Linux Multipathing

Edward Goggin
EMC Corporation
egoggin@emc.com

Alasdair Kergon
Red Hat
agk@redhat.com

Christophe Varoqui

christophe.varoqui@free.fr

David Olien
OSDL
dmo@osdl.org

Abstract

Linux multipathing provides io failover and path load sharing for multipathed block devices. In this paper, we provide an overview of the current device mapper based multipathing capability and describe Enterprise level requirements for future multipathing enhancements. We describe the interaction amongst kernel multipathing modules, user mode multipathing tools, hotplug, udev, and kpartx components when considering use cases. Use cases include path and logical unit re-configuration, partition management, and path failover for both active-active and active-passive generic storage systems. We also describe lessons learned during testing the MD scheme on high-end storage.

1 Introduction

Multipathing provides the host-side logic to transparently utilize the multiple paths of a redundant network to provide highly available and higher bandwidth connectivity between hosts and block level devices. Similar to how TCP/IP re-routes network traffic between 2 hosts, multipathing re-routes block io to an alternate path in the event of one or more path connectivity failures. Multipathing also transparently shares io load across the multiple paths to the same block device.

The history of Linux multipathing goes back at least 3 years and offers a variety of different architectures. The multipathing personality of the multidisk driver first provided block device multipathing in the 2.4.17 kernel. The Qlogic FC HBA driver has provided multipathing across Qlogic FC initiators since 2002. Storage system OEM vendors like IBM, Hitachi, HP, Fujitsu, and EMC have provided multipathing solutions for Linux for several years. Strictly software vendors like Veritas also provide Linux multipathing solutions.

In this paper, we describe the current 2.6 Linux kernel multipathing solution built around the kernel's device mapper block io framework and consider possible enhancements. We first describe the high level architecture, focusing on both control and data flows. We then describe the key components of the new architecture residing in both user and kernel space. This is followed by a description of the interaction amongst these components and other user/kernel components when considering sev-

eral key use cases. We then describe several outstanding architectural issues related to the multipathing architecture.

2 Architecture Overview

This chapter describes the overall architecture of Linux multipathing, focusing on the control and data paths spanning both user and kernel space multipathing components. Figure 1 is a block diagram of the kernel and user components that support volume management and multipath management.

Linux multipathing provides path failover and path load sharing amongst the set of redundant physical paths between a Linux host and a block device. Linux multipathing services are applicable to all block type devices, (e.g., SCSI, IDE, USB, LOOP, NBD). While the notion of what constitutes a path may vary significantly across block device types, for the purpose of this paper, we consider only the SCSI upper level protocol or session layer definition of a path—that is, one defined solely by its endpoints and thereby indifferent to the actual transport and network routing utilized between endpoints. A SCSI physical path is defined solely by the unique combination of a SCSI initiator and a SCSI target, whether using iSCSI, Fiber Channel transport, RDMA, or serial/parallel SCSI. Furthermore, since SCSI targets typically support multiple devices, a logical path is defined as the physical path to a particular logical device managed by a particular SCSI target. For SCSI, multiple logical paths, one for each different SCSI logical unit, may share the same physical path.

For the remainder of this paper, "physical path" refers to the unique combination of a SCSI initiator and a SCSI target, "device" refers to a SCSI logical unit, and a "path" or logical path

refers to the logical connection along a physical path to a particular device.

The multipath architecture provides a clean separation of policy and mechanism, a highly modular design, a framework to accommodate extending to new storage systems, and well defined interfaces abstracting implementation details.

An overall philosophy of isolating mechanism in kernel resident code has led to the creation of several kernel resident frameworks utilized by many products including multipathing. A direct result of this approach has led to the placement of a significant portion of the multipathing code in user space and to the avoidance of a monolithic kernel resident multipathing implementation. For example, while actual path failover and path load sharing take place within kernel resident components, path discovery, path configuration, and path testing are done in user space.

Key multipathing components utilize frameworks in order to benefit from code sharing and to facilitate extendibility to new hardware. Both kernel and user space multipathing frameworks facilitate the extension of multipathing services to new storage system types, storage systems of currently supported types for new vendors, and new storage system models for currently supported vendor storage systems.

The device mapper is the foundation of the multipathing architecture. The device mapper provides a highly modular framework for stacking block device filter drivers in the kernel and communicating with these drivers from user space through a well defined libdevmapper API. Automated user space resident device/path discovery and monitoring components continually push configuration and policy information into the device mapper's multipathing filter driver and pull configuration and path state information from this driver.
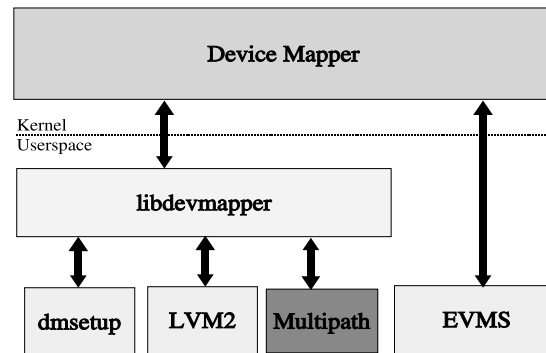
Userspace Architecture

Figure 1: Overall architecture

The primary goals of the multipathing driver are to retry a failed block read/write io on an alternate path in the event of an io failure and to distribute io load across a set of paths. How each goal is achieved is controllable from user space by associating path failover and load sharing policy information on a per device basis.

It should also be understood that the multipath device mapper target driver and several multipathing sub-components are the only multipath cognizant kernel resident components in the linux kernel.

3   Component Modules

The following sections describe the kernel and user mode components of the Linux multipathing implementation, and how those components interact.

3.1   Kernel Modules

Figure 2 is a block diagram of the kernel device mapper. Included in the diagram are components used to support volume management as well as the multipath system. The primary kernel components of the multipathing subsystem are

- the device mapper pseudo driver

- the multipathing device mapper target driver

- multipathing storage system Device Specific Modules (DSMs),

- a multipathing subsystem responsible for run time path selection.

