Multi Path Switching

Problem Statement & Investigation Report

Glenn C. Everhart

August 30, 1996

Problem Statement: A variety of situations exist or will shortly exist in which there will be more than one path to a device, and in which these paths will need to be coordinated so that only one is active at a time, and so that VMS sees only one, cluster-unique, name for this storage clusterwide, regardless of physical path to the storage being used.

This situation exists with SCSI clusters now, where direct SCSI paths to a disk exist at the same time as MSCP served paths. It will shortly exist with HSZ series controllers where these are attached to more than one SCSI bus in a cluster. It can be expected to appear in Fibre Channel connection topologies and in other intelligent controllers in the future. Moreover, when the QIOserver is introduced, it will define additional paths to storage which will need to be coordinated and to fail over.

To maintain file system integrity, each storage entity (disk, generally) in VMS must have a name which is consistent and unique across a cluster. To provide greatest system robustness and availability, it should be possible to switch from one path to another, at least when one fails, and possibly at other times.

This is needed for disks in the shortest time frame. Similar controls for other device types are desirable.

Goals:

* Support HSZ failover for HSZ7x type controllers where two SCSI busses are connected to a single machine.
* Support other multipath failover cases such as SCSI to / from MSCP paths or to and from other server types (*e.g., QIOserver*).
* Be compatible with planed HSG failover mechanism (which is generally similar to the HSZ one, with some differences due to the changes between SCSI-2 and SCSI-3)

Non-Goals:

* Code for more than two paths initially, though the code ***must*** extend easily to "n" paths.
* Support the case where both HSZ controllers are on a single bus (this is supported within the HSZ and needs no system support)
* Solve the device naming problem generally
* Dynamic routing or load balancing between paths to a device in full detail. (That is,it is expected that the solution must function to switch a failed path, but it is not initially necessary for a solution to load share between multiple paths dynamically in the first design pass.)
* Describe details of compatibility with the HSG proposed failover scheme.

Problem Components

There are two components to this problem: how to arrange that I/O be directed to the correct place (and how exactly to switch it), and how device names may be kept cluster unique where there is more than one path to a given piece of storage, and these paths' "natural" names are in general not the same. These will be discussed separately here.

Switching Solutions:

1: Non-Solution: SCSI Port-Class relinking

To deal with the problems only of HSZ, it was initially considered that some form of altering SCSI connections' purely SCSI structure based "routing" to devices might be feasible for the switching needed here. However, in principle, two SCSI busses can be controlled by entirely different SCSI port drivers, so that an attempt to alter connections on the fly at port level could involve considerable complexity in ensuring that port driver specific structures were initialized as the port drivers expect. (These initializations are not all alike.) Also, a "port level" approach does not deal with the appearance of multiple class driver units after autoconfiguration. That is, one must move not one, but several devices' structures such that no timing windows are opened up for the rest of VMS or for the SCSI subsystem. Idle drivers might be revectored, but any links between SCSI structures and class drivers would need to be traced and reset, and any future asynchronous events would need to be blocked from access to the structures during this time, and any port driver specificities in the SCDT in particular would be a problem. This can probably be done, but looked error prone and complex to build and to test.

Finally, this approach would not help with failover between SCSI and MSCP or other servers; it is peculiar to the internals of SCSI (as they currently stand), and represents an interface which has never been externally constrained to follow a set design. Rather, this interface has been mutable, and may remain that way.

2: Non-Solution: Add Code to DKdriver

It is possible to imagine adding switching code directly into DKdriver and possibly other class drivers which might redirect I/O as needed. In DUdriver, this approach is used, and DUdriver is quite tightly connected with the MSCP server and mount verify. In DUdriver, I/O is switched by requeueing, basically in the beginning of the start-io path, to the "other" path if such is needed. However, the details of most of the processing are heavily involved with MSCP, and rather complex. This part of the driver interface is a large and complex one and touches many places in VMS, so that an attempt to duplicate the DU processing in DK would be a sizable increase in DKdriver's complexity. This would introduce considerable risk into DKdriver and may in the end prove completely infeasible. (It is worth noting that this complexity and the difficulty of maintaining it are among the reasons for the qio server project.) The current DUdriver code does not in fact switch to a local DK path now, even when the UCBs and CDDBs are "cross linked", probably because the DU data structures are not completely set up in DKdriver (and don't exist at all in some other disk drivers). What is more of a problem, the path switching done between DUdriver, the MSCP server, and SCS is specialized to exactly two paths maximum between a piece of storage and a processor. When a new served path must be found, MSCP commands are sent to attempt this. While this is appropriate for MSCP storage, it is not particularly efficient for other drivers.

