SNIA®

Storage Networking Industry Association

NVM PM Remote Access for High Availability

Technical White Paper

May 2019

ABSTRACT: *This paper explores the requirements and desirable design characteristics that High Availability extensions to the NVM.PM.FILE mode of the SNIA NVM Programming Model might impose on high speed networking.*

USAGE

DISCLAIMER

The information contained in this publication is subject to change without notice. The SNIA makes no warranty of any kind with regard to this document, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The SNIA shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this document.

Suggestions for revisions should be directed to http://www.snia.org/feedback/.

This page intentionally left blank

Contents

1 Purpose

The purpose of this document is to establish the context and desirable design characteristics for the use of high speed networks as a transport for remote access to persistent memory (PM) in high availability implementations of the SNIA NVM Programming model. The resulting set of requirements is summarized at the end of the document.

2 Scope

This non-normative document pertains specifically to the NVM.PM.FILE mode of the SNIA NVM Programming Model. Some implementations of the programming model may provide high availability (HA) by communicating with remote persistent memory. The term “Remote” refers to persistent memory that is not attached to the same CPU complex as an application that is using the NVM Programming Model. The term “application” refers to the user space consumer of PM as described in section 4 of the NVM Programming Model. A CPU complex comprises the CPU, memory and support chips for a single or multi-socket server. There are many ways to implement remote PM communication including Remote Direct Memory Access (RDMA) over ethernet or InfiniBand networks and other operations supported by OpenCAPI and Gen-Z.

With this in mind the following terms are used pervasively in this document in order to avoid any unnecessary implication of implementation specifics.

- RPM – Remote Persistent Memory
- RPMA – Remote Persistent Memory Access

The intent of this terminology is to enable a range of RPM and RPMA implementations while describing characteristics of RPMA that could reduce the overhead of correct HA operation.

The term “High Availability” (HA) refers to the use of redundancy to achieve fault tolerance that limits or eliminates downtime. Redundancy includes duplication of hardware and data in patterns that avoid single (or multiple) points of failure. This in turn involves reasoning about independent fault domains and a wide range of erasure coding techniques. These techniques are generally well known and widely implemented. They form an important backdrop for this document but they are not broadly re-iterated herein.

This document neither addresses nor precludes shared data beyond the extent necessary to enable failover of data access for high availability. This can be formally described as a type of “Release Consistency” as defined by Gharachorloo et al. in “Memory consistency and event ordering in scalable shared-memory multiprocessors,” ISCA, 1990, pp. 15–26. Release consistency assures that memory state is made globally consistent at certain release points. In this case, failover comprises the release point. The failing unit is forced to cease operation and the state of one or more durable replicas is used to establish global consistency by means of post processing such as transaction aborts, completion of transaction commits or consistency checking processes (e.g., fsck).

This document describes requirements and desirable design characteristics that are visible to an application or within a data-path such that they affect performance or real

time data recoverability. Management functionality is not addressed in this paper. For example, hardware discovery, system configuration, monitoring and reliability, availability and serviceability (RAS) capabilities such as troubleshooting and repair are considered to be management capabilities.

Security considerations such as privacy and integrity are explored in the SNIA "Persistent Memory Hardware Threat Model" white paper. While the treat model exposes some new considerations related to PM, no new network requirements were identified. This document does not cover those topics other than to suggest support for integrity checking with RPM.

3 Memory Access Hardware Taxonomy

There are a number of ways to describe hardware access paths to memory. The memory connectivity taxonomy in this section is intended to add clarity to various remote memory access use cases, including those related to high availability.

High availability use cases described in this paper align with the networked persistent memory access model described in sections 3.3 and 3.4. This is because a loosely coupled server environment using is the most common way to assure the fault independence needed for high availability.

3.1 Persistent Memory (PM) latency landscape

Latency is a key consideration in choosing a connectivity method for memory or storage. Latency refers to the time it takes to complete an access such as a read, write, load or store. Figure 1 illustrates storage latencies that span 6 orders of magnitude between hard disks and memory. The span of each bar is intended to represent typical range of latencies for example technologies.

There are two very important latency thresholds that change how applications see storage or memory represented by the background color bands in this figure. These thresholds are used by system designers when implementing access to stored data, to determine whether the access is to be synchronous, polled or asynchronous. In today's large non-uniform memory access (NUMA) systems, latencies of up to 200 nS are generally considered to be acceptable. NUMA systems must be very responsive because CPU instruction processing on a core or thread is suspended during the memory access. Latencies of more than 200 nS in a memory system quickly pile up, resulting in wasted CPU time.

On the other hand, when an application does IO for a storage access that is expected to take more than 2-3 uS, it will usually choose to block a thread or process. The CPU will execute a context switch and make progress on another thread or process until it is notified that the access is complete. For latencies between 200 nS and 2 uS it may be preferable for the CPU to poll for IO completion as this consumes one thread or core but does not slow down the rest of the CPU.

