Proposal for IDM Lock Manager

Revision 1.0 January 22, 2020

# 

[Abstract](#_uh5c1co77ju)

[1 Introduction](#_lkfpymkwcq69)

[2 Lock manager analysis](#_cq7orahbxbkm)

[2.1 Initialization](#_q9om6x5vu9h0)

[2.2 Lockspace](#_3wb2iv9a9flu)

[2.3 Membership management](#_yatdmhmbbe1p)

[2.4 Lock algorithm](#_7eetaxp56sx1)

[2.5 Timeout and fence](#_ssg7xafkpxtf)

[2.7 Interfaces](#_h2a3v9ssi1nl)

[3 Requirement for IDM lock manager](#_ez5axlk8hmaa)

[3.1 Initialization](#_yzqw2wy6swfu)

[3.2 Lockspace](#_r0rjqze50j8f)

[3.3 Membership management](#_h9bmumk5jxl6)

[3.4 Lock algorithm](#_ilfghog6v0t4)

[3.5 Timeout and fence](#_c4f19jve1oth)

[3.6 Requirement summary](#_j7oliees1i93)

[3.7 Interfaces](#_22elrtm44oz3)

[4 Compute node software stack design](#_dm0vdss8ifmy)

[4.1 Initialization](#_hktkp5dloj08)

[4.2 Data structure](#_lfok3ckyhblp)

[4.2.1 Data structure in lvmlockd](#_ueem5g2hl3zn)

[4.2.2 Data structure in IDM lock manager](#_ldwi1xkq4ijo)

[4.3 Membership management](#_et1t6sgakoec)

[4.4 Lock algorithm](#_yblt2nluna8p)

[4.4.1 Principles for locking majority](#_2yhz4mqia6cx)

[4.4.2 Acquisition majority locking](#_qr2f4oimu4zl)

[4.4.3 Lock renewal](#_82c31q9y8pcp)

[4.4.4 Lock mode converting](#_p96zra8kudd5)

[4.4.5 Lock releasing](#_wh65dlu76n43)

[4.4.6 LVB](#_78329ghew2kw)

[4.4.7 State machine for drive list](#_7g0r8upes0ly)

[4.5 Timeout and fence](#_9ypfvm1043gg)

[4.6 IDM thread vs IDM daemon](#_uz4hzz658er4)

[4.7 APIs](#_4nmbjhngc6kp)

[4.7.1 lvmlockd lock manager interface](#_5ardnlhxqctx)

[4.7.2 IDM wrapper APIs](#_6i77n2vbsyee)

[4.7.3 IDM RAID APIs](#_3uc8169o6le8)

[4.7.4 IDM SCSI wrapper APIs](#_ral31md4k0a)

[5 Development and test plan](#_8mif0xq57qdh)

[5.1 Stage 1: Parallel development](#_4u9g09kvtrcq)

[5.2 Stage 2: Integration & test for IDM lock manager](#_eezldak6zb4k)

[5.3 Stage 3: Integration & testing for the whole stack](#_kj8nhjb8keek)

[5.4 Stage 4: Failure inducement testing](#_nnxdjfvjuwvt)

[5.5 Timeline](#_wpgcvbws4yt5)

[6 Development and test plan (if IDM firmware is delayed)](#_tkij2jth63pe)

[6.1 Stage 1: IDM lock manager development with pthread mutex emulation](#_j3tdl9o35u3y)

[6.2 Stage 2: Whole stack enabling with pthread mutex backend](#_1y8nn3nm8yfq)

[6.3 Stage 3: IDM SCSI wrapper APIs development](#_9c0n697odvnd)

[6.4 Stage 4: Integration IDM lock manager with IDM SCSI wrapper APIs](#_x1atkcgzu0lx)

[6.5 Stage 5: Testing the whole stack with IDM SCSI wrapper APIs](#_q2h1226o5i0u)

[6.6 Stage 6: Failure inducement testing](#_t9hluk4o0bkw)

[6.7 Conclusion](#_q8vtsyey2lym)

[Appendix A: Workload estimation](#_hqksiw0ooso)

[A.1 Development workload estimation](#_3omr4o9rvhpn)

[A.2 Test workload estimation](#_cqjn31jo3nru)

[Appendix B: Open issues](#_ubdej2thbd52)

[B.1 Adoption orphan lock](#_ifxw4m2s1im)

[B.2 IDM state machine](#_qgrrqexq5b8i)

[B.3 SCSI command](#_tv44syb1zf7a)

# 

# 

# 

# Abstract

This is a design document for developing the lock manager based on the Seagate’s in-drive mutex (IDM) for the Linux logical volume manager (LVM) . In order to clearly describe the thoughts for the design, this document is divided into the following parts:

* Introduction for the background information
* Dive into existed lock managers for analysis
* Give out the requirement for IDM lock manager
* Design with the top to bottom approach to define the software architecture and the contained components, record the observed issues and summary candidate solutions
* Development and Test Plan for guiding implementation and defines the brief milestones to achieve good software quality.

# 1 Introduction

Let’s quickly go through the software architecture for the whole software stack; this can help us to have a global picture for what we will work on and what’s the target for development with Seagate IDM.

The diagram 1 shows the architecture of a typical cluster based around iSCSI drives:



Diagram 1: Software architecture

To use drives in more general fashion, LVM is introduced to create physical volumes, volume groups and logical volumes, thus it can abstract the hardware features from drives. Since the LVM is the shared resource crossing hosts, LVM needs to use the locking mechanism to protect the LVM, thus this allows the hosts to share logical volumes or exclusive access according to the different usage cases.

For this document, our target is to design a locking scheme based on Seagate IDM and integrate into lvmlockd for LVM, thus we can see the updated architecture based on the mutex resident in drives:



Diagram 2: The updated software architecture

Simply to say, to support the Seagate IDM as a LVM lock manager, we need to implement drive firmware based locking scheme, integrate this locking scheme into lvmlockd, and facilitate the IDM in RAID for locking related operations.

# 2 Lock manager analysis

This chapter tries to deliver lock manager analysis for two outputs. The first output is the required functionalities for lock manager, this can make us clear what the lock manager it is; the second output result is the interfaces between lvmlockd and lock managers, this eventually can allow us to understand how to use lock managers.

For some critical functionalities in the lock manager, we will dive into details for two factors: data structure and workflow; this is helpful for a comprehensive understanding of lock manager’s internal mechanism and the gathered knowledge can guide us for IDM lock manager design.

## 2.1 Initialization

There are three initial phases before we can use a lock. These three phases are for different purposes and it’s relevant to different entities initialization.

The first initial phase is for launching service daemons, at this stage the lvmlockd daemon and lock manager daemons will be launched. The main accomplishment in daemon initialization is to get ready for connections, thus lvmlockd is ready to accept request from LVM2 tools based on local socket /run/lvm/lvmlockd.socket; and lock managers are ready to communicate with lvmlockd and their connection channel is either based on socket /run/sanlock for sanlock daemon or based on device node /dev/misc/dlm-control for DLM kernel module. Comparing against the sanlock lock manager, the main difference for DLM lock manager is DLM has established cluster membership at this stage, which we will leave related discussion in section ‘cluster membership’ for discussion.

The second initial phase is for creating a volume group. This phase attaches to LVM tool vgcreate for creating a volume group, vgcreate notifies lvmlockd and lock managers can take this occasion for preparation locking; note that at this time point locks will not be acquired due VG is inactive. DLM simply generates volume group argument by combining DLM version and cluster name with format 1.0.0:cluster\_name; sanlock is much more complex in this phase, since sanlock relies on this phase to setup a shared storage, the shared storage is a hidden logical volume lvmlock; it is created by vgcreate in LVM core layer but it is handed over to lvmlockd, lvmlockd formats this volume with predefined layout. The usage convention is that the first block in the logical volume lvmlock is reserved for host ID leasing and lvmlockd needs to define offsets for lock leasing in the following blocks; the sanlock daemon directly uses the predefined layout and it concentrates on leasing algorithm.

The third initial phase is for activating a volume group; this phase is for users to start using a volume group and its logical volumes, e.g. this phase can be launched by vgchange command. During the initialization, lvmlockd creates a lockspace context per volume group wise, thus every volume group has its individual locking context, both DLM and sanlock create lockspace in this phase. The core thing in this step is to create a context to reflect producer-consumer relationship between an application who consumes locks and locking space which provides locks.

## 2.2 Lockspace

As naming ‘space’ indicates, lockspace is a context based implementation in lock managers. To understand the context, we need to answer the following questions:

* Context life cycle;
* What objects are managed by lockspace?
* What’s the purpose for the context management?

Since a lockspace is created when a volume group is to be activated and it will be deleted when deactivate this volume group, we can simply consider a lockspace’s life cycle is the same with volume group’s activation.

For managed objects in lockspace, let’s review the data structures in lvmlockd and lock manager separately. lvmlockd’s lockspace data structure is defined as below. lvmlockd maintains the connection within context for volume group, host ID, locks (resources is a synonym for locks) and incoming actions on locks.

struct lockspace {

struct list\_head list; /\* lockspaces \*/

char name[MAX\_NAME+1];

char **vg\_name**[MAX\_NAME+1];

char vg\_uuid[64];

char vg\_args[MAX\_ARGS+1]; /\* lock manager specific args \*/

char vg\_sysid[MAX\_NAME+1];

int8\_t lm\_type; /\* lock manager: LM\_DLM, LM\_SANLOCK \*/

void \*lm\_data;

uint64\_t **host\_id**;

uint64\_t free\_lock\_offset; /\* for sanlock, start search for free lock here \*/

int free\_lock\_sector\_size; /\* for sanlock \*/

int free\_lock\_align\_size; /\* for sanlock \*/

uint32\_t start\_client\_id; /\* client\_id that started the lockspace \*/

pthread\_t thread; /\* makes synchronous lock requests \*/

pthread\_cond\_t cond;

pthread\_mutex\_t mutex;

unsigned int create\_fail : 1;

unsigned int create\_done : 1;

unsigned int thread\_work : 1;

unsigned int thread\_stop : 1;

unsigned int thread\_done : 1;

unsigned int sanlock\_gl\_enabled: 1;

unsigned int sanlock\_gl\_dup: 1;

unsigned int free\_vg: 1;

unsigned int kill\_vg: 1;

unsigned int drop\_vg: 1;

struct list\_head **actions**; /\* new client actions \*/

struct list\_head **resources**; /\* resource/lock state for gl/vg/lv \*/

};

lvmlockd is the central place to maintain the relationship for objects, the lock manager is not necessary to maintain the duplicate info with lvmlockd. So the lock manager, like the sanlock daemon uses the data structure space for membership management, this data structure includes host ID allocation, host renewal recording and watchdog handler.

struct space {

struct list\_head list;

char space\_name[NAME\_ID\_SIZE];

uint32\_t space\_id; /\* used to refer to this space instance in log messages \*/

uint64\_t host\_id;

uint64\_t host\_generation;

struct sync\_disk host\_id\_disk;

uint32\_t io\_timeout;

uint32\_t set\_bitmap\_seconds;

uint32\_t flags; /\* SP\_ \*/

uint32\_t used\_retries;

uint32\_t renewal\_read\_extend\_sec; /\* defaults to io\_timeout \*/

uint32\_t rindex\_op;

int sector\_size;

int align\_size;

int max\_hosts;

int renew\_fail;

int space\_dead;

int killing\_pids;

int external\_remove;

int thread\_stop;

int wd\_fd;

int event\_fds[MAX\_EVENT\_FDS];

struct sanlk\_host\_event host\_event;

uint64\_t set\_event\_time;

pthread\_t thread;

pthread\_mutex\_t mutex; /\* protects lease\_status, thread\_stop \*/

struct lease\_status lease\_status;

struct host\_status host\_status[DEFAULT\_MAX\_HOSTS];

struct renewal\_history \*renewal\_history;

int renewal\_history\_size;

int renewal\_history\_next;

int renewal\_history\_prev;

};

After reviewing these data structures, we can get a conclusion that the lockspace in lvmlockd is used to maintain relationship between hosts and locks for volume group wise. In lock manager, it creates a lockspace data structure, but this structure is simply used for membership maintenance. And the lock manager needs to know who the lock’s owner is, so it adds locks token into client’s token list.

Lockspace can work as another role is namespace. E.g. if the system has two mounted file systems, the lock manager can create two lockspaces, every lockspace is dedicated to serve one filesystem; thus if there have the file nodes with the same file name under these two file systems, lock manager can distinguish the file’s lock since they are under different lockspaces. Namespace would be mandatory for a locking scheme used by the distributed file system, but it is not necessary for LVM project.

## 2.3 Membership management

Within a cluster, there are multiple nodes to connect with each other; and there are multiple applications resident in one node that use locking schemes to share resources for different usages. We can simply consider hosts and applications are entities in a cluster, membership management maintains valid execution entities which can be granted locks.

This section tries to answer what’s membership management with the questions:

* What’s the granularity for membership maintenance?
* How to validate membership?
* How to handle a failure membership?

At first glance, both DLM and sanlock take host as a member unit. But they vary if we compare their underlying mechanisms.

DLM relies on external cluster management module (e.g. corosync) to establish a whole picture for membership in cluster. So let’s see what’s the purpose for corosync; corosync is a cluster engine running on every node within a cluster, it complies with closed process group (CPG) API for the closed process group messaging semantics. CPG’s basic idea is: once a node is joined, a join message is sent to every node in the cluster; when the node leaves the cluster, either voluntarily, or as a result of failure, a leave message is sent to every remaining node. DLM has a daemon program dlm\_controld which translates the cluster membership from corosync’s CPG to DLM kernel module, finally DLM kernel module is the place to manage membership for a host.

Sanlock is quite different from DLM from the perspective of storing media for the membership management, this yields different membership granularity. DLM uses the kernel module for membership maintenance, so it uses a single copy for membership management and all applications (and all volume groups) on a host can share this copy; sanlock uses a *shared* storage media to manage membership, here ‘shared’ doesn’t mean the storage is shared by all applications or by all VGs, the shared storage is only shared for a specific usage within the same cluster; e.g, volume group 1 (VG1) has its logical volume VG1/lvmlock to maintain the membership, volume group 2 (VG2) has another logical volume VG2/lvmlock for the same purpose; as a result, even though these two VGs are deployed in the same cluster, but every VG has its own storage for membership management.

In short DLM maintains membership in a single copy on a host, sanlock maintains multiple copies for membership, every copy is only used for a dedicated lockspace.

Lock manager expects hosts to operate in a cluster with two behaviours:

Firstly, every host should have a unique identity, e.g. sanlock requires every host to set a unique host ID within the range [1..2000], it uses delta lease algorithm to ensure an ID can be allocated to only one host and if any later host registers duplicate ID will return failure.

Secondly, every host has responsibility to prove its liveness by renewing its membership periodically, e.g. in sanlock daemon, every lockspace has a thread and this thread renews delta leases for every 20 seconds by writing a new timestamp into the delta lease sector; this keeps other hosts posted in the same cluster to declaim that “I am alive so that I am valid for my acquired locks”.

For some reasons, a host might fail to renew itself, the lock manager needs to handle these kinds of failures. This will be deferred to discuss in section “2.5 Timeout and fence”.

## 2.4 Lock algorithm

Let’s see what lock model is implemented in lock managers. Lock Manager shall support two locking modes: 1) Exclusive (EX), 2) Protected Read (PR); lvmlockd uses EX mode and PR mode for exclusive and shareable locking.

Lock manager allows to convert lock from one mode to another: it allows to convert lock from exclusive mode to shareable mode, and when the lock is acquired with shareable mode, if the lock has only one owner, it can be converted to exclusive mode. Lock manager needs to check if convert command’s initiator is the lock’s owner, otherwise, the mode conversion is not permitted. Below code is to check if a client is valid owner for a lock in sanlock daemon:

for (i = 0; i < cl->tokens\_slots; i++) {

token = cl->tokens[i];

if (!token)

continue;

if (memcmp(token->r.lockspace\_name, res.lockspace\_name,

NAME\_ID\_SIZE))

continue;

if (memcmp(token->r.name, res.name, NAME\_ID\_SIZE))

continue;

found = 1;

break;

}

When an application tries to acquire a lock, it can specify the operation is block mode or not. If the operation is block mode, the lock request might be granted immediately, but it’s possible to wait for a long time if the lock has been leased to another host; so the waiting time varies. For non-block mode, the lock function returns without blocking, whether the application has acquired the lock successfully or not. Below code is the related usage in sanlock:

/\*

\* Don't block waiting for a failed lease to expire since it causes

\* sanlock\_acquire to block for a long time, which would prevent this

\* thread from processing other lock requests.

\*/

flags |= SANLK\_ACQUIRE\_OWNER\_NOWAIT;

memset(&opt, 0, sizeof(opt));

sprintf(opt.owner\_name, "%s", "lvmlockd");

rv = sanlock\_acquire(lms->sock, -1, flags, 1, &rs, &opt);

Lock Value Block (LVB) is a data block is associated with one lock; DLM supports max to 32 bytes char array and sanlock supports max to 4096 bytes for LVB separately. In lvmlockd, LVB has been applied to indicate the global or VG metadata has been changed, though lvmlockd doesn’t directly update the metadata cache but it wants to output log to guide other LVM commands to sync the metadata cache from storage. lvmlockd uses data structure val\_blk to maintain the lock version and store it into LVB:

/\* val\_blk version \*/

#define VAL\_BLK\_VERSION 0x0101

/\* val\_blk flags \*/

#define VBF\_REMOVED 0x0001

struct val\_blk {

uint16\_t version;

uint16\_t flags;

uint32\_t r\_version;

};

struct val\_blk vb;

memcpy(&vb, rds->vb, sizeof(vb));

rv = sanlock\_set\_lvb(0, rs, (char \*)&vb, sizeof(vb));

In the locking algorithm, deadlock and corresponding detection is an important topic. The document “Programming Locking Applications” [1] lists three possible deadlocks:

1. circular deadlock: the deadlock occurs when two or more processes are blocking with each other in a cycle of granted and blocked lock requests.
2. recursive deadlock: the same node acquires the lock but with incompatible lock modes.
3. conversion deadlock: conversion deadlock occurs when the requested mode of the lock at the head of the convert queue is incompatible with the granted mode of some other lock also on the convert queue. The first lock cannot convert because its requested mode is incompatible with a currently granted lock. The other lock cannot convert because the convert queue is strictly FIFO.

To resolve the deadlock issue, lock manager can detect deadlock if the same owner occurs more than once in the lock chain, it will notify the lock owner for the deadlock and usually the lock owner can release the lock and acquire it again to resolve the deadlock issue. So far, LVM tools and lvmlockd don’t have code to resolve deadlock due its locking sequence is simple and fixed, which is hard to cause deadlock issues.

## 2.5 Timeout and fence

Let’s firstly clarify the concept for lock timeout, it means the usage of a lock is expired, the owner doesn’t occupy the lock anymore and the lock will be available for other users; so *timeout* refers to lock owner’s life cycle and it can alter lock’s states (locked -> expired -> unlock).

The question is how lock managers apply the timeout on an owner and its acquired locks.

The timeout is maintained for host wise or VG wise, e.g. sanlock daemon uses delta lease to maintain membership for VG wise. Every lockspace thread periodically updates its host ID for delta lease; this approach is good for performance since every host only updates its timeout in the delta leases and doesn't need to traverse all its associated locks for timeout management.

After a failed host is timeout, it’s not safe to grant its acquired locks to other hosts; a race condition can happen between the failed host and the new granted hosts. Let’s look at an example: if host(a) fails to renew its membership, but host(a) still assumes itself can use the corresponding logical volume; since the lock lease is timeout, if host(b) acquires this lock and access the corresponding logical volume, finally this will cause data corruption due to both host(a) and host(b) can modify the same volume.

To tackle the race condition as mentioned, we need to fence out the failed host; fencing can deactivate the relevant block devices on the failed host, or relies on watchdog to restart the system as the last resort. So a second timeout mechanism ‘watchdog timeout’ is introduced. We will talk about fencing later in this section, now focus on timeout mechanism; the complete timeout period is shown as below:



Diagram 3: timeout for renewal membership and watchdog

The parameter io\_timeout (is set as 10s) is passed when a host joins into the lockspace:

rv = sanlock\_add\_lockspace\_timeout(&lms->ss, 0, sanlock\_io\_timeout);

And WATCHDOG\_FIRE\_TIMEOUT is a macro defined in sanlock repository:

#define WATCHDOG\_FIRE\_TIMEOUT 60

Lock manager needs to provide a fence, which allows the failed applications or hosts to exit from the cluster and release the shared resource properly. Let’s first see the fencing in the lock manager as shown in below diagram:



Diagram 4: fencing different components

Fencing handles several failures:

1. The first type is lvm tool failure; e.g. when the command vgchange is crashed in the middle of its execution, lvmlockd can detects the client connection failure for vgchange and it automatically releases the locks belonging to vgchange by calling function client\_purge().
2. The second failure is application failure. An application registers as a client for the lock manager; if the lock manager detects the application disconnection (e.g. use poll() to monitor connection from application), lock manager can release the acquired resources for the failed application.
3. The third failure is the I/O or networking failures. If lock manager detects the I/O failure or fabric issues when access the locking media, e.g. shared storage cannot be accessed properly, lock manager needs to kill applications so that allow the relevant applications to exit from the cluster and don’t use the shared resource anymore. To kill applications, approach one is to send signal SIGTERM/SIGKILL to the applications, another way is to invoke the predefined command with kill path and kill parameters (e.g. use lvmlockctl command to deactivate block devices and release locks for volume group: /sbin/lvmlockctl --kill testvg2).
4. The fourth failure relies on the watchdog to recovery system. If the lock manager fails to kill an application or the lock manager continuously cannot renew membership for the host, lock manager skips to pet watchdog and the watchdog will reset the whole system.

