VMS High Level Design Specification for Multipath Switching

Digital Confidential

Glenn C. Everhart
4-June-1997

Abstract:

This design specification describes the design for multipath failover switching, where multiple paths to storage devices exist and must be failed over.

This document covers the high level design concepts in the design spec for use by a wide review audience.

Introduction

This document describes a proposed method for supporting failover between multiple paths that may exist to peripheral devices in an OpenVMS system. The goal of this document is to facilitate a wide review of the proposed concepts by all who may be impacted, or may be able to provide valuable input. The facilitate this, the design is presented at a high level, and as succinctly as possible. Additional design details are available from the author. Interested parties may also wish to refer to the PS, PP, and IR in STAR::DOCD$:[EVMS.PROJECT_DOCUMENTS] as **-MULTIPATH-FAILOVER where ** is DS, IR, or PP.

Problem Statement

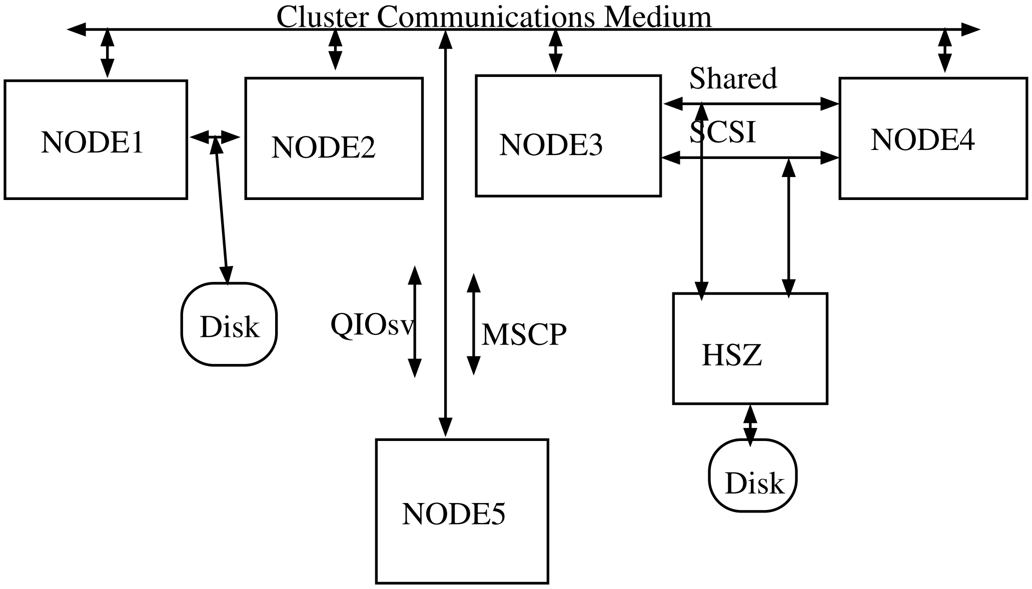
OpenVMS has long supported failover between multiple paths to an MSCP storage device. The paths may include multiple local paths, as well as an MSCP-served path. Since this type of multipath failover is specific to MSCP devices, it was implemented in the MSCP class drivers.

There are three developments that make this type of multipath support inadequate for the future:

1. SCSI clusters provide a configuration in which it is desirable to be able to switch between a local SCSI path and an MSCP served path.

2. Multi-ported SCSI devices are beginning to become available, and are an important component in high availability systems. It is expected that all future parallel SCSI RAID array controllers will have multiple ports, and all FC devices will be multiported.

3. It is expected that the QIO server and the MSCP server will be run simultaneously on some systems in the cluster. This poses the need for switching between two different served paths to a device.



NODE3 and NODE4 share a disk via an HSZ controller and can access it via either of two direct paths, or via served paths. Failover between these is needed.

If NODE1 or NODE2 lose direct access to their shared disk a served path exists which might be failed over to. If both MSCP and QIO serving exist, failover might be from direct path to MSCP path to QIOserver path.

NODE5 can access the disks via MSCP or QIOserver paths but needs to be able to fail over access to the other if one breaks.

Fig. 1 - Varieties of multi-path access

In light of these new developments, it is clear that a mechanism for multipath switching that is more general than the current MSCP-only approach is required.

Design Challenges

There are two main problems which must be dealt with in this project.

1. Creating technology to allow I/O to be switched between two or more paths so they will appear to the rest of OpenVMS as a single device with a single, cluster-unique name, and

2. Choosing the proper name for the device so that the cluster may use the device consistently and without inducing routing or other instabilities and so that the name will be constant across boots, predictable, and usable even in the face of widely varying connection topologies. The name choice must not impose significant workload on human operators either.

The Hardware Environment

The following is background information about what various controllers do.

Device identification is expected to be available in several forms. Three of these are:

Real WWNs
1. A real WWN - each device has a real worldwide-unique ID (WWN) which can be obtained by commands directly from the device. Ultimately most SCSI storage is expected to use this model.

Controller Serial Numbers
2. Controllers may supply their own identifiers. A variant of this is used in the HSZ70 and HSZ80 controller series, in which controller serial numbers for both paths of a dual path system are available when one identifies the device.

Controller supplied settable identifiers
3. The HSZ70/80 and HSG controllers will supply a customer chosen number which can be used as an allocation class for units controlled thereby.

SCSI-3 and Fibre Channel are expected to eventually provide a non-volatile WWN for each storage unit. This will be accomplished via something like an Ethernet ROM in each discrete drive, and in each RAID controller. The WWN for a virtual RAID storage device is expected to be a concatenation of the controller's ROM ID, a timestamp, and an incarnation number, stored with the unit's metadata.

