VMS Design Specification for Multipath Failover

DIGITAL CONFIDENTIAL

Revision/Update Information: X1.1.0

Glenn C. Everhart

3–February–1997

Abstract

This design specification describes the design for Multipath Failover between multiple paths to storage devices.

Digital Equipment Corporation
Maynard, Massachusetts

1 Introduction

This chapter describes the issues involved in supporting failover where multiple paths exist to OpenVMS devices. These multiple paths can exist due to SCSI controllers which permit a device to be connected to a VMScluster system by more than one path, in general across more than one SCSI bus. Closely related issues exist where devices might be connected by multiple paths to a cluster where some parts of the path between a device and a system are served. To provide more reliable service, OpenVMS needs to have a facility to allow access to these devices to fail over from one path to another in the event one path fails, without manual intervention in the failover.

The chapter will also discuss some elements of the OpenVMS I/O system calling routines which must be modified to support a uniform handling of multiple path failover. While drivers may need to be modified to provide the most complete benfits of this failover support, partial support for path failover can be offered even for unmodified drivers.

This chapter outlines the internal changes to the OpenVMS Alpha I/O support routines, I/O data structures, and device drivers, that will be made to provide this support for multipath failover.

2 Problem Statement Rehash

New hardware is becoming available which provides more than one path to a given device. In the past, OpenVMS has (at least in the SCSI area) used the path to a device in its name, but with these new situations, this can no longer be done. It is essential that a given piece of storage have one name, coherent across the cluster and which does not change between bootstraps, to access one piece of storage, lest file corruption occur on it. To achieve failure recovery, it is also necessary to allow use of alternate paths to access a device when the primary path fails. In arranging for failover, it is infeasible to redesign the entire SCSI I/O stack to retrofit the kind of switching which is deeply embedded in the DSA architecture, but at the same time it is highly desirable to allow failover to use served paths as well as direct ones. Also, when DUdriver was introduced, it was assumed all future disk subsystems would use it and that special purpose code was therefore reasonable to locate within DUdriver itself. It seems valuable to learn from the experience that failover code within any driver will need to be redesigned.

For these reasons it was determined that a software switching layer might provide a long term more maintainable solution. Such a solution can be largely uncoupled from specific drivers' code and from the rest of the OVMS I/O subsystem, making it possible to provide failover functions on future devices cleanly. Such a system must however be designed to minimize added overhead in the I/O path for normal I/O cases.

3 Goals

The goals of the project are the following:

1. Support failover between multiple paths for HSZ controllers on VMS clusters for at least the I/O functions done by MSCP
2. Support other failover cases such as failover between a direct SCSI path and a served path to a disk
3. Provide some failover functionality, even if configuration information must be entered by hand, for generic 3rd party controllers.

4. Not insert any features making it difficult to do failover to non disk devices (though these are not intended to be supported initially)

4 Secondary or Derived Goals

1. Provide support for making I/O intercepts more efficient, and for supporting multiple levels of nested intercepts. (This design uses an intercept, and the cost of making the support work right for the generic case is not greatly different from the cost of special case support.)
2. Support general case disks which use runtime drivers for switched path access, so that served disks (system or other), shadow disks members, etc., can be used (though not all configurations would be supported in the first release)

5 Non-Goals

It is not a goal of this project to support fully dynamic path selection (e.g., where I/O is distributed to each of several paths to balance the load across all of them). This may come later but the initial design contemplates path switching when an "old" path fails. Some controllers permit simultaneous use of multiple paths now, but accounting will need to exist somehow to ensure that all I/O is flushed at cluster state transitions to prevent possible corruption problems.

It is also not a goal (at least initially) to support MOP booting in any special way.

6 Images Affected

The following images are expected to be altered by this code:

1. IO_ROUTINES.EXE (work in SYSQIOREQ and IOSUBNPAG)
2. IRPDEF.SDL (to add a stack at the end and a couple pointers)
3. SYS$DKDRIVER.EXE (support for HSZ, and for start_io access to all IRPs)
4. SYS$SWDRIVER.EXE (new image, the switching driver)
5. SWSWERVER.EXE (new image, the switching server)

7 Overall Software Layout

The path failover subsystem consists of some I/O switching code structured as an intercept driver, some modifications to IO_ROUTINES in the OpenVMS exec to permit control over all IRPs by an intercept, and a server (one of which would in principle reside in each cluster node) which will coordinate global naming choices and ensure consistency in names used from early in the bootstrap process on.

Based on the existing code, the estimated lines of code count for the completed job is roughly 6000-7000 lines of code on each platform. Nearly all of this will be identical in both.

There are two main problems to be dealt with in this project. The first is creating technology able to perform I/O switching between multiple data paths which can allow a single device name (and therefore apparently only one path) to be used at all times while in fact data may traverse any of several paths. (An ancillary part of this problem is allowing or preventing the chosen path to be served, regardless of the local processor's normal actions in this way,

depending on whether it is sensible cluster wide for serving to occur, i.e., whether a local path actually exists and is up to the storage even if that path's name is not the one being used globally.) The second problem is the choice and maintenance of a single name for a device in all parts of a cluster, even where widely varying physical connections and connection geometries may exist.