It is however important to consider that driver mount verification code must be called to ensure that related components be notified, as mount verification is the signal used in DUdriver that path modification may be needed.

A simpler variant of this approach would be to include some "generic" switching code early in DKdriver which would simple queue packets to other driver start-io entries. This would also increase the complexity of the DKdriver driver interface, but less so than attempting to duplicate DUdriver processing. While this is a feasible approach, it has two problems:

1. It would be specialized to DKdriver, and would need to be separately added to any other drivers in the future where failover may be desired. MKdriver and GKdriver may turn out to be early candidates.

2. Each DK unit "knows" only about its own device, where multiple path switching would need additional structures and controls to be added to the DK interface so that groupings of devices might be known. The class driver is not at the most appropriate level of globality to perform this operation, and again this would mean easily user visible changes to the DKdriver program interface, which would have to be do-able in any future drivers which incorporated the modifications.

Note that either of these approaches involves requeueing the IRP to the "real" destination device, so that the requeueing overhead exists in any related approach.

To the extent that DK-internal code lacks a well defined interface at its back end, it would tend to become specialized over time and would be likely to become harder with time to move to other drivers where it was needed. The lack of generality of this approach makes it appear less desirable than the final one.

The approach of adding code to DKdriver seems less than optimal.

3: Solution: Class Switching

The one interface in the I/O driver system which is well defined and constrained to be relatively stable is that at its top. Control transfer into drivers must follow conventions OVMS has long established and which are documented. Therefore, a switching function using this interface should be fairly universal, and may reduce the amount of work to keep it current as future VMS designs arise. It is a confirmation of the value of such a choice that the DUdriver failover control is managed very close to the "top" (i.e., start-io) entry of that driver also.

Therefore a simpler approach has been investigated. In this approach, a dedicated device switching subsystem inserts itself at the class driver external entry points and serves to abstract the device name used a level from the actual device driver paths. Thus, above this level, the OVMS system sees one device with one name which corresponds to the one piece of storage underneath. Below this level, two or more paths may exist, all leading to the same physical storage. This level's job, then, is to switch all I/O seen by the rest of OVMS (at a documented and fairly stable interface) between these underlying paths. It is not necessary that I/O order be preserved (since the SCSI drivers do not in general preserve it), except that in certain well defined circumstances such as path or cluster node failure, mount verify, and a few more, the guarantees offered by SCSI of order must be maintained. (See Figure 3).

This subsystem must offer a control interface distinct from the underlying drivers, which can work with components able to group devices together, select one name consistently across a cluster for the storage device, and allow communication to each underlying device for those switching situations when sequentiality across all paths must be ensured. (A server communicating with an intercept driver is the scheme in mind here.)

Note that the external VMS driver interface is the input interface of this intercept driver, since it will insert its processing in the start-io points of underlying drivers, and that its output interface is also this same external, documented VMS driver interface. These are documented and rather stable interfaces, and simple enough to control. While a separate subsystem may not have direct internal access to underlying driver private structures, this is a software engineering advantage. If additional information is needed, it must be explicitly made available and added somehow to the underlying driver's external interface, or some driver-private inspection method must be devised to aid processing in some cases. Path switching when mount verification occurs (as the usual case is for DUdriver) can be handled without additions to the external interface,since mount verify currently can operate with any driver. With a general purpose switch subsystem, however, switchover functions can be used for any multiple paths to storage, of whatever nature, and it becomes possible to create cluster consistent names for devices across widely divergent interconnects, by creating a dummy alias device name if need be and arranging that actual I/O pass through other paths. (This is a possible future item.)

It should be noted too that the MSCP server currently operates promptly when it sees mount verification and uses underlying facilities to find another server node if one is available. It would normally be expected that other servers may wish to do this as well. Should such support be desired, the switching software layer will need to send additional requests to a served path which will cause a new path to be probed for. The current suggested solution will allow the MSCP server's mount verify processing to happen before giving up on that path, so already it deals with MSCP.

Naming Solutions:

Background:

The path switching solution is related to device naming in its need to ensure that no device will be accessed via different names in the cluster. While this must be done uniformly across a cluster, the namespace can be the current one or any future one; other paths will be hidden (and the search and scan routines must be altered slightly to hide any N such hidden paths) but can exist. A path can come into being with its own name, and the path switching system can handle it under another name, even well after boot, so long as the name being hidden has not begun to be used. Apart from ensuring cluster commonality, path switching is largely independent of the choice of naming schemes.

