TruCluster Production Server Software

Application Programming Interfaces

Part Number: AA-QL8PC-TE

January 1998

Product Version:	TruCluster Production Server Software Version 1.5
Operating System and Version:	DIGITAL UNIX Version 4.0D

This manual describes the application programming interfaces (APIs) of the TruCluster Production Server Software product.

Digital Equipment Corporation
Maynard, Massachusetts

Contents

About This Manual

This manual describes the application programming interfaces (APIs) of the TruCluster™ Production Server Software product.

Audience

This manual is for programmers writing distributed applications that need the synchronization services of the distributed lock manager (DLM) or the cluster information services.

Organization

This manual contains two chapters and an index. A brief description follows:

Chapter 1	Describes how to use the features of the DLM.
Chapter 2	Describes how to use the cluster information services.

Related Documents

Consult the following TruCluster Software Products manuals for assistance in cluster configuration, installation, and administration tasks:

- TruCluster Software Products *Release Notes*—Documents known restrictions and other important information about the Production Server.
- TruCluster Software Products *Hardware Configuration*—Describes how to set up the processors that are to become cluster members, and how to configure cluster shared storage.
- TruCluster Software Products *Software Installation*—Describes how to install the Production Server on the systems that are to participate in the cluster.
- TruCluster Software Products *Administration*—Describes cluster-specific administration tasks, such as those required to set up an available server environment (ASE) within a cluster. It also shows how to configure, start, and manage distributed raw disk (DRD) services and other available services.

- TruCluster Production Server Software *MEMORY CHANNEL Application Programming Interfaces*—Describes the application programming interfaces that allow programming to the features of the MEMORY CHANNEL™ hardware.

Reader's Comments

DIGITAL welcomes any comments and suggestions you have on this and other DIGITAL 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`
- Mail:

  Digital Equipment Corporation
  UBPG Publications Manager
  ZKO3-3/Y32
  110 Spit Brook Road
  Nashua, NH 03062-9987

  A Reader's Comment form is located in the back of each printed manual. The form is postage paid if you mail it in the United States.

Please include the following information along with your comments:

- The full title of the book and the order number. (The order number is printed on the title page of this book and on its back cover.)
- The section numbers and page numbers of the information on which you are commenting.
- The version of DIGITAL UNIX that you are using.
- If known, the type of processor that is running the DIGITAL UNIX software.

The DIGITAL 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 DIGITAL technical support office. Information provided with the software media explains how to send problem reports to DIGITAL.

Conventions

The following typographical conventions are used in this manual:

`%` `$`	A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bourne, Korn, and POSIX shells.
`#`	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.
[|] { | }	In syntax definitions, brackets indicate items that are optional and braces indicate items that are required. Vertical bars separating items inside brackets or braces indicate that you choose one item from among those listed.
. . .	In syntax definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.
`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.
$\boxed{\text{Return}}$	In an example, a key name enclosed in a box indicates that you press that key.
Ctrl/*x*	This symbol indicates that you hold down the first named key while pressing the key or mouse button that follows the slash. In examples, this key combination is enclosed in a box (for example, $\boxed{\text{Ctrl/C}}$).

1 Distributed Lock Manager

This chapter describes how to use the distributed lock manager (DLM) to synchronize access to shared resources in a cluster. It contains the following discussions:

- How the DLM synchronizes the accesses of multiple processes to a specific resource (Section 1.1).
- The concepts of resources, resource granularity, namespaces, resource names, and lock groups (Section 1.2).
- The concepts of locks, lock modes, lock compatibility, lock management queues, lock conversions, and deadlock detection (Section 1.3).
- How to use the `dlm_unlock` function to dequeue lock requests and the `dlm_cancel` function to cancel lock conversion requests (Section 1.4).
- 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 1.5).
- How applications can perform local buffer caching (Section 1.6).

Section 1.7 provides a code example showing the basic DLM operations.

1.1 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 1–1 lists the functions the DLM provides. These functions are available in the `libdlm` library for use by applications.

