previous sections discussed locking techniques and concepts that are useful to all applications. The following sections discuss specialized features of the distributed lock manager (DLM).

9.6.1 Asynchronous Completion of a Lock Request

The `dlm_lock`, `dlm_locktp`, and `dlm_cvt` functions complete when the lock request has been granted or has failed, as indicated by the return status value.

If you want an application not to wait for completion of the lock request, have it use the `dlm_quelock`, `dlm_quelocktp`, and `dlm_quecvt` functions. These functions return control to the calling program after the lock request is queued. The status value that is returned by these functions indicates whether the request was queued successfully or was rejected. After a request is queued, the calling program cannot access the resource until the request is granted.

Calls to the `dlm_quelock`, `dlm_quelocktp`, and `dlm_quecvt` functions must specify the address of a completion routine. The completion routine runs when the lock request is successful or unsuccessful. The DLM passes to the completion routines status information that indicates the success or failure of the lock request.

Note

If an application wants the DLM to deliver completion notifications, it must call the `dlm_set_signal` function once before making the first lock request requiring one. Alternatively, the application can periodically call the `dlm_notify` function. The `dlm_notify` function enables a process to poll for pending notifications and request their delivery, without needing to call the `dlm_set_signal` function. The polling method is not recommended.

9.6.2 Notification of Synchronous Completion

The DLM provides a mechanism that allows processes to determine whether a lock request is granted synchronously; that is, if the lock is not placed on the CONVERTING or WAITING queue. By avoiding the overhead of signal delivery and the resulting execution of a completion routine, an application can use this feature to improve performance in situations where most locks are granted synchronously (as is normally the case). An application can also use this feature to test for the absence of a conflicting lock when the request is processed.

This feature works as follows:

- If the `DLM_SYNCSTS` flag is set in a call to the `dlm_lock`, `dlm_locktp`, `dlm_cvt`, `dlm_quelock`, `dlm_quelocktp`, or `dlm_quecvt` function, and a lock is granted synchronously, the function returns a status value of `DLM_SYNCH` to its caller. In the case of the `dlm_quelock`, `dlm_quelocktp`, and `dlm_quecvt` functions, the DLM delivers no completion notification.
- If a lock request that is initiated by a `dlm_quelock`, `dlm_quelocktp`, and `dlm_quecvt` function call is not completed synchronously, the

function returns a status value of `DLM_SUCCESS`, indicating that the request has been queued. The DLM delivers a completion notification when the lock is granted successfully or the lock grant fails.

9.6.3 Blocking Notifications

In some applications that use the DLM functions, a process must know whether it is preventing another process from locking a resource. The DLM informs processes of this by using blocking notifications. To enable blocking notifications, the *`blkrtn`* parameter of the lock request must contain the address of a blocking notification routine. When the lock prevents another lock from being granted, a blocking notification is delivered and the blocking notification routine is exeuted.

The DLM provides the blocking notification routine with the following parameters:

`notprm`	Context parameter of the blocking lock. This parameter was supplied by the caller of the `dlm_lock`, `dlm_locktp`, `dlm_quelock`, `dlm_quelocktp`, `dlm_cvt`, or `dlm_quecvt` function in the lock request for the blocking lock.
`blocked_hint`	The *`hint`* parameter from the first blocked lock. This parameter was supplied by the caller of the `dlm_lock`, `dlm_locktp`, `dlm_quelock`, `dlm_quelocktp`, `dlm_cvt`, or `dlm_quecvt` function in the lock request for the first blocked lock.
`lkid_p`	Pointer to the lock ID of the blocking lock.
`blocked_mode`	Requested mode of the first blocked lock.

By the time the notification is delivered, the following conditions can still exist:

- The lock can still be blocked.
- The blocked lock can be released by the application; therefore, no locks are actually blocked.
- The blocked lock can be selected as a deadlock victim and the request failed to break a deadlock cycle.
- The blocked lock can be released by the application and another lock queued that is now blocked; therefore, a completely different lock is actually blocked.

- Other locks are backed up behind the original blocked lock or subsequently queued blocked lock.

Because these conditions are possible, the DLM can make no guarantees about the validity of the *`blocked_hint`* and *`blocked_mode`* parameters at the time that the blocking routine is executed.

Note

If an application wants the DLM to deliver blocking notifications, it must call the `dlm_set_signal` function once before making the first lock request requiring a blocking notification.

Also, if the signal that is specified in the `dlm_set_signal` call is blocked, the blocking notification will not be delivered until the signal is unblocked. Alternatively, the application can periodically call the `dlm_notify` function. The `dlm_notify` function enables a process to poll for pending notifications and request their delivery. The polling method is not recommended.

9.6.4 Lock Conversions

Lock conversions perform the following functions:

- Promoting or demoting lock modes — The DLM provides mechanisms to maintain a low-level lock and convert it to a higher-level lock mode when necessary. A procedure normally needs an Exclusive (EX) or Protected Write (PW) mode lock while writing data. However, you do not want the procedure to keep the resource exclusively locked all the time, because writing may not always be necessary. Maintaining an EX or PW mode lock prevents other processes from accessing the resource. Lock conversions allow a process to request a low-level lock at first and then convert the lock to a higher-level lock mode (PW mode, for example) only when it needs to write data. However, because the process may not always need to write to the resource, make sure it requests a lock conversion to a lower-level lock mode when it has finished writing.

  Note

  Other types of applications, for which shared write access may not be all that important, can also use the DLM's lock conversion features. For example, an application may require a master instance, while allowing for slave or secondary instances of the same application. Locks and lock conversions can be used to implement a quick recovery scheme in case the master instance fails.

- Maintaining values that are stored in a resource lock value block — A lock value block is a small piece of memory that is shared between holders of locks on the same resource. Some applications of locks require the use of the lock value block. If a version number or other data is maintained in the lock value block, you need to maintain at least one lock on the resource so that the value block is not lost. In this case, processes convert their locks to Null locks rather than dequeuing them when they have finished accessing the resource. See Section 9.6.6 for further discussion of using lock value blocks.
- Improving performance in some applications — To improve performance in some applications, all resources that might be locked are locked with Null (NL) mode locks during initialization. You can convert the NL mode locks to higher-level locks as needed. Usually a conversion request is faster than a new lock request, because the necessary data structures have already been built. However, maintaining any lock for the life of a procedure uses system dynamic memory. Therefore, the technique of creating all necessary locks as NL mode locks and converting them as needed improves performance at the expense of increased memory requirements.

9.6.4.1 Queuing Lock Conversions

To perform a lock conversion, a procedure calls the `dlm_cvt` or `dlm_quecvt` function. The lock that is being converted is identified by the *`lkid_p`* parameter. A lock must be granted before it can be the object of a conversion request.

9.6.4.2 Forced Queuing of Conversions

To promote more equitable access to a given resource, you can force certain conversion requests to be queued that would otherwise be granted. A conversion request with the `DLM_QUECVT` flag set is forced to wait behind any already queued conversions. In this manner, you can specify the `DLM_QUECVT` flag to give other locks a chance of being granted. However, the conversion request is granted immediately if no conversions are already queued.

The `DLM_QUECVT` behavior is valid only for a subset of all possible conversions. Table 9–5 defines the set of conversion requests that are permitted when you specify the `DLM_QUECVT` flag. Illegal conversion requests fail with a return status of `DLM_BADPARAM`.

Table 9–5: Conversions Allowed When the DLM_QUECVT Flag is Specified

Mode at Which Lock is Held	**Mode to Which Lock is Converted**					
	NL	**CR**	**CW**	**PR**	**PW**	**EX**
Null (NL)	—	Legal	Legal	Legal	Legal	Legal
Concurrent Read (CR)	—	—	Legal	Legal	Legal	Legal
Concurrent Write (CW)	—	—	—	—	Legal	Legal
Protected Read (PR)	—	—	—	—	Legal	Legal
Protected Write (PW)	—	—	—	—	—	—
Exclusive (EX)	—	—	—	—	—	—

9.6.5 Parent Locks

When a process calls the `dlm_lock`, `dlm_locktp`, `dlm_quelock`, or `dlm_quelocktp` function to issue a lock request, it can declare a parent lock for the new lock by specifying the parent ID in the *`parid`* parameter. Locks with parents are called sublocks. A parent lock must be granted before the sublocks belonging to the parent can be granted in the same or some other mode.

The benefit of using parent locks and sublocks is that they allow low-level locks (Concurrent Read or Concurrent Write) to be held at a coarse granularity, while higher-level (Protected Write or Exclusive mode) sublocks are held on resources of a finer granularity. For example, a low-level lock might control access to an entire file, while higher-level sublocks protect individual records or data items in the file.

Assume that a number of processes need to access a database. The database can be locked at two levels: the file and individual records. When updating all the records in a file, locking the whole file and updating the records without additional locking is faster and more efficient. But, when updating selected records, locking each record as it is needed is preferable.

To use parent locks in this way, all processes request locks on the file. Processes that need to update all records must request Protected Write (PW) or Exclusive (EX) mode locks on the file. Processes that need to update individual records request Concurrent Write (CW) mode locks on the file, and then use sublocks to lock the individual records in PW or EX mode.

In this way, the processes that need to access all records can do so by locking the file, while processes that share the file can lock individual records. A number of processes can share the file-level lock at concurrent write mode, while their sublocks update selected records.

9.6.6 Lock Value Blocks

The lock value block is a structure of DLM_VALBLKSIZE unsigned longwords in size that a process associates with a resource by specifying the *valb_p* parameter and the DLM_VALB option in calls to DLM functions. When the lock manager creates a resource, it also creates a lock value block for that resource. The DLM maintains the resource lock value block until there are no more locks on the resource.

When a process specifies the DLM_VALB option and a valid address in the *valb_p* parameter in a new lock request and the request is granted, the contents of the resource lock value block are copied to the process's lock value block from the resource lock value block.

When a process specifies the *valb_p* parameter and the DLM_VALB option in a conversion from PW mode or EX mode to the same or a lower mode, the contents of the process's lock value block are stored in the resource lock value block.

In this manner, processes can pass (and update) the value in the lock value block along with the ownership of a resource. Table 9–6 indicates how lock conversions affect the contents of the process's and the resource's lock value block.

Table 9–6: Effect of Lock Conversion on Lock Value Block

Mode at Which Lock Is Held	**Mode to Which Lock Is Converted**					
	NL	**CR**	**CW**	**PR**	**PW**	**EX**
Null (NL)	Read	Read	Read	Read	Read	Read
Concurrent Read (CR)	—	Read	Read	Read	Read	Read
Concurrent Write (CW)	—	—	Read	—	Read	Read
Protected Read (PR)	—	—	—	Read	Read	Read
Protected Write (PW)	Write	Write	Write	Write	Write	Read
Exclusive (EX)	Write	Write	Write	Write	Write	Write

When granted PW or EX mode locks are released using the dlm_unlock function, the address of a lock value block is specified in the *valb_p* parameter and the DLM_VALB option is specified, the contents of the process's lock value block are written to the resource lock value block. If the lock being released is in any other mode, the lock value block is not used.

In some situations, the resource lock value block can become invalid. When this occurs, the DLM warns the caller of a function specifying the *valb_p* parameter by returning the completion status of DLM_SUCCVALNOTVALID or

DLM_SYNCVALNOTVALID. The following events can invalidate the resource lock value block:

- A process holding a PW or EX mode lock on a resource terminates abnormally.
- A node participating in locking fails and a process on that node was holding (or might have been holding) a PW or EX mode lock on the resource.
- A process holding a PW or EX mode lock on the resource calls the `dlm_unlock` function to dequeue this lock and specifies the flag DLM_INVVALBLK in the *flags* parameter.