## 2.7 Interfaces

Below table lists lvmlockd wrapper APIs and lock manager APIs:

Table 1: lvmlockd wrapper APIs and lock manager functions

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Request | DLM wrapper APIs | DLM functions | Sanlock wrapper APIs | Sanlock functions |
| init\_vg | lm\_init\_vg\_dlm() |  | lm\_init\_vg\_sanlock() | sanlock\_version()  sanlock\_write\_lockspace()  sanlock\_write\_resource() |
| free\_vg | lm\_hosts\_dlm()  lm\_rem\_resource\_dlm() |  | lm\_hosts\_dlm()  lm\_rem\_resource\_sanlock() | sanlock\_get\_hosts() |
| busy\_vg | lm\_hosts\_dlm() |  | lm\_hosts\_sanlock() | sanlock\_get\_hosts() |
| find\_free\_lock | n/a |  | n/a |  |
| start\_vg | lm\_prepare\_lockspace\_dlm()  lm\_add\_lockspace\_dlm() | dlm\_kernel\_version()  dlm\_new\_lockspace() | lm\_prepare\_lockspace\_sanlock()  lm\_add\_lockspace\_sanlock() | sanlock\_register()  sanlock\_killpath()  sanlock\_restrict()  sanlock\_add\_lockspace\_timeout()  sanlock\_set\_config() |
| start\_wait |  |  |  |  |
| stop\_vg | lm\_rem\_lockspace\_dlm() | dlm\_release\_lockspace() | lm\_rem\_lockspace\_sanlock() | sanlock\_rem\_lockspace()  sanlock\_write\_lockspace() |
| vg\_update | Deferred to update in unlock |  | Deferred to update in unlock |  |
| query\_lock\_lv | handled in lvmlockd to query resource | n/a | handled in lvmlockd to query resource | n/a |
| init\_lv |  |  | lm\_init\_lv\_sanlock | sanlock\_read\_resource()  sanlock\_write\_resource() |
| free\_lv |  |  | lm\_free\_lv\_sanlock | sanlock\_write\_resource() |
| rename\_vg\_before | Handled in lvmlockd to check if have hosts | n/a | Handled in lvmlockd to check if have hosts | n/a |
| rename\_vg\_final | n/a | n/a | lm\_rename\_vg\_sanlock() | sanlock\_read\_lockspace()  sanlock\_write\_lockspace() |
| running\_lm | lm\_is\_running\_dlm() | Directly access /sys/kernel/config/dlm/cluster/cluster\_name | lm\_is\_running\_sanlock() | sanlock\_version() |
| refresh\_lv | lm\_refresh\_lv\_start\_dlm() | Execute command ‘lvm lvchange --refresh --partial --nolocking vgname/lvname’ with function dlmc\_run\_start() | n/a |  |
| lock\_gl | lm\_lock\_dlm() | dlm\_ls\_lock\_wait() | lm\_lock\_sanlock()  lm\_unlock\_sanlock() | sanlock\_acquire()  sanlock\_get\_lvb()  sanlock\_release()  sanlock\_set\_lvb() |
| lock\_vg | lm\_lock\_dlm() | dlm\_ls\_lock\_wait() | lm\_lock\_sanlock()  lm\_unlock\_sanlock() | sanlock\_acquire()  sanlock\_get\_lvb()  sanlock\_release()  sanlock\_set\_lvb() |
| lock\_lv | lm\_lock\_dlm() | dlm\_ls\_lock\_wait() | lm\_lock\_sanlock()  lm\_unlock\_sanlock() | sanlock\_acquire()  sanlock\_get\_lvb()  sanlock\_release()  sanlock\_set\_lvb() |

# 3 Requirement for IDM lock manager

This chapter maps the requirement for IDM lock manager based on the analyzed features in the previous chapter.

Since we don’t arrive at the point for software design, IDM lock manager is uncertain: it could be the same thing with the drive firmware if lvmlockd directly talks to drive firmware, or lvmlockd might interact with an IDM daemon and thus IDM lock manager contains two components: IDM daemon and drive firmware. The requirement is coming from lvmlockd and it is for an abstracted IDM lock manager.

## 3.1 Initialization

At the first initial phase when launch lvmlockd, we need to confirm if the IDM locking scheme is available or not. E.g. if lvmlockd can make success in communicating with IDM lock manager, lvmlockd can assume the IDM is supported.

In the second initial phase for creating a VG, this is a good chance to check if IDM lock manager is supported; if there is any info we can be reused from the first initial phase, we can directly decide if IDM is supported and avoid sending messages (or SCSI commands) to IDM lock manager.

The third initial phase is for activating a VG. IDM lock manager needs to confirm if it has sufficient resources for the new activating VG, and need to register membership into IDM lock manager. Note that VGs might use different drives for its IDM locking, thus at this stage the host registers into multiple drives for the membership maintenance.

As discussed IDM lock manager doesn’t use extra shared storage for locking storing and management and it uses slots MAXLBA+(n) for locking management. The reason I mention this is because using MAXLBA+(n) approach can significantly reduce complexity for initial phases due we don’t need to initialize shared storage.

## 3.2 Lockspace

We cannot simply apply the previous chapter’s lockspace analysis on IDM lock manager. It’s natural for DLM and sanlock lock managers to create lockspace, the reason is applications can connect with lock managers when creating a lockspace; and DLM and sanlock neither can use a centric place to store membership and locks, lockspace creates connection between hosts and locks. IDM has significant differences with two points, the first difference is hosts don’t need to take extra steps to connect with IDM lock manager and can directly send SCSI commands to drive; the second critical point is locks and hosts, so it can easily create ownership between locks and hosts if without lockspace.

Let’s consider if IDM requires namespace with lockspace. I.e, two VGs (say vg1 and vg2) both create a logical volume (e.g. vg1/testlv, vg2/testlv), since two logical volumes use the same name testlv thus introduces naming confliction if they are under the same namespace. IDM uses Universally Unique IDentifier (UUID) as VG and LV naming, as wikipedia stated “the probability that a UUID will be duplicated is not zero, it is close enough to zero to be negligible”, the naming conflict with the same UUID has very little chance.

To avoid complexity in IDM lock manager, we can take IDM drive firmware as a common lock space and below are requirements for IDM lock manager:

IDM lock manager uses a list for membership maintenance, and the host (or application) needs to send a command to join to this list.

IDM lock manager uses a list to track locks, any new allocated locks need to be added into this list.

If any user acquires a lock, the IDM lock manager needs to maintain relationship between locks and their users.

IDM lock manager supports namespace, to simplify the code, we can simply take VG’s UUID as namespace ID.

## 3.3 Membership management

A merit for IDM is that we can use it as a single copy for membership maintenance.

The first requirement for IDM lock manager is unique IDs. Since we have ambitions to use IDM to support multiple instances on the same host, thus we can use two IDs host\_uuid and process\_id to present unique ownership. E.g. If two applications (one is lvmlockd and another is the distributed file system) on the same host use IDM concurrently, these two apps can use the same host\_uuid and unique process\_id to distinguish these two owners.

The second requirement is for membership renewal. Applications or IDM daemon needs to renew membership, IDM lock manager needs to maintain membership renewal or the owner has expired.

It’s transparent for IDM firmware for how to set host\_uuid and process\_id in applications; for some usage cases, a host can simply use host\_uuid and don’t use process\_id (or always set process\_id to 0), the host can use a thread or timer to renew its membership; if multiple applications on the same host create connection with IDM firmware, then every application need to use both host\_uuid and process\_id to renew membership for itself.

## 3.4 Lock algorithm

IDM lock manager needs to support two lock modes: shareable mode and exclusive mode.

For exclusive mode, IDM lock manager tracks the single owner for the lock; for shareable lock mode, IDM lock manager needs to track all owners who have acquired this lock.

When the lock is acquired with shareable mode and it has a single owner, it’s permitted to convert to exclusive mode by the same owner. When the lock is acquired with exclusive mode, it’s allowed to convert to shareable mode by the same owner.

At the current stage, IDM lock manager can only provide the non-block operation; any lock operations will return immediately without waiting. This is corresponding to trylock and unlock operations.

Applications directly use GL/VG/LV UUID to refer a lock, IDM lock manager automatically allocate lock when at the first time acquire it and free resource when all users free this lock (e.g. for optimization, IDM lock manager don’t need to release lock immediately and defer with Least Recently Used (LRU) replacement).

IDM lock manager needs to consider support Lock Value Block (LVB) up to maximum 32 bytes. This is easier for upstreaming full feature support in lvmlockd.

Deadlock detection could be taken as a low priority and it’s not mandatory for lvmlockd.

IDM lock manager should provide debugging methods to dump state for locks, even if the IDM with unlock mode and without any owners, for this case we should consider if there are any ‘leak’ for locks, or this is caused by host failure and timeout. IDM lock manager can implement garbage reclaiming for this case.

## 3.5 Timeout and fence

IDM lock manager tracks timeout for every owner.

Application needs to pass two parameters for timeouts: renewal timeout and watchdog timeout. IDM lock manager needs to use these two timeouts: the renewal timeout means the owner has expired its ownership, and watchdog timeout means it’s safe to grant the locks associated with the expired owner to other owners.



Diagram 5: Two phases timeout for IDM

The section 4.5 Timeout and fence introduces the concept **quiescent period** for IDM white list, we reuse the watchdog timeout for setting IDM **quiescent period.**

If any owner renews itself between the renewal timeout and watchdog timeout, it depends on IDM lock manager how to decide if the owner can renew successfully or not. If it cannot renew, then afterwards any renewal and any related lock operations will fail.

All the fencing happens on host, either in lvmlockd or in IDM daemon; fence is not handled in IDM firmware.

## 3.6 Requirement summary

Table 2: Requirements for IDM

|  |  |  |  |
| --- | --- | --- | --- |
| Requirement | Feature | Description | IDM Commands |
| Initialization | Phase 1: Launch daemons | Check if the IDM locking scheme is supported or not. | idm\_version()  We need to use idm\_version() to confirm if IDM is supported or not. |
| Phase 2:  Create VG |  |  |
| Phase 3:  Activate VG | ~~When start VG, we need to know if IDM has enough resources to support the VG~~  Registers membership into IDM lock manager |  |
| Lockspace | Host list | Maintain host list in IDM lock manager | ~~idm\_add\_owner()~~  ~~idm\_rem\_owner()~~  We don’t need to explicitly add the owner into drives, when acquiring the IDM, driver firmware automatically adds the host into its whitelist. When releasing the IDM, drive firmware will remove the host from its whitelist. |
| Lock list | Maintain lock list in IDM lock manager | We don’t need to explicitly allocate / free locks, we can use VG\_UUID and LV\_UUID to directly refer to a lock. |
| User count | Check has how many owners for a lock. | idm\_lock\_owner\_count() |
| Namespace | Make sure there are no naming conflicts for LVs.  Note: LVs in two different VGs might have the same LV UUID, so unique logical volume identifier for LV is:  VG uuid + LV uuid | LV’s ID = VG\_UUID (32B) +  LV\_UUID (32B) |
| Membership | Multiple instance | Support multiple instances on the same host | Host\_uuid + process\_id |
| Lock algorithm | Lock mode | Shareable and exclusive lock mode | IDM\_LOCK\_SHERABLE |
| Opt mode | Nonblock or block | IDM\_LOCK\_NONBLOCK |
| Lock operations | Acquire / release / convert | idm\_lock()  idm\_unlock()  idm\_convert\_lock() |
| LVB | Support lock value block | idm\_get\_lvb()  idm\_set\_lvb() |
| Deadlock detection | ~~This is not mandatory and we can take it as low priority~~ |  |
| Timeout | Support timeouts | Support two stages timeout: renewal and watchdog timeout | idm\_add\_owner(...,  renew\_timeout,  wd\_timeout) |

## 3.7 Interfaces

Table 3: IDM Wrapper APIs in lvmlockd

|  |  |  |
| --- | --- | --- |
| Lvmlockd  Request | Description | IDM Wrapper APIs |
| init\_vg | When users specify to use the IDM lock scheme for creating VG, need to confirm if IDM is supported or not. | lm\_idm\_version() |
| free\_vg | Don’t need to release any resources in free\_vg, all resources will be released in stop\_vg rather than free\_vg. |  |
| busy\_vg | Decide if VG is busy based on the host/owner count for VG lock. | lm\_idm\_owner\_count(vg\_uuid,  NULL)  Note: for VG lock, lv\_uuid is passed with NULL. |
| find\_free\_lock | Find an available lock.  This is not needed by DLM; but sanlock needs to find a free slot for lock and this lock slot will be used by init\_lv. We can defer to allocate idm in init\_lv. |  |
| start\_vg | Activate VG, add ownership and allocate a lock for VG.  We need to add ownership only once for the same drive per vg\_uuid. The drive firmware will return error for duplicate ownership registration. | ~~idm\_add\_owner(vg\_uuid,~~  ~~host\_uuid, proc\_id,~~  ~~renew\_timeout,~~  ~~wd\_timeout)~~ |
| start\_wait | Count how many lockspaces are started by a specific client; this is only finished in lvmlockd and It’s not related to the lock manager. |  |
| stop\_vg | Deactivate VG, remove ownership and free VG lock. | lm\_idm\_owner\_count(host\_uuid,  proc\_id) |
| vg\_update | Increase its resource version number, and this resource version number is deferred to write into LVB when unlock.  When next time any other hosts acquire the same lock, it compares the resource version number in LVB and its local resource version number so can get to know VG has been updated by other hosts. | lm\_lock\_idm()  `> idm\_raid\_read\_lvb()  lm\_unlock\_idm()  `> idm\_raid\_write\_lvb() |
| query\_lock\_vg | Query a VG lock with its lock name, returns 0 for success with the lock mode; otherwise, if cannot find lock will return failure. |  |
| query\_lock\_lv | Query a LV lock with its lock name, returns 0 for success with the lock mode; otherwise, if cannot find lock will return failure. |  |
| init\_lv | Prepare for the new LV and allocate lock resources for the created LV.  Sanlock uses it to allocate a paxos lease as LV lock and initialize the lock name into paxos lease’s block. This is not needed for IDM. |  |
| free\_lv | Cleanup for the freed LV.  Sanlock uses it to cleanup paxos lease to “#unused” state. This is not needed by IDM.. |  |
| rename\_vg\_before | Before rename a VG, it needs to check if there are any other hosts using VG, so can inquire with lm\_host().  If there are other hosts using VG, returns -EBUSY. | lm\_idm\_owner\_count(vg\_uuid,  NULL) |
| rename\_vg\_final | Rename lock name when the VG is renamed. Since IDM doesn’t use VG name and only uses VG UUID, thus, we don’t need to do anything for VG renaming. |  |
| running\_lm | Return success when IDM lock is supported, otherwise, return failure. | lm\_idm\_version() |
| refresh\_lv | This request is dedicated to DLM, lvmlockd uses dlm interface to ask all nodes running dlm\_controld to run a command:  lvm lvchange --refresh --nolocking <path>  This request is not supported by IDM. |  |
| lock\_gl | Acquire a lock; if this lock is not existed in IDM drive firmware, IDM driver firmware needs to allocate one for it:  op = LD\_OP\_LOCK  rt = LD\_RT\_GL  mode = LD\_LK\_SH / LD\_LK\_EX | lm\_lock\_idm(gl\_uuid,  NULL,  mode,  IDM\_LOCK\_NONBLOCK,  host\_uuid,  proc\_id) |
| Convert a lock:  op = LD\_OP\_LOCK  rt = LD\_RT\_GL  mode != old\_mode | lm\_convert\_idm(gl\_uuid,  NULL,  mode,  IDM\_LOCK\_NONBLOCK,  host\_uuid,  proc\_id) |
| Releasing a lock; if this lock is not used by any one, IDM drive firmware can release it:  op = LD\_OP\_LOCK  rt = LD\_RT\_GL  mode = LD\_LK\_UN | lm\_unlock\_idm(gl\_uuid,  NULL,  host\_uuid,  proc\_id) |
| lock\_vg | Acquire a lock; if this lock is not existed in IDM drive firmware, IDM driver firmware needs to allocate one for it:  op = LD\_OP\_LOCK  rt = LD\_RT\_VG  mode = LD\_LK\_SH / LD\_LK\_EX | lm\_lock\_idm(vg\_uuid,  NULL,  mode,  IDM\_LOCK\_NONBLOCK,  host\_uuid,  proc\_id) |
| Convert a lock:  op = LD\_OP\_LOCK  rt = LD\_RT\_VG  mode != old\_mode | lm\_convert\_idm(vg\_uuid,  NULL,  mode,  IDM\_LOCK\_NONBLOCK,  host\_uuid,  proc\_id) |
| Releasing a lock; if this lock is not used by any one, IDM drive firmware can release it:  op = LD\_OP\_LOCK  rt = LD\_RT\_VG  mode = LD\_LK\_UN | lm\_unlock\_idm(vg\_uuid,  NULL,  host\_uuid,  proc\_id) |
| lock\_lv | Acquire a lock; if this lock is not existed in IDM drive firmware, IDM driver firmware needs to allocate one for it:  op = LD\_OP\_LOCK  rt = LD\_RT\_LV  mode = LD\_LK\_SH / LD\_LK\_EX | lm\_lock\_idm(vg\_uuid,  lv\_uuid,  mode,  IDM\_LOCK\_NONBLOCK,  host\_uuid,  proc\_id) |
| Convert a lock:  op = LD\_OP\_LOCK  rt = LD\_RT\_VG  mode != old\_mode | lm\_convert\_idm(vg\_uuid,  lv\_uuid,  mode,  IDM\_LOCK\_NONBLOCK,  host\_uuid,  proc\_id) |
| Releasing a lock; if this lock is not used by any one, IDM drive firmware can release it:  op = LD\_OP\_LOCK  rt = LD\_RT\_VG  mode = LD\_LK\_UN | lm\_unlock\_idm(vg\_uuid,  lv\_uuid,  host\_uuid,  proc\_id) |
| kill\_vg | Kill VG so disallow any operations afterwards |  |
| drop\_vg | Stop the lockspace thread and release resource |  |
| dump\_log | Dump lock state. |  |
| dump\_info | Dump log buffer. |  |
| stop\_all | Stop all volume groups and their lockspaces and resources. |  |

# 4 Compute node software stack design

Chapter 3 describes how lock manager interacts with IDM drive, this chapter will decide the best route to modify lvmlockd and other helper programs to exploit IDM. Besides the design needs to consider the implementation, we should consider ahead the design to be friendly for the testing and support a feasible testing strategy. I avoid bringing up potential software stack at the start of this chapter, the reason is if we can review the implementation with detailed flow and figure out potential difficulties, this can give us a clear idea to summarize software stack design at the end of this chapter.

## 4.1 Initialization

On the system with support Seagate IDM, we need firstly to install the Seagate Propeller package and Seagate cluster manager utilities Speedboat.

In the Seagate Propeller package, the IDM based lvmlockd will be launched. This is corresponding to the first initialization phase which was mentioned in the previous chapters. At this stage lvmlockd and IDM lock manager begin to run but it doesn’t access any IDM devices, we simply assume there are no errors. If there are any potential issues (e.g. I/O connection issue) for IDM, we can defer to handle failure in the later sequence.