Table 1–1: Distributed Lock Manager Functions

Function	Description
`dlm_cancel`	Cancels a lock conversion request
`dlm_cvt`	Synchronously converts an existing lock to a new mode
`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 and/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 and/or transaction IDs
`dlm_rd_attach`	Attaches a process or process lock group to a recovery domain

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

Function	Description
`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
`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 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.
- The protections and ownership (for instance, the user IDs and group IDs) employed within the namespace must be consistent throughout the cluster.
- 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 process IDs are not unique among cluster members, an application must not use a process ID to construct process-specific names for files or other resources managed by the DLM resources.
- Because the DLM delivers signals, completion notifications, and blocking notifications to the process, avoid using the DLM API functions in a threaded application.
- 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.

1.2 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 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 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. Once 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.

1.2.1 Resource Granularity

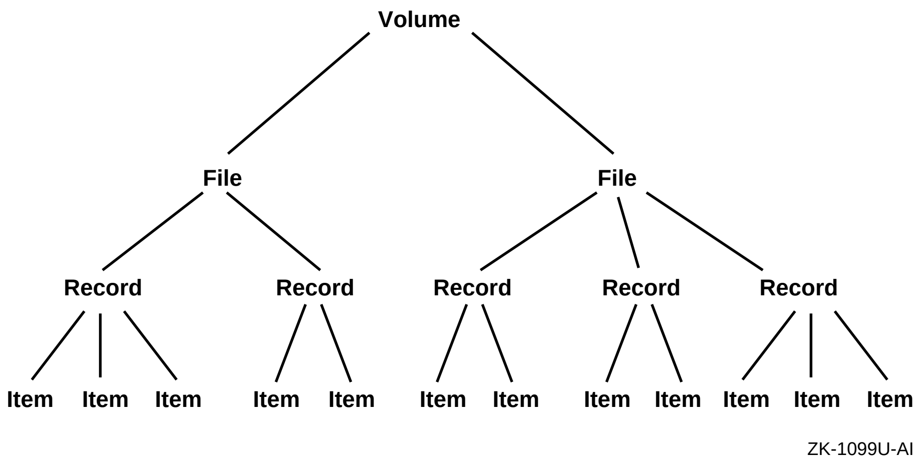
Many resources can be divided into smaller parts. As long as a part of a resource can be identified by a resource name, the part can be locked.

Figure 1–1 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 shown in Figure 1–1 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 1.5.5 for more information about parent locks and and sublocks.

Figure 1–1: Model Database



1.2.2 Namespaces

A **namespace** can be viewed as a container for resource names. Multiple namespaces exist to provide separation of unrelated applications for reasons of security and modularity.

A namespace is qualified by effective user ID or effective group ID.

Access to a namespace based on a user ID is limited to holders of that user ID. Access to a namespace based on a group ID is limited to members of that group.

Security is based on 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.

Cooperating processes must use the same namespace to coordinate locks for a given resource. A process must join a namespace before attempting to acquire a lock on a resource in that namespace. When the process calls the `dlm_nsjoin` function, 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. 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.

1.2.3 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 specified by the process (`resnam`)—The name 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 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
Resource 1	14	disk1	5
Resource 1	14	disk1	5

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

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

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

1.3 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` functions. The request specifies the following parameters:

- A namespace handle obtained from a prior call to the `dlm_nsjoin` function—The DLM checks a process's right to access a namespace before allowing it to obtain and manipulate locks on resources in that namespace. See Section 1.2.2 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 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 to 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 1.2.2 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 will then use 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 1.3.1 for more information about lock modes.

Null mode locks (see Section 1.3.1) are compatible with all other lock modes and are 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 and the mode of the new request is compatible with the existing lock. See Section 1.3.2 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 1.3.4 for additional information.

1.3.1 Lock Modes

The mode of a lock determines whether or not the resource can be shared with other lock requests. Table 1–2 describes the six lock modes.

Table 1–2: 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. 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 (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.

Table 1–2: Lock Modes (cont.)

Mode	Description
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.

1.3.2 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 1–3 shows the compatibility of the lock modes.