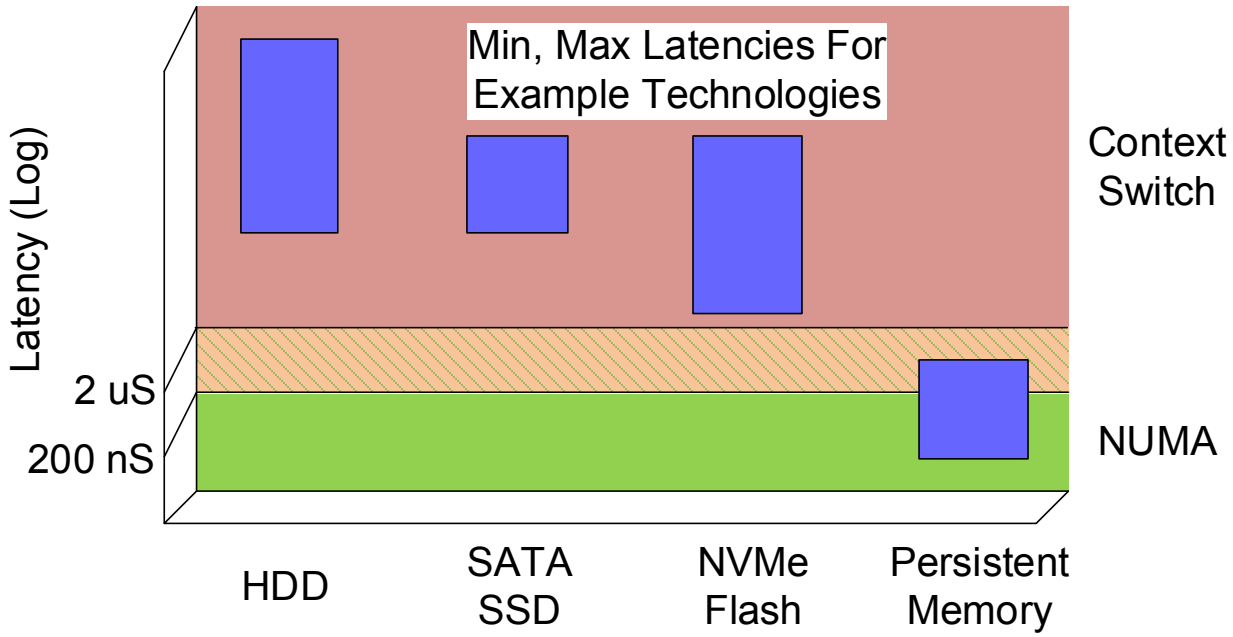

Figure 1 – Storage Latency Ranges Impact Software

Local or disaggregated persistent memory (see sections 3.2 and 3.3) can fall into the NUMA range of Figure 1. Networked persistent memory (sections 3.4) does not. Since the high availability use cases described in this document involve networked persistent memory, they can quickly slow applications down to IO speeds. This tends to reverse the performance gains made in the transition to persistent memory unless remote direct memory access (RDMA) is optimized for high availability persistent memory use cases.

3.2 Local Persistent Memory

Local persistent memory is generally in the same server as the processors accessing it. This is illustrated in Figure 2 in a dual socket system where DIMMs and NVDIMMs are connected to CPU's which are in turn connected using a cache coherent inter-socket interconnect that is specific to the processor architecture. Local memory is accessed using the NVM Programming Model without any remote access considerations. For the purpose of this taxonomy, all of the memory in this illustration is local because it is part of a single server node. Although the illustration assumes that memory controllers are integrated into CPU's, memory attached to controllers outside of CPU's but within the server is still considered local.

A single server does not avoid single points of failure and it integrates the attached memory using cache coherency protocol into a single symmetric multi-processing environment. This makes it a single fault domain for the purpose of high availability management, meaning that there are single points of failure within the server that can cause the entire server to fail.

