

Symmetric Cluster Architecture and Component Technical Specifications

David Teigland
Red Hat, Inc.

version 3.0
June 29, 2004

Revision History		
1.0	5/2002	"Framework for Cluster Software Components"
2.0	6/2002	"Cluster Systems - Concepts and Components"
3.0	9/2003	"Symmetric Cluster Architecture"

Contents

1	Symmetric Cluster Architecture	3
1.1	Introduction	3
1.2	Background	3
1.3	Architecture Design Principles and Goals	4
1.4	Component Definitions	5
1.5	GFS Requirements	6
1.6	Concepts	7
2	Functional Descriptions	9
2.1	CCS	9
2.2	CMAN	9
2.2.1	Connection Manager	9
2.2.2	Service Manager	10
2.3	GDLM	14
2.3.1	Lock Spaces	14
2.3.2	Lock Requests and Conversions	14
2.3.3	Callbacks	15
2.3.4	Node Failure	15
2.4	LOCK_DLM	15
2.5	Fence System	16

3	Technical Designs and Specifications	17
3.1	CCS	17
3.1.0.1	User Interface	17
3.1.0.2	Application Interface	17
3.1.0.3	Special CMAN Usage	17
3.1.0.4	Config Updates	17
3.2	CMAN	18
3.2.1	Cluster User Interface	18
3.2.2	Cluster Application Interface	18
3.2.2.1	Messages	19
3.2.2.2	Information	19
3.2.3	Node Information	21
3.2.4	Membership	21
3.2.4.1	Joining the cluster	21
3.2.4.2	Heartbeats	21
3.2.4.3	Leaving the cluster	22
3.2.5	Transitions	22
3.2.6	Quorum	22
3.2.6.1	Quorum device	23
3.2.6.2	Two-node clusters	23
3.2.7	Event Notification	24
3.2.7.1	Kernel callbacks	24
3.2.7.2	Userland signals	25
3.2.7.3	Service Manager control	25
3.2.8	Barriers	25
3.2.9	Communications	26
3.2.10	Services	27

3.2.11	Service API	27
3.2.11.1	Service formation	27
3.2.11.2	Service control	28
3.2.12	Service Events	29
3.3	GDLM	30
3.3.1	Motivation	30
3.3.2	Requirements	31
3.3.3	Lock Spaces	32
3.3.3.1	Usage	32
3.3.3.2	Functions: lockspace handling	32
3.3.4	Resources, Locks and Queues	33
3.3.4.1	Lock ID's	34
3.3.4.2	Functions: RSB/LKB structures	35
3.3.5	Cluster Management	36
3.3.5.1	Functions: node information	36
3.3.5.2	Control mechanics	37
3.3.5.3	Recovery states	39
3.3.5.4	Functions: recovery management	39
3.3.5.5	Cluster information	41
3.3.6	Lock Requests	41
3.3.6.1	Usage	41
3.3.6.2	Request processing	44
3.3.6.3	Unlock request processing	50
3.3.6.4	Canceling a request	51
3.3.6.5	AST handling	52
3.3.7	Resource Directory	52
3.3.7.1	Removing entries	53

3.3.7.2	Functions: resource directory	54
3.3.8	Hierarchical Locking	54
3.3.8.1	Usage	55
3.3.8.2	Resource tree structure	55
3.3.9	Lock Value Blocks	56
3.3.9.1	Usage	56
3.3.9.2	Value block copying	56
3.3.10	Lock Ranges	57
3.3.10.1	Usage	57
3.3.11	Deadlock	57
3.3.12	Recovery	58
3.3.12.1	Blocking requests	58
3.3.12.2	Update node list	59
3.3.12.3	Rebuild resource directory	59
3.3.12.4	Clear defunct requests	59
3.3.12.5	Mark requests to resend	60
3.3.12.6	Clear defunct locks	60
3.3.12.7	Update resource masters	61
3.3.12.8	Rebuild locks on new masters	61
3.3.12.9	Finish and cleanup	62
3.3.12.10	Event waiting	62
3.3.12.11	Functions: Request queue	63
3.3.12.12	Functions: Nodes list	63
3.3.12.13	Functions: Resource directory	63
3.3.12.14	Functions: Lock queue	64
3.3.12.15	Functions: Purging locks	64
3.3.12.16	Functions: Updating resource masters	64

	6
3.3.12.17 Functions: Recover list	65
3.3.12.18 Functions: Rebuilding locks	65
3.3.12.19 Functions: Event waiting	67
3.3.13 Communications	67
3.3.13.1 Sending requests	67
3.3.13.2 Receiving requests	68
3.3.13.3 Recovery requests	68
3.3.13.4 Functions: low-level communications	68
3.3.13.5 Functions: mid-level communications	69
3.3.13.6 Functions: recovery communications	70
3.3.14 Future Work	70
3.4 LOCK_DLM	70
3.5 Fence	70
3.6 Appendix A: Practical Usage	71
3.7 Appendix B: Service Group Examples	71

Chapter 1

Symmetric Cluster Architecture

1.1 Introduction

The Symmetric Cluster Architecture is a design for software components that together support the GFS and CLVM clustering products.

This design document begins with architectural requirements, motivations and definitions of components. It continues with functional descriptions of the components and ends with detailed technical design specifications.

1.2 Background

The project of defining and implementing this cluster architecture began in 2000/2001. It has been reviewed and revised at many levels in parallel with development. The earliest development (2000) was on the Connection Manager portion of the CMAN cluster manager. The latest development was on the GDLM lock manager that had a late start (5/02) because of an aborted plan to adopt an open source DLM from IBM. Some of the results of this architecture have made their way into earlier products (most notably CCS and the general cluster and node properties defined in configuration files.)

The years of research and education leading up to this architectural design produced dramatically new levels of understanding about this software and what it required. Early knowledge about the problems being solved was primitive but evolved through lengthy experimentation and study to the point where this comprehensive, top-down design could be undertaken.

Early GFS concepts revolved around storing GFS lock state in storage devices using special SCSI commands (DLOCK/DMEP). These ideas were accompanied by little to no understanding of what cluster management meant or what would be required to correctly support GFS. As we came to realize new extents of the problem, various aspects to clustering were forced into the DLOCK/DMEP locking system (and even protocol). We were slowly discovering that GFS would require far more advanced locking and clustering support than could realistically be provided by SCSI-based device locks.

Hardware devices that implemented the DLOCK/DMEP commands were scarce to non-existent. To develop and test GFS without the necessary hardware, we created small network servers to emulate a DLOCK/DMEP

device. Switching between DLOCK/DMEP hardware and emulators motivated the invention of the GFS Lock Harness API. One lock module could send requests to an actual DLOCK/DMEP device and another could send requests to a network server. The servers were antithetical to GFS's symmetric philosophy and chief selling points, of course, but they were merely for development purposes.

Eventually we realized that DMEP hardware devices would not become a reality. This was about the same time we needed to release GFS to customers. Dismayed, we were forced to require customers to use the DMEP emulator. We had long known that the alternative to device locking was a Distributed Lock Manager (DLM) but it had now become a pressing need. We had neither time nor expertise to develop one ourselves, so we began a search to adopt a DLM from elsewhere.

Meanwhile, with customers left using the DMEP emulator there were some simple things we could do to improve its performance. Mainly, this meant not strictly adhering to the DMEP protocol which among other things did not allow for callbacks. The result of these lock server improvements was called GULM and it eventually replaced the DMEP emulator.

The DLM search ended when IBM released some DLM code as open source. We jumped on this and began working with it in the hope that it could be linked with GFS in very short order. After several months, however, we realized the IBM DLM had serious problems and would need extensive engineering work from us to be useful. We decided instead to develop our own DLM from scratch.

The larger architecture design and CMAN developments had continued throughout the process of finding a DLM. One of the first architectural principles had been that the cluster manager must be a unique, symmetric, stand-alone entity that would independently support any lock manager, GFS and CLVM. It was nearly complete, allowing us to focus on DLM development.

Faced with the development time of the DLM and customer intolerance for the DLOCK/DMEP/GULM lineage of server-based locking, it was decided to add basic fail-over to GULM to keep customers interested while the distributed clustering and locking subsystems were finished.

DLM development has been completed providing the full set of symmetric clustering and locking subsystems to support GFS and CLVM.

1.3 Architecture Design Principles and Goals

There are a few general ideas that guide the development of the architectural components.

1. *Symmetry*

Symmetry has long been a chief design principle of our clustering products. Each component of this architecture has been designed from the start with this symmetry in mind. As such, the components when combined produce a uniformly symmetric system as a whole, exploiting to the fullest extent the advantages inherent in such architectures.

2. *Modularity*

A second key advantage distinguishing GFS in the past has been its modularity. This allows GFS to adapt quickly to new environments and makes it exceedingly extensible. Modular components can also be shared among products avoiding duplication and allowing products to work together (e.g. CLVM and GFS using common subsystems). This architecture furthers the modular tradition, understanding and defining new components from its inception.

3. *Simplicity*

The clustering products should remain simple, conceptually and from a management perspective. An overly complex architecture makes the products difficult to understand and use and prohibits them from being easily extended or adapted in the future. The symmetry principle supports this to a large extent as symmetric clusters are far simpler conceptually (although often more difficult to design technically.)

4. *Correctness*

Customers place high trust in the software that manages their data. They must believe that the software will not corrupt the data or simply lose it. Even one case of a customer losing data can have devastating effect on our reputation. Every scenario where safety is at risk has been thoroughly investigated and addressed in this design.

1.4 Component Definitions

The Symmetric Cluster Architecture defines the following distinct software components. Each component has its own functional requirements and design. The components are defined here with respect to their dependence on the others.

CCS	Cluster Configuration System
CMAN	Cluster Manager
FENCE	I/O Fencing System
GDLM	Global Distributed Lock Manager
LOCK_DLM	GFS Lock Module for the DLM
GFS	Global File System
CLVM	Cluster Logical Volume Manager

The modular components are clearly separate in function so they can form the basis of multiple products.

CCS

- Depends on no other component (for basic function).
- Provides static configuration parameters to other components.

CMAN

- Depends on CCS for cluster and node identifiers.
- Provides cluster and service management to Fence, GDLM, GFS/LOCK_DLM, and CLVM.

FENCE

- Depends on CCS for node fencing parameters and fence device parameters. Depends on CMAN for cluster management.
- Provides I/O Fencing for GFS/LOCK_DLM and CLVM.

GDLM

- Depends on CMAN for cluster management.
- Provides lock management for GFS/LOCK_DLM and CLVM.

GFS/LOCK_DLM

- Depends on CMAN for cluster management. Depends on Fence for I/O Fencing. Depends on GDLM for lock management.
- Provides nothing to other components.

CLVM

- Depends on CMAN for cluster management. Depends on GDLM for lock management.
- Provides nothing to other components.

1.5 GFS Requirements

This is an overview of GFS requirements. The specific requirements are covered in more depth in the later descriptions of how they are met.

GFS relies on external components to provide the following functions. They are provided to GFS through a "lock module" plugged into GFS's Lock Harness interface. In this architecture, the lock module is LOCK_DLM. (The lock management requirements dominate the others – the term "lock module" and the LOCK prefix reflect this.) LOCK_DLM primarily acts as a bridge to the independent CMAN and GDLM clustering/locking managers, but GFS-specific semantics are implemented by the GFS lock module, not the general lock manager.

1. *Lock Management*

Each GFS file system has a unique name specified when it's created. This name is given to the lock module which must use it to associate different GFS file systems with unique lock spaces.

GFS's locking requirements are standard and quite simple. A GFS node will only request a single lock on a resource at once. Resources are uniquely identified by GFS using 12 bytes (binary data composed of two integers). GFS uses three lock modes: "exclusive" and two shared modes ("shared" and "deferred") that are incompatible with each other and exclusive.

GFS lock and unlock requests are asynchronous. GFS provides a callback function for notification of completed requests and for blocking callbacks. A blocking callback should be accompanied by the mode of lock being requested.

Normal GFS requests can be modified by three flags. The first is a "try" flag that indicates the request should not block if the lock cannot be immediately granted. Blocking callbacks should not be sent if the "try" request fails. The other two flags are optional and are used to enhance performance. The "any" flag indicates that either of the shared modes may be granted – preferably the one compatible with existing granted modes. The "one_cb" flag used with "try" indicates that a single blocking callback should be sent if the request cannot be granted immediately.

GFS requires selected resources have a 32 byte value block available for reading and writing in conjunction with lock operations. The value should be written with the release of an exclusive lock and read when any lock is acquired. The value block should be cleared to zero if the most recent value is lost during recovery.

GFS requires the lock module to avert conversion deadlock when locks are promoted. The lock manager is permitted, however, to release a lock in the process of promoting it if the result is subsequently flagged to indicate that the lock was released.

GFS must be able to cancel an outstanding request. Once canceled a request should complete immediately. If the request was not successful the result must be flagged as having been canceled.

The lock manager with the support of the cluster manager must only allow locks to be held by nodes that have the file system mounted. A node that fails with the file system mounted must have its locks released before normal operation is resumed. The released locks of a failed node must not be granted until GFS has completed its recovery for the failed node (with the following exception).

A GFS node performing recovery must be able to acquire locks that are otherwise not granted during recovery. GFS identifies these special requests with the "noexp" flag. The lock module must distinguish these special requests from ordinary requests and allow them to proceed as usual.

2. *Name-space Management*

GFS requires a unique node identifier to use for journal selection. The integer identifier provided at mount time must be the lowest unused ID beginning from zero. The scope is all nodes with the file system mounted. This allows the limited number of GFS journals to all be used.

3. *Recovery*

GFS requires a cluster manager to know which nodes have each file system mounted at any point in time. One or more nodes with file system F mounted must receive a "recovery needed" callback if any other node with F mounted fails. The callback must identify the journal ID used by the failed node (see previous section). Upon receiving this callback, one of the live GFS nodes will perform GFS recovery using the journal of the failed node.

GFS expects to receive recovery callbacks for a journal ID until it indicates it's done (through the lock module interface "lm_recovery_done" function.) That is, if a node recovering a journal fails, another node must be sent the recovery callback for the partially-recovered journal in addition to a callback for the journal of the second failed node. This holds true all the way to a single remaining node.

4. *I/O Fencing*

The clustering and fencing systems must guarantee that a failed node has been fenced successfully from the shared storage it was using (or has been fully reset) before GFS recovery is initiated for the failed node.

5. *Exclusive Initialization*

When a GFS file system is mounted on a node, it needs to be told if the node is the first in the cluster to mount the particular file system. In combination with being first, GFS on the first mounting node must have exclusive access to the entire file system until it indicates through the lock module interface that it has completed initialization and others may mount.

When a node is given this exclusive initialization right for a file system, the clustering and fencing systems must guarantee that every other node in the cluster (nodes that might have had the particular file system mounted in the past) is in not in a state having hung with GFS mounted since last being reset or fenced. i.e. If the state (including fencing history) of a node X is unknown at the time node Y requests the exclusive initialization right to any file system that X might have mounted in the past, X must be fenced before Y is given that right.

1.6 Concepts

This section lays out the concepts being dealt with and the simplicity that should be conveyed to the user. This is simplicity in the natural definition of objects, their properties and the relationships among them. These are the concepts a user will want to comprehend to effectively set up and manage the software.

Entities

Cluster	A group of nodes.
Node	A machine that can join and leave a cluster.
Resource	Something used by nodes in a cluster.
Service	A program run by nodes in a cluster. Often related to accessing a resource.

Policies

- A node can be a member of one cluster.
- A resource belongs to one cluster.
- A node in a cluster can access the resources in the cluster.
- All nodes are equal.
- Access to shared resources is controlled by services running on the nodes.
- Arbitrary cluster members can start and stop using arbitrary cluster resources at any time without affecting any other cluster members.
- The failure of a cluster member should only affect other cluster members that are using the same shared resources as the failed node.

Properties

Cluster

- name: unique name
- members: list of nodes that are members

Node

- name: unique name
- cluster: the cluster the node is member of

Resource

- name: unique name
- cluster: the name of the cluster whose members can use the resource

Resources exist in many forms and at many levels. A resource could be a device, a file, a record, a piece of metadata, or any other entity that can be consistently named among nodes or processes. Services accessing shared resources often require synchronization involving some form of locking.

The generic properties of a service are not generally relevant when managing the system. The taxonomy above is extended further in the area of services when describing the internal function of the cluster manager.

Chapter 2

Functional Descriptions

Description of purpose, main parts and functions of each component. Specific interface definitions and implementation details are left for the technical design specifications.

2.1 CCS

CCS provides access to a single cluster configuration file. The CCS daemon (`ccsd`) running on each node manages this file and provides access to it. When `ccsd` is started it finds the most recent version of the file among the cluster nodes. It keeps the file in sync among nodes if updates are made to it while the cluster is running.

2.2 CMAN

The CMAN cluster manager has two parts: Connection Manager and Service Manager. The main requirements of CMAN are to manage the membership of the cluster (connection manager) and to manage which members are using which services (service manager). CMAN operates in the kernel and both parts are completely symmetric – every node is the same.

CMAN has no specific knowledge of higher level cluster services; it is an independent, stand-alone system using only CCS. As a general cluster manager it should naturally not function specifically for select higher level applications. While GFS, GDLM, CLVM, etc. can sit above it, CMAN can be used for entirely different services or purposes.

2.2.1 Connection Manager

Connection Manager (`cnxman`) tracks the liveness of all nodes through periodic "HELLO" messages sent over the network. `Cnxman`'s list of current live nodes defines the membership of the cluster. Managing changes to cluster membership is the main task of `cnxman`.

A node joins the cluster by broadcasting the cluster's unique ID on the network. Other members of the

cluster will accept the new node and add it to the membership list through a multi-step transition process whereby all members agree on the new addition. Transitions are coordinated by a dynamically selected node. The command `'cman_tool join'` will initiate this procedure on a node and cause the node to join the cluster.

A current member can leave the cluster by running the command `'cman_tool leave'`. Again, the leave process involves a multi-step transition process through which all remaining members agree on the membership change. A node cannot leave the cluster (except by force) until all higher level clustering systems dependent upon CMAN have shut down.

The failure of a cluster node is detected when its HELLO messages are not received by other members for a predetermined number of seconds. When this happens, the remaining cluster members step through a transition process to remove the failed node from the membership list. A failed node can rejoin the cluster once it has been reset.

The cnxman cluster can suffer from a "split-brain" condition in the event of a network partition. Two groups of nodes can both form a cluster of the same name. If both clusters were to access the same shared data, it would be corrupted. Therefore, cnxman must guarantee, using a quorum majority voting scheme, that only one of the two split clusters becomes active. This means that to safely cope with split-brain scenarios, cnxman must only enable cluster services when over half (a majority) of nodes are members. This behavior can be modified by assigning an unequal number of votes to different machines.

To join the CMAN cluster, the `cman_tool` program gets the unique cluster name from the cluster configuration file (through CCS.) This means that the definitive step of associating a node with a particular cluster comes when the CCS daemon selects a cluster configuration file to use. Only nodes named in the cluster configuration can join the cluster.

When a node joins the cluster, cnxman assigns it a unique, non-zero node ID (integers beginning with 1). Once assigned, a node will retain its node ID for the life of the cluster – if the node leaves and rejoins the same cluster it will have the same ID (cnxman does not reuse node ID's for different nodes). Exported cnxman functions can be used to get a listing of all nodes, current members, or to look up node information (name, IP address, etc).

2.2.2 Service Manager

Service Manager (SM) manages which nodes are using which services. It uses "Service Groups" to represent different instances of services operating on dynamic groups of nodes in the cluster.

In the case of a symmetric service where all nodes can produce results independently, it is critical that all nodes running the service agree on the service group membership while any are producing results. The service group membership is often a factor determining the output of the cluster service – any disagreement can result in inconsistent output. Because of this, the operation of the service must be temporarily suspended when the group of nodes running the service changes. When the new group comes to agreement on the new group membership, the operation of the service can be resumed ¹.

It will be useful to add concepts associated with services to the definitions presented earlier. The new entities are defined, along with their properties, followed by some general rules governing them.