8 Summary of Impact on Device Drivers

Because it was deemed simpler than to insert switching code into every driver, this design inserts a layer of switching processing "into" existing drivers on the fly. This layer intercepts driver entry points (by creating and using an edited copy of the Driver Dispatch Table) for one path into the device, and uses this path to provide the name and identity for the underlying storage. In effect this adds switching capability to any driver, and also provides some simple methods for communicating with the switching code which would be more difficult to devise by inserting the code into each driver separately. Other paths are hidden from the system, but are known to the switching code, and are used as needed by the switching code. The code is designed so that one path is used at a time, to avoid synchronization problems, and the path is switched when mount verify is found to be active for a time. (The switch does not occur when Mount Verify starts, because in SCSI drivers it is possible to enter Mount Verify when a bus reset occurs and the path is still good. Therefore switching is done after mount verify has been active a short time. This also guarantees that old operations are quiesced before new ones begin.) A hysteresis is to be used in the process so that path switching occurs more slowly for a time after it has already happened. This will prevent looping where controllers do their switching slowly, or when someone is booting. Thus, for a period of perhaps a few minutes after a path switch, switching will be inhibited but after that period will occur shortly after MV starts.

In cases where it can be determined at configuration that a dual path controller is present (SCSI INQUIRY "2P" bit set and WW ID found, or we find an HSZ in dual bus mode), the device driver load code will rename the device so found, so that the path used can be named the same whichever path is used. For HSZ controllers, the unit number will be left alone and the allocation class obtained from the HSZ (and controller letter perhaps likewise). Where this is done, the switching code will be loaded also in anticipation of need for it.

A number of changes will be incorporated into IOSUBNPAG and SYSQIOREQ to permit multiple I/O intercept levels (including ours) to use a "shortcut" process for I/O post processing, so that postprocessing be able to avoid detours through IPL 4 interrupts save for possibly the very last posting of the request. In addition, changes will be made to ensure that all IRPs pass through the standard start_io entry point.

Nevertheless, other entries to disk drivers for pending_io, Mount Verify start and end, I/O cancel, auxiliary entry (used by MSCP to find a new path), and alternate start will be vectored to the correct places in addition to the start_io entry point.

Thus the switching code will act merely as a switch inserted inside of one of the driver units for devices, assuming its identity but in fact maintaining its own common input queue and directing I/O to one of a set of "path" devices at a time. The underlying device drivers and UCBs all exist (but all but one are hidden), and I/O request packets are modified and forwarded to the appropriate underlying drivers, and switching is done when the paths go into mount verify. Because the actual entry points are intercepted (by replacing the DDTAB), any pre-existing UCB pointers will remain valid. Where activity may start before all paths

can be made available, this means that it is not necessary to try to track every pointer to the path, but merely to ensure that naming is common.

Should a driver not be modified and should its path name remain one of the names it would otherwise have, the system can be set up later than driver load time, and switching will still work. Where the paths are served, the naming will already be the same. The renaming above makes it simple to boot from the known device cases, however, by "just working" for boot requests on any path.

It is seen here that switching of unmodified drivers can be handled at least after a fashion provided only that a driver follow the OpenVMS Step 2 interface standards. However, certain modifications are possible in a driver which can facilitate operation.

Drivers may notify the switching code when they detect conditions which should result in changes to path availability. (Note that error conditions which may not be recoverable on a single path may become recoverable if another path exists. Clear separation of all such cases from truly irrecoverable conditions will greatly facilitate path switching.)

These changes can be changes such as the HSZ controllers contemplate where failure of one of a pair of controllers causes notice to be sent to users of the remaining controller that the other controller has failed.

Should a driver not elect to cooperate in these ways, however, the multipath system can still be configured manually to switch between paths when the current path is in mount verify for a period of time (currently 3 IO$_PACKACK packets from mount verify must pass, which takes 3 to 4 seconds, unless a switch has occurred within a longer hysteresis time, perhaps 300 seconds, in which case up to 300 PACKACKs may be awaited, to allow for booting or for slow switching controllers), and so long as the path name is chosen the same clusterwide, failover can be used. However, without some cooperation, the only failover detection available is that a device has gone into mount verify and that it has been in that state for some period of time.

The switching logic must also ensure that some bits of UCB status information are maintained. These include some DEVSTS and STS bits in the UCB. SCSI needs DEVCHAR2 to have a couple bits maintained for use by parts of VMS interested in seeing if the device is on a multipath bus as well. Finally, error and operation count bits need to be summarized in the UCB the system "sees", but separate counts for each path are needed for debugging and configuration purposes. Some of these can be fetched (as MSCP does) on request; others will need to be maintained in the UCB fields during I/O operations. (I/O posting time would be a good place.)

To handle error conditions which may occur in unaltered drivers which can be recovered with a new path, the switching driver will change device errors of types SS$_CTRLERR, SS$_DRVERR, SS$_DEVOFFLINE, SS$_IVSTSFLG, SS$_PATHLOST, SS$_BUS_PHASE_ERROR, or SS$_LINKABORT to be treated as retriable once. In this case, the switching driver will convert these error codes to SS$_MEDOFL (to trigger mount verify) if mount verify is sensible for this disk, and will record the IRP address, and make no further changes in the IRP status when this IRP eventually is retried. If the path is truly failed, this will cause a path switch and may succeed. If not, the IRP can be expected to fail again if there was truly an error condition.

It is not desirable to have insertion of the switching driver to prevent local functions on DK units; it is better that functions like IO$_DIAGNOSE or IO$_AUDIO still work where it makes sense. Since we will know that the "main" path's FDT routines process all IRPs, we can check at start_io time if the FDT routine addresses are the same (letting the I/O by then), or if the I/O functions are "generally permitted" exceptions. Otherwise it will return the I/O with a "derailed" status (an error status yet to be named). This will allow maximal switching of I/O, yet keep strange I/O from being passed to drivers not expecting it. We could in principle switch FDT routines, but that would result in nonstandard functions becoming legal or illegal. A refinement of the intercept might, however, note that a preferred set of FDT routines can exist (e.g. where a direct and served path to a SCSI disk both exist with the served path seen first) which might not be the default's. However, QIOserver will mitigate this problem so switching of FDTs will be deferred. (To switch them, we just need to keep a copy of the FDT table in our intercept UCB and point at it with our replacement DDTAB. Dynamics of this need some checking but for disks this is unlikely to cause much trouble.)