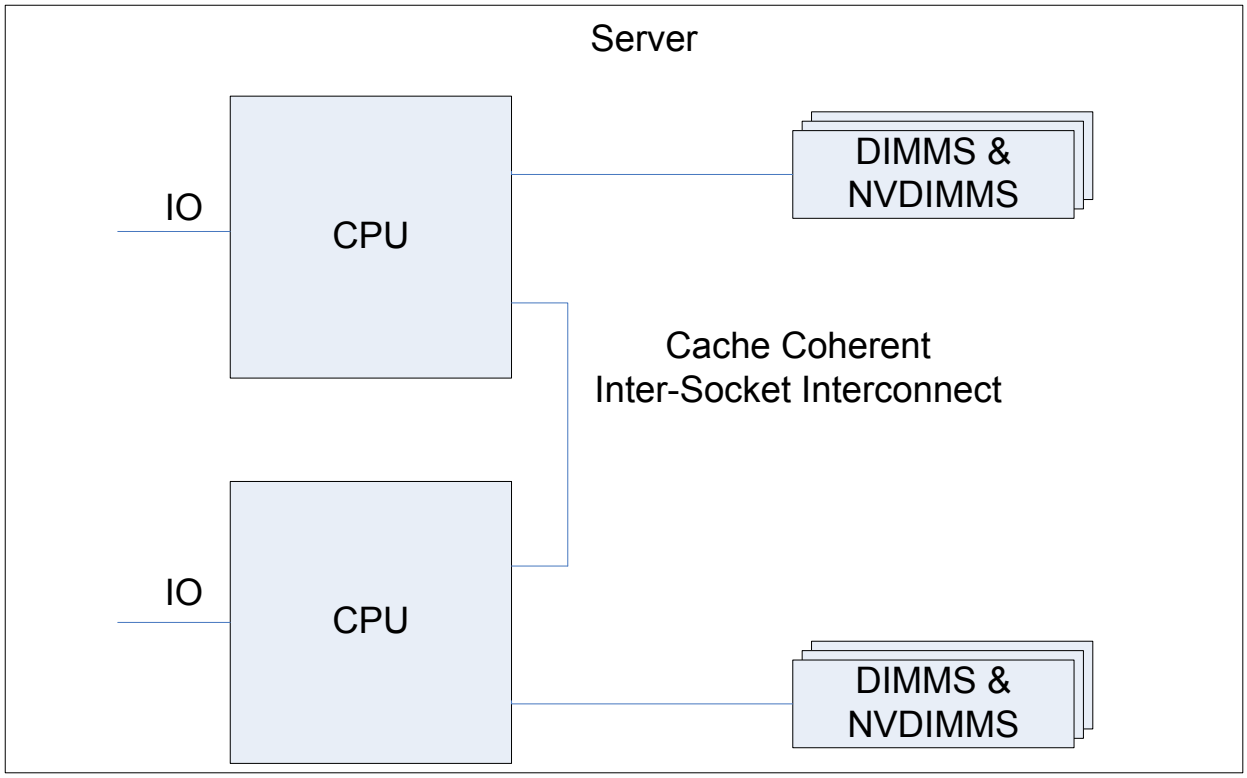

Figure 2 - Local Memory

3.3 Disaggregated Persistent Memory

The concept of disaggregated memory is used to illustrate cases where memory that is not contained within a server is still accessed at memory speed. It is shown in Figure 3 as a memory pool with its own controller connected through a low latency memory interconnect. Implementations of disaggregated memory include recently defined open memory interconnect standards such as Gen-Z and OpenCAPI. Disaggregated memory may or may not be cache coherent with the CPUs in the servers to which it is connected.

Disaggregated memory still looks like memory to CPU's. It operates at memory speed in cache line size units and it may be subject to distance limitations to insure sufficiently low latency. Disaggregated memory is made scalable through the use of optical networks such as those based on silicon photonics to increase the distance of memory speed interfaces. Memory speed refers to access that is suitable for a Load/Store programming model. This requires an operation (Load/Store) rate and latency that allows CPU's to stall during memory access without unacceptable loss of overall CPU performance.

Some disaggregated memory systems may allow memory that is directly connected with one CPU to be part of the pool that is shared with another. Disaggregated memory that is not cache coherent requires the use of distributed programming techniques such as those used in clusters rather than the symmetric multi-processing techniques that apply within a single server.

Disaggregated memory may or may not be a separate fault domain from the servers depending on implementation.

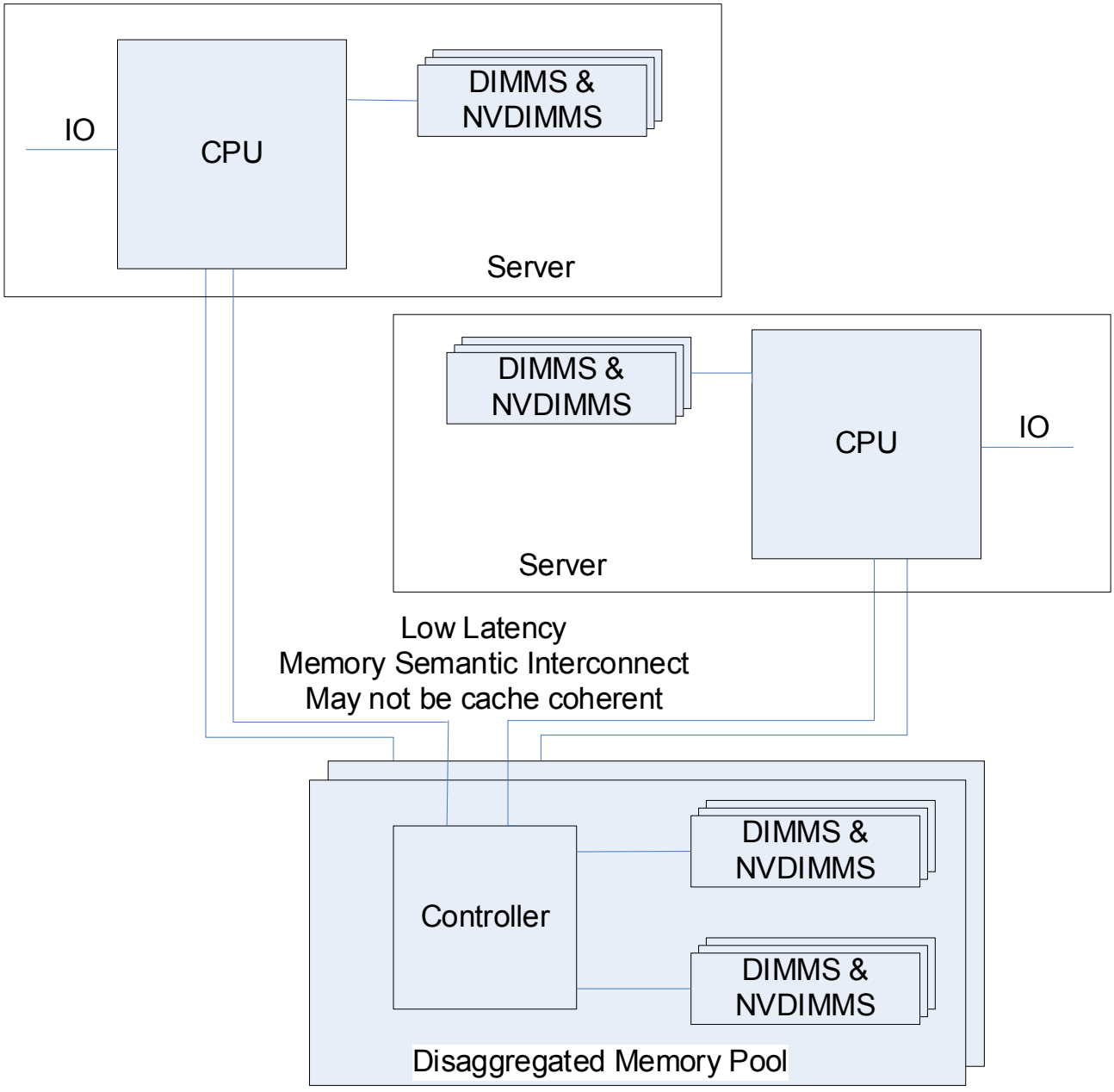

Figure 3 - Disaggregated Memory

While open memory interconnect standards can implement disaggregated memory they are not limited to that purpose. They also enable memory driven computing that includes access to storage, networks and accelerators in a memory semantic interconnect. Physical configurations other than that illustrated by Figure 3 are feasible including co-location of memory pool components with processors. A much broader range of use cases is illustrated in "New Interconnects". In addition to cache line memory access, open memory interconnect standards allow asynchronous bulk data transfers using operations such as "Put" and "Get".