¹In a client-server architecture, the essential management and recovery functions, into which the group membership factor, occur on the server only. Clients do not make independent decisions based on factors like service group membership, even if they keep their own record of the nodes copied from the server. A transition from one membership group to another (including necessary adjustments within effected services) occurs on the server without stopping the service on all clients. Replies to client requests may at most be delayed momentarily as the server makes the adjustment.

Entities

Cluster	A group of nodes.
Node	A machine that can join and leave a cluster.
Resource	Something used by nodes in a cluster.
Service	A program run by nodes in a cluster. Often related to accessing a resource.
Service Group	The group of nodes running an instance of a service.

The new addition is a service group which is a conceptual entity created and managed by SM. Nodes become part of service groups through the services they use. GFS, GDLM and Fence are all symmetric services that form service groups. A node mounting a GFS file system will become a member of a service group for each of the three services.

Properties

Cluster

- name: unique name
- members: list of nodes that are members

Node

- name: unique name
- cluster: the cluster the node is member of

Resource

- name: unique name
- cluster: the name of the cluster whose members can use the resource

Service

- type: the type of service (GFS, GDLM, Fence Domain)
- level: the order of recovery among layered services

Service Group

- service: the type of service
- name: a unique name identifying the instance of the service in the cluster
- members: the group of nodes running the service instance together

Policies

- A service performs no function until it is instantiated.
- Any number of instances of a service can exist; each is distinct and identified by a unique name in the cluster.
- A service group is the result of a service instantiation.
- A node that is a cluster member can instantiate services and join/leave the associated service groups.
- A node can be in any number of service groups at the same time.

- An instance of a service is identified by a unique name in the cluster. Any cluster member instantiating and joining a service of the same type and name will be in the same service group.
- To use a resource, a node may be part of multiple service groups, each controlling a different aspect of sharing the resource.
- The unique name of a service instance is often based on the name of the shared resource to which the service is supporting access.

One part of the Service Manager function should already be clear. SM is in charge of managing service groups in the cluster. This involves instantiating them, managing nodes joining and leaving them, and managing the recovery of the appropriate service groups when cluster nodes fail.

If a program needs to be cluster-aware and know which nodes are running it, it can become a SM service with its instantiations managed as service groups. This can be done using the SM API detailed later. What it means for a program to be under the management of SM is the topic of the next section.

Service Control

When we say that a service requires "cluster management" we are referring to the fact that the service needs to respond to certain cluster events as they occur. When a service registers to be managed/controlled by SM, it provides a set of callback functions through which the SM can manage it given cluster events affecting it. We refer to the use of these callback management functions as service "control". There are three control methods that apply to services:

1. **stop** - suspend - membership of the service group will be changed.
2. **start** - recover - begin any recovery necessary given the new service group members.
3. **finish** - resume - all service group members have completed start.

When a service receives a "stop" it is aware that other services will be reconfiguring themselves (recovering) and may produce results inconsistent with the previous configuration of the cluster. The service is also aware that it will be reconfigured itself and must not produce any results assuming a particular service group membership.

When a service receives a "start" it knows that other services at a lower level have completed their recovery and are in a reliable state. It also knows that all nodes in the service group have received a stop, i.e. when it starts it knows that no other group members will be operating based on the previous membership. The new list of service group members is provided with the start. This service can now do its own recovery. When done starting, the service notifies SM through an API callback function.

When a service receives a "finish" it knows that all other members of the service group have completed their start. Any recovery state can be cleared. (Recovery state may need to be maintained until finish is received due to the fact that a service group member may fail while starting.) The service can safely resume normal operation, aware that all service group members have recovered in agreement.

The three functions are ordinarily received by a service in order, but a stop can be received while starting if a node in the group fails during start. Specific guarantees about the sequence and ordering of these control functions as seen by a service will be given later.

Service Events

A service event is an occasion where a service group transitions from one set of members to a different set of members. A complete event entails stop, start and finish. A service event is caused by one of the following:

- A node joins the service group.
- A node leaves the service group.
- A node in the service group fails.

At present, SM allows one node to join or leave a service group at a time. Not until a currently joining or leaving node has finished will another node be permitted to begin joining or leaving. Enhancements can be added to overcome the limitations of this serialization.

A Connection Manager transition occurs when a cluster member fails or a node joins or leaves the cluster. At the end of one of these transitions, the Service Manager is notified of the change. The only change that interests SM is the failure of a cluster member. When this happens, SM looks through all service groups to find if the failed node is a member of any. Any service group the failed node is part of is subject to a service event to remove the node and recover the service on the remaining service group members.

As was briefly mentioned in the Connection Manager's quorum description, SM will not start cluster services if the Connection Manager determines that quorum is lost. While quorum is lost, stopped service groups remain stopped but some service groups remain running (not stopped) because they were unaffected by the specific node failure (the failed node wasn't in the service group). When quorum is regained by a node joining the cnxman cluster (or intervention to reduce the number of expected votes), SM service events will continue with any stopped services being started.

These methods prevent services from being *started* in two split-brain clusters. However, an unaffected service may have continued running in the original inquorate cluster while the same service is to be started in the new, split, quorate cluster. Corruption of shared data is averted because the first service to be started in the new cluster is the fencing service. When the fencing service is started it fences all the nodes partitioned in the original inquorate cluster. Their access to shared storage will therefore be cut off before the same services are started in the new cluster.

Layered Services

A fundamental issue in the cluster architecture design has emerged; it is that of "layered services". Services often have a dependence upon each other. This creates layers of services where one layer relies on the layer below it to operate. Coordinating all the levels is the job of the cluster manager. The cluster manager relies functionally on no other cluster components and controls all the other clustering services based on its first-hand knowledge of node state, relative service levels and service group membership.

A service has an intrinsic level based on its function. Lower level services provide functions for higher level services. During recovery the service manager starts services from lowest level to highest. A high level service could not start if the service it depends on has not yet started. (Multiple services can exist at the same level and are started at once.)

GFS/LOCK_DLM relies upon the Fencing service and the GDLM service to be completely recovered and fully operational before it begins its own recovery. Therefore, both Fence and GDLM are lower service levels.

Service Types

The following table shows what constitutes a service group for each service type.

- A "Fence Domain" (FD) is a service group created by the Fence service.
- A "Lock Space" (LS) is a service group created by the GDLM service.

- A "Mount Group" (MG) is a service group created by the LOCK_DLM service.
- A FD's function is to I/O fence service group members that fail.
- A LS's function is to manage locks held by service group members.
- A MG's function is to send GFS recovery callbacks for service group members that fail.

2.3 GDLM

The GDLM lock manager is a VMS-like distributed lock manager. It follows the traditional VaxCluster DLM model that many other commercial DLM's have copied. The locking and recovery methods are not particular to GFS or CLVM or any other application. GDLM was designed to depend only on the cluster manager (CMAN) and can be used independently of any other systems.

In both the kernel and user space, GDLM exports a standard locking API that is as comprehensive as any application would generally require. GDLM distinct features are the algorithms used internally to distribute the lock management across all nodes in the cluster removing bottlenecks while remaining fully recoverable given the failure of any node or number of nodes.

2.3.1 Lock Spaces

Applications can create and use their own GDLM lock spaces (minor applications can use a common "default" lock space). GDLM will return a lock space context pointer to any application that requests a lock space with an arbitrary name. There is no limit to the number of lock spaces that can be created and used on a single node or throughout the cluster.

Application instances on the same or different nodes in the cluster requesting lock spaces with the same name will be a part of the same lock space. Each lock space can be created with specific features enabled or disabled. GFS, for example, creates lock spaces for each of its file systems with the NOTIMERS flag set indicating that GDLM should not return lock requests that have been waiting, ungranted, for too long.

2.3.2 Lock Requests and Conversions

Locks of a specified mode are requested against a named resource. GDLM's maximum resource name length is 64 bytes (GFS uses 12 or 24 byte resource names, depending on encoding style). The resource name is not interpreted by GDLM, but treated as 64 binary bytes of data.

There is no limit to the number of locks that can be held at once by the same or different nodes against a particular resource. The first node requesting a lock against a resource becomes the resource's master.

Locks with the following modes can be requested against resources: Null (NL), Concurrent Read (CR), Concurrent Write (CW), Protected Read (PR), Protected Write (PW), and Exclusive (EX). The standard compatibility matrix for these modes is included in the technical specifications. GFS's lock modes map to NL, EX, CW and PR.

Once a lock is obtained against a resource, it can either be converted to a different mode or unlocked. (The common conversion semantics are different from the narrower semantics used by GFS, although a translation

can effectively be made between them.) To convert an existing lock, GDLM returns with the first request a random "lock ID" that is used to reference the lock in subsequent requests.

A lock request can optionally specify a range. By default when no range is specified, the full range is assumed. Multiple nodes can hold exclusive locks with non-overlapping ranges against a resource at once. A lock request can also specify a Lock Value Block (LVB): 32 bytes of arbitrary data associated with a resource that can be saved and retrieved with a lock request. Finally, parent locks can be specified in lock requests to form hierarchical resource trees. Each level of the tree has a separate name space.

Time-out based deadlock detection can be enabled or disabled per lock space with the time-out period configurable in the configuration file. A per request flag can also be set that causes GDLM to detect conversion deadlock and resolve such events by temporarily demoting the granted mode of one or more locks (this is a GFS-oriented feature.)

The resource directory provides a simple mapping of resource names to resource masters. It is distributed across all nodes in a lock space using a shared hash function on resource names. All nodes are presently given a directory weight of one resulting in equal distribution. (Configurable weights is a future enhancement.)

2.3.3 Callbacks

In DLM terminology, a callback is an AST (Asynchronous System Trap). An AST either reports that an asynchronous lock request is complete or that an existing lock is blocking another lock that has been requested (a Blocking AST). The AST function called by GDLM is provided by the application with each lock request. An AST argument is also provided that will be passed back to the application to identify the specific lock.

2.3.4 Node Failure

There are very few if any interesting external issues involved in node failure or recovery. The lock manager will not (and should not) impose any special policy or requirements related to recovery on the applications.

When a node in a GDLM lock space fails (or leaves the lock space voluntarily) the CMAN Service Manager restarts the lock space with the new set of nodes in the lock space service group. This is all invisible externally. When GDLM adjusts to the new set of nodes (recovery), any locks held by previous nodes will simply disappear and previously blocked requests will be granted.

2.4 LOCK_DLM

LOCK_DLM represents GFS in the symmetric clustering and locking world. It interacts with the external GDLM lock manager, CMAN cluster manager and Fence system on behalf of GFS. It is primarily a bridge interfacing with the API's of these other systems. There are some minor GFS-specific functions it implements itself so that external managers remain general-purpose and not tied to a specific product.

To support GFS, LOCK_DLM plugs into GFS's Lock Harness, providing the full set of functions required by GFS. The LOCK_DLM implementation of all these functions is detailed in the technical specifications.

At mount time for a specific GFS file system, LOCK_DLM requires that the node already be a cluster member, otherwise the mount fails. If the node is a cluster member LOCK_DLM adds a reference count to

the CMAN module preventing the node from leaving the cluster while GFS file systems dependent on the cluster are mounted. (Other subsystems relying on CMAN also add reference counts when started for the same purpose.)

A second important check at mount time handled by LOCK_DLM is that the cluster name encoded in the specific file system's superblock (by `gfs_mkfs`) matches the name of the cluster the node is in. This check is necessary to prevent nodes from different clusters from mounting the same file system at once and quickly corrupting it.

With respect to locking, LOCK_DLM creates a new GDLM lock space for each file system at mount time. The unique file system name (also available from the file system's superblock) is used as the lock space name. With respect to fencing, LOCK_DLM checks that the node is a member of a fence domain at mount time. If the node fails while the file system is mounted, the node will be fenced. LOCK_DLM also creates its own CMAN/SM Service Group at mount time which it calls a "mount group". The members of the mount group are the nodes with that specific file system currently mounted. Nodes in a LOCK_DLM mount group are responsible for recovering the GFS journals of other mount group members if they fail. In this regard LOCK_DLM must also keep track of which journals the different mount group members are using.

2.5 Fence System

The fence daemon (`fenced`) must be started on each node after joining the cluster and before using GFS. `Fenced` determines which nodes to I/O fence based on the "fence domain" (FD) membership. The default fence domain is a SM service group and the members are all nodes in the cluster running `fenced`.

When a node needs to be I/O fenced, `fenced` looks up the node-specific fencing parameters from CCS. It then calls the specified fence agent to carry out the actual fencing.

Once a node joins an FD it will be fenced by another FD member if it fails before leaving. In practice, nodes in the cluster join one common ("default") FD before they use CLVM or a GFS file system. A node leaves the default FD once it is done using CLVM/GFS – it can then be shut down without being fenced.

Any cluster member can join a fence domain and be subject to fencing, although it must be able to carry out fencing operations on any other fence domain members.

Chapter 3

Technical Designs and Specifications

3.1 CCS

The cluster configuration file is kept in the `/etc/cluster/` directory on each cluster node. Managing and providing access to this file is the job of the CCS daemon, `ccsd`.

3.1.0.1 User Interface

3.1.0.2 Application Interface

3.1.0.3 Special CMAN Usage

3.1.0.4 Config Updates

3.2 CMAN

CMAN is a general purpose, kernel based cluster manager designed to support other kernel and userland systems dependant on cluster information and event notification. CMAN is also symmetric and quorum-based; all cluster members are peers.

CMAN has two main parts, the Connection Manager (Cnxman) and the Service Manager (SM). Cnxman manages which nodes are in the cluster, handling things such as membership, transitions (joining and leaving the cluster), and quorum. SM is quite separate and manages service groups instantiated by cluster services in the kernel and userland. Services are dynamically registered with SM and are thereafter controlled with standard callback functions. SM defines "Service Groups" where each service group represents a collection of nodes using a particular instance of a service.

The Cnxman portion of CMAN is activated and begins managing membership as soon as a node joins the cluster. This is all that's required for user level cluster information and notification to be available. The cluster members, as reported by Cnxman, are simply all nodes that have joined the cluster.

The SM portion of CMAN will be unused on a node unless (until) a system that uses the SM is started. This does not affect the Cnxman operation at all. SM combines the cluster membership information from Cnxman with the particular service registrations to manage the Service Groups.

3.2.1 Cluster User Interface

A node must first join the cluster before any cluster services or applications will operate. To join the cluster, a node must simply run the command: `cmn_tool join`. This tells the CMAN kernel module to initiate the process of joining the cluster. The command simply initiates the join process and returns before the process is complete.

Different clusters may exist on the same network and must be distinguishable, so a cluster is uniquely identified by a name defined in the cluster configuration file (see figure ??). The `cmn_tool` program looks up this name (from the CCS system) to determine what cluster to join. A cluster is initially formed by the first node attempting to join it.

A cluster member can leave the cluster with the command: `cmn_tool leave`. If other clustering systems are still running on the node, this command will fail and report a "busy" error (unless the force option is used.)

CMAN gets the local node name from `system_utsname.nodename` when the kernel module is loaded. This name can be changed through the cluster application interface covered below.

Cluster status and members, can be displayed on any member by viewing `/proc/cluster/status` and `/proc/cluster/nodes`.

3.2.2 Cluster Application Interface

CMAN has a cluster socket API to allow userspace to communicate with the cluster manager and also to provide some basic inter-node communication facilities. This API is defined in the header file `cnxman-socket.h` and is summarized here ¹.

¹Information from Patrick Caulfield

To use the interface, a socket of address family `AF_CLUSTER` and type `SOCK_DGRAM` must be created as shown in figure 3.1 using the `socket(2)` system call. The returned file descriptor can be used as usual with `ioctl` and to send and receive data.

```
fd = socket(AF_CLUSTER, SOCK_DGRAM, CLPROTO_CLIENT);
```

Figure 3.1: Cluster socket creation

3.2.2.1 Messages

CMAN supports sending and receiving data between cluster members on its cluster sockets. This is not a high-speed bulk data channel. It is a packet based, reliable, unwindowed transport. Messages are limited to 1500 bytes. Client applications bind to an 8-bit port number in a manner similar to IP sockets - only one client can be bound to a particular port at a time. Port numbers below 10 are special; activity on them will not be blocked when the cluster is in transition or inquorate.

- `sendmsg(2)` is used to send messages to a specific node or to all other nodes in the cluster. The latter will be sent as a broadcast/multicast message (depending on the cluster configuration) to all nodes simultaneously. Nodes are specified by their node ID rather than IP address.
- `recvmsg(2)` receives messages from other cluster members. The `sockaddr` is filled in with the node ID of the sending node. `poll(2)` is supported. `recvmsg()` can also receive informational messages which are delivered as Out-Of-Band (OOB) messages if `MSG_OOB` is passed to `recvmsg()`.

3.2.2.2 Information

The following `ioctl(2)`'s are supported on cluster sockets to access cluster and node information.

- `SIOCCLUSTER_NOTIFY`
Tells CMAN to send the calling process a signal every time a node joins or leaves the cluster. The argument passed is a signal number.
- `SIOCCLUSTER_REMOVE_NOTIFY`
Removes the signal notification.
- `SIOCCLUSTER_GETMEMBERS`
Returns a list of cluster members. The argument is either `NULL` (in which case the number of members is returned) or a pointer to an array of `struct cl_cluster_node` which will be filled in.
- `SIOCCLUSTER_GETALLMEMBERS`
Same as previous, but also includes nodes that were previously members but are not any longer. Note that this does not include configured nodes that have never joined the cluster, only nodes that were once part of the current cluster.
- `SIOCCLUSTER_ISQUORATE`
Returns 1 if the cluster is currently quorate, 0 if not.

- **SIOCCLUSTER_ISACTIVE**
Returns 1 if the node is a cluster member, 0 if not.
- **SIOCCLUSTER_ISLISTENING**
Returns 1 if a program on the specified node is listening on a given port number, 0 if not. The parameter is a pointer to a `struct cl_listen_request`. This ioctl will wait until a response has been received from the node or a cluster state transition occurs.
- **SIOCCLUSTER_GET_VERSION**
Returns the version number of the CMAN cluster software. The parameter is a `struct cl_version` that is filled in.
- **SIOCCLUSTER_GET_JOINCOUNT**
Returns the number of active cluster subsystems on this node. Once this "join count" is zero, the node can properly leave the cluster.
- **SIOCCLUSTER_BARRIER**
Userspace interface to the barrier system. The parameter is a `struct cl_barrier_info`. See the kernel documentation for more information. CAP_ADMIN privilege is needed for this operation. (This userspace access to the barrier system is not fully tested.)
- **SIOCCLUSTER_SET_VOTES**
Changes the number of votes cast by this member of the cluster. Quorum will be recalculated and the other nodes notified. CAP_ADMIN privilege is needed for this operation.
- **SIOCCLUSTER_SETEXPECTED_VOTES**
Changes the number of votes expected by the cluster. Quorum will be recalculated and the other nodes notified. CAP_ADMIN privilege is needed for this operation.
- **SIOCCLUSTER_KILLNODE**
Forcibly remove another member from the cluster. The parameter is a node number. CAP_ADMIN privilege is needed for this operation.
- **SIOCCLUSTER_SERVICE_REGISTER**
Registers a service with the Service Manager. The parameter is the service name. The cluster socket itself identifies the user program to the SM in the kernel and must remain open for the lifetime of the service program.
- **SIOCCLUSTER_SERVICE_UNREGISTER**
Unregisters a service with the Service Manager; takes no parameters apart from the cluster socket that was originally used to register the service.
- **SIOCCLUSTER_SERVICE_JOIN**
Instructs SM to join the service group for the registered service associated with the cluster socket. Takes no parameters.
- **SIOCCLUSTER_SERVICE_LEAVE**
Instructs SM to leave the service group for the registered and joined service associated with the cluster socket. Takes no parameters.
- **SIOCCLUSTER_SERVICE_SETSIGNAL**
Tells CMAN to deliver a signal to the calling process when a change occurs in the service group membership. The cluster socket must be associated with a registered service. The signal number is the parameter.

- `SIOCCLUSTER_SERVICE_STARTDONE`

The service program must use this `ioctl` when it has completed processing a start event from SM. The parameter is the event ID of the start event that was processed.

- `SIOCCLUSTER_SERVICE_GETEVENT`

Returns details of the next service event for the service group associated with the cluster socket. The parameter is a `struct cl_service_event`.