9.7 Local Buffer Caching Using DLM Functions

Applications can use the distributed lock manager (DLM) to perform local buffer caching (also called distributed buffer management). Local buffer caching allows a number of processes to maintain copies of data (for example, disk blocks) in buffers that are local to each process, and to be notified when the buffers contain invalid data due to modifications by another process. In applications where modifications are infrequent, you may save substantial I/O by maintaining local copies of buffers — hence, the names local buffer caching or distributed buffer management. Either the lock value block or blocking notifications (or both) can be used to perform buffer caching.

9.7.1 Using the Lock Value Block

To support local buffer caching using the lock value block, each process maintaining a cache of buffers maintains a Null (NL) mode lock on a resource that represents the current contents of each buffer. (For this discussion, assume that the buffers contain disk blocks.) The lock value block that is associated with each resource is used to contain a disk block version number. The first time that a lock is obtained on a particular disk block, the application returns the current version number of that disk block in the lock value block of the process.

If the contents of the buffer are cached, this version number is saved along with the buffer. To reuse the contents of the buffer, the NL mode lock must be converted to Protected Read (PR) mode or Exclusive (EX) mode, depending on whether the buffer is to be read or written. This conversion returns the latest version number of the disk block. The application compares the version number of the disk block with the saved version number. If they are equal, the cached copy is valid. If they are not equal, the application must read a fresh copy of the disk block from disk.

Whenever a procedure modifies a buffer, it writes the modified buffer to disk and then increments the version number before converting the corresponding lock to NL mode. In this way, the next process that attempts to use its local copy of the same buffer finds a version number mismatch and must read the latest copy from disk, rather than use its cached (now invalid) buffer.

9.7.2 Using Blocking Notifications

Blocking notifications are used to notify processes with granted locks that another process with an incompatible lock mode has been queued to access the same resource.

You may use blocking notifications to support local buffer caching in two ways. One technique involves deferred buffer writes; the other technique is an alternate method of local buffer caching without using lock value blocks.

9.7.2.1 Deferring Buffer Writes

When local buffer caching is being performed, a modified buffer must be written to disk before the EX mode lock can be released. If a large number of modifications are expected (particularly over a short period of time), you can reduce disk I/O by maintaining the EX mode lock for the entire time that the modifications are being made, and writing the buffer once.

However, this prevents other processes from using the same disk block during this interval. This can be avoided if the process holding EX mode lock has a blocking notification. The notification will notify the process if another process needs to use the same disk block. The holder of the EX mode lock can then write the buffer to disk and convert its lock to NL mode (which allows the other process to access the disk block). However, if no other process needs the same disk block, the first process can modify it many times, but write it only once.

Note

After a blocking notification is delivered to a process, the process must convert the lock to receive any subsequent blocking notifications.

9.7.2.2 Buffer Caching

To perform local buffer caching using blocking notifications, processes do not convert their locks to NL mode from PR or EX mode when they are finished with the buffer. Instead, they receive blocking notifications whenever another process attempts to lock the same resource in an incompatible lock mode. With this technique, processes are notified that their cached buffers are invalid as soon as a writer needs the buffer, rather than the next time that the process tries to use the buffer.

9.7.3 Choosing a Buffer Caching Technique

The choice between using version numbers or blocking notifications to perform local buffer caching depends on the characteristics of the application. An application that uses version numbers performs more lock conversions, while one that uses blocking notifications delivers more notifications. Note that these techniques are compatible; some processes can use one technique at the same time that other processes use the other. Generally speaking, blocking notifications are preferred in a low-contention environment, while version numbers are preferred in a high-contention environment. You may even invent combined or adaptive strategies.

In a combined strategy, the applications use specific techniques. If a process is expected to reuse the contents of a buffer in a short amount of time, blocking notifications are used; if there is no reason to expect a quick reuse, version numbers are used.

In an adaptive strategy, an application makes evaluations on the rate of blocking notifications and conversions. If blocking notifications arrive frequently, the application changes to using version numbers; if many conversions take place and the same cached copy remains valid, the application changes to using blocking notifications.

For example, consider the case where one process continually displays the state of a database, while another occasionally updates it. If version numbers are used, the displaying process must always verify to see that its copy of the database is valid (by performing a lock conversion); if blocking notifications are used, the displaying process is informed every time that the database is updated. However, if updates occur frequently, using version numbers is preferable to continually delivering blocking notifications.

9.8 Distributed Lock Manager Functions Code Example

The following programs show the basic mechanisms that an application uses to join a namespace and establish an initial lock on a resource in that namespace. They also demonstrate such key distributed lock manager (DLM) concepts such as lock conversion, the use of lock value blocks, and the use of blocking notification routines.

The `api_ex_master.c` and `api_ex_client.c` programs, which are listed in Example 9–1 and available from the `/usr/examples/cluster` directory (when the `TCRMAN`*xxx* subset is installed), can execute in parallel on the same cluster member or on different cluster members. You must run both programs from accounts with the same user ID (UID) and you must start the `api_ex_master.c` program first. They display output similar to the following:

```
% **api_ex_master**
&
api_ex_master: grab a EX mode lock
api_ex_master: value block read
api_ex_master: expected empty value block got <>
api_ex_master: start client and wait for the blocking notification to
        continue
% **api_ex_client**
&
        api_ex_client: grab a NL mode lock
 api_ex_client: value block read
 api_ex_client: expected empty value block got <>
 api_ex_client: converting to NL->EX to get the value block.
 api_ex_client: should see blocking routine run on master
*** api_ex_master: blk_and_go hold the lock for a couple of seconds
*** api_ex_master: blk_and_go sleeping
*** api_ex_master: blk_and_go sleeping

*** api_ex_master: blk_and_go setting done
api_ex_master: now convert (EX→EX) to write the value block
<abc>
*** api_ex_master: blkrtn: down convert to NL
api_ex_master: waiting for blocking notification
 api_ex_client: value block read
api_ex_master: trying to get the lock back as PR to read value block
 api_ex_client: expected <abc> got <abc>
 *** api_ex_client: blkrtn: dequeue EX lock to write value block <>
 *** api_ex_client: hold the lock for a couple of seconds
 *** api_ex_client: sleeping
 *** api_ex_client: sleeping
 *** api_ex_client: sleeping
        api_ex_client: sleeping waiting for blocking notification
api_ex_master: value block read
 api_ex_client: done
api_ex_master: expected <efg> got <efg>
api_ex_master done
```

Example 9–1: Locking, Lock Value Blocks, and Lock Conversion

```
/************************************************************************
*
*
*                       api_ex_master.c
*
*
*
************************************************************************/


/* cc -g -o api_ex_master api_ex_master.c -ldlm */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <signal.h>

#include <sys/dlm.h>

char *resnam = "dist shared resource";
char *prog;
int done = 0;

#ifdef DLM_DEBUG
int dlm_debug = 2;
#define Cprintf if(dlm_debug)printf
```

Example 9–1: Locking, Lock Value Blocks, and Lock Conversion (cont.)

```
#define Dprintf if(dlm_debug >= 2 )printf
#else /* DLM_DEBUG */
#define Cprintf ;
#define Dprintf ;
#endif /* DLM_DEBUG */

void
error(dlm_lkid_t *lk, dlm_status_t stat)
{
 printf("%s: lock error %s on lkid 0x%lx\n", prog, dlm_sperrno(stat), lk);
        abort();
}
void
blk_and_go(callback_arg_t x, callback_arg_t y, dlm_lkid_t *lk, dlm_lkmode_t blkmode)
{
 int i;

 printf("*** %s: blk_and_go hold the lock for a couple of seconds\n", prog);
 for (i = 0; i < 3; i++) {

  printf("*** %s: blk_and_go sleeping\n", prog);
  sleep(1);
 }
 printf("*** %s: blk_and_go setting done\n", prog);
 /* done waiting */
 done = 1; 13
}
void
blkrtn(callback_arg_t x, callback_arg_t y, dlm_lkid_t *lk, dlm_lkmode_t blkmode)
{
 dlm_status_t stat;

 Cprintf("*** %s: blkrtn: x 0x%lx y 0x%lx lkid 0x%lx blkmode
  %d\n", prog, x, y, *lk, blkmode);
 printf("*** %s: blkrtn: down convert to NL\n", prog);
 if ((stat = dlm_cvt(lk, DLM_NLMODE, 0, 0, 0, 0, 0, 0)) != DLM_SUCCESS)
  error(lk, stat); 16
 /* let waiters know we're done */
 done = 1;
}
main(int argc, char *argv[])
{
 int resnlen, i;
 dlm_lkid_t lkid;
 dlm_status_t stat;
 dlm_valb_t vb;
 dlm_nsp_t nsp;

 */ this proram must be run first */

 /* first we need to join a namespace */
 if ((stat = dlm_nsjoin(getuid(), &nsp, DLM_USER)) != DLM_SUCCESS) { 1
                printf("%s: can't join namespace\n", argv[0]);
  error(0, stat);
 }

 prog = argv[0];

 /* now let DLM know what signal to use for blocking routines */
 dlm_set_signal(SIGIO, &i); 2

 Cprintf("%s: dlm_set_signal: i %d\n", prog, i);

 resnlen = strlen(resnam); 3

        /* get EX mode lock and establish blocking notif routine */
 Cprintf("%s: grab a EX mode lock\n", prog);
 stat = dlm_lock(nsp, (uchar_t *)resnam, resnlen, 0, &lkid, DLM_EXMODE,
   &vb, (DLM_VALB | DLM_SYNCSTS), 0, 0, blk_and_go, 0); 4
 /*
  * since we're the only one running it
  * had better be granted DLM_SYNCH status
```

Example 9–1: Locking, Lock Value Blocks, and Lock Conversion (cont.)

```
  */
 if(stat != DLM_SYNCH) { [5]
  printf("%s: dlm_lock failed\n", prog);
  error(&lkid, stat);
 }
 /* newly-created value block should be empty */
 printf("%s: value block read\n", prog);
 printf("%s: expected empty value block got <%s>\n", prog, vb.valblk);
 if (strlen(vb.valblk)) {  [6]
  printf("%s: lock: value block not empty\n", prog);
  error(&lkid, stat);
 }
 printf("%s: start client and wait for the blocking notification to
    continue\n", prog);
 while (!done)
  sleep(1); [7]

 done = 0;
 /* put a known string into the value block */
 (void) strcat(vb.valblk, "abc"); [14]

 printf("%s: now convert (EX→EX) to write the value block <%s>\n", prog, vb.valblk);
 /* use a new blocking routine */
 stat = dlm_cvt(&lkid, DLM_EXMODE, &vb, (DLM_VALB | DLM_SYNCSTS), 0, 0, blkrtn, 0); [15]

 /*
  * since we own (EX) the resource the
  * convert had better be granted SYNC
  */
 if(stat != DLM_SYNCH) {
  printf("%s: convert failed\n", prog);
  error(&lkid, stat);
 }

        printf("%s: waiting for blocking notification\n", prog);
 while (!done)
  sleep(1);
 printf("%s: trying to get the lock back as PR to read value block\n", prog);
 stat = dlm_cvt(&lkid, DLM_PRMODE, &vb, DLM_VALB, 0, 0, 0, 0); [19]
 if (stat != DLM_SUCCESS) {
  printf("%s: error on conversion lock\n", prog);
  error(&lkid, stat);
 }
 printf("%s: value block read\n", prog);
 printf("%s: expected <efg> got <%s>\n", prog, vb.valblk);
 /* compare to the other known string */
 if (strcmp(vb.valblk, "efg")) {
  printf("%s: main: value block mismatch <%s>\n", prog, vb.valblk);
  error(&lkid, stat); [23]
 }
 printf("%s done\n", prog); [24]
 exit(0);
}
/*********************************************************************
*                                                                    *
*                       api_ex_client.c                              *
*                                                                    *
*********************************************************************/

/* cc -g -o api_ex_client api_ex_client.c -ldlm */

#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <signal.h>

#include <sys/dlm.h>

char *resnam = "dist shared resource";
char *prog;
```