3.1.1   Device Mapper

The device mapper provides a highly modular kernel framework for stacking block device filter drivers. These filter drivers are referred to as

Device Mapper Kernel Architecture

Userspace Applications		
ioctl interface	filesystem interface	
control interface		block interface
core device-mapper		
mapping / target interface		
linear	mirror	snapshot	multipath

log · kcopyd

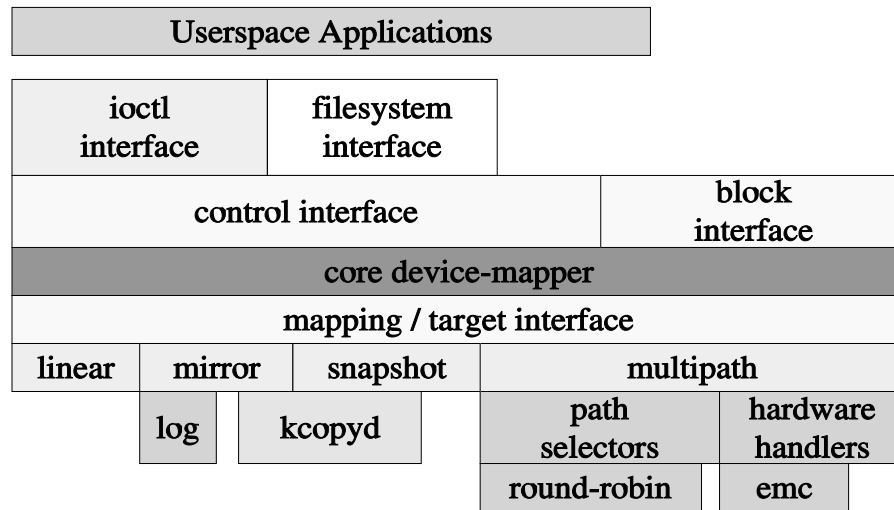path selectors · hardware handlers

round-robin · emc

Figure 2: device mapper kernel architecture

target drivers and are comparable to multidisk personality drivers. Target drivers interact with the device mapper framework through a well defined kernel interface. Target drivers add value by filtering and/or redirecting read and write block io requests directed to a mapped device to one or more target devices. Numerous target drivers already exist, among them ones for logical volume striping, linear concatenation, and mirroring; software encryption; software raid; and various other debug and test oriented drivers.

The device mapper framework promotes a clean separation of policy and mechanism between user and kernel space respectively. Taking this concept even further, this framework supports the creation of a variety of services based on adding value to the dispatching and/or completion handling of block io requests where the bulk of the policy and control logic can reside in user space and only the code actually required to effectively filter or redirect a block

io request must be kernel resident.

The interaction between user and kernel device mapper components takes place through device mapper library interfaces. While the device mapper library currently utilizes a variety of synchronous ioctl(2) interfaces for this purpose, fully backward compatible migration to using Sysfs or Configfs instead is certainly possible.

The device mapper provides the kernel resident mechanisms which support the creation of different combinations of stacked target drivers for different block devices. Each io stack is represented at the top by a single mapped device. Mapped device configuration is initiated from user space via device mapper library interfaces. Configuration information for each mapped device is passed into the kernel within a map or table containing one or more targets or segments. Each map segment consists of a start sector and length and a target driver specific number of

target driver parameters. Each map segment identifies one or more target devices. Since all sectors of a mapped device must be mapped, there are no mapping holes in a mapped device.

Device mapper io stacks are configured in bottom-up fashion. Target driver devices are stacked by referencing a lower level mapped device as a target device of a higher level mapped device. Since a single mapped device may map to one or more target devices, each of which may themselves be a mapped device, a device mapper io stack may be more accurately viewed as an inverted device tree with a single mapped device as the top or root node of the inverted tree. The leaf nodes of the tree are the only target devices which are not device mapper managed devices. The root node is only a mapped device. Every non-root, non-leaf node is both a mapped and target device. The minimum device tree consists of a single mapped device and a single target device. A device tree need not be balanced as there may be device branches which are deeper than others. The depth of the tree may be viewed as the tree branch which has the maximum number of transitions from the root mapped device to leaf node target device. There are no design limits on either the depth or breadth of a device tree.

Although each target device at each level of a device mapper tree is visible and accessible outside the scope of the device mapper framework, concurrent open of a target device for other purposes requiring its exclusive use such as is required for partition management and file system mounting is prohibited. Target devices are exclusively recognized or claimed by a mapped device by being referenced as a target of a mapped device. That is, a target device may only be used as a target of a single mapped device. This restriction prohibits both the inclusion of the same target device within multiple device trees and multiple references to the same target device within the same device

tree, that is, loops within a device tree are not allowed.

It is strictly the responsibility of user space components associated with each target driver to

- discover the set of associated target devices associated with each mapped device managed by that driver

- create the mapping tables containing this configuration information

- pass the mapping table information into the kernel

- possibly save this mapping information in persistent storage for later retrieval.

The multipath path configurator fulfills this role for the multipathing target driver. The `lvm(8)`, `dmraid(8)`, and `dmsetup(8)` commands perform these tasks for the logical volume management, software raid, and the device encryption target drivers respectively.

While the device mapper registers with the kernel as a block device driver, target drivers in turn register callbacks with the device mapper for initializing and terminating target device metadata; suspending and resuming io on a mapped device; filtering io dispatch and io completion; and retrieving mapped device configuration and status information. The device mapper also provides key services, (e.g., io suspension/resumption, bio cloning, and the propagation of io resource restrictions), for use by all target drivers to facilitate the flow of io dispatch and io completion events through the device mapper framework.