- `SIOCCLUSTER_SERVICE_GETMEMBERS`

Returns `struct cl_cluster_node` details for each node that was a service group member in the last start event read by the `GETEVENT` `ioctl`.

- `SIOCCLUSTER_SERVICE_GLOBALID`

Returns the global unique ID of the service group associated with the registered and joined cluster socket.

`setsockopt(2)` operations are used to start and stop CMAN on a node, configure multicast communications and change the node name. They all require `CAP_ADMIN` privilege and should only be used by `cman_tool` and its derivatives.

3.2.3 Node Information

3.2.4 Membership

3.2.4.1 Joining the cluster

To join a cluster, a node begins listening on the network for `HELLO` messages from existing cluster members. It sends a join request (`JOINREQ`) to the source of the first `HELLO` it hears. The node receiving the join request will then begin a transition to add the new node to the cluster.

The first node to join the cluster will not hear any `HELLO` messages on the network and will wait for a period of `joinwait_timeout` seconds. Once this time has expired, the node will form the cluster by itself with itself as the first member.

To prevent multiple nodes from forming initial clusters by themselves in parallel, nodes send a `NEWCLUSTER` message when they begin the join process. When a node waiting to form a cluster sees a `NEWCLUSTER` message from another node it will back off and wait an additional period of time listening for a `HELLO` message before forming a cluster on its own. The back off period is based on a simple hash of a node's name causing at least one node to back off a slightly shorter period of time than others. This node will then form a cluster on its own and any other nodes waiting to form a cluster will see its `HELLO` message and join it.

The timeout values mentioned can be configured as follows and have the specified defaults if they are not:
`joinwait_timeout`, default 11 seconds
`join_timeout`, default 30 seconds

3.2.4.2 Heartbeats

When a node joins the cluster, the `cman_hbeat` thread is started to send heartbeat messages every `hello_timer` seconds.

When the thread is awoken (by a timer) it sends a HELLO broadcast/multicast message to all other nodes in the cluster. Upon getting this message, other nodes update the "last_hello" time for the sending node.

After sending HELLO, the heartbeat thread scans the list of cluster members looking at the last_hello time of each. If this time is beyond the deadnode.timeout period a transition is started to remove the node from the cluster. The thread also checks for a dead quorum device and recalculates quorum if one exists and has failed.

The timing values mentioned can be configured as follows and have the specified defaults if they are not:
 hello_timer, default 5 seconds
 deadnode.timeout, default 21 seconds

3.2.4.3 Leaving the cluster

3.2.5 Transitions

3.2.6 Quorum

Quorum is implemented by a voting scheme where each node is given a number of votes to contribute to the cluster when it joins. When the sum of votes from all cluster members is greater than or equal to a fixed "quorum_votes" value, quorum is declared. When nodes leave the cluster or fail, their votes are removed. The cluster becomes inquorate when the sum of member votes falls below quorum_votes.

The number of votes assigned to each node is a static property of a node set during configuration. By default, if no votes value is specified, a node has 1 vote. The cluster's "expected_votes" value is derived from this; it is the sum of all possible votes (the sum of votes for all nodes in the config file.) Quorum_votes is then set to $(\text{expected_votes} / 2) + 1$. If all nodes are given the default 1 vote, quorum is achieved when over half the nodes in the configuration file are members.

If more nodes join the cluster than were originally seen when the cluster was formed, the vote total of all members will exceed expected_votes. In this case the effective in-core expected_votes value is dynamically increased to the sum of all votes. To maintain the split-brain protection that quorum provides, this value does not decrease when nodes subsequently leave the cluster and the vote count decreases.

An administrator can manually change the expected_votes value in a running cluster with the command: `cman_tool expected <votes>`. This must be done with caution to avoid split-brain clusters from becoming quorate.

CMAN exports the `kc1_is_quorate` function in the kernel. It simply reports if the cluster is quorate or not. An `ioctl` on a cluster socket (mentioned above) does the same for user space.

Quorum is simply a boolean global property of the cluster. In general, Cnxman continues running as usual whether the cluster has quorum or not. Cluster services, however, are not enabled when the cluster is inquorate. The enabling of cluster services is discussed further in a later section.

The following values can be configured as shown and have the specified defaults if they are not:
 expected_votes, default is sum of votes of all nodes
 votes, default 1

3.2.6.1 Quorum device

CMAN has a kernel interface to support in general devices or other mechanisms that can contribute to quorum (figure 3.2). This may be particularly useful for clusters with two (or any even number) of votes. A software module would be written to manage or use a specific quorum device. The module would register with CMAN and simply report if the device is available/alive and contributing its votes to the current cluster or not. The quorum device module would be registered on each cluster node.

```
int kcl_register_quorum_device(char *name, int votes);

int kcl_unregister_quorum_device(void);

int kcl_quorum_device_available(int yesno);
```

Figure 3.2: generic quorum device interface

3.2.6.2 Two-node clusters

Clusters with only two nodes are of particular interest. With the traditional quorum algorithm and the traditional assignment of one vote per node, if one node fails the other will be unable to operate because the cluster is inquorate.

option 1

This is the simplest but least interesting option and is only a partial solution to the problem. With this option one node is given one vote and the other node is given zero votes. If the node with one vote fails, quorum is lost and the remaining node is inoperable until the failed node returns. However, if the node with zero votes fails, quorum will be maintained and the remaining node can continue running.

option 2

This option is to use a quorum device as described in the previous section. Both nodes and the quorum device are all given one vote. Either node can fail and the remaining node can continue operating with the quorum device accessible. A node must be able to exclusively claim ownership of the quorum device in the event that it can no longer communicate with the other node. There are currently no quorum device modules available for CMAN.

option 3

The third option is simple to use but more subtle to understand. In this arrangement, both nodes are given one vote, but `expected_votes` is set and fixed at one as an exception for this special two-node case. This unusual behavior allows two things:

1. Split-brain can occur. Both nodes can form independent clusters by themselves, become quorate, and begin enabling clustering services (fencing, lock manager, file system.) An explanation of how this becomes safe is described next.
2. If both nodes form a cluster together (as would usually happen when there is no network-partition/split-brain), then either one of the nodes can fail and the cluster will remain quorate, cluster recovery will proceed, and the remaining node will continue operating. This is our aim.

How is it safe to allow two-node split-brain to happen? As previously mentioned, during a split-brain event both of the separately formed clusters become quorate and begin enabling cluster services independently. If the GFS service level is enabled on both nodes, the file system would be quickly corrupted.

When cluster services are enabled, the lowest service level, namely, the Fence Domain (FD) service, is enabled first. Not until the FD service has started will the higher level services be enabled (GDLN, GFS).

In the process of starting, the FD service handles the "unknown node state problem" (UNSP - a larger, more general issue discussed elsewhere.) When only two nodes are present, the solution to UNSP also prevents harm from split-brain. The FD service solves the UNSP (and makes a two-node split-brain safe) by fencing any nodes with "unknown state" when the FD is first started in the cluster. Specifically, these are the nodes listed in cluster.xml that have not joined the cluster by the time the FD is enabled.

In practice it would look like this: when there's a split-brain, the FD service is started on both nodes which will attempt to fence each other – it's a race. The winner will go ahead enabling the other services and operate by itself. The loser will either be rebooted or unable to access the storage depending on the fencing method. Either way, GFS on the winner will operate safely.

If a Fibre Channel switch fencing method is used, it is possible that both nodes would succeed in fencing each other from the storage and then both be stuck with GFS unable to access the file system. So, a fencing method that reboots the other node is preferred to avoid this case.

The conditions illustrated above had both nodes starting up when they encounter the split-brain. Things work slightly differently if split-brain happens while the nodes are both cluster members or one is a cluster member and the other is starting up.

If both nodes are members and split-brain happens, services on both nodes are stopped and then restarted in the layered fashion already mentioned. Once again there will be a fencing race which one node will win. If one node is a member and the other is joining when a split-brain occurs, then the second node creates its own separate cluster. The second node will fence the other when enabling services. Again, GFS will be enabled and unfenced on only one node at any point in time.

3.2.7 Event Notification

There are three types of notification that result from a cluster event. The first two, kernel callbacks and user signals, are simply available for general use and serve no specific purpose in this architecture. The third is an internal notification from Cnxman to Service Manager which may result in SM initiating recovery for a registered service.

3.2.7.1 Kernel callbacks

Any kernel subsystem can register a callback function with Cnxman (figure 3.3). It will be called when the cluster has been reconfigured in some way. A "reason" parameter in the callback indicates what has occurred and a node ID parameter is filled in with a node ID when relevant. Possible reasons are: CLUSTER_RECONFIG, DIED, LEAVING, NEWNODE. A kernel subsystem must remove its registered callback when it is shut down or removed from the kernel.

```

int kcl_add_callback(void (*callback)(kcl_callback_reason, long));

int kcl_remove_callback(void (*callback)(kcl_callback_reason, long));

```

Figure 3.3: kernel callback interface

3.2.7.2 Userland signals

The `SIOCCLUSTER_NOTIFY` ioctl on a cluster socket is used by a user program to register for cluster event notification. CMAN will send the program a signal if the configuration changes. The specific signal number is specified as the ioctl argument. (Also see Cluster Application Interface.)

3.2.7.3 Service Manager control

When the cluster membership changes, Cnxman notifies the Service Manager. If a cluster member has failed, SM will check if the failed node was a member of any service groups. If it was SM will coordinate recovery for all service groups the node was in. If the node failure caused the cluster to lose quorum, SM will suspend services the failed node was in, but not restart them until the cluster regains quorum. (The cluster may regain quorum by a new node joining the cluster, the failed node rejoining, or by administrator intervention.)

A somewhat unusual phenomenon may arise given the behavior described here. This happens when a cluster member fails causing the cluster to lose quorum. Service groups that were *not* used by the node are *not* suspended by SM; these services are not affected by the departed node and need no recovery. These services continue to run as usual even while the cluster is inquorate. This can also happen if a node simply leaves the cluster causing quorum to be lost.

This behavior is safe in split-brain scenarios due to the way services are enabled in a newly formed cluster. If a new split cluster is formed and becomes quorate, it will begin enabling services with the lowest level first as usual. The low service level is I/O fencing. Nodes on the inquorate side of the split cluster, some of which may still be running services, will be fenced by the fence domain on the quorate side of the cluster when the fence domain is enabled. The services running in the inquorate cluster are then cut off from shared resources before the same services are enabled in the new quorate cluster on the other side of the network partition.

3.2.8 Barriers

CMAN implements general cluster barriers. The barriers are used internally and are exported for external use in the kernel (figure 3.4) and user space (shown previously.)²

kcl_barrier_register

Register a new barrier with the given name in which the given number of nodes are expected to participate. This returns an error if the name is already registered with different number of nodes. It returns success if the name is already registered with the same number of nodes or not registered. Note that registering a barrier on one node registers it on all other nodes in the cluster as well. If the number of nodes is 0, the barrier waits for all cluster members to join.

²Information from Patrick Caulfield

```

int kcl_barrier_register(char *name, unsigned int flags, unsigned int nodes);

int kcl_barrier_wait(char *name);

int kcl_barrier_delete(char *name);

int kcl_barrier_setattr(char *name, unsigned int attr, unsigned long arg);

int kcl_barrier_cancel(char *name);

```

Figure 3.4: kernel barrier interface

kcl_barrier_wait

Enable the named barrier and wait for the barrier to be reached on all participating nodes. The return values from this function may be 0: the barrier was reached by all nodes, -ESRCH: the cluster left a state transition and the number of nodes specified for the barrier was not the special number 0, -EINTR: a signal is waiting for the process that was waiting, -ENOTCONN: the barrier was canceled.

kcl_barrier_delete

Remove the named barrier from the system. AUTODELETE barriers are automatically removed when all nodes have passed.

kcl_barrier_setattr

Set or change properties of a barrier with the following attribute options:

- BARRIER_SETATTR_AUTODELETE - Toggle the auto-delete flag.
- BARRIER_SETATTR_ENABLED - Enable the barrier (alternative to actually calling `kcl_barrier_wait`.)
- BARRIER_SETATTR_NODES - Change the expected number of nodes.
- BARRIER_SETATTR_CALLBACK - Set the kernel callback function for the barrier. The parameters are the barrier name and an integer status (see return value for `kcl_barrier_wait`.)
- BARRIER_SETATTR_TIMEOUT - Set a time-out on an inactive barrier. Barriers otherwise have no time-out.

kcl_barrier_cancel

Cancel an outstanding barrier. If a node is waiting on the barrier, -ENOTCONN is returned. If a callback is set it is called with a status of -ECONNRESET.

3.2.9 Communications

Document messaging, `cman_comms` thread. All nodes must be visible to each other - some form of broadcast medium required.

Document `cnxman` communications layer.

3.2.10 Services

Cluster services in this context are subsystems like GFS, GDLM, or the Fence Domain system. Strict control mechanisms are necessary between them and the cluster manager (especially through direct callback functions.) As a symmetric cluster manager, CMAN/SM is designed to support symmetric services, although server-based (asymmetric) services can naturally be used as well.

SM is agnostic regarding particular services and refers to all services generically, representing any service instance as a Service Group. The separation between "cluster members" as managed by Cnxman and "service group members" as managed by SM allows a great degree of flexibility. General properties of service groups have already been listed in section 2.2.2.

3.2.11 Service API

The SM interface is given in `service.h`. A set of exported functions are called by a service to set things up with SM and to shut down. Control flow in the other direction, SM to the service, is through the set of callback functions each service provides with its registration.

3.2.11.1 Service formation

The basic functions for setting up a service on a node are shown in figure 3.5: `register`, `join`, `leave`, `unregister`. The first two, `register` and `join`, set up a service and get it running on a node. The last two, `leave` and `unregister`, shut down the service.

```
int kcl_register_service(char *name, int namelen, int level, struct kcl_service_ops *ops,
                        int unique, void *servicedata, uint32 *local_id);

void kcl_unregister_service(uint32 local_id);

int kcl_join_service(uint32 local_id);

int kcl_leave_service(uint32 local_id);

void kcl_global_service_id(uint32 local_id, uint32 *global_id);

void kcl_start_done(uint32 local_id, int event_id);
```

Figure 3.5: `service.h` service interface

kcl_register_service

When a service is instantiated on a node (as opposed to the service being available but not used in a running context) it first calls `kcl_register_service`. This function sets up local structures for the service but does not activate it or make the node known to the service group.

For registration the service provides a unique name to identify the service group. The name must distinguish the service from others and among multiple instances of the service. The name may originate from a command used to start the service or it may be the name of the specific resource the service instance is providing access to. The service also provides a set of callback functions and an opaque context argument to

be used with the callbacks. The level parameter is defined for each service in `service.h`. It is chosen according to the dependencies between services. Register returns a locally unique ID that is used to reference the local service group in subsequent calls.

kcl_join_service

After registration, a service calls `kcl_join_service` to actually join other cluster members in the given service group. The local ID returned from registration is the only parameter. Once this function is called, the first start callback from SM may be received; potentially before the join function returns. There may also be an indefinite delay after the join function returns and before the first start callback is received. This could be the case if the cluster is inquorate (as discussed below, services are not enabled/started until the cluster is quorate.)

kcl_global_service_id

Once started, a service may call `kcl_global_service_id` which returns a globally unique identifier for the service group. This may be of general use to a service as the unique name used to register may not be convenient to use in internal functions or communications.

kcl_leave_service

When a service wishes to shut down on a node, it should leave the service group using `kcl_leave_service`. Again, the local ID is the only parameter. The service should continue to operate until the leave call returns. Another node may fail during the process of leaving the service group and the service may be stopped and started to handle the necessary recovery while waiting for the leave function to return. Before the leave function returns a final stop will be the last callback the service receives.

kcl_unregister_service

Once a service has left the service group, the local structures can be removed by calling `kcl_unregister_service` using the local ID for the final time.

kcl_start_done

This function is called by a service to notify SM that it has completed processing a start callback. The service only need call this function for the most recent start callback in the event of interrupted recoveries. The event ID parameter is the ID from the start callback to which the `kcl_start_done` call corresponds.

3.2.11.2 Service control

Cluster management is applied to services through the service callback functions in figure 3.6. A cluster service is controlled by SM to whatever extent it reacts to the callbacks.

stop

The stop (or suspend) callback tells the service that the group membership is to change. The service instance suspends operations to the extent that it would produce incorrect results while operating with multiple nodes having inconsistent views of the service group membership. The stop callback is not acknowledged by the service. In general, the service should process the stop synchronously, i.e. before its stop callback function returns. The only stop parameter is the service's context pointer.

The service must be able to handle a stop callback called at any time, even while it's processing a start

```

struct kcl_service_ops
{
    int (*stop)(void *servicedata);
    int (*start)(void *servicedata, uint32 *nodeids, int count, int event_id, int type);
    void (*finish)(void *servicedata, int event_id);
}

```

Figure 3.6: `service.h` service control functions

callback or waiting for a finish callback, i.e. a stop can follow another stop, a start or a finish callback.

Stop will be called if a new node joins the service group or if a service group member leaves or fails. It is guaranteed that stop will be called and will have *returned* on all service group members before a start callback is delivered to any group members. Furthermore, it is guaranteed that if a node has failed, all service groups the node was in have been stopped before any are started.

start

The start (or recover) callback tells the service that the group membership has been changed and provides the updated member list. The start is delivered to the service on all members of the new service group. This includes new members that have joined and excludes old members that departed. After a node joins a service group, the first start callback is when the service actually begins its operation. The start is asynchronous and it is an appropriate place for a service to do recovery or reconfigure to work with the new member list. When this asynchronous processing of the start is complete, the service must call `kcl_service_done` (see above) to notify SM.

If a service group member fails while the service is processing the start, all remaining group members will receive a stop callback, regardless if the node has called `kcl_service_done` or not. The stop effectively resets the nodes in the service group and prepares them to receive another start callback. If a node is still processing a start when a stop is delivered, the start should abort. There is no need to call `kcl_start_done` for an aborted start.

The event ID parameter is used to distinguish start callbacks, to match a specific finish callback to a specific start and to match a `kcl_start_done` with a specific start. The `nodeids` parameter is an array of `count` node ID's that form the new service group membership. The service is responsible for freeing this memory.

finish

The finish (or resume) callback tells the service that all service group members have completed start processing and have called `kcl_start_done`. This is an appropriate time for the service to resume normal operations as it knows all group members have completed recovery. The event ID parameter is the same as that used for the start for which the finish applies.

3.2.12 Service Events

Internal SM handling of events.

3.3 GDLM

This section details the design, construction and unique advantages provided by GDLM.

3.3.1 Motivation

DLM's are the de facto standard for high availability distributed locking. Several features of a DLM are the basis for this and are covered below.

- *Availability*

DLM offers the highest possible form of availability. There is no number of nodes that can fail, and no selection of nodes that can fail such that the DLM cannot recover and continue to operate.

- *Performance*

DLM achieves excellent performance by increasing the likelihood of local processing. This performance advantage is a simple result of the way DLM's are designed. Each node "masters" locks that it creates. When a node is operating on its own files/data, it is master of the corresponding locks and its requests for the locks are immediate, requiring no network requests at all.

When multiple nodes are using the same locks and contending for exclusive access to them, the performance becomes equivalent to a single lock server with one network request/reply.

Furthermore in the case of contention, the lock manager role of arbitrating among contending requests is distributed among all nodes in the cluster, so there is no one node relied upon to handle all the work. What can be very high load on a single server can be negligible when distributed across many nodes. The slowdown caused by a heavily loaded lock manager disappears.

- *Elimination of Bottlenecks*

Bottlenecks in a cluster architecture severely limit both performance and scalability. DLM is designed to eliminate any possible bottleneck. The following briefly describes this.

Bottleneck issues arise quickly when single machines are used as servers. This includes many things: memory, CPU, and network limitations. A DLM, however, distributes lock traffic evenly across all nodes. As more nodes are added, more distribution takes place to reduce any additional load.

Memory bottlenecks In a replicating lock manager, each lock server consumes memory to hold a copy of the entire cluster's lock state. This can become very large and if running on a GFS node, the lock server or application can be forced to swap to disk.