In summary, the driver modifications needed to operate with path switching are small and in some sense optional (in that manual modes of operation remain possible in their absence), though either driver or I/O exec modifications (or both) are needed to permit an intercept to control all IRPs as this one must. Those modifications are essential.

9 Summary of Name Consistency Work

When IOGEN detects a SCSI class driver being loaded, if it is to load a DK unit where INQUIRY data indicates presence of an HSZ, or if is to load a device with the two path bit set in INQUIRY (a SCSI-3 bit, which we expect fibre channel to set) and a worldwide ID present, IOGEN will load switching code including the driver and will initialize it. This code will rename the just-loaded DK unit before anything can point at it. For the initial pass, HSZ units and perhaps devices with worldwide IDs will be given an allocation class obtained from the HSZ itself (and possibly a controller letter also obtained from the HSZ). (The allocation class would be in a numeric range distinct from those generally used.) Should device names be generated from worldwide IDs, then allocation class, unit number, and controller letter are all available to be set also.

10 Functional / Design Specification

10.1 Overview

There are three components to the design: the intercept driver, the server, and the driver interface mods.

The intercept driver component (including some modifications to a few OVMS I/O Exec routines) consists of an intercept driver which will control I/O coming to a "primary path" in OVMS and distribute it along one or more paths whose UCBs will be in existence already when it is connected to them. The UCB address in this system is kept the same, but the routines it dispatches to are varied by replacing the DDT. Thus pre-existing channels to the device are not invalidated, though the control and the device name may be altered. In effect, the intercepted driver is given switching capability by this means. (In some cases, the driver name is altered at load time (e.g. when a multipath HSZ device is seen, or where a device with dual path flagged in its INQUIRY data is seen) but the intercepted name is still the one used.) I/O requests coming to the "primary path" (that is, the one the rest of

OVMS has pointers to) will be routed by this intercept driver to one of its known data paths with a minimum of overhead and processed along the selected path by pre-existing drivers which will then not need to know anything about failover. When the current path returns IRPs with error statuses, the switching driver will arrange for VMS exec routines to place the primary path into mount verify status, or to take it back out again when mount verify completes. The actual path switching will be restricted so that it will take place after mount verify has been active for a short time, or where notification is available that a path has failed and where the path is known to be idle. In addition, the switching driver will provide a means to address each of its subordinate paths individually, a means which will be known to and used by its server.

The server component will exist in each system in a cluster and will provide the global view of the cluster namespace needed to successfully coordinate I/O through multiple paths. Its functions will include:

1. Coordination of enabled path status

   Heuristics for selection of preferred paths and for locating all such paths are feasible only for certain types of controllers. For example, HSZ controllers can only be used one at a time for a given device, so a path failure to one device should disable the use clusterwide of that path. The server will be obtaining information about paths for its monitoring and so is well placed to use this information to prevent failover cycling by blocking paths. (The blockage need not be "forever"; disablng a path when it fails somewhere for as little as several minutes (but longer than any dwells we impose) should stabilize the system in most configurations.)

2. Initiation of serving status for devices.

   In many cases devices will not initially allow themselves to be served until the multipath server becomes active. As a general matter, it is incorrect for a device to let itself be served unless it has a local path to the device being served, and the local path is usable (and actually being used). The multipath server must determine, given its global knowledge of path names, when servers such as QIOserver or MSCP server may validly serve traffic from a node's primary name to other cluster nodes. Once this serving has been initiated, the multipath server must also inform the switching driver of such served paths as appear. Because the switching driver continues to use a pre-existing "principal path" for its activities, the insertion of its switching logic does not require changes to any VMS parts above it which are using the primary path. Where a system has come up and discovers that it is using a path for its system disk which is inadmissible in the cluster, because another name should have been used, the multipath server will need to force the system to hang. The same would be true where other configuration parameters are incompatible across a cluster, though wherever possible the configuration files will be generated so that such errors will never be seen without there having been manual intervention.

   To prevent "served" traffic from migrating aimlessly, the switching driver component will only permit its path to be served when it is using a local connection, resetting served paths if necessary to ensure this.

3. Monitoring paths.

The multipath server will be able to issue periodic commands to devices in each path group to test whether the paths remain valid, and will signal the switching driver to switch to good paths when it finds errors, provided the path currently being used is idle at the time. This will avoid the need to do mount verify because the switching driver will "know" that the underlying path has no outstanding I/O and simply be able to use a new path. While the server must issue an IO$_PACKACK function along its new path, and IO$_AVAILABLE along the old, to ensure the paths are made available, these are less time consuming than full mount verify.

If "underlying path" driver "hints" are available, these are also sent by the switching driver to the multipath server. Such hints can pass information about a path failing, about whether underlying errors indicate hard path failure (so reducing the time needed for awaiting mount verify processing), and about failback situations where a controller has become functional again. The response in case these signals are available is similar to what is done where monitoring of local I/O is done.

One reason for locating this monitoring in a server is that it makes it simple to ensure that monitoring is done only on paths where other signalling is not readily available. Thus monitoring I/O would normally be sent out only on local devices' paths, not over servers. The multipath server will (once initialized) have a database internally to provide this information.

The driver interface modifications are far less code than the other components of the system, but at least the modifications to the OVMS I/O Exec are essential. The driver modifications involve adding some heuristics to drivers (DKdriver first) which will detect some device types (notably HSZ50/HSZ70) and will enable serving on a preferred path only for those devices, otherwise disabling it. They will also use an alternate success status to indicate to the switching driver when an HSZ has sent information indicating that a controller has failed or has become again usable, so that these events may be responded to by other components.