There is one effect that is necessary, and that is that whatever is needed to ensure common device names clusterwide must be provided. This means in particular that when an alternate path comes up which is not the cluster-required one, some effort is necessary to see that it is not itself served to other nodes where the primary path exists and not used. The preferred switching solution can switch served path UCBs also, but the uniformly named cluster access path must be present and used first, so that only one path is used. This will require some additions in DKdriver to ensure that the "right" path is served, and alternate path access is delayed until the switching system is present. Figure 1 shows some possibilities with SCSI clusters and dual pathed HSZ which will illustrate some of the complexity involved. The picture with QIOserver and MSCP server added is different in detail but not different in concept. It is likely that the type of processing in DUTUSUBS and in DKdriver which looks for other paths to the same storage will have to be extended to a multipath case. While detail will change, the overall concept need not.

Cluster Consistency Strategies

To accomplish cluster consistent naming, it is essential that some method exist which will prevent use of extra paths, even if they are visible, before the switching code can start. Refer again to Figure 1 and it will be apparent that this is not trivial, particularly since a cluster cannot expect to have any fully shared storage. There are several approaches that might be used to achieve this.

Naming Solution 1: Common Heuristic
This approach involves use of the same heuristic decision algorithm in the entire cluster for determining which name to use. Thus, for example, where multiple controllers exist, there must be some way by which a driver coming up can find out whether the present UCB is for the lowest or highest numbered controller, and the algorithm would be "use the path with the lowest serial number controller". The proposed HSZ firmware makes this information available in its INQUIRY data. To deal with multiple paths for SCSI, it would suffice to add a test at the end of unit init to check whether this UCB was the lowest numbered controller one, and if not, hide the UCB so it will not come into use. Other controller services proposed for SCSI-3 will have similar capabilities, and a common heuristic will permit selection of the proper direct path. So long as servers do not start for nonpreferred paths, this will mean that the same, and common, device name will be seen everywhere. Similarly one could have a rule by which any server path being set up checks pre-existing paths and hides any new path if it is not preferred.

This will mean code in the MSCP driver to detect also QIOserver presence and vice versa, as well as to detect (as currently) local paths. Also code would be needed in DKdriver and any other affected drivers to check for whether a path was optimal, as well as code to hide other pre-existing served paths or to hide itself if another path existed currently.

Note: In the presence of switching software, initially configuring a served UCB as the "primary" name to be used is not as performance critical as might otherwise be thought, since any failover will permit the switching code to switch to a local path if one exists. It may even be feasible to force mount verify processing to achieve this after the switching code starts.

The common heuristic approach can work provided that devices with multiple paths can supply some per-path information to drive the decision. It does not require boot path modifications, and does not require intra-cluster negotiation. The processing involved is similar to what is now present. Because only one local path would be ever used to access a device for purposes of determining its cluster name, there could be no conflicts of names. Once other local paths were connected, of course, they could be made available via server, or one could serve the multipath alias device with suitable enabling modifications. The former, with corresponding MSCP and DUTUSUBS modifications, appears simplest since the latter generates routing issues with MSCP packets whose solution could be complex.

The approach does have a possible disadvantage though. That is that the port chosen by heuristic may turn out to be connected to only one system, where another port of some device may be connected to a shared bus, so that instead of instantly allowing two direct paths with no name conflict, only one is allowed, until failover to the direct paths is possible after loading of the switching components. This can be a temporary problem; certainly the HSZ firmware permits one (via sending SCSI START to the other port) to force mount verify to happen and failover to occur on the "less connected" port, so long as some global site of intelligence exists to determine when such should be done. Once the switching software is in place, a local path can be used even though the device **name** came from a served path. (See Figures 2 and 3 for the connectivity.)

The common heuristic in the HSZ case would be to use the controller marked as preferred by the customer, of course, this information being available also. Basically, however, so long as only one local device name can be chosen by local paths (and this is guaranteed on shared SCSI busses already), naming in a cluster will be unique.

Naming Solution 2: Common Configuration File

If every cluster node is able to access a common configuration file, and this file is required to be the same even if storage access requires multiple copies of it to exist, then such a file could be used to select which UCBs should be enabled. It would be necessary to also have cluster code check, at cluster state transitions, that all mappings agreed, and hang any node with a different map, which could involve a sizeable amount of traffic. This traffic cannot well be avoided, though, since where separate files are involved, they can get out of synchronization. John Hallyburton's IR on SCSI naming discusses some methods in detail for how this might happen.

Making a config file work would require that somehow, early in the life of the system, it should be read in from disk so that driver unit init routines could determine whether a particular device should be connected. Cluster state transition or something similar whch runs early in the system life must also compare this data for any disks being actively used at minimum, so as to ensure that no disk being used on a system differs in name from that same disk anywhere else. Identifying the "same disk" means using the unique device ID.