A DLM distributes lock state (memory) among all nodes. When DLM locks are mastered locally, no memory is replicated for them anywhere. For DLM locks mastered remotely, two copies of the lock are kept, one in the memory of the node owning the lock and one on the lock's master node. In contrast, when using three redundant lock servers, there are four copies of every lock. Therefore, not only is the DLM's memory usage evenly distributed across all nodes, but simple calculations show the DLM's total usage to be half or less.

CPU bottlenecks A single lock manager must process all requests from all nodes in the cluster. If it is also replicating state, it also has to process replication traffic among all redundant servers. The limitations of the CPU can quickly be reached by adding nodes or increasing file system load. With a DLM, just as before, all this processing is balanced across all nodes.

Network bottlenecks A DLM is not a replication system and therefore has far less network traffic which consequently increases performance. A replicating lock server quickly becomes impractical because of all the replication overhead. Operations become very slow because of heavy network traffic and response is only as fast as the slowest server. Each replicating node added makes things worse.

- *Scalability*

Many of the DLM characteristics mentioned above for performance also contribute to scalability. Bottlenecks are very often a limitation to scalability which is why symmetric architectures (like GFS and DLM) are designed with extra effort to be symmetric (it's much simpler and quicker to manage things asymmetrically from a single server, the problems are trivial in comparison.)

When all nodes using a DLM split the management tasks there is no one node or subset of nodes that can become bogged down due to the addition of too many nodes or too much application load.

- *Manageability*

The DLM requires no special nodes or special programs that an administrator would need to manage. The DLM maintains the symmetric "all nodes are equal" concept. This is a fundamentally simpler organization than other common systems that require various collections of nodes and lock management programs to be understood, arranged, started, configured, and managed separately. Additionally, the requirements of a separately managed lock management system are often different than the main requirements presented by the real cluster being managed. The intersection of these two differing sets of requirements can be confusing and far more work intensive than the management-free DLM.

- *Kernel Implementation*

When using DLM there are no user-space components that the kernel subsystems actively rely upon. This can be a big benefit as loads increase and memory becomes scarce.

GFS is a kernel service and farming its functions out to user-space is not only slow but dangerous in low-memory situations as it can lead to machine deadlocks. DLM being entirely in kernel space can manage its own memory more directly.

This becomes even more important with the kernel part of CLVM, as it deals with suspended devices which can cause page cache memory to balloon out of control while metadata is updated. As painfully discovered, mixing user-space and kernel space in this way can result in complex deadlocks beyond our control.

3.3.2 Requirements

External interfaces and behaviors:

- Support GFS's lock request interface.
- Support CLVM's lock request interface.
- Support standard DLM lock request interface.
- Support arbitrary independent lock spaces.
- Be fully recoverable from all possible failure conditions.
- Be stand-alone (independent of particular usage).
- Be self-configuring (no user input or management required).
- In-progress requests simply block during recovery.
- Requests processed and returned after a failed node's locks are cleared.

3.3.3 Lock Spaces

A lock space in a specific instance of the GDLM service. Any application that would like to take advantage of distributed locking can create a lock space to use. As instances of the application on different nodes request the same lock space name, they will operate together in the same lock space. The specific node ID's of the nodes in the lock space are provided each time the Service Manager starts the lock space following a transition.

Within a lock space, resources an application references itself will exist; none others. The GFS application must, of course, use a separate lock space for each file system because identical resource names will exist in every file system. e.g "inode 128".

So, multiple independent lock spaces allow many applications (different types and different instances of the same type) to operate at once independently from each other in the cluster.

3.3.3.1 Usage

This interface is used by GDLM applications such as GFS to request new lock spaces and subsequently release them.

```
int gdlm_new_lockspace(char *name, int namelen, gdlm_lockspace_t **lockspace, int flags);

int gdlm_release_lockspace(gdlm_lockspace_t *lockspace);
```

Figure 3.7: gdlm.h lock space usage

gdlm_new_lockspace: The `name` and `namelen` parameters naturally refer to the name of the lock space being requested. The `lockspace` parameter is the lock space context pointer returned to the application to be used as the basis for future lock requests. When the function returns, the lock space can be used immediately. The `flags` parameter can enable or disable certain features in a lockspace. Currently, the only flag is `NOTIMERS` which causes requests to not be subject to time-outs after waiting for too long.

gdlm_release_lockspace: The `lockspace` parameter is the same context pointer as was returned from the lock space creation.

3.3.3.2 Functions: lockspace handling

Within GDLM a lock space is represented by a `gd_ls_t` structure called the "lockspace" or "ls" struct. It is analogous to a file system's superblock structure. The following functions deal with lockspace structures.

`gd_ls_t *find_lockspace_by_global_id(uint32 id)`

Searches GDLM's global list of lockspaces to find one with the specified (global) ID. (A lockspace global ID is defined later.)

`gd_ls_t *find_lockspace_by_name(char *name, int namelen)`

Searches GDLM's global list of lockspaces to find one with the specified name.

Null	no access is requested
Concurrent Read	read access is requested with no other restriction
Concurrent Write	write and read access is requested with no other restriction
Protected Read	read access is requested with no write access granted to other processes
Protected Write	write and read access is requested with no write access granted to other processes
Exclusive	read and write access is requested with no read or write access granted to other processes

Table 3.1: Resource Lock Mode Description

```
gd_ls_t *allocate_ls(int namelen)
```

Allocates memory for and returns a `gd_ls_t` structure for a lockspace with a name of specified length.

```
void free_ls(gd_ls_t *ls)
```

Frees the specified lockspace structure.

```
int new_lockspace(char *name, int namelen, gdlm_lockspace_t **lockspace, int flags)
```

Allocates and initializes a unique lockspace with given name. Registers and joins a corresponding Service Manager service group using the lockspace name.

```
int gdlm_release_lockspace(gdlm_lockspace_t *lockspace)
```

Leaves the Service Manager service group associated with the lockspace and deallocates everything associated with the given lockspace. The lockspace must continue all its regular group functions until the function leaving the service group has returned.

```
int gdlm_new_lockspace(char *name, int namelen, gdlm_lockspace_t **lockspace, int flags)
```

Does system-level GDLM initialization if not done already. Calls `new_lockspace` shown above.

3.3.4 Resources, Locks and Queues

There are two fundamental objects in a lock space: a resource block (RSB) representing a resource and a Lock Block (LKB) representing a lock. (These terms are inherited from VaxCluster nomenclature.) The two corresponding internal structures are: `gd_res_t`, `gd_lkb_t`.

An LKB contains a pointer to the RSB it is held against. The node owning the lock is identified in the LKB's `nodeid` field. An LKB also has a granted mode and requested mode. Both modes are valid while an LKB is being converted, otherwise only the granted or requested mode is valid. The following table shows the possible LKB modes and their compatibility and table 3.1 gives a general interpretation of the modes.

	NL	CR	CW	PR	PW	EX
NL	1	1	1	1	1	1
CR	1	1	1	1	1	0
CW	1	1	1	0	0	0
PR	1	1	0	1	0	0
PW	1	1	0	0	0	0
EX	1	0	0	0	0	0

An RSB contains its unique identifying name in addition to three lock queues. The name is treated as

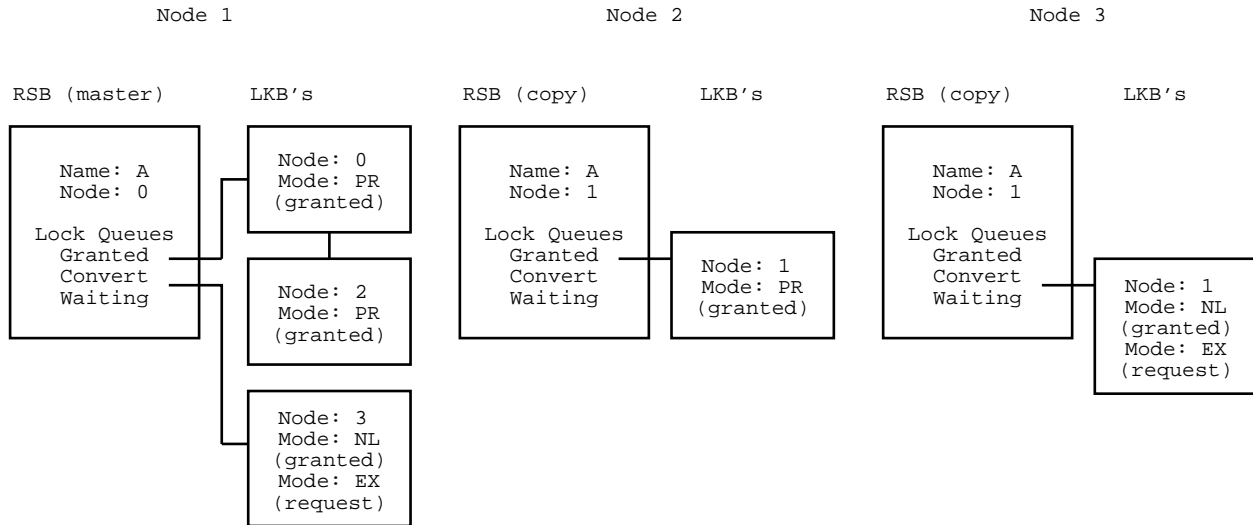


Figure 3.8: An RSB's lock queues

an arbitrary binary sequence of data up to 64 bytes in length; it is not interpreted by GDLM. The three lock queues are: granted, converting and waiting. The granted queue contains currently granted locks on the resource. The convert queue contains granted locks that are being converted to a requested mode not compatible with existing granted locks. The wait queue contains ungranted locks with requested modes incompatible with the granted mode of existing granted locks.

An RSB has a master node that is defined in the Resource Directory. The first node to request a lock on an RSB becomes its master. A subsequent node wanting to lock the RSB must first look up the master node ID in the Resource Directory. The locks from all nodes against an RSB are kept in the master RSB's queues.

A node holding a lock against a remotely mastered RSB also keeps a local copy of the RSB. This is where the node keeps a copy of its own locks which officially exist on the master RSB. These copies are used in recovery if the master node of the RSB fails.

The node ID in the master RSB is set to zero (`res_nodeid`). The LKB's owned by the RSB master also have node ID's set to zero, otherwise the node ID identifies the owner. In the case of a non-master RSB copy, the RSB node ID identifies the master and the LKB's (all locally owned) also have node ID's of the master node. Figure 3.8 gives a simple illustration.

When a lock is requested or converted, its requested mode is compared with the granted mode of all existing locks on a resource. If it is compatible with them the request is granted immediately and the lock is placed on the RSB granted queue. If the request is not compatible with existing locks, "blocking AST's" are sent to the nodes holding the locks preventing the request from being granted. The request is kept on the converting or waiting queue.

Whenever a lock is converted to a lower mode or unlocked an attempt is made to grant locks on the convert and wait queues. Demoting a lock to a lower mode will never block.

3.3.4.1 Lock ID's

When a new LKB is created it is assigned an ID that is used to identify and look up the LKB uniquely in the lockspace. The LKB can be referenced using its ID without referring to the resource it's held against.

The LKB ID's are unique only within the context of the local lockspace, i.e. the same LKB ID may refer to a different lock if used on another machines or in another lockspace.

LKB ID's are 32 bits and have two 16 bit parts. The bottom 16 bits are a random number between 0 and `lockidtbl_size-1`. This random number specifies the bucket for the LKB in the lockspace's `lockidtbl` hash table (holds all LKB's in the lockspace.) The upper 16 bits are a sequentially assigned per-bucket id. (Because the 16 bit ID's per bucket can roll over, a new ID must be checked against the ID of all LKB's in the bucket to avoid duplication.)

When a lock is held against a remotely mastered RSB, one LKB exists on the node holding the lock and another LKB representing the same lock exists on the master (the "master copy" or MSTCPY.) In addition to its own ID, an LKB saves the ID of the matching LKB on the remote node. Both local and remote ID's are included in messages between nodes, each node using the ID appropriate for its own LKB and simply saving the other to include in messages. For a local lock held against a locally mastered RSB, the remote LKB ID is not used.

3.3.4.2 Functions: RSB/LKB structures

```
int find_or_create_rsb(gd_ls_t *ls, gd_res_t *parent, char *name, int namelen,
                     int create, gd_res_t **rp)
```

Find an RSB with given name in lockspace's RSB list and return it. If create is set and the RSB is not found, create an RSB with the given name and return it. When an RSB is found, increment reference count of the structure. When a parent RSB is provided, both name and parent must match to return an existing RSB.

```
void release_rsb(gd_res_t *r)
```

Decrement the reference count of the RSB structure. Remove it from the lockspace's RSB list and deallocate if the reference count becomes zero. Also decrement the reference count of the parent RSB if one exists. Remove the resource directory entry for the RSB if it was removed.

```
void hold_rsb(gd_res_t *r)
```

Increment the reference count of the RSB structure.

```
gd_res_t *search_hashchain(osi_list_t *head, gd_res_t *parent, char *name, int namelen)
```

Search the given list for an RSB with given name.

```
void lkb_add_ordered(osi_list_t *new, osi_list_t *head, int mode)
```

Insert a lock's list element into the given list in order of mode.

```
void lkb_enqueue(gd_res_t *r, gd_lkb_t *lkb, int type)
```

Put a lock on the RSB queue specified by type (granted, converting or waiting).

```
int lkb_dequeue(gd_lkb_t *lkb)
```

Remove a lock from whichever RSB queue it is on.

```
int lkb_swqueue(gd_res_t *r, gd_lkb_t *lkb, int type)
```

A combination of the previous two operations done atomically, removing the lock from a queue and add it to a new one.

```
void res_lkb_enqueue(gd_res_t *r, gd_lkb_t *lkb, int type)
```

```
int res_lkb_dequeue(gd_lkb_t *lkb)
```

```
int res_lkb_swqueue(gd_res_t *r, gd_lkb_t *lkb, int type)
```

Versions of above that lock the RSB before doing the operation.

```
gd_lkb_t *create_lkb(gd_ls_t *ls)
```

Allocate a new LKB structure with a new unique ID.

```
void release_lkb(gd_ls_t *ls, gd_lkb_t *lkb)
```

Remove the given LKB from a lockspace's LKB hash table (lockidtbl) and free it.

```
gd_lkb_t *find_lock_by_id(gd_ls_t *ls, gdlm_lkid_t lkid)
```

Look up the LKB with given ID in a lockspace's LKB hash table (lockidtbl).

3.3.5 Cluster Management

A GDLM lockspace is managed by the Service Manager as a Service Group. Each time SM gives the lockspace the "start" callback, the group of node ID's using the lockspace is provided. A lockspace keeps two lists relative to nodes: `ls_nodes` and `ls_nodes_gone`. How and when these lists are updated is covered in the Recovery section. In general, the `ls_nodes` list is the current group of nodes in the lockspace. The `ls_nodes_gone` list is empty except during recovery when it contains nodes that were in the previously running lockspace but are not in the present lockspace.

Any node using a lockspace is represented by one `gd_node_t` structure in a global GDLM node list. This structure contains globally constant (not variable or lockspace-specific) properties of a node: its unique node ID and IP address (both from the cluster manager).

The `gd_csb_t` structure represents a node in the context of a specific lockspace (Cluster System Block is a term from VaxClusters). It points to the global `gd_node_t` structure and contains the lockspace-specific property `gone_event`, an event ID discussed later identifying the transition during which the node left the lockspace.

When a lockspace is created and becomes a Service Group, SM returns for it two ID's. A local ID that uniquely identifies the lockspace on the node and a global ID that uniquely identifies the lockspace in the cluster. The global ID is the most widely used as it is part of all inter-node GDLM communications.

3.3.5.1 Functions: node information

```
gd_node_t *allocate_node(void)
```

Allocate a node structure for the global nodes list.

```
void free_node(gd_node_t *node)
```

Free a node structure.

```
gd_csb_t *allocate_csb(void)
```

Allocate a CSB structure for a lockspace-specific nodes list.

```
void free_csb(gd_csb_t *csb)
```

Free a CSB structure.

```
gd_node_t *search_node(uint32 nodeid)
```

Search the global nodes list for a node structure with specified node ID.

```
void put_node(gd_node_t *node)
```

Release a reference on a global node structure. When the reference count reaches zero, release it.

```
int get_node(uint32 nodeid, gd_node_t **ndp)
```

Return an existing node structure with specified node ID, incrementing its reference count or allocate a new node structure with the specified node ID

```
int init_new_csb(uint32 nodeid, gd_csb_t **ret_csb)
```

Return a new lockspace-specific CSB structure with specified node ID. The CSB structure points to a global node structure for the node in question that contains the node's global properties.

```
void release_csb(gd_csb_t *csb)
```

A combination of `put_node` and `free_csb`.

```
uint32 our_nodeid(void)
```

Returns the local node ID. This value is obtained from the cluster manager when the lockspace is first started.

```
void add_ordered_node(gd_ls_t *ls, gd_csb_t *new)
```

Add a CSB structure to a lockspace's nodes list in order of node ID's.

```
int in_nodes_gone(gd_ls_t *ls, uint32 nodeid)
```

Returns TRUE if a node with specified node ID is in a lockspace's `ls_nodes_gone` list.

3.3.5.2 Control mechanics

A lockspace is controlled by the Service Manager using the three functions discussed in section 2.2.2. The following describes how GDLM handles these three callback functions and responds to the cluster events they signal. This is the management of the recovery process. The details of the recovery process itself are left for a later section.

GDLM has a specific thread (`gdlm_recoverd`) in charge of processing control input from the Service Manager. This thread also carries out the recovery routines it initiates. This is a global thread that works on behalf of all GDLM lock spaces.

start

When SM calls a lockspace's `gdlm_ls_start` function, it means that the lockspace should be enabled to do request processing. To do that the lockspace must step through a recovery procedure cooperatively with all nodes in the lockspace. This recovery process is executed asynchronously by the `gdlm_recoverd` thread. When recovery is done, the lockspace signals to SM that it's done using the `kcl_start_done` function.

The `gdlm_ls_start` function allocates a `gd_recover_t` structure to record the relevant information for asynchronous processing of the start. The structure saves the list of node ID's provided by SM that represent the new lockspace membership. The start is unique and sequentially ordered among other control functions by the "event ID" provided by SM in each start callback. The event ID is saved in `gd_recover_t` and in the lockspace's `ls_last_start` field.

stop

When SM calls a lockspace's `gdlm_ls_stop` function, it means there is a forthcoming change to the lockspace's membership and the lockspace should stop normal processing so all lockspace nodes can reconfigure. If the lockspace were to continue processing requests after stop, the lockspace would produce inconsistent results

as different nodes may be operating on the basis of differing lockspace membership. Stopping the lockspace while all nodes adjust the membership and perform recovery is essential to correct operation.

The `gdlm_ls_stop` function acquires the lockspace's `in_recovery` lock in write mode. This lock is held in read mode by all processes while processing requests in GDLM (also by `gdlm_recvd` while processing remote requests.) Acquiring this lock guarantees that when the stop callback returns to SM, no further lock processing will occur. The lockspace's `ls_last_stop` field is set to the current value of `ls_last_start`.

finish

When SM calls a lockspace's `gdlm_ls_finish` function, it means all nodes that received the prior start have completed recovery. The finish callback is accompanied by an event ID that matches the event ID of the start to which the finish applies. This is saved in the lockspace's `ls_last_finish` field.

Callback processing

Once a lockspace is created, the first SM callback it receives is a start. This arrives during or immediately after the return of `kcl_join_service` (from the SM API).

After a lockspace receives any of the callbacks above (stop, start, finish), `gdlm_recoverd` looks at the state recorded from the callback (or callbacks, because the callbacks are processed asynchronously multiple can be received between processing of the last) to determine what needs to be done. The possible items recorded by the callback functions are: flags indicating which callback functions have been received since last checking, new `gd_recover_t` struct(s) from start, and the `ls_last_stop`, `ls_last_start`, `ls_last_finish` values.

For any lockspace that may have some recovery to handle, the `gdlm_recoverd` thread calls `do_ls_recovery`. The first thing done by this function is a call to `next_move` that collects and combines all the unprocessed callback state and decides what the next action should be. All possible combinations of callback state can be reduced to one of the following which `next_move` will return: `DO_STOP`, `DO_START`, `DO_FINISH`, `DO_FINISH.STOP`, `DO_FINISH.START`. If the move includes a start, then the relevant `gd_recover_t` struct is returned as well.