More important are the I/O exec modifications. Changes to DKdriver made for Alpha in OVMS 6.2 caused it to extract IRPs from its input queue without passing any standard OVMS driver entry point. This, coupled with the fact that OVMS does not always call any driver entry point when an IRP is inserted on a driver queue, makes it impossible for an I/O intercept to control all I/O to a device with any reasonable overhead. Prior to that point, such interception has been possible and reliable, and is used by a number of third party products (which have been unable to explain problems they have seen because DK listings were censored in V6.2). The changes were made to DKdriver due to stack overrun problems, which hit when DKdriver was given large amounts of traffic, and no good alternative was thought of.

There is also a problem in that too much I/O in ANY Alpha VMS driver where there is no delay causes kernel stack overflow, due to recursion via the REQCOM function. The DKdriver issue is of similar origin, but in different position.

REQCOM and DKdriver will be fixed so that every IRP passes through the start_io entry and does not overflow stack. This can be done (as proof of concept) by testing SP at each REQCOM (or each IRP pulled off its queue by DKdriver) and, if it is within, say, a pagelet of its current page bottom, PROBEing the next page down. If this reveals the stack is in its last pagelet, the driver or REQCOM could simply fork and have the fork routine call the start_io entry, having cleaned off the stack. This solution works in tests and will remove an Alpha driver problem first reported in VMS 1.5 which can crash the system unpredictably.

In this way, the switching code can gain control and redispatch the IRP as needed, Note that even if the same routine is called from either place, the EXE_STD$QIODRVPKT call junks the return status, and normal initiate calls do not define a return status, specifying R0 may be destroyed. Because this solution guarantees control at either start_io or at the callout for every IRP, the UCB input queue can be used as is, and no separate one need be maintained. Also, an IRP will be seen at only one point in the normal path. (It will still be necessary to dispatch various error paths including altstart, cancel, Mount Verify start/end, auxiliary I/O (DUdriver uses it), and pending_io, but these will mainly just redispatch the IRP to the appropriate driver. After underlying MV start is called, it will be necessary to perform IRP marking as noted earlier to record which FDT routines were used for IRPs on the fly at that moment, and to restore state so it can be re-saved properly when a new path may be used. This is only a short bit of code however.

In order to speed up the kind of context save and restore, a new "argument stack" will be added to the end of the IRP definition also, with a pointer and limit, to hold small amounts (~5 longwords for this intercept) of data between sending an IRP along and seeing it at postprocessing. Where an IRP lacks this, or the stack gets used up, (as determined by its size field), the intercept will have to allocate pool and free it, but in most cases this will allow pool not to have to be allocated. A stack is used because it is no more complex than using other storage, and because it can facilitate other intercepts in handling "streaming" the way Unix does. Initially this stack can just be a constant. At some time in the future it might be desirable to make it a sysgen parameter. The work of handling IRPs that might not conform is unfortunately necessary for a point VMS release, even though I expect they will be rare.

We will next describe the designs in greater detail of the parts of the switching facility.

10.2 Switching Driver

The switching driver is an OVMS device driver in outline, but serves primarily as a place to store kernel code which pre and post processes I/O requests for other drivers. It is accessed directly via some functions handled within its FDT processing, which intercept another driver or disconnect the intercept, but in fact its start-io routine is a dummy which does nothing, and considerable data is held in its UCB. In fact, the intercept scheme copies the intercepted driver’s DDTAB structure into its own UCB and edits it, retaining pointers to the prior DDTAB, so that the intercept can be removed or others added in a possible chain of intercepts without these damaging other intercepts already present. A "magic number" is used to flag that the intercept scheme is in use, and the intercept area of the intercept UCB also contains a pointer to the intercept driver’s DPTAB, so that when intercept code gains control, it is possible for the code to ensure it has located its own intercept UCB. Since the UCB$L_DDT offset of the intercepted driver points to an offset within the intercept UCB, negative offsets from this can find intercept UCB sites, and the added fields allow the intercepted code to be sure it has found its own data. (In the normal case this means that the intercepted code locates the intercept UCB with a compare and a move. In production code, a version of the search routine which takes out the fork lock and one that does not will be needed, so that this overhead can be avoided where an intercepted call will already hold forklock.)

The switching driver uses the IO$_FORMAT call, with any function modifiers, to connect or disconnect the intercept. One argument is supplied, the address of a control buffer. The first longword of this buffer must be 1 to intercept a disk, 2 to remove the intercept, 3 to add a new disk as a path to the intercept, and otherwise is illegal. In case 3, the second longword contains the index of the disk to add (as a sanity check; it must be the next path) in the low word and an enabled/disabled flag in the high word to set whether the path is initially marked usable or not. The remainder of the buffer contains the name of the disk being intercepted as a character array. (The second $QIO parameter giving the size of the buffer is used to construct a descriptor.) The device so indicated is intercepted. (The routine which calls this function checks that it is indeed a disk class device.) (When undoing an intercept, the switch driver also checks that it is undoing the correct intercept and makes no changes unless this is so.) The interception operation also stores (for quick reference) the addresses of the DDTAB entries it intercepts in the intercept UCB, so that the original routines can be quickly located. Entries so stored are:

1. ddt$ps_mntver_2
2. ddt$ps_altstart_2
3. ddt$ps_pending_io
4. ddt$l_aux_routine
5. ddt$ps_cancel_2

Some additional functions will handle the passing of setup information such as server identity information into the switching code. This will avoid the need for the server to know anything about the switching driver's UCB extension format.