Table 1–3: 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

Table 1–3: Compatibility of Lock Modes (cont.)

Mode of Requested Lock	Resource Group Grant Mode					
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

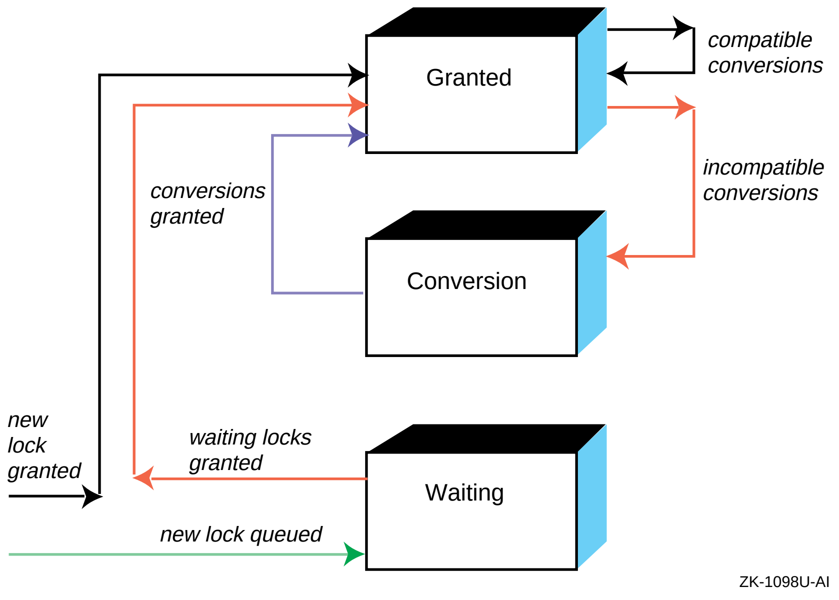
1.3.3 Lock Management Queues

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 1–2.

Figure 1–2: 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 conversion 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 conversion 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.)

1.3.4 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 wish 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 conversion 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 immediately following it on the conversion queue. The DLM continues to grant requests until the conversion queue is empty or it encounters an incompatible lock.

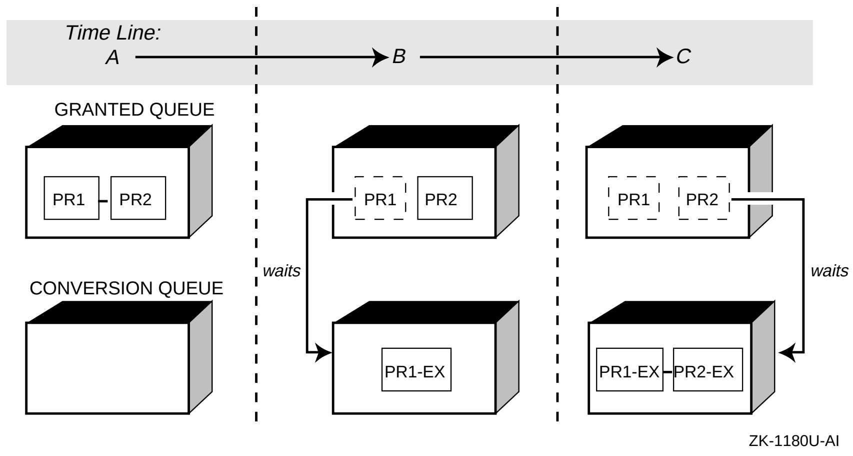
When the conversion queue is empty, the DLM checks the waiting queue. It grants the first lock request on the waiting queue if it is compatible with the locks currently granted. The DLM continues to grant requests until the waiting queue is empty or it encounters an incompatible lock.