A multi-ported device is required to return the same WWN over all of its ports. This means that the WWN can be used as the mechanism for detecting multiple paths, and as the basis for a path-independent device name.

Multi-port SCSI devices are expected to be implemented in a variety of ways. At least the following three characteristics are expected:

Failover
1. A particular logical storage unit is available for reads and writes on one path at a time. The device stays on that path until the host performs an explicit operation to cause a switch. It is highly desirable for such a device to be capable of answering Inquiry commands on either path at any time, so that the operating system can detect and monitor all the possible paths. It is also desirable for this type of device to provide Unit Attention when its status changes (i.e., when failover or failback occurs.

One Path At A Time
2. A particular logical storage unit is capable of being active on any path, but only one path may be active at any time. In this case no special action (such as SCSI START) is needed to switch paths.

Simultaneous Access
3. A particular logical storage unit is capable of simultaneous access on any port. Access can be mixed in any fashion desired, or switched in any fashion desired.

The Hardware Reality

HSZ40/50/70/80 controllers: There is firmware in field test now that supports multi-path failover on the HSZ40/50/70/80 controllers. This firmware provides the following functions, in no particular order:

1. All controllers can process an INQUIRY at any time for each device attached.
2. Only one path at a time may actually use the device; the others receive DEVICE BUSY indications.
3. Switch-over occurs whenever a SCSI START command is seen by a controller which is not currently controlling a device.
4. INQUIRY data tells the controller serial number of all controllers able to access each device and the current owner. This information is presented even if one of the controllers is down, so long as the HSZ is in multipath mode.
5. When a controller fails (or reappears), any I/O on the other controller is returned with UNIT ATTENTION and some Digital specific indicators that failure or recovery of a controller has taken place.
6.When a switch occurs, all commands that were queued to the failed controller are cancelled. (This is in fact not used in OVMS in this proposal since we will switch on mount verify or after sending IO$_PACKACK only. The PACKACK function ensures that all outstanding commands are drained through the system before completing, by timeout if not by actual execution.)

HSZ80 controller: The HSZ80 is expected later this year. It is expected to support the same functions as the HSZ40/50/70 controllers. It is expected to support real world wide IDs.

HSZ22 Controller: The HSZ22 is expected later this year. It supports fully simultaneous access on its two host ports. It is expected to support real world wide IDs.

The HSG controller: The Fibre Channel controller is expected late this year. It is expected to support worldwide IDs (WWNs). It will be used as a failover device.

Seagate Fibre Channel Drives: One-path-at-a-time, real WWNs.

Naming

These naming conventions will require a complete solution to naming to use the worldwide IDs in final form. This document does not address this full naming form, since the necessary time to produce such would be a schedule problem. Rather, to obtain support somewhat earlier, the HSZ hardware will be given some additional abilities. For initial support, HSZ controllers will return a customer-chosen number which can be used as an allocation class. Use of WWNs or use of controller serial numbers as a practical matter requires the same handling...a system which might use configuration files coordinated by cluster communications...which is to be built later. To avoid delay in support for HSZ hardware, this system will use the supplied allocation class from the device itself to ensure multipath controllers have the same name. For testing and perhaps a bit of old version support, a means to set port allocation classes the same for all paths to the same devices will be created also, subject to various configuration and use constraints.

The switching system will use this HSZ feature to support the HSZ series early, and the separate naming work will build the infrastructure needed to totally virtualize device names to support WW IDs in final form.

Design Goals

Highest priority:

* Failover for non foreign-mounted disks, including system, shadow, and quorum. Support for HSZ* is the highest priority.
* Support for the failover model only, rather than attempting any sort of simultaneous access. Dynamically balancing loads is not a goal at this time.
* Alpha only.

Medium priority:

* SCSI to MSCP switching

* MSCP to QIO server switching. (This will ultimately be needed also on VAX.)

Low priority:

* Foreign-mounted disks. We will support disks on which mount verify works at first, and extending beyond that model is an extension that may be added in the future. There will be however a command to manually switch disks which will work for idle /FOReign mounted disks.

* Changes to make I/O intercepts more efficient. Some general changes will be needed for this project, but a formally defined intercept interface will not be added.

* Foreign (3rd party)controllers. As will be seen below, certain vendor unique commands and status information can be used to make the switching more flexible and responsive. These will be supported for HSZs, but similar support for third party controllers are quite low in priority, as they would each require new special case code. The generic "switchover at mount verify" model will be available, but the design will not go to any lengths to support arbitrary controllers with special case code.

* Tapes. Tapes need different status preserved than disks, and tape mount verify is currently far too slow to make switching tapes attractive. For HSZ connected tapes, the name altering work which will ensure all disks have the same name for the same device will also ensure that only one UCB exists for tapes in that situation and that their names are taken from the HSZ allocation class.

Not planned:

* VAX support, other than as described above.
* Not all SCSI adapters may be "fully" supported. For switchover to be useful, all I/O sent to an adapter must eventually either finish or time out, and not leave the adapter or driver in some hung state, and if a hardware failure is fixed, the port driver must be able to resume operation. If either of these conditions does not hold, a path can become unusable until the next boot and the switching code can do nothing about it. Such conditions cannot be exhaustively found, so that support will have to be declared as potentially vulnerable to such should they be found..