Once the new DDTAB is constructed, the actual interception is accomplished by storing the new DDTAB address in the intercepted device's UCB$L_DDT address. (The DDB$L_DDT entry is not in fact used in I/O dispatch and is left alone.) Normal multipath setup will thus call this entry once for the principal path, then add several alternate paths. Sanity checks for ensuring that intercepted devices are of acceptable type (initially disk) and not already intercepted will be moved to the switching driver also, so that kernel code and complexity in the server can be reduced.

IRPs which might be "in flight" when this DDTAB replacement was made would not be counted up in the intercept outstanding I/O, nor would they be counted down at I/O posting time, so they would have no effect on the counter. Before a path could be switched, however, a small time delay would need to occur if the switching were to be forced by the server. This time delay would allow these operations to be completed.

The switching driver also needs to learn whether it can safely set the IRP's FASTIO and FINIPL8 bits, to get I/O completion to operate without needing to incur an extra IPL 4 interrupt and another fork. When we intercept a driver, its I/O posting will take place, if we use the shortcut, at fork IPL, as will any intercepts inserted after ours. It is therefore desirable to have our interception control maintain a flag that can be set by well-behaved subsequent intercepts to indicate they cannot handle IPL 8 postprocessing. This flag may be safely set where a driver indicates (in its UCB$L_STS) that fast io is permitted. If we find we can set it (and it will be OK for DK or DU to do so), the IRPs can be posted with the FASTIO and the FINIPL8 bits set to allow our code to gain control at fork IPL. The switching code's postprocessing area will of course replace the original status and post the I/O again, but will have avoided an extra detour in doing so. It will be the responsibility of

any other intercept to either handle the call to REQCOM at higher IPL, or to clear the flag. We will also not use this trick for known cases such as the shadow driver where REQCOM must be at IPL 4.

Another function implemented solely within the switching driver is the "switch path" one, which uses the IO$_RETCTR function for the purpose. This causes the current intercepted device to switch to the "next" available path, provided that the switching driver outstanding I/O count is zero. (This count is incremented when I/O is added to the underlying path's I/O queue or when I/O is started, decremented when I/O is removed from the underlying queue (as at mount verify), and decremented when I/O completes. Thus if it is zero one can be sure the path is in fact idle.)

The unit init routine of the switching driver computes several addresses used in processing. Notably, since a routine called at I/O post time receives only the IRP address, which has had IRP$L_UCB pointed at the underlying driver (which may or may not be the principal path one), the system maintains a list of entries, one per intercept unit. These load a register with a fixed offset and call a common postprocessing routine which uses this offset to load the intercept UCB address from another internal table. This table is filled in for an intercept driver unit at unit init, and the address is computed which must be placed into IRP$L_PID in order to gain control after underlying layers have finished with the IRP. This permits a fast test (i.e., is IRP$L_PID the same as the value this intercept driver will use?) to determine whether an IRP has been modified for a path or not.

The switching driver's START_IO routine is needed to avoid complexity in the server. When a new path is to be switched to by the server, it is vital to send a pack-ack down that path to ensure any old I/O is flushed out before new I/O is sent. In general this path will still be marked with the UCB BSY bit, so a simple $QIO will not work. However, the switch driver's start_io will contain code to take an ordinarly packack and direct it to an underlying driver's start_io by marking the packack as though from mount verify and calling IOC$INITIATE to send it to the soon-to-be-active driver if that driver is still marked busy. Since the switching driver can tag the IRP itself, it can ensure it is passed correctly. This scheme means that the server need not know the format of an IRP to generate it on its own, and avoids the need to design new system services to communicate with the switching code. Thus even though the bulk of the switching driver could as well be a generic execlet, its form as a pseudo driver is of value.

The remainder of the switching driver consists of routines which gain control when the principal path driver is called at various of its entry points, and a couple of utility routines. They eventually will call the original entries but must ensure that the IRPs are set up properly to use the currently selected path. The processing in these routines will now be described.

1. Steal_Pending_IO

   Pending_IO is called to queue an IRP onto the driver queue for some fast-path functions.

   This entry notes which path is current and calls the corresponding entry in the underlying driver's DDTAB. If an alternate path is called and the IRP is queued, it will be requeued to the "primary" path subsequently so that it will appear only in the primary queue. (R0 returns as -1 if the IRP is not queued.)

2. Steal_Aux_Routine

The aux_routine entry in DUdriver is used to tell an MSCP server that it must find another path and is called just prior to the start of mount verify. This routine in the intercept driver merely calls the underlying driver's routine, if any exists, and returns, merely dispatching the call to the correct path's driver.

3. Steal_Cancel

   Because our intercept use the IRP$L_PID field of each IRP switched, the exec I/O cancel routine will generally not find matching PIDs to cancel. However, the Steal_Cancel routine checks to see if the cancel is coming from the MSCP server and if so switches the value in IRP$L_PID to an alternate one maintained within the switching driver, and sets up R4 to point at a fake PCB which has this address in its PCB$L_PID offset. This will allow underlying cancel routines (if coded right; there is a long standing dkdriver bug in this area!) to recognize IRPs which "came" from the MSCP server since the MSCP server uses the same trick. The routine then calls the underlying driver's cancel_io entry for the path in use. Cancel_io posts the IRPs that are cancelled, so the active-IRP count maintained in the switching driver is correctly maintained by the count-down at I/O posting time.

4. Steal_Mount_Ver

   The mount verify start/stop entry is called when mount verify is starting and the current IRP passed to it should be put on a pending queue, and when mount verify is ending and I/O needs to be restarted from the pending queue.

   When mount verify is starting, this routine calls the underlying path's mount verify entry, then moves any IRPs on the underlying input queue into the "prime" path queue (preserving order), de-specializing them along the way so they are ready to be processed for either path.