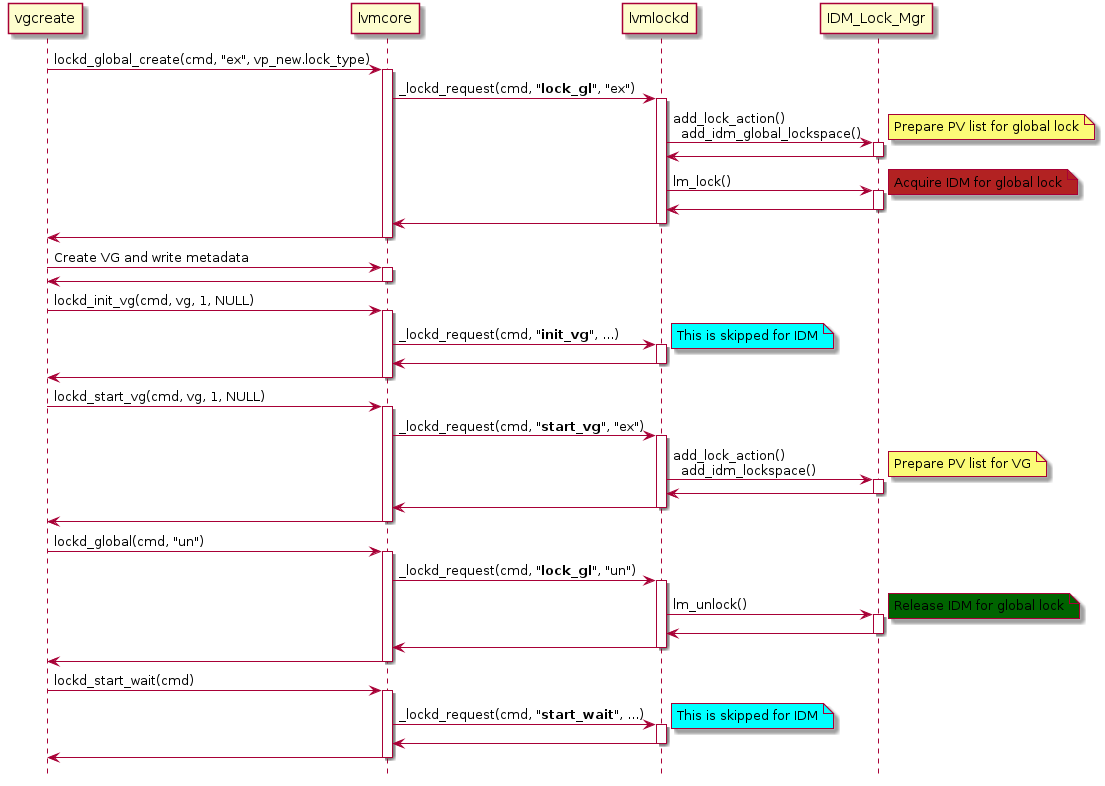


Diagram 6: Initialization for IDM lock manager

In the second initial phase, the system creates VGs by Speedboat. The global lock is required to protect metadata for creating VGs; a difficult thing is how to prepare global lock for IDM bootstrap. As shown in the diagram, when first time the user tries to acquire global lock, there have no any existed VG thus the lock manager has no knowledge of associated PVs; to resolve this issue, IDM lock manager can read out the PV list from Speedboard configuration file, this PV list is predefined and can be used for majority locking algorithm for global lock. This approach has the benefit that the global lock’s PV list is not dependent on any VGs and it can get ready at the start of day.

When the new VG is created, the global lock is used to protect metadata accessing; the detailed steps for acquiring and releasing locking will be deferred to explain in the lock algorithm section.

The request start\_vg is invoked automatically in the second initial phase, or manually in the third initial phase with command vgchange. start\_vg is a good occasion to prepare PV list for VG; the reason is that at this point it only acquires global lock but not VG lock so it’s smoothly to prepare PV list for VG lock and we can avoid chicken and egg problem between VG lock and PV list (E.g. if we must acquire VG lock before we can read out its PV list, but VG lock is dependent on PV list).

Simply to say, after initialization we have two kinds of PV lists: global lock’s PV list and VG’s PV lists, and how to maintain the related data structures will be described in the next section.

## 4.2 Data structure

### 4.2.1 Data structure in lvmlockd

We have a discussion for lockspace, the design tries to avoid complexity for context management in either host or drive firmware. As discussed, using UUID based reference for host and lock can simplify the context management rather than handler based management. The requirement for majority locking algorithm introduces complexity for data structure, PV and drive list are required to acquire lock, and the drive list is required to refresh and release lock. Essentially, lockspace is about how to define data structure, so this section focuses on data structure definition.

Let’s first summarize the UUID management. In the diagram, lockspace is a common data structure and lockspace::lm\_data points to lock manager specific data; IDM lock manager can use this pointer to refer idm\_lockspace; **idm\_lockspace::host\_uuid** records host UUID and **idm\_lockspace::process\_id** records process ID.

lockspace::vg\_uuid and resource::name records VG’s and LV’s UUID respectively, this is not friendly to unify low level flow with these two IDs, we can use **idm\_lock::uuid[64]** to record lock’s UUID with 64 bytes size. **idm\_lock::flags** is used to track the lock mode, e.g. shareable mode.

The data structure lockspace adds two fields: pvs and pv\_num; the PV list are passed from lvmcore when start a VG (or global lock), these two fields track the PV list, thus IDM lock manager knows to send IDM commands to which IDM drives. Since multiple physical volumes can reside in the same one drive, we need to convert PV list to drive list and stores into **idm\_lock::drive\_list**. The drive list is used to acquire lock, every drive cannot grant lock to all requesters, so **idm\_lock::score\_board[]** and **idm\_lock::score** are used to present which and how many drives have granted lock to self; these two fields are used to refresh lock with majority locking algorithm.

IDM lock has a vb pointer **idm\_lock::vb**, which is used to point lock value block. It points a dynamically allocated buffer, this buffer is used to write and read IDM’s lvb in drives.

So the data structures are summarized below:

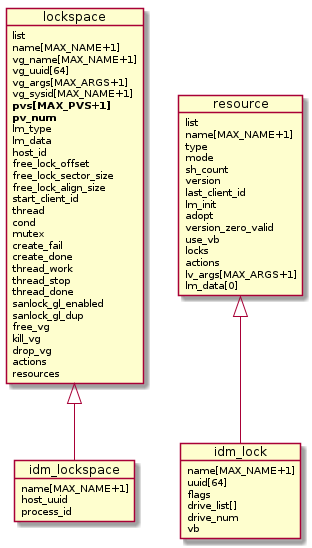


Diagram 7: Data structure in lvmlockd

Membership

idm\_lockspace::host\_uuid

idm\_lockspace::process\_id

IDM Locking

idm\_lock::uuid[64]

idm\_lock::flags

idm\_lock::vb

Majority locking

idm\_lock::drive\_list

Idm\_lock::drive\_num

### 4.2.2 Data structure in IDM lock manager

In the IDM lock manager, one low layer is to implement the majority locking; we use a client data structure to keep tracking the connection from lvmlockd or other applications who want to use IDM locks. **client::killpath** and **client::killargs** are used to launch commands or send signals for failure handling.

lockspace data structure is used to mainly for VG level’s renewal and maintain a watchdog; for any renewal failure, **lockspace::renew\_fail** is set to indicate the serious issue caused by losing lock lease; for this case, IDM lock manager should to use **client::killpath** and **client::killargs** to deactivate block devices and ‘drop’ VG; and generally, the lockspace structure will be destroyed at the end of this action and the watchdog device is closed.

Lockspace needs to maintain locks for the same VG, it can be a VG lock or LV locks. **lock::lock\_uuid** is used to track lock’s UUID, **lock::mode** is for lock’s mode; and to support majority locking algorithm, **lock::drive\_list** is used to track the whole drive list.

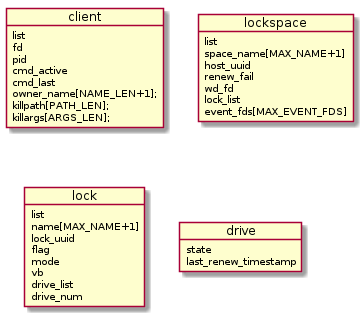


Diagram 8: Data structure in IDM lock manager

Every drive node on the drive list uses **drive::state** to track the drive state transition, based on the drive state and **drive::last\_renew\_timestamp**, the majority lock flows can predict when the IDM will be timeout on the failed drive; and according to the drive state, majority lock flows can call appropriate functions.

The drive states will be divided into three types: unaccess, accessed, and accessed\_and\_failed. **unaccess** is the initial state before a host uses IDM; **accessed** is the state after a host has acquired an IDM on the drive; **accessed\_and\_failed** state is to indicate the drive fails after acquiring the IDM on the drive.

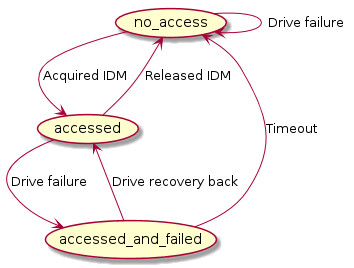


Diagram 9: Drive state machine

**Updated for implementation:**

The drive state machine has been refined in the implementation, it is divided into below states:

* IDM\_INIT: the mutex is in init state, it might be unlocked or released in drive;
* IDM\_BUSY: the mutex is used by other hosts, can try to break mutex if other hosts have been timeout;
* IDM\_DUPLICATE: the initial host has acquired the mutex and returns error when the host tries to acquire twice, for this case, need to release the mutex firstly and acquire it again;
* IDM\_LOCK: the mutex has been granted to the host;
* IDM\_TIMEOUT: the mutex has been timeout for the host;
* IDM\_FAULT: fail to access the drive after the mutex was granted to the host.

## 4.3 Membership management

As described in the previous section, we use **idm\_lockspace::host\_uuid** and **idm\_lockspace::process\_id** to present membership; this can ensure the membership is unique for any host and its applications. In the low level IDM lock manager, these two IDs will be combined and stored into **lockspace::host\_uuid**.

The second requirement for membership is to use an appropriate approach to renew membership for locks. At the beginning it was planned to renew membership for host wise, this means when a member (a host or an application) renews, all the locks lease for this member will be automatically extended until next timeout window. This is an efficient way to renew membership which is shown below.

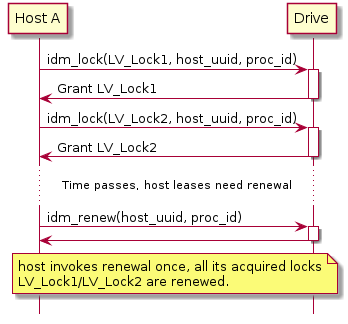


Diagram 10: Host wise renewal

After introducing raid fault tolerance, a lock is not only resident in single drive but it can span on multiple drives. It’s possible for a host to renew once for all its acquired locks with SCSI command, but the operation is not strictly atomic. If a lock resides in multiple drives, any drive might fail and drive failures can be accumulated to cause the half-a-brain failure; every lock’s failure is specific and might be different from other locks, even the locks belong to the same VG. For this reason, we need to renew with lock wise rather than VG wise.

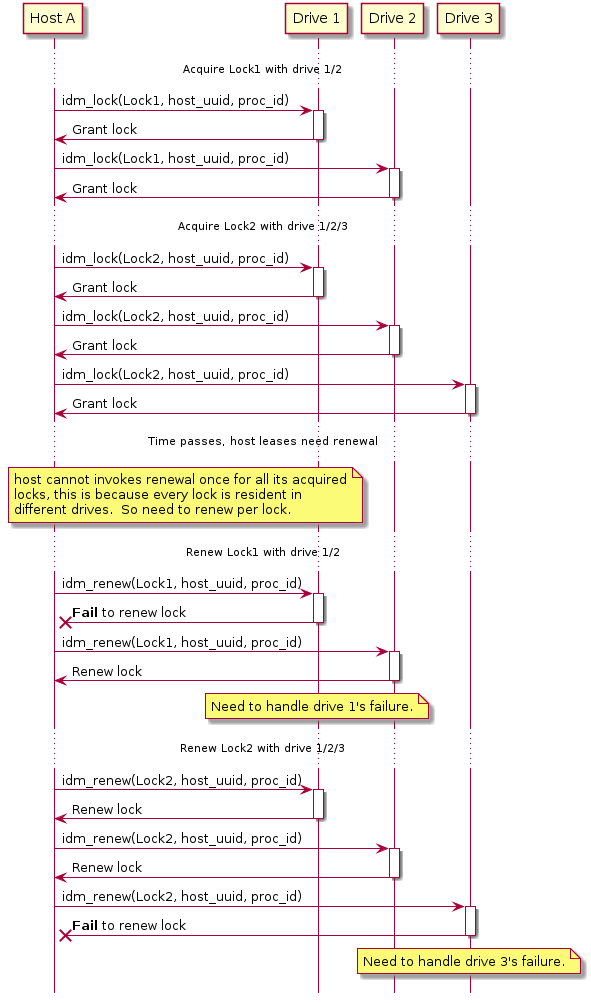


Diagram 11: lock wise renewal

In this section we decide the data structure and approach for membership renewal, the detailed implementation for locking renewal will be discussed in the next section for locking algorithm, both for the IDM SCSI wrapper APIs and raid fault tolerance related flow.

## 4.4 Lock algorithm

To allow lvmlockd to easier use IDM for RAID and support raid fault tolerance, it’s reasonable to add a new layer encapsulates locking operations for RAID and invokes IDM SCSI wrapper to send commands to drives. The complete routes from lvmlockd wrapper APIs to IDM SCSI wrapper are shown as below:

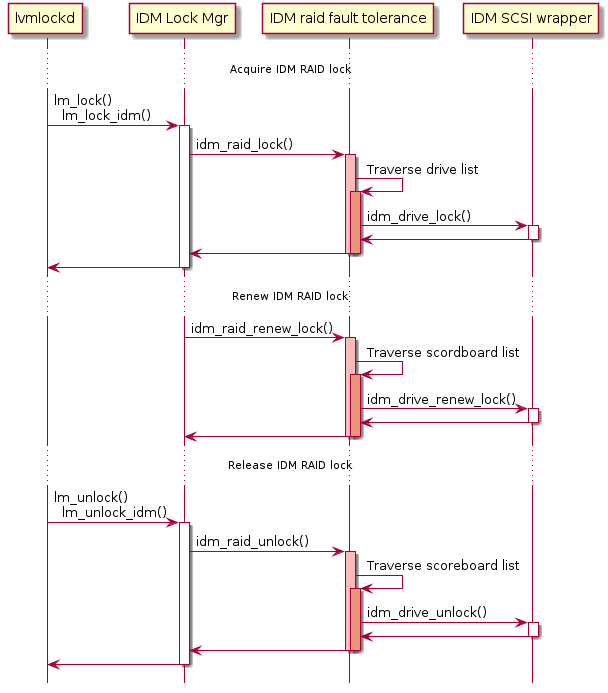


Diagram 12: IDM RAID fault tolerance layer

### 4.4.1 Principles for locking majority

Let’s claim the locking majority principles.

* A host can be granted the lock through voting majority on RAID drives.
* A host with a locking majority has a **duty** to seek to enlarge the majority of any lock that was not unanimous across the drives.
* A host with a locking majority has a **duty** to propagate the latest IDM state if it finds any inconsistent states caused by drive’s failure.
* A host with a locking majority has a **duty** to propagate the latest LVB if it finds any inconsistent states caused by drive’s failure.
* An owning exclusive IDM contributes to the lock majority of both the exclusive and shared associated locks.
* A host with a locking majority relies on the IDM to enforce the promotion restriction of no other shareable locks besides requesting owner. And when converting the lock mode, it shall maintain a list of owned quorums and issue demotion if failed to promote a quorum.
* A host has a **duty** to release IDM for drives; if drives fail, the host needs to use a drive list to track failed drives and loop to try releasing IDM until release successfully or until the predicted timeout has been reached.

### 4.4.2 Acquisition majority locking

Two drive lists are used for majority locking algorithm: **idm\_lockspace::drive\_list** is the list for all relevant drives for a specific lock, **idm\_lock::score\_board** is the drive list recording drives which have granted the lock. So drive\_list is an input parameter and score\_board is an output after acquiring a lock. The majority calculation formula is:

count(score\_board) >= count(drive\_list) / 2 + 1

When acquiring majority lock, if simply return after acquiring majority count (count(drive\_list) / 2 + 1), this will be dangerous since any drive failure will cause losing majority when renewing the lock. For this reason, this is why we claim the principle to seek to enlarge the majority of any lock that was not unanimous across the drives.

During acquiring majority lock, it’s possible to find an IDM has been acquired by the same host duplicately. This can happen if the drive failed when the host tried to release the IDM at the previous time, the failed drive recoveries back, then in the next time the host acquires the IDM and finds the IDM has been granted already with the same host. For this case, the drive firmware needs to return error -ELOCK\_DUPLICATE, the host can know the IDM has been acquired by itself. The two times’ acquisitions might have different lock modes, to ensure acquiring IDMs with the exact same state, the host needs firstly release the lock to cleanup the IDM content in drive firmware and then acquire the IDM again.

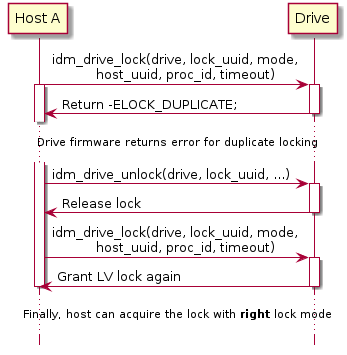


Diagram 13: Sync with ***correct*** state when acquire lock

The diagram below suggests traversing all drives in drive\_list and tries to acquire locks as possible in all associated drives; this can give enough room for tolerance drives fault. If the host cannot achieve majority, it backoffs to release all acquired IDMs; after waiting for 10ms, it can start next round to acquire IDMs from the first drive for the majority locking. To avoid infinite loop, the flow will give up to acquire locking until timeout.

When acquiring the majority locking, the drive state will be changed from unaccess state to accessed state; if the drive is failed at this time point, simply keep the drive state as unaccess.

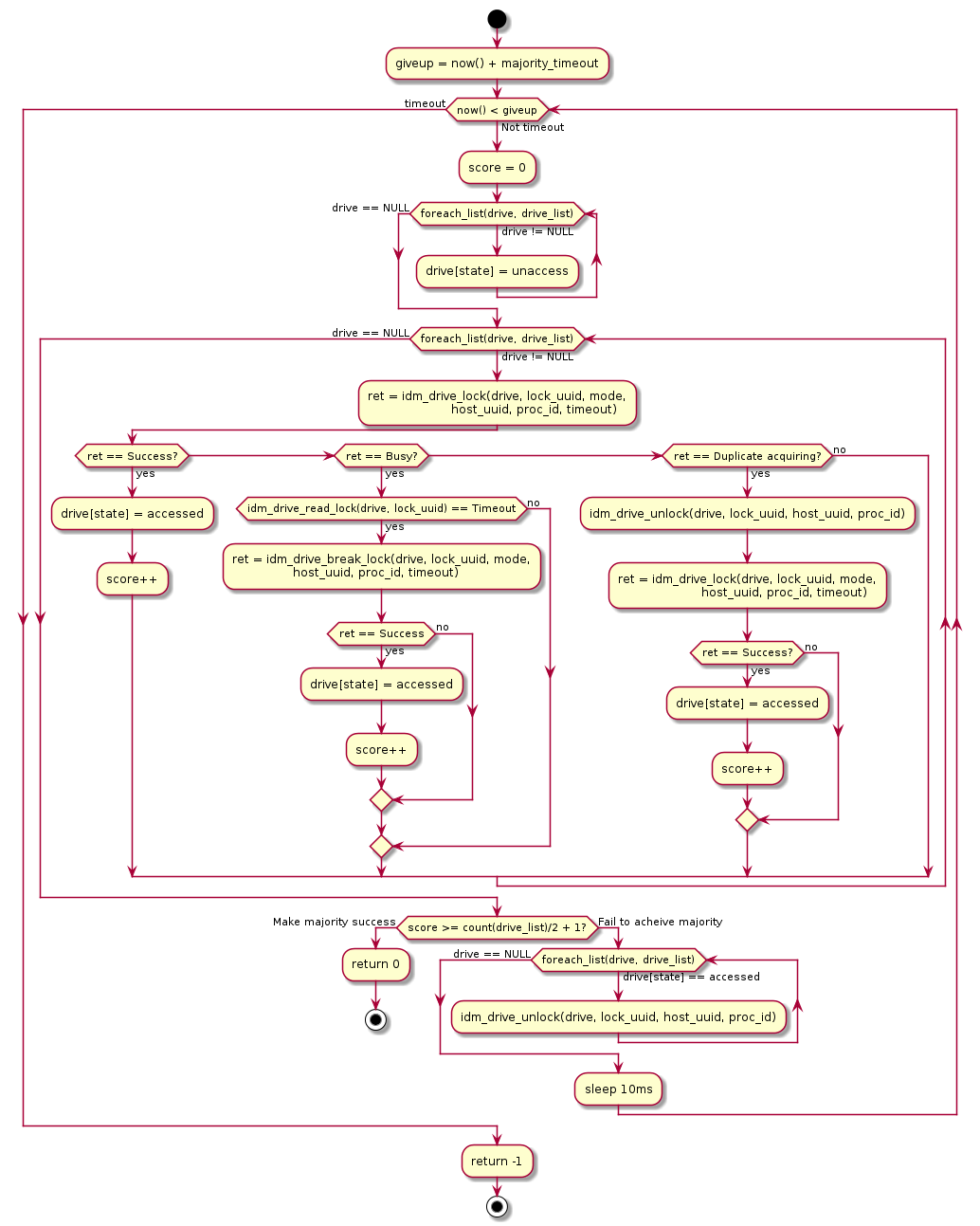


Diagram 14: Majority locking flow

**Updated for implementation:**

There is a limitation for the PV list length, since now the different components use socket to send messages, and it doesn’t support variable length messages, this is the reason now we predefine the PV list to support max to 32. If later requires to support a bigger PV list, we need to consider extending this part or optimizing to use a more flexible method to send messages (e.g. to support variable length for PV list). This issue is tracked in the ticket: <https://github.com/Seagate/propeller/issues/104>.