Example 9–1: Locking, Lock Value Blocks, and Lock Conversion (cont.)

```
int done = 0;

#ifdef DLM_DEBUG
int dlm_debug = 2;
#define Cprintf if(dlm_debug)printf
#define Dprintf if(dlm_debug >= 2 )printf
#else /* DLM_DEBUG */
#define Cprintf ;
#define Dprintf ;
#endif /* DLM_DEBUG */

void
error(dlm_lkid_t *lk, dlm_status_t stat)
{
 printf("\t%s: lock error %s on lkid 0x%lx\n", prog, dlm_sperrno(stat), *lk);
 abort();
}

/*
 * blocking routine that will release the lock and in doing so will
 * write the resource value block.
 */
void
blkrtn(callback_arg_t x, callback_arg_t y, dlm_lkid_t *lk, dlm_lkmode_t blkmode)
{
 dlm_status_t stat;
 dlm_valb_t   vb;
 int          i;

 Cprintf("*** %s: blkrtn: x 0x%lx y 0x%lx lkid 0x%lx blkmode
  %d\n", prog, x, y, *lk, blkmode);
 printf("\t*** %s: blkrtn: dequeue EX lock to write value block <%s>\n", prog, vb.valblk);
 printf("\t*** %s: hold the lock for a couple of seconds\n", prog);
 for (i = 0; i < 3; i++) {
  printf("\t*** %s: sleeping\n", prog);
  sleep(1);
 }
 /* make sure its clean */
 bzero(vb.valblk, DLM_VALBLKSIZE);
 /* write something different */
 (void) strcat(vb.valblk, "efg"); 20
 if((stat = dlm_unlock(lk, &vb, DLM_VALB)) != DLM_SUCCESS) error(lk, stat); 21
 /* let waiters know we're done */
 done = 1;
}
main(int argc, char *argv[])
{
 int resnlen, i;
 dlm_lkid_t lkid;
 dlm_status_t stat;
 dlm_nsp_t nsp;
 dlm_valb_t vb;

 /* first we need to join a namespace */
 if ((stat = dlm_nsjoin(getuid(), &nsp, DLM_USER)) !=
 DLM_SUCCESS) {
  printf("\t%s: can't join namespace\n", argv[0]);
  error(0, stat); 8
 }

 prog = argv[0];

 /* now let DLM know what signal to use for blocking routines */
 dlm_set_signal(SIGIO, &i);
 Cprintf("\t%s: dlm_set_signal: i %d\n", prog, i); 9

 resnlen = strlen(resnam);
        Cprintf("\t%s: resnam %s\n", prog, resnam);

 printf("\t%s: grab a NL mode lock\n", prog);
 stat = dlm_lock(nsp, (uchar_t *)resnam, resnlen, 0, &lkid, DLM_NLMODE,
  &vb, (DLM_VALB | DLM_SYNCSTS), 0, 0, 0, 0);
```

Example 9–1: Locking, Lock Value Blocks, and Lock Conversion (cont.)

```
 /* NL mode better be granted SYNC status */
 if(stat != DLM_SYNCH) {  10
  printf("\t%s: dlm_lock failed\n", prog);
  error(&lkid, stat);
 }
 /* should be nulls since master hasn't written anything yet */
 printf("\t%s: value block read\n", prog);
 printf("\t%s: expected empty value block got <%s>\n", prog, vb.valblk);
 if (strlen(vb.valblk)) {  11
  printf("\t%s: value block not empty\n", prog);
  error(&lkid, stat);
 }

 done = 0;
 printf("\t%s: converting to NL->EX to get the value block.\n", prog);
        printf("\t%s: should see blocking routine run on master\n", prog);
 stat = dlm_cvt(&lkid, DLM_EXMODE, &vb, DLM_VALB, 0, 0, blkrtn, 0);  12
 if(stat != DLM_SUCCESS) {
  printf("\t%s: dlm_cvt failed\n", prog);
  error(&lkid, stat);
 }
 /* should have read what master wrote, "abc" */
 printf("\t%s: value block read\n", prog);
 printf("\t%s: expected <abc> got <%s>\n", prog, vb.valblk);
 if (strcmp(vb.valblk, "abc")) {  17
  printf("\t%s: main: value block mismatch <%s>\n", prog, vb.valblk);
  error(&lkid, stat);
 }
 /* now wait for blocking from master */
 printf("\t%s: sleeping waiting for blocking notification\n", prog);
 while (!done)
  sleep(1);  18
 printf("\t%s: done\n", prog);  22
        exit(0);
}
```

1 The `api_ex_master.c` program calls the `dlm_nsjoin` function to join the namespace of the resource on which it will request a lock. This namespace is the current process's UID, as obtained from the `getuid` system call. This namespace allows access from processes that are holding the effective UID of the resource owner, as indicated by the `DLM_USER` parameter. If successful, the function returns a namespace handle to the location that is indicated by the `nsp` parameter.

2 The `api_ex_master.c` program calls the `dlm_set_signal` function to specify that the DLM is to use the SIGIO signal to send completion and blocking notifications to this process.

3 The `api_ex_master.c` program obtains the length of the resource name to be supplied in the subsequent call to the `dlm_lock` function call. The name of the resource is `dist shared resource`.

4 The `api_ex_master.c` program calls the `dlm_lock` function to obtain an Exclusive mode (`DLM_EXMODE`) lock on the `dist shared resource` resource in the `uid` namespace. The namespace handle, resource name, and resource name length are all supplied as required parameters.

The `DLM_SYNCSTS` flag causes DLM to return `DLM_SYNCH` status if the lock request is granted immediately. If the function call is successful, the DLM returns the lock ID of the Exclusive mode (EX) lock to the location that is specified by the `lkid` parameter.

This function call also specifies the `DLM_VALB` flag and a location to and from which the contents of the lock value block for the resource are written or read. The DLM copies the resource's lock value to this location when the lock requested by the `dlm_lock` function call is granted. Finally, the function call specifies the blocking notification routine `blk_and_go`. The DLM will call this routine after the lock has been granted and is blocking another lock request.

5 The `api_ex_master.c` program examines the status value that is returned from the `dlm_lock` function call. If the status value is not `DLM_SYNCH` status (the successful condition value that is requested by the `DLM_SYNCSTS` flag in the `dlm_lock` function call), the lock request has had to wait for the lock to be granted. Because no other programs that are interested in this lock are currently running, this cannot be the case.

6 The `api_ex_master.c` program verifies that the contents of the value block that the DLM has written to the location that is specified by the `vb` parameter are empty.

7 The `api_ex_master.c` program waits for you to start the `api_ex_client.c` program. It will resume when its Exclusive mode (`DLM_EXMODE`) lock on the `dist shared resource` receives blocking notification that it is blocking a lock request on the same resource from the `api_ex_client.c` program.

8 After you start it, the `api_ex_client.c` program calls the `dlm_nsjoin` function to join the `uid` namespace; that is, the same namespace that the process that is running the `api_ex_master.c` program previously joined.

9 The `api_ex_client.c` program, like the `api_ex_master.c` program, calls the `dlm_set_signal` function to specify that the DLM is to use the SIGIO signal to send completion and blocking notifications to this process.

10 The `api_ex_client.c` program calls the `dlm_lock` function to obtain a Null mode (`DLM_NLMODE`) lock on the same resource on which the process that is running the `api_ex_master.c` already holds an Exclusive mode lock. The `DLM_SYNCSTS` flag causes DLM to return `DLM_SYNCH` status if the lock request is granted immediately. This lock request should be granted immediately, because the Null mode (NL) lock is compatible with the previously granted Exclusive mode lock. This function call also specifies the `DLM_VALB` flag and a pointer

to a lock value block. The DLM copies the resource's lock value to this location when the lock that is requested by the `dlm_lock` function call is granted.

11 The `api_ex_client.c` program examines the contents of the value block that the DLM has written to the location that is specified by the `vb` parameter. The value block should be empty because the `api_ex_master.c` program has not yet written to it.

12 The `api_ex_client.c` program calls the `dlm_cvt` function to convert its Null mode lock on the resource to Exclusive mode. It specifies a blocking notification routine named `blkrtn`. Because the process that is running the `api_ex_master.c` program already holds an Exclusive lock on this resource, it is blocking the `api_ex_client.c` program's lock conversion request. However, because the Exclusive mode lock that is taken out by the `api_ex_master.c` program specifies a blocking notification routine, the DLM uses the SIGIO signal to send the process that is running the `api_ex_master.c` program a blocking notification, triggering its blocking notification routine (`blk_and_go`).

13 The `blk_and_go` routine sleeps for 3 seconds and then sets the `done` flag, which causes the `api_ex_master.c` program to resume.

14 The `api_ex_master.c` program writes the string `abc` to its local copy of the resource's value block.

15 The `api_ex_master.c` program calls the `dlm_cvt` function to write to the lock value block. To do so, it converts its Exclusive mode lock on the resource to Exclusive mode (`DLM_EXMODE`), specifying the lock ID, the location of its copy of the value block, and the `DLM_VALB` flag as parameters to the function call. The `DLM_SYNCSTS` flag causes DLM to return `DLM_SYNCH` status if the lock request is granted immediately. This lock conversion request should be granted immediately because the process already holds an Exclusive mode lock on the resource.

The `dlm_cvt` function call also specifies the `blkrtn` routine as a blocking notification routine. The DLM will call this blocking notification routine immediately because this Exclusive mode lock on the resource blocks the lock conversion request from the `api_ex_client.c` program.

16 The `api_ex_master.c` program's `blkrtn` routine runs and immediately tries to downgrade its lock on the resource from Exclusive mode to Null mode by calling the `dlm_cvt` function. This call should succeed immediately.

17 As soon as this conversion takes place, the `api_ex_client.c` program's lock conversion request succeeds. (The Null mode lock that is held by the process running the `api_ex_master.c` program is compatible with the Exclusive mode lock that is now held by the process

running the api_ex_client.c program.) In upgrading the Null mode lock to Exclusive mode, the DLM copies the resource lock value block to the process that is running the api_ex_client.c program. At this point, the api_ex_client.c program can see the abc text string that the api_ex_master.c program wrote previously to the resource's lock value block.

18 The api_ex_client.c program goes to sleep waiting for a blocking notification.

19 The api_ex_master.c program, which has been sleeping since it downgraded its lock on the dist shared resource resource, calls the dlm_cvt function to convert its Null mode lock on the resource to protected read (DLM_PRMODE) mode. Because the process that is running the api_ex_client.c program already holds an Exclusive lock on this resource, it is blocking the api_ex_master.c program's lock conversion request. (That is, the Exclusive mode and protected read locks are incompatible.) However, because the Exclusive mode lock that is taken out by the api_ex_client.c program specifies a blocking notification routine, the DLM delivers it a blocking notification by sending it a SIGIO signal, triggering its blocking notification routine (blkrtn).

20 The blkrtn blocking notification routine in the api_ex_client.c program sleeps for a few seconds and writes the text string efg to its local copy of the resource's value block.

21 The blkrtn blocking notification routine in the api_ex_client.c program calls the dlm_unlock function to release its lock on the resource. In specifying the address of its local copy of the resource's lock value block and the DLM_VALB flag, it requests the DLM to write the local copy of the value block to the resource when its lock granted mode is Protected Write (DLM_PWMODE) or Exclusive (DLM_EXMODE). The granted mode here is DLM_EXMODE so the local copy of the value block will be written to the resource's lock value block.

22 The api_ex_client.c program completes and exists.

23 As soon as the process that is running the api_ex_client.c program releases its lock on the resource, the api_ex_master.c program's lock conversion request succeeds. In upgrading the Null mode lock to protected read mode, the DLM copies the resource lock value block to the process that is running the api_ex_master.c program. At this point, the api_ex_master.c program can see the efg text string that the api_ex_client.c program wrote previously to the resource's lock value block.