The device mapper framework is itself a component driver within the outermost `generic_make_request` framework for block devices. The `generic_make_request`

framework also provides for stacking block device filter drivers. Therefore, given this architecture, it should be at least architecturally possible to stack device mapper drivers both above and below multidisk drivers for the same target device.

The device mapper processes all read and write block io requests which pass through the block io subsystem's `generic_make_request` and/or `submit_bio` interfaces and is directed to a mapped device. Architectural symmetry is achieved for io dispatch and io completion handling since io completion handling within the device mapper framework is done in the inverse order of io dispatch. All read/write bios are treated as asynchronous io within all portions of the block io subsystem. This design results in separate, asynchronous and inversely ordered code paths through both the `generic_make_request` and the device mapper frameworks for both io dispatch and completion processing. A major impact of this design is that it is not necessary to process either an io dispatch or completion either immediately or in the same context in which they are first seen.

Bio movement through a device mapper device tree may involve fan-out on bio dispatch and fan-in on bio completion. As a bio is dispatched down the device tree at each mapped device, one or more cloned copies of the bio are created and sent to target devices. The same process is repeated at each level of the device tree where a target device is also a mapped device. Therefore, assuming a very wide and deep device tree, a single bio dispatched to a mapped device can branch out to spawn a practically unbounded number of bios to be sent to a practically unbounded number of target devices. Since bios are potentially coalesced at the device at the bottom of the `generic_make_request` framework, the io request(s) actually queued to one or more target devices

at the bottom may bear little relationship to the single bio initially sent to a mapped device at the top. For bio completion, at each level of the device tree, the target driver managing the set of target devices at that level consumes the completion for each bio dispatched to one of its devices, and passes up a single bio completion for the single bio dispatched to the mapped device. This process repeats until the original bio submitted to the root mapped device is completed.

The device mapper dispatches bios recursively from top (root node) to bottom (leaf node) through the tree of device mapper mapped and target devices in process context. Each level of recursion moves down one level of the device tree from the root mapped device to one or more leaf target nodes. At each level, the device mapper clones a single bio to one or more bios depending on target mapping information previously pushed into the kernel for each mapped device in the io stack since a bio is not permitted to span multiple map targets/segments. Also at each level, each cloned bio is passed to the map callout of the target driver managing a mapped device. The target driver has the option of

1. queuing the io internal to that driver to be serviced at a later time by that driver,

2. redirecting the io to one or more different target devices and possibly a different sector on each of those target devices, or

3. returning an error status for the bio to the device mapper.

Both the first or third options stop the recursion through the device tree and the `generic_make_request` framework for that matter. Otherwise, a bio being directed to the first target device which is not managed by the device mapper causes the bio to exit the device mapper

framework, although the bio continues recursing through the `generic_make_request` framework until the bottom device is reached.

The device mapper processes bio completions recursively from a leaf device to the root mapped device in soft interrupt context. At each level in a device tree, bio completions are filtered by the device mapper as a result of redirecting the bio completion callback at that level during bio dispatch. The device mapper callout to the target driver responsible for servicing a mapped device is enabled by associating a target_io structure with the bi_private field of a bio, also during bio dispatch. In this fashion, each bio completion is serviced by the target driver which dispatched the bio.

The device mapper supports a variety of push/pull interfaces to enhance communication across the system call boundary. Each of these interfaces is accessed from user space via the device mapper library which currently issues ioctls to the device mapper character interface. The occurrence of target driver derived io related events can be passed to user space via the device mapper event mechanism. Target driver specific map contents and mapped device status can be pulled from the kernel using device mapper messages. Typed messages and status information are encoded as ASCII strings and decoded back to their original form according dictated by their type.

3.1.2 Multipath Target Driver

A multipath target driver is a component driver of the device mapper framework. Currently, the multipath driver is position dependent within a stack of device mapper target drivers: it must be at the bottom of the stack. Furthermore, there may not be other filter drivers, (e.g., multidisk), stacked underneath it. It must be stacked im-mediately atop driver which services a block request queue, for example, `/dev/sda`.

The multipath target receives configuration information for multipath mapped devices in the form of messages sent from user space through device mapper library interfaces. Each message is typed and may contain parameters in a position dependent format according to message type. The information is transferred as a single ASCII string which must be encoded by the sender and decoded by the receiver.

The multipath target driver provides path failover and path load sharing. Io failure on one path to a device is captured and retried down an alternate path to the same device. Only after all paths to the same device have been tried and failed is an io error actually returned to the io initiator. Path load sharing enables the distribution of bios amongst the paths to the same device according to a path load sharing policy.

Abstractions are utilized to represent key entities. A multipath corresponds to a device. A logical path to a device is represented by a path. A path group provides a way to associate paths to the same device which have similar attributes. There may be multiple path groups associated with the same device. A path selector represents a path load sharing algorithm and can be viewed as an attribute of a path group. Round robin based path selection amongst the set of paths in the same path group is currently the only available path selector. Storage system specific behavior can be localized within a multipath hardware handler.

The multipath target driver utilizes two sub-component frameworks to enable both storage system specific behavior and path selection algorithms to be localized in separate modules which may be loaded and managed separately from the multipath target driver itself.

3.1.3 Device Specific Module

A storage system specific component can be associated with each target device type and is referred to as a hardware handler or Device Specific Module (DSM). A DSM allows for the specification of kernel resident storage system specific path group initialization, io completion filtering, and message handling. Path group initialization is used to utilize storage system specific actions to activate the passive interface of an active-passive storage system. Storage system specific io completion filtering enables storage system specific error handling. Storage system specific message handling enables storage system specific configuration.

DSM type is specified by name in the multipath target driver map configuration string and must refer to a DSM pre-loaded into the kernel. A DSM may be passed paramters in the configuration string. A hardware context structure passed to each DSM enables a DSM to track state associated with a particular device.

Associating a DSM with a block device type is optional. The EMC CLARiion DSM is currently the only DSM.