1.3.5 Deadlock Detection

The DLM can detect two forms of deadlock:

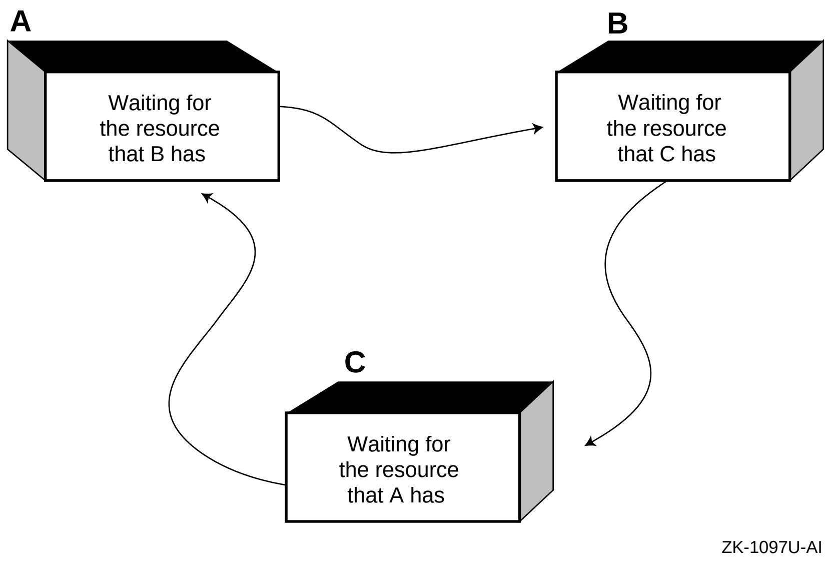
- Conversion deadlock—A conversion deadlock is one in which a conversion request has a granted mode that is incompatible with the requested mode of another conversion request ahead of it in the conversion queue. For example, in Figure 1–3, there are two granted PR mode locks on a resource (that is, the resource grant mode is PR). One PR mode lock tries to convert to EX mode and, as a result, must wait in the conversion 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 conversion queue. However, the first lock's request will never be granted, because its requested mode (EX) is incompatible with the second lock's granted mode (PR). The second lock's request will never be granted because it is waiting behind the first lock's request in the conversion queue.

Figure 1–3: 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 1–4, three processes have queued requests for resources that cannot be accessed until the current locks held are dequeued (or converted to a lower lock mode). Each process is waiting on another process to dequeue its lock request.

Figure 1–4: 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 conversion 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 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

You must 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 involving its own locks.

1.4 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 conversion. 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`* 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 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 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 shown in Table 1–4.

Table 1–4: 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 1.5.6 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.

1.4.1 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 conversion queue. Cancellation causes a lock in the conversion 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 specified in the conversion request to indicate that the request has been canceled. The returned status is `DLM_CANCELLED`.

1.5 Advanced Locking Techniques

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

1.5.1 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 an application does not want to wait for completion of the lock request, it should 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 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.

1.5.2 Notification of Synchronous Completion

The DLM provides a mechanism that allows processes to determine if a lock request is granted synchronously; that is, if the lock is not placed on the conversion 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 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.

1.5.3 Blocking Notifications

In some applications that use the DLM functions, a process must know if it is preventing another process from locking a resource. The DLM informs processes of this through the use of 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 executed.

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`

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 could still exist:

- The lock could still be blocked.
- The blocked lock could have been released by the application; therefore, no locks are actually blocked.
- The blocked lock could have been selected as a deadlock victim and the request failed to break a deadlock cycle.
- The blocked lock could have been 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 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.

Note also that if the signal 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.

1.5.4 Lock Conversions

Lock conversions perform the following functions:

- Promoting or demoting lock modes.
- Maintaining a low-level lock and converting 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, the procedure should not 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.
- Maintaining values stored in a resource lock value block—A lock value block is a small piece of memory 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 1.5.6 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.

1.5.4.1 Queuing Lock Conversions

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

1.5.4.2 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 there are no conversions already queued.

The `DLM_QUECVT` behavior is valid only for a subset of all possible conversions. Table 1–5 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 1–5: 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)	—	—	—	—	—	—
Concurrent Read (CR)	—	—	Legal	Legal	Legal	Legal
Concurrent Write (CW)	—	—	—	Legal	Legal	Legal
Protected Read (PR)	—	—	Legal	—	Legal	Legal
Protected Write (PW)	—	—	—	—	—	—
Exclusive (EX)	—	—	—	—	—	—

1.5.5 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.