For the purpose of this document disaggregated memory is viewed as an RPM use case even though its remote-ness is a matter of degree. Note that Figure 3 illustrates the potential for redundancy with multiple paths to multiple instances of memory pool components.

3.4 Networked Persistent Memory

Networked memory is accessed through a high speed network rather than directly through a memory interface. Figure 4 shows two servers connected with network adapters. Memory access is achieved over the network using protocols such as message passing and RDMA. The two servers in Figure 4 are in separate fault domains.

The network adapter communicates with the NVDIMMs through the CPU in this configuration. Depending on the CPU architecture there may be volatile buffers or caches on the path from the network adapter to the NVDIMMs.

Networked persistent memory is not cache coherent with the CPU. Unlike local or disaggregated persistent memory where all of the NVDIMMs can be part of a single system image, the NVDIMMs on a remote node are not part of a single system image.

Although only two servers are illustrated in Figure 4, many servers may be attached to the same network. There may be many-to-many relationships between the data stored in various servers. It is also possible to have servers with no NVDIMMs access networked persistent memory on other servers.

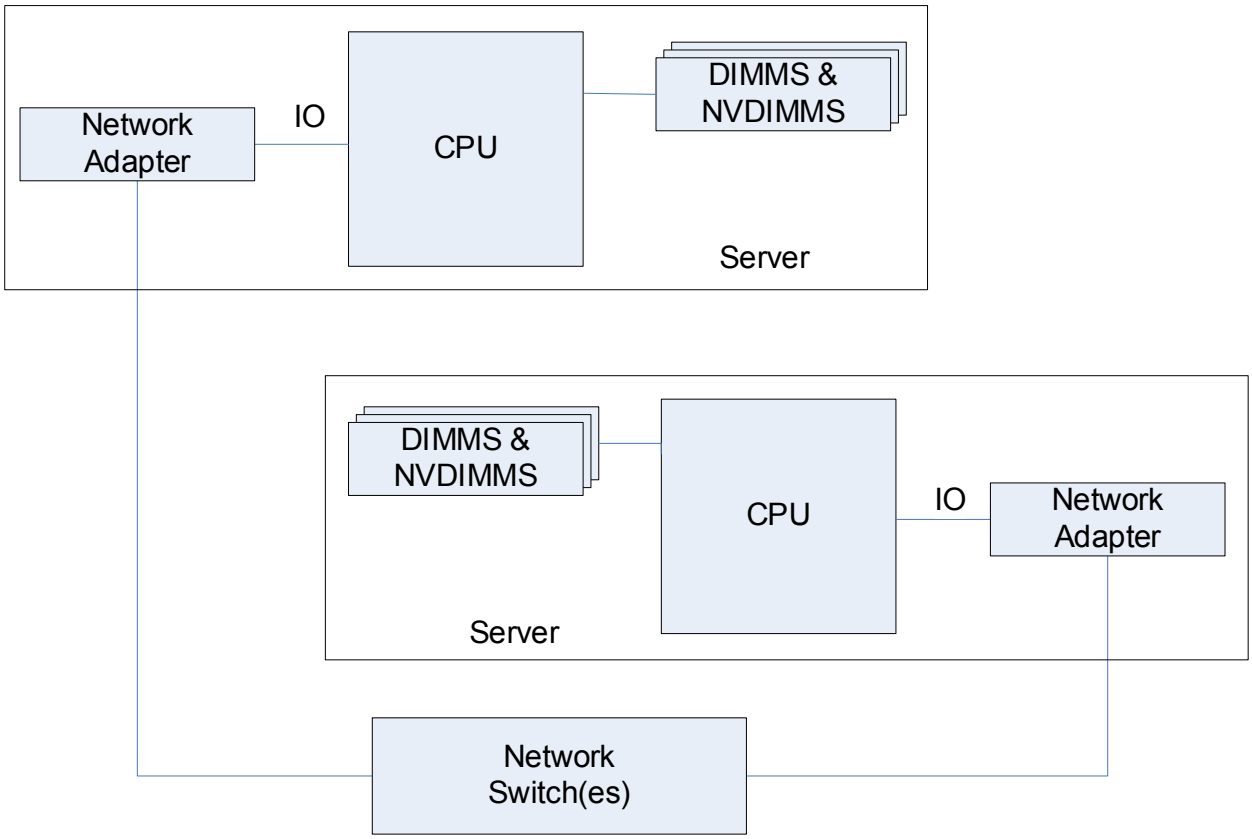

Figure 4 - Remote Memory

Disaggregated and Networked PM techniques may appear in the same system. The result is a system architecture that includes common access to PM across CPU's, accelerators and storage or network interface cards. That architecture enables the following data flows without CPU intervention.

- network adapters can implement RPM by directly accessing disaggregated PM.
- accelerators can use RPM.
- Memory controller features such as encryption can be applied directly to RPM

4 Recoverability Definitions

Since recovery from failure is the purpose of high availability use cases it is important to understand recovery and recoverability in some detail. There are established principles for this in enterprise storage systems but less is known about persistent memory recovery.