Design Overview:

This design assumes that all paths to a device will have the same OVMS device name. The method for ensuring this will be discussed later.

When a system detects a device for the first time it checks to see if there is a already a UCB for it. If not it creates one. This will be the "master" UCB for the life of the system. All I/Os will be issued to this UCB, and from here they may be redirected. The job of the switching code is to allow this path to represent a tuple of one or more paths and act as their global alias while retaining for debug purposes data on individual paths.

If a system detects that there is already a UCB for a device name, it creates a new UCB, but this UCB is (*must be*!) hidden so that system search and I/O database scan code will not find it, though the switching code will use it. This can be accomplished by moving the "extra" UCBs off the main I/O lists so that only switching code will know where they are, and of course those who can interpret its structures. SDA will need to be taught this. The intercept linkage will provide means of going in either direction between intercept UCB and the UCB intercepted. However, UCBs are linked in many places, and these links are used by components all over VMS. Therefore a new flag will be added to the UCB which I call HIDEUCB; this can be placed in the DEVCHAR2 field. The scan_iodb and searchint routines in iosubnpag will be taught never to return UCBs marked with this marker. The SWdriver code that sets up a UCB as a second or later tuple member will set this flag, thus hiding it. Note that DUdriver does not seem to use the scan_iodb or searchint routines, so that its processing should be unaltered. Because the internal I/O lists will not be altered, SDA changes can be avoided also.

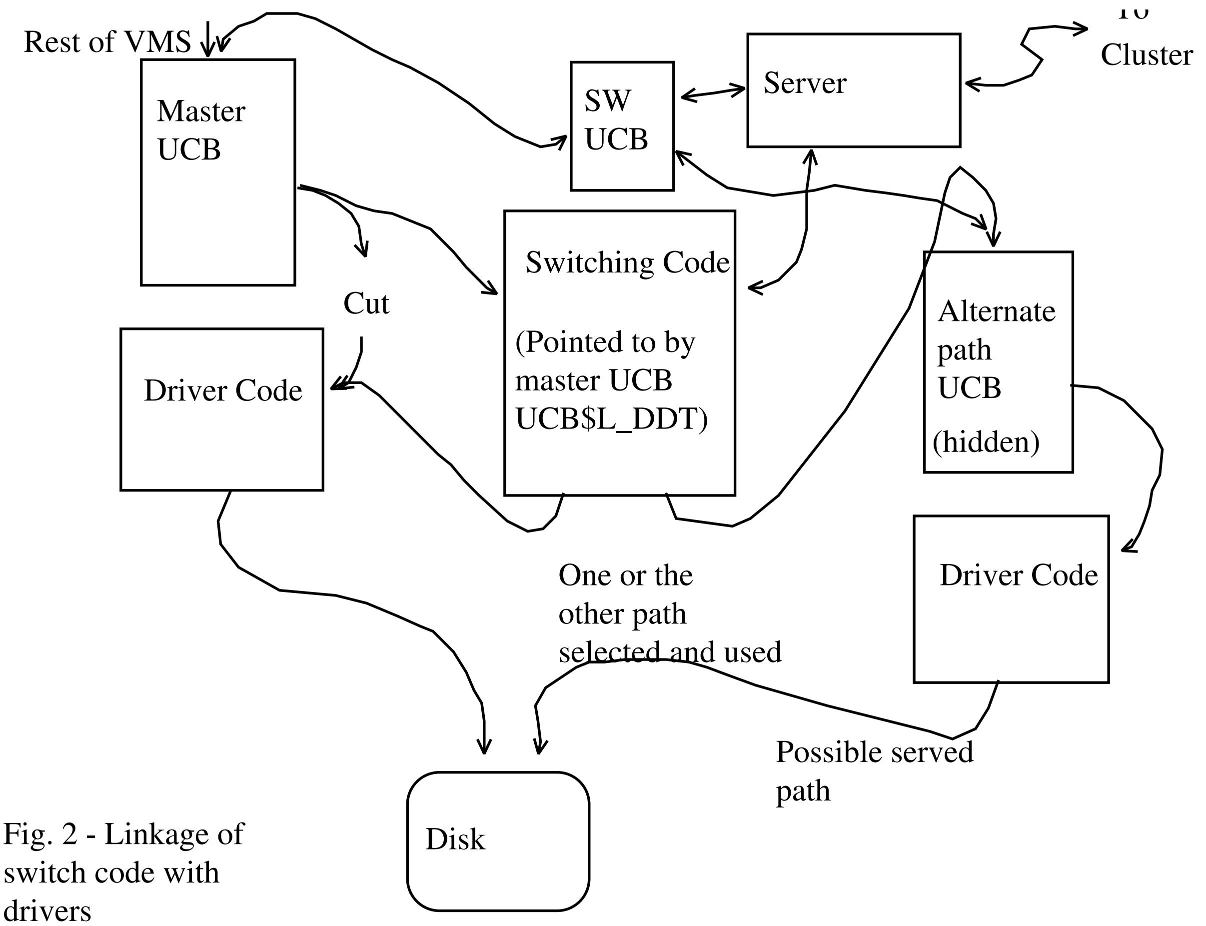


Fig. 2 - Linkage of switch code with drivers

Note that the **first** UCB created might have pointers to it from other areas in the system. All other UCBs must be hidden upon creation to prevent anything from gaining a pointer to them other than the switching code. Code to do this will go into the driver loading routines and other areas which create new UCBs. In the case of SCSI and MSCP, this code resides in DKDRIVER and in DUTUSUBS.