1.5.6 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* parameter and the `DLM_VALB` flag 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` flag and a valid address in the *valb* 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* parameter and the `DLM_VALB` flag 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 1–6 shows how lock conversions affect the contents of the process's and the resource's lock value block.

Table 1–6: 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	Read
Protected Read (PR)	—	—	—	Read	Read	Read

Table 1–6: Effect of Lock Conversion on Lock Value Block (cont.)

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

Note that 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* parameter, and the DLM_VALB flag 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* parameter by returning the completion status of DLM_VALNOTVALID. 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.

1.6 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 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.

1.6.1 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 associated with each resource is used to contain a disk block version number. The first time 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 will find a version number mismatch and must read the latest copy from disk, rather than use its cached (now invalid) buffer.

1.6.2 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.

1.6.2.1 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 the 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.

1.6.2.2 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 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 the process tries to use the buffer.

1.6.3 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 check 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 the database is updated. However, if updates occur frequently, using version numbers is preferable to continually delivering blocking notifications.

1.7 Distributed Lock Manager Functions Code Example

The following programs show the basic mechanisms 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, listed in Example 1–1 and available from the `/usr/examples/cluster` directory, 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 1–1: 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
#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 */
```

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

```
        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 program 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
         */
        if(stat !=  DLM_SYNCH) {
                printf("%s: dlm_lock failed\n", prog);
                error(&lkid, stat);  5
        }
        /* 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)) {
                printf("%s: lock: value block not empty\n", prog);
                error(&lkid, stat);  6
        }
        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);
        }
```

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

```
        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;
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;
```

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

```
        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);
        /* NL mode better be granted SYNC status */
        if(stat !=  DLM_SYNCH) {
                printf("\t%s: dlm_lock failed\n", prog);
                error(&lkid, stat);  10
        }
        /* 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)) {
                printf("\t%s: value block not empty\n", prog);
                error(&lkid, stat);  11
        }

        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);
```

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

```
        if (strcmp(vb.valblk, "abc")) {
                printf("\t%s: main: value block mismatch <%s>\n",
                prog, vb.valblk);
                error(&lkid, stat);  17
        }
        /* 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. It is a namespace that allows access from processes 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 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 indicates that the DLM should 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 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 checks the status value returned from the `dlm_lock` function call. If the status value is not `DLM_SYNCH` status (the successful condition value 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 interested in this lock are currently running, this should not be the case.

6 The api_ex_master.c program checks that the contents of the value block the DLM has written to the location 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 program calls the dlm_nsjoin function to join the uid namespace: that is, the same namespace that the process 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 running the api_ex_master.c already holds an exclusive mode lock. The DLM_SYNCSTS flag indicates that the DLM should 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 requested by the dlm_lock function call is granted.

11 The api_ex_client.c program checks the contents of the value block the DLM has written to the location 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 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 taken out by the api_ex_master.c program specifies a blocking notification routine, the DLM uses the SIGIO signal to send the process 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 three 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 indicates that the DLM should 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 held by the process running the `api_ex_master.c` program is compatible with the exclusive mode lock 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 running the `api_ex_client.c` program. At this point, the `api_ex_client.c` program should 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 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 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`).

$\boxed{20}$ 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.

$\boxed{21}$ The `blkrtn` routine 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.

$\boxed{22}$ The `api_ex_client.c` program completes and exits.

$\boxed{23}$ As soon as the process 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 running the `api_ex_master.c` program. At this point, the `api_ex_master.c` program should see the `efg` text string that the `api_ex_client.c` program wrote previously to the resource's lock value block.

$\boxed{24}$ The `api_ex_master.c` program completes and exits.

2 Cluster Information Services

This chapter describes how to use the TruCluster Production Server information services to obtain information about cluster members and services. It includes the following:

- An overview of the TruCluster Production Server information services (Section 2.1).
- Examples that show how to use the TruCluster Production Server information services (Section 2.2).