4.1 Data Durability vs. Data Availability

Common approaches to redundancy generally support one or both of the following goals.

- High Durability – Data will not be lost regardless of failures, up to the number of failures that the redundancy scheme is designed to tolerate. If the media containing the data can be removed, re-inserted into a new slot and recovered,

data is only lost if removable media modules themselves fail. Otherwise failures of system components other than media modules can also cause data loss.

- High Availability – Data will remain accessible to hosts regardless of failures up to the number of failures that the redundancy scheme is designed to tolerate. Failure of any component between a given host and the data may make that data inaccessible to that host, so redundancy is required for all such components.

If an application requires only high durability, local data redundancy such as RAID across NVDIMMs will suffice. If an application requires high availability as well, remote data redundancy such as RAID across servers or external storage nodes is required. This is illustrated by Figure 5 wherein the local and remote memory of Figure 4 are overlayed with red lines indicating the data flow of a store (ST) operation.

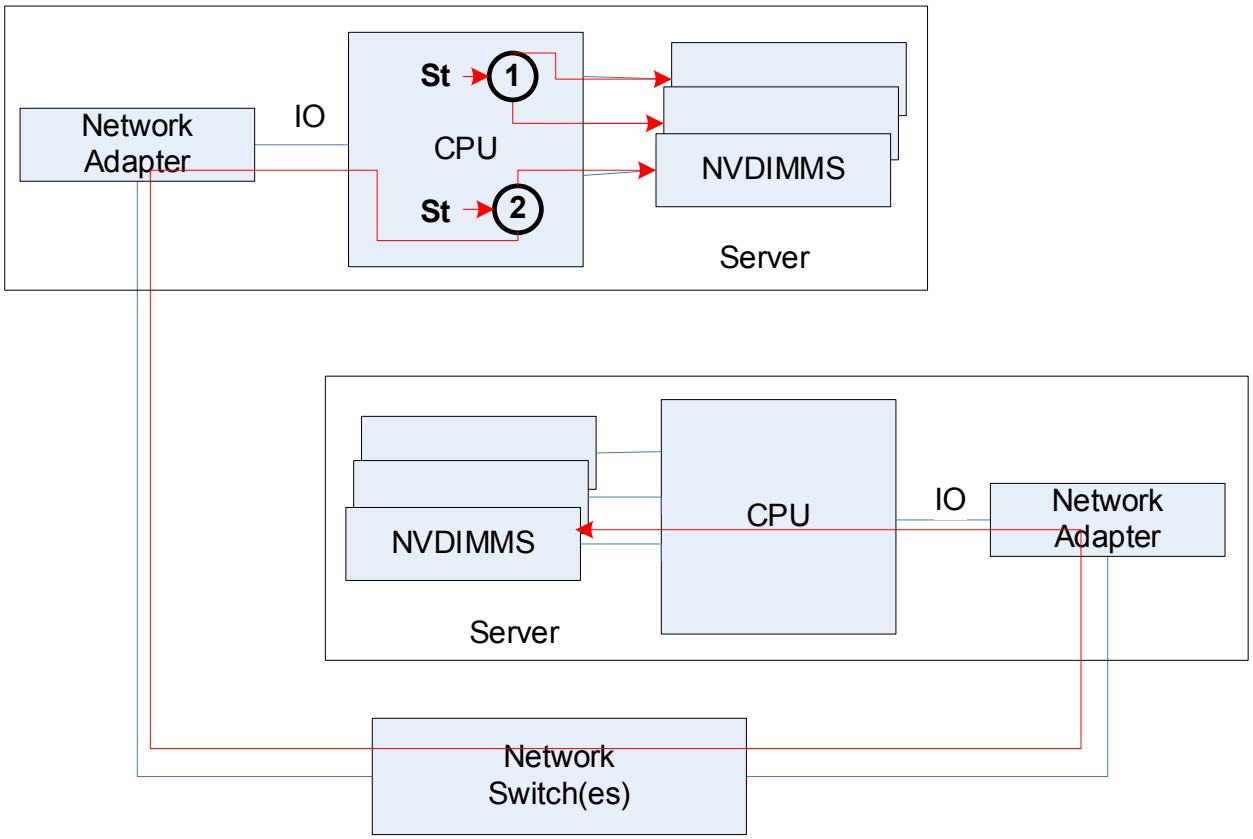

Figure 5 - High Durability vs High Availability

With persistent memory the need to update data redundancy begins with a CPU instruction with an operand that changes a memory location. This is illustrated in Figure 5 within the upper CPU as “St”. A high durability function is represented by the circled numeral 1 where data is mirrored between NVDIMMs in the same server. A high availability function that also provides high durability is represented by the circled numeral 2 where data is mirrored between NVDIMMs on separate servers.

Figure 5 shows exactly 2 copies of data for simplicity. More sophisticated redundancy schemes such as RAID 5 or local erasure coding also apply. The ability of a more sophisticated redundancy scheme to provide high durability and/or high availability

depends on how data is laid out across NVDIMMs and servers. The chief motivations for remote or even geographically distributed copies is the criticality of distributing copies across fault domains.

The distinction between high durability and high availability makes it clear that high availability requires networked access to persistent memory. The network in this figure plays an important “fault isolation” role for high availability. It minimizes the probability that a hardware failure in one server can affect access to redundant data. The role of networks in providing fault isolation for high availability exposes the dilemma of high availability at memory speed.