The `next_move` function combines SM callbacks to arrive at one of the "DO" values summarized in the following list. Because the callbacks are processed asynchronously and only a start is acknowledged when completed, multiple callbacks can arrive between processing. The event ID's recorded for each start in the `gd_recover_t` struct and the values `ls_last_stop`, `ls_last_start`, `ls_last_finish` allow the ordering of the callbacks to be determined.

- `DO_STOP`: A lone stop was received, or a start was received followed by a stop.
- `DO_START`: A lone start was received, or a stop was received followed by a start.
- `DO_FINISH`: A lone finish was received.
- `DO_FINISH.STOP`: A finish was received followed by a stop, or a finish was received followed by a start followed by a stop.
- `DO_FINISH.START`: A finish was received followed by a stop followed by a start.

`do_ls_recovery` combines the "DO" value from `next_move` with the current lockspace state to determine what recovery functions to call and to assign the next lockspace state. The main recovery function `ls_reconfig` is called when a `DO_START` or `DO_FINISH.START` is returned from `next_move`.

3.3.5.3 Recovery states

A lockspace is moved through the states (`ls_state`) listed below by `do_ls_recovery`. The next state is determined by a combination of the `next_move` value and the success or failure of `ls_reconfig`, the recovery function called by `do_ls_recovery`. Figure 3.9 shows the state transitions. In the figure, START, STOP, FINISH, FINISH_STOP, FINISH_START represent the `next_move` values shown above. The "ok" and "error" values represent the result of `ls_reconfig`. The recovery routines in `ls_reconfig` can return an error if a lockspace member node fails during recovery.

- INIT: A lockspace is created in this state. Waiting for a start.
- INIT_DONE: A lockspace is done with recovery from its first start. Waiting for a finish.
- CLEAR: A lockspace is not in any recovery state having finished the previous recovery successfully.
- WAIT_START: A lockspace has received a stop or the recovery for the last start has failed. Waiting for a start.
- RECONFIG_DONE: A lockspace is done with recovery from a start. Waiting for a finish.

3.3.5.4 Functions: recovery management

`gd_recover_t *allocate_recover(void)`

Called from `gdlm_ls_start` to allocate a `gd_recover_t` structure with which to save the state associated with the start for asynchronous processing. This state includes the list of nodes being started and the event ID of the start.

`void free_recover(gd_recover_t *gr)`

Frees a `gd_recover_t` structure.

`int gdlm_ls_stop(void *servicedata)`

GDLM's stop function called by the Service Manager.

`int gdlm_ls_start(void *servicedata, uint32 *nodeids, int count, int event_id, int type)`

GDLM's start function called by the Service Manager.

`void gdlm_ls_finish(void *servicedata, int event_id)`

GDLM's finish function called by the Service Manager.

`void enable_locking(gd_ls_t *ls, int event_id)`

Enable locking based on the completion of recovery for the event with specified ID. Locking will not be enabled if a more recent stop event has arrived. Locking is enabled by setting the LS_RUN lockspace bit and unlocking the `in_recovery` write lock.

`int ls_reconfig(gd_ls_t *ls, gd_recover_t *gr)`

Called when a lockspace is started to do recovery.

`int ls_first_start(gd_ls_t *ls, gd_recover_t *gr)`

Called in place of `ls_reconfig` to do first-time recovery (which is reduced) the first time a lockspace is started.

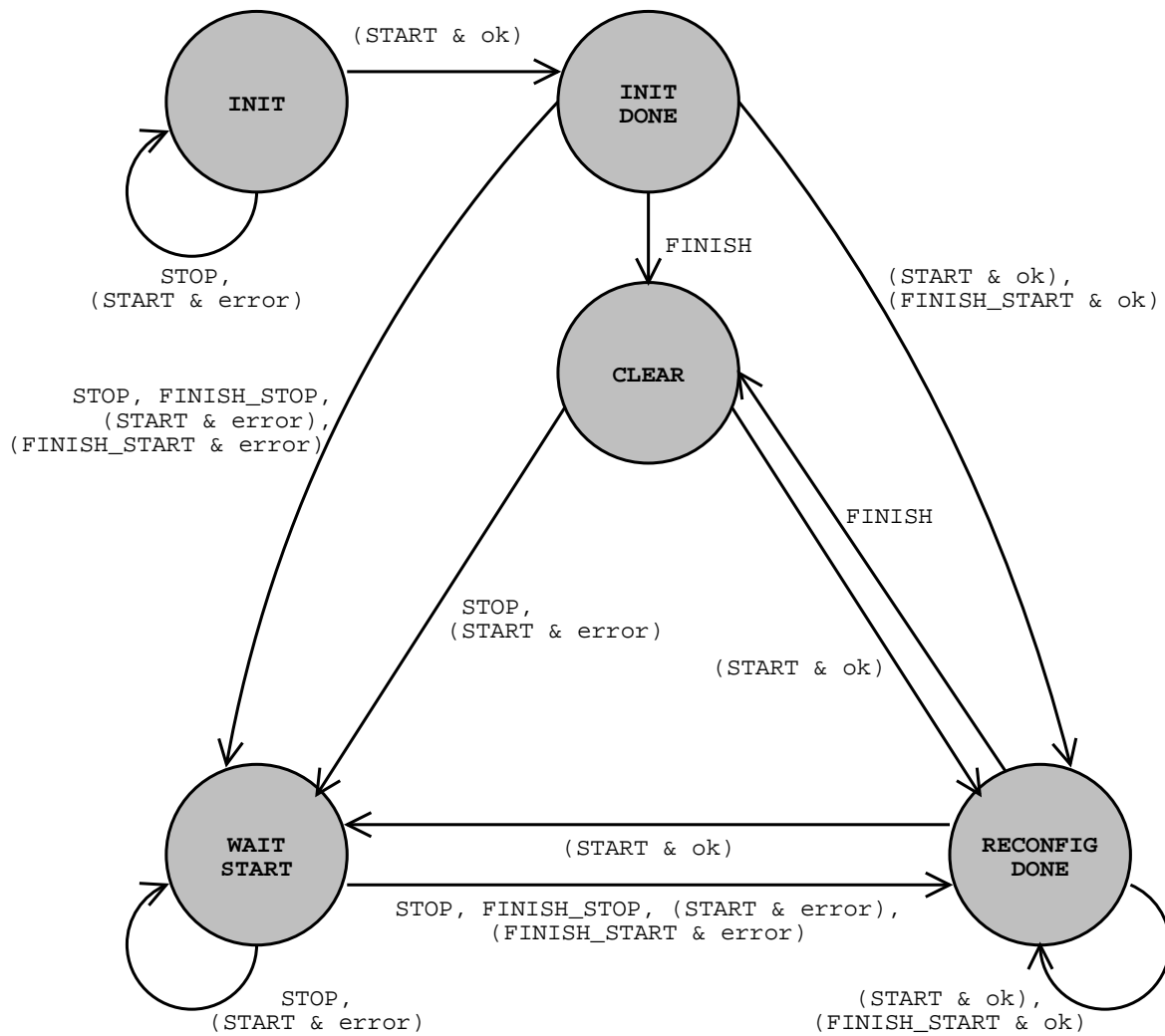


Figure 3.9: Lockspace recovery states

```
void clear_finished_nodes(gd_ls_t *ls, int finish_event)
```

Called when a finish is received to remove node structures from the `ls_nodes_gone` list of nodes for which recovery was conducted.

```
int next_move(gd_ls_t *ls, gd_recover_t **gr_out, int *finish_out)
```

Combines the state recorded from Service Manager callback functions to determine what recovery action to take next.

```
void do_ls_recovery(gd_ls_t *ls)
```

Step through the lockspace recovery states based on input from `next_move` and the success or failure of recovery routines (`ls_reconfig`) that it calls.

```
gd_ls_t *get_work(int clear)
```

Called by the `gdlm_recoverd` thread to get a lockspace for which recovery is necessary. A lockspace is marked as needing recovery attention with the WORK flag set by `ctrlld_kick`.

```
int gdlm_ctrlld(void *arg)
```

The top-level `gdlm_recoverd` function that processes recovery for lockspaces.

```
void ctrlld_kick(gd_ls_t *ls)
```

Marks a lockspace as needing recovery attention. Called by Service Manager callback functions on the lockspaces referenced.

3.3.5.5 Cluster information

In addition to being controlled by the cluster manager in response to cluster transitions, GDLM needs to query the cluster manager for some basic information. First, GDLM needs to know the node ID of the node it's running on, i.e. its own node ID. Second, GDLM needs to look up the IP address of another node ID. The Service Manager provides lock space membership in terms of node ID's. GDLM needs to communicate with these nodes and therefore also needs IP addresses. There are basic exported CMAN functions that provide this information along with a variety of other information that is unused.

3.3.6 Lock Requests

This section discusses the primary function of GDLM, the processing of asynchronous lock requests. It begins with the exported lock request API and continues with details of the internal request processing.

3.3.6.1 Usage

Figure 3.10 shows the GDLM locking interface comprised of the two functions `gdlm_lock` and `gdlm_unlock`. Before a description of their use in an application, one GDLM structure used in the interface, `gdlm_lksb_t`, should be described.

Shown in figure 3.11, the Lock Status Block (LKSb, another term from VaxCluster's DLM) is a structure used for both input and output parameters. Every lock request must be accompanied by an LKSb structure allocated by the application.

The LKSb provides a place for GDLM to return results of a request. The results are ready and can be read by the application once the asynchronous completion callback (AST) is called. There can be multiple parts

```

int gdlm_lock(gdlm_lockspace_t *lockspace, uint32 mode, gdlm_lksb_t *lksb, uint32 flags,
             void *name, unsigned int namelen, gdlm_lkid_t parent,
             void (*lockast)(void *astarg), void *astarg, void (*bast)(void *astarg, int mode),
             gdlm_range_t *range);

int gdlm_unlock(gdlm_lockspace_t *lockspace, gdlm_lkid_t lkid, uint32 flags, gdlm_lksb_t *lksb,
               void *astarg);

```

Figure 3.10: gdlm.h locking interface

to the result, but the main one is the "status" (**sb_status**). The status indicates the success or failure of the request in general; if successful (status is zero) the other results are valid. The second result is the lock ID (**sb_lkid**) of the granted lock that is used for conversions and to unlock. The third result is the LVB (**sb_lvbptr**) which is described in a later section and the fourth result is flags (**sb_flags**) containing extra information.

The LKSB is also how an application provides input for requests. The lock ID field is used to specify a lock to convert or unlock. The LVB field is used to update a resource's LVB in conjunction with a conversion or unlock.

```

struct gdlm_lksb
{
    int          sb_status;
    gdlm_lkid_t sb_lkid;
    char         sb_flags;
    char         *sb_lvbptr;
};
typedef struct gdlm_lksb gdlm_lksb_t;

```

Figure 3.11: gdlm.h lock status block

Making a lock request

Acquiring a lock on a resource is a matter of calling `gdlm_lock` with proper parameters. Calling one of the functions in figure 3.10 is often referred to as "queuing a request" because the result is returned asynchronously. If the function call returns a non-zero value, the request failed to be queued, will not be processed, and no AST will be called.

The essential elements of a lock request are: a resource name and length, the mode of lock being requested, an LKSB reference for input and output values, a completion callback function pointer, a blocking callback function pointer, and a parameter to include in both callback functions. Additional settings are: a parent lock ID for hierarchical locking, a range to do range locking and flags. The following list explains each parameter to `gdlm_lock`.

- *lockspace*: the service needing to use GDLM should create its own lockspace. The lockspace pointer is returned by GDLM when a lockspace is created (figure 3.7) by the application and it provides a context for all subsequent GDLM lock requests.
- *mode*: the mode of the requested lock (table 3.1).
- *lksb*: a pointer to a Lock Status Block allocated by the application per lock. When an initial lock request is complete (the AST is called), the LKSB status field gives the result of the operation. A

status of zero means the requested lock has been granted. A status of -EAGAIN means the NOQUEUE flag was used and the requested lock could not be granted immediately. Other errors mean there was some kind of non-standard internal error.

If the request was successful, the LKSB lock ID field was filled in by GDLM with the granted lock's unique ID (per-node). This ID must be used as input in the same field for subsequent conversions or unlocking of the lock.

- *flags*: NOQUEUE means the request should not be queued if the lock cannot be granted immediately. If this is the case, the request should return -EAGAIN in the LKSB status field. CONVERT must be set if the request is a conversion of an existing lock. QUECVT forces a conversion to the back of the convert queue. All other conversion requests ahead of it must be granted before it can, enforcing a FIFO ordering on the convert queue. This can be used to avoid indefinite postponement and is only applicable when converting a lock to a more restrictive mode. CONVDEADLK allows GDLM to demote the granted state of a converting lock to avoid conversion deadlock.
- *name*: the name of the resource being locked. Treated as an arbitrary array of binary data.
- *namelen*: the name length in bytes.
- *parent*: the parent lock ID used for hierarchical locking. (See section on hierarchical locking.)
- *lockast*: the function GDLM should call when the request completes.
- *astarg*: the parameter GDLM should include in callback functions related to the lock being requested. Generally used by the application to identify the specific lock to which the callback pertains.
- *bast*: the function GDLM should call when the lock being requested blocks another request. Using this function's mode parameter, GDLM specifies the mode that the blocked request has requested.
- *range*: a structure specifying a range with start and end values. (See section on range locking.)

If AST routines or the AST parameter are provided in conversion requests, they overwrite the previously set values. AST routines should avoid blocking; they may submit other GDLM requests.

Making an unlock request

To release a lock from a resource the `gdlm_unlock` function is used. The following list describes the function parameters.

- *lockspace*: see previous description
- *lkid*: the lock ID of the lock being released. In `gdlm_lock` this value is specified in the LKSB.
- *flags*: CANCEL means that any outstanding request for the specified lock should be canceled. If the request was a conversion, the lock will be returned to its previously granted state. The LKSB status returned for the canceled request will be -GDLM_ECANCEL.
- *lksb*: the result of the unlock is returned in the status field. A status of -GDLM_EUNLOCK indicates success. An unlock should never fail.
- *astarg*: the parameter GDLM should include in the completion callback function. The AST function used in the prior `gdlm_lock` call is called by GDLM when the unlock is complete.

3.3.6.2 Request processing

There are three possible outcomes when an application queues a new lock request. Three similar results are possible for a conversion request.

Possible outcomes for a queued lock request (a Failed result is only possible when the request includes the NOQUEUE flag):

1. Finished. A new LKB is on the RSB's granted queue and a completion AST indicating success is sent to the application.
2. Failed. No new LKB exists and a completion AST indicating "try again" is sent to the application.
3. Delayed. A new LKB is on the RSB's wait queue and blocking AST's are sent to applications holding locks blocking the request.

Possible outcomes for a queued conversion request:

1. Finished. The LKB remains on the RSB's granted queue and a completion AST indicating success is sent to the application.
2. Failed. The LKB remains on the RSB's granted queue (with no mode change) and a completion AST indicating "try again" is sent to the application.
3. Delayed. The LKB is on the RSB's convert queue and blocking AST's are sent to applications holding locks blocking the request.

The following sequences show the main steps and functions involved in processing lock requests. Each outcome type (finished, failed, delayed) is considered individually for both locally and remotely mastered RSB's. Each request sequence begins in the unspecified context of an arbitrary application. Generally, each numerical step (1, 2, 3...) is called by the previous, forming a call trace. This is broken either by a transition to remote processing on a master node (M1, M2...), or by a change in thread context where the new thread is specified in brackets, e.g. [gdlm_recvd]. Each step performed on a remote master (M1, M2...) is called by the top-level `process_cluster_request()` that runs in the `gdlm_recvd` context.

A. local new request finishes

1. `dml_lock()`
 checks input params
 creates new LKB
2. `dml_lock_stage1()`
 `find_or_create_rsb()` returns RSB, finding existing or creating new
 master lookup for new RSB (local or remote)
3. `dml_lock_stage2()`
4. `dml_lock_stage3()`
 `can_be_granted()` returns TRUE
 `grant_lock()` adds LKB to RSB's granted queue and queues completion AST

5. `deliver_ast()` [`gdlm_astd`]
 - sets LKSB status to 0
 - calls application AST

B. local new request fails

1-3. A.1-3

4. `dml_lock_stage3()`
 - `can_be_granted()` returns FALSE
 - marks LKB to be deleted
 - queues completion AST
5. `deliver_ast()` [`gdlm_astd`]
 - sets LKSB status to -EAGAIN
 - calls application AST
 - frees LKB

C. local new request is delayed

1-3. A.1-3

4. `dml_lock_stage3()`
 - `can_be_granted()` returns FALSE
 - adds LKB to RSB's waiting queue
 - `send_blocking_ast()` queues local BAST's and sends remote BAST messages

Later, within the context of another request that unlocks or down-converts a lock on the same RSB:

5. `grant_pending_locks()`
 - `can_be_granted()` returns TRUE
 - `grant_lock()` adds LKB to RSB's granted queue and queues completion AST
6. `deliver_ast()` [`gdlm_astd`]
 - sets LKSB status to 0
 - calls application AST

D. local conversion request finishes

1. `dml_lock()`
 - checks input params
2. `convert_lock()`
 - `find_lock_by_id()` finds LKB being converted
3. `dml_convert_stage2()`
 - `can_be_granted()` returns TRUE

`grant_lock()` leaves LKB on RSB's granted queue and queues completion AST
`grant_pending_locks()` grants unblocked LKB's

4. `deliver_ast()` [`gdlm_astd`]
 sets LKSB status to 0
 calls application AST

E. local conversion request fails

1-2. D.1-2

3. `dml_convert_stage2()`
`can_be_granted()` returns FALSE
 adds LKB back to RSB's granted queue with original mode
 queues completion AST
4. `deliver_ast()` [`gdlm_astd`]
 sets LKSB status to -EAGAIN
 calls application AST

F. local conversion request is delayed

1-2. D.1-2

3. `dml_convert_stage2()`
`can_be_granted()` returns FALSE
 adds LKB to RSB's convert queue
`send_blocking_ast()` queues local BAST's and sends remote BAST messages

Later, within the context of another request that unlocks or down-converts a lock on the same RSB:

4. `grant_pending_locks()`
`can_be_granted()` returns TRUE
`grant_lock()` adds LKB to RSB's granted queue and queues completion AST
5. `deliver_ast()` [`gdlm_astd`]
 sets LKSB status to 0
 calls application AST

G. remote new request finishes

1. `dml_lock()`
 checks input params
 creates new LKB
2. `dml_lock_stage1()`
`find_or_create_rsb()` returns local RSB, finding existing or creating new

master lookup for new RSB (local or remote)

3. `dml_lock_stage2()`
4. `remote_stage()`
`add_to_lockqueue()` adds LKB to list awaiting reply (LQSTATE_WAIT_CONDGRANT)
5. `send_cluster_request()`
forms request message (REMCMD_LOCKREQUEST)
sends request to master node
- M1. `process_cluster_request()` [gdlm_recvd]
- M2. `remote_stage2()`
creates new master LKB
- M3. `dml_lock_stage3()`
`can_be_granted()` returns TRUE
`grant_lock()` adds LKB to RSB's granted queue
`reply_and_grant()` sends completion/granted reply to requesting node
6. `process_cluster_request()` [gdlm_recvd]
`find_lock_by_id()` gets LKB
7. `process_lockqueue_reply()`
`remove_from_lockqueue()` removes LKB from list awaiting reply
adds LKB to local RSB's granted queue
queues completion AST
8. `deliver_ast()` [gdlm_astd]
sets LKSB status to 0
calls application AST

H. remote new request fails

1-5. G.1-5

- M1. `process_cluster_request()` [gdlm_recvd]
- M2. `remote_stage2()`
creates new master LKB
- M3. `dml_lock_stage3()`
`can_be_granted()` returns FALSE
marks LKB to be deleted
- M4. frees LKB
- M5. sends reply to requesting node
6. `process_cluster_request()` [gdlm_recvd]
`find_lock_by_id()` gets LKB