In the implementation, the drive number has been changed into three classes in implementation: ‘good\_drive\_num’ presents the number for drives which have been found corresponding SG node and it counts the drive number after merged the duplicate PV list; ‘fail\_drive\_num’ means the number of drives have not found corresponding SG node; ‘total\_drive\_num’ is the sum value of ‘good\_drive\_num’ and ‘fail\_drive\_num’. ‘total\_drive\_num’ will result in an oversized majority because it has not been reduplicated (too many votes needed to proceed). The ticket <https://github.com/Seagate/propeller/issues/97> is used to track related discussion.

### 4.4.3 Lock renewal

When renewing a lock for its majority, communication errors (e.g. fabric failure) can cause the half-a-brain failure and lead to the renewal failure. In below diagram, it gives an example that host A has acquired a LV lock, afterwards host A tries to renew the LV lock and the half-a-brain failure happens which is caused by drive 2 and 3 communication timeout; when host A receives timeout failure from drive 3, the drive whitelists is not safe due to the drive 3 lease expiration.

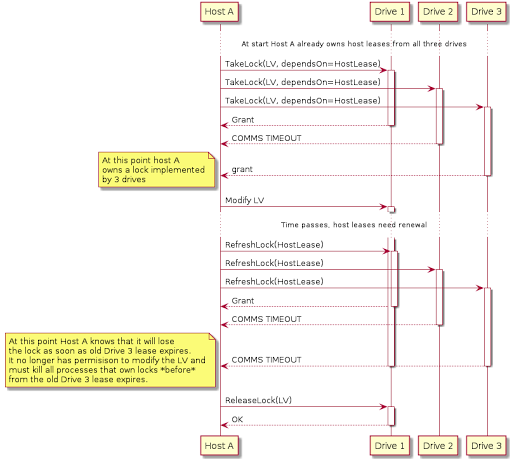


Diagram 15: Drives half-a-brain failure

The renewal failure can cause a serious consequence for the host losing the lock lease, the lock manager must take action to deactivate the corresponding block devices so that avoid data corruption. On the other hand, the locking renewal needs to use a more robust algorithm to avoid the half-a-brain failure easily happening.

Firstly, renewal majority can be calculated with the formula below. This formula is different from the previous section for acquiring locking is for the drive list with even number; when acquiring a lock, the majority condition must be 50% drives plus 1, afterwards if the owner renews the lock for at least 50% drives, other hosts have no chance to achieve the condition (50% drives plus 1), thus it’s safe for the owner to keep the majority.

if ((count(drive\_list) is even number)

count(score\_board) >= count(drive\_list) / 2

else

count(score\_board) >= count(drive\_list) / 2 + 1

The optimization for even number drives is very useful for the lock resident in only 2 drives, at beginning when acquiring lock with 2 drives for majority, if later any drive fails, the lock can be renewed successfully if only one drive is alive; this allows the majority algorithm to be more robust.

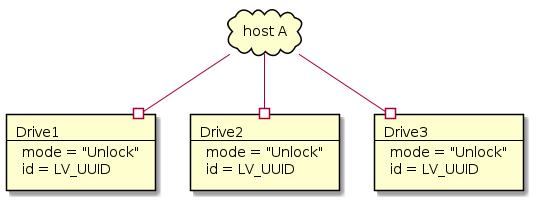
Note, this approach cannot apply on drives with odd numbers, 50% of odd numbers (e.g. for fixed point calculation, 50% of 3 is 1 and the rest part is 2) will give other hosts a chance to acquire the majority. So for the case of odd number drives, renewal lock keeps to use the same majority formula with the acquiring lock.

Secondly, to avoid the half-a-brain failure, another improvement is to try to acquire IDM on the whole drive list rather than only renewal IDMs which have been acquired. To renew the whole drive list, except we can use this chance to enlarge majority cross all drives, the host with a locking majority can also utilize this chance to sync IDM states.

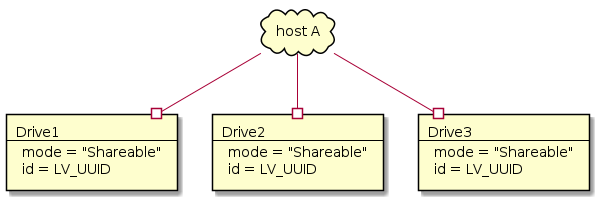
To enlarge the majority when renewing IDMs on all drives, it’s possible the IDM on some drives have not been granted to the host due to the drives failure before when acquired locking majority. For this case, drive::state is unaccess, the host will take this chance to acquire the IDM.

When renewing a lock, we need to handle the case for inconsistent IDM states in drives. Let’s see the detailed state transition which causes inconsistent IDM states and at the end leads to the majority locking broken.

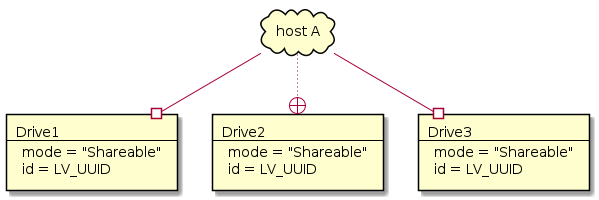
Step 1: Initial state is a LV lock working on three drivers with unlock mode.



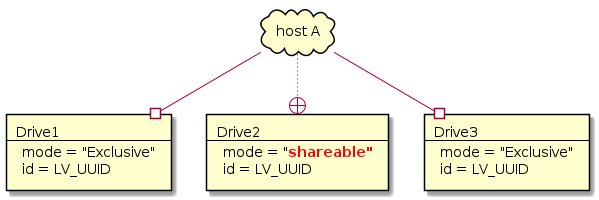
Step 2: Host acquires LV lock with shareable mode and achieves the majority.



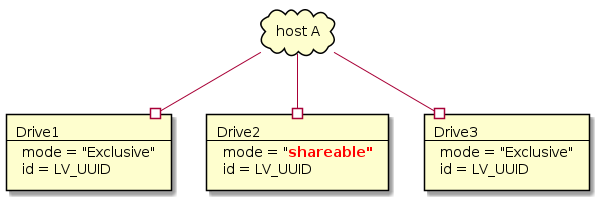
Step 3: Drive 2 fails and the host disconnects with it; host A still keeps its majority by renewal drives 1 and 3.



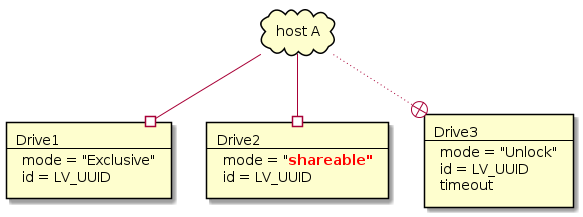
Step 4: The host converts the lock from shareable mode to exclusive mode; since drive 2 is disconnected, the host a fails to change lock mode on drive 2.



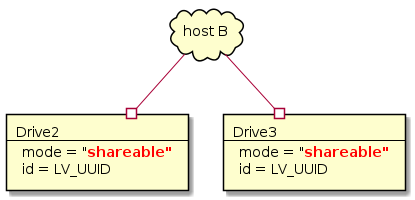
Step 5: The host A reconnects with drive 2; the LV lock is not timeout so host A renews the lock on drive 2 successfully.



Step 6: The drive 3 fails and host A disconnects with it, and the lock is timeout in drive 3.



Step 7: Host B tries to acquire the lock with shareable mode, though it fails to acquire IDM in drive 1, but it can achieve majority by acquiring IDMs successfully on drives 2 and 3.



At the end, host A and B both acquire the same lock with majority locking, one is with exclusive mode and another is with shareable mode. To fix this issue, we need to sync the IDM state in step 5: when host A tries to renew its lease, the lock manager needs to pass the locking mode associated with lock’s UUID to drive; in the drive firmware, it compares the locking mode between the passed parameter and its local copy, when it finds the mismatching between these two values, the firmware returns an specific error (e.g. -ELOCKMODE\_MISMATCH) to notify the IDM lock manager. When receiving the error, the IDM lock manager needs firstly to release and acquire again for the IDM with the new lock mode.

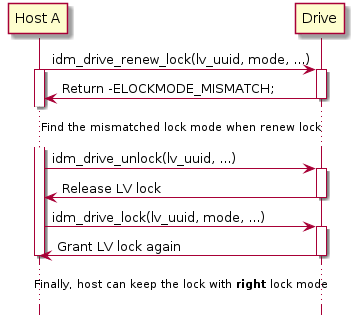


Diagram 16: Sync the ***correct*** state when renew lock

When renewing a lock, we need to handle drive states:

* If drive is unaccess state, try to acquire IDM on it to enlarge majority;
* If drive is accessed state, renew the IDM; if the return result is success, the drive state will keep as accessed state, otherwise will transit to accessed\_and\_failed state;
* If drive is accessed\_and\_failed state, try to renew the IDM; if return the lock mode mismatching issue, as explained above we need to release and acquire IDM again, thus the state can be transferred to accessed state; if cannot make success, it will stay at accessed\_and\_failed state.

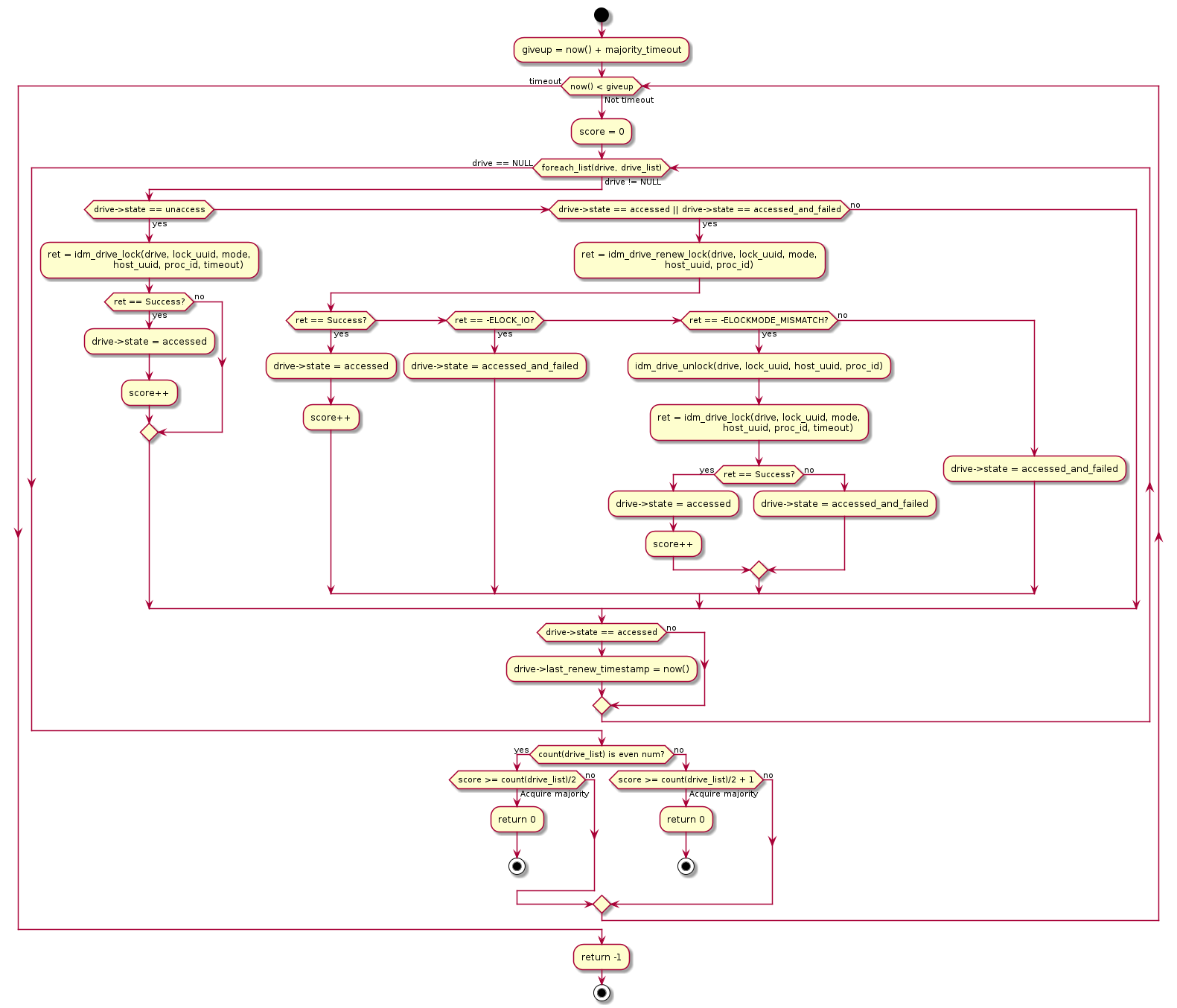


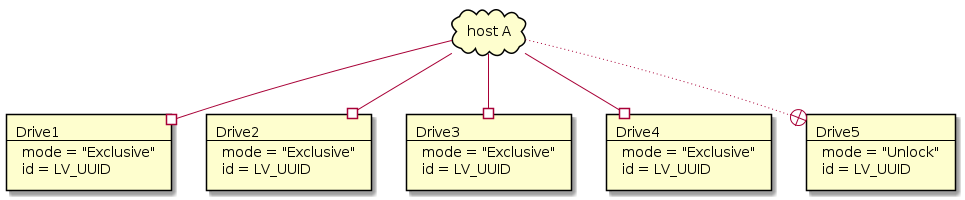
Diagram 17: Majority lock renewal flow

### 4.4.4 Lock mode converting

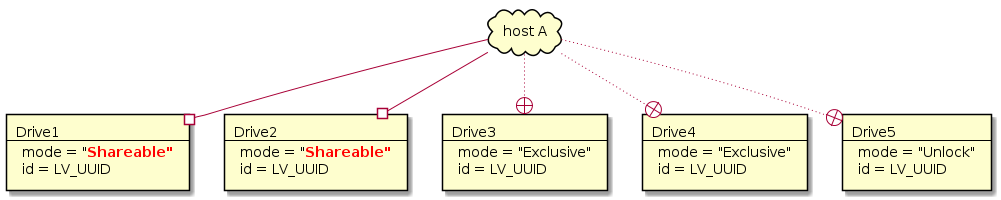
For converting lock mode, even has failure during lock mode conversion, if the host can achieve the majority with new lock mode, the conversion function can return success and allow application to use the new lock mode.

Drives failure in the middle of converting might cause a mess that lock mode will not be consistent across all drives and even lose the majority with the new mode. If failed to convert lock mode and rollback, it is possible to be more complex since drive failures might happen during rollback, thus some drives use new mode and some drives use old mode. This can cause a serious issue; let’s see the scenario below.

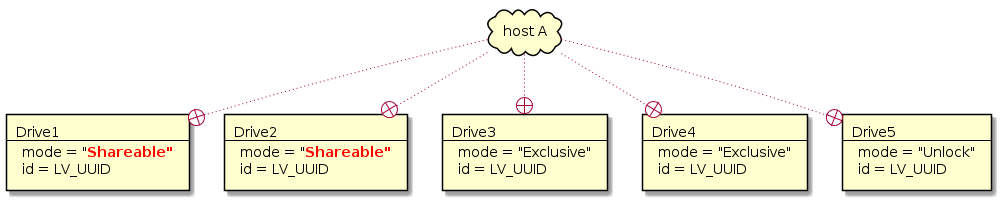
Step 1: when host A starts to convert lock mode from exclusive mode to shareable, drive 1/2/3/4 are connected but drive 5 is disconnected by any hardware reason.



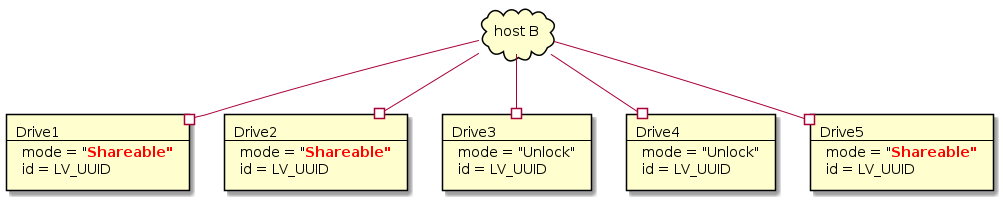
Step 2: host A has converted the lock mode on drives 1/2 from exclusive mode to shareable mode; but other drives have failed. So drives 3/4 both contains the old exclusive lock mode, and drive 5 always is unlocked.



Step 3: host A has found it cannot make success to convert to shareable mode with majority, thus it plans to rollback to exclusive mode. But at this time point, all drives are disconnected with host A.



Step 4: host A keeps to use the lock with exclusive mode; host A assumes even it cannot access drives, but it can continue to use the lock until timeout. At this time point host B can acquire successfully on drive 1/2/5 with shareable mode. Finally, at the meantime, host A acquires the lock with exclusive mode and host B acquires the same lock with shareable mode.



To resolve this issue, we can use guidelines for lock mode converting:

* When converting from shareable mode to exclusive mode, it must meet the condition for majority (50% drives + 1), otherwise rollback to shareable mode and returns error to notify upper layer.
* When converting from exclusive mode to shareable mode, it can always make success; if cannot achieve the majority with shareable mode, we can defer to use lockspace thread to sync IDM state with shareable mode.
* An extra action can be considered for if previous time the lock mode converting returns success but without majority, we can set a flag to disable the mode converting on the lock, until the lockspace thread syncs the mode with majority and clear the flag.

The detailed lock converting flow is as below:

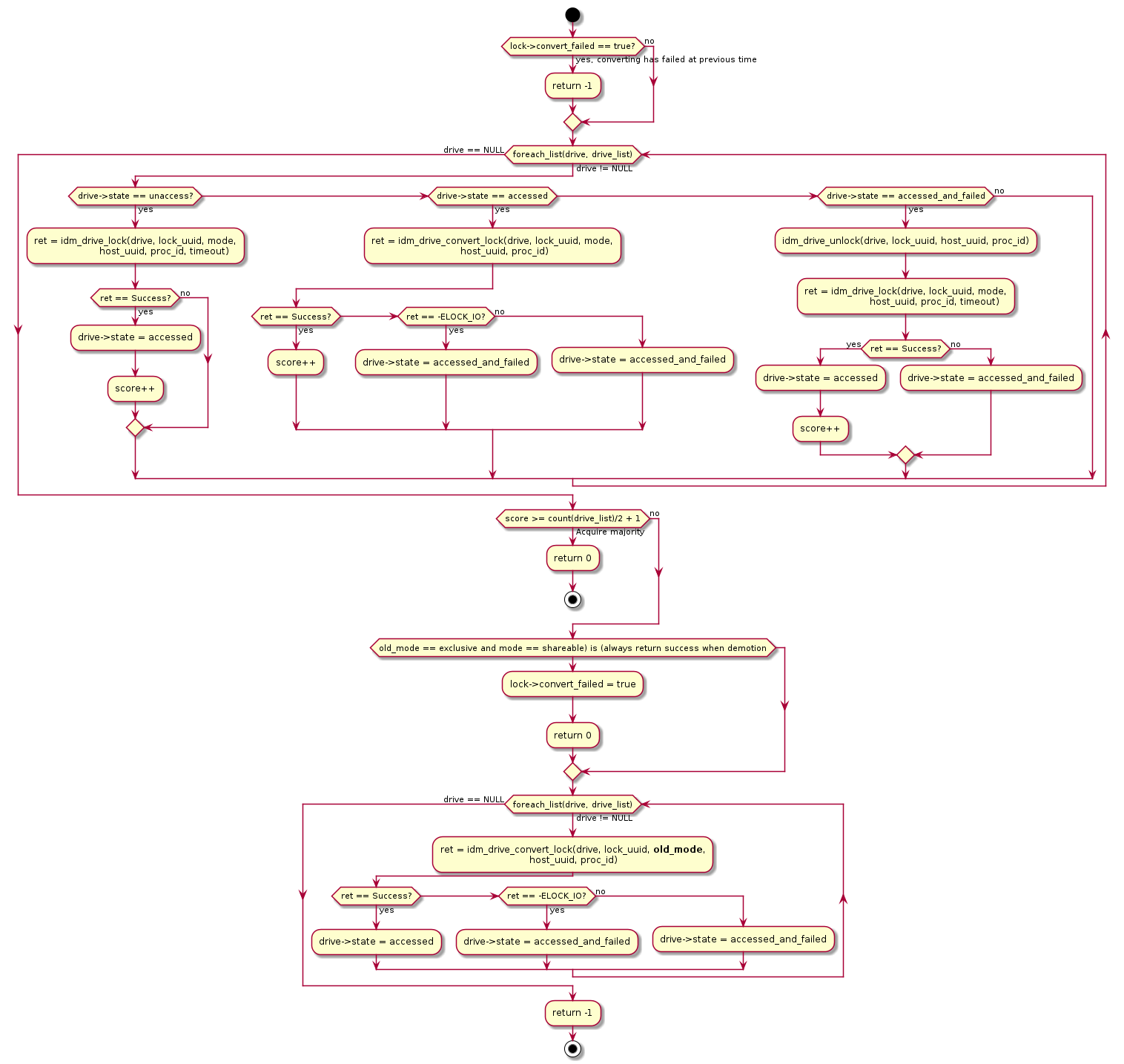


Diagram 18: Majority lock converting flow

### 4.4.5 Lock releasing

Releasing lock traverses drives in the drive list and releasing the IDMs one by one; if there is any failure on the drive, the drive state will be recorded and the lockspace thread needs to release IDM for the failed drives until timeout. Below flow can be used both in the releasing function and lockspace thread.