4.2 Visibility vs Persistence

PM implementation experience has shown that developers need to be sensitive to the distinction between cross-process visibility of data written to local PM, and persistence of data in PM. The term visibility refers to the ability of computation or device elements in a system to read (or “see”) an updated data value that originated elsewhere. This is different from persistence, which refers to the ability of data to survive power loss. Both concepts have to do with the propagation of data being written. As a specific example, it is incorrect to assume that a compare and swap instruction with an operand that refers to PM is sufficient to implement a persistent lock. This is not true because the result of the instruction may become visible to other processes before it becomes persistent. As a result a lock may be granted to a process just before a power loss that causes the lock to revert to a prior state, leading to incorrect concurrency. In other words in today’s systems, visibility and persistence are not achieved together atomically.

When a full range of PM and RPM are taken into consideration, visibility and persistence may occur in either order. For example, in a local multi-processor implementation, symmetric multi-processing protocols insure that all processes on all processors see updates to data either before or at the same time as the data becomes persistent. On the other hand some RPM use cases implement eventual consistency whereby data propagates throughout a distributed system in the background concurrently with ongoing usage. With eventual consistency, persistence may occur before data is visible to all potential consumers.

In RPM use cases it is possible for partial completion of a write to achieve visibility but not persistence even though both are intended. This makes it important to separate consumers of visibility and consumers of persistence into distinct two roles. Actions that are intended to achieve persistence should fail if persistence was not achieved regardless of whether visibility was achieved.

The response to remote persistence failure and the ongoing relationship between visibility and persistence is up to the application and the user mode libraries that it uses. For example, one can imagine several responses to persistence failure after visibility success.

- Visibility is reversed and so the entire action fails and can be retried
- Visibility is enabled while an ongoing process tries to resynchronize persistence
- A distributed application is designed to require only eventual persistence

Note that the last two responses may be applicable at the same time.

4.3 Consistency Points

In order to recover correctly from a failure, all of the data items recovered must have correct values relative to each other from the application's point of view. The meaning of "correct" in this case is entirely up to the application. For example financial transactions involving multiple accounts must yield a correct result even if a hardware failure occurs during the transaction.

Applications use a variety of techniques to assure consistency, primarily by controlling the order of changes to individual data items in such a way that a consistent state can always be achieved after failure. One common way to achieve this is to use transactions. There is often some data processing required after a failure to bring an entire data image into a consistent state. For example, uncommitted transactions may need to be rolled back.

Since a failure can occur at any time, systems must be prepared to convert any data state that could result from a hardware failure or restart into a consistent state. This is much easier to achieve if applications designate certain instants in time during execution as consistency points. By identifying consistency points an application can allow underlying infrastructure to orchestrate recovery that always results in a consistent data image.

For example, in today's enterprise storage systems applications can coordinate the creation of snapshots with storage systems and file systems using protocols like Microsoft's Volume Shadow Copy Service (VSS™). VSS allows applications to orchestrate storage snapshots at points in time when application data is consistent. That is fine for backups because they are infrequent compared to IO's, and even more infrequent compared to memory accesses.

As another example, suppose an application was able to involve persistent memory in transactions so that the completion of each transaction represented a consistency point. "NVM Atomics", the subject of a SNIA white paper, suggests a standard way for applications to view transactions that could enable this type of interaction.

The important thing about consistency points relative to RPM and high availability is that they create opportunities to optimize networked persistent memory communication. This application level requirement may feed subordinate requirements that can affect RPM implementations such as ordering and atomicity with respect to failure (power or hardware) or reset.

4.4 Crash Consistency in Disk Based Systems

Crash consistency is another common recovery model in today's storage systems. Since the dawn of computing time, disk drives have defined the gold standard for all types of storage system behavior. Disk drives perform multiple reads and/or writes concurrently so the order of completion of outstanding operations is indeterminate. In addition, if power fails during a write it may be partially completed. Some storage systems offer additional guarantees about write completion. These give rise to the

"Atomicity Granularity" attributes of the SNIA NVM Programming Model. Operating systems may provide additional semantics atop these primitive behaviors as well.

Since disk drives and storage systems offer such weak ordering guarantees, applications must be prepared to recover from any state of the writes that were in flight when a failure occurs. This brings us to the concept of crash consistency, in which the state of a storage system after a failure need only match the indeterminate write order guarantee of a group of disk drives.

More formally, a storage subsystem state is considered crash consistent if it could have resulted from power loss of a group of direct attached disks given the sequence of write commands and completions leading up to the failure. This means that there is a rolling window of outstanding write requests whose order is uncertain. Applications must be able to recover from any order of those requests and must account for storage system atomicity nuances in the process. For an application, recovery from a crash consistent image is the same as a cold restart after a system crash.

4.5 Crash Consistency in PM Systems

Now consider the map-and-sync methodology described in the NVM.PM.FILE mode of the NVM Programming Model. Sync has a very specific meaning. The only guarantee that sync makes is that all stores in the address range of the sync that occurred before the sync are in persistent memory when the sync completes. Sync does not otherwise restrict the order in which data reached persistent memory. For example, if cache lines 1 through 5 were written in order by the application before the sync, cache line 5 might have reached persistent memory first, possibly before the sync even started. This flexibility enables potential write order optimization for cache performance. Unfortunately it also creates ordering uncertainty analogous to that of crash consistency in disk based systems.

The lack of ordering certainty gives rise to a lowest common denominator for NVM.PM.FILE recovery similar to that which exists for disk drives. Specifically, the application is uncertain as to which of the store instructions between two sync actions will appear in persistent memory after a failure that occurs before completion of the second sync action. If the actions and attributes of the NVM Programming Model are all that is available then the application must execute additional sync actions whenever the order of stores to persistent memory matters.