3.1.4 Path Selection Subsystem

A path selector enables the distribution of io amongst the set of paths in a single path group.

Path selector type is specified by name in the multipath target driver map configuration string and must refer to a path selector pre-loaded into the kernel. A path selector may be passed paramters in the configuration string. The path selector context structure enables a path selector type to track state across multiple ios to the paths of a path group.

Each path group must be associated with a path selector. A single round robin path selector exists today.

3.2 User Modules

Figure 3 outlines the architecture of the user-mode multipath tools. Multipath user space components perform path discovery, path policy management and configuration, and path health testing. The multipath configurator is responsible for discovering the network topology for multipathed block devices and for updating kernel resident multipath target driver configuration and state information. The multipath daemon monitors the usability of paths both in response to actual errors occurring in the kernel and proactively via periodic path health testing. Both components share path discovery and path health testing services. Furthermore, these services are implemented using an extensible framework to facilitate multipath support for new block device types, block devices from new vendors, and new models. The kpartx tool creates mapped devices for partitions of multipath managed block devices.

3.2.1 Multipath Configurator

Path discovery involves determining the set of routes from a host to a particular block device which is configured for multipathing. Path discovery is implemented by scanning Sysfs looking for block device names from a multipath configuration file which designate block device types eligible for multipathing. Each entry in /sys/block corresponds to the gendisk for a different block device. As such, path discovery is independent of whatever path transport is used between host and device. Since devices are assumed to have an identifier attribute which is unique in both time and space, the

low level device drivers

calls

execs

listen

ioctl

add/del block device

block

sysfs

device
mapper

test
failed
paths

hotplug

NetLink
uevents

kernel space

user space

udev

1) wait devmap events
2) reinstate paths
3) switch path group
    (failback)

(re)configure
multipath
device
maps

configure
partitions
device
maps

1) add path
2) remove path
3) add devmap
4) remove devmap

kpartx

multipath

multipathd

manual exec

Figure 3: multipath tools architecture

cumulative set of paths found from Sysfs are coalesced based on device UID. Configuration driven multipath attributes are setup for each of these paths.

The multipath configurator synchronizes path configuration and path state information across both user and kernel multipath components. The current configuration path state is compared with the path state pulled from the multipath target driver. Most discrepancies are dealt with by pushing the current configuration and state information into the multipath target driver. This includes creating a new multipath map for a newly discovered device; changing the contents of an existing multipath map for a newly discovered path to a known device, for a path to a known device which is no longer visible, and for configuration driven multipath attributes which may have changed; and for updating the state of a path.

Configuration and state information are passed between user and kernel space multipath components as position dependent information as a single string. The entire map for a mapped device is transferred as a single string and must be encoded before and decoded after the transfer.

The multipath configurator can be invoked manually at any time or automatically in reaction to a hotplug event generated for a configuration change for a block device type managed by the multipathing subsystem. Configuration changes involve either the creation of a new path or removal of an existing path.

3.2.2   Multipath Daemon

The multipath daemon actively tests paths and reacts to changes in the multipath configuration.

Periodic path testing performed by the multipath daemon is responsible for both restoring failed paths to an active state and proactively failing active paths which fail a path test. Currently, while the default is to test all active and failed paths for all devices every 5 seconds, this interval can be changed via configuration directive in the multipath configuration file. The current non-optimized design could be enhanced to reduce path testing overhead by

- testing the physical transport components instead of the logical ones

- varying the periodic testing interval.

An example of the former for SCSI block devices is to

- associate across all devices those paths which utilize common SCSI initiators and targets and

- for each test interval test only one path for every unique combination of initiator and target.

An example of the latter is to vary the periodic test interval in relationship to the recent past history of the path or physical components, that is, paths which fail often get tested more frequently.

The multipath daemon learns of and reacts to changes in both the current block device configuration and the kernel resident multipathing configuration. The addition of a new path or the removal of an already existing path to an already managed block device is detected over a netlink socket as a uevent triggered callback which adds or removes the path to or from the set of paths which will be actively tested. Changes to the kernel resident multipathing state are detected as device-mapper generated event callbacks. Events of this kind invole block io errors, path state change, and changes in the highest priority path group for a mapped device.

3.2.3 Multipath Framework

The multipath framework enables the use of block device vendor specific algorithms for

1. deriving a UID for identifying the physical device associated with a logical path

2. testing the health of a logical path

3. determining how to organize the logical paths to the same device into separate sets,

4. assigning a priority to each path

5. determining how to select the next path within the same path set

6. specifying any kernel resident device specific multipathing capabilities.

While the last two capabilities must be kernel resident, the remaining user resident capabilities are invoked either as functions or executables. All but item four and item six are mandatory. A built-in table specifying each of these capabilities for each supported block device vendor and type may, but need not be, overridden by configuration directives in a multipath configuration file. Block device vendor and type are derived from attributes associated with the Sysfs device file probed during device discovery. Configuration file directives may also be used to configure these capabilities for a specific storage system instance.

A new storage system is plugged into this user space multipath framework by specifying a configuration table or configuration file entry for the storage system and providing any of the necessary, but currently missing mechanism needed to satisfy the six services mentioned above for the storage system. The service selections are specified as string or integer constants. In some cases, the selection is made from a restricted domain of options. In other cases a new mechanism can be utilized to provide the required service. Services which are invoked as functions must be integrated into the multipath component libraries while those invoked as executables are not so restricted. Default options provided for each service may also be overridden in the multipath configuration file.

Since the service which derives a UID for a multipath device is currently invoked from the multipath framework as an executable, the service may be, and in fact is now external to the multipath software. All supported storage systems (keep in mind they are all SCSI at the moment) utilize `scsi_id(8)` to derive a UID for a SCSI logical unit. Almost all of these cases obtain the UID directly from the Vendor Specified Identifier field of an extended SCSI inquiry command using vital product page 0x83. This is indeed the default option. Although `scsi_id` is invoked as an executable today, a `scsi_id` service library appears to be in-plan, thereby allowing in-context UID generation from this framework in the near future.