2.1 Overview

The TruCluster Production Server information services exist primarily to provide an infrastructure for the cluster monitoring and administration utilities (such as `cnxshow` and `drd_ivp`) provided with the Production Server product. Other applications can use them as well.

Table 2–1 lists the functions provided by the TruCluster Production Server information services. These functions are available in the `libcnx` library for applications.

Table 2–1: Cluster Information Services

Function	Description
`clu_get_ase_drdsvcs_byname`	Obtains the distributed raw disk (DRD) services list from the specified cluster node
`clu_get_ase_enabled`	Determines whether or not availability software is enabled on a specified cluster node
`clu_get_ase_id_byname`	Obtains the `ASE_ID` of the specified cluster node
`clu_get_ase_nodes_byname`	Obtains the available server environment (ASE) member list from the specified cluster node
`clu_get_aseinfo_byname`	Obtains ASE-specific information from the specified cluster member system

Table 2–1: Cluster Information Services (cont.)

Function	Description
`clu_get_cluster`	Obtains a description of a cluster and its member nodes
`clu_get_cluster_net`	Obtains the name of the cluster interconnect interface on the local node
`clu_get_cnxdirector`	Obtains the name of the node that is running the connection manager director daemon (`cnxmgrd`)
`clu_get_nodebyname`	Obtains information about a named cluster node
`clu_get_nodebycsid`	Obtains information about the cluster node identified by the specified cluster ID (CSID)
`clu_get_qdisk`	Obtains information about each tie-breaker disk configured in the cluster

2.2 Using the TruCluster Production Server Information Services

The following examples show the use of the TruCluster Production Server information services.

The `cluinfo.c` program, listed in Example 2–1 and available from the `/usr/examples/cluster` directory, calls the `clu_get_cluster` and `clu_get_qdisk` functions to obtain information about the cluster configuration and the names and status of the member systems. It displays output similar to the following:

```
% cluinfo
-----------------------------------------------
               Cluster summary
-----------------------------------------------

The director is    : canarymc
  suspended        : no

Virtual hub is present
  disks defined : 1
  disks required: 1
  Disks names:
      /dev/rrz9a
```

```
The members are:

 cluster node name : canarymc
 host name         : canary.sun.ra.com
 node state is     : clu_mem_member
 incarnation       : ccaf0
 cluster system id : 0001,0001


 cluster node name : cheatmc
 host name         : cheat.sun.ra.com
 node state is     : clu_mem_member
 incarnation       : d1730
 cluster system id : 0001,0002
```

Example 2–1: Using the clu_get_cluster and clu_get_qdisk Functions

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/cluster_defs.h>

/* cluinfo.c
 *
 * A simple program to collect information about the cluster.
 *
 */

/* Makefile
 * cluinfo : cluinfo.c
 *       cc -o cluinfo cluinfo.c -lcnx
 *       chmod 555 cluinfo
 * clean   :
 *       -rm cluinfo
 */

/* Translate a Boolean value to yes or no */
#define xlate_bool(c)((c)? str_yes : str_no)
char            str_yes[] = "yes";
char            str_no[]  = "no";

/* Buffers to receive the requested data */
clu_node_t      nodebuf[MAX_CLUSTER_NODES];
clu_cluster_t   clubuf;
clu_qdisk_t     dbuf;

/* Format membership state into a readable string. */
char           *
xlate_mbr_state(clu_mem_state_t m)
{
#define NUM_MBR_STATES 5
    static char    *unkn = "unknown_state";
    static char    *mbr[NUM_MBR_STATES] = {
                                       "clu_mem_invalid",
                                       "clu_mem_unknown",
                                       "clu_mem_new",
                                       "clu_mem_member",
                                       "clu_mem_removed",
    };
    if (m > NUM_MBR_STATES - 1)
        return unkn;
    else
```

Example 2–1: Using the clu_get_cluster and clu_get_qdisk Functions (cont.)