24 The api_ex_master.c program completes and exits.

10 Memory Channel Application Programming Interface Library

The Memory Channel Application Programming Interface (API) implements highly efficient memory sharing between Memory Channel API cluster members, with automatic error-handling, locking, and UNIX style protections. This chapter contains information to help you develop applications based on the Memory Channel API library. It explains the differences between Memory Channel address space and traditional shared memory, and describes how programming using Memory Channel as a transport differs from programming using shared memory as a transport.

This chapter also contains examples that show how to use the Memory Channel API library functions in programs. You will find these code files in the `/usr/examples/cluster/` directory. Each file contains compilation instructions.

The chapter discusses the following topics:

- Initializing the Memory Channel API library (Section 10.1)
- Understanding the Memory Channel multirail model (Section 10.2)
- Tuning your Memory Channel configuration (Section 10.3)
- Troubleshooting (Section 10.4)
- Initializing the Memory Channel API library for a user program (Section 10.5)
- Accessing Memory Channel address space (Section 10.6)
- Using clusterwide locks (Section 10.7)
- Using cluster signals (Section 10.8)
- Accessing cluster information (Section 10.9)
- Comparing shared memory and message passing models (Section 10.10)
- Answering questions asked by programmers who use the Memory Channel API to develop programs for TruCluster Server systems (Section 10.11)

10.1 Initializing the Memory Channel API Library

To run applications that are based on the Memory Channel API library, the library must be initialized on each host in the Memory Channel API cluster. The `imc_init` command initializes the Memory Channel API library and allows applications to use the API. Initialization of the Memory Channel API library occurs either by automatic execution of the `imc_init` command at system boot time, or when the system administrator invokes the command from the command line after the system boots.

Initialization of the Memory Channel API library at system boot time is controlled by the `IMC_AUTO_INIT` variable in the `/etc/rc.config` file. If the value of this variable is set to 1, the `imc_init` command is invoked at system boot time. When the Memory Channel API library is initialized at boot time, the values of the `-a` *`maxalloc`* and `-r` *`maxrecv`* flags are set to the values that are specified by the `IMC_MAX_ALLOC` and `IMC_MAX_RECV` variables in the `/etc/rc.config` file. The default value for the *`maxalloc`* parameter and the *`maxrecv`* parameter is 10 MB.

If the `IMC_AUTO_INIT` variable is set to zero (0), the Memory Channel API library is not initialized at system boot time. The system administrator must invoke the `imc_init` command to initialize the library. The parameter values in the `/etc/rc.config` file are not used when the `imc_init` command is manually invoked.

The `imc_init` command initializes the Memory Channel API library the first time it is invoked, whether this happens at system boot time or after the system has booted. The value of the `-a` *`maxalloc`* flag must be the same on all hosts in the Memory Channel API cluster. If different values are specified, the maximum value that is specified for any host determines the clusterwide value that applies to all hosts.

After the Memory Channel API library has been initialized on the current host, the system administrator can invoke the `imc_init` command again to reconfigure the values of the *`maxalloc`* and *`maxrecv`* resource limits, without forcing a reboot. The system administrator can increase or decrease either limit, but the new limits cannot be lower than the current usage of the resources. Reconfiguring the cluster from the command line does not read or modify the values that are specified in the `/etc/rc.config` file. The system administrator can use the `rcmgr`(8) command to modify the parameters and have them take effect when the system reboots.

You must have root privileges to execute the `imc_init` command.

10.2 The Memory Channel Multirail Model

The Memory Channel multirail model supports the concept of physical rails and logical rails. A physical rail is defined as a Memory Channel hub with its cables and Memory Channel adapters, and the Memory Channel driver for the adapters on each node. A logical rail is made up of one or two physical rails.

A cluster can have one or more logical rails up to a maximum of four. Logical rails can be configured in the following styles:

- Single-rail (Section 10.2.1)
- Failover pair (Section 10.2.2)

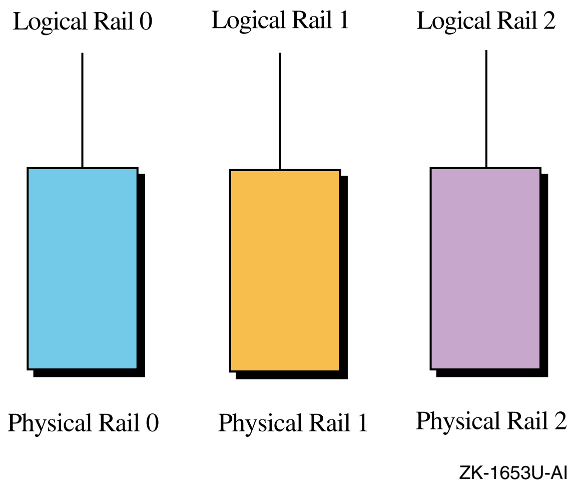
10.2.1 Single-Rail Style

If a cluster is configured in the single-rail style, there is a one-to-one relationship between physical rails and logical rails. This configuration has no failover properties; if the physical rail fails, the logical rail fails.

A benefit of the single-rail configuration is that applications can access the aggregate address space of all logical rails and utilize their aggregate bandwidth for maximum performance.

Figure 10–1 shows a single-rail Memory Channel configuration with three logical rails, each of which is also a physical rail.

Figure 10–1: Single-Rail Memory Channel Configuration



10.2.2 Failover Pair Style

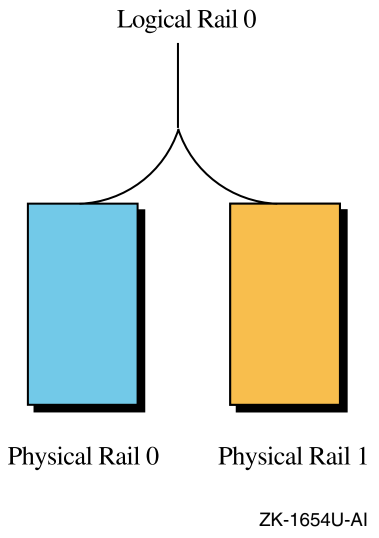
If a cluster is configured in the failover pair style, a logical rail consists of two physical rails, with one physical rail active and the other inactive. If the active physical rail fails, a failover takes place and the inactive physical rail is used, allowing the logical rail to remain active after the failover. This failover is transparent to the user.

The failover pair style can only exist in a Memory Channel configuration consisting of two physical rails.

The failover pair configuration provides availability in the event of a physical rail failure, because the second physical rail is redundant. However, only the address space and bandwidth of a single physical rail are available at any given time.

Figure 10–2 shows a multirail Memory Channel configuration in the failover pair style. The illustrated configuration has one logical rail, which is made up of two physical rails.

Figure 10–2: Failover Pair Memory Channel Configuration



10.2.3 Configuring the Memory Channel Multirail Model

When you implement the Memory Channel multirail model, all nodes in a cluster must be configured with an equal number of physical rails, which are configured into an equal number of logical rails, each with the same failover style.

The system configuration parameter `rm_rail_style`, in the `/etc/sysconfigtab` file, is used to set multirail styles. The `rm_rail_style` parameter can be set to one of the following values:

- Zero (0) for a single-rail style
- 1 for a failover pair style

The default value of the rm_rail_style parameter is 1.

The rm_rail_style parameter must have the same value for all nodes in a cluster, or configuration errors may occur.

To change the value of the rm_rail_style parameter to zero (0) for a single-rail style, change the /etc/sysconfigtab file by adding or modifying the following stanza for the rm subsystem:

```
rm:  rm_rail_style=0
```

Note

We recommend that you use sysconfigdb(8) to modify or to add stanzas in the /etc/sysconfigtab file.

If you change the rm_rail_style parameter, you must halt the entire cluster and then reboot each member system.

Note

A cluster will fail if any logical rail fails. See Section 10.4.3 for more information.

Error handling for the Memory Channel multirail model is implemented for specified logical rails. See Section 10.6.6 for a description of Memory Channel API library error-management functions and code examples.

Note

The Memory Channel multirail model does not facilitate any type of cluster reconfiguration, such as the addition of hubs or Memory Channel adapters. For such reconfiguration, you must first shut down the cluster completely.

10.3 Tuning Your Memory Channel Configuration

The imc_init command initializes the Memory Channel API library with certain resource defaults. Depending on your application, you may require more resources than the defaults allow. In some cases, you can change certain Memory Channel parameters and virtual memory resource parameters to overcome these limitations. The following sections describe these parameters and explain how to change them.

10.3.1 Extending Memory Channel Address Space

The amount of total Memory Channel address space that is available to the Memory Channel API library is specified using the *maxalloc* parameter of the `imc_init` command. The maximum amount of Memory Channel address space that can be attached for receive on a host is specified using the *maxrecv* parameter of the `imc_init` command. The default limit in each case is 10 MB. (Section 10.1 describes how to initialize the Memory Channel API library using the `imc_init` command.)

You can use the `rcmgr`(8) command to change the value that is used during an automatic initialization by setting the variables `IMC_MAX_ALLOC` and `IMC_MAX_RECV`. For example, you can set the variables to allow a total of 80 MB of Memory Channel address space to be made available to the Memory Channel API library clusterwide, and to allow 60 MB of Memory Channel address space to be attached for receive on the current host, as follows:

```
   rcmgr set IMC_MAX_ALLOC 80
   rcmgr set IMC_MAX_RECV 60
```

If you use the `rcmgr`(8) command to set new limits, they will take effect when the system reboots.

You can use the Memory Channel API library initialization command, `imc_init`, to change both the amount of total Memory Channel address space available and the maximum amount of Memory Channel address space that can be attached for receive, after the Memory Channel API library has been initialized. For example, to allow a total amount of 80 MB of Memory Channel address space to be made available clusterwide, and to allow 60 MB of Memory Channel address space to be attached for receive on the current host, use the following command:

```
 imc_init -a 80 -r 60
```

If you use the `imc_init` command to set new limits, they will be lost when the system reboots, and the values of the `IMC_MAX_ALLOC` and `IMC_MAX_RECV` variables will be used as limits.

10.3.2 Increasing Wired Memory

Every page of Memory Channel address space that is attached for receive must be backed by a page of physical memory on your system. This memory is nonpageable; that is, it is wired memory. The amount of wired memory on a host cannot be increased infinitely; the system configuration parameter `vm_syswiredpercent` will impose a limit. You can change the `vm_syswiredpercent` parameter in the `/etc/sysconfigtab` file.

For example, if you want to set the `vm_syswiredpercent` parameter to 80, the `vm` stanza in the `/etc/sysconfigtab` file must contain the following entry:

```
vm:  vm_syswiredpercent=80
```

If you change the `vm_syswiredpercent` parameter, you must reboot the system.

Note

The default amount of wired memory is sufficient for most operations. We recommend that you exercise caution in changing this limit.

10.4 Troubleshooting

The following sections describe error conditions that you may encounter when using the Memory Channel API library functions, and suggest solutions.

10.4.1 IMC_NOTINIT Return Code

The `IMC_NOTINIT` status is returned when the `imc_init` command has not been run, or when the `imc_init` command has failed to run correctly.

The `imc_init` command must be run on each host in the Memory Channel API cluster before you can use the Memory Channel API library functions. (Section 10.1 describes how to initialize the Memory Channel API library using the `imc_init` command.)

If the `imc_init` command does not run successfully, see Section 10.4.2 for suggested solutions.

10.4.2 Memory Channel API Library Initialization Failure

The Memory Channel API library may fail to initialize on a host; if this happens, an error message is displayed on the console and is written to the `messages` log file in the `/usr/var/adm` directory. Use the following list of error messages and solutions to eliminate the error:

- `Memory Channel is not initialized for user access`

  This error message indicates that the current host has not been initialized to use the Memory Channel API.

  To solve this problem, ensure that all Memory Channel cables are correctly attached to the Memory Channel adapters on this host. See Section 10.4.3 for more information on fatal errors that are caused by problems with the physical Memory Channel configuration or interconnect.