Path health testing is invoked as a service function built into the libcheckers multipath library. While SCSI specific path testing functions already exist in this library based on reading sector 0 (this is the default) and issuing a TUR, path health testing can be specified to be storage system specific but must be included within libcheckers.

The selection of how to divide up the paths to the same device into groups is restricted to a set of five options:

- failover

- multibus

- group-by-priority

- group-by-serial

- group-by-node-name.

Failover, the default policy, implies one path per path group and can be used to disallow path load sharing while still providing path failover. Multibus, by far the most commonly selected option, implies one path group for all paths and is used in most cases when access is symmetric across all paths, e.g., active-active storage systems. Group-by-priority implies a grouping of paths with the same priority. This option is currently used only by the active-passive EMC CLARiion storage array and provides the capability to assign a higher priority to paths connecting to the portion of the storage system which has previously been assigned to be the default owner of the SCSI logical unit. Group-by-serial implies a grouping based on the Vendor Specified Identifier returned by a VPD page 0x80 extended SCSI inquiry command. This is a good way to group paths for an active-passive storage system based on which paths are currently connected to the active portion of the storage system for the SCSI logical unit. Group-by-node-name currently implies a grouping by by SCSI target.

Paths to the same device can be assigned priorities in order to both enable the group-by-priority path grouping policy and to affect path load sharing. Path group priority is a summation of the priority for each active path in the group. An io is always directed to a path in the highest priority path group. The `get_priority` service is currently invoked as an executable. The default option is to not assign a priority to any path, which leads to all path groups being treated equally. The `pp_balance_paths` executable assigns path priority in order to attempt to balance path usage for all multipath devices across the SCSI targets to the same storage system. Several storage system specific path priority services are also provided.

Path selectors and hardware contexts are specified by name and must refer to specific kernel resident services. A path selector is mandatory and currently the only option is round-robin. A hardware context is by definition storage system specific. Selection of hardware context is optional and only the EMC CLARiion storage system currently utilizes a hardware context. Each may be passed parameters, specified as a count followed by each parameter.

3.2.4 Kpartx

The kpartx utility creates device-mapper mapped devices for the partitions of multipath managed block devices. Doing so allows a block device partition to be managed within the device mapper framework as would be any whole device. This is accomplished by reading and parsing a target device's partition table and setting up the device-mapper table for the mapped device from the start address and length fields of the parition table entry for the partition in question. Kpartx uses the same devmapper library interfaces as does the multipath configurator in order to create and initialize the mapped device.

4 Interaction Amongst Key Kernel and User Components

The interaction between key user and kernel multipath components will be examined while considering several use cases. Device and path configuration will be considered first. Io scheduling and io failover will then be examined in detail.

4.1 Block Device and Path Discovery

Device discovery consists of obtaining information about both the current and previous multipath device configurations, resolving any differences, and pushing the resultant updates into the multipath target driver. While these tasks are primarily the responsibility of the multipath configurator, many of the device discovery services are in fact shared with the multipath daemon.

The device discovery process utilizes the common services of the user space multipath framework. Framework components to identify, test, and prioritize paths are selected from pre-established table or config driven policy options based on device attributes obtained from probing the device's Sysfs device file.

The discovery of the current configuration is done by probing block device nodes created in Sysfs. A block device node is created by udev in reaction to a hotplug event generated when a block device's request queue is registered with the kernel's block subsystem. Each device node corresponds to a logical path to a block device since no kernel resident component other than the multipath target driver is multipath cognizant.

The set of paths for the current configuration are coalesced amongst the set of multipath managed block devices. Current path and device configuration attributes are retrieved configuration file and/or table entries.

The previous device configuration stored in the collective set of multipath mapped device maps is pulled from the multipath target driver using target driver specific message ioctls issued by the device-mapper library.

Discrepancies between the old and new device configuration are settled and the updated device

configuration and state information is pushed into the multipath target driver one device at a time. Several use cases are enumerated below.

- A new mapped device is created for a multipath managed device from the new configuration which does not exist in the old configuration.

- The contents of an existing multipath map are updated for a newly discovered path to a known device, for a path to a known device which is no longer visible and for multipath attributes which may have changed. Examples of multipath attributes which can initiate an update of the kernel multipath device configuration are enumerated below.

  - device size
  - hardware handler
  - path selector
  - multipath feature parameters
  - number of path groups
  - assignment of paths to path groups
  - highest priority path group

- Path state is updated based on path testing done during device discovery.

Configuration updates to an existing multipath mapped device involve the suspension and subsequent resumption of io around the complete replacement of the mapped device's device-mapper map. Io suspension both blocks all new io to the mapped device and flushes all io from the mapped device's device tree. Path state updates are done without requiring map replacement.

Hotplug initiated invocation of the multipath configurator leads to semi-automated multipath response to post-boot time changes in the block

device configuration. For SCSI target devices, a hotplug event is generated for a SCSI target device when the device's gendisk is registered after the host attach of a SCSI logical unit and unregistered after the host detach of a SCSI logical unit.

4.2   Io Scheduling

The scheduling of bios amongst the multiple multipath target devices for the same multipath mapped device is controlled by both a path grouping policy and a path selection policy. While both path group membership and path selection policy assignment tasks are performed in user space, actual io scheduling is implemented via kernel resident mechanism.

Paths to the same device can be separated into path groups, where all paths in the same group have similar path attributes. Both the number of path groups and path membership within a group are controlled by the multipath configurator based on one of five possible path grouping policies. Each path grouping policy uses different means to assign a path to a path group in order to model the different behavior in the physical configuration. Each path is assigned a priority via a designated path priority callout. Path group priority is the summation of the path priorities for each path in the group. Each path group is assigned a path selection policy governing the selection of the next path to use when scheduling io to a path within that group.