   When mount verify is ending, we call the underlying end-MV entry, then call the standard end-MV entry to actually restart I/O if we are on an alternate path. (We do not want to call the main path end-MV if not on the main path since it may be important to keep some driver busy flags set.)

5. Steal_Altstart

   Since calls to the alternate start-io entry don't change paths, the Steal_Altstart entry simple processes the IRP to work correctly along the current path and calls the underlying path's alt-start-io entry. Since the rest of VMS knows only about the principal path, the routine takes out the fork lock for a different path if one is to be used and restores it on completion. (Note that I/O to the alternate start entry IS switched, but mount verify generates calls to normal start_io and so the calls to altstart_io do not cause switches.)

6. Undomods

   This entry is a utility entry which is passed the intercept UCB address and the IRP address. It merely undoes changes made to the fields IRPL_PID, IRPL_MEDIA, IRP$L_STS, and IRP$L_UCB and counts down the number of IRPs pending on a device. For speed, the test code uses some longwords stored in IRP$Q_QIO_P1 through IRP$Q_QIO_P4 which are normally not needed by the time these driver entries are called. If the IRP has a stack, this will be used for arguments, but if not, a structure will be allocated in pool, with a listhead in the intercept UCB, to hold context. This routine will free such.

7. Stealstart

This entry is the location where start_io calls to the principal path come. Ultimately it should be called from a driver callout to get the best performance, once that can be agreed to. However it will be described here as it has been with no callout.

This routine first checks for special function modifier combinations, so that "all function modifiers" IRPs are sent to the principal path with the function modifiers stripped, and other combinations of function modifiers (with most set; the low few bits select path) do the same but direct the IRP to other paths. It is expected these entries will be used where the path is otherwise idle or for calls not synchronized with the UCB busy bit. These paths provide a (narrow) path for calling underlying drivers and going around the multipath abstraction. It will be used to send packack functions to a new path when path switching must be done by the server component. (Note: another method needs to be supplied to mimic direct calls to INITIATE without UCB BUSY synchronization. The intercept driver's start_io will be used to facilitate this.)

The routine then checks to see if this IRP is a mount verify PACKACK function. If so, it counts the number of these seen in succession. When this exceeds a threshold count (currently 3), path switching is performed. This switching sets the preferred path to the next enabled path (the enabled flags are within the switching driver and are set as the result of the server's sending messages to the switching driver that notify it of a new path) and resets the count of MV Packacks needed to switch again. The routine also formulates a message to send to the server notifying it of this and sends it to a preset mailbox which the server must establish and register with the switching driver for this purpose. The CALL_WRTMAILBOX macro is used to send the message.

If the IRP is not a mount verify packack, the count is cleared to indicate that something else has been seen.

The IRP is then modified to point to the correct underlying path, and the outstanding IO count is bumped. The fields IRPL_PID, IRPL_MEDIA, IRP$L_STS, and IRP$L_UCB are saved on the IRP argument stack (or a pool structure) and altered in this operation. Provided the underlying driver(s) can support fast finish I/O with the FINIPL8 bit set (this bit allows system routines to be called directly from fork IPL) and that the I/O is not from SHDRIVER nor from mount verify, the fields IRP$M_FINIPL8 and IRP$M_FAST_FINISH are set. Also, the IRP$M_SWAPIO field is set, so that if an underlying path becomes unready, the principal path will be allowed to enter mount verify immediately. This will prevent large numbers of IRPs from potentially being placed on the driver input queue from IPL 8 and overflowing the kernel stack. Since we restore the original STS field, the last I/O posting will see the correct status and this will not interfere with normal split I/O processing. Once this has been done, the routine queues the IRP to the correct path (actually, just calls the principal path if that was the path in use) if the IRP was not from mount verify, or uses CALL_INITIATE to call the underlying path driver (with the correct forklock held) if the IRP came from MV and thus is not synchronized with UCB Busy. In all cases, the IRP$L_PID field will allow the switching driver to regain control at the FIXSPLIT entry when I/O posting is done on the IRP.

8. FixSplit

The FixSplit routine is called when an IRP intercepted by the multipath switching driver is posted. It is entered with R5 containing the IRP address and R4 containing an offset into the UCB address table maintained within the switching driver, and uses this information to locate the intercept driver UCB, and incidentally to clear flags that the IRP has been processed, so that split I/O can redo the work as it must. The code then

loads the old IRP status, and if the new status does not indicate it must already be at fork, it first acquires the intercept driver's device lock (at IPL 8), restores IRP$L_STS, and acquires the FORK spinlock. If the IRP is not a MV IRP, and the device is not in Mount Verify, the routine then ensures I/O is kept going by draining the multipath queue into the current path queue. It restores the rest of the input IRP's fields, and then tests the returned I/O status.

Where the I/O status has the 16384 bit set (dkdriver uses no return status codes with this; nor do others inspected), a message is sent to the server's input mailbox telling what device returned a failover or failback function. If the I/O status indicates some error occurred, and if the principal path is a mounted disk subject to mount verification, the switching driver then calls the EXE$MOUNT_VER routine in the exec to test whether mount verify should begin. Note that at this point the IRP appears to have come from from the principal path, not any secondary one being used, though the switching driver must reset busy on the main path when mount verify is ended later with a secondary path being used. Because the IRP at this point appears to come from the principal path, which OVMS knows of and has mounted, that path enters mount verification, though the switching driver redirects IRPs to the appropriate underlying driver, with the effect that the actual mount verification work uses the path the switching driver needs it to.

If the IRP had success status or if the MV tests done by EXE$MOUNT_VER indicate that the IRP should be posted, the IRP count of active IRPs on the device is decremented and the IRP posted (with its fields repaired so the "final" posting can take place. Then the locks taken out are returned and the routine exits.