Switching Code Role

The switching code gains control at the driver start_io entry (and at a number of others, but it must see each IRP at driver start_io) and directs the IRP to the chosen path by either calling the original driver code (if the path used is using the "master UCB") or by sending the IRP to the correct alternate path's UCB and driver code. I/O goes through a "master" UCB so that any pre-existing UCB pointers to the first UCB will always be valid even with switching inserted in each driver. Such pointers can easily be present for quorum disks, shadowed disks, system disks, and the like, even from early in the system's life, since the system may have booted one such or may have begun using one path before any other(s) could be configured.

The switching code gains control before an IRP enters its underlying driver and again when the IRP is posted so that it can "see" error returns and react to them.

When a system detects an error on a path, it first attempts to categorize it as being remediable by path switching or not. In the former case, if the device is mounted such that mount verify can be entered, it will arrange to place the master UCB into mount verify state. Mount verify does not always mean a path has failed; in SCSI disks it can mean only that someone has rebooted or a device hung and caused DKdriver to issue a RESET. Therefore only after a few failed IO$_PACKACK IRPs are seen from mount verify does the path switching code try the next enabled path. (A larger number are required immediately after a path switch, to prevent ping-ponging.)

In cases where a controller failure is signalled to the driver as part of an otherwise successful I/O, the switching code will send a message to a server, which will exploit additional interfaces to the switching code to switch idle paths which may now require it. An outstanding operations count, incremented when IRPs enter a path and decremented when they are posted, identifies idle paths. A number of features of the server are facilitated by making the switching code look formally like a driver; thus it will be so described.

It may be added that "unusual" functions (like IO$_DIAGNOSE for SCSI) can be supported by permitting their use provided that either the functions are "universal" disk functions (such as MSCP already supports) or the FDT addresses which would have been called are also correct for the underlying driver. Otherwise such functions must be blocked in the switch layer.

Mount Verify Flow

Note that shadowing uses calls to SHDRIVER instead of to EXE$MOUNT_VER and does not need to have IRPs requeued (since SHdriver has the original IRP and issues new I/O) in the following. However, the processing is basically similar and will be described below without the added complexity. The code will of course deal with both types of mount verification so that shadow set members will be able to have paths switched also.

The way mount verify occurs in DKdriver with this code in place is the following:

1. An IRP comes to the switching postprocessing with a status that should start mount verify, generally SS$_MEDOFL (media offline).
2. The switching driver restores the IRP context so IRP$L_UCB points to the UCB the rest of VMS knows, IRP$L_PID is the original PID, and so on, and calls EXE$MOUNT_VER to see if mount verify should occur. If so, this call generates a call to the driver mount-verify-start entry, which is fielded initially by switching code.
3. The switching code calls the correct underlying device. In the case of DK, DKdriver lets all other I/O finish and sets its UCB Busy bit before returning. so that subsequent new I/O is suspended. (Old I/O that completes in the meanwhile finds mount verify in progress, so is

eventually all requeued in this way.) Thus by the time MV is initiated, the device's I/O has completed somehow, via timeout if not via any other means. It is ***necessary*** that underlying I/O will be timed out or otherwise completed, or will have timed out in devices, when mount verify has run. Failed I/O will be requeued ahead of any that may have arrived later. (Note that disk subsystems do not promise in-order command execution.)

We rely here on I/O finishing eventually, and on the environment guaranteeing that I/O will be drained or removed when packack is issued, and that new I/O is blocked at this time. Errors that hang the I/O stream or that cause controllers or drivers not to come back can cause the device I/O to hang or a path to become unusable. In the latter case, however, the system can continue to use still-usable paths.The switching system will provide an increment in function in these cases, though it cannot guarantee against device hangs in underlying hardware.

4. If the UCB of the underlying device is not the main UCB, the IRP in question must be moved to the primary UCB's input queue, putting it ahead of any that may have arrived meanwhile so that order is preserved, and is by the switching code, so that when the dust settles all I/O is back on the primary UCB input queue.
5. Now mount verify's thread starts issuing packack to the master device UCB, which the switching driver sends to the appropriate underlying device (it uses initiate, not insioq, for this). After a few of these packacks are counted, the switching driver decides it's time to switch sides, records the time, and sends the next packack to its "next" enabled path. If this packack succeeds, mount verify will eventually end.
6. At MV end, the underlying driver MV end entry is called to ensure that the underlying driver is ready to accept I/O, with IRPs in the idle queue configured for its queue and queued to it. This restarts I/O for the path. (A timer is also set up so the switch code can delay longer in switching again after this switch.)

Note that for mount verify, it is not necessary that the device be found idle initially; nor will it generally be the case. Rather, at this point the mount verify processing itself idles the device by letting I/O finish (in DK's case). (Other drivers may be able to abort I/O and have it be requeued in that way. As long as the MV start/end entry points are used as described, however, the switch code will work.)

The difference from normal MV start and stop is that paths are changed after MV has been in progress. It is assumed that MV in progress means that I/O has been stopped, and DKdriver relies on the ability to time out I/O eventually to provide this guarantee. In addition, the use of PACKACK along the new path before any new I/O is released ensures the path is up and ready (and in the SCSI case is a sequential command which would ensure all other device activity has finished, though due to the single streaming before MV starts this is not actually relied on).

It is necessary that EXE$MOUNT_VER be called with the IRP looking like a normal IRP for the master UCB in order for all this to work.