- `Memory Channel API - insufficient wired memory`

  This error message indicates that the value of the `IMC_MAX_RECV` variable in the `/etc/config` file or the value of the `-r` option to the `imc_init` command is greater than the wired memory limit that is specified by the configuration parameter `vm_syswiredpercent`.

  To solve this problem, invoke the `imc_init` command with a smaller value for the *maxrecv* parameter, or increase the system wired memory limit as described in Section 10.3.2.

10.4.3 Fatal Memory Channel Errors

Sometimes the Memory Channel API fails to initialize because of problems with the physical Memory Channel configuration or interconnect. Error messages that are displayed on the console in these circumstances do not mention the Memory Channel API. The following sections describe some of the more common reasons for such failures.

10.4.3.1 Logical Rail Failure

If any logical rail fails, a system panic occurs on one or more hosts in the cluster, and the following error message is displayed on the console:

```
panic (cpu 0): rm_delete_context: fatal MC error
```

To solve this problem, ensure that the hub is powered up and that all cables are connected properly; then halt the entire cluster and reboot each member system.

10.4.3.2 Logical Rail Initialization Failure

If the logical rail configuration for a logical rail on this node does not match that of a logical rail on other cluster members, a system panic occurs on one or more hosts in the cluster, and error messages of the following form are displayed on the console:

```
rm_slave_init
rail configuration does not match cluster expectations for logical rail
0
logical rail 0 has failed initialization
rm_delete_context: lcsr = 0x2a80078, mcerr = 0x20001, mcport =
0x72400001
panic (cpu 0): rm_delete_context: fatal MC error
```

This error can occur if the configuration parameter `rm_rail_style` is not identical on every node.

To solve this problem, follow these steps:

1. Halt the system.
2. Boot `/genvmunix`.
3. Modify the `/etc/sysconfigtab` file as described in Section 10.2.3.
4. Reboot the kernel with Memory Channel API cluster support (`/vmunix`).

10.4.4 IMC_MCFULL Return Code

The `IMC_MCFULL` status is returned if there is not enough Memory Channel address space to perform an operation.

The amount of total Memory Channel address space that is available to the Memory Channel API library is specified by using the *`maxalloc`* parameter of the `imc_init` command, as described in Section 10.4.2.

You can use the `rcmgr`(8) command or the Memory Channel API library initialization command, `imc_init`, to increase the amount of Memory Channel address space that is available to the library clusterwide. See Section 10.3.1 for more details.

10.4.5 IMC_RXFULL Return Code

The `IMC_RXFULL` status is returned by the `imc_asattach` function, if receive mapping space is exhausted when an attempt is made to attach a region for receive.

Note

The default amount of receive space on the current host is 10 MB.

The maximum amount of Memory Channel address space that can be attached for receive on a host is specified using the *`maxrecv`* parameter of the `imc_init` command, as described in Section 10.1.

You can use the `rcmgr`(8) command or the Memory Channel API library initialization command, `imc_init`, to extend the maximum amount of Memory Channel address space that can be attached for receive on the host. See Section 10.3.1 for more details.

10.4.6 IMC_WIRED_LIMIT Return Code

The `IMC_WIRED_LIMIT` return value indicates that an attempt has been made to exceed the maximum quantity of wired memory.

The system configuration parameter `vm_syswiredpercent` specifies the wired memory limit; see Section 10.3.2 for information on changing this limit.

10.4.7 IMC_MAPENTRIES Return Code

The `IMC_MAPENTRIES` return value indicates that the maximum number of virtual memory map entries has been exceeded for the current process.

10.4.8 IMC_NOMEM Return Code

The `IMC_NOMEM` return status indicates a `malloc` function failure while performing a Memory Channel API function call.

This will happen if process virtual memory has been exceeded, and can be remedied by using the usual techniques for extending process virtual memory limits; that is, by using the `limit` command and the `unlimit` command for the C shell, and by using the `ulimit` command for the Bourne shell and the Korn shell.

10.4.9 IMC_NORESOURCES Return Code

The `IMC_NORESOURCES` return value indicates that there are insufficient Memory Channel data structures available to perform the required operation. However, the amount of available Memory Channel data structures is fixed, and cannot be increased by changing a parameter. To solve this problem, amend the application to use fewer regions or locks.

10.5 Initializing the Memory Channel API Library for a User Program

The `imc_api_init` function is used to initialize the Memory Channel API library in a user program. Call the `imc_api_init` function in a process before any of the other Memory Channel API functions are called. If a process forks, the `imc_api_init` function must be called before calling any other API functions in the child process, or an undefined behavior will result.

10.6 Accessing Memory Channel Address Space

The Memory Channel interconnect provides a form of memory sharing between Memory Channel API cluster members. The Memory Channel API library is used to set up the memory sharing, allowing processes on different members of the cluster to exchange data using direct read and write operations to addresses in their virtual address space. When the memory sharing has been set up by the Memory Channel API library, these direct read and write operations take place at hardware speeds without involving the operating system or the Memory Channel API library software functions.

When a system is configured with Memory Channel, part of the physical address space of the system is assigned to the Memory Channel address space. The size of the Memory Channel address space is specified by the `imc_init` command. A process accesses this Memory Channel address space by using the Memory Channel API to map a region of Memory Channel address space to its own virtual address space.

Applications that want to access the Memory Channel address space on different cluster members can allocate part of the address space for a particular purpose by calling the `imc_asalloc` function. The *key* parameter associates a clusterwide key with the region. Other processes that allocate the same region also specify this key. This allows processes to coordinate access to the region.

To use an allocated region of Memory Channel address space, a process maps the region into its own process virtual address space, using the `imc_asattach` function or the `imc_asattach_ptp` function. When a process attaches to a Memory Channel region, an area of virtual address space that is the same size as the Memory Channel region is added to the process virtual address space. When attaching the region, the process indicates whether the region is mapped to receive or transmit data, as follows:

- Transmit — Indicates that the region is to be used to transmit data on Memory Channel. When a process writes to addresses in this virtual address region, the data is transmitted over the Memory Channel interconnect to the other members of the Memory Channel API cluster.

  To map a region for transmit, specify the value `IMC_TRANSMIT` for the *dir* parameter to the `imc_asattach` function.

- Receive — Indicates that the region is to be used to receive data from Memory Channel. In this case, the address space that is mapped into the process virtual address space is backed by a region of physical memory on the system. When data is transmitted on Memory Channel, it is written into the physical memory of any hosts that have mapped the region for receive, so that processes on that system read from the same area of physical memory. The process does not receive any data that is transmitted before the region is mapped.

  To map a region for receive, use the value `IMC_RECEIVE` as the *dir* parameter for the `imc_asattach` function.

A process can attach to a Memory Channel region in broadcast mode, point-to-point mode, or loopback mode. These methods of attach are described in Section 10.6.1.

Memory sharing using the Memory Channel interconnect is similar to conventional shared memory in that, after it is established, simple accesses to virtual address space allow two different processes to share data. However, there are two differences between these memory-sharing mechanisms that you must allow for, as follows:

- When conventional shared memory is created, it is assigned a virtual address. In C programming terms, there is a pointer to the memory. This single pointer can be used both to read and write data to the shared memory. However, a Memory Channel region can have two different virtual addresses assigned to it: a transmit virtual address and a receive virtual address. In C programming terms, there are two different pointers to manage; one pointer can only be used for write operations, the other pointer is used for read operations.
- In conventional shared memory, write operations are made directly to memory and are immediately visible to other processes that are reading from the same memory. However, when a write operation is made to a Memory Channel region, the write operation is not made directly to memory but to the I/O system and the Memory Channel hardware. This means that there is a delay before the data appears in memory on the receiving system. This is described in more detail in Section 10.6.5.

10.6.1 Attaching to Memory Channel Address Space

The following sections describe the ways in which a process can attach to Memory Channel address space. There are three ways in which a process can attach to Memory Channel address space, as follows:

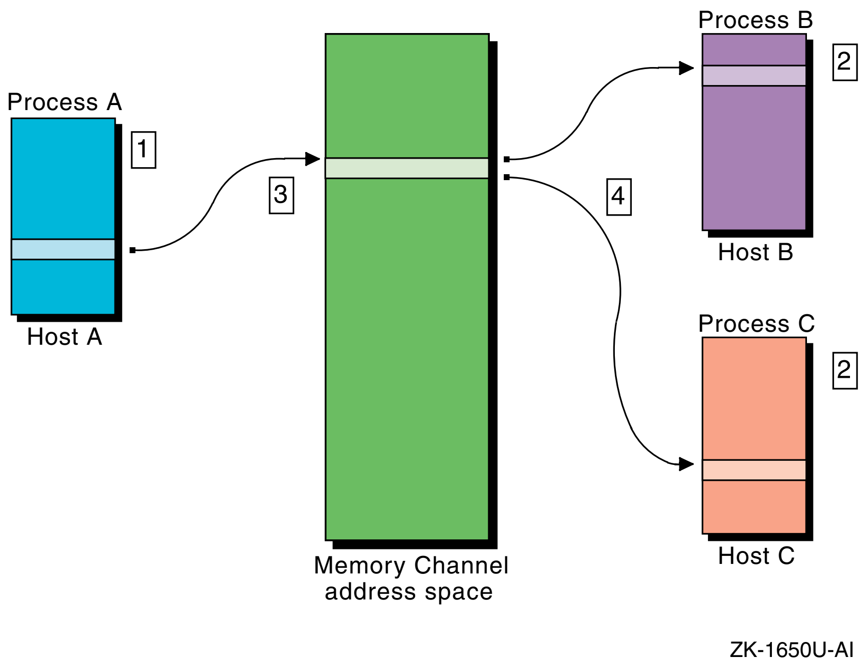
- Broadcast attach (Section 10.6.1.1)
- Point-to-point attach (Section 10.6.1.2)
- Loopback attach (Section 10.6.1.3)

This section also explains initial coherency, reading and writing Memory Channel regions, latency-related coherency, and error management, and includes some code examples.

10.6.1.1 Broadcast Attach

When one process maps a region for transmit and other processes map the same region for receive, the data that the transmit process writes to the region is transmitted on Memory Channel to the receive memory of the other processes. Figure 10–3 shows a three-host Memory Channel implementation that shows how the address spaces are mapped.

Figure 10–3: Broadcast Address Space Mapping



With the address spaces that are mapped as shown in Figure 10–3, note the following:

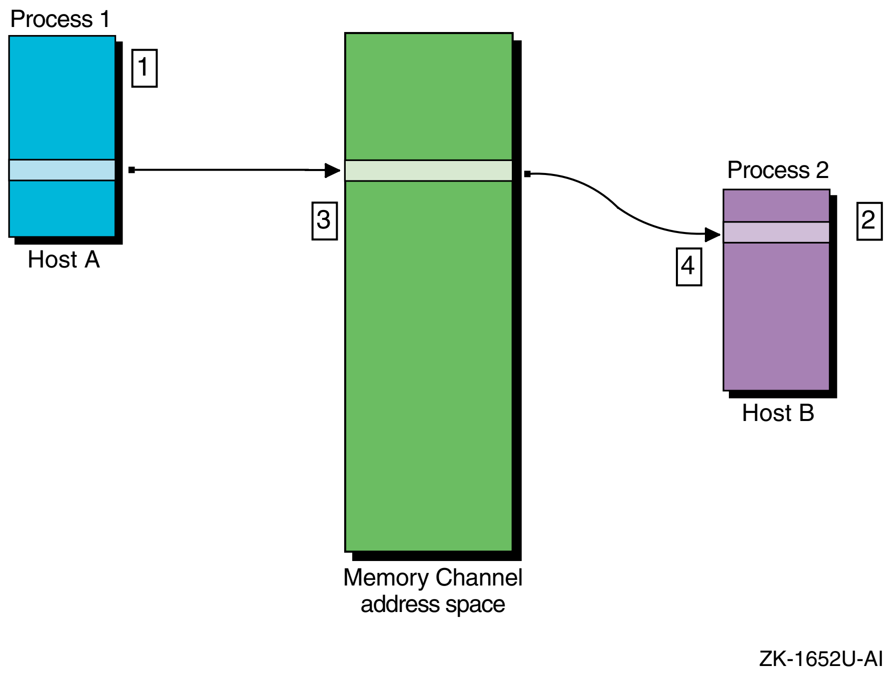
1. Process A allocates a region of Memory Channel address space. Process A then maps the allocated region to its virtual address space when it attaches the region for transmit using the `imc_asattach` function.
2. Process B and Process C both allocate the same region of Memory Channel address space as Process A. However, unlike Process A, Process B and Process C both attach the region to receive data.
3. When data is written to the virtual address space of Process A, the data is transmitted on Memory Channel.
4. When the data from Process A appears on Memory Channel, it is written to the physical memory on Hosts B and C that backs the virtual address spaces of Processes B and C that were allocated to receive the data.