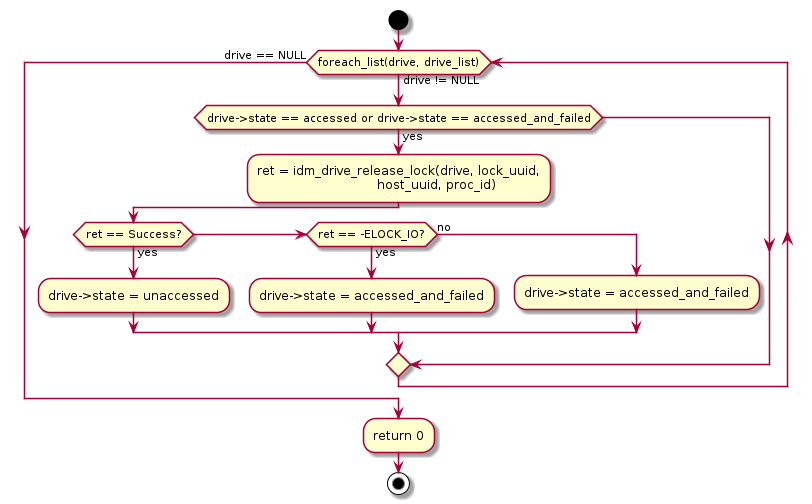


Diagram 19: Majority lock releasing flow

### 4.4.6 LVB

To allow all hosts to work on the same page after the VG metadata has been modified, lock’s LVB is used for syncing VG metadata. lvmlockd’s LVB is defined as below:

struct val\_blk {

uint16\_t version;

uint16\_t flags;

uint32\_t r\_version;

};

VG lock uses val\_blk::r\_version to track VG’s metadata version. Below example shows after host A has modified VG, it increments val\_blk::r\_version from 0 to 1 and stores into lock’s LVB; on the other hand, when host B tries to acquire VG lock, it reads out val\_blk::r\_version and finds the LVB’s r\_version (1) is bigger than its local version number resource->version (0), so host B knows that other hosts must have modified VG and it must read its VG metadata from disk and drop the stale cached data. lvmlockd uses this method to notify hosts to invalidate its VG cached data.

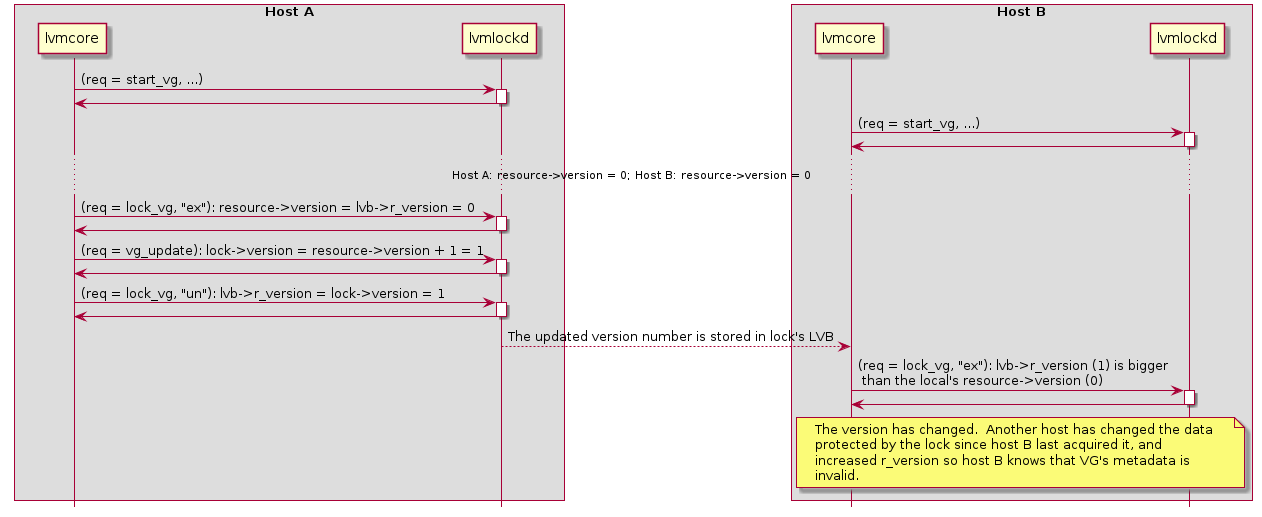


Diagram 20: Syncing version with LVB within two hosts

Below are the guidelines to ensure the LVB date is synchronized between readers and writers:

* Either writing or reading LVB, hosts need to access successfully for majority drives (>= 50% drives + 1); otherwise returns error to notify up layers for the failure.
* If writing LVB fails, needs to roll back to the old version number.
* When reading out LVB, the host should find out the highest version number from all accessed drives.

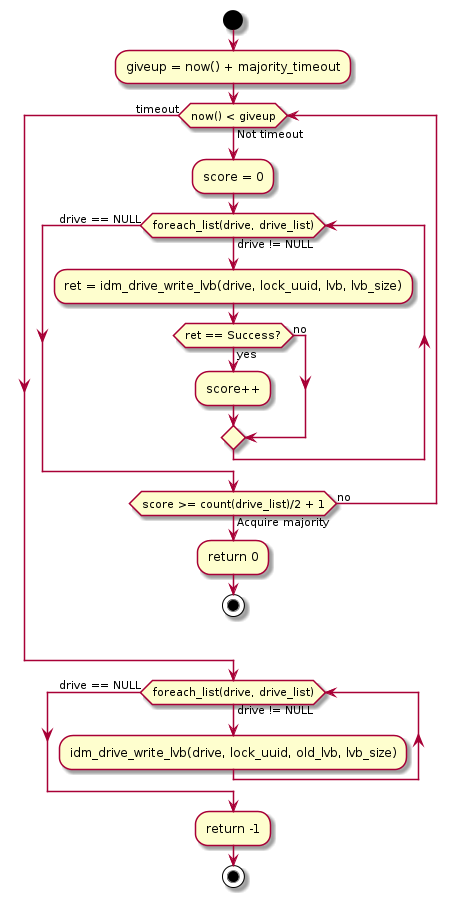


Diagram 21: LVB writing flow

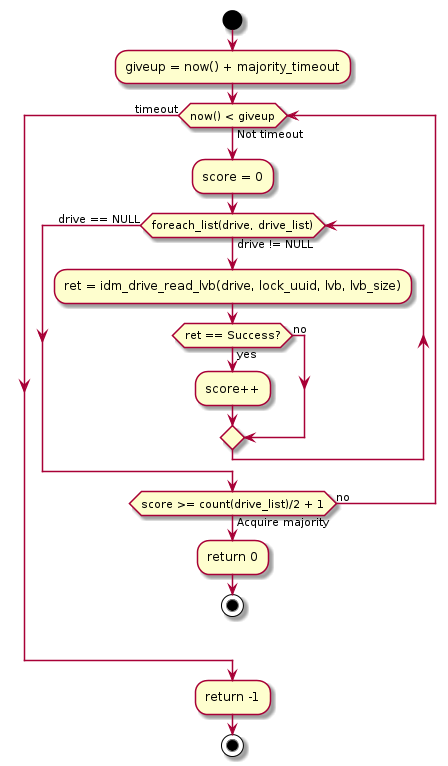


Diagram 22: LVB reading flow

In the next section, we will discuss more details for the drive list. A main usage for IDM’s LVB is to sync drive list after the VG has been changed by one of the hosts. If a host has changed VG, e.g. adds new PVs into VG, it can send the request vg\_update to update the PV list in the lock manager. But the issue is the drive list has not updated in other hosts, so the drive list might be inconsistent on these hosts. When hosts detects the VG’s version number has been incremented in LVB, users can manually stop and start VG again, finally all hosts can have the consistent PV list for the updated VG.

### 

Diagram 23: LVB is used for syncing VG metadata

**Updated for implementation:**

Since the mutex might be destroyed in drive firmware, this can cause the issue for LVB to be reset to zeros and restart to increment the version number. This might cause below error:  
  
Host A:

Step 1: Acquire VG1 mutex, read LVB, version number is 0;

Step 2: Increase version number from 0 to 1, release VG1 mutex.

Host B:

Step 1: Acquire VG1 mutex, read LVB, version number is 1;

Step 2: Increase version number from 1 to 2, release VG1 mutex.

Drive firmware has no sufficient memory, release mutex and LVB

Host C:

Step 1: Acquire VG1 mutex, read LVB, version number is 0;

Step 2: Increase version number from 0 to 1, release VG1 mutex.

Host A:

Step 1: Acquire VG1 mutex, read LVB, version number is 1; **it finds the version number has not been changed, so assume the VG1 has not been changed, thus don’t flush cached metadata**.

Step 2: Increase version number from 0 to 1, release VG1 mutex.

Due to the drive firmware might release mutex and reset LVB, the version number stored in LVB will rollback to zero and increment again. For this reason, it causes host A cannot correctly detect the LVB has been changed when it activates VG1 at the second time, so fail to flush cached metadata.

To ensure the version number can be continuously incremented, especially after the mutex has been removed from drive, lvmlockd saves the timestamp into mutex’s LVB and maintains the mapping between the version number and timestamp. Since the timestamp will increment and don’t rollback to zero, so if any host has changed the version number, it will update the latest timestamp into LVB, other hosts will know the LVB has been updated. If the LVB is reset to zero, hosts will take this as a special case and notify to flush cached metadata. Let’s see below example for the timestamp usage:

Host A:

Step 1: Acquire VG1 mutex, read out LVB with the timestamp is 0;

Step 2: Detects the timestamp is zero, always notify daemon to flush cached metadata;

Step 3: Increase version number from 0 to 1, save timestamp T1 into LVB, release VG1 mutex.

Host B:

Step 1: Acquire VG1 mutex, read LVB with the timestamp is T1;

Step 2: Detects the timestamp T1 is different, notify daemon to flush cached metadata;

Step 3: Increase version number from 0 to 1, save timestamp T2 into LVB, release VG1 mutex.

Drive firmware has no sufficient memory, release mutex and LVB

Host C:

Step 1: Acquire VG1 mutex, read LVB with the timestamp is 0;

Step 2: Detects the timestamp is zero, always notify daemon to flush cached metadata;

Step 3: Increase version number from 0 to 1, save timestamp T3 into LVB, release VG1 mutex.

Host A:

Step 1: Acquire VG1 mutex, read LVB with the timestamp is T3;

Step 2: Detects the timestamp T3 is different with its local timestamp T1, notify daemon to flush cached metadata;

Step 3: Increase version number from 1 to 2, save timestamp T4 into LVB, release VG1 mutex.

### 4.4.7 State machine for drive list

This section reviews detailed state machine transition for drive list (or PV list).

Global and VG lock stores the PV list into lockspace::pvs. When every time acquire IDM majority lock, the IDM wrapper layer needs to convert lockspace::pvs to drive list and store into idm\_lock::drive\_list.

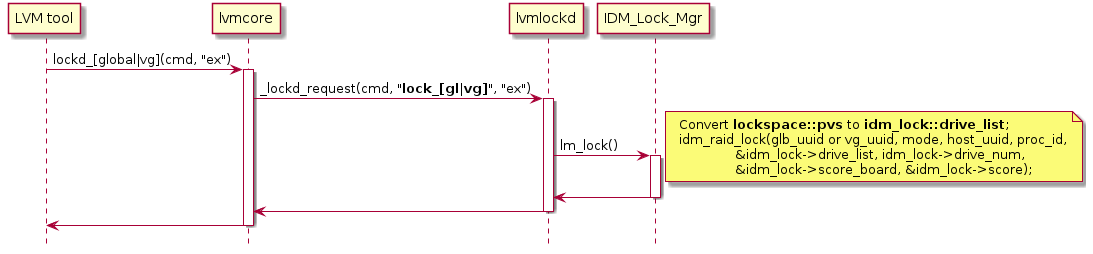


Diagram 24: Drive list for global and VG locks

The global lock doesn’t change its drive list, this is different from VG lock. If any VG metadata has been modified, VG can use vg\_update to update lockspace::pvs.

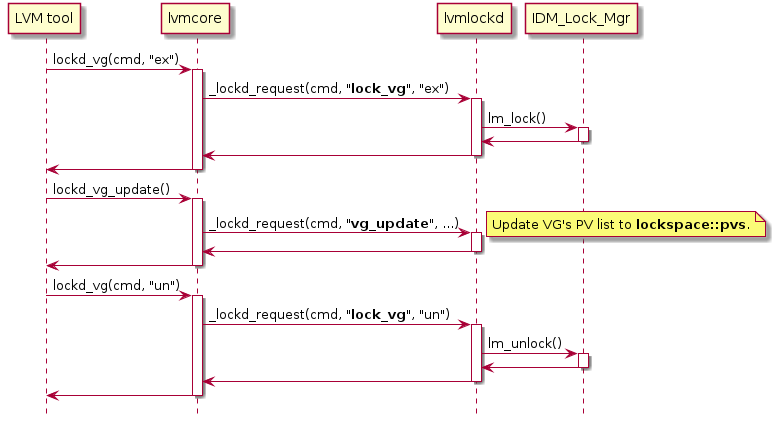


Diagram 25: Drive list updating for VG lock

As we discussed, we have some rough ideas to use VG drive list for both VG lock and LV lock. Since LV has less drives than VG level, LV uses VG’s drive list for majority locking, it can acquire more IDMs and avoid the chance for the half-a-brain failure. Let’s review the detailed flow for this and find potential issues based on this assumption, which is explained in the below steps:

Step 1: Let’s assume there are 7 drives in the cluster; so at the beginning, the global lock adds these 7 drives into its drive list.



Step 2: Create vg1 with the partitions on drives 1/2; when creating VG, we need to acquire global lock by occupying IDM majority IDMs on drives 1~7, this can protect the race condition when two hosts create the VGs with the same name simultaneously. Since drives 1/2 are associated with vg1, create drive list for vg1 as below:



Step 3: Create lv1 in vg1, when creating the logical volume it acquires vg1’s lock based on vg1’s drive list; after creating LV it releases vg1’s lock:



Step 4: Host A activates lv1 by acquiring lv1’s lock with exclusive mode; lv1 uses vg1’s drive list, so lv1’s lock resides in drives 1/2:



Step 5: Later extend vg1 with drives 3/4/5, this is protected by global lock; after executing vgextend command, drives 3/4/5 are added into vg1’s drive list:



Step 6: When host B activates lv1, it uses the updated vg1’s drive list; though it fails to acquire IDMs in drives 1/2 (which has been acquired by host A) but it can make success in drives 3/4/5 with exclusive mode; finally both host A and host B can acquire lv1’s lock with exclusive mode:



To resolve the elaborated issue, let’s use LV’s PV list for LV’s lock. LV lock doesn’t have vg\_update liked request to update PV list for logical volume level, if we store LV’s PV list in the lock manager, we have no chance to update PV list and drive list when LV is changed. For this reason, when everytime acquire LV lock, pass LV’s PV list is passed from lvmcore to lvmlockd, and lvmlockd builds up drive list based on the info passed from lvmcore; to gain a LV lock we should successfully obtain the lv\_uuid lock from all reachable drives in the PV list.

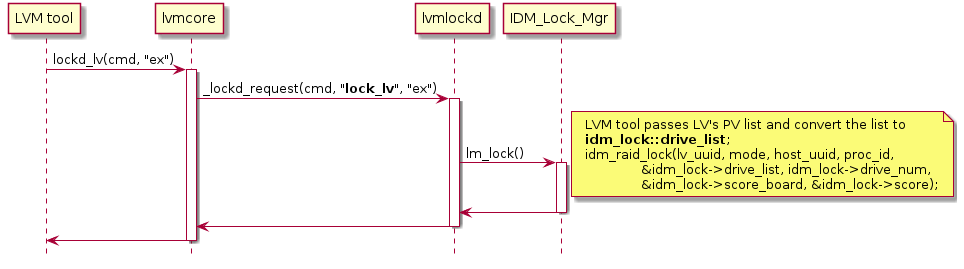


Diagram 26: Drive list updating for LV lock

## 4.5 Timeout and fence

The failures can be caused by different reasons, we need to handle these failures separately:

* LVM tool failure;
* lvmlockd failure;
* IDM lock manager (thread or daemon) failure;
* IDM drives failure.

The first type is LVM tool failure. LVM tool (e.g. vgchange, lvchange commands) creates the socket connection with lvmlockd, when the LVM tool exits abnormal, lvmlockd can detect the client disconnecting, thus lvmlockd automatically releases the locks with the client ID. LV lock usually is used as "persistent" lock, this is different from global and VG locks; even if the LVM tool crashes, LV lock will not be released so can promise the LV’s safety.

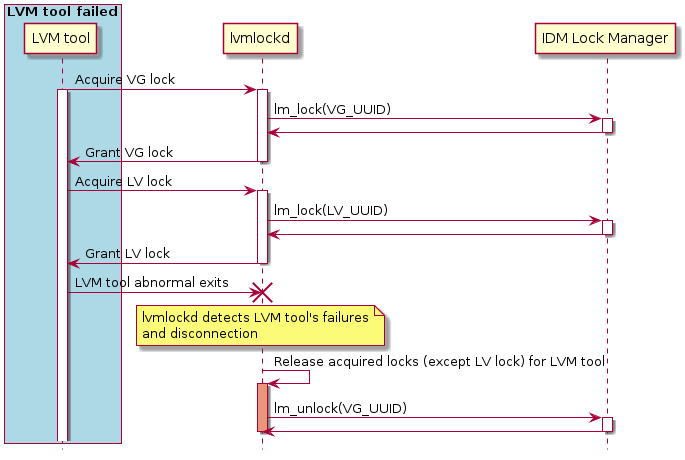


Diagram 27: LVM tool failure handling

The second failure is lvmlockd failure. When lvmlockd registers into IDM lock manager, it needs to claim the orphan locks cannot be released in the lock manager and can be adopted after lvmlockd is relaunched.

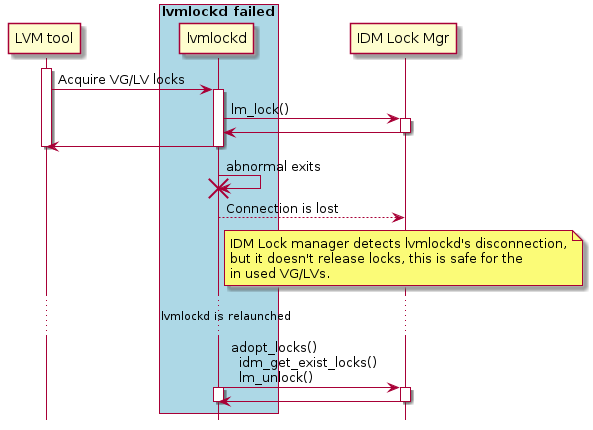


Diagram 28: lvmlockd failure handling

The third type failure is IDM lock manager abnormal exiting. As suggested in lvmlockd manual, if the lock manager daemon fails or exits while any lock is acquired is started,

the local watchdog will reset the host.

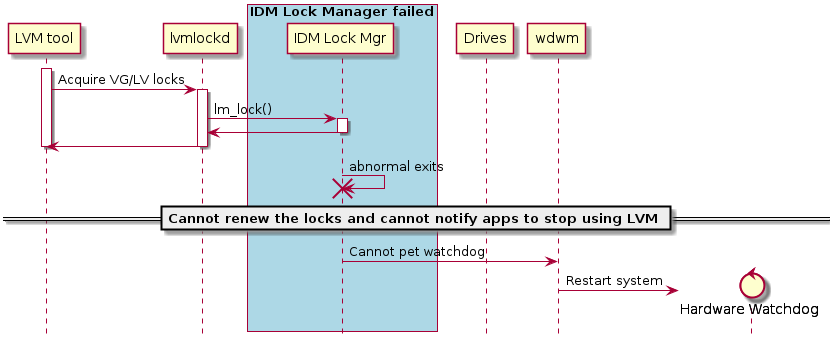


Diagram 29: IDM lock manager failure handling with watchdog

For the IDM lock manager failure, the unresponsive host will be removed from the drive’s white list when the IDM is timeout. After the IDM lock manager exits abnormally, from then on the host cannot renew IDM for all drives, but we cannot ensure all drives have fenced out the host from all drives’ white list at the same time. So the key point is how to promise the unresponsive host can be fenced out from all white lists at a specific time point.