For shadow driver member UCBs, processing is generally similar except that the switch code must call into SHDRIVER to perform the mount verify operations, must redispatch SHDRIVER packacks, and need not requeue IRPs (nor is MV start called) since SHdriver does not expect or need this. Rather, SHdriver will issue new I/Os when appropriate, so nothing need be restarted either.

The Switching Driver

The switching code is composed primarily of a number of routines which must reside in an execlet plus a number of controls which are most convenient to implement where it acts as a driver as well. The main part of the switching code is inserted ahead of the master path's "Start_IO" processing. This is accomplished by replacing UCB$L_DDT with an edited DDT for the master path which points to the switching code. The switching code saves a few IRP fields (so they can be restored later), ensures the IRP is correct for the path to be used, arranges to gain control at I/O postprocessing (at IPL 8 without any detour through IPL 4 if possible), and transfers the IRP to the driver code of the currently active path.

Where the switching code detects mount verify PACKACKs, it counts them for the path (zeroing the count if other activity is seen) and when the count exceeds a threshold, it switches paths and allows the PACKACKs from Mount Verify to traverse alternate paths instead, as long as they are marked "enabled". The threshold is kept high for a time after a path switches in order to prevent re-switching contention. (If commands to switch a controller to a new path occur too often, the controller may be unable to ever complete a switch. It is essential that enough time be allowed for hardware to finish switching its path. Hence, right after a switch of path, the system is inhibited from switching again for a longer time, even if the path seems not to be any good at first.) In this case, mount verify is presumed to have ensured that outstanding I/O along a failed path has been quiesced.

The switching driver also must handle cases where switching is done as a result of notification of controller failure or possibly external command. First, it will contain code which will recognize these events when signalled by a controller (e.g., an HSZ) and send a message to a server which will be pre-registered with it when this happens. Since these events will come from one device but refer to others in the system, it creates a layering problem to embed any path changing code into switching code whose focus is on one tuple of paths. By having a server do this work, the necessary globality can be achieved, and the server itself can be user mode code, simpler to debug and able to use all OVMS services for its purposes. However, this server must be able to command the switching code to switch a path, and the switching code must be able to accomplish this, and to support what the server needs.

This support includes having the ability to track when I/O is busy, to be able to stall I/O on a path when switching is needed, and to perform the necessary switch. It also must include the ability to allow an IO$_PACKACK to be sent (without UCB busy bit synchronization) along the path to be switched to (which in turn will flush out any outstanding I/O and will ensure that the path can actually be used by (among other things) sending SCSI START to it. The switching software will be able to block additional I/O by re-entering an IRP on the master input queue and returning without further dispatching the IRP. This will not require any alteration of the UCB busy bit, which underlying drivers manipulate. By counting IRPs that have been sent to a driver and not yet posted, the switching code can tell when I/O is still pending on a path, so that it will switch I/O only when the path is inactive. Finally, the start_io entry of the switching driver is perfect for converting IRPs containing the IO$_PACKACKs needed by the server into IOC$INITIATE calls to the appropriate underlying driver when it is idle, so that no code need be written which would have to generate a new IRP completely. (This must be synchronized so that keepalive PACKACKs will not interfere with mount verification;

such synchronization is simplest within the switching driver itself.)
To support use of switching for shadow driver component devices, the switching driver will also need to sense that its path represents a shadow virtual unit (there is a UCB bit to tell this) and call the shadow driver back in lieu of calling EXE$MOUNT_VER in that case.

One other duty of the switching driver will be to ensure its server is functioning and if not, to start it.

In addition, the switching driver contains all modules needed to manage switching data and to allow other system routines determine the state of paths. This is handled by a routine callable from high IPL which is pointed to by the switching driver's DDT mount verify entry (which is otherwise unused since the SW device is not mount verifiable). This routine accepts an argument block which permits such functions as connecting a new device to a path, removing a device from a path, telling whether a device is intercepted, reporting statistics on paths, returning addresses of the use-order and enable/disable information for paths (so that an external routine can disable certain paths from use or specify the order in which paths must be tried), connecting the server to the driver, or disconnecting intercepts. This provides a system callable way to duplicate functions also present via the IO$_PHYSICAL function calls to the SW devices.

The Switching Server

The switching server resides in each cluster node and is the piece of the system which knows about the global state. It executes in user mode for the most part but knows about the "back door" entry points to the switching driver and uses them for its functions.

The initial version of the switching system will arrange for all members of each "tuple" of paths to have the same device name from initial driver load on as described below. Accordingly, the switching server will not need to know what names to assign anything, but it will play a role in setting up these paths and will perform various maintenance operations on them.

The server's duties are as follows:

1. Maintaining "keepalive" monitoring of paths.
The server must send periodic I/O to each local path it has to a bus, so that if a failure occurs it will be dealt with promptly if possible. This I/O is directed to each path via "back door" redirection in the switching driver which recognizes requests for particular paths and directs them as needed, regardless of the path being used for all other requests. These are expected to be SCSI INQUIRY commands and will be sent only once every minute or few minutes via the IO$DIAGNOSE interface. No such keepalives are envisioned for non-SCSI disks. (The inquiry case will handle local SCSI disks. Other keepalives for other cases may be added as needed.) It is envisioned that path failure notice will give rise to OPCOM messages from the server also. These INQUIRY messages will be sent (in the initial version at least) only to devices which are local SCSI connections, since initially no transport capable of sending such commands over a cluster exists. Different keepalives must be selected carefully to avoid undesired side effects on