```
        return mbr[m];
}

main(int arc, char **argv)
{
    int             clustatus, dskstatus;

    /* Collect information about the cluster and its nodes */
    clustatus = clu_get_cluster(&clubuf, nodebuf, MAX_CLUSTER_NODES);

    /* Collect information about tie-breaker disks */
    dskstatus = clu_get_qdisk(&dbuf);

    if (clustatus == CLU_SUCCESS)
    {
        int             i;
        clu_node_t     *p, *e;

        /* Display the cluster summary */
        (void) printf("------------------------------------------\n");
        (void) printf("               Cluster summary\n");
        (void) printf("------------------------------------------\n\n");
        (void) printf("The director is    : %s\n", clubuf.curr_direct_name);
        (void) printf("  suspended        : %s\n\n", xlate_bool(clubuf.suspended));

        /* Display disk information */
        if (dskstatus == CLU_SUCCESS)
        {
    /* Display disk information if there is a virtual hub present */

            if (dbuf.have_vhub)
            {
                (void) printf("Virtual hub is present\n");
                (void) printf("  disks defined : %d\n",
                              dbuf.num_qdisk_entries);
                (void) printf("  disks required: %d\n",
                              dbuf.num_qdisk_access_rqd);
                (void) printf("  Disks names:\n");
                for (i = 0; i < MAX_QDSKS; i++)
                {
                    if (dbuf.qdisk_vec[i].qdisk_name[0] != ’ ’)
                         (void) printf("      %s\n", dbuf.qdisk_vec[i].qdisk_name);
                }
            }
            else
            {
                (void) printf("Virtual hub is not present.\n");
                (void) printf("  Tie-breaker disks are not required.\n");
            }

            /* Display the membership list */
            (void) printf("\nThe members are:\n\n");
            for (p = nodebuf, e = &nodebuf[MAX_CLUSTER_NODES]; p < e; p++)
            {
                if (p->clu_node_name[0] != ’ ’)
                {
                    /* Information about a single node */
                    (void) printf(" cluster node name : %s\n",
                                  p->clu_node_name);
                    (void) printf(" host name         : %s\n",
                                  p->clu_hostname);
                    (void) printf(" node state is     : %s\n",
                                  xlate_mbr_state(p->clu_node_state));
                    (void) printf(" incarnation       : %lx\n",
                                  p->clu_node_incarnation);
                    (void) printf(" cluster system id : %04x,%04x\n",
                                  CLU_CSID_SEQN(p->clu_node_csid),
                                  CLU_CSID_IDX(p->clu_node_csid));
                    (void) printf("\n\n");
                }
            }
```

Example 2–1: Using the clu_get_cluster and clu_get_qdisk Functions (cont.)

```
        }
    }
    return 0;
}
```

The `cludo.c` program, listed in Example 2–2 and available from the `/usr/examples/cluster` directory, calls the `clu_get_cluster`, `clu_get_nodebyname`, `clu_get_ase_enabled`, `clu_get_ase_id_byname`, and `clu_get_ase_nodes_byname` functions to obtain information about individual cluster member systems. It displays output similar to the following:

```
% cludo
Contacting mcclu17 to collect the following
 cluster node name : mcclu17
 host name         : clu17.sun.ra.com
 ase enabled       : no
 ase information not available


Contacting mcclu5 to collect the following
 cluster node name : mcclu5
 host name         : clu5.sun.ra.com
 ase enabled       : yes
 ase id            : 5
 ase nodes         : mcclu5 mcclu8


Contacting mcclu8 to collect the following
 cluster node name : mcclu8
 host name         : clu8.sun.ra.com
 ase enabled       : yes
 ase id            : 5
 ase nodes         : mcclu5 mcclu8
```

Example 2–2: Obtaining Information from Cluster Member Systems

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/cluster_defs.h>

/*
 * cludo.c
 *
```