Cluster transition hooking can be accomplished by sending Worldwide ID / device name pairs to the cluster coordinator, to verify that no mispairings are seen. The size of this list is likely not to be excessive; perhaps 128 bits per device could encode the pairs, with 64 devices per 1K of memory, so that 2048 devices would occupy 32K. When a node was joining a cluster it could then send its pairings to the coordinator node which would validate that no different matches were used.

It is possible that the lock system could be used instead, within cluster code called from IOGEN, so that at boot time the device name validity can be established. In this case, locks could be taken out with names corresponding to devices, and lock values corresponding to worldwide IDs. Should a node acquire such a lock and see a different value from its preferred one, it would "know" that another node already running had used that name with a different device, and bugcheck after a message. This would need to live in the swapper process to allow it to live as long as the system,and could be implemented by a doorbell (to reduce the number of locks that must be constantly present) which the cluster coordinator would use, or by having a lot of reserved locks. (If cluster logicals were implemented and loaded early enough at boot, even those might be used.)

Recommendations re: common names:

For initial operation, the "common heuristic" solution appears much simpler to implement and would be well to implement first. This has the extra virtue that it may simplify backporting of this capability. Over the longer haul, it is likely that a configuration file approach will be needed to handle naming issues. The naming project needs to ensure only one name per device in any case, and is a better place to put this kind of information.

Even if a cut-down "new naming" scheme is used to only extend LUNs in a more or less fixed way (let unit number be, say, 400*SCSI ID + SCSI LUN, fixed algorithm for scsi3 and/or fibre), this means that multipath failover can accomplish what it needs to. If multiple paths never are configured, there may be only one path to select from, which is not bad either.

Defect Containment

The investigation has resulted in a driver and control suite already which will serve as a source of a code count. The software written for this purpose (not counting some library functions used to allow the optional configuration file to be free form) totals some 3216 lines of code. It is estimated that another ~250 lines of code will be needed for the automatic controller-pair recognition, and the DKdriver lines already added (to side copies) to support these functions total 180. Thus there are so far about 3400 lines of code and the total for HSZ failover functionality may be expected to total when all is said and done 3650 to 4000 (to pick a round number) lines of code.

The bogey number of defects expected in 4000 lines of code at one per 40 lines of code would be 100. However, for code which is unit tested already (the driver and control daemon code) this estimate is reported high, and an estimate of 10 defects per KLOC is suggested for that segment of the code. This would mean about 34 defects in the code so far, plus another ~10 in code to be generated.

Not all of this code is new (in that some older virtual disk driver examples were built on which have been functioning for several years) and the switching driver code has been tested in one system, which is why it is expected that a lower defect count will cover the code so far.

Methods for defect removal include (in addition to unit tests):

* Overall design - minimal modifications will be introduced into the (already complex) SCSI drivers to support the failover functions. This can be expected to be the chief contributor to defect containment, since the effects of changes to existing SCSI drivers form a small fraction of the overall effort and their function is limited to reporting information to the failover system on the whole.
* Reviews. It will be important to have the code in the driver reviewed so that its design, and particularly its detailed control flow, can be reviewed. The same goes for the server components particularly where privileged.
* Stress testing. The code must be tested in SMP and large cluster environments to catch any timing subtleties.

Resources:

Glenn Everhart will work on this project.

Issues:

* Details of the function of QIOserver have not been finalized at this time, and these may have some effect, although designs are being shared and the mesh currently looks good.

* Should the SCSI committee not ultimately support supplying information about all paths when querying a device path, the "common heuristic" approach cannot be used. This would be a change from current plans, but could happen. The approach will be fine for HSZs, though, and by the time such a change could occur, it is likely that the code for unique worldwide name support will have been completed, so that the configuration file approach should be workable.

Figures:

The following figures are illustrations of some of the types of connections which may occur, considering HSZ dual paths and their connections as particular examples.

Figure 1 illustrates two possible ways a dual-bus HSZ can be connected to a cluster (in addition to which the case exists of two busses on a single machine where the busses are not shared SCSI busses).

Figure 2 illustrates the way naming works to the rest of VMS, again using the HSZ system, which is one near-term dual path system, when using an intercept layer.

Figure 3 is meant to show the switch server in the picture. It is implicit here that the switch driver has interfaces for its control and communication which are distinct from those of any underlying drivers. (For that reason, controls do not need to be added to any underlying drivers of whatever sort.)