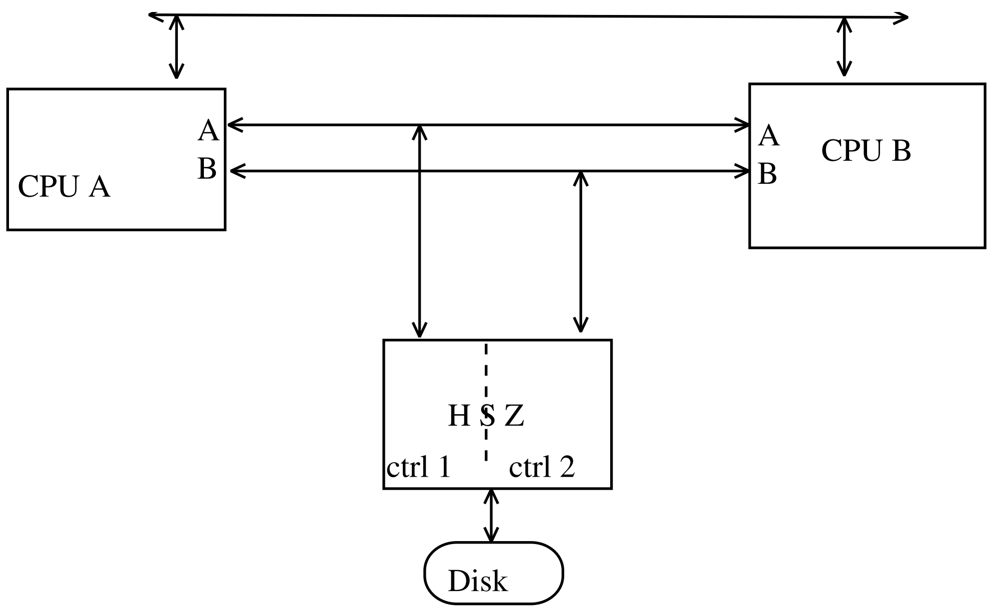
disk management software (such as jukebox control software which might think the keepalive meant that a platter must be moved into a drive, as a particular example of such undesired effects).

2. Handle known controller-specific path conflict issues.

The server must also use its internal information about multiple path devices to manage conflicts which are global in scope. It is able to enable or disable the switching code from using a path that is known to it in order to do this.

This is so because there needs to be a policy enforcer. For example, it is to be preferred that processors use direct paths to devices over served ones, but there are cases where this is not good for the system as a whole.

Consider, for a simple case, the following configuration, a 2 node cluster with dual shared SCSI busses, one each connected to a controller of a dual controller HSZ.



Now suppose that CPU A's A connection drops out and A and B switch to bus B. (A switches first and sends a START to the HSZ. The HSZ now switches over, and CPU B now sees its device connection on Bus A is unavailable and it switches also.) Now if CPU B's B bus connection drops out, we don't want to let CPU B switch back to bus A because that will cause CPU A's connection to fail, because the HSZ can only permit access on one bus at a time.

If we ONLY let CPU B decide based on its local connections, it could switch back to Bus A and cause CPU A to lose connection again, leading to a possible ping-pong between busses.

(In the presence of served links, of course, CPU B could fail over to a served connection, but we want to avoid pulling connections out from under working processors.)

To deal with a situation like this, the server process must know that when the bus A path fails on node A, it should not be used again until repaired. Now in actual practice, we do not get an unambiguous indication that a bus has failed all the time, and the SCSI code at least has not been written to allow one to tell when a bus is failed apart from when a device has failed or perhaps just hung persistently. Accordingly, the server will run code which looks at HSZ paths (it will know which these are from its INQUIRY data) and when a local HSZ path fails, it will mark that path disabled for a period of time (via a special function sent to the switch driver) and communicate this to the rest of the servers in the cluster so they will do likewise. The period of time is intended to be an adjustable parameter obtained in some straightforward fashion, perhaps a local process logical, but at least minutes. This solution will only be completely usable after booting, so that booting from a path disabled everywhere else will defeat it. When switching is initiated, therefore, and no enabled path can be found, the switch logic will disregard the enable timeouts and proceed with just its normal timers including hysteresis. This will prevent an excessive down time due to booting.

This kind of intelligence needs to be present for known devices, so that we know that for example the HSZ is a "failover" type device and requires this constraint.

Notice that if CPU B were booted from bus A, while CPU A was still unable to access Bus A, there would be contention, but in general CPU A would get blocked, perhaps until CPU B started serving the disk, and then A would be able to switch to that path. This is at least a stable configuration, but is not as good as having both switch to use Bus B instead. This is why the switching driver level code needs to be able to idle a path also. The server can be set to detect a situation in which more processors could access a device directly, and then by commanding a path to be idled and then commanding a switch, the server can force a more optimal solution. While all of this complexity is not planned for the initial version, it is important to allow for it architecturally so that such intelligence can be added incrementally.

(If the switching server were not present, the switching driver could only move round-robin to different paths. While this would likely settle on a served path for one node, the time to reach stability may be long.)

The initial server will not implement any sophisticated switching policy, but simply allow round robin access to all paths as the driver will perform itself. However, it provides an architectural location for this function when needed.

3. Orchestrate the failover of paths found to have gone bad, or failback when they are fixed where HSZs report that this has happened.

This feature is a speed optimization, in that an HSZ can tell the driver for disk 1 that the other controller has failed. The server will have a global enough view that it can switch disks on that other controller if they are idle, thus avoiding the mount verify delays. As it is not strictly needed for correct operation, this feature is low priority for the first server version.