More formally, a persistent memory range is crash consistent if its contents at the start of recovery could have resulted from the pattern of stores and syncs executed on the initiators (processors or other sources of memory access) with data in flight to the persistent memory prior to failure. In both disk drives and persistent memory, some aspect of data atomicity with respect to failure is built into the crash consistency assertion. Specifically, both the order and atomicity properties that are guaranteed for local media must be duplicated at the remote site. The NVM programming model describes atomicity for both disk drives and persistent memory. Based on the PM model, unless the atomicity of fundamental data types provided by the local processor is conveyed to the remote node, applications will need to use error checking such as CRC on all data structures that need atomicity. The error check must be stored in such a way

that atomicity can be verified after a failure that calls the remote copy of the data into use. This is covered in more detail in section 6.5.

Crash consistency applies to literal data images as seen by processors. If crash consistency is applied across nodes with different types of processors, the memory layout at each node must be such that the applications running on the processor(s) connected to that memory see the same data image created at the local site. This must account for processor architecture specific bit and byte ordering practices. Crash consistency does not account for other types of data formatting as might appear in the presentation layer of a network stack.

Crash consistency is a complex approach to recovery from an application standpoint. It also forces considerable overhead to precisely communicate every sync action to networked persistent memory. This further illustrates the motivation for some notion of consistency points such as persistent memory transactions and their relevance to high availability use cases.

4.6 Recovery Point Objective

Another analogy between persistent memory and enterprise storage systems relates to the concept of a recovery point objective (RPO). A recovery point objective is the maximum acceptable time period prior to a failure or disaster during which changes to data may be lost as a consequence of recovery. Data changes preceding a failure or disaster by at least this time period are preserved for recovery. Recovery point objectives are part of today's disk based disaster recovery service level agreements. Although they are most often expressed in terms of time, recovery point objectives can also be specified as an amount of data changed, either in terms of bytes or operations such as writes, stores or transactions.

Zero is a valid RPO value. In today's disaster recovery systems an RPO of 0 mandates synchronous remote replication. As a result at least one round trip to the remote site and back is added to the time it takes to do a write. In addition, enough bandwidth must be available to transmit every write to a remote site even if the same data blocks are written repeatedly in rapid succession. Clearly this high level of consistency comes at a significant cost in performance.

A non-zero RPO allows writes to flow to remote sites without slowing down local writes, as long as the remote site does not get too far behind the local site. In addition, there are opportunities to gather multiple writes to the same address within the RPO time window into one write to the remote site.

The NVM.PM.FILE mode of the NVM Programming model includes an "Optimized Flush" action which insures that a list of memory address ranges have been flushed from the CPU to PM. These groups of address ranges must also, at some point, become redundant in networked persistent memory. If we apply the recovery point objective concept to persistent memory then we can delay transmission of data to networked persistent memory so long as a consistency point is achieved at the remote side within the RPO time window. Delayed transmission allows data transmission to be batched into larger messages which reduces the net overhead of high availability.

Having introduced the concept of RPO we can consider the state of memory at the end of any “Optimized Flush” action to be used as a consistency point. If an application is managing durability using only “Optimized Flush” and/or “Sync” actions then the consistency point can be at least crash consistent. If an application is more involved in managing durability atomically as with transactional persistent memory, the consistency point may be more optimal. In either case the RPO can be used to determine how often one of those candidate consistency points actually appears in remote PM. As with remote replication, this requires additional time in order to optimize the flow of data to networked persistent memory.

The data for a consistency point can be placed in networked PM in any order that results in a state that meets the requirements of a candidate consistency point. For a crash consistent candidate, the state of networked PM must adhere to the constraints imposed by optimized flush or sync actions generated by the application. If the consistency point is stronger, the constraints imposed by additional application interaction such as transaction constructs must also be applied to the state of networked PM. Both of these include the atomicity considerations described in section 4.4.

Write intensive applications that truly require RPO=0 are not likely to experience good performance with persistent memory. RPO=0 imposes at least one network round trip per optimized flush or sync. In addition, today’s systems do not assure that data has reached persistent memory on the remote PM before the remote data placement completes from the local server’s point of view. This could require another network round trip just to assure durability at the remote node.

Network characteristics can be very important to the ability of a system to meet RPO’s that involve RPM. Generally, network deployments must be analyzed to determine the range of RPO’s a network can support based primarily on its latency and bandwidth characteristics. Such analysis is a well understood topic unto itself that is beyond the scope of this white paper.

4.7 Recovery Scenarios

To explore data recovery scenarios more deeply, consider the implications of the Error Handling appendix of the NVM Programming Model specification. This, combined with reasoning about sync/flush semantics and consistency points enables enumeration of several scenarios based on the following criteria:

- Did a server fail? Server failures include anything that inhibits an application running on a server other than a storage or memory device from accessing the data that is in its local memory.
- Was a server forced to restart?
- Did a precise, contained memory exception occur?
- Is the application able to backtrack to a recent consistency point without restarting, such as by aborting transactions?
- How up to date (fresh) is the redundant data?

Permutations of these criteria create a handful of recovery scenarios.

Scenario	Redundancy freshness	Exception	Application backtrack without restart	Server Restart	Server Failure
In Line Recovery	Better than sync	Precise and contained	NA	No	No
Backtracking Recovery	Consistency point	Imprecise and contained	Yes	No	No
Local application restart	Consistency point	Not contained	No	NA	No
		NA	NA	Yes	No
Application Failover	Consistency point	NA	NA	NA	Yes

The following sections elaborate on each scenario.