10.6.1.2 Point-to-Point Attach

An allocated region of Memory Channel address space can be attached for transmit in point-to-point mode to the virtual address space of a process on another node. This is done by calling the `imc_asattach_ptp` function with a specified host as a parameter. This means that writes to the region are sent only to the host that is specified in the parameter, and not to all hosts in the cluster.

Regions that are attached using the `imc_asattach_ptp` function are always attached in transmit mode, and are write-only. Figure 10–4 shows a two-host Memory Channel implementation that shows point-to-point address space mapping.

Figure 10–4: Point-to-Point Address Space Mapping



With the address spaces mapped as shown in Figure 10–4, note the following:

1. Process 1 allocates a region of Memory Channel address space. It then maps the allocated region to its virtual address space when it attaches the region point-to-point to Host B using the `imc_asattach_ptp` function.
2. Process 2 allocates the region and then attaches it for receive using the `imc_asattach` function.
3. When data is written to the virtual address space of Process 1, the data is transmitted on Memory Channel.
4. When the data from Process 1 appears on Memory Channel, it is written to the physical memory that backs the virtual address space of Process 2 on Host B.

10.6.1.3 Loopback Attach

A region can be attached for both transmit and receive by processes on a host. Data that is written by the host is written to other hosts that have attached the region for receive. However, by default, data that is written by the host is not also written to the receive memory on that host; it is written only to other hosts. If you want a host to see data that it writes, you must specify the `IMC_LOOPBACK` flag to the `imc_asattach` function when attaching for transmit.

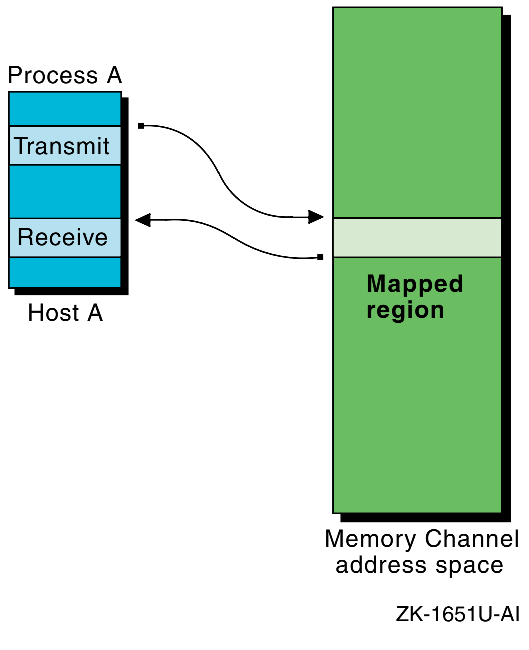
The loopback attribute of a region is set up on a per-host basis, and is determined by the value of the `flag` parameter to the first transmit attach on that host.

If you specify the value `IMC_LOOPBACK` for the `flag` parameter, two Memory Channel transactions occur for every write, one to write the data and one to loop the data back.

Because of the nature of point-to-point attach mode, looped-back writes are not permitted.

Figure 10–5 shows a configuration in which a region of Memory Channel address space is attached both for transmit with loopback and for receive.

Figure 10–5: Loopback Address Space Mapping



10.6.2 Initial Coherency

When a Memory Channel region is attached for receive, the initial contents are undefined. This situation can arise because a process that has mapped the same Memory Channel region for transmit might update the contents of the region before other processes map the region for receive. This is referred to as the initial coherency problem. You can overcome this in two ways:

- Write the application in a way that ensures that all processes attach the region for receive before any processes write to the region.
- At allocation time, specify that the region is coherent by specifying the `IMC_COHERENT` flag when you allocate the region using the `imc_asalloc` function. This ensures that all processes will see every update to the region, regardless of when the processes attach the region.

  Coherent regions use the loopback feature. This means that two Memory Channel transactions occur for every write, one to write the data and one to loop the data back; because of this, coherent regions have less available bandwidth than noncoherent regions.

10.6.3 Reading and Writing Memory Channel Regions

Processes that attach a region of Memory Channel address space can only write to a transmit pointer, and can only read from a receive pointer. Any attempt to read a transmit pointer will result in a segmentation violation.

Apart from explicit read operations on Memory Channel transmit pointers, segmentation violations will also result from operations that cause the compiler to generate read-modify-write cycles; for example:

- Postincrement and postdecrement operations.
- Preincrement and predecrement operations.
- Assignment to simple data types that are not an integral multiple of four bytes.
- Use of the `bcopy`(3) library function, where the `length` parameter is not an integral multiple of eight bytes, or where the source or destination arguments are not 8-byte aligned.
- Assignment to structures that are not quadword-aligned (that is, the value returned by the `sizeof` function is not an integral multiple of eight). This refers only to unit assignment of the whole structure; for example, `mystruct1 = mystruct2`.

10.6.4 Address Space Example

Example 10–1 shows how to initialize, allocate, and attach to a region of Memory Channel address space, and also shows two of the differences between Memory Channel address space and traditional shared memory:

- Initial coherency, as described in Section 10.6.2
- Asymmetry of receive and transmit regions, as described in Section 10.6.3

The sample program shown in Example 10–1 executes in master or slave mode, as specified by a command-line parameter. In master mode, the program writes its own process identifier (PID) to a data structure in the global Memory Channel address space. In slave mode, the program polls a data structure in the Memory Channel address space to determine the PID of the master process.

Note

Make sure that your programs are flexible in their use of keys to prevent problems resulting from key clashes. We recommend that you use meaningful, application-specific keys.

Example 10–1: Accessing Regions of Memory Channel Address Space

```
/* /usr/examples/cluster/mc_ex1.c */

#include <c_asm.h>
#include <sys/types.h>
#include <sys/imc.h>
#define VALID 756

main (int argc, char *argv[])
{
    imc_asid_t   glob_id;
    typedef struct {
                   pid_t     pid;
                   volatile int valid; 1
                  } clust_pid;

    clust_pid   *global_record;
    caddr_t     add_rx_ptr = 0, add_tx_ptr = 0;
    int         status;
    int         master;
    int         logical_rail=0;

    /* check for correct number of arguments /*

    if (argc != 2) {
                   printf("usage: mcpid 0|1\n");
                   exit(-1);
                  }

    /* test if process is master or slave */
```

Example 10–1: Accessing Regions of Memory Channel Address Space (cont.)

```
    master  = atoi(argv[1]); 2


    /* initialize Memory Channel API library */

    status  = imc_api_init(NULL); 3


    if (status < 0) {
                     imc_perror("imc_api_init::",status); 4
                     exit(-2);
                    }

    imc_asalloc(123, 8192, IMC_URW, 0, &glob_id,
                         logical_rail); 5

    if (master) {
                 imc_asattach(glob_id, IMC_TRANSMIT, IMC_SHARED,
                              0, &add_tx_ptr); 6

                 global_record = (clust_pid*)add_tx_ptr; 7
                 global_record->pid = getpid();
                 mb(); 8
                 global_record->valid = VALID;
                 mb();
                }

    else        {  /* secondary process */

                 imc_asattach(glob_id, IMC_RECEIVE, IMC_SHARED,
                              0, &add_rx_ptr); 9

                 (char*)global_record  = add_rx_ptr;

                 while ( global_record->valid != VALID)
                       ;  /* continue polling */ 10

                 printf("pid of master process is %d\n",
                        global_record->pid);
                }
   imc_asdetach(glob_id);
   imc_asdealloc(glob_id); 11
}
```

$\boxed{1}$ The `valid` flag is declared as volatile to prevent the compiler from performing any optimizations that might prevent the code from reading the updated PID value from memory.

$\boxed{2}$ The first argument on the command line indicates whether the process is a master (argument equal to 1) or a slave process (argument not not equal to 1).

3 The `imc_api_init` function initializes the Memory Channel API library. Call it before calling any of the other Memory Channel API library functions.

4 All Memory Channel API library functions return a zero (0) status if successful. The `imc_perror` function decodes error status values. For brevity, this example ignores the status from all functions other than the `imc_api_init` function.

5 The `imc_asalloc` function allocates a region of Memory Channel address space with the following characteristics:

- `key=123` — This value identifies the region of Memory Channel address space. Other applications that attach this region will use the same key value.
- `size=8192` — The size of the region is 8192 bytes.
- `perm=IMC_URW` — The access permission on the region is user read and write.
- `id=glob_id` — The `imc_asalloc` function returns this value, which uniquely identifies the allocated region. The program uses this value in subsequent calls to other Memory Channel functions.
- `logical_rail=0` — The region is allocated using Memory Channel logical rail zero (0).

6 The master process attaches the region for transmit by calling the `imc_asattach` function and specifying the `glob_id` identifier, which was returned by the call to the `imc_asalloc` function. The `imc_asattach` function returns `add_tx_ptr`, a pointer to the address of the region in the process virtual address space. The `IMC_SHARED` value signifies that the region is shareable, so other processes on this host can also attach the region.

7 The program points the global record structure to the region of virtual memory in the process virtual address space that is backed by the Memory Channel reason, and writes the process ID in the `pid` field of the global record. Note that the master process has attached the region for transmit; therefore, it can only write data in the field. An attempt to read the field will result in a segmentation violation; for example:

```
(pid_t)x = global_record->pid;
```

8 The program uses memory barrier instructions to ensure that the `pid` field is forced out of the Alpha CPU write buffer before the `VALID` flag is set.

9 The slave process attaches the region for receive by calling the `imc_asattach` function and specifying the `glob_id` identifier, which was returned by the call to the `imc_asalloc` function. The

imc_asattach function returns add_rx_ptr, a pointer to the address of the region in the process virtual address space. On mapping, the contents of the region may not be consistent on all processes that map the region. Therefore, start the slave process before the master to ensure that all writes by the master process appear in the virtual address space of the slave process.

10 The slave process overlays the region with the global record structure and polls the valid flag. The earlier declaration of the flag as volatile ensures that the flag is immune to compiler optimizations, which might result in the field being stored in a register. This ensures that the loop will load a new value from memory at each iteration and will eventually detect the transition to VALID.

11 At termination, the master and slave processes explicitly detach and deallocate the region by calling the imc_asdetach function and the imc_asdealloc function. In the case of abnormal termination, the allocated regions are automatically freed when the processes exit.

10.6.5 Latency Related Coherency

As described in Section 10.6.2, the initial coherency problem can be overcome by retransmitting the data after all mappings of the same region for receive have been completed, or by specifying at allocation time that the region is coherent. However, when a process writes to a transmit pointer, several microseconds can elapse before the update is reflected in the physical memory that corresponds to the receive pointer. If the process reads the receive pointer during that interval, the data it reads might be incorrect. This is known as the latency-related coherency problem.

Latency problems do not arise in conventional shared memory systems. Memory and cache control ensure that store and load instructions are synchronized with data transfers.

Example 10–2 shows two versions of a program that decrements a global process count and detects the count reaching zero (0). The first program uses System V shared memory and interprocess communication. The second uses the Memory Channel API library.