The server will use its "back doors" to the switching code to order switching of devices whose paths have failed. Should the devices be busy at the time, it is expected mount verify will start (and in most cases will already be in progress) to switch the busy paths. Failback at least for HSZs will work the same way. However, when the server detects a failed path, it will ensure the switching driver's hysteresis timeout is dropped so that mount verify can quickly switch paths too. The server will receive failover and failback messages when the switching driver level code detects HSZ messages have been received (as a result of a small bit of relay code in DKdriver to signal this when it occurs). This function is driven off the receipt of those messages as well as its own polling activity.

Other Processing Needed:

1. Locating sets of devices that are paths to the same device.
When a new UCB is being created, the code must look for the switching driver, ensuring a unit is loaded if necessary, and connect the new UCB to the first. It is essential that any code creating a new UCB for a path check that no other UCB of the same name exists already; this is already done in DUdriver and DKdriver. If a pre-existing UCB is found, the code creating a second (or later) one must hide it so that no channels will be opened to paths other than to the first. This search code can conveniently be added to where load_driver is called.. Since a new DDB may need to be created, it is most appropriate to hook new driver units in above load_driver. A routine which can be called whenever a new device is added of an interesting type (i.e., for the moment, a disk) is available which will search for any pre-existing UCB and hook the new UCB in with the switching code if one is found of the same name. In addition it will create a new SW unit if one is needed (via cloning a UCB).

This is addressed in two places in the code base. For the boot drive, the DDB has initially only one class driver unit, so it is sufficient to have the DKDRIVER unit init code test its INQUIRY data to see whether it is booting an HSZ of the type containing a hardware allocation class, and if so, to simply have it store that allocation class in its DDB before setting the runtime drive online.

For non boot devices, it is necessary to do the check in IOGEN. The iogen-scsi-config module will initially ensure SWdriver is loaded (loading it if it was not) and check INQUIRY data to see if an HSZ device allocation class exists before loading each class driver unit. If so, it will send this allocation class (biased by 20000000 hex) to load_driver, which will strip the bias and ensure that the DDB is created with the needed allocation class. The loaded class device will also have its DDB port ID filled in and have the controller letter folded to A. The device allocation class overrides all other allocation classes. It must be in the range 1-255 as the code currently exists. This avoids many issues which arose with Port Allocation Classes since a PAC could have value zero.

Once the driver is loaded and given the correct allocation class, a utility routine which is added inside IOSUBNPAG is called to check for a pre-existing UCB of the same name. If one is found, a routine in SWDRIVER is called to add the new UCB to the path tuple for that name and enable its use. (This routine sets the first UCB up as the master name.) The code within SWDRIVER itself, once it has recorded the new path, hides the "extra" UCB. (For testing this is done by changing the device class to DC$_MISC. Later it will flag the UCB with a HIDEUCB characteristic in devchar2, and the scan_iodb and searchint routines will be altered to respect this flag and not return any hidden UCBs. Also the NO_ASSIGN bit will be set to prevent the UCB from being used.)

In addition to autoconfig, it is necessary to deal with SYSMAN IO CONNECT commands. This requires changes to SYSMANIOGEN.C which issue a READ to the port driver being used (which generates an INQUIRY) if the device is SCSI, and code to test the result for an HSZ allocation class and pass this to the driver load routine. This ensures that the DDB created will reflect the correct allocation class and prevents incorrect DDBs from being used.

It is also necessary for code in DUTUSUBS and DKDRIVER unit init, which in V7.1 checks for same named served devices, to call the switching "hook" routine mentioned above to allow failover to be done between direct and served paths automatically. When QIOserver is written, similar code must be added to check for its paths and ensure, too, that these are linked in.

Finally, SWdriver manipulates the MSCP controller model and serveable bits in the UCB seen by the rest of the system so that when its path is not a directly connected one, the device is seen to be not serveable, and serveable when a direct path is in use. The MSCP server needs one change in the ONLINE routine to test this so that it will not attempt to put a device online to a cluster if it is serveable. Formerly this needed to be tested only at MSCP initialization, but with switching, this became dynamic.

2. Manual switchover, enable, or connect command.
It will be necessary to have a command which will tell the switching system to fail over, to idle I/O through it (needed for failover), to disable or enable a path, or (for testing at least,and possibly for emergency situations) to connect switching code between some devices. This processing will be added to SET via some convenient means. Whether a new image will be needed or the code will just be added to existing code is undecided, but the bulk of the actual work will be via calls to functions in the switching driver itself, which will have all that is needed to connect I/O built in. Corresponding SHOW commands will be needed, as will commands to show added information about individual path errors, I/O counts, counts of switches of paths, and the like. A new image is provided which can set up or tear down path tuples (though teardown is for testing only) and which can command manual switching of a path and can cause SWDRIVER to idle or unidle a path.

Modifications to Existing Code Needed

The switching system is designed to require minimal changes in other parts of OpenVMS. Nevertheless, certain support for it will be needed in various places, notably:

* DKdriver:
DKdriver needs to be fixed to do in its start_io loop whatever is decided upon for the fix to REQCOM which is needed generally (see below). This will allow all IRPs to be seen at the start_io entry or at altstart_io by the switching system. DKdriver also must be taught to recognize the HSZ failover and failback sequences and signal these to the switching system. Finally, the DKdriver code in unit init which finds pre-existing paths must be altered somewhat to use a new "hide this UCB" flag instead of the dual path flag now used, and to search for any kind of same-name connection pre-existing, so that the hiding done will not be specific to MSCP served paths or to only two paths. The actual "hiding" is done in the SWdriver routine that sets up a device in a switching tuple. For initial testing, this is implemented by setting the

device class of the "hidden" device to DC$_MISC and setting the "no_assign" bit of the device to prevent access to it. Ultimately a "HIDEUCB" flag in UCB$L_DEVCHAR2 will hide the UCB by means of the scan_iodb and searchint routines being altered to never return a UCB so flagged.