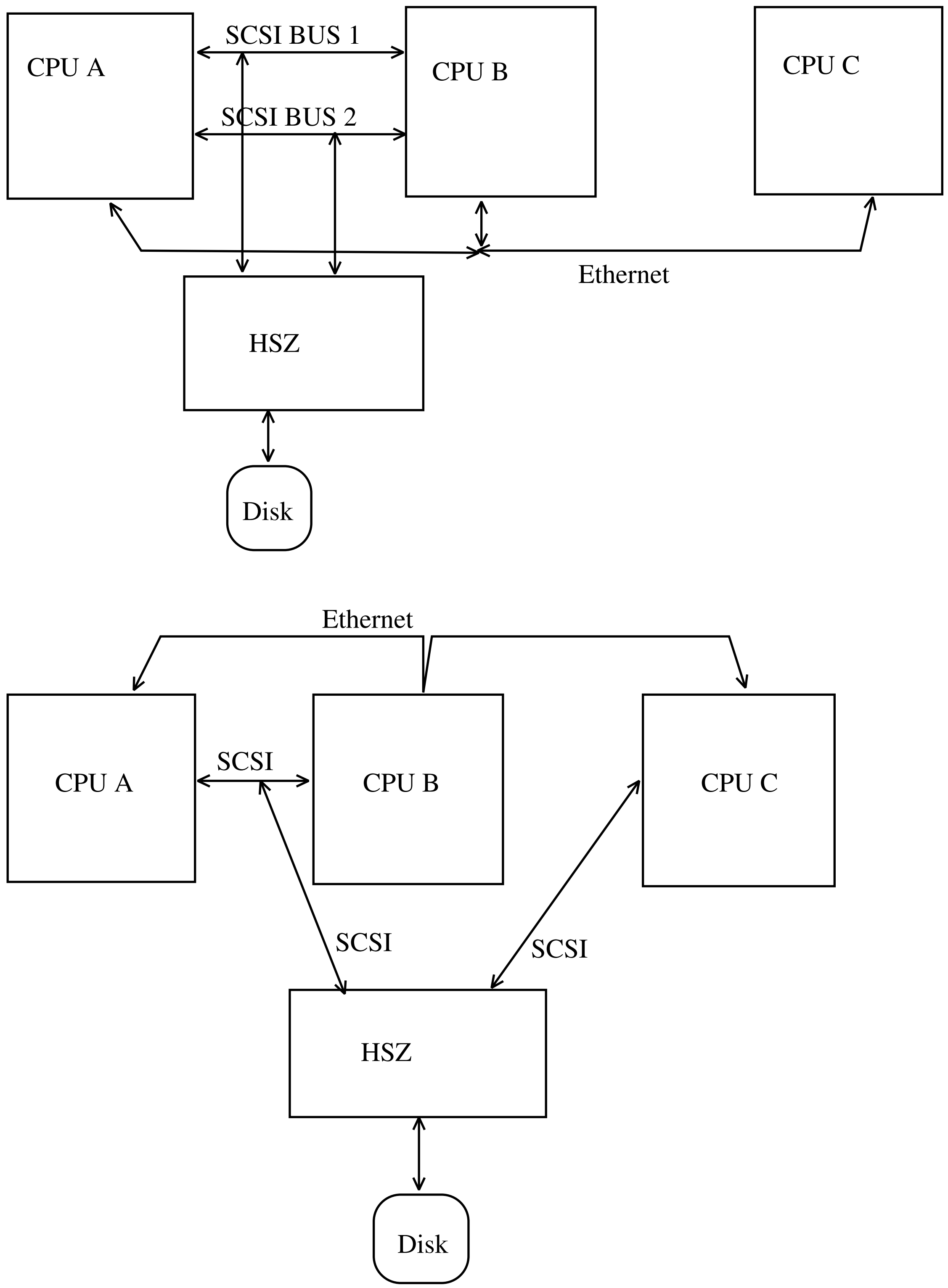

Fig. 1 - Some HSZ Configurations

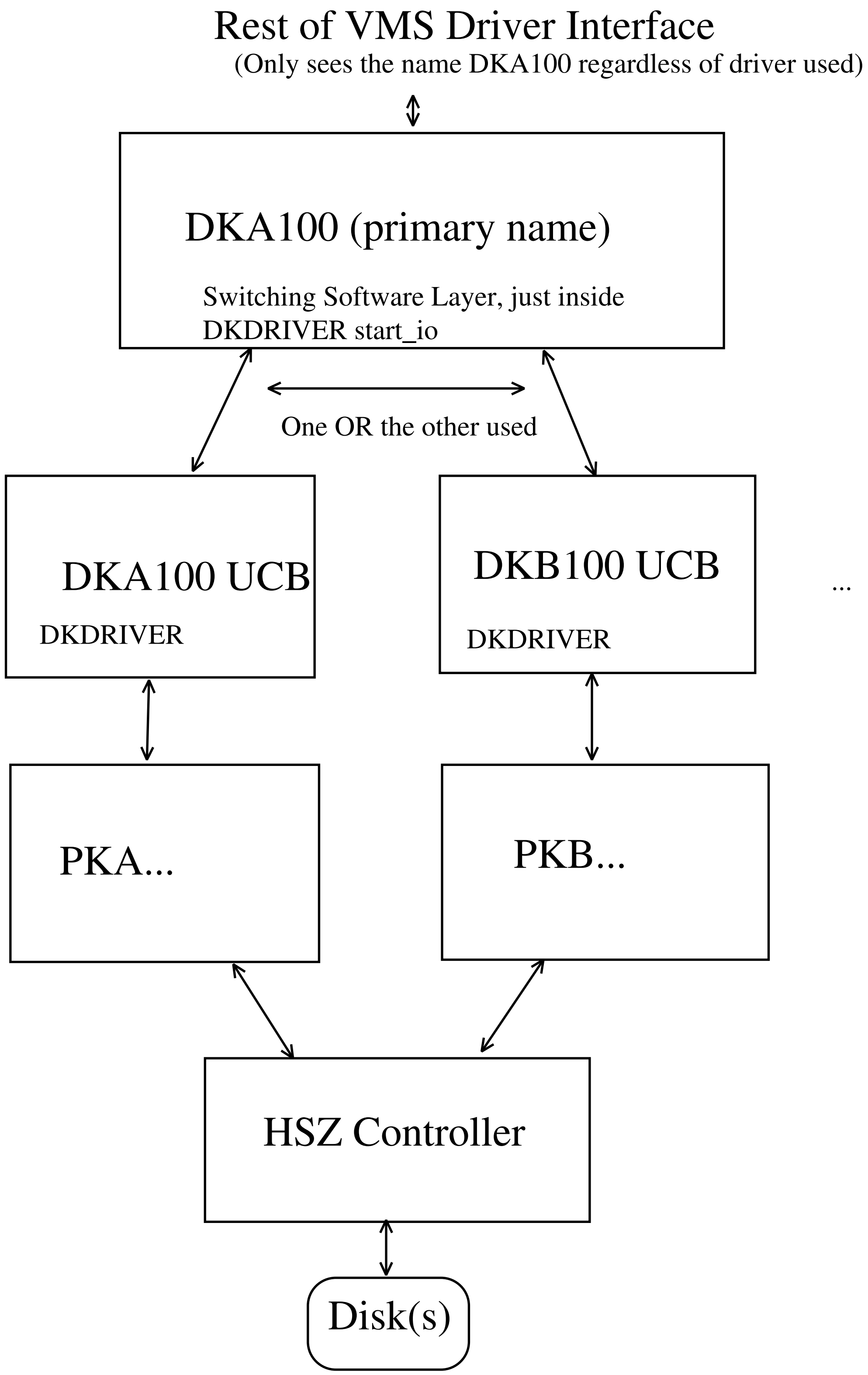

Figure 2. The Software Layering Used

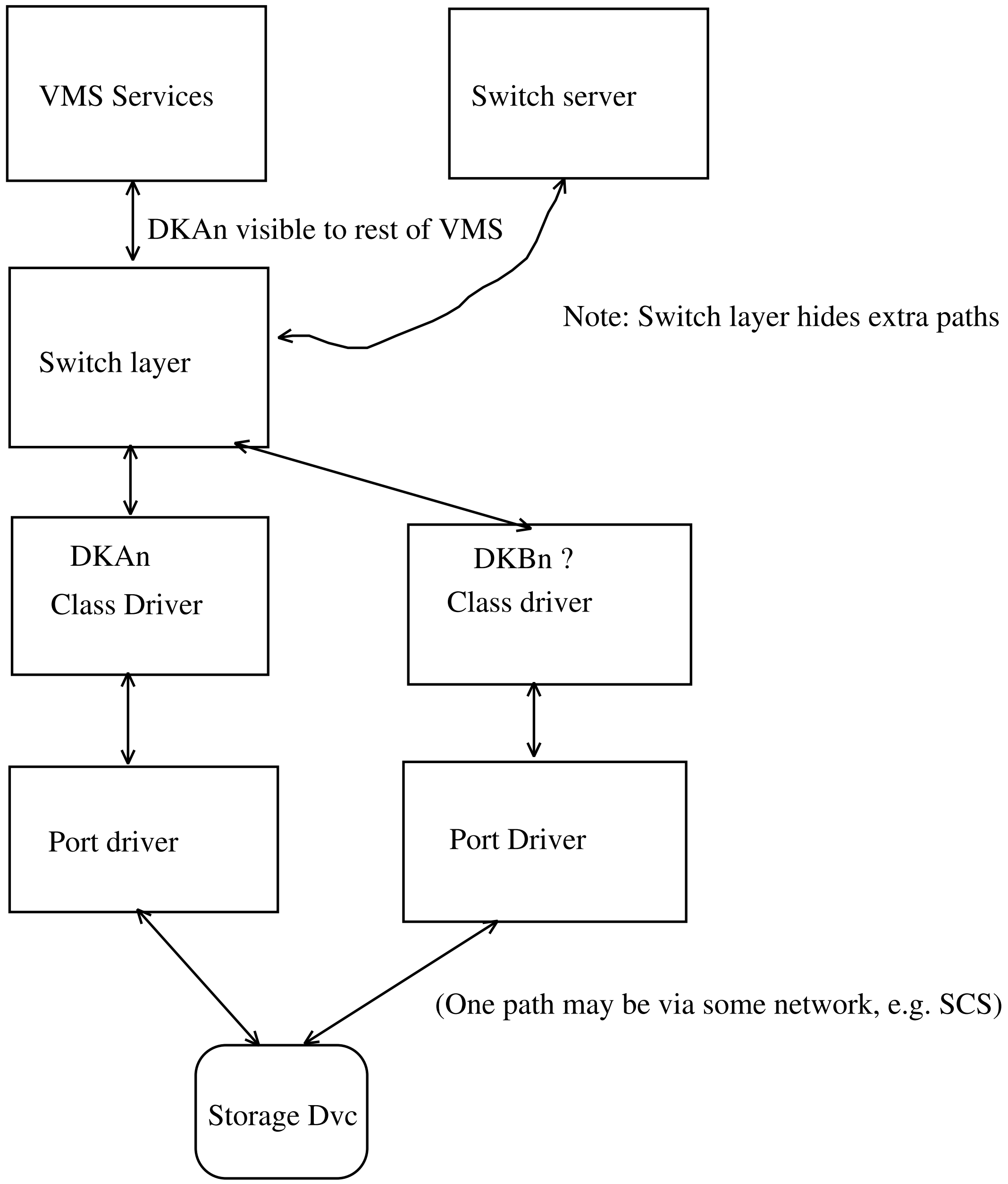

Figure 3. Layering with Switching Subsystem Added above Class Drivers

Appendix A

Some Technical Details

Since some work on a prototype was done as one of the early tasks of this IR, it seems fitting to present some of the thought and detail that went into this prototype, and some considerations that can lead to its evolution toward a final system.

Overview:

The thinking which went into the server prototype was that there will need to be in each machine which has a multipath device a server, whose job it will be to group devices together and control what clusterwide names are used, as well as to allow periodic checking to do timely switching, and a switching intercept driver which will insert itself into the DDTAB entries of the primary path driver chosen and requeue IRPs to the appropriate paths it "knows" about. The server will know some intimate details of the