Example 10–2: System V IPC and Memory Channel Code Comparison

```
/* /usr/examples/cluster/mc_ex2.c */

/****************************************
 *********  System V IPC example  *******
 ****************************************/

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
main()
```

Example 10–2: System V IPC and Memory Channel Code Comparison (cont.)

```
{
   typedef struct {
                   int  proc_count;
                   int  remainder[2047]
                  } global_page;
   global_page  *mypage;
   int  shmid;

   shmid = shmget(123, 8192, IPC_CREAT |  SHM_R |  SHM_W);

   (caddr_t)mypage = shmat(shmid, 0,  0);  /* attach the
                                              global region     */

   mypage->proc_count ++;               /* increment process
                                           count                */

   /* body of program goes here */
           .
           .
           .
   /* clean up */

   mypage->proc_count --;               /* decrement process
                                           count                */
   if (mypage->proc_count == 0 )
                printf("The last process is exiting\n");
           .
           .
           .
}

/****************************************
 *******  Memory Channel example  *******
 ****************************************/

#include <sys/types.h>
#include <sys/imc.h>
main()
{
   typedef struct {
                   int  proc_count;
                   int  remainder[2047]
                  } global_page;
   global_page  *mypage_rx, *mypage_tx; 1
   imc_asid_t   glob_id;
   int          logical_rail=0;
   int          temp;

   imc_api_init(NULL);

   imc_asalloc(123, 8192, IMC_URW | IMC_GRW, 0, &glob_id,
               logical_rail); 2

   imc_asattach(glob_id, IMC_TRANSMIT, IMC_SHARED,
                IMC_LOOPBACK, &(caddr_t)mypage_tx); 3

   imc_asattach(glob_id, IMC_RECEIVE, IMC_SHARED,
                0, &(caddr_t)mypage_rx); 4
```

Example 10–2: System V IPC and Memory Channel Code Comparison (cont.)

```
   /* increment process count */

   mypage_tx->proc_count = mypage_rx->proc_count + 1; [5]

   /* body of program goes here */
           .
           .
           .
   /* clean up */

   /* decrement process count

   temp = mypage_rx->proc_count - 1 [6]

   mypage_tx->proc_count = temp;

   /* wait for MEMORY CHANNEL update to occur */

   while (mypage_rx->proc_count != temp)
          ;

   if (mypage_rx->proc_count == 0 )
                printf("The last process is exiting\n");
           .
           .
           .
}
```

[1] The process must be able to read the data that it writes to the Memory Channel global address space. Therefore, it declares two addresses, one for transmit and one for receive.

[2] The `imc_asalloc` function allocates a region of Memory Channel address space. The characteristics of the region are as follows:

- `key=123` — This value identifies the region of Memory Channel address space. Other applications that attach this region will use the same key value.
- `size=8192` — The size of the region is 8192 bytes.
- `perm=IMC_URW | IMC_GRW` — The region is allocated with user and group read and write permission.
- `id=glob_id` — The `imc_asalloc` function returns this value, which uniquely identifies the allocated region. The program uses this value in subsequent calls to other Memory Channel API library functions.
- `logical_rail=0` — The region is allocated using Memory Channel logical rail zero (0).

3 This call to the `imc_asattach` function attaches the region for transmit at the address that is pointed to by the `mypage_tx` variable. The value of the `flag` parameter is set to `IMC_LOOPBACK`, so that any time the process writes data to the region, the data is looped back to the receive memory.

4 This call to the `imc_asattach` function attaches the region for receive at the address that is pointed to by the `mypage_rx` variable.

5 The program increments the global process count by adding 1 to the value in the receive pointer, and by assigning the result into the transmit pointer. When the program writes to the transmit pointer, it does not wait to ensure that the write instruction completes.

6 After the body of the program completes, the program decrements the process count and tests that the decremented value was transmitted to the other hosts in the cluster. To ensure that it examines the decremented count (rather than some transient value), the program stores the decremented count in a local variable, `temp`. It writes the decremented count to the transmit region, and then waits for the value in the receive region to match the value in `temp`. When the match occurs, the program knows that the decremented process count has been written to the Memory Channel address space.

In this example, the use of the local variable ensures that the program compares the value in the receive memory with the value that was transmitted. An attempt to use the value in the receive memory before ensuring that the value had been updated may result in erroneous data being read.

10.6.6 Error Management

In a shared memory system, the process of reading and writing to memory is assumed to be error-free. In a Memory Channel system, the error rate is of the order of three errors per year. This is much lower than the error rates of standard networks and I/O subsystems.

The Memory Channel hardware reports detected errors to the Memory Channel software. The Memory Channel hardware provides two guarantees that make it possible to develop applications that can cope with errors:

- It does not write corrupt data to host systems.
- It delivers data to the host systems in the sequence in which the data is written to the Memory Channel hardware.

These guarantees simplify the process of developing reliable and efficient messaging systems.

The Memory Channel API library provides the following functions to help applications implement error management:

- `imc_ckerrcnt_mr` — The `imc_ckerrcnt_mr` function looks for the existence of errors on a specified logical rail on Memory Channel hosts. This allows transmitting processes to learn whether or not errors occur when they send messages.
- `imc_rderrcnt_mr` — The `imc_rderrcnt_mr` function reads the clusterwide error count for the specified logical rail and returns the value to the calling program. This allows receiving processes to learn the error status of messages that they receive.

The operating system maintains a count of the number of errors that occur on the cluster. The system increments the value whenever it detects a Memory Channel hardware error in the cluster, and when a host joins or leaves the cluster.

The task of detecting and processing an error takes a small, but finite, amount of time. This means that the count that is returned by the `imc_rderrcnt_mr` function might not be up-to-date with respect to an error that has just occurred on another host in the cluster. On the local host, the count is always up-to-date.

Use the `imc_rderrcnt_mr` function to implement a simple and effective error-detection mechanism by reading the error count before transmitting a message, and including the count in the message. The receiving process compares the error count in the message body with the local value that is determined after the message arrives. The local value is guaranteed to be up-to-date, so if this value is the same as the transmitted value, then it is certain that no intervening errors occurred. Example 10–3 shows this technique.

Example 10–3: Error Detection Using the imc_rderrcnt_mr Function

```
/* /usr/examples/cluster/mc_ex3.c */

/****************************************
*********  Transmitting Process  *********
*****************************************/

#include <sys/imc.h>
#include <c_asm.h>
main()
{
        typedef struct {
                volatile        int     msg_arrived;
                int     send_count;
                int     remainder[2046];
        } global_page;
        global_page     *mypage_rx, *mypage_tx;
        imc_asid_t      glob_id;
        int             i;
        volatile        int err_count;
```

Example 10–3: Error Detection Using the imc_rderrcnt_mr Function (cont.)

```
        imc_api_init(NULL);

        imc_asalloc (1234, 8192, IMC_URW, 0, &glob_id,0);
        imc_asattach (glob_id, IMC_TRANSMIT, IMC_SHARED, IMC_LOOPBACK,
                      &(caddr_t)mypage_tx);
        imc_asattach (glob_id, IMC_RECEIVE, IMC_SHARED, 0,
                      &(caddr_t)mypage_rx);

        /*  save the error count  */
        while (  (err_count = imc_rderrcnt_mr(0) ) < 0 )
                ;

        mypage_tx->send_count = err_count;

        /* store message data */
        for (i = 0; i < 2046; i++)
                mypage_tx->remainder[i] = i;

        /* now mark as valid */
        mb();

        do {
                mypage_tx->msg_arrived = 1;
        } while (mypage_rx->msg_arrived != 1);  /* ensure no error on
                                                   valid flag          */

}


/****************************************
***********  Receiving Process  **********
*****************************************/

#include <sys/imc.h>
main()
{
        typedef struct {
                volatile        int     msg_arrived;
                int     send_count;
                int     remainder[2046];
        } global_page;
        global_page     *mypage_rx, *mypage_tx;
        imc_asid_t      glob_id;
        int             i;
        volatile        int err_count;

        imc_api_init(NULL);

        imc_asalloc (1234, 8192, IMC_URW, 0, &glob_id,0);
        imc_asattach (glob_id, IMC_RECEIVE, IMC_SHARED, 0,
                      &(caddr_t)mypage_rx);

        /* wait for message arrival */
        while ( mypage_rx->msg_arrived == 0 )
                ;

        /* get  this systems error count */
        while ( (err_count = imc_rderrcnt_mr(0) ) < 0 )
                ;
```

Example 10–3: Error Detection Using the imc_rderrcnt_mr Function (cont.)

```
        if (err_count == mypage_rx->send_count) {
                /* no error, process the body */
                            .....
        }
        else {
                /* do error  processing */
                        ......
        }
}
```

As shown in Example 10–3, the `imc_rderrcnt_mr` function can be safely used to detect errors at the receiving end of a message. However, it cannot be guaranteed to detect errors at the transmitting end. This is because there is a small, but finite, possibility that the transmitting process will read the error count before the transmitting host has been notified of an error occurring on the receiving host. In Example 10–3, the program must rely on a higher-level protocol informing the transmitting host of the error.

The `imc_ckerrcnt_mr` function provides guaranteed error detection for a specified logical rail. This function takes a user-supplied local error count and a logical rail number as parameters, and returns an error in the following circumstances:

- An outstanding error is detected on the specified logical rail
- Error processing is in progress
- The error count is higher than the supplied parameter

If the function returns successfully, no errors have been detected between when the local error count was stored and the `imc_ckerrcnt_mr` function was called.

The `imc_ckerrcnt_mr` function reads the Memory Channel adapter hardware error status for the specified logical rail; this is a hardware operation that takes several microseconds. Therefore, the `imc_ckerrcnt_mr` function takes longer to execute than the `imc_rderrcnt_mr` function, which reads only a memory location.

Example 10–4 shows an amended version of the send sequence shown in Example 10–3. In Example 10–4, the transmitting process performs error detection.

Example 10–4: Error Detection Using the imc_ckerrcnt_mr Function

```
/* /usr/examples/cluster/mc_ex4.c */

/**********************************************/
/*  Transmitting Process With Error Detection */
/**********************************************/

#include <c_asm.h>
#define mb() asm("mb")

#include <sys/imc.h>
main()
{
        typedef struct {
                volatile        int     msg_arrived;
                int     send_count;
                int     remainder[2046];
        } global_page;
        global_page     *mypage_rx, *mypage_tx;
        imc_asid_t      glob_id;
        int             i, status;
        volatile        int     err_count;

        imc_api_init(NULL);

        imc_asalloc (1234, 8192, IMC_URW, 0, &glob_id,0);
        imc_asattach (glob_id, IMC_TRANSMIT, IMC_SHARED, IMC_LOOPBACK,
                     &(caddr_t)mypage_tx);
        imc_asattach (glob_id, IMC_RECEIVE, IMC_SHARED, 0,
                     &(caddr_t)mypage_rx);

        /*  save the error count  */
        while (  (err_count = imc_rderrcnt_mr(0) ) < 0 )
                ;

        do {
                mypage_tx->send_count = err_count;

                /* store message data */
                for (i = 0; i < 2046; i++)
                        mypage_tx->remainder[i] = i;

                /* now mark as valid */
                mb();

                mypage_tx->msg_arrived = 1;

        /*  if error occurs, retransmit */

        } while ( (status = imc_ckerrcnt_mr(&err_count,0)) != IMC_SUCCESS);
}
```

10.7 Clusterwide Locks

In a Memory Channel system, the processes communicate by reading and writing regions of the Memory Channel address space. The preceding sections contain sample programs that show arbitrary reading and writing of regions. In practice, however, a locking mechanism is sometimes needed to provide controlled access to regions and to other clusterwide resources. The Memory Channel API library provides a set of lock functions that enable applications to implement access control on resources.

The Memory Channel API library implements locks by using mapped pages of the global Memory Channel address space. For efficiency reasons, locks are allocated in sets rather than individually. The `imc_lkalloc` function allows you to allocate a lock set. For example, if you want to use 20 locks, it is more efficient to create one set with 20 locks than five sets with four locks each, and so on.