Example 2–2: Obtaining Information from Cluster Member Systems (cont.)

```
 * Determine membership and collect information from each node.
 *
 */

/*
 * cc -o cludo cludo.c -lcnx
 */

/* Translate a Boolean value to yes or no */
#define xlate_bool(c)((c)? str_yes : str_no)
char            str_yes[] = "yes";
char            str_no[] = "no";

/* Buffers to receive the requested data */
clu_node_t      nodebuf[MAX_CLUSTER_NODES];
clu_cluster_t   clubuf;

main(int arc, char **argv)
{
    int             clustatus;
    clu_node_t     *p, *e;

    /*
     * Collect information about the cluster and its nodes.  This list will
     * be used to communicate with each member node.
     */
    clustatus = clu_get_cluster(&clubuf, nodebuf, MAX_CLUSTER_NODES);
    if (clustatus != CLU_SUCCESS) {

        printf("Error obtaining cluster information\n");
        exit(1);
    }
    /*
     * Contact each member node and request more information.
     */
    p = nodebuf;                       /* mark the start */
    e = &nodebuf[MAX_CLUSTER_NODES];  /* mark the  end */
    for (; p < e; p++) {
        if (p->clu_node_name[0] != ’ ’) {
            clu_node_t      info;
            int             status;
            int             ase;

            (void) printf("Contacting %s to collect the following\n",
                             p->clu_node_name);
            status = clu_get_nodebyname(p->clu_node_name, &info);
            ase = clu_get_ase_enabled(p->clu_node_name);

            if (status != CLU_SUCCESS)

                (void) printf("error contacting that node\n");
            else {
                /* Information about a single node */

                (void) printf(" cluster node name : %s\n",
                              info.clu_node_name);

                (void) printf(" host name         : %s\n",
                              info.clu_hostname);
```

Example 2–2: Obtaining Information from Cluster Member Systems (cont.)

```
                (void) printf(" ase enabled       : %s\n", xlate_bool(ase));

                /* If ASE is enabled, collect additional information */
                if (ase) {
                    char            buf[CLU_MAX_GETINFO_BUF_SIZE];

                    /* Collect the domain id */
                    status = clu_get_ase_id_byname(
                          info.clu_hostname, buf, CLU_MAX_GETINFO_BUF_SIZE);
                    if (status == CLU_SUCCESS)
                        printf(" ase id            : %s", buf);

                    /* Collect the identity of nodes in the ase */
                    status = clu_get_ase_nodes_byname(
                          info.clu_hostname, buf, CLU_MAX_GETINFO_BUF_SIZE);
                    if (status == CLU_SUCCESS)
                        printf(" ase nodes         : %s", buf);
                } else

                    (void) printf(" ase information not available\n");
            }

            (void) printf("\n\n");
        }
    }
    return 0;
}
```

Index

How to Order Additional Documentation

Technical Support

If you need help deciding which documentation best meets your needs, call 800-DIGITAL (800-344-4825) before placing your electronic, telephone, or direct mail order.

Electronic Orders

To place an order at the Electronic Store, dial 800-234-1998 using a modem from anywhere in the USA, Canada, or Puerto Rico. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	—	Local Digital subsidiary or approved distributor
Internal (submit an Internal Software Order Form, EN-01740-07)	—	SSB Order Processing – NQO/V19 *or* U.S. Software Supply Business Digital Equipment Corporation 10 Cotton Road Nashua, NH 03063-1260

Reader’s Comments

TruCluster Production Server Software
Application Programming Interfaces
AA-QL8PC-TE

Digital welcomes your comments and suggestions on this manual. Your input will help us to write documentation that meets your needs. Please send your suggestions using one of the following methods:

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

If you are not using this form, please be sure you include the name of the document, the page number, and the product name and version.

Please rate this manual:

	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Usability (ability to access information quickly)	☐	☐	☐	☐

Please list errors you have found in this manual:

Page	Description

Additional comments or suggestions to improve this manual:

____

____

____

____

What version of the software described by this manual are you using? ________________

Name, title, department ____________________________
Mailing address ____________________________
Electronic mail ____________________________
Telephone ____________________________
Date ____________________________

Do Not Cut or Tear − Fold Here and Tape

digital™

NO POSTAGE
NECESSARY IF
MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST−CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
UEG PUBLICATIONS MANAGER
ZKO3−3/Y32
110 SPIT BROOK RD
NASHUA NH 03062−9987

Do Not Cut or Tear − Fold Here

Cut on
Dotted
Line