switching driver controls, and the switching driver will offer special controls to allow the server to pass commands to individual paths, providing a "back door" opening beyond the abstraction that the main path is one path only.

The switching driver can also internally watch for mount verify IRPs and use them to signal itself to switch paths, and will keep track of what activity exists at its subordinate drivers.

Since the switching will work via an intercept, the device **name** can be one of the original path names, and no VMS data structures above the intercepted driver need be touched. Thus any risk introduced by adding a switching component is isolated therein. The queueing that is added to the path is exactly what is added for DUdriver, or which is used for striping drivers and some other "performance enhancing" VMS components. Thus the overhead should not be a large worry.

Some Internals:

In order to implement this approach, it is necessary to select a "primary path" from among the paths to the device, so that this path's name can be used clusterwide. (This choice must be the same for all nodes in a cluster.) That done, a switching layer which can exist just ahead of the underlying drivers must control all driver entry points from outside. These are stored in the DDTAB structure. The primary ones are start-io, pending-io, mount-verify-start/finish, altstart-io, i/o cancel, and of course the FDT tables.

The switching functions contemplated here should not need to alter FDT entries, but will need control to intercept the primary path's driver entries. (Other path drivers should be hidden and set to no-assign status to keep them from being used.)

The basic approach considered implies that a new UCB I/O queue will be used instead of the primary driver's queue as the *system* input for I/O to one or more of the paths. It then is the switching layer's job to vector this I/O to underlying paths, to keep track of what is active and what idle, and to ensure that on path transfer, all I/O is revectored to the correct place. In addition it is necessary to provide the ability for specialized software to control path selection.