To facilitate the initial coordination of distributed applications, the `imc_lkalloc` function allows a process to atomically (that is, in a single operation) allocate the lock set and acquire the first lock in the set. This feature allows the process to determine whether or not it is the first process to allocate the lock set. If it is, the process is guaranteed access to the lock and can safely initialize the resource.

Instead of allocating the lock set and acquiring the first lock atomically, a process can call the `imc_lkalloc` function and then the `imc_lkacquire` function. In that case, however, there is a risk that another process might acquire the lock between the two function calls, and the first process will not be guaranteed access to the lock.

Example 10–5 shows a program in which the first process to lock a region of Memory Channel address space initializes the region, and the processes that subsequently access the region simply update the process count.

Example 10–5: Locking Memory Channel Regions

```
/* /usr/examples/cluster/mc_ex5.c */

#include <sys/types.h>
#include <sys/imc.h>

main ( )
{
    imc_asid_t   glob_id;
    imc_lkid_t   lock_id;
    int          locks = 4;
    int          status;

    typedef struct {
                    int    proc_count;
                    int    pattern[2047];
                   } clust_rec;

    clust_rec *global_record_tx, *global_record_rx; [1]
    caddr_t      add_rx_ptr = 0, add_tx_ptr = 0;
    int     j;

    status  = imc_api_init(NULL);

    imc_asalloc(123, 8192, IMC_URW, 0, &glob_id, 0);
```

Example 10–5: Locking Memory Channel Regions (cont.)

```
    imc_asattach(glob_id, IMC_TRANSMIT, IMC_SHARED,
                 IMC_LOOPBACK, &add_tx_ptr);

    imc_asattach(glob_id, IMC_RECEIVE, IMC_SHARED,
                 0, &add_rx_ptr);

    global_record_tx = (clust_rec*) add_tx_ptr; [2]
    global_record_rx = (clust_rec*) add_rx_ptr;


    status = imc_lkalloc(456, &locks, IMC_LKU,  IMC_CREATOR,
                         &lock_id); [3]
    if (status == IMC_SUCCESS)
    {

    /* This is the first process. Initialize the global region  */

        global_record_tx->proc_count = 0; [4]
        for (j = 0; j < 2047; j++)
                global_record_tx->pattern[j] = j;

        /* release the lock */
        imc_lkrelease(lock_id, 0); [5]

    }


   /* This is a secondary process */

    else if (status == IMC_EXISTS)
    {
       imc_lkalloc(456, &locks, IMC_LKU, 0, &lock_id); [6]

       imc_lkacquire(lock_id, 0, 0, IMC_LOCKWAIT); [7]

       /* wait for access to region */

       global_record_tx->proc_count = global_record_rx->proc_count+1; [8]

       /* release the lock */

       imc_lkrelease(lock_id, 0);

    }

    /* body of program goes here */
           .
           .
           .
    /* clean up */

    imc_lkdealloc(lock_id); [9]
    imc_asdetach(glob_id);
    imc_asdealloc(glob_id);
}
```

1 The process, in order to read the data that it writes to the Memory Channel global address space, maps the region for transmit and for receive. See Example 10–2 for a detailed description of this procedure.

2 The program overlays the transmit and receive pointers with the global record structure.

3 The process tries to create a lock set that contains four locks and a `key` value of `456`. The call to the `imc_lkalloc` function also specifies the `IMC_CREATOR` flag. Therefore, if the lock set is not already allocated, the function will automatically acquire lock zero (0). If the lock set already exists, the `imc_lkalloc` function fails to allocate the lock set and returns the value `IMC_EXISTS`.

4 The process that creates the lock set (and consequently holds lock zero (0)) initializes the global region.

5 When the process finishes initializing the region, it calls the `imc_lkrelease` function to release the lock.

6 Secondary processes that execute after the region has been initialized, having failed in the first call to the `imc_lkalloc` function, now call the function again without the `IMC_CREATOR` flag. Because the value of the `key` parameter is the same (`456`), this call allocates the same lock set.

7 The secondary process calls the `imc_lkacquire` function to acquire lock zero (0) from the lock set.

8 The secondary process updates the process count and writes it to the transmit region.

9 At the end of the program, the processes release all Memory Channel resources.

When a process acquires a lock, other processes executing on the cluster cannot acquire that lock.

Waiting for locks to become free entails busy spinning and has a significant effect on performance. Therefore, in the interest of overall system performance, make your applications acquire locks only as they are needed and release them promptly.

10.8 Cluster Signals

The Memory Channel API library provides the `imc_kill` function to allow processes to send signals to specified processes executing on a remote host in a cluster. This function is similar to the UNIX `kill`(2) function. When the `kill` function is used in a cluster, the signal is sent to all processes whose process group number is equal to the absolute value of the PID, even if that process is on another cluster member. The PID is guaranteed to be unique across the cluster.

The main differences between the `imc_kill` function and the `kill` function are that the `imc_kill` function does not allow the sending of signals across cluster members nor does it support the sending of signals to multiple processes.

10.9 Cluster Information

The following sections discuss how to use the Memory Channel API functions to access cluster information, and how to access status information from the command line.

10.9.1 Using Memory Channel API Functions to Access Memory Channel API Cluster Information

The Memory Channel API library provides the `imc_getclusterinfo` function, which allows processes to get information about the hosts in a Memory Channel API cluster. The function returns one or more of the following:

- A count of the number of hosts in the cluster, and the name of each host.
- The number of logical rails in the cluster.
- The active Memory Channel logical rails bitmask, with a bit set for each active logical rail.

The function does not return information about a host unless the Memory Channel API library is initialized on the host.

The Memory Channel API library provides the `imc_wait_cluster_event` function to block a calling thread until a specified cluster event occurs. The following Memory Channel API cluster events are valid:

- A host joins or leaves the cluster.
- The logical rail configuration of the cluster changes.

The `imc_wait_cluster_event` function examines the current representation of the Memory Channel API cluster configuration item that is being monitored, and returns the new Memory Channel API cluster configuration.

Example 10–6 shows how you can use the `imc_getclusterinfo` function with the `imc_wait_cluster_event` function to request the names of the members of the Memory Channel API cluster and the active Memory Channel logical rails bitmask, and then wait for an event change on either.

Example 10–6: Requesting Memory Channel API Cluster Information; Waiting for Memory Channel API Cluster Events

```
/* /usr/examples/cluster/mc_ex6.c */

#include <sys/imc.h>

main ( )
{

    imc_railinfo    mask;
    imc_hostinfo    hostinfo;

    int             status;
    imc_infoType    items[3];
    imc_eventType   events[3];


    items[0] = IMC_GET_ACTIVERAILS;
    items[1] = IMC_GET_HOSTS;
    items[2] = 0;

    events[0] = IMC_CC_EVENT_RAIL;
    events[1] = IMC_CC_EVENT_HOST;
    events[2] = 0;

    imc_api_init(NULL);

    status = imc_getclusterinfo(items,2,mask,sizeof(imc_railinfo),
                                &hostinfo,sizeof(imc_hostinfo));

    if (status != IMC_SUCCESS)
        imc_perror("imc_getclusterinfo:",status);

    status = imc_wait_cluster_event(events, 2, 0,
                                mask, sizeof(imc_railinfo),
                                &hostinfo, sizeof(imc_hostinfo));

    if ((status != IMC_HOST_CHANGE) && (status != IMC_RAIL_CHANGE))
        imc_perror("imc_wait_cluster_event didn't complete:",status);

}   /*main*/
```

10.9.2 Accessing Memory Channel Status Information from the Command Line

The Memory Channel API library provides the `imcs` command to report on Memory Channel status. The `imcs` command writes information to the standard output about currently active Memory Channel facilities. The output is displayed as a list of regions or lock sets, and includes the following information:

- The type of subsystem that created the region or lock set (possible values are IMC or PVM)
- An identifier for the Memory Channel region

- An application-specific key that refers to the Memory Channel region or lock set
- The size, in bytes, of the region
- The access mode of the region or lock set
- The username of the owner of the region or lock set
- The group of the owner of the region or lock set
- The Memory Channel logical rail that is used for the region
- A flag specifying the coherency of the region
- The number of locks that are available in the lock set
- The total number of regions that are allocated
- Memory Channel API overhead
- Memory Channel rail usage

10.10 Comparison of Shared Memory and Message Passing Models

There are two models that you can use to develop applications that are based on the Memory Channel API library:

- Shared memory
- Message passing

At first, the shared memory approach might seem more suited to the Memory Channel features. However, developers who use this model must deal with the latency, coherency, and error-detection problems that are described in this chapter. In some cases, it might be more appropriate to develop a simple message-passing library that hides these problems from applications. The data transfer functions in such a library can be implemented completely in user space. Therefore, they can operate as efficiently as implementations based on the shared memory model.

10.11 Frequently Asked Questions

This section contains answers to questions that are asked by programmers who use the Memory Channel API to develop programs for TruCluster systems.

10.11.1 IMC_NOMAPPER Return Code

Question: An attempt was made to do an attach to a coherent region using the `imc_asattach` function. The function returned the value `IMC_NOMAPPER`. What does this mean?

Answer: This return value indicates that the `imc_mapper` process is missing from a system in your Memory Channel API cluster.

The `imc_mapper` process is automatically started in the following cases:

- On system initialization, when the configuration variable `IMC_AUTO_INIT` has a value of 1. (See Section 10.1 for more information about the `IMC_AUTO_INIT` variable.)
- When you execute the `imc_init` command for the first time.

To solve this problem, reboot the system from which the `imc_mapper` process is missing.

This error may occur if you shut down a system to single-user mode from `init` level 3, and then return the system to multi-user mode without doing a complete reboot. If you want to reboot a system that runs TruCluster Server software, you must do a full reboot of that system.

10.11.2 Efficient Data Copy

Question: How can data be copied to a Memory Channel transmit region in order to obtain maximum Memory Channel bandwidth?

Answer: The Memory Channel API `imc_bcopy` function provides an efficient way of copying aligned or unaligned data to Memory Channel. The `imc_bcopy` function has been optimized to make maximum use of the buffering capability of a Compaq Alpha CPU.

You can also use the `imc_bcopy` function to copy data efficiently between two buffers in standard memory.

10.11.3 Memory Channel Bandwidth Availability

Question: Is maximum Memory Channel bandwidth available when using coherent Memory Channel regions?

Answer: No. Coherent regions use the loopback feature to ensure local coherency. Therefore, every write data cycle has a corresponding cycle to loop the data back; this halves the available bandwidth. See Section 10.6.1.3 for more information about the loopback feature.

10.11.4 Memory Channel API Cluster Configuration Change

Question: How can a program determine whether a Memory Channel API cluster configuration change has occurred?

Answer: You can use the new `imc_wait_cluster_event` function to monitor hosts that are joining or leaving the Memory Channel API cluster, or to monitor changes in the state of the active logical rails. You can write a program that calls the `imc_wait_cluster_event` function in a separate thread; this blocks the caller until a state change occurs.

10.11.5 Bus Error Message

Question: When a program tries to set a value in an attached transmit region, it crashes with the following message:

```
Bus error (core dumped)
```

Why does this happen?

Answer: The data type of the value may be smaller than 32 bits (in C, an `int` is a 32–bit data item, and a `short` is a 16–bit data item). The Compaq Alpha processor, like other RISC processors, reads and writes data in 64–bit units or 32–bit units. When you assign a value to a data item that is smaller than 32 bits, the compiler generates code that loads a 32–bit unit, changes the bytes that are to be modified, and then stores the entire 32–bit unit. If such a data item is in a Memory Channel region that is attached for transmit, the assignment causes a read operation to occur in the attached area. Because transmit areas are write-only, a bus error is reported.

You can prevent this problem by ensuring that all accesses are done on 32–bit data items. See Section 10.6.3 for more information.

Index