To resolve this issue, it’s suggested to add a quiescent period into drive’s firmware. Let’s see the timing diagram, the timeout period is defined as 60s. At the 20s time point, it’s the last time the host can renew IDMs with majority, so drive 3 has failed at this time point. If the IDM lock manager continues to run for 30 seconds, at the 50s time point drive 1 is renewed. This means the drive 3 is timeout at 80s and drive 1 is timeout at 110s due to drive 1 had been renewed for 30 seconds longer. We can add a **quiescent period** in the drive, the quiescent period is used to ensure the IDM is timeout in all drives and the host is impossible to write any data to any drive; so after the quiescent period the IDM can be safely granted to other hosts.

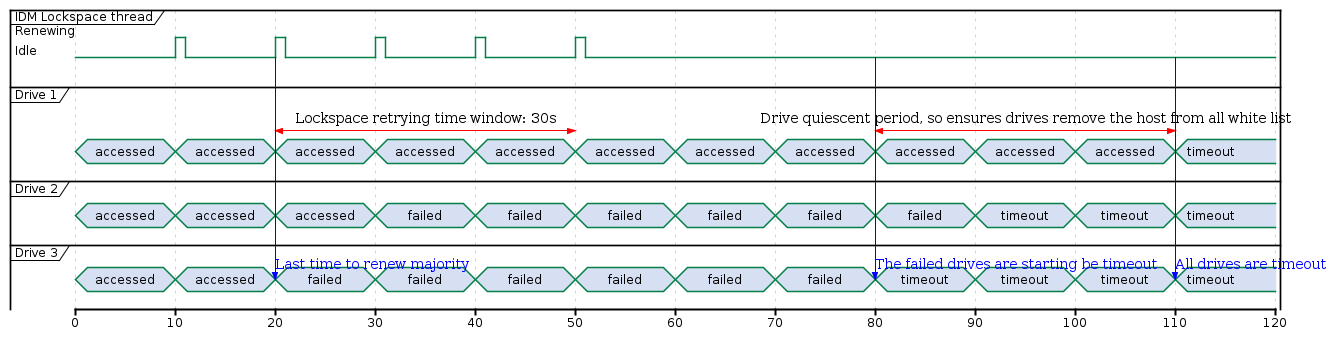


Diagram 30: Timing for drive timeout and quiescent period

Based on the diagram, the drive’s quiescent period is decided with the formula:

quiescent\_period = stopping\_renew - Last\_time\_renew\_majority\_success

= IDM lock manager’s retrying time window

So this requires the IDM lock manager and drive firmware to use the consistent period with each other. As result, the third failure handling can changed without watchdog restarting:

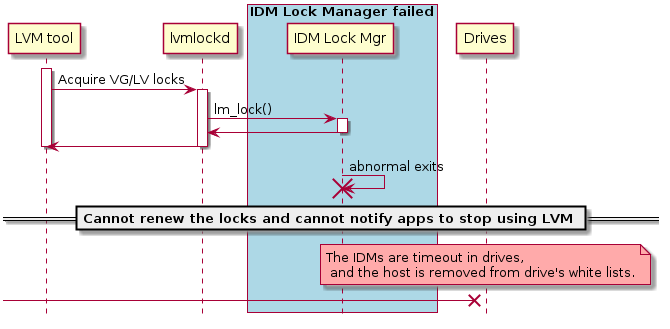


Diagram 31: IDM lock manager failure handling with IDM whitelist

The fourth failure is IDM drive failure. If IDM lock manager detects the I/O or fabric failures and fails to renew locks, the failure handling contains two directions processing:

1. The first direction is from IDM lock manager to application/lvmlockd. This direction is to notify application/lvmlockd to deactivate logical volume, in the below diagram, 'lvmlockctl --kill\_vg' is invoked. lvmlockctl needs to safely deactivate PV devices, it also sends request kill\_vg to lvmlockd so that lvmlockd can set flag kill\_vg so avoid any new coming lock operations for the VG.
2. The second direction is from app/lvmlockd to IDM lock manager, this process is mainly for cleanup locks and lockspace. So lvmlockctl sends request 'drop\_vg' to lvmlockd, lvmlockd iterates lock list to release all locks associated with VG and finally remove the lockspace.

Only when make the two directions processing success, IDM lock manager can ensure the applications on host A has exited gracefully and the related VG/LV have been deactivated;

The key point is to deactivate VG/LVs properly when host loses lock's lease; after the lock lease expires, the lock can be safely granted to other hosts.

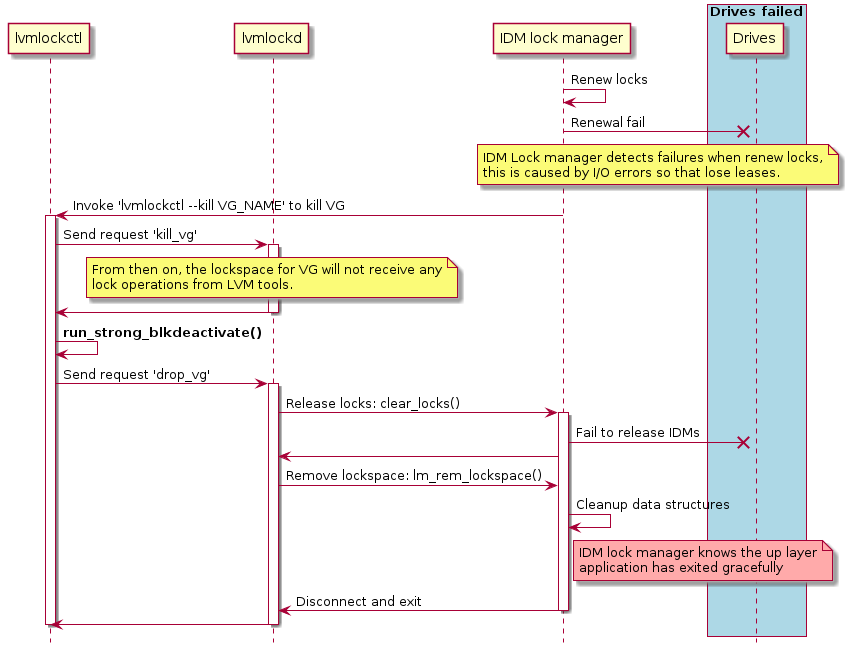


Diagram 32: Drive failure handling

If fail to deactivate block devices and also fail to renew locks, for this case, the application cannot be stopped to use LVM and the IDM lock manager cannot keep the lock’s lease, as a last resort the IDM lock manager skips to pet watchdog and the watchdog will reset the whole system.

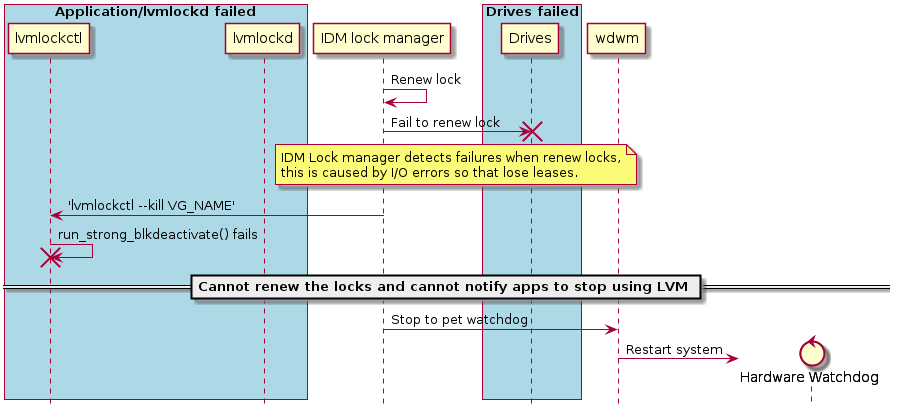


Diagram 33: Fatal failure handling with watchdog

This case we can consider using the drive's whitelist to prevent the unresponsive host from writing any data into drives. IDM lock manager uses the specific time window for retrying renewal after losing the majority and drives must use the same time window for the quiescent period. After the quiescent period, even the host reconnects with drives, the applications on the host cannot access the drives until the host can acquire IDMs again with drives. For this reason, we don’t need to restart the system.

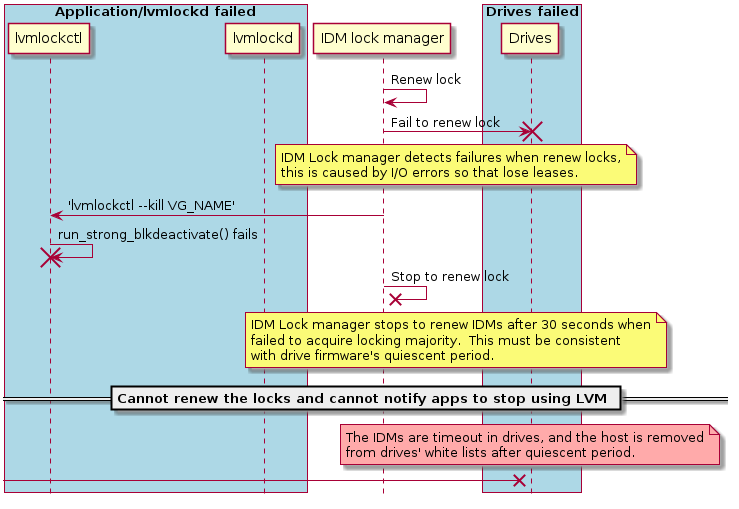


Diagram 34: Fatal failure handling with IDM whitelist

**Updated for implementation:**

White list function has been postponed to the next phase to implement in drive firmware, so current implementation cannot fence out hosts after timeout in drive firmware. See ticket: <https://github.com/Seagate/propeller/issues/88>.

When emulate fabric failure, if drives disconnect and reconnect again, the drives’ device node name and SG node name might be altered in the system, e.g. before disconnecting, a drive has name /dev/sda and /dev/sg1, after it disconnecting and reconnecting, its device name might be changed to /dev/sdb and /dev/sd2; if we want to IDM lock manager to handle this case, we need to track the drive’s unique identifier; when the IDM lock manager fails to renew the mutex or fail to find the device path, it can use the unique identifier to iterate all drives and if there have matched drive, the drive might appear in another data path, or the drive has recovered back, for either case the IDM lock manager can use the new found drive path to send SCSI commands. So this can allow the lock manager to be more reliable. This should be enhanced if have clear requirement for this:

<https://github.com/Seagate/propeller/issues/98>.

## 4.6 IDM thread vs IDM daemon

IDM lock manager should appear as a runtime entity for continuous lock renewal, and if monitor any failure and take corresponding actions for failures; in the previous section we use the naming IDM lock manager, but we don’t explicitly define the implementation form for IDM lock manager. This section proposes two forms to implement IDM lock manager: one is using a dedicated thread running in lvmlockd and another solution is creating a dedicated IDM lock manager daemon.

The first proposed solution implements IDM lock manager as a thread in lvmlockd, which is described with below details:

* IDM majority lock is implemented lvmlockd.
* IDM wrapper APIs call IDM majority lock APIs for locking related operations.
* The IDM thread needs to cooperate with IDM majority locking to handle drives failure.
* Every time when acquiring an IDM lock, the lock will be added into a dedicated lock list; and the lock will be removed from the lock list when release it.
* The IDM lock list is mainly used by IDM thread, the thread will periodically be woken up (e.g. 20s) to renewal locks ownership.
* When lvmlockd fails, since IDM thread is contained in lvmlockd as a child thread, so IDM thread cannot pet watchdog and trigger system reset. This is a difference from the conclusion in section 4.5 for lvmlockd failure.



Diagram 35: Software components with IDM threads

For the testing strategy, we can test IDM lockSCSI wrapper APIs on top of Seagate Propeller package, this is low level testing for locking operations. And we can use lvmlockd testing cases for integration testing. Since the IDM thread is resident in lvmlockd, a hard thing is to test IDM thread and majority locking, e.g. the logics for lock renewal and failure handling, etc; and adding stubs in IDM thread will let the test be more complex.

As a side topic, as we have reviewed lvmlockd flows and get to know that lvmcore layer uses flock to protect the local concurrency accessing, if we implement pthread mutex version for IDM, we have no chance to test it since the up layer has used flock for locking and low level locking doesn’t really work for local multiple instances.

The second candidate solution is to write IDM lock daemon, to understand this solution, it can be divided into below details:

* The IDM wrapper layer in lvmlockd is a very thin layer to simply send request to the lock daemon;
* The IDM daemon implements RAID failure tolerance and majority lock APIs, and the IDM daemon renews locks ownership and handles failures.
* If IDM daemon detects failure, it can kill lvmlockd and this can avoid reset the system; the reset system will be the last resort.
* Just to clarify, this solution is using almost the same architecture with sanlock/DLM with a dedicated locking daemon, though we cannot see any reduction for software architecture. In fact, the IDM daemon will be implemented much simpler than sanlock’s daemon, since host and locking lease has been moved from IDM locking daemon to drives and it doesn’t maintain the lockspace context as complex as sanlock daemon.



Diagram 36: Software components with IDM daemon

For the testing strategy, it is divided into three layers testing.

IDM SCSI wrapper APIs testing:

* The scope is the IDM SCSI wrapper APIs related testing which is based on the Seagate Propeller package.
* Test IDM’s acquiring, releasing, renewal and converting lock.
* Test IDM’s LVB.
* Test IDM’s inquiring state.
* On a host, test multithreading scenario with specific flow and random flows.
* Test two or more initiators on multiple hosts.
* Use an automatic testing framework to test cases. (pytest tests: python for C testing).

IDM majority lock testing:

* The scope is to test IDM majority locking algorithm.
* Test its functionality and reliability. To prove if the algorithm is robust or not.
* Test acquiring, releasing, renewal and converting lock.
* Test LVB.
* Test inquiring lock state.
* Test drive list maintenance.
* Test drive state machine with adding stub in daemon.
* Test random drive failures.
* Test flows for lock\_gl / lock\_vg / lock\_lv.
* Test flows with single instance and multiple instances on a host with pthread mutex.
* Test flows with single instance and multiple instances on a host with IDM.
* Test flows with single instance and multiple instances on two hosts with IDM.
* Test corner cases which are described in section “4.4 lock algorithm”.
* Timeout flow handling for app disconnection failure, IDM lock manager (thread or daemon) failure and IDM drives failure.
* Verify the failure handling with killpath and signal.
* Test watchdog handling to restart system with serious failures.
* Stress test with locking general flows.

Integration testing:

* The scope is to test the integration IDM into lvmlockd; need to cover common LVM tool usages.
* Test LVM2 with lvm2 test suites in the folder test.
* Test based on lvmlockd interfaces with adding support in the test/Makefile and corresponding test scripts: ‘check\_lvmlockd\_idm: Run tests with lvmlockd and sanlock’. At the current stage we don’t design the cases for running different lock schemes within the same cluster and only focus on the IDM scheme.
* Test with the most often used LVM tool’s commands:  
  vgcreate vgremove vgextend lvcreate lvremove lvextend  
  vgchange lvchange vgs pvcreate pvmove pvremove
* Test the failure handling with lvmlockd and lvmlockctl.
* Test all cases on the local system.
* Test all cases in the cluster system with 2~3 hosts.
* Stress test with LVM commands.

Table 4: Comparison between IDM thread and IDM daemon

|  |  |  |
| --- | --- | --- |
| Features | IDM thread in lvmlockd | IDM daemon |
| IDM SCSI wrapper APIs | Seagate Propeller package | Seagate Propeller package |
| IDM majority lock | Implemented in Seagate Propeller package, or implemented in IDM thread | Implemented in lock manager daemon |
| Membership maintenance | IDM thread | IDM daemon |
| Failure handling | Deactivate block devices  Heavily rely on watchdog reset | Deactivate block devices  Kill app with killpath and signal  Watchdog reset is the last resort |
| IDM SCSI wrapper APIs test | Seagate Propeller package | Seagate Propeller package |
| Majority lock test | lvmlcokd | IDM daemon |
| Integration test | lvmlockd / lvm2 | lvmlockd / lvm2 |
| Fault injection | Stub in lvmlockd | Stub in IDM daemon |
| Pthread mutex emulation | Cannot support due to flock is used in lvmcore | Replace IDM majority lock with pthread mutex in IDM daemon |
| Releasing | Avoid to create extra repository and the IDM lock manager is maintained in lvmlockd | Introduce new program and need to create repository for maintenance |
| License | GPLv2.1 | Can be decided by Seagate and changed at different time points |
| Overall Complexity | High  The reason is if mix two functionalities (lvm lock daemon and IDM lock daemon) into the same software, the code will be more complex by the majority locking; and it’s more difficult for maintenance.  This design might introduce risk if we consider the IDM thread is not flexible enough and can be hard to support failure handling and extend support for corner cases. | Middle  The overall complexity is middle; though this solution introduces an extra layer for IDM daemon, the daemon allows the software layers with cleaner logic and the neat code.  The IDM design is helpful for reducing complexity in IDM daemon, e.g. IDM daemon can be much simpler than sanlock daemon implementation, since host lease and lock lease have been moved to drive. |
| Design Effort | Middle  lvmlockd and IDM threads directly interact with low level locking APIs, we don’t need to take time to design extra layers for lock manage daemon.  It’s good to emphasize that the design might be complex for IDM threads, especially for failure handling. | Middle to high  Need to design the two layers of communications: between lvmlockd and IDM daemon, and between IDM daemon and lock APIs.  The benefit is the software architecture is clean with clear functionality boundary. This will deliver more benefits for implementing and maintaining the codes. |
| Test Effort | High  The test effort is divided into two parts: IDM SCSI wrapper testing and lvm2/lvmlockd integration testing.  The most difficult thing is the IDM thread has complex logic but it resides in lvmlockd, then it’s hard to test IDM thread since we cannot directly interact with IDM thread’s functions. This can introduce extra efforts to add stub for IDM thread unit testing and the testing code will be very hard to maintain. | Middle  The test effort is divided into three parts: IDM SCSI wrapper testing, lock majority and IDM daemon testing, lvm2/lvmlockd integration testing.  The testing approach would be simple with this design: we will give sufficient testing on SCSI wrapper APIs and majority locking in the IDM daemon, this can allow the IDM lock manager to be more reliable and we should savetime for integration tests with lvmlockd. |

Conclusion: based on up comparison, the IDM daemon design is preferred rather than adding IDM thread in lvmlockd. Especially, if we consider the complexity introduced by RAID and consider what’s the best way for testing, IDM daemon is the winner. Because with the IDM daemon, we have more flexibility to handle complexity introduced by RAID, can use a closer interface to test locking algorithm, and easily extend the IDM daemon to emulate drives failure with fault injection. Furthermore, from the maintainence’s view, all codes maintained in an IDM dedicated repository would be much easier than merging into lvmlockd which lives in LVM2 repository.