Path group membership and path selection information are pushed into the kernel where it is then utilized by multipath kernel resident components to schedule each bio on one of multipath paths. This information consists of the number of path groups, the highest priority path group, the path membership for each group (target devices specified by dev_t), the name of the path selection policy for each group, a

count of optional path selection policy parameters, and the actually path selection policy parameters if the count value is not zero. As is the case for all device mapper map contents passed between user and kernel space, the collective contents is encoded and passed as a single string, and decoded on the other side according to its position dependent context.

Path group membership and path selection information is pushed into the kernel both when a multipath mapped device is first discovered and configured and later when the multipath configurator detects that any of this information has changed. Both cases involve pushing the information into the multipath target driver within a device mapper map or table. The latter case also involves suspending and resuming io to the mapped device during the time the map is updated.

Path group and path state are also pushed into the kernel by the multipath configurator independently of a multipath mapped device's map. A path's state can be either active or failed. Io is only directed by the multipath target driver to a path with an active path state. Currently a path's state is set to failed either by the multipath target driver after a single io failure on the path or by the multipath configurator after a path test failure. A path's state is restored to active only in user space after a multipath configurator initiated path test succeeds for that path. A path group can be placed into bypass mode, removed from bypass mode, or made the highest priority path group for a mapped device. When searching for the next path group to use when there are no active paths in the highest priority path group, unless a new path group has been designated as the highest priority group, all path groups are searched. Otherwise, path groups in bypass mode are first skipped over and selected only if there are no path groups for the mapped device which are not in bypass mode.

The path selection policy name must refer to an already kernel resident path selection policy module. Path selection policy modules register half dozen callbacks with the multipath target driver's path selection framework, the most important of which is invoked in the dispatch path of a bio by the multipath target driver to select the next path to use for the bio.

Io scheduling triggered during the multipath target driver's bio dispatch callout from the device mapper framework consists of first selecting a path group for the mapped device in question, then selecting the active path to use within that group, followed by redirecting the bio to the selected path. A cached value of the path group to use is saved with each multipath mapped device in order to avoid its recalculation for each bio redirection to that device. This cached value is initially set from to the highest priority path group and is recalculated if either

- the highest priority path group for a mapped device is changed from user space or

- the highest priority path group is put into bypassed mode either from kernel or user space multipathing components.

A cached value of the path to use within the highest priority group is recalculated by invoking the path selection callout of a path selection policy whenever

- a configurable number of bios have already been redirected on the current path,

- a failure occurs on the current path,

- any other path gets restored to a usable state, or

- the highest priority path group is changed via either of the two methods discussed earlier.

Due to architectural restrictions and the relatively (compared with physical drivers) high positioning of the multipath target driver in the block io stack, it is difficult to implement path selection policies which take into account the state of shared physical path resources without implementing significant new kernel resident mechanism. Path selection policies are limited in scope to the path members of a particular path group for a particular multipath mapped device. This multipath architectural restriction together with the difficulty in tracking resource utilization for physical path resources from a block level filter driver makes it difficult to implement path selection policies which could attempt to minimize the depth of target device request queues or the utilization of SCSI initiators. Path selectors tracking physical resources possibly shared amongst multiple hosts, (e.g., SCSI targets), face even more difficulties.

The path selection algorithms are also impacted architecturally by being positioned above the point at the bottom of the block io layer where bios are coalesced into io requests. To help deal with this impact, path reselection within a priority group is done only for every n bios, where n is a configurable repeat count value associated with each use of a path selection policy for a priority group. Currently the repeat count value is set to 1000 for all cases in order to limit the adverse throughput effects of dispersing bios amongst multiple paths to the same device, thereby negating the ability of the block io layer to coalesce these bios into larger io requests submitted to the request queue of bottom level target devices.

A single round-robin path selection policy exists today. This policy selects the least recently used active path in the current path group for the particular mapped device.

4.3 Io Failover

While actual failover of io to alternate paths is performed in the kernel, path failover is controlled via configuration and policy information pushed into the kernel multipath components from user space multipath components.

While the multipath target driver filters both io dispatch and completion for all bios sent to a multipath mapped device, io failover is triggered when an error is detected while filtering io completion. An understanding of the error handling taking place underneath the multipath target driver is useful at this point. Assuming SCSI target devices as leaf nodes of the device mapper device tree, the SCSI mid-layer followed by the SCSI disk class driver each parse the result field of the `scsi_cmd` structure set by the SCSI LLDD. While parsing by the SCSI mid-layer and class driver filter code filter out some error states as being benign, all other cases lead to failing all bios associated with the io request corresponding to the SCSI command with -EIO. For those SCSI errors which provide sense information, SCSI sense key, Additional Sense Code (ASC), and Additional Sense Code Qualifier (ASCQ) byte values are set in the `bi_error` field of each bio. The -EIO, SCSI sense key, ASC, and ASCQ are propagated to all parent cloned bios and are available for access by the any target driver managing target devices as the bio completions recurse back up to the top of the device tree.

Io failures are first seen as a non-zero error status, (i.e., -EIO), in the error parameter passed to the multipath target driver's io completion filter. This filter is called as a callout from the device mapper's bio completion callback associated with the leaf node bios. Assuming one exists, all io failures are first parsed by the storage system's hardware context's error handler. Error parsing drives what happens next for the path, path group, and bio associated with the io

failure. The path can be put into a failed state or left unaffected. The path group can be placed into a bypassed state or left unaffected. The bio can be queued for retry internally within the multipath target driver or failed. The actions on the path, the path group, and the bio are independent of each other. A failed path is unusable until restored to a usable state from the user space multipath configurator. A bypassed path group is skipped over when searching for a usable path, unless there are no usable paths found in other non-bypassed path groups. A failed bio leads to the failure of all parent cloned bios at higher levels in the device tree.