* DUdriver
DUdriver's code (actually in DUTUSUBS) which recognizes pre-existing paths will need to know about the new hidden-UCB flag also and should be taught to use this flag as well to hide its extra paths where not dealing with a strictly-MSCP situation. The new routine to find and hook in a new device into a switching tuple will do this work.

* IOSUBNPAG and SYSQIOREQ will need to have the I/O posting shortcut fixed so that it distinguishes calls to REQCOM from calls to things like CALL_POST in continuing next I/O where appropriate. SYSQIOREQ will need to set up a new IRP "argument stack" and should in the process store the original IRP$L_PID setting in an IRP cell that is not dual purpose.

* SHOW or perhaps some new image will need to be able to display statistics for errors, I/Os, and so on for individual paths on request. The first UCB seen will accumulate all this information, but for diagnostic purposes it will need to be separately available. The thought is to include this information in the switching driver UCB and provide a path for it to be accessed. The control image already contains reporting functions. As currently constituted, it merely reports paths by number, since all are named alike, and gives I/O and error counts for each and which is current (and last switch times).

* SDA will need to at least know about the new "hide this UCB" flag so it can be displayed symbolically.

* SET will need the functions described above.

* MKdriver needs to prevent duplicate paths from being loaded. (Currently contemplated code merely attempts to disable second and later paths.)

* DRdriver, TUdriver, and MDdriver may need to be modified to do like-name searching if they come to be switched in similar ways to DK.

* An IRP field needs to be set up to hold pointers to context information.

Naming Setup

The switching system includes some code to be added in the boot path also which will ensure that multipath devices receive the same names globally. The method presented here is not the "final" one which will eventually be needed. Ultimately we will have virtualized names which will be obtained from worldwide IDs, using a configuration file and cluster name checking to ensure stability and consistency. For now, however, we will obtain device names from the devices themselves. (Note that this is how DUdriver handles these issues, and that served paths already have the same device names across the cluster; we need to ensure this for SCSI disks now as well.)

When SCSI class drivers are being loaded, IOGEN has INQUIRY data from them which will give the device type (so HSZs in multipath failover mode can be recognized) and will also have the dual path bit if the devices are SCSI-3 compliant (as fibre channel devices are expected to be). In this latter case they will also have a worldwide ID.

When one of these multipath devices is found, IOGEN will obtain from the INQUIRY data a "controller allocation class". This controller allocation class will be set at HSZ consoles (as it can be for HSC already) and the customer must ensure it is cluster unique.

Additional Requirements

Wherever I/O completes on Alpha with no delay, the call to REQCOM does a recursive call to driver start_io. This can overflow the kernel stack. DKdriver experienced this in its start_io processing and in V6.2 on alpha was changed to extract IRPs from its queue by hand to avoid the recursion and stack overflow.

Unfortunately, this broke every third party intercept of DK devices since it meant that an intercept of driver start_io would not see all IRPs, and no alternative exists. A number of caching products were the chief victims. This behavior also makes switching unfeasible unless it be fixed,and the fix will at the same moment cure the stack overflow problems experienced with every Alpha driver now. Several ways to fix it exist and are outlined in the detailed design description.

The switching system will require that the kernel stack overflow problems in REQCOM be fixed and will need a small amount of work in the IOGEN code to enable naming to be consistent and to allow use of existing hardware, but most of the system will be in separate, new modules which will logically become part of every driver needing their services. The server will initially be modest in size because naming responsibility need not be built yet. However, any disk paths needing failover, and able to be handled satisfactorily by mount verify, will be able to use these services.

It is desirable that in future code, however, attention be devoted to having well separated error codes which return where a path has failed vs. where a device has failed, so that distinguishing cases where path failover should be tried need not check error codes for a shopping list of such as is now necessary. Also, the naming problem solution will address the issue of very large numbers of paths in that it will not be feasible to configure all such on all systems. Smaller systems in particular will need to be able to configure only a subset of all devices which might conceivably ever be connected.

Summary:

The project will handle switching between multiple paths with code added just "below" driver start_io in the first path to appear to a storage device.

Switching will take place after mount verify has been active for a time, will have hysteresis to prevent thrashing, and will allow use of one path at a time only.

The name of all paths will be made the same, and whenever a path UCB is created it will be tested to ensure that if it is a second path, switching code will be called to mark it as such and get it off the normal I/O device lists. For this project the names will come from the device controllers.

Serving will be enabled where (and only where) the switch is "connected" to a local path, thus keeping from generating routing loops or other pathologies.

A server will do liveness testing and enforce policies about connection switch order, preferred connections, and the like.

The server will be loaded early in boot, and the name arbitration will take place early in boot, but after the cluster is up for all but the boot device, so that the naming work which is done will be upward compatible with a more general naming scheme to be implemented separately.

Drivers and the I/O subsystem will be affected minimally and I/O operations will incur only a small overhead as a result of the switching code.

Some accomodations to handle tapes and the like and permit manual path switching are part of the design also.