## 4.7 APIs

Table 5: IDM functions mapping in multiple layers

|  |  |  |  |
| --- | --- | --- | --- |
| lvmlockd lock manager interface | IDM wrapper APIs | IDM RAID APIs | IDM SCSI wrapper APIs |
| lm\_is\_running\_idm() | idm\_version() |  |  |
| lm\_add\_lockspace\_idm() | idm\_connect() |  |  |
| lm\_rem\_lockspace\_idm() | idm\_disconnect() |  |  |
|  | idm\_setup\_signal() |  |  |
|  | idm\_setup\_killpathl() |  |  |
| lm\_lock\_idm() | idm\_lock() | idm\_raid\_lock() | idm\_drive\_init() |
| idm\_drive\_lock() |
| lm\_unlock\_idm() | idm\_unlock() | idm\_raid\_unlock() | idm\_drive\_unlock() |
| idm\_drive\_destroy() |
| lm\_convert\_idm | idm\_convert() | idm\_raid\_convert\_lock() | idm\_drive\_convert\_lock() |
|  | idm\_renew():  Debugging purpose | idm\_raid\_renew\_lock() | idm\_drive\_renew\_lock() |
| idm\_drive\_break\_lock() |
|  | idm\_write\_lvb() | idm\_raid\_write\_lvb() | idm\_drive\_write\_lvb() |
|  | idm\_read\_lvb() | idm\_raid\_read\_lvb() | idm\_drive\_read\_lvb() |
| lm\_hosts\_idm() | idm\_count() | idm\_raid\_count() | idm\_drive\_count() |
|  | idm\_lock\_state():  Debugging purpose | idm\_raid\_lock\_state() | idm\_drive\_lock\_state() |
|  |  |  | idm\_drive\_whitelist():  Debugging purpose |
|  | idm\_enable\_random\_failure()  Debugging purpose |  |  |
|  | idm\_get\_random\_failure()  Debugging purpose |  |  |
|  | idm\_disable\_random\_failure()  Debugging purpose |  |  |

### 4.7.1 lvmlockd lock manager interface

int lm\_is\_running\_idm(void);

int lm\_add\_lockspace\_idm(struct lockspace \*ls, int adopt);

int lm\_rem\_lockspace\_idm(struct lockspace \*ls, int free\_vg);

int lm\_lock\_idm(struct lockspace \*ls, struct resource \*r, int mode,

struct val\_blk \*vb\_out, int adopt);

int lm\_unlock\_idm(struct lockspace \*ls, struct resource \*r,

uint32\_t r\_version,

struct val\_blk \*vb\_out, int adopt);

int lm\_convert\_idm(struct lockspace \*ls, struct resource \*r, int mode,

uint32\_t r\_version);

int lm\_hosts\_idm(struct lockspace \*ls, int notify);

### 4.7.2 IDM wrapper APIs

int idm\_version(int \*version);

int idm\_connect(int \*sock);

int idm\_disconnect(int sock);

int idm\_set\_signal(int sock, int signo);

int idm\_set\_killpath(int sock, const char \*path, char \*args);

struct idm\_lock\_id {

char vg\_uuid[UUID\_LEN];

char lv\_uuid[UUID\_LEN];

};

struct idm\_lock\_op {

uint32\_t mode;

char \*\*drives;

uint32\_t drive\_num;

int timeout; /\* -1 means unlimited timeout \*/

int quiescent;

};

int idm\_lock(int sock, int pid, struct idm\_lock\_id \*id,

struct idm\_lock\_op \*op);

int idm\_unlock(int sock, struct idm\_lock\_id \*id);

int idm\_convert(int sock, struct idm\_lock\_id \*id,

Uint32\_t mode);

int idm\_renew(int sock, struct idm\_lock\_id \*id);

int idm\_write\_lvb(int sock, int pid, struct idm\_lock\_id \*id,

char \*lvb, int lvblen);

int idm\_read\_lvb(int sock, int pid, struct idm\_lock\_id \*id,

char \*lvb, int lvblen);

int idm\_count(int sock, struct idm\_lock\_id \*id, int \*count);

struct idm\_lock\_state {

uint32\_t mode;

bool is\_timeouted;

};

int idm\_lock\_state(int sock, struct idm\_lock\_id \*id,

struct idm\_lock\_state \*state);

#define IDM\_RANDOM\_FAILURE\_ALL

#define IDM\_RANDOM\_FAILURE\_LESS\_50\_PERCENTAGE

#define IDM\_RANDOM\_FAILURE\_MORE\_50\_PERCENTAGE

int idm\_enable\_random\_failure(int sock, struct idm\_lock\_id \*id,

int failure\_rate);

int idm\_disable\_random\_failure(int sock, struct idm\_lock\_id \*id);

int idm\_get\_random\_failure(int sock, struct idm\_lock\_id \*id,

int \*failed, int \*total);

### 4.7.3 IDM RAID APIs

int idm\_raid\_lock(char \*lock\_uuid, int mode,

char \*host\_uuid, char \*proc\_id,

char \*\*drive\_list, int drive\_num,

int timeout, int quiescent);

int idm\_raid\_unlock(char \*lock\_uuid, char \*host\_uuid, char \*proc\_id,

char \*\*drive\_list, int drive\_num);

int idm\_raid\_convert\_lock(char \*lock\_uuid, int mode,

char \*host\_uuid, char \*proc\_id,

char \*\*drive\_list, int drive\_num);

int idm\_raid\_renew\_lock(char \*lock\_uuid, int mode,

char \*host\_uuid, char \*proc\_id,

char \*\*drive\_list, int drive\_num);

int idm\_raid\_write\_lvb(char \*lock\_uuid, void \*lvb, int lvb\_size,

char \*\*drive\_list, int drive\_num);

int idm\_raid\_read\_lvb(char \*lock\_uuid, void \*lvb, int lvb\_size,

char \*\*drive\_list, int drive\_num);

int idm\_raid\_count(char \*lock\_uuid, int \*count,

char \*\*drive\_list, int drive\_num);

int idm\_raid\_state(char \*lock\_uuid,

int \*mode, int \*is\_timeout,

char \*\*drive\_list, int drive\_num);

### 4.7.4 IDM SCSI wrapper APIs

int idm\_drive\_init(char \*lock\_uuid,

char \*host\_uuid, int pid,

chat \*drive);

int idm\_drive\_destroy(char \*lock\_uuid,

char \*host\_uuid, int pid,

chat \*drive);

int idm\_drive\_lock(char \*lock\_uuid, int mode,

char \*host\_uuid, int pid,

chat \*drive, int timeout,

int quiescent);

int idm\_drive\_unlock(char \*lock\_uuid,

char \*host\_uuid, int pid,

chat \*drive);

int idm\_drive\_convert\_lock(char \*lock\_uuid, int mode,

char \*host\_uuid, int pid,

chat \*drive);

int idm\_drive\_renew\_lock(char \*lock\_uuid, int mode,

char \*host\_uuid, int pid,

chat \*drive);

int idm\_drive\_break\_lock(char \*lock\_uuid, chat \*drive);

int idm\_drive\_write\_lvb(char \*lock\_uuid, void \*lvb, int lvb\_size,

chat \*drive);

int idm\_drive\_read\_lvb(char \*lock\_uuid, void \*lvb, int lvb\_size,

chat \*drive);

int idm\_drive\_count(char \*lock\_uuid, int \*count,

chat \*drive);

int idm\_drive\_lock\_state(char \*lock\_uuid,

int \*mode,

int \*is\_timeout,

chat \*drive);

int idm\_drive\_whitelist(char \*drive,

char \*\*whitelist,

int \*whitelist\_num);

Table 6: IDM functions and the required SCSI commands

|  |  |
| --- | --- |
| Function | SCSI commands |
| idm\_drive\_init() | OPERATION CODE = 89h (COMPARE AND WRITE)  LBA = MAXLBA + XXXh (TBD)  MUTEX\_OP = 001b (Mutex Init)  Resource ID = lock\_uuid (VG\_UUID << 32 | LV\_UUID)  Host ID = host\_uuid |
| idm\_drive\_destroy() | OPERATION CODE = 89h (COMPARE AND WRITE)  LBA = MAXLBA + XXXh (TBD)  MUTEX\_OP = 111b (Mutex Destroy)  Resource ID = lock\_uuid (VG\_UUID << 32 | LV\_UUID)  Host ID = host\_uuid |
| idm\_drive\_lock() | OPERATION CODE = 89h (COMPARE AND WRITE)  LBA = MAXLBA + XXXh (TBD)  MUTEX\_OP = 010b (Mutex TryLock)  Resource ID = lock\_uuid (VG\_UUID << 32 | LV\_UUID)  Host ID = host\_uuid  Class = 0h (Exclusive) or 2h (Shared) |
| idm\_drive\_unlock() | OPERATION CODE = 89h (COMPARE AND WRITE)  LBA = MAXLBA + XXXh (TBD)  MUTEX\_OP = 100b (Mutex Unlock)  Resource ID = lock\_uuid (VG\_UUID << 32 | LV\_UUID)  Host ID = host\_uuid |
| idm\_drive\_convert\_lock() | OPERATION CODE = 89h (COMPARE AND WRITE)  LBA = MAXLBA + XXXh (TBD)  MUTEX\_OP = 101b (Mutex Refresh)  Resource ID = lock\_uuid (VG\_UUID << 32 | LV\_UUID)  Host ID = host\_uuid  Class = 0h (Exclusive) or 2h (Shared)  Countdown = IDM\_DEFAULT\_TIMEOUT |
| idm\_drive\_renew\_lock() | OPERATION CODE = 89h (COMPARE AND WRITE)  LBA = MAXLBA + XXXh (TBD)  MUTEX\_OP = 101b (Mutex Refresh)  Resource ID = lock\_uuid (VG\_UUID << 32 | LV\_UUID)  Host ID = host\_uuid  Class = 0h (Exclusive) or 2h (Shared)  Countdown = IDM\_DEFAULT\_TIMEOUT |
| idm\_drive\_break\_lock() | OPERATION CODE = 89h (COMPARE AND WRITE)  LBA = MAXLBA + XXXh (TBD)  MUTEX\_OP = 110b (Mutex BreakLock)  Resource ID = lock\_uuid (VG\_UUID << 32 | LV\_UUID) |
| idm\_drive\_write\_lvb() | OPERATION CODE = 89h (COMPARE AND WRITE)  LBA = MAXLBA + XXXh (TBD)  MUTEX\_OP = 101b (Mutex Refresh)  Resource ID = lock\_uuid (VG\_UUID << 32 | LV\_UUID)  Metadata = LVB (maximum to 64 bytes) |
| idm\_drive\_read\_lvb() | OPERATION CODE = 88h (READ (16))  LBA = MAXLBA + XXXh (TBD)  Issue: The Resource ID is only included in IDM Read Status Payload, but how to specify lock UUID when send READ command. Should use LBA to specify lock UUID? |
| idm\_drive\_count() | OPERATION CODE = 88h (READ (16))  LBA = MAXLBA + XXXh (TBD)  Issue 1: The Resource ID is only included in IDM Read Status Payload, but how to specify lock UUID when send READ command. Should use LBA to specify lock UUID?  Issue 2: How to read out the owner count when multiple hosts acquiring the same IDM? |
| idm\_drive\_lock\_state() | OPERATION CODE = 88h (READ (16))  LBA = MAXLBA + XXXh (TBD)  Issue 1: The Resource ID is only included in IDM Read Status Payload, but how to specify lock UUID when send READ command. Should use LBA to specify lock UUID?  Issue 2: The lock mode (exclusive, shareable, un) and Timeout use the same state field. Thus we cannot get out the complete info for lock state.  Issue 3: State = 0101h (Locked), does it mean exclusive locked? |
| idm\_drive\_whitelist() | TBD. |

**Updated for implementation:**  
When implement the IDM SCSI wrapper API, for better performance to support multi issue SCSI commands, introduced new APIs to support async mode:

int idm\_drive\_lock\_async(char \*lock\_id, int mode, char \*host\_id,

char \*drive, uint64\_t timeout, uint64\_t \*handle);

int idm\_drive\_unlock\_async(char \*lock\_id, int mode, char \*host\_id,

char \*lvb, int lvb\_size, char \*drive,

uint64\_t \*handle);

int idm\_drive\_convert\_lock\_async(char \*lock\_id, int mode, char \*host\_id,

char \*drive, uint64\_t timeout,

uint64\_t \*handle);

int idm\_drive\_renew\_lock\_async(char \*lock\_id, int mode, char \*host\_id,

char \*drive, uint64\_t timeout, uint64\_t \*handle);

int idm\_drive\_break\_lock\_async(char \*lock\_id, int mode, char \*host\_id,

char \*drive, uint64\_t timeout, uint64\_t \*handle);

int idm\_drive\_write\_lvb\_async(char \*lock\_id, char \*host\_id,

char \*lvb, int lvb\_size,

char \*drive, uint64\_t \*handle);

int idm\_drive\_read\_lvb\_async(char \*lock\_id, char \*host\_id,

char \*drive, uint64\_t \*handle);

int idm\_drive\_read\_lvb\_async\_result(uint64\_t handle, char \*lvb, int lvb\_size,

int \*result);

int idm\_drive\_lock\_count\_async(char \*lock\_id, char \*host\_id,

char \*drive, uint64\_t \*handle);

int idm\_drive\_lock\_count\_async\_result(uint64\_t handle, int \*count,

int \*self, int \*result);

int idm\_drive\_lock\_mode\_async(char \*lock\_id, char \*drive, uint64\_t \*handle);

int idm\_drive\_lock\_mode\_async\_result(uint64\_t handle, int \*mode, int \*result);

int idm\_drive\_async\_result(uint64\_t handle, int \*result);

The drive firmware can support maximum to 128 mutex entries, this limitation is described in the ticket: <https://github.com/Seagate/propeller/issues/105>. If the mutex has been unlocked, the mutex still will occupy the memory space in drive. When detected the error -ENOMEM reported by the mutex firmware, needs to invoke IDM SCSI command to release the mutex which is timeout or in ‘unlock’ state. So two new APIs to handle the case when memory is used out, idm\_drive\_read\_group() is used to read out all mutex from drives, and idm\_drive\_destroy() is to destroy mutex by the given lock ID.

int idm\_drive\_read\_group(char \*drive, struct idm\_info \*\*info\_ptr, int \*info\_num);

int idm\_drive\_destroy(char \*lock\_id, int mode, char \*host\_id, char \*drive);

# 5 Development and test plan

The development and test plan proposes four stages, every stage tries to avoid dependency within components and defines the main achievement for the milestone. The plan is summarized in the table.

Table 7: Development and test plan

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Stages | Description | Milestone | Main working items | Estimation  (days / 1 person) |
| Stage 1 | Parallel developing for  IDM drive firmware,  IDM SCSI wrapper APIs,  and IDM lock manager with pthread mutex emulation  29 workdays / 1 person | Pass unit test for IDM SCSI wrapper APIs  Pass unit test for IDM lock manager with pthread mutex backend | IDM SCSI wrapper APIs development | 4 |
| IDM lock manager with pthread mutex development | 13 |
| Unit test for IDM lock manager with pthread mutex emulation | 12 |
| Stage 2 | Integration IDM lock manager with IDM SCSI wrapper APIs & Testing  **Dependency: IDM Drive firmware (est 10 days)**  12 workdays / 1 person | Pass integration test for IDM lock manager and IDM SCSI wrapper APIs | Test for IDM SCSI wrapper APIs | 7 |
| Integration test for IDM lock manager and IDM SCSI wrapper APIs | 5 |
| Stage 3 | Integration IDM into the LVM whole stack  20 workdays / 1 person | Pass integration test and stress test for LVM whole stack with IDM locking scheme | lvmlockd wrapper layer development | 3 |
| Smoke testing for lvmlockd and IDM lock manager (pthread mutex backend) | 2 |
| Enable IDM testing in LVM test framework and stress testing cases development | 5 |
| Integration test and stress test (desktop scale testing) for whole stack | 10 |
| Stage 4 | Failure inducement testing  15 workdays / 1 person | Pass failure inducement testing for LVM whole stack with desktop scale | Develop for failure inducement cases: host failure, drive failure caused by disconnection and fabric failure | 5 |
| Failure inducement test (desktop scale testing with 1 drive and 2 drives) | 10 |

## 5.1 Stage 1: Parallel development

Stage 1 is used for parallel software components development. From the start point, we want to avoid dependency within components and improve efficiency, two main working threads will be kicked off at the same time:

* IDM SCSI wrapper APIs can be developed separately as one developing progress;
* The second working thread is for IDM lock manager development. To avoid the dependency between IDM lock manager and IDM SCSI wrapper APIs, we can use pthread mutex as a backend to support the first stage’s IDM lock manager development.

The main achievement at this stage contains:

* IDM SCSI wrapper APIs
  + IDM SCSI wrapper APIs development is completed in 4 work days with 1 person; all the APIs are defined in section 4.7.4 IDM SCSI wrapper APIs and should be finished in this stage.
* IDM lock manager
  + IDM lock manager with pthread mutex backend will take 13 work days / 1 person;
  + Developing the unit test for IDM lock manager will take 9 work days / 1 person;
  + Take 3 work days for testing IDM lock manager with pthread mutex backend.
  + If test case failures have been found, need to locate if the failures are caused by pthread mutex faulty or by the bugs in the code; pthread mutex faulty should be statement in test reporting for later’s checking (e.g. for regression testing and comparison), and need fix the found bugs.

At this stage, the milestone is passing unit test separately for IDM SCSI wrapper APIs and for IDM lock manager with pthread mutex backend. The estimated total workload is **29 work days / one person**; if you invest more engineers for development and testing, the shortest path is to take **16 work days**, which is shown in the timeline diagram in section 5.3 Timeline.

## 5.2 Stage 2: Integration & test for IDM lock manager

This stage will complete integration for IDM lock manager and IDM SCSI wrapper APIs; this stage works as a bridge between stage 1 and stage 3, so it is important to achieve qualified software with testing at this stage and we have more confidence to move forward to whole stack integration.

IDM drive firmware is prerequisites for this phase and , it’s assumed to be completed in 10 work days, this part needs to be estimated by Seagate for more accurate workloads.

The main achievement at this stage contains:

* IDM SCSI wrapper APIs
  + Developing the unit test cases for IDM SCSI wrapper APIs is estimated as 4 work days / 1 person; and will take 3 days to pass unit test on IDM SCSI wrapper APIs.
* IDM lock manager
  + Integrate IDM lock manager with IDM SCSI wrapper APIs; and complete the integration test for IDM lock manager and IDM SCSI wrapper APIs.

At this stage, the milestone is passing the test for the IDM lock manager with IDM SCSI wrapper APIs as backend; this means the IDM lock manager (and its majority locking algorithm) is reliable to work on IDM drives. The main effort is for the testing; the estimated workload is **12 work days / one person**.

## 5.3 Stage 3: Integration & testing for the whole stack

This stage will accomplish all development and test for IDM in LVM. At the end, the IDM locking scheme is ready for use in LVM tools.

The main achievements are:

* lvmlockd
  + lvmlockd development with IDM wrapper is completed.
  + Completion smoke testing for lvmlockd and IDM lock manager (pthread mutex backend).
* Whole stack
  + Integration lvmlockd with IDM lock manager and low levelSCSI wrapper APIs;
  + Completion of the LVM self contained test and stress test on multiple virtual machines.

At this stage, the milestone is passing the integration test on the local PC and on multiple VMs. The estimated workload is **20 work days / one person**; we can use the overlap time with stages 1/2 for lvmlockd development and test case development, thus can avoid dependency and the shortest path is to take **10 work days** between the two milestones in stage 2 and stage 3.

## 5.4 Stage 4: Failure inducement testing

This stage will finish failure inducement testing; the testing can prove the software quality has reached a certain level for robustness and reliability and is ready for big scale testing in the cluster.

The failure inducement testing contains:

* A test for a host failure whilst owning lock: the host can be fenced out with system reboot, or can rely on drive’s whitelist after quencient period expiration.
* A test for drive failure whilst granting lock: the host can continue its lease after drives disconnect from host and recovery back before lock’s timeout; otherwise, the host should be fenced out after lock’s timeout expiration.
* A test for fabric failure: for a fabric failure, the host can continue its lease after fabric recovery back and drives are reconnected without lock’s timeout; otherwise, the host can be fenced out after lock’s timeout expiration, no matter if the fabric can recover back or not.

At this stage, the milestone is failure inducement testing on the desktop scope for the local testing and the multiple VMs testing. The estimated workload is **15 work days / one person**; which includes 5 work days for test case development and 10 work days for testing and fixing bugs.

## 5.5 Timeline

The timeline analysis tries to get rid of the unnecessary dependencies and find out the shortest paths to achieve the milestones for the three stages.

For easier understanding the timeline diagram, it uses below legends:

* Purple box: the assumed development workload for drive firmware
* Green box: development workload
* Orange box: test workload
* Black rhombus: brief achievements and milestones

The vertical red line is used to indicate the major milestone and it crosses the milestones tagged with black rhombus; it also presents the boundary between different stages. Two vertical red lines should be noted:

* The red line at day 16: it presents the end of stage 1 and a division between stage 1 and stage 2;
* The red line at day 21: it presents the end of stage 2 and a division between stage 2 and stage 3.
* The red line at day 31: it presents the end of stage 3 and a division between stage 3 and stage 4.

In theory, this project can be accomplished in **41 work days** with parallel development and test.

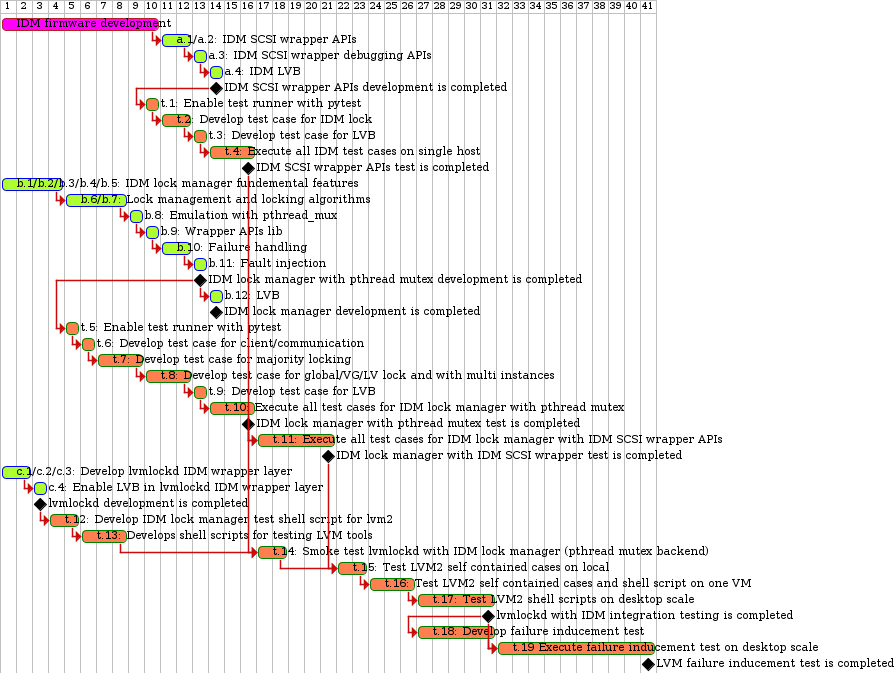


Diagram 38: Timeline for development and test plan

# 6 Development and test plan (if IDM firmware is delayed)

The development and test plan proposes five stages, every stage tries to avoid dependency within components and defines the main achievement for the milestone. The plan is summarized in the table.