11 Base VMS I/O code

There are at least three modifications needed for some base I/O modules.

1. SYSQIOREQ and IOSUBNPAG modifications

   SYSQIOREQ and IOSUBNPAG need to be altered so REQCOM uses the "jump back" strategy mentioned above, so that stack will not overflow.

2. IOSUBNPAG Postprocessing Shortcut

   The processing of the FINIPL8 bit in IOSUBNPAG needs to cause I/O to be resumed where the caller issued REQCOM type macros, and not when they issued CALL_POST type macros, rather than relying upon the system routine called to do this. The problem lies in nested intercepts, where lower intercepts will be called via this code, but may not realize it. An upper level intercept cannot economically trace all intercepts it intercepted and resume their I/O when appropriate. The information about the call type should not be discarded. A simple change to the FAST_FINISH macro is all that is needed to enable this.

3. IOSUBNPAG search routines

   To allow full hiding of extra paths, the search routines and IODB scan routines in IOSUBNPAG need to honor a new device hiding bit which can be set in a UCB to flag that it should not appear in displays. The NO_ASSIGN bit already exists to prevent use of a UCB, but displays showing the same device several times, or showing redundant different names for a device should be avoided.

4. DKDRIVER changes

DKDRIVER at least must be modified to use the callout strategy mentioned above so that all its IRPs will be available at a callout at the start_io entry. This allows the stealstart routine to see every IRP.

In addition to these changes, which are the minimum required, it is highly desirable that IRPs acquire an argument stack and that a driver callout be defined which could be used for DKdriver to allow an intercept access to all IRPs. This callout could be inserted in DKdriver, DUdriver, and perhaps DRdriver readily, and it might prove advantageous to build a switching intercept which would only work with drivers so equipped. (Certainly the intercept could be simpler and cleaner.) It should also be mentioned that a modification to permit mount verify to handle /FOREIGN mounted disks has valuable properties for database customers.

12 Server Process

The server process receives data from a configuration file, from cluster locking communications with other servers in other cluster nodes, from user commands (in a manual command mode), and from the switching driver sending it messages. It (and generally only it) sends special I/O requests to path members at path switch time, and it controls the path available status at each node for each device. The multipath server process is the one place in each cluster node which has a global view of the paths available to different I/O devices, for cases where more than one path exists. The multipath server must also coordinate enabling servers like the MSCP server to become active on the appropriate device paths.

Processing to be done in the server is in several major blocks.

The first activity is to read any command line controls, in case this is a manual invocation which will specify pairings and actions by hand. In such a case, the utility will operate only on the selected devices with actions such as setup to associate a particular unit of the switching driver with one or more paths to some other device. Qualifiers will control whether the path is enabled or not, whether to connect or disconnect the switching driver, whether to enable serving or not, and whether or not to stay active to process messages.

For initial support of HSZ hardware, we will modify the driver loading code to detect HSZxx controllers in multipath mode (their SCSI INQUIRY data permits this) and will pick an allocation class and port letter from the HSZ itself to do this, using it "as if" it were a port allocation class. The unit number will be left alone, but since controller letters will be forced to "A" this means that each path will be named the same. Ultimately this allocation class will be set from the controller console. However, until this can be done, the following method will be used:

1. When an HSZ is detected, its controller serial numbers will be arranged in ascending numerical order. A lock will be queued in EX mode on this name to register that this CPU has this pair of controllers.
2. A hash, based on two or three SYSGEN parameters, will be computed on this value and used as the allocation class.
3. To detect collisions, another EX lock will be queued on this hash value.
4. Now $GETLKI can examine the interference list for the two locks. If we find any processor on the interference list for the lock on the hash, which is not also on the interference list for the controller numbers, we know we have had a collision. We print a message and halt. (This is just about what happens with port allocation class.) The

customer can then either change the SYSGEN parameters for the hash to avoid collisions or connect the offending HSZ in single attach mode.

This is statistically likely to produce an adequate solution most of the time, though not a 100% guarantee. It could be wise to supply a tool that could be used to determine collisions before connecting a new HSZ to aid customers. However, once new HSZ firmware appears, a customer can simply assign a unique allocation class for himself.

A similar treatment can be made for WW IDs as an initial pass, by the way. Where a WW ID is seen, the hash may give not only an allocation class but a unit number, to allow a few more bits variability. The test is almost identical to what is currently in place for port allocation classes and requires no full fledged communications. Ultimately such will be needed however.

The server on each node will need to recognize any alternate paths that appear for devices, and connect them to the switching code.

The server process must communicate with other servers for "special cases" in managing enabled paths. In this way, for example, when a node's SCSI controller dies which is used to connect to one of the controllers on an HSZ, that path should be disabled for all systems connected to it. This must be done for HSZ since only one controller at a time is permitted to be accessing a particular device. If another node may use the path that the first node cannot, and the first node starts using another path, the paths will bounce in and out of mount verify indefinitely. If the first path is declared unusable everywhere, however, the second failure will cause the node to seek a served path instead, keeping access stable. In order to detect a hard failure, keeping timestamps for failover will be needed, and some TBD heuristic will need to exist to set a path as "permanently" down, or to reset it under DCL control.

The server process may also need to integrate functions of the QIOserver, but that integration is not part of this document at present. The QIOserver also needs a process with a global view of configurations.

The server process, finally, will be able to poll devices by issuing I/O to them occasionally to discover if they are active. Since it will have data about what controllers exist, it will be able to issue I/O to only one device per controller to limit the amount of added I/O traffic it injects to ensure paths are active. The polling will allow generation and receipt of status indications about "other paths" from HSZs and similar devices, thus limiting the time to discover broken paths.