This approach does require some additional data over that of, in effect, swapping between two whole SCSI port drivers, since one switches each device, not each port. Still, where some multipath points may exist via servers and some via SCSI, it provides a single approach which can handle all cases, regardless of the interconnects.

Maintaining a separate input queue is facilitated by the DDTAB entries. When a new I/O is to be initiated, it is added to the device queue by calling the pending_io entry, or is started directly from the start-io entry. When mount verify is started, the mount verify entry is called to do the actual requeueing of the IRP back to the device, and the mount verify end entry is called to actually start new I/O after mount verify. When altstart is called, the switch layer will be able to use internal information to tell where the IRP supplied needs to be sent. Finally, by altering the IRP$L_PID field of each IRP, the switch layer is able to gain control after each I/O so it can tell when I/O is done, and can monitor I/O for purposes of initiating mount verify conditions when the driver just completing the actual work happens to be one of the paths not known as mounted by VMS. (A testbed experiment has validated the feasibility of this "third party" switching , demonstrating it working in both directions.)

The treatment of mount verify entries is important here. When mount verify start is called, the underlying driver’s routine must be called (if present) to insert the current IRP on the driver’s wait queue, and then the intercept must move it to the multipath input queue. When mount verify ends, the intercept will move the first IRP through its operations and into the path’s driver’s input queue, and then call the mount verify end routine to actually begin I/O again.

It is intended that path switching will occur either **after** a failed path has gone into mount verify processing (so that the I/O system will have been fully idled, and **after** mount verify starts to give SCSI drivers a chance to recover from SCSI RESET before concluding that a path has in fact failed, and so the MSCP/DUdriver code may seek a different served path if possible), or when the switching paths are known to be idle. Idleness in the second case will be determined by counting. The I/O outstanding count is the number of IRPs into the driver less the number of IRPs postprocessed out. When this is zero, the driver is necessarily idle. When paths are switched when idle, the intent is also that packack should be issued on both paths, to force sequential processing at the device and thus guarantee again that no path being abandonned can have any remaining I/O. This is in fact a stronger guarantee than is needed, since during most operations, the paths could perfectly well be shared. Only where there is a need from an upper layer to enforce sequentiality is it necessary that this sequentiality be carried to the lower levels, or when a condition at the device requires only one path to be used for correct recovery (as for example bad block replacement done by a CPU might be).

The approach here is to defer any consideration of performing such synchronization dynamically for a time, but to make path failover work. The approach under consideration does not preclude a much more dynamic style of path sharing, but this is not intended immediately.

Mount Verify and /Foreign Mounts
The mount verify service functions only with a normally mounted device. It is desirable for similar service to be optionally available for foreign device pairs, where a database vendor may be handling the disk itself. This cannot be the default, but is sensible as a general matter.

Fortunately, there is a server available which is able to handle much of the complexity here. If this function is implemented, it is feasible for the switching driver to requeue its I/O to its input, set the mount verify bits in underlying devices, and have the server process perform essentially the same operations that mount verify does. This will allow the same path failover to be offered for foreign mounted devices, should a site so select, that occurs on mounted ones. Since some database vendors operate on non-filestructured disks, this will permit a significant functional gain for their support. Again, driver routines can be called even for secondary drivers, so that underlying code will notice no changes.

Code Assists

Some simple optimizations will be present to detect whether underlying drivers have such DDTAB entries as mount verify start/end, pending_io, and so on, so that IRPs which can be handled more simply by just requeueing to the multipath input queue need not be queued and then moved. However, an underlying driver may be able to speed up operations if it is able to provide some kind of assist so that the path switching system might know when purely local-path processing of a path has completed. This should be possible to add "unobtrusively" in return status from a packack, for example, since we are able in the intercept code to look at any IRP field. There need not even be any user visible changes. The initial scheme of waiting for some number of Mount Verify pack-ack functions (IO$_PACKACK) is usable but crude and might be improved by incorporating some knowledge of the specifics of what was going on at lower levels. (This can mean noting that a bus reset is being handled in SCSI, or noting that a call to MSCP revalidate may be pending, as opposed to complete.) A status bit meaning either "the local processing has failed; try switching if possible" or "more local processing is pending, delay a switch" might be used. The former allows one to expedite switching, while the latter permits one to delay it. The packack count one would use varies accordingly. Alternatively there could be two bits, one per meaning.

Underlying driver support is highly desirable, but it is noteworthy that a multipath switching layer can be constructed with no driver modification at all. This will make retrofit, with at least partial function, easy. Such a solution will not handle full multipath, but could be used in cases such as devices on a shared SCSI bus, permitting failover between MSCP and a direct SCSI path.