7. `process_lockqueue_reply()`
 - `remove_from_lockqueue()` removes LKB from list awaiting reply
 - marks LKB to be deleted
 - queues completion AST
8. `deliver_ast()` [`gdlm_astd`]
 - sets LKSB status to `-EAGAIN`
 - calls application AST
 - frees LKB

I. remote new request is delayed

1-5. G.1-5

- M1. `process_cluster_request()` [`gdlm_recvd`]
- M2. `remote_stage2()`
 - creates new master LKB
- M3. `dml_lock_stage3()`
 - `can_be_granted()` returns `FALSE`
 - adds LKB to RSB's waiting queue
 - `send_blocking_ast()` queues local BAST's and sends remote BAST messages
- M4. sends reply to requesting node
6. `process_cluster_request()` [`gdlm_recvd`]
 - `find_lock_by_id()` gets LKB
7. `process_lockqueue_reply()`
 - `remove_from_lockqueue()` removes LKB from list awaiting reply
 - adds LKB to local RSB's waiting queue

Later, within the context of another request that unlocks or down-converts a lock on the same RSB:

- M5. `grant_pending_locks()`
 - `can_be_granted()` returns `TRUE`
 - `grant_lock()` adds LKB to RSB's granted queue
 - `remote_grant()` sends granted message to requesting node (`REMCMD_LOCKGRANT`)
8. `process_cluster_request()` [`gdlm_recvd`]
 - `find_lock_by_id()` gets LKB
 - adds LKB to local RSB's granted queue
 - queues completion AST
9. `deliver_ast()` [`gdlm_astd`]
 - sets LKSB status to 0
 - calls application AST

J. remote conversion request finishes

1. `dml_lock()`
checks input params
2. `convert_lock()`
`find_lock_by_id()` finds LKB being converted
3. `remote_stage()`
`add_to_lockqueue()` adds LKB to list awaiting reply (`LQSTATE_WAIT_CONVERT`)
4. `send_cluster_request()`
forms request message (`REMCMD_CONVREQUEST`)
sends request to master node
- M1. `process_cluster_request()` [`gdlm_recvd`]
`find_lock_by_id()` gets master LKB
- M2. `dml_convert_stage2()`
`can_be_granted()` returns TRUE
`grant_lock()` leaves LKB on RSB's granted queue
`reply_and_grant()` sends completion/granted reply to requesting node
5. `process_cluster_request()` [`gdlm_recvd`]
`find_lock_by_id()` gets LKB
6. `process_lockqueue_reply()`
`remove_from_lockqueue()` removes LKB from list awaiting reply
adds LKB to local RSB's granted queue
queues completion AST
7. `deliver_ast()` [`gdlm_astd`]
sets LKSB status to 0
calls application AST

K. remote conversion request fails

1-4. J.1-4

- M1. `process_cluster_request()` [`gdlm_recvd`]
`find_lock_by_id()` gets master LKB
- M2. `dml_convert_stage2()`
`can_be_granted()` returns FALSE
adds LKB back to RSB's granted queue with original mode
- M3. sends reply to requesting node
5. `process_cluster_request()` [`gdlm_recvd`]
`find_lock_by_id()` gets LKB
6. `process_lockqueue_reply()`
`remove_from_lockqueue()` removes LKB from list awaiting reply
adds LKB back to RSB's granted queue with original mode

queues completion AST

7. `deliver_ast()` [`gdlm_astd`]
 - sets LKSB status to `-EAGAIN`
 - calls application AST

L. remote conversion request is delayed

1-4. J.1-4

- M1. `process_cluster_request()` [`gdlm_recvd`]
 - `find_lock_by_id()` gets master LKB
- M2. `dml_convert_stage2()`
 - `can_be_granted()` returns `FALSE`
 - adds LKB to RSB's convert queue
- M3. sends reply to requesting node
5. `process_cluster_request()` [`gdlm_recvd`]
 - `find_lock_by_id()` gets LKB
6. `process_lockqueue_reply()`
 - `remove_from_lockqueue()` removes LKB from list awaiting reply
 - adds LKB to local RSB's convert queue

Later, within the context of another request that unlocks or down-converts a lock on the same RSB:

- M4. `grant_pending_locks()`
 - `can_be_granted()` returns `TRUE`
 - `grant_lock()` adds LKB to RSB's granted queue
 - `remote_grant()` sends granted message to requesting node (`REMCMD_LOCKGRANT`)
7. `process_cluster_request()` [`gdlm_recvd`]
 - `find_lock_by_id()` gets LKB
 - adds LKB to local RSB's granted queue
 - queues completion AST
8. `deliver_ast()` [`gdlm_astd`]
 - sets LKSB status to 0
 - calls application AST

3.3.6.3 Unlock request processing

Unlock requests are asynchronous and are processed much like conversion requests. Unlock requests can only succeed, however; they will never fail or be delayed in the sense of new or conversion requests. The following two sequences show the steps in processing unlock requests for local and remotely mastered locks.

A. local unlock request

1. `dml_unlock()`
checks input params
2. `dml_unlock_stage2()`
removes LKB from RSB granted queue
marks LKB to be deleted
queues completion AST
3. `deliver_ast()` [`gdlm_astd`]
sets LKSB status to `-GDLM_EUNLOCK`
calls application AST

B. remote unlock request

1. `dml_unlock()`
checks input params
2. `remote_stage()`
`add_to_lockqueue()` adds LKB to list awaiting reply (`LQSTATE_WAIT_UNLOCK`)
3. `send_cluster_request()`
forms request message (`REMCMD_UNLOCKREQUEST`)
sends request to master node
- M1. `process_cluster_request()` [`gdlm_recvd`]
`find_lock_by_id()` gets master LKB
- M2. `dml_unlock_stage2()`
removes LKB from RSB granted queue
frees LKB
- M3. sends reply to requesting node
4. `process_cluster_request()` [`gdlm_recvd`]
`find_lock_by_id()` gets LKB
5. `process_lockqueue_reply()`
`remove_from_lockqueue()` removes LKB from list awaiting reply
removes LKB from local RSB's granted queue
queues completion AST
6. `deliver_ast()` [`gdlm_astd`]
sets LKSB status to `-GDLM_EUNLOCK`
calls application AST

3.3.6.4 Canceling a request

An outstanding request can be canceled using the `gdlm_unlock` with the `CANCEL` flag. The `gdlm_unlock` return value or the completion AST status will be `-EINVAL` if the lock is granted before the request can be canceled. When successful, the cancel will cause the lock to be returned to its previously granted state (or deleted in the case of an initial request being canceled) and the completion AST status will be set to `-GDLM_ECANCEL`.

3.3.6.5 AST handling

The `gdlm_astd` thread is dedicated to calling completion and blocking AST functions in all lock spaces. Each LKB saves AST function pointers (and an argument pointer) that were provided in the request for the lock represented by the LKB. When a completion or blocking AST for the lock is needed, the LKB is placed on a global list of LKB's requiring ASTs to be called (`ast_queue`).

While LKB's exist on the `ast_queue`, the thread removes one at a time in the `process_ast` function. The `deliver_ast` function is then used to call either the completion or blocking AST for the LKB (it is called twice, once for each, if both types are needed). These routines must be aware that an LKB can be placed back on the `ast_queue` at any time once it has been removed for either type of AST. Locks must not be held while calling the application AST function because it may call back into GDLM with another request that can potentially queue another AST for the same LKB.

After delivering an AST, the `gdlm_astd` thread will free the LKB if the AST delivery was the last step in deleting it. This can also result in the associated RSB being freed.

Within the locking routines, the `queue_ast` function is called with an LKB and AST type to cause an AST to be delivered. In the previous request processing sequences, this was referred to as "queues completion AST" or "queues local BAST's". This will add the LKB to the local `ast_queue` if the LKB is local. If the LKB is remote, the function sends a message that will cause the remote node to add the LKB to its `ast_queue`. Remote completion ASTs are queued on the remote node implicitly with a message to a remote node indicating that a request has been granted.

3.3.7 Resource Directory

A lock space's Resource Directory (RD) stores the mapping of resource names to master node ID's. RD lookup requests provide a resource name as input and the reply contains the node ID of the master. If the requested RSB name is not in the RD, a new RD entry is created with the requested name and the master node set to the requesting node.

Every node in the lockspace operates a fraction of the RD. A hash function common to all nodes statically translates a resource name to a RD node ID. The RD segment on this node identifies the master node for the resource. So, an RSB has two associated node ID's. The first is the node holding the RD mapping and the second is the master node. These two may or may not be the same; the former is static and the later dynamic.

An RD entry is represented by a `gd_resdata_t` structure. This structure contains a resource name and the node ID of the RSB's master. Each node keeps its RD entries on a hash table (`ls_resdir_hash`). The lower 16 bits of a resource name's hash value are used for this hash table while the upper 16 bits of the hash value are used to determine the RD node.

When a node requests a lock on an RSB that does not exist locally, a RD lookup must take place to determine the RSB's master. The master ID is saved in the RSB so further lock requests on the RSB do not require an RD lookup. This lookup step, mentioned in the previous section simply as "master lookup for new RSB" may be a local or remote operation. It will most likely be remote (more likely with more nodes in the lock space) because the RD hash function evenly distributes the RD entries among nodes.

In the best case for a new lock request, no network requests are needed: the RD lookup is local and the master becomes local. In the worst case, two network requests are needed: the RD lookup is remote and a master exists that is remote.

The following steps are involved in looking up a resource master. It begins in `d1m_lock_stage1()` after `get_directory_nodeid()` returns a remote node ID; it is an expansion of the step in the previous section labeled "master lookup for new RSB".

1. `remote_stage()`
 `add_to_lockqueue()` adds LKB to list awaiting reply (LQSTATE_WAIT_RSB)

2. `send_cluster_request()`
 forms request message (REMCMD_LOOKUP)
 sends request to node

RD1. `process_cluster_request()` [gdlm_recvd]

RD2. `get_resdata()`
 finds existing entry for given name or creates new entry

RD3. sends reply to requesting node

3. `process_cluster_request()` [gdlm_recvd]
 `find_lock_by_id()` gets LKB

4. `process_lockqueue_reply()`
 `remove_from_lockqueue()` removes LKB from list awaiting reply
 sets RSB master node ID

3.3.7.1 Removing entries

When a master RSB is deleted its resource directory entry is also deleted. After unlinking the RSB from the local RSB list, the master node sends a removal message to the resource's directory node (or removes it directly if the directory node is also be the master node.) RD removal messages are asynchronous and unacknowledged which can lead to a variety of special cases described in the following.

Case 1

Problem: An RSB is deleted on master node M and an RD removal message is sent to node D. Node L sends a RD lookup to node D for same resource. Node D receives the lookup before the removal and a message specifying node M as master is returned to node L even though node M no longer has any record of the resource. Case 3 can arise here if a lookup now originates on node M. Case 2 occurs when the removal message from node L is processed by node D.

Solution: Node L sends its lock request to node M. Node M recreates the RSB and becomes the master for the resource again. When node M processes the lock request from L in `remote_stage2()` it passes `create=1` into `find_or_create_rsb()` to permit creation of a new RSB.

Case 2

Problem: In the previous example, the removal message processed on node D must not remove the entry because it is now in use again, having just been sent to node L. The removal message is invalid having been meant for the previous "version" of the specific RD entry. The removal message must be recognized as invalid and ignored by node D.

Solution: A counter is kept on each RD entry that is incremented for each lookup. The counter is returned with the lookup reply and is then included with the lock request when sent to the master node. The master node keeps track of the latest counter value for the RSB. When the master sends a removal message to the

directory node, it includes the latest counter value it has observed. Ordinarily, this matches the counter value in the RD and the entry is removed.

In the case where a lookup has occurred for the RD entry just before the removal message is received, as in the example above, the RD entry will have a larger counter value than the removal request. If this is the case the removal request is ignored.

Case 3

Problem: In the first example, a new request on node M could occur after M has sent the removal request, and after L's RD lookup. A lookup from M will be sent to node D for the resource. Node D will reply indicating that M itself is the master.

Solution: Node M should accept that the RD may indicate it is the master of a resource it is looking up.

Handling removal requests during recovery is simple. Because the resource directory is rebuilt during recovery from existing RSB's, removal requests for deleted RSB's can be discarded. Also, lookup counters in all RD entries and in all RSB's are reset to one to avoid the large task of finding correct counter values (there are no removal races to contend with during recovery so resetting all counters to one disrupts nothing.)

3.3.7.2 Functions: resource directory

```
uint32 name_to_directory_nodeid(gd_ls_t *ls, char *name, int length)
```

Returns the node ID of the directory node for the named resource.

```
uint32 get_directory_nodeid(gd_res_t *rsb)
```

Calls the previous function with the name of the RSB.

```
uint32 rd_hash(gd_ls_t *ls, char *name, int len)
```

Using the given name, compute a scaled hash value for the resource directory.

```
void add_resdata_to_hash(gd_ls_t *ls, gd_resdata_t *rd)
```

Add a new resource data struct to the resource directory hash table.

```
gd_resdata_t *search_rdbucket(gd_ls_t *ls, char *name, int namelen, uint32 bucket)
```

Search a hash table list for a resource data struct matching given name.

```
void remove_resdata(gd_ls_t *ls, uint32 nodeid, char *name, int namelen, uint8 sequence)
```

Remove resource data struct for given name from resource directory hash table.

```
int get_resdata(gd_ls_t *ls, uint32 nodeid, char *name, int namelen, gd_resdata_t **rdp, int recovery)
```

Find and return a resource data structure for given name, creating a new one if none is found.

3.3.8 Hierarchical Locking

Hierarchical locks allow structured or optimized forms of locking by applications. In particular, resources related in a tree structure are often a good candidate for hierarchical locks.

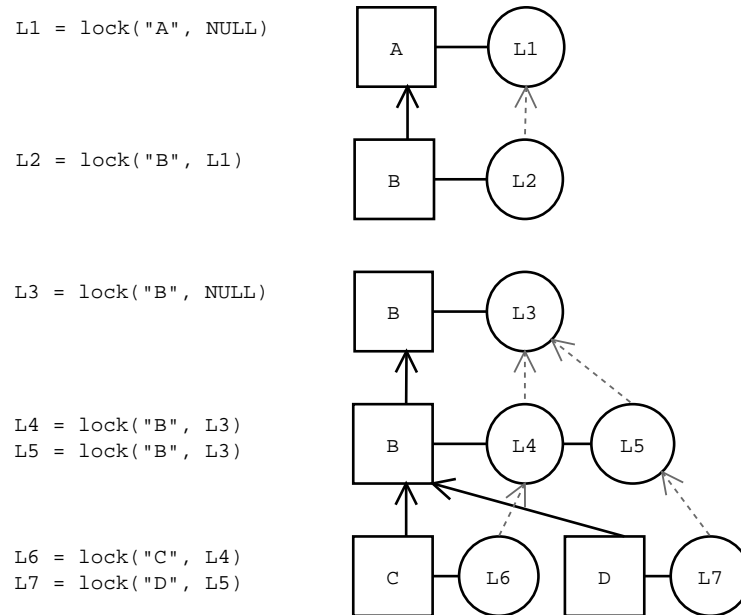


Figure 3.12: Hierarchical Locking

3.3.8.1 Usage

A "resource tree" is an association of parent/child RSB's. The resource tree is created using the parent parameter of the `gdlm_lock` function. An entire resource tree is mastered on the same node, a fact applications can use to optimize performance. As stated earlier, only local processing is required when a node locks RSB's it has mastered.

Given RSB A locked with LKB L1, a child-RSB B can be locked by requesting L2 for B and using L1 as the parent. A will be parent of B and L1 will be parent of L2.

The resource name space at each level of a resource tree is distinct, so child-RSB's with the same name but different parents are separate. Figure 3.12 illustrates these concepts. Each square represents a unique RSB and each circle a unique LKB. A solid arrow represents an RSB parent pointer and a dotted arrow an LKB parent pointer. The "L1", "L2", etc designations correspond to LKB ID's. Resource names in the figure are letters "A" to "D". The sample "lock" function takes a resource name and parent lock ID. The figure can be read from top to bottom as a sequence of operations.

3.3.8.2 Resource tree structure

The lockspace has a list of all root resources (`ls_rootres`, `res_rootlist`) and a root resource has a list of all its sub-resources (`res_subreslist`). So, a resource tree is kept in a flat list belonging to the root. All RSB's have a depth (root depth is 1), a pointer to the parent RSB (root parent is null), and a pointer to the root RSB (root points to itself).

A lock also keeps a pointer to its parent lock and a parent lock keeps a count of its children (but no list). While a lock has a positive child count it cannot be unlocked. This ensures that the resource tree remains connected.

3.3.9 Lock Value Blocks

A Lock Value Block (LVB) is 32 bytes of memory associated with a resource that is read and written by applications using locks against the RSB. The LVB data is read from or written to memory provided by the application and referenced in the LKSB (figure 3.11).

3.3.9.1 Usage

An application can write data to an RSB's value block by:

1. Obtaining a lock on the RSB with a mode of EX or PW.
2. Setting the LKSB's `sb_lvbptr` field to point at the data to be written.
3. Converting or unlocking the lock using the VALBLK flag in the request.

An application can read the data from an RSB's value block by:

1. Setting the LKSB's `sb_lvbptr` field to point at the memory where data should be written.
2. Converting or requesting a new lock using the VALBLK flag.

These rules imply that while an application holds a lock on an RSB in a mode greater than NL, the RSB's value block will not change. Holding a NL lock on an RSB guarantees nothing about the future contents of the LVB.

3.3.9.2 Value block copying

Figure 3.13 shows two locks held against a resource; one local and one remote. The resource's value block is in use by both nodes. Each copy of the value block in the system is illustrated. The master node maintains the official copy (LVB A), referenced by the master RSB structure. Local LKB's point to the LVB memory provided by the application through an LKSB (LVB's B and D). Memory is allocated by the RSB master to keep a copy of the remote lock's LVB (LVB C).

Copying of LVB values is illustrated using figure 3.13 for the following lock operations.

Node	Operation type	Mode	Copying
1	initial request	any	A → B
2	initial request	any	A → C, C → E, E → D
1	conversion	NL to higher	A → B
1	conversion	EX to lower	B → A
2	conversion	NL to higher	A → C, C → E, E → D
2	conversion	EX to lower	D → F, F → C, C → A
1	unlock	EX	B → A
2	unlock	EX	D → F, F → C, C → A

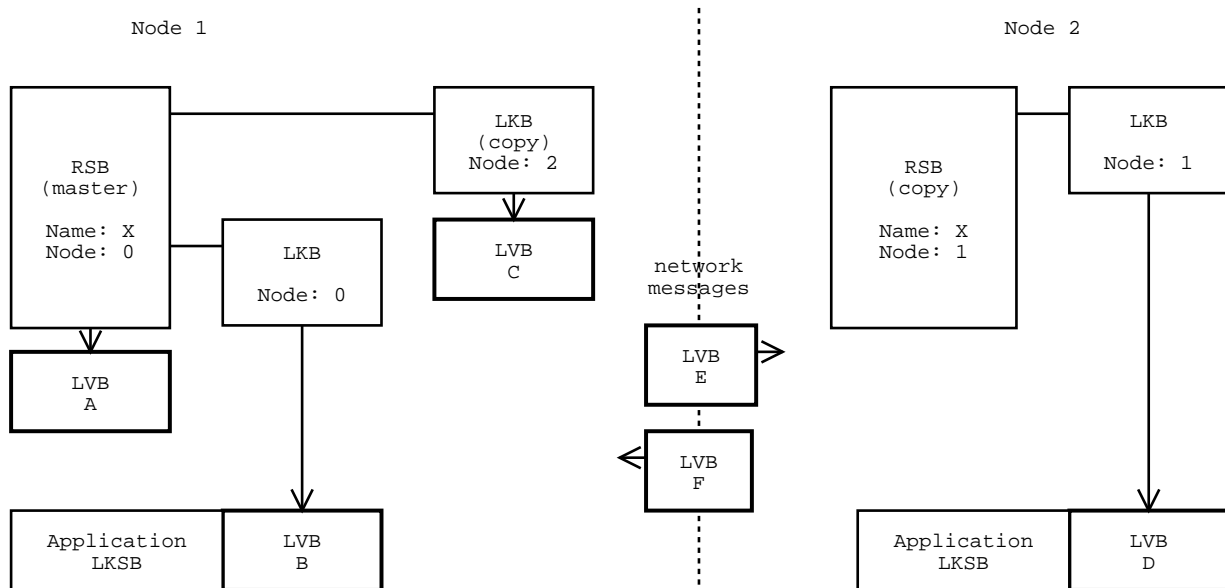


Figure 3.13: Lock Value Blocks

3.3.10 Lock Ranges

The range parameter in the lock request function allows a lock to be requested against part of a resource instead of the whole thing. This can be used to support things such as POSIX/`fcntl()` locking.