Io retry is done exclusively in a dedicated multipath worker thread context. Using a worker thread context allows for blocking in the code path of an io retry which requires a path group initialization or which gets dispatched back to `generic_make_request`—either of which may block. This is necessary since the bio completion code path through the device mapper is usually done within a soft interrupt context. Using a dedicated multipath worker thread avoids delaying the servicing of non-multipath related work queue requests as would occur by using the kernel's default work queue.

Io scheduling for path failover follows basically the same path selection algorithm as that for an initial io dispatch which has exhausted its path repeat count and must select an alternate path. The path selector for the current path group selects the best alternative path within that path group. If none are available, the next highest priority path group is made current and its path selector selects the best available path. This algorithm iterates until all paths of all path groups have been tried.

The device mapper's kernel resident event mechanism enables user space applications to determine when io related events occur in the

kernel for a mapped device. Events are generated by the target driver managing a particular mapped device. The event mechanism is accessed via a synchronous device mapper library interface which blocks a thread in the kernel in order to wait for an event associated with a particular mapped device. Only the event occurrence is passed to user space. No other attribute information of the event is communicated.

The occurrence of a path failure event (along with path reinstatement and a change in the highest priority path group) is communicated from the multipath target driver to the multipath daemon via this event mechanism. A separate multipath daemon thread is allocated to wait for all multipath events associated with each multipath mapped device. The detection of any multipath event causes the multipath daemon to rediscover its path configuration and synchronize its path configuration, path state, and path group state information with the multipath target driver's view of the same.

A previously failed path is restored to an active state only as a result of passing a periodically issued path health test issued by the multipath daemon for all paths, failed or active. This path state transition is currently enacted by the multipath daemon invoking the multipath configurator as an executable.

A io failure is visible above the multipathing mapped device only when all paths to the same device have been tried once. Even then, it is possible to configure a mapped device to queue for an indefinite amount of time such bios on a queue specific to the multipath mapped device. This feature is useful for those storage systems which can possibly enter a transient all-paths-down state which must be ridden through by the multipath software. These bios will remain where they are until the mapped device is suspended, possibly done when the mapped device's map is updated, or when a previously failed path is reinstated. There are no practical

limits on either the amount of bios which may be queued in this manner nor on the amount of time which these bios remain queued. Furthermore, there is no congestion control mechanism which will limit the number of bios actually sent to any device. These facts can lead to a significant amount of dirty pages being stranded in the page cache thereby setting the stage for potential system deadlock if memory resources must be dynamically allocated from the kernel heap anywhere in the code path of reinstating either the map or a usable path for the mapped device.

5   Future Enhancements

This section enumerates some possible enhancements to the multipath implementation.

5.1   Persistent Device Naming

The cryptic name used for the device file associated with a device mapper mapped device is often renamed by a user space component associated with the device mapper target driver managing the mapped device. The multipathing subsystem sets up udev configuration directives to automatically rename this name when a device mapper device file is first created. The dm-<minor #> name is changed to the ASCII representation of the hexi-decimal values for each 4-bit nibble of the device's UID utilized by multipath. Yet, the resultant multipath device names are still cryptic, unwieldly, and their use is prone to error. Although an alias name may be linked to each multipath device, the setup requires manipulcation of the multipath configuration file for each device. The automated management of multipath alias names by both udev and multipath components seems a reasonable next step.

It should be noted that the Persistent Storage Device Naming specification from the Storage Networking SIG of OSDL is attempting to achieve consistent naming across all block devices.

5.2   Event Mechanism

The device mapper's event mechanism enables user space applications to determine when io related events occur in the kernel for a mapped device. Events are generated by the target driver managing a particular mapped device. The event mechanism is currently accessed via a synchronous device mapper library interface which blocks a thread in the kernel in order to wait for an event associated with a particular mapped device. Only the event occurrence is passed back to user space. No other attribute information of the event is communicated.

Potential enhancements to the device mapper event mechanism are enumerated below.

1. Associating attributes with an event and providing an interface for communicating these attributes to user space will improve the effectiveness of the event mechanism. Possible attributes for multipath events include (1) the cause of the event, (e.g., path failure or other), (2) error or status information associated with the event, (e.g., SCSI sense key/ASC/ASCQ for a SCSI error), and (3) an indication of the target device on which the error occurred.

2. Providing a multi-event wait synchronous interface similar to select(2) or poll(2) will significantly reduce the thread and memory resources required to use the event mechanism. This enhancement will allow a single user thread to wait on events for multiple mapped devices.

3. A more radical change would be to integreate the device-mappers event mechanism with the kernel's kobject subsystem. Events could be send as uevents to be received over an AF_NETLINK socket.

5.3   Monitoring of io via Iostat(1)

Block io to device mapper mapped devices cannot currently be monitored via iostat(1) or /proc/diskstats. Although an io to a mapped device is tracked on the actual target device(s) at the bottom of the generic_make_request device tree, io statistics are not tracked for any device mapper mapped devices positioned within the device tree.

Io statistics should be tracked for each device mapper mapped device positioned on an io stack. Multipathing must account for possibly multiple io failures and subsequent io retry.

5.4   IO Load Sharing

Additional path selectors will be implemented. These will likely include state based ones which select a path based on the minimum number of outstanding bios or minimum round trip latency. While the domain for this criteria is likely a path group for one mapped device, it may be worth looking sharing io load across actual physical components, (e.g., SCSI initiator or target), instead.

5.5   Protocol Agnostic Multipathing

Achieving protocol agnostic multipathing will require the removal of some SCSI specific affinity in the kernel, (e.g., SCSI-specific error information in the bio), and user, (e.g., path discovery), multipath components.

5.6 Scalable Path Testing

Proactive path testing could be enhanced to support multiple path testing policies and new policies created which provide improved resource scalability and improve the predictability of path failures. Path testing could emphasize the testing of the physical components utilized by paths instead of simply exhaustively testing every logical path. For example, the availability through Sysfs of path transport specific attributes for SCSI paths could will make it easier to group paths which utilize common physical components. Additionally, the frequency of path testing can be based on the recent reliability of a path, that is, frequently and recently failed paths are more often.