13 DKdriver (and other driver) support

Performance of path switching can be enhanced by support in DKdriver for known SCSI controllers. Also, the job of controlling when a device becomes available for serving is simpler if the driver can cooperate. Apart from issues of driver intercept, the key areas are recognition of signals from some controllers that paths need to change somewhere in the system, hints about when paths may need to change (as opposed to simply passing through the SS$_MEDOFL status, which for SCSI may simply mean a bus reset occurred), and checking for preferred path in setting a device serveable.

When HSZs detect that a controller has failed, they will return auxiliary unit attention type status to the host(s) for I/O to devices on the still good controller, indicating failover of some other devices. The idea here is that DKdriver should return an alternate success status (set the 16384 bit) to the caller when this happens, so that the switching driver can send a message to its server process, strip the extra bits, and complete the I/O. The server process then will switch any devices using the now-broken path which are idle, making mount verify unnecessary in that case. Something similar should happen on failback. This added processing is short and will be added to the unit attention code within DKdriver (which currently generates success status). This will minimize the need for mount verify and avoid delays in path availability in some cases.

Some SCSI port drivers may be able to distinguish loss of SCSI bus integrity from simple SCSI reset. Ultimately when this happens, it would be desirable to have the SS$_MEDOFL status that returns be modified, perhaps by using part of the 4th word of the IOSB, so that the switching code can tell that it need not wait for 3 mount verify PACKACK commands before switching in this case. Alternatively where the port knows that RESET has occurred, a second signal to let the switching layer see this could be used to prevent path switching where there may be no reason to do it as soon. These changes will not however be made initially due to their complexity and the possibility that they may impair failover in some cases; these abilities of port control chips are not well explored in the OpenVMS group at present.

14 Interactions with Other Parts of OpenVMS

The interactions of this subsystem with the rest of OpenVMS are expected to be restricted to device drivers and similar low level code. The switching driver is intended to be basically invisible to all layers above it. However, certain minor changes may be noted.

1. If any multipath controllers have only one path connected and the switching software is not used, the alternate success statuses mentioned above may be seen instead of SS$_ NORMAL.
2. Underlying intercepts could make it impossible to use the post processing shortcut, which would introduce a detour through IPL 4 at every I/O post. This might be detectable.
3. It must be remembered that even from a client perspective, switching may be necessary, in situations where more than one server type exists, simply to provide stability of naming clusterwide.

As can be seen, these effects are primarily startup effects. Runtime speed of I/O is expected to be essentially unaffected. These effects moreover are balanced by the fact that path failover will now work across multiple paths to devices, and will require essentially no further changes to other drivers to work. Driver processing and I/O processing, apart from the small modifications mentioned above, are unaffected by the switching subsystem, which can therefore be debugged largely independently.

15 Performance

The path switching capability must not come at the expense of I/O performance. Where path switching is not used, performance must not be altered at all, and in multipath cases the goal is to reduce performance by 0.2% or less.

The first subgoal is easily achieved. The intercept used is a per disk intercept which cannot alter anything where not used. A small amount of added processing at driver unit init is left out of consideration here but will be minor.

The second subgoal will need to be measured. While no process context detours for IRPs exist, there are some INSQUE/REMQUE instructions which involve PALcode calls, and forklock is acquired and released by INSIOQC which is used to send a modified IRP to any secondary path for normal I/O. The amount of extra queueing could be less if the underlying driver had a callout; this would eliminate most of it in the mainline, but the overhead of an added INSIOQC is unavoidable along secondary paths. This call may or may not use INSQUE, depending on how busy the underlying driver is, however, and in particular with DKdriver or DUdriver the path will often be unbusy. Thus it is expected that the second subgoal can be met or exceeded, provided that no underlying intercept code which prevents the post processing shortcut from working exists.

It is expected that performance will be measured by doing some I/O bound operation and measuring CPU and real time with and without the intercept active, and along the primary and along at least one secondary path to the same device. Existing tests can be used for this.

16 Technical Risk

The chief technical risk item in this project is that switching paths may prove to have some unforeseen problem which is not understood and makes a redesign necessary. This risk has been largely retired by the construction and testing of a two path intercept able to switch from one path to the other at mount verify.

Other risk items have to do with whether the IRP structure and underlying driver structure can be modified as recommended here. If they cannot, intercepts will need to allocate and free pool for each I/O, which could easily bring performance below desired values. This is in some sense more a political than a technical risk, since the proposed intercepts can be made such that old drivers will not require any changes, and failover need not be supported on all possible drivers.

It is also necessary that the changes to REQCOM and DKDRIVER to ensure that all IRPs are seen at start_io entries (or altstart_io) be agreed upon and made. Testing can proceed without this, but production will require it.

The control server implements the manual controls noted, but the locking protocols described above, and related parts of the server component of multipath switching, have not been implemented in the prototype investigation phase. If there are issues in the early boot environment which cause difficulties, these will have to be dealt with. However, this is basically user - mode application level code and no major hangups are anticipated.

17 Schedule

It is expected that coding and initial testing of a working "production" version of this code should take about a month and a half from completion of design reviews, given that there exists a "lab" version already. A production version must be tested on uniprocessors, multiprocessors, and clusters, and needs to be tried with HSZs, served paths, and various options of connection. The locking logic needed for handling configuration files will be written somewhat generically, since it appears likely that the same sort of logic will be needed for extended device naming support. Further incremental work on the server part of this project is to be expected as QIOserver functions are integrated into it, and perhaps more such will be needed should device naming support be moved in as well.

18 Revision History

Change bars are used to identify all substantial changes and additions between versions of this document.

1. **Revision X0.1, January 2, 1997**
    1. Everhart - Initial draft 02-Jan-1997
    2. Everhart - Revision 1 28-January 1997