3.3.10.1 Usage

There are some simple rules governing the behavior of range locking:

- A lock request can specify a range (start and end values) by passing a pointer to a `gdlm_range_t` structure (figure 3.14.)
- The maximum range (0 - `FFFFFFFF.FFFFFFFF`) is used if no range is specified in a request.
- A conversion request can be used to change a lock's mode, range, or both.
- Two locks on a resource are incompatible if their modes are incompatible *and* their ranges overlap.
- Unlock requests operate as usual and do not specify ranges; a lock is released with whatever range it presently has.
- Blocking callbacks do not currently include the range specified by a blocked request. This will be added if it becomes useful.

3.3.11 Deadlock

GDLM does not search for possible deadlocks in a lockspace. There are two features that can help an application, however. The first is request timers. After waiting ungranted for a configurable period of time,

```

struct gdlm_range
{
    uint64    ra_start;
    uint64    ra_end;
};
typedef struct gdlm_range gdlm_range_t;

```

Figure 3.14: gdlm.h lock range structure

a lock request will time out, be canceled and a status of `-EDEADLOCK` will be returned. This can be a signal to the application of potential deadlock. The application can make the request again or take some other action. This feature can be disabled when creating a lock space using the `NOTIMERS` flag (timers are not used in GFS lock spaces). The feature could also be enabled and disabled per request.

If an application is making requests in such a way that conversion deadlock can occur, GDLM has a special flag that can be set per request (`CONVDEADLK`) that causes conversion deadlocks to be detected and forcibly resolved internally. The method of resolution is to demote to `NULL` the granted mode of one or more converting locks so that one of the requests can proceed. The locks that are demoted remain queued for their requested modes and will complete when they are granted. When the affected locks complete, however, the `DEMOTED` flag is set in the `LKSB` to notify the application that the originally granted mode was released in the process of promotion.

3.3.12 Recovery

Each node in a lock space performs the same series of recovery steps in parallel. Barriers are used to synchronize all the nodes at the end of some steps. This is necessary only when a step relies on the previous being completed on *all* nodes.

Many recovery steps involve requesting information from other nodes, so during recovery all nodes must be ready to send requested information to the others. As previously mentioned, the `gdlm_recoverd` thread executes the recovery steps while the `gdlm_recvd` thread processes requests received from other nodes in addition to replies to its own requests.

3.3.12.1 Blocking requests

As previously mentioned, the `in_recovery` lock held in write mode is used to block request processing during recovery. Local requests attempt to acquire the lock in read mode and block until the write lock is released when recovery is complete. This means that a context making an asynchronous lock request will block during recovery.

The `gdlm_recvd` thread also acquires `in_recovery` in read mode while processing remote requests. However, this thread must continue to function during recovery and cannot block. Therefore, if a request that is not recovery-related is received and would block on `in_recovery`, it is saved on a list of delayed requests to be processed after recovery completes. This is the lockspace's "requestqueue" list.

The requestqueue should not contain an excessive number of delayed requests because most new requests block in the requesting context before any remote operations are initiated. The requestqueue holds requests that were being processed on other nodes when recovery began. It may also hold requests that were made

and sent from nodes on which recovery was initiated slightly later than on other nodes.

3.3.12.2 Update node list

As the first recovery step, the function `ls_nodes_reconfig` is called to update a lockspace's `ls_nodes` and `ls_nodes_gone` lists (mentioned previously) based on the latest member list from the cluster manager. Node structures for departed members (those in `ls_nodes` but not in the latest member list) are moved to `ls_nodes_gone`. Node structures are added to `ls_nodes` for new members.

The number of departed nodes is returned to the caller, but before returning a barrier is used to wait for all nodes to complete this step of adjusting the lockspace membership. This is necessary because the next step depends on all nodes using the new membership as the basis for rebalancing directory entries.

The first time a lockspace is started and performs recovery, a shortened version of this function is used (`ls_nodes_init`) that simply adds node structures to the `ls_nodes` list for all current members.

3.3.12.3 Rebuild resource directory

The second recovery step, `resdir_rebuild_local`, rebuilds the resource directory on the new group of nodes. Even if nodes are only being added this step is necessary because the directory segment mapped to each node will change with the change in node count.

A node first clears all existing RD entries in `resdir_clear`. In the main body of this step the node sends a RECOVERNAMES request to each node asking for the resource names that it (the requesting node) is now in charge of keeping in its segment of the directory. When a node receives this request, it looks through its list of RSB's for which it is master and calculates the resource directory node for each (based on the name.) If the calculated directory node ID matches the requesting node's ID, the RSB name is included in the reply to the requesting node. The function on the receiving end is `resdir_rebuild_send`.

In `resdir_reuild_local`, each RECOVERNAMES request is synchronous. When the reply is received, all the RSB names are read from the returned data and new RD entries are created for each, the master being set to the ID of the node replying. Multiple requests and replies may be needed to one node to transfer all the names. Specially formatted reply data indicates the end of the resource names a node has to send. To handle these multiple requests, each request contains the last RSB name that was received. This is used by the node sending the names to find the next starting point.

Waiting for a RECOVERNAMES reply must be interruptible as the other node (or any other node) may have failed requiring the current recovery process to be aborted.

3.3.12.4 Clear defunct requests

A node will next purge resource directory related requests that are in progress (lookups, removals, replies) in `purge_requestqueue`. All these requests that originated before recovery are now invalid due to the rebuilding of the directory. Nodes with outstanding requests of this type will resend them to the appropriate nodes once recovery is complete.

Requests received by a node after recovery has begun are saved on the lockspace's requestqueue. It is from this list that the requests are removed. In addition, requests from failed nodes are also removed from the requestqueue at the same time.

At this point a barrier is used to wait for all nodes to finish rebuilding the resource directory (again, waiting will be interrupted and recovery aborted if a node fails.)

3.3.12.5 Mark requests to resend

During normal operation, LKB's for outstanding remote requests are kept on the lockspace's "lockqueue" list. The LKB's "lockqueue state" defines the exact reason for being on the lockqueue. An LKB is removed from the lockqueue when the reply for the request is processed.

Due to recovery, some of the outstanding LKB requests on the lockqueue may need adjustment. The `lockqueue_lkb_mark` function goes through the lockqueue and flags some LKB's for special handling during recovery and some for resending after recovery.

If an LKB has a lockqueue state of `WAIT_RSB` it is a new lock request and a resource directory lookup is in progress. The LKB is flagged (`LQRESEND`) to have the lookup resent after recovery (probably to a new directory node.) If the destination node for the lookup is still alive and received the request after recovery began, its `purge_requestqueue` will remove it. Or, if the destination node processed the request before recovery began, the requesting node may have a reply in its own requestqueue. This reply would also be purged (by the local `purge_requestqueue`.) All the purging sets the stage for cleanly resending the requests when recovery finishes.

The other lockqueue LKB's of interest are those with outstanding requests sent to nodes that are now gone. These requests are flagged to be resent (`LQRESEND`) to the new master node when recovery finishes (the new master node will be determined later in recovery.)

Among those LKB's marked for resending, if an LKB has a lockqueue state of `WAIT_CONDGRANT` it is a new lock request that does not yet have a granted state. This LKB is additionally flagged with `NOREBUILD` to prevent it from being rebuilt on the new RSB master later in recovery.

Also among LKB's marked for resending, an LKB may have a lockqueue state of `WAIT_CONVERT`. The `LQCONVERT` flag is set on these to indicate that when the lock is rebuilt on the new RSB master, the LKB should be considered granted, not converting, as the conversion request will be resent to the new master when recovery finishes.

If the recovery as a whole is due to nodes being added and not removed, the main recovery ends here. The Service Manager is notified through `kcl_start_done` that local recovery is complete. The last steps of cleanup and resending requests will occur when the finish callback from SM signals that all nodes have completed.

3.3.12.6 Clear defunct locks

The next step is `restbl_lkb_purge` which deletes locks held by departed nodes from all mastered RSB's. This results in releasing LKB's in addition to RSB's which subsequently have no locks remaining. The `purge_queue` function is applied to each of the lock queues of an RSB. It should be clear that this step only applies to master RSB's. Non-master RSB copies only maintain copies of locally held locks and no locks for other nodes that may have failed.

The only inter-node communication that may arise in this step is the sending of resource directory removal messages when RSB's are released. When sent during recovery, these messages go through the recovery-specific communication routines.

3.3.12.7 Update resource masters

The `restbl_rsb_update` function assigns new master nodes to RSB's that were mastered on departed nodes. All nodes go through their list of RSB copies (i.e. RSB's they are not master of) and for those that specify a departed node as master, a lookup request is sent to the resource directory. The first node to look up the new master ID becomes the master (just as lookups work in normal operation.) When other nodes look up the master of the same RSB, the new master node ID is returned. In this way, new masters are assigned and new master node ID's are updated in the non-master RSB copies.

The `rsb_master_lookup` function is called by `restbl_rsb_update` to find the master of a particular RSB. If the directory node for the RSB happens to be local, the lookup is immediate. If the directory node is remote, an asynchronous lookup message (GETMASTER) is sent to the directory node and the RSB is added to the lockspace's "recover list".

The recover list is used for different purposes during recovery; at this stage it contains RSB's for which directory lookups are outstanding. A temporary message ID is also generated that is both saved in the RSB and sent with the lookup message. The message ID allows the reply to be matched with the correct RSB without sending the entire resource name in the reply.

The RD lookup is sent as a recovery-specific message (so it is processed and not added to the requestqueue) and is handled on the directory node like an ordinary lookup. On the requesting node, `restbl_rsb_update_recv` is called to handle the reply. This function finds the particular RSB in the lockspace's recover list using the returned message ID, removes the RSB from the recover list and updates the RSB's master node ID. The `set_new_master` function is called to propagate the new master ID to the RSB's LKB's and sub-RSB's.

When all lookup requests have been sent, `restbl_rsb_update` sleeps until the recover list is empty (all replies received and RSB's updated and removed from the list.) Once again, a node failure before the completion of this step must abort the current recovery process.

3.3.12.8 Rebuild locks on new masters

Once new RSB masters are known, a node's local copies of the locks it holds (against the remastered RSB's) must be rebuilt on them. To do this rebuilding the `rebuild_rsbs_send` function goes through each root RSB that has a new remote master and sends all its LKB's (and sub-RSB's and their LKB's) to the new master node in a NEWLOCKS message. These non-master RSB copies are put on the lockspace's recover list when the LKB's are sent to the new master. A count is also kept in these RSB's of the number of its LKB's that have been sent. NEWLOCKS messages are asynchronous.

When a new master receives LKB's in a NEWLOCKS message (`rebuild_rsbs_recv`), it creates master LKB copies and adds them to its master RSB. It then returns to the sending node in a NEWLOCKIDS message a list of LKB ID's corresponding to the new LKB's that it created. In this message, each new LKB ID is preceded by the ID of the remote node's corresponding LKB ID. (As mentioned earlier, a node must know the remote ID of a corresponding LKB so it can be included in messages referring to the lock.)

When the sending node receives a NEWLOCKIDS message (`rebuild_rsbs_lkids_recv`) it extracts the pairs of LKB ID's from the data. The first ID is used to find its own LKB and the second is saved in this LKB as the new remote LKB ID. When a new LKB ID is received, the RSB's sent-LKB count is decremented. When this count becomes zero the RSB is removed from the recover list. The `rebuild_rsbs_send` function completes when the recover list is empty.

Two factors affect how `rebuild_rsbs_send` packs LKB's into messages for sending. For efficiency, as many consecutive LKB's destined for the same node are packed into one message as possible. So, LKB's for

multiple RSB's remastered to the same node may be sent in one message. This only occurs to the extent that sequential RSB's in the root list are mastered on the same new node (RSB's are not skipped over to find those that can be sent together.) Second, all the LKB's a node holds against an RSB may not fit in one message (this is especially likely for resource trees.)

For these reasons a message may become full at any point and the routines packing LKB's into messages must be able to start a new message to the same or a different node at any point.

After this step the main recovery is complete and the the Service Manager is notified via `kcl_start_done`.

3.3.12.9 Finish and cleanup

After all nodes complete the main recovery and notify the Service Manager, SM on each will call the lockspace's finish callback. Once again, the `gdlm_recoverd` thread does the processing for this callback based on the current state (`RECONFIG_DONE`) and output of `next_move` (`DO_FINISH`).

First, the `clear_finished_nodes` function removes node structures from the `ls_nodes_gone` list. In the case of cascading failures and restarted recovery procedures, there may be nodes from a more recent recovery event in the list as well. A node's `gone_event` indicates the recovery event corresponding to its departure. Using this and the most recent `finish_event` value in the lockspace, nodes with a `gone_event` value less than or equal to the last `finish_event` are removed.

Second, the `enable_locking` function releases the recovery's write lock on `in_recovery` allowing requests that blocked for recovery to continue. This is only permitted, however, if there has not been another stop callback for a new recovery since the finish callback was received.

Finally, the `finish_recovery` function gets requests going that were affected during recovery. The `resend_cluster_requests` function resends requests that were previously marked as needing to be resent. The `restbl_grant_after_purge` function grants lock requests that can now be granted because locks from failed nodes were purged. The `process_requestqueue` function processes requests that were received during recovery and saved on the requestqueue.

3.3.12.10 Event waiting

There are two types of event waiting used during recovery. The first is a kind of barrier in which all nodes wait for a specific status bit to be set on all other nodes before proceeding. The second type waits for a local function, provided as a parameter, to return non-zero. The test function is called repeatedly to check if the wait can complete. Both of these waiting methods must abort and return an error if the lockspace is stopped due to a node failure. When this happens, the current recovery process will be aborted so a new one can be started.

Two functions are used for the first barrier-style waiting method. The first, `gdlm_wait_status_all`, polls all nodes in the lockspace, waiting for a specified bit to be set (bit provided as a function parameter.) The second, `gdlm_wait_status_low`, also waits for a specified bit to be set, but only on the node in the lockspace with the lowest node ID.

To wait for a given status bit, `X`, to be set on all nodes, the lockspace member with the lowest node ID calls `gdlm_wait_status_all(X)` and other lockspace members call `gdlm_wait_status_low(X_ALL)`. When wait function returns on the low node, it sets status bit `X_ALL` which the other nodes then see. All nodes then continue. This approach results in $O(N)$ messages instead of $O(N^2)$ all-to-all messages.

The second, local, type of wait function is `gdlm_wait_function` for which a test function is provided as a parameter. This wait function returns to the caller only when the test function returns a non-zero value or the lockspace has been stopped for another recovery (in which case the wait function returns a non-zero error value.)

3.3.12.11 Functions: Request queue

```
void add_to_requestqueue(gd_ls_t *ls, int nodeid, char *request, int length)
```

Saves a message for processing later (when recovery finishes.) Allocates a `struct rq_entry` along with a buffer to hold the message and adds it to the lock space's requestqueue.

```
void process_requestqueue(gd_ls_t *ls)
```

Removes each saved message from the lock space's requestqueue and processes the message as usual in `process_cluster_request`. N.B. cluster requests usually processed by `gdlm_recvd` thread, but these are processed by `gdlm_recoverd`.

```
void purge_requestqueue(gd_ls_t *ls)
```

Looks through all saved messages in requestqueue and removes those related to the resource directory (lookups, replies, removals) and those from departed nodes.

3.3.12.12 Functions: Nodes list

```
int ls_nodes_reconfig(gd_ls_t *ls, gd_recover_t *gr, int *neg_out)
```

Removes departed nodes from the lock space's `ls_nodes` list based on the new member list saved in the `gd_recover_t` struct. The removed node structs are moved to `ls_nodes_gone`. Adds entries to `ls_nodes` for new nodes. Returns the number of departed nodes through the last parameter.

```
int ls_nodes_init(gd_ls_t *ls, gd_recover_t *gr)
```

Adds an entry to the lock space's `ls_nodes` list for each of the lock space members saved in the `gd_recover_t` struct.

```
int nodes_reconfig_wait(gd_ls_t *ls)
```

Waits for all nodes in the lock space to complete `ls_nodes_reconfig` (or the equivalent `ls_nodes_init` for new members.)

```
void clear_finished_nodes(gd_ls_t *ls, int finish_event)
```

Called to finish a particular recovery event. The `finish_event` event ID corresponds to the recovery event to which the finish callback refers. The function removes node structs from `ls_nodes_gone` for departed nodes which left the lock space no later than the given event. The event ID is recorded for a node when it departs by `ls_nodes_reconfig`.

3.3.12.13 Functions: Resource directory

```
int resdir_rebuild_local(gd_ls_t *ls)
```

Rebuilds resource directory entries that map into the calling node's directory segment. Sequentially collects resource names for existing RSB's from all nodes in the cluster via a RECOVERNAMES request. Adds a new RD entry for each name returned with a master node ID set to the replying node.

```
int resdir_rebuild_send(gd_ls_t *ls, char *inbuf, int inlen, char *outbuf, int outlen, uint32 nodeid)
```

Processes a RECOVERNAMES request from a node in `resdir_rebuild_local`. Goes through all locally mastered resources and returns the name of each that maps to the requesting node. Multiple request/reply messages are usually needed to transfer all the data, so starting resource names are included in requests.

```
int resdir_rebuild_wait(gd_ls_t *ls)
```

Waits for all nodes in the lock space to complete `resdir_rebuild_local`. Uses the lock space status bits `RESDIR_VALID` and `RESDIR_ALL_VALID`.

```
void gdlm_resmov_in(gd_resmov_t *rm, char *buf)
```

Copies a resource name record from a RD rebuild message, handling byte swapping.

```
void resdir_clear(gd_ls_t *ls)
```

Removes and frees all local RD entries from the lock space's RD hash table.

3.3.12.14 Functions: Lock queue

```
void lockqueue_lkb_mark(gd_ls_t *ls)
```

In-progress LKB requests waiting on the lock space's lockqueue are flagged as needing to be resent if the requests were sent to a departed node or if the request was related to a resource directory operation.

```
void resend_cluster_requests(gd_ls_t *ls)
```

Resend LKB requests waiting on the lock space's lockqueue that were flagged by `lockqueue_lkb_mark`. If the LKB has been remastered to the local node, it no longer needs to be sent anywhere, but simply processed locally.

```
void process_remastered_lkb(gd_lkb_t *lkb, int state)
```

Called by `resend_cluster_request` to process an LKB request locally. Before recovery the request was remote and waiting for a remote reply but the recovery process resulted in the request now being local.

3.3.12.15 Functions: Purging locks

```
int restbl_lkb_purge(gd_ls_t *ls)
```

Remove and free all LKB's that are held by departed nodes.

```
int purge_queue(gd_ls_t *ls, osi_list_t *queue)
```

Same as previous, but acts on an arbitrary queue (granted, convert, wait) of an arbitrary RSB.

```
void restbl_grant_after_purge(gd_ls_t *ls)
```

Go through all RSB's and grant locks on convert and wait queues that can be granted. There may be grantable locks due to locks from departed nodes being purged.

3.3.12.16 Functions: Updating resource masters

```
int restbl_rsb_update(gd_ls_t *ls)
```

Goes through local root resources and for each RSB with a master that has departed and looks up the new master node ID from the resource directory. All RD requests are made individually and asynchronously. The requesting node may become the master itself as the RD assigns mastery to the first node to look up the new master. RSB's being updated are kept in the recover list while awaiting a reply.