6 Architectural Issues

This section describes several critical architectural issues.

6.1 Elevator Function Location

The linux block layer performs the sorting and merging of IO requests (elevator modules) in a layer just above the device driver. The dm device mapper supports the modular stacking of multipath and RAID functionality above this layer.

At least for the device mapper multipath module, it is desirable to either relocate the elevator functionality to a layer above the device mapper in the IO stack, or at least to add an elevator at that level.

An example of this need can be seen with a multipath configuration where there are four equivalent paths between the host and each target. Assume also there is no penalty for switching paths. In this case, the multipath module wants to spread IO evenly across the four paths. For each IO, it may choose a path based on which path is most lightly loaded.

With the current placement of the elevator then, IO requests for a given target tend to be spread evenly across each of the four paths to that target. This reduces the chances for request sorting and merging.

If an elevator were placed in the IO stack above the multipath layer, the IO requests coming into the multipath would already be sorted and merged. IO requests on each path would at least have been merged. When IO requests on different paths reach their common target, the IO's will may nolonger be in perfect sorted order. But they will tend to be near each other. This should reduce seeking at the target.

At this point, there doesn't seem to be any advantage to retaining the elevator above the device driver, on each path in the multipath. Aside from the additional overhead (more memory occupied by the queue, more plug/unplug delay, additional cpu cycles) there doesn't seem to be any harm from invoking the elevator at this level either. So it may be satisfactory to just allow multiple elevators in the IO stack.

Regarding other device mapper targets, it is not yet clear whether software RAID would benefit from having elevators higher in the IO stack, interspersed between RAID levels. So, it maybe be sufficient to just adapt the multipath layer to incorporate an elevator interface.

Further investigation is needed to determine what elevator algorithms are best for multipath. At first glance, the Anticipatory scheduler seems inappropriate. It's less clear how the deadline scheduler of CFQ scheduler would

perform in conjunction with multipath. Consideration should be given to whether a new IO scheduler type could produce benefits to multipath IO performance.

6.2 Memory Pressure

There are scenarios where all paths to a logical unit on a SCSI storage system will appear to be failed for a transient period of time. One such expected and transient all paths down use case involves an application transparent upgrade of the micro-code of a SCSI storage system. During this operation, it is expected that for a reasonably short period of time likely bounded by a few minutes, all paths to a logical unit on the storage system in question will appear to a host to be failed. It is expected that a multipathing product will be capable of riding through this scenario without failing ios back to applications. It is expected that the multipathing software will both detect when one or more of the paths to such a device become physically usable again, do what it takes to make the paths usable, and retry ios which failed during the all paths down time period.

If this period coincides with a period of extreme physical memory congestion it must still be possible for multipath components to enable the use of these paths as they become physically usable. While a kernel resident congestion control mechanism based on block request allocation exists to ward off the over commital of page cache memory to any one target device, there are no congestion control mechanisms that take into account either the use of multiple target devices for the same mapped device or the internal queuing of bios within device mapper target drivers.

The multipath configuration for several storage systems must include the multipath feature `queue_if_no_path` in order to not immediately return to an application an io request whose transfer has failed on every path to its device. Yet, the use of this configuration directive can result in the queuing of an indefinite number of bios each for an indefinite period of time when there are no usable paths to a device. When coincident with a period of heavy asynchronous write-behind in the page cache, this can lead to lots of dirty page cache pages for the duration of the transient all paths down period.

Since memory congestion states like this cannot be detected accurately, the kernel and user code paths involved with restoring a path to a device must not ever execute code which could result in blocking while an io is issued to this device. A blockable (i.e., `__GFP_WAIT`) memory allocation request in this code path could block for write-out of dirty pages to this device from the synchronous page reclaim algorithm of `__alloc_pages`. Any modification to file system metadata or data could block flushing modified pages to this device. Any of these actions have the potential of deadlocking the multipathing software.

These requirements are difficult to satisfy for multipathing software since user space intervention is required to restore a path to a usable state. These requirements apply to all user and kernel space multipathing code (and code invoked by this code) which is involved in testing a path and restoring it to a usable state. This precludes the use of fork, clone, or exec in the user portion of this code path. Path testing initiated from user space and performed via ioctl entry to the block scsi ioctl code must also conform to these requirements.

The pre-allocation of memory resources in order to make progress for a single device at a time is a common solution to this problem. This approach may require special case code for tasks such as the kernel resident path testing. Furthermore, in addition to being "locked to core," the user space components must only

invoke system calls and library functions which also abide by these requirements. Possibly combining these approaches with a bit of congestion control applied against bios (to account for the ones internally queued in device-mapper target drivers) instead of or in addition to block io requests and/or a mechanism for timing out bios queued within the multipath target driver as a result of the `queue_if_no_path` multipath feature is a reasonable starting point.

7 Conclusion

This paper has analyzed both architecture and design of the block device multipathing indigenous to linux. Several architectural issues and potential enhancements have been discussed.

The multipathing architecture described in this paper is actually implemented in several linux distributions to be released around the time this paper is being written. For example, SuSE SLES 9 service pack 2 and Red Hat AS 4 update 1 each support Linux multipathing. Furthermore, several enhancements described in this paper are actively being pursued.

Please reference `http://christophe.varoqui.free.fr` and `http://sources.redhat.com/dm` for the most up-to-date development versions of the user- and kernel-space resident multipathing software respectively. The first web site listed also provides a detailed description of the syntax for a multipathing device-mapper map.

Proceedings of the
Linux Symposium

Volume One

July 20nd–23th, 2005
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Stephanie Donovan, *Linux Symposium*

Review Committee

Gerrit Huizenga, *IBM*
Matthew Wilcox, *HP*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Matt Domsch, *Dell*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*