Table 8: Development and test plan

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Stages | Description | Milestone | Main working items | Estimation  (days / 1 person) |
| Stage 1 | IDM lock manager with pthread mutex emulation  25 workdays / 1 person | Pass unit test for IDM lock manager with pthread mutex backend  Note: Implementation of log framework based on syslog | IDM lock manager with pthread mutex development | 13 |
| Unit test for IDM lock manager with pthread mutex emulation | 12 |
| Stage 2 | Integration IDM lock manager (with pthread mutex) into the LVM whole stack  13 workdays / 1 person | Pass integration test and stress test for LVM whole stack with pthread mutex backend | lvmlockd wrapper layer development | 3 |
| Smoke testing for lvmlockd and IDM lock manager (pthread mutex backend) | 2 |
| Enable IDM testing in LVM test framework and stress testing cases development | 5 |
| Integration test and stress test (**with single host**) for whole stack (**split1**) | 3 |
| Stage 3 | IDM SCSI wrapper APIs development  8 workdays / 1 person | Develop IDM SCSI wrapper APIs and test cases, we can use this period to give more time buffer for for IDM drive firmware | IDM SCSI wrapper APIs development | 4 |
| Develop test cases for IDM SCSI wrapper APIs (**split2**) | 4 |
| Stage 4 | Integration IDM lock manager with IDM SCSI wrapper APIs & Testing  **Dependency: IDM Drive firmware**  10 workdays / 1 person | Pass unit test for IDM SCSI wrapper APIs  Pass integration test for IDM lock manager and IDM SCSI wrapper APIs | **Enable lab for remote accessing IDM drives (new)** | **2** |
| Test for IDM SCSI wrapper APIs (**split2**) | 3 |
| Integration test for IDM lock manager and IDM SCSI wrapper APIs | **5** |
| Stage 5 | Integration IDM into the LVM whole stack  7 workdays / 1 person | Pass integration test and stress test for LVM whole stack with IDM locking scheme | Integration test and stress test (desktop scale testing) for whole stack (**split1**) | 7 |
| Stage 6 | Failure inducement testing  15 workdays / 1 person | Pass failure inducement testing for LVM whole stack with desktop scale | **Develop for failure inducement cases: host failure, drive failure caused by disconnection and fabric failure** | **5** |
| Failure inducement test (desktop scale testing with 1 drive and 2 drives) | 10 |

## 6.1 Stage 1: IDM lock manager development with pthread mutex emulation

If the IDM firmware needs to take longer to deliver, Stage 1 would be focusing on IDM lock manager development with pthread mutex emulation. Connected between stage 1 and 2, we can do best to achieve the milestone for enabling the whole stack by using pthread mutex as the backend.

The main achievement at this stage contains:

* IDM lock manager
  + IDM lock manager with pthread mutex backend will take 13 work days / 1 person;
  + Developing the unit test for IDM lock manager will take 9 work days / 1 person;
  + Take 3 work days for testing IDM lock manager with pthread mutex backend.
  + If test case failures have been found, need to locate if the failures are caused by pthread mutex faulty or by the bugs in the code; pthread mutex faulty should be statement in test reporting for later’s checking (e.g. for regression testing and comparison), and need fix the found bugs.
  + Implement the log system based on syslog for analysis and debugging.

At this stage, the milestone is passing unit test for IDM lock manager with pthread mutex backend. The estimated total workload is **25 work days / one person**.

## 6.2 Stage 2: Whole stack enabling with pthread mutex backend

This stage will integrate IDM lock manager with lvmlock, so the whole stack with pthread mutex backend is ready for use in LVM tools. Afterwards, we can use stage 2 as a good base to move forward for integration IDM firmware.

The main achievements are:

* lvmlockd
  + lvmlockd development with IDM wrapper is completed for 3 work days / 1 person
  + Completion smoke testing for lvmlockd and IDM lock manager (pthread mutex backend) for 2 work days / 1 person.
* Whole stack
  + Completion of the LVM self contained test for 5 work days / 1 person and stress test on a single virtual machine for 3 days / 1 person.

At this stage, the milestone is passing the integration test on the local PC and on a single VMs. The estimated workload is **13 work days / one person**.

## 6.3 Stage 3: IDM SCSI wrapper APIs development

This stage will develop IDM SCSI wrapper APIs and write the test cases for wrapper APIs.

The main achievements are:

* IDM SCSI wrapper APIs
  + IDM SCSI wrapper APIs development is completed in 4 work days with 1 person; all the APIs are defined in section 4.7.4 IDM SCSI wrapper APIs and should be finished in this stage.
  + Developing the unit test cases for IDM SCSI wrapper APIs is estimated as 4 work days / 1 person.

At this stage, the milestone is IDM SCSI wrapper APIs and corresponding test cases are ready and wait for IDM drive firmware to start testing. The estimated workload is **8 work days / one person**.

## 6.4 Stage 4: Integration IDM lock manager with IDM SCSI wrapper APIs

This stage will complete integration for IDM lock manager and IDM SCSI wrapper APIs. IDM drive firmware is prerequisites for this phase.

The main achievement at this stage contains:

* IDM SCSI wrapper APIs
  + Enable lab for remote accessing IDM drives for 2 work days;
  + And take 3 days to pass the unit test on IDM SCSI wrapper APIs.
* IDM lock manager
  + Integrate IDM lock manager with IDM SCSI wrapper APIs; and complete the integration test for IDM lock manager and IDM SCSI wrapper APIs for 5 work days.

At this stage, the milestone is passing the test for the IDM lock manager with IDM SCSI wrapper APIs as backend; this means the IDM lock manager (and its majority locking algorithm) is reliable to work on IDM drives. The main effort is for the testing; the estimated workload is **10 work days / one person**.

## 6.5 Stage 5: Testing the whole stack with IDM SCSI wrapper APIs

This stage will accomplish the whole stack test for IDM in LVM. At the end, the IDM locking scheme is ready for use in LVM tools.

The main achievements are:

* Whole stack
  + Completion of the LVM self contained test and stress test with IDM SCSI wrapper APIs (desktop scale and multiple virtual machines) .

At this stage, the milestone is passing the integration test on the local PC and on multiple VMs. The estimated workload is **7 work days / one person**.

## 6.6 Stage 6: Failure inducement testing

This stage will finish failure inducement testing; the testing can prove the software quality has reached a certain level for robustness and reliability and is ready for big scale testing in the cluster.

The failure inducement testing contains:

* A test for a host failure whilst owning lock: the host can be fenced out with system reboot, or can rely on drive’s whitelist after quencient period expiration.
* A test for drive failure whilst granting lock: the host can continue its lease after drives disconnect from host and recovery back before lock’s timeout; otherwise, the host should be fenced out after lock’s timeout expiration.
* A test for fabric failure: for a fabric failure, the host can continue its lease after fabric recovery back and drives are reconnected without lock’s timeout; otherwise, the host can be fenced out after lock’s timeout expiration, no matter if the fabric can recover back or not.

At this stage, the milestone is failure inducement testing on the desktop scope for the local testing and the multiple VMs testing. The estimated workload is **15 work days / one person**; which includes 5 work days for test case development and 10 work days for testing and fixing bugs.

## 6.7 Conclusion

In theory, this project can be accomplished in **78 work days** with serialized development and testing.

The first 38 days is used to focus on developing the whole stack with the pthread mutex backend, and finish the limited integration testing on a single host. We can take extra 8 days in stage 3 to finish IDM SCSI wrapper APIs and develop corresponding test cases. So we can have the total 46 days waiting for the IDM drive firmware to be ready.

After IDM drive firmware is ready, the last 32 days are used to focus on integration and stress testing with IDM firmware backend.

Compared to Chapter 5 which is planned for 76 days / one person, this chapter will take 2 days longer, the extra workload is introduced by: the stage 3 “Enable lab for remote accessing IDM drives: 2 work days”.

# Appendix A: Workload estimation

To achieve more accurate workload estimation, an excel table is created to track the detailed list for development and test items; the workload estimation are concluded from the table: <https://docs.google.com/spreadsheets/d/1uUXaIdq0nHnP2fOw0_dJkQpDYDlhjRX5nOuyHecNV1s/edit#gid=0>

## A.1 Development workload estimation

Table 8: Development workload estimation

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Work Item | Component | Features | Priority | Precondition | Estimation  (days / 1 person) | |
| a.1 | IDM SCSI wrapper APIs | Build system | High | n/a | 0.25 | Total:  3.5 |
| a.2 | Initialization/deinitialization,  Acquiringing/releasing, Renewal/breaking, Converting, Reading user count | High | a.1 | 1.5 |
| a.3 | Debugging feature: Dump IDM state; Dump whitelist | High | a.2 | 1 |
| a.4 | LVB | Low | a.2 | 0.5 |
| b.1 | IDM Lock manager | Build system | High | n/a | 0.25 | Total:  ~ 14 |
| b.2 | Main loop flow | High | b.1 | 1 |
| b.3 | Log system | High | b.2 | 1 |
| b.4 | Client management | High | b.2 | 1 |
| b.5 | Communication management: Define message format, parse message and composite message, idm\_version() | High | b.4 | 1 |
| b.6 | Lockspace and lock management: Lockspace thread, Lock allocation and releasing, Lockspace and lock state dumping | High | b.5 | 1.5 |
| b.7 | Majority locking algorithm: Acquiringing/releasing, Renewal/breaking, Converting, Reading user count,  Reading lock state. | High | b.6 | 2.5 |
| b.8 | Emulation with pthread mutex | High | b.7 | 1 |
| b.9 | Wrapper API lib | High | b.7 | 1 |
| b.10 | Failure handling | Middle | b.7 | 2 |
| b.11 | Fault injection | Middle | b.7 | 1 |
| b.12 | LVB | Low | b.7 | 0.5 |
| c.1 | lvmlockd integration | Build system | High | n/a | 0.1 | Total:  2.5 ~ 3 |
| c.2 | Lock manager wrapper layer: Lockspace, Acquiring/releasing lock, Converting lock mode, Inquiring host count | High | b.9  c.1 | 1.5 |
| c.3 | Global lock specific handling: Support for global lock specific flow, and create a dedicated lockspace for global lock. | High | c.2 | 0.5 |
| c.4 | LVB | Low | c.2 | 0.5 |

## A.2 Test workload estimation

Table 9: Test workload estimation

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Work Item | Component | Features | Priority | Precondition | Estimation  (days / 1 person) | |
| t.1 | IDM SCSI wrapper APIs | Test framework with testing runner (pytest with C program) | High | n/a | 1 | Total:  7 |
| t.2 | Develop test case (refer to IDs 2~27 in spreadsheet page1): Initialization/deinitialization,  Acquiringing/releasing, Renewal/breaking, Converting, Reading user count | High | a.2  t.1 | 2 |
| t.3 | Develop test cases for LVB | Middle | a.4 t.1 | 1 |
| t.4 | Execute all test cases on single host | High | t.2 t.3 | 3 |
| t.5 | IDM Lock manager | Test framework with testing runner (pytest with C program): reuse t.1 | High | n/a | 0 | Total:  15 |
| t.6 | Develop test case (refer to IDs 2~3 in spreadsheet page2):  Client connection test  Communication test | High | b.5  t.5 | 1 |
| t.7 | Develop test case (refer to IDs 4~29 in spreadsheet page2):  Lockspace test  Majority locking test | High | b.9  t.5 | 3 |
| t.8 | Develop test case (refer to IDs 30~39 in spreadsheet page2):  Global/VG/LV lock testing  Develop test case (refer to IDs 42~47 in spreadsheet page2):  Multi instances testing for global/VG/LV lock | High | b.9 t.5 | 3 |
| t.9 | Develop test case (IDs 40~41 in spreadsheet page2):  LVB testing | Low | b.12  t.8 | 1 |
| t.10 | Execute all test cases with pthread mutex emulation, and sort out the failed test cases which are caused by pthread mutex faulty. | High | b.9  t.8 | 3 |
| t.11 | Execute all test cases with integration test with IDM lock manager / IDM SCSI wrapper APIs / drive firmware | High | t.9  t.4  a.4 | 4 |
| t.12 | lvmlockd and lvm2 testing | Develop shell script for deployment idm lock scheme and integrate into lvm/test/shell folder | High | n/a | 2 | Total: 15 |
| t.13 | Develop shell test case for using LVM tools to create/delete VG, create/delete LV, activate/deactivate LV, etc. Finally, the shell script can run stress testing on multiple hosts. | High | n/a | 3 |
| t.14 | Test LVM2 self contained cases on local PC with IDM drives. | High | c.4  t.11 | 2 |
| t.15 | Using KVM/QEMU to setup a virtual host with Debian filesystem, VM installs iscsi-initiator and can send commands to IDM drives. Test LVM2 self contained cases and developed shell test cases on the VM. | High | c.4  t.11 | 3 |
| t.16 | Using KVM/QEMU to set up 3 virtual hosts and execute shell script for global/VG/VL lock stress testing. | High | c.4  t.11 | 5 |

# Appendix B: Open issues

## B.1 Adoption orphan lock

If lvmlockd has failed, lvmlockd can adopt orphans from the lock manager. When creating a connection between lvmlockd and IDM lock manager, lvmlockd needs to tell IDM lock manager to keep orphan locks even if lvmlockd is disconnected. When the next time the lvmlockd is relaunched, lvmlockd can send a request to ask IDM lock manager to return the complete contexts for lockspace and locks, lvmlockd compares with device mappers so can know which VGs are in used and which VGs have been deactivated during lvmlockd’s temporary exiting. Finally, lvmlockd can adopt the orphans and recover the lockspace only for activated block devices.

The latest lvmlockd code always returns failure for adoption, since the commit 117160b27e51 ("Remove lvmetad") has been merged into lvm2 git repository, lvmetad is removed. As a side effect, the function get\_lockd\_vgs() always returns -1, and thus it's no chance for lvmlockd to know the activated VGs and cannot adopt orphan locks after the lvmlockd exits abnormal and relaunch.

Feedback from mailing list:

On Thu, Jan 09, 2020 at 04:19:53PM +0800, Leo Yan wrote:

> Hi all,

>

> Since the commit 117160b27e51 ("Remove lvmetad") has been merged into

> lvm2 git repository, lvmetad is removed.

>

> As side effect, in lvmlockd the function get\_lockd\_vgs() returns -1.

> Thus it's no chance for lvmlockd to adopt orphan locks after the

> lvmlockd exits abnormal and relaunch. If we connect with the sanlock

> lock manager, its daemon keeps running on the host and renew all

> locks, this would be safe for the in used LV on the host; but on the

> other hand, lvmlockd cannot adopt the locks, thus we have no chance to

> release related LVM locks anymore.

>

> For this case, I understand we can release locks by directly use low

> level sanlock commands, just wander if we have more smooth approach for

> lvmlockd to adopt locks and release the locks gracefully after lvmlockd

> failure? Or do I miss anything?

You are correct. I have not yet replaced the lock-adoption code in

lvmlockd because it was not clear that anyone was using it. I'm happy

to hear that it was useful for you, and sorry that it's been disabled.

I hope to have it replaced in the coming year, and would be happy to

help anyone interested in working on it.

Dave

Resolution: adoption orphan lock is facilitated for handling lvmlockd failure when lvmlockd is relaunched. LVM maintainer David Teigland gave the reason for not replacing the lock-adoption code is “it was not clear that anyone was using it”, this means lvmlockd is mature enough and it’s hard to fail, so usually the lock adoption code is not used.

Since the design proposes to add a shim layer to hook IDM lock manager into lvmlockd, this doesn’t not hurt the stability of lvmlockd and lvmlockd will be stable as before. Thus adoption orphan lock can be a low priority for IDM as well and suggest monitoring the mailing list for the latest development status. And until we find an usage case for adoption lock, we can restart related work for it.

## B.2 IDM state machine

The IDM state machine can be directly transferred from “Uninitialized” state to “Locked” state. This can lead to an error with the flow:

Step 1: Call idm\_drive\_init() and return success;

Step 2: Call idm\_drive\_lock() and returns error with busy.

Since in step 1 the IDM’s state machine has been transformed from “Uninitialized” state to “Locked” state, so in step 2 when explicitly acquire the IDM, the state machine will be transferred from “Locked” state to “Locked” state and reports failure.

Resolution: suggest to add an extra “Initialized” state; so in step 1 when call idm\_drive\_init() the IDM state will be transformed from “Uninitialized” state to “Initialized” state, in step 2 when acquire lock the state will be transformed from “Initialized” state to “Locked” state and return success.

This issue should be resolved in stage 1 when developing drive firmware and this is prerequisites for IDM SCSI wrapper APIs.

## B.3 SCSI command

SCSI commands need to be clarified and extended so can support upper layer’s requirement.

The first question is how to specify lock UUID when reading IDM data? For the extended READ (16) command, we need to specify LBA to read out locking payload. There have three functions need to read out IDM content:

idm\_drive\_read\_lvb() / idm\_drive\_count() / idm\_drive\_lock\_state()

Resolution: IDM specification doesn’t explain what’s the value for LBA and how to specify lock UUID. One candidate solution is to add new mutex operation ‘Mutex Inquire’ for the Extended COMPARE AND WRITE command, so firstly send COMPARE AND WRITE command with mutex operation ‘Mutex Inquire’ and tell drive firmware which IDM we are interested in, and then use the extended READ(16) command to read out the IDM state. Furthermore, the drive firmware needs to handle the concurrency reading, e.g. two hosts inquire IDM state with different lock UUIDs concurrently.

The second question is the read data format. In the read status payload, the ‘state’ field is used to present lock mode (Locked, unlocked, Multiple Shared Locks) and timeout. This mixes two attributions lock mode and timeout, thus it cannot clearly express the state (e.g an IDM is locked but it’s timeout).

Resolution: Suggest divide the ‘state’ field divide into two sub fields, e.g. bit 0-3 is used for lock mode, now support Exclusive locked, Shareable locked, or Unlocked; later can extend to support more modes as needed; bit 4-7 is used to indicate is the IDM has been timeout or not.

The third question is for reading out white list from drive firmware. It’s good to support this feature so it’s easier for debugging any potential issues.

Resolution: This is to be decided and there is no information in the IDM specification. Suggest firstly send a write command to notify drive firmware for request reading white list, then the next step uses READ(16) command to read out payload, in the payload it needs to indicate is the white list reading is ended or not; if yes, stop to send READ(16) command, otherwise, continue to send READ(16) commands until receive the payload with ending flag for while list.

The fourth question is missing the quiescent period for white list in the drive firmware. The quiescent period can be supported for every IDM, or it can be a global variable. So far the APIs are designed to set quiescent period per IDM.

Resolution: This is to be decided and there is no information in the IDM specification. Suggest to pass a quiescent period for acquiring an IDM when sending Extended COMPARE AND WRITE command. IDM firmware needs to wait for granting IDM to new hosts until the expiration for IDM timeout plus quiescent period.

The SCSI commands should be resolved in stage 1 when developing drive firmware and this is prerequisites for IDM SCSI wrapper APIs.

## B.4 Retrieve PV list

We need to decide how to retrieve PV list and pass from lvmcore to lvmlockd and IDM lock manager. In lvmcore, we can create a new function to extract PV list from VG:

struct pv\_data {

char name[128];

uuid [64];

list;

}

/\* Create PV list for VG \*/

dm\_list\_iterate\_items(pvl, &vg->pvs) {

char pv\_uuid[64];

pv = pvl->pv;

new\_pv = malloc(sizeof(struct pv\_data));

strncpy(new\_pv->name, pv\_dev\_name(pv), 128);

pv\_uuid[0]='\0';

if (!id\_write\_format(&pv->id, pv\_uuid, sizeof(pv\_uuid)))

stack;

memcpy(new\_pv->uuid, pv\_uuid, 64);

add\_list(&new\_pv->list, xxxx);

}

/\* Create PV list for LV \*/

dm\_list\_iterate\_items(pvl, &vg->pvs) {

dm\_list\_iterate\_items(pvseg, &pv->segments) {

char pv\_uuid[64];

pv = pvl->pv;

if (strcmp(lv\_name, pvseg->lvseg->lv->name)

continue;

new\_pv = malloc(sizeof(struct pv\_data));

strncpy(new\_pv->name, pv\_dev\_name(pv), 128);

pv\_uuid[0]='\0';

if (!id\_write\_format(&pv->id, pv\_uuid, sizeof(pv\_uuid)))

stack;

memcpy(new\_pv->uuid, pv\_uuid, 64);

add\_list(&new\_pv->list, xxxx);

}

}