`int restbl_rsb_update_recv(gd_ls_t *ls, uint32 nodeid, char *buf, int length, int msgid)`
 Processes the reply for a RD lookup made from `restbl_rsb_update`. Removes the RSB corresponding to the reply from the lock space's recover list and sets the new master node ID in the RSB. Calls `set_new_master` to propagate the new master ID to LKB's on the RSB.

`int rsb_master_lookup(gd_res_t *rsb, gd_rcom_t *rc)`
 Finds the master node ID for the given RSB from the resource directory. Does a local lookup if the resource name maps to the local RD segment, or sends a lookup request to a remote node, keeping the RSB in the recover list while awaiting a reply.

`void set_new_master(gd_res_t *rsb)`
 Propagates the new master node ID, already set in the given root RSB, to the LKB's, sub-RSB's and their LKB's.

`void set_rsb_lvb(gd_res_t *rsb)`
 Called by `set_new_master` when a local RSB copy has become the real master RSB in the lock space. This function goes through all LKB's attempting to find an up to date copy of the LVB to assign to the RSB. If none are found, the LVB on a remastered RSB is zeroed.

`void set_master_lkbs(gd_res_t *rsb)`
 Called by `set_new_master` to propagate the RSB's node ID to LKB's on the three queues (granted, convert, waiting.)

`void set_lock_master(osi_list_t *queue, int nodeid)`
 Called by `set_master_lkbs` to set the node ID on all LKB's in a list.

3.3.12.17 Functions: Recover list

`int recover_list_empty(gd_ls_t *ls)`
 Test if recovery list is empty.

`int recover_list_count(gd_ls_t *ls)`
 Returns the number of entries in the recovery list.

`void recover_list_add(gd_res_t *rsb)`
 Adds the given RSB to the recover list if it's not already there.

`void recover_list_del(gd_res_t *rsb)`
 Removes the given RSB from the recover list.

`gd_res_t *recover_list_find(gd_ls_t *ls, int msgid)`
 Returns an RSB from the recover list with the specified message ID.

`void recover_list_clear(gd_ls_t *ls)`
 Removes all RSB's from the recover list.

3.3.12.18 Functions: Rebuilding locks

`int rebuild_rsbs_send(gd_ls_t *ls)`
 Sends LKB's for a newly remastered RSB to the new master node (in a NEWLOCKS message.) This also includes sub-RSB's their LKB's.

```

int rebuild_rsbs_recv(gd_ls_t *ls, int nodeid, char *buf, int len)
Receives new locks for a recently mastered RSB from the holder of the locks. Creates master LKB copies
(MSTCPY) for the locks on the RSB queues. Returns the local lock ID's of the newly created MSTCPY
LKB's to the sending node.

int rebuild_rsbs_lkids_recv(gd_ls_t *ls, int nodeid, char *buf, int len)
Receives new remote lock ID's from a master node to which locks were sent in rebuild_rsbs_send.

gd_res_t *deserialise_rsb(gd_ls_t *ls, int nodeid, gd_res_t *rootrsb, char *buf, int *ptr)

int deserialise_lkb(gd_ls_t *ls, int rem_nodeid, gd_res_t *rootrsb,
char *buf, int *ptr, char *outbuf, int *outoffp)

gd_lkb_t *find_by_remlkid(gd_res_t *rootrsb, int nodeid, int remid)

gd_lkb_t *search_remlkid(osi_list_t *statequeue, int nodeid, int remid)

gd_res_t *find_by_remasterid(gd_ls_t *ls, int remasterid, gd_res_t *rootrsb)

void fill_rcom_buffer(gd_ls_t *ls, rcom_fill_t *fill, uint32 *nodeid)

gd_res_t *next_remastered_rsb(gd_ls_t *ls, gd_res_t *rsb)

int pack_rsb_tree(gd_ls_t *ls, gd_res_t *rsb, rcom_fill_t *fill)

int pack_rsb_tree_remaining(gd_ls_t *ls, gd_res_t *rsb, rcom_fill_t *fill)

int pack_subrsbs(gd_res_t *rsb, gd_res_t *in_subrsb, rcom_fill_t *fill)

int pack_one_subrsb(gd_res_t *rsb, gd_res_t *subrsb, rcom_fill_t *fill)

int pack_lkb_remaining(gd_res_t *r, rcom_fill_t *fill)

int pack_lkb_queues(gd_res_t *r, rcom_fill_t *fill)

int pack_lkb_queue(gd_res_t *r, osi_list_t *queue, rcom_fill_t *fill)

int pack_one_lkb(gd_res_t *r, gd_lkb_t *lkb, rcom_fill_t *fill)

void serialise_rsb(gd_res_t *rsb, char *buf, int *offp)

int lkbs_to_remaster(gd_res_t *r)

gd_res_t *next_subrsb(gd_res_t *subrsb)

int rsb_length(gd_res_t *rsb)

void serialise_lkb(gd_lkb_t *lkb, char *buf, int *offp)

int lkb_length(gd_lkb_t *lkb)

void get_bytes(char *bytes, int *len, char *buf, int *offp)
char get_char(char *buf, int *offp)
uint64 get_int64(char *buf, int *offp)

```

```

int get_int(char *buf, int *offp)
void put_char(char x, char *buf, int *offp)
void put_bytes(char *x, int len, char *buf, int *offp)
void put_int64(uint64 x, char *buf, int *offp)
void put_int(int x, char *buf, int *offp)

void rebuild_freemem(gd_ls_t *ls)

rebuild_node_t *find_rebuild_root(gd_ls_t *ls, int nodeid)

void have_new_lkid(gd_lkb_t *lkb)

void need_new_lkid(gd_res_t *rsb)

void expect_new_lkids(gd_res_t *rsb)

```

3.3.12.19 Functions: Event waiting

```

int gdlm_recovery_stopped(gd_ls_t *ls)

void gdlm_wait_timer_fn(void *data)

int gdlm_wait_function(gd_ls_t *ls, int (*testfn)(gd_ls_t *ls))

int gdlm_wait_status_all(gd_ls_t *ls, unsigned int wait_status)

int gdlm_wait_status_low(gd_ls_t *ls, unsigned int wait_status)

```

3.3.13 Communications

The "lowcomms" communications layer is responsible for maintaining network connections between nodes and sending and receiving messages. It is common to all lockspaces: one set of TCP socket connections and one set of threads. This requires a lockspace identifier to be included in the header of all messages. The lowcomms subsystem uses two kernel threads, `gdlm_sendd` and `gdlm_recvd`.

The send and receive API's use the same node ID's used in the rest of GDLM. IP addresses are looked up for new node ID's from the cluster manager (which originally provided the node ID's to GDLM in the "start" callback's member list.) The default TCP port for GDLM sockets is 21064 and can be changed by setting `gdlm/tcp-port` in the `gdlm.ccs` configuration file.

3.3.13.1 Sending requests

Messages are sent by the `gdlm_sendd` thread; this prevents threads that initiate messages from blocking and promotes merging of messages. A `struct writequeue_entry` is a unit of data that will be sent to another node. A list of these entries is kept per remote node connection.

When a message is to be sent, the function `lowcomms_get_buffer` is first called. It is given the destination node ID and the length of the message. This function returns a pointer to a data buffer into which the

message can be written. When the message has been filled in, the function `midcomms_send_buffer` is called which transforms data into network byte order and calls `lowcomms_commit_buffer` to queue the message for sending.

If `lowcomms_get_buffer` finds no existing writequeue entries for the given node ID's connection, a new entry structure is allocated along with a single page for the data. The page address is returned to the caller as the buffer into which the message can be copied. A pointer to the entry is also returned as the message handle for committing it.

If `lowcomms_get_buffer` finds an existing writequeue entry to the given node and if the requested length will fit into the remaining space in the entry's page, then the current offset into the existing page is returned to the caller as the message buffer. In this way messages can be combined, reducing network transmissions especially when many small messages are sent.

3.3.13.2 Receiving requests

Messages are received by the `gdlm_recvd` thread. This thread also does the GDLM processing for the messages it receives. When data becomes available on the socket for a remote connection, the connection structure is added to a global list of connections with data to receive. A page is allocated per connection into which a message is copied. The message is passed to `midcomms_process_incoming_buffer` which transforms data into host byte order and then calls `process_cluster_request` for ordinary messages or `process_recovery_comm` for recovery-specific requests.

3.3.13.3 Recovery requests

The function `rcom_send_message` is used by recovery routines to send messages and data to other nodes. If data is expected in reply, the function is synchronous, waiting for the reply data. The routine will return an error if the lock space is stopped while waiting for a reply.

3.3.13.4 Functions: low-level communications

```
void lowcomms_data_ready(struct sock *sk, int count_unused)
```

```
void lowcomms_write_space(struct sock *sk)
```

```
void lowcomms_connect_sock(struct connection *con)
```

```
void lowcomms_state_change(struct sock *sk)
```

```
int add_sock(struct socket *sock, struct connection *con)
```

```
void close_connection(struct connection *con)
```

```
int receive_from_sock(struct connection *con)
```

```
int accept_from_sock(struct connection *con)
```

```
int connect_to_sock(struct connection *con)
```

```
int create_listen_sock()

struct writequeue_entry *new_writequeue_entry(struct connection *con, int allocation)

struct writequeue_entry *lowcomms_get_buffer(int nodeid, int len, int allocation , char **ppc)

void lowcomms_commit_buffer(struct writequeue_entry *e)

void free_entry(struct writequeue_entry *e)

int send_to_sock(struct connection *con)

int lowcomms_close(int nodeid)

int lowcomms_send_message(int nodeid, char *buf, int len, int allocation)

void process_sockets()

void process_output_queue()

void process_state_queue()

void clean_writequeues()

int read_list_empty()

int gdlm_recvd(void *data)

int write_and_state_lists_empty()

int gdlm_sendd(void *data)

void daemons_stop()

int daemons_start()

int lowcomms_max_buffer_size()

int lowcomms_start()
```

3.3.13.5 Functions: mid-level communications

```
void host_to_network(void *msg)

void network_to_host(void *msg)

void copy_from_cb(void *dst, const void *base, unsigned offset, unsigned len, unsigned limit)

int midcomms_process_incoming_buffer(int nodeid, const void *base,
unsigned offset, unsigned len, unsigned limit)

void midcomms_send_buffer(struct gd_req_header *msg, struct writequeue_entry *e)
```

```
int midcomms_send_message(uint32 nodeid, struct gd_req_header *msg, int allocation)
```

3.3.13.6 Functions: recovery communications

```
int rcom_response(gd_ls_t *ls)
int rcom_send_message(gd_ls_t *ls, uint32 nodeid, int type, gd_rcom_t *rc, int need_reply)
void rcom_process_message(gd_ls_t *ls, uint32 nodeid, gd_rcom_t *rc)
void process_reply_sync(gd_ls_t *ls, uint32 nodeid, gd_rcom_t *reply)
void process_reply_async(gd_ls_t *ls, uint32 nodeid, gd_rcom_t *reply)
void rcom_process_reply(gd_ls_t *ls, uint32 nodeid, gd_rcom_t *reply)
void process_recovery_comm(uint32 nodeid, struct gd_req_header *header)
```

3.3.14 Future Work

GDLM can be enhanced with the following features:

- *Dynamic lock remastering.* The master node of a resource is changed while the lock space is operating. The new master is chosen due to its high volume of lock operations against the resource relative to the current master node.
- *Lock directory weights.* A static "weight" value is assigned to certain nodes that causes a larger segment of the lock directory to be stored on them or none at all.
- *Dedicated lock servers.* One or more nodes can be designated as resource masters in a lock space. This replaces the common policy where the first node to lock a resource becomes its master.

3.4 LOCK_DLM

TODO: add details of LOCK_DLM design and implementation.

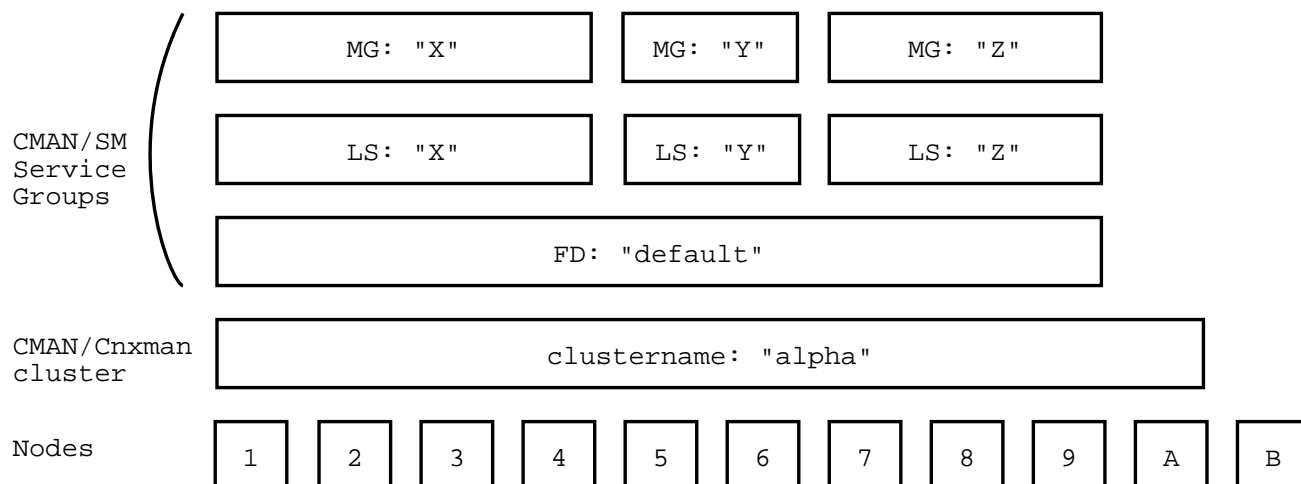
3.5 Fence

TODO: add details of Fencing system design and implementation.

3.6 Appendix A: Practical Usage

A summary of all the practical steps required to set up and use the software described in this document.

3.7 Appendix B: Service Group Examples



Three GFS file systems in use with names X, Y and Z.

```

gfs_mkfs -p lock_dlm -t alpha:X -j 12 /dev/volume1
gfs_mkfs -p lock_dlm -t alpha:Y -j 12 /dev/volume2
gfs_mkfs -p lock_dlm -t alpha:Z -j 12 /dev/volume3
  
```

```

Nodes 1-A:      ccsd -d /dev/alpha_cca
Nodes 1-A:      cman_tool join
  
```

```

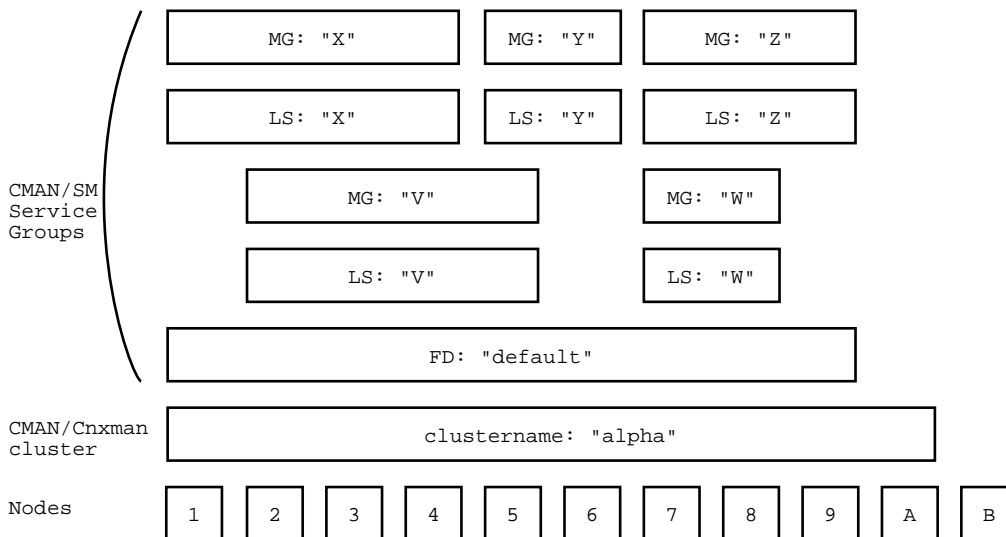
Nodes 1,2,3,4:  mount -t gfs /dev/volume1 /dir
Nodes 5,6:      mount -t gfs /dev/volume2 /dir
Nodes 7,8,9:    mount -t gfs /dev/volume3 /dir
  
```

Example: node 9 fails

```

Nodes 1,2,3,4,5,6,7,8,A: Cnxman detects failure
Nodes 1,2,3,4,5,6,7,8,A: Cnxman transition removes node 9 from cluster
Nodes 1,2,3,4,5,6,7,8,A: Cnxman tells SM of new cluster membership
Nodes 1,2,3,4,5,6,7,8,A: SM stops service group FD-"default"
Nodes 7,8:        SM stops service group LS-"Z"
Nodes 7,8:        SM stops service group MG-"Z"
Nodes 1,2,3,4,5,6,7,8,A: SM starts FD-"default" and 9 is fenced by the
member of FD-"default" with the lowest ID (1)
Nodes 7,8:        SM starts LS-"Z" and GDLM recovers lock space "Z"
Nodes 7,8:        SM starts MG-"Z" and GFS recovers journal of 9
  
```

Figure 3.15: Service Groups Example 1



Five GFS file systems in use with names V, W, X, Y and Z.

```

gfs_mkfs -p lock_dlm -t alpha:V -j 12 /dev/volume1
gfs_mkfs -p lock_dlm -t alpha:W -j 12 /dev/volume2
gfs_mkfs -p lock_dlm -t alpha:X -j 12 /dev/volume3
gfs_mkfs -p lock_dlm -t alpha:Y -j 12 /dev/volume4
gfs_mkfs -p lock_dlm -t alpha:Z -j 12 /dev/volume5
  
```

```

Nodes 1-A:      ccsd -d /dev/alpha_cca
Nodes 1-A:      cman_tool join
  
```

```

Nodes 2,3,4,5: mount -t gfs /dev/volume1 /dir1
Nodes 7,8:      mount -t gfs /dev/volume2 /dir2
Nodes 1,2,3,4: mount -t gfs /dev/volume3 /dir3
Nodes 5,6:      mount -t gfs /dev/volume4 /dir4
Nodes 7,8,9:   mount -t gfs /dev/volume5 /dir5
  
```

Example: node 9 fails

```

Nodes 1,2,3,4,5,6,7,8,A: Cnxman detects failure
Nodes 1,2,3,4,5,6,7,8,A: Cnxman transition removes node 9 from cluster
Nodes 1,2,3,4,5,6,7,8,A: Cnxman tells SM of new cluster membership
Nodes 1,2,3,4,5,6,7,8,A: SM stops service group FD-"default"
Nodes 7,8:              SM stops service group LS-"Z"
Nodes 7,8:              SM stops service group MG-"Z"
Nodes 1,2,3,4,5,6,7,8,A: SM starts FD-"default" and 9 is fenced by the
member of FD-"default" with the lowest ID (1)
Nodes 7,8:              SM starts LS-"Z" and GDLM recovers lock space "Z"
Nodes 7,8:              SM starts MG-"Z" and GFS recovers journal of 9
  
```

Example: node 5 fails

```

Nodes 1,2,3,4,6,7,8,9,A: Cnxman detects failure
Nodes 1,2,3,4,6,7,8,9,A: Cnxman transition removes node 5 from cluster
Nodes 1,2,3,4,6,7,8,9,A: Cnxman tells SM of new cluster membership
Nodes 1,2,3,4,6,7,8,9,A: SM stops service group FD-"default"
Nodes 2,3,4:            SM stops service group LS-"V"
Node 6:                 SM stops service group LS-"Y"
Nodes 2,3,4:            SM stops service group MG-"V"
Node 6:                 SM stops service group MG-"Y"
Nodes 1,2,3,4,6,7,8,9,A: SM starts FD-"default" and 5 is fenced by the
member of FD-"default" with the lowest ID (1)
Nodes 2,3,4:            SM starts LS-"V" and GDLM recovers lock space "V"
Node 6:                 SM starts LS-"Y" and GDLM recovers lock space "Y"
Nodes 2,3,4:            SM starts MG-"V" and GFS recovers journal of 5
Node 6:                 SM starts MG-"Y" and GFS recovers journal of 5
  
```

Figure 3.16: Service Groups Example 2