4.7.1 In line recovery

In this scenario, the primary copy of a memory location is lost and if a copy is available (or the equivalent) the data is recovered during a memory exception without any application disruption. The control flow for this scenario is as follows:

- A precise, contained memory exception interrupts the application. The exception handler of the NVM.PM.FILE implementation handles the exception,
- The NVM.PM.FILE implementation determines that it can recover the lost data either locally or from networked PM.
- The NVM.PM.FILE implementation restores the lost data to local PM
- The application returns from the exception, causing the interrupted instruction to successfully retry the memory access.
- The application continues from that point without any application level exception handling or recovery.

This type of recovery requires that the recovered data be the most recently written data. Sync semantics do not guarantee sufficient recency for this type of recovery. Consider the following sequence of events:

```
A := 1;
OptimizedFlush(…&A…);
A := 2;
B:= A;
<processor automatically flushes 2 -> A before sync>
C:= A;
<failure to read A from PM causes interrupt during C:=A;>
<NVM.PM.FILE implementation restores value 1 -> A based on latest sync>
<processor repeats C:=A, assigns value 1->C;
OptimizedFlush(…&A,&B,&C…);
```

If there were no failure, A, B and C would all equal 2 at the end of the above code segment. However, a failure may occur such that B equals 2 and A and C equal 1. That is because nothing about map and sync semantics keeps the processor from flushing cached variables to PM before the sync action. Therefore any redundancy that is created during or after sync may not be sufficiently up to date to restore data in such a way as to assure correct application execution without backtracking (see section 4.7.2).

The RPO logic described above commences with the sync command. This means that even when RPO=0, backtracking is required during recovery to adjust work in progress, by means such as aborting transactions. Note also that this is really a high durability scenario rather than a high availability scenario because there was no server failure.

4.7.2 Backtracking recovery

In this scenario an application is able to recover from memory exceptions by identifying, aborting and retrying transactions, or other application specific equivalents.
The control flow for this scenario is as follows:

- A contained memory exception interrupts the application. The exception handler of the NVM.PM.FILE implementation handles the exception. Backtracking recovery is potentially applicable even if the exception is not precise. An imprecise exception does not allow resumption of execution at the interrupted instruction.
- The NVM.PM.FILE implementation determines that it can recover the lost data either locally or from networked PM.
- The NVM.PM.FILE implementation restores the lost data to local PM. The restored data is not guaranteed to be any more recent than the last consistency point. All committed transactions must be included in the last consistency point or in consistency points before that.
- NVM.PM.FILE may be able to determine whether the page containing read data in error has been modified since the last flush. If it has not been modified, the error handler can restore the data and transparently resume execution without backtracking. If that happens then the remaining steps in this description do not apply.
- The application receives an exception event or signal along with an indication of the address ranges that were restored. If all of the restored data is guaranteed to be covered by committed transactions then the application can return from the exception and continue processing in line. Depending on the application and/or transaction implementation the contents of some roll forward logs in committed transactions may need to be re-applied to the recovered page before returning from the exception. If some of the data is covered by uncommitted transactions and the rest is covered by committed transactions then backtracking recovery proceeds by aborting transactions and resuming application work flow at a point that will cause aborted transactions to be retried.

This scenario clearly describes a relationship between transactions and recovery, since aborting transactions is the means of backtracking referenced. It would be helpful for the transaction service to assist in determining which of the recovered data items are related to a given transaction. Such a determination could then be used to ascertain the

minimum set of transactions that need to be aborted or reapplied to recover from the restoration.

Depending on the application this type of recovery may require RPO=0 with respect to transaction commits. On the other hand, some applications may be able to recover from arbitrarily old memory states without restarting.

4.7.3 Local Application Restart

In this scenario an application restarts in order to complete recovery from a data loss. The term restart is used here to refer to the resumption of application execution from an initial state such as would occur after the application's process(es) were killed. This scenario applies if neither the in line nor the backtracking scenarios were applicable and the server running the application has not failed. The control flow for this scenario is as follows.

- The application restarts. This could be the result of decisions by the application itself, some other hardware, software or administrative intervention, or power loss.
- Recovery code that may be specific to the application or part of a transaction service uses NVM.PM.FILE.GET_ERROR_INFO to identify persistent memory ranges that may require recovery over and above that which may have occurred before the restart. If data recovery is required, human or file system intervention may be required to restore data to a consistency point based on file system redundancy features or backups.
- At this point the persistent memory image must represent a consistency point as described above. Application specific code or a transaction service cleans up the consistency point by completing committed transactions and aborting uncommitted transactions.
- The application completes the restart based on the cleaned up persistent memory image and resumes processing. Application work flows that involved aborted transactions may need to retry those transactions.

This type of recovery can use RPO>0 on all of the data in a persistent memory image. The reference here to a persistent memory image is significant in that all of the data within the scope of the application must be restored to a state that represents the same consistency point.

4.7.4 Application Failover

In this scenario a server failure forces the application to restart on another server. This is generally the result of hardware failure that causes data to be inaccessible to applications running on a server, or that renders it incapable of running an application. For the purpose of this description the entire server is considered to be failed if any part of it has failed. In cases of intermittent or partial failure a failover policy must determine when the server is designated as failed.

A failover relationship must be constructed and maintained with the target server(s) of a failover including the following capabilities.

- Server failure must be detected and communicated to a server capable of taking over.

- The failing server must stop execution and be isolated from non-failing servers so as to insure that no artifacts of its execution could interfere with ongoing operation.
- The server taking over must have or obtain access to a persistent memory image that represents a consistency point from which the application can restart.

An example control flow for this scenario is as follows.

- A non-failing server capable of taking over an application (or a portion thereof) from a failing server is notified of the server failure. This can be the result of the failing server detecting its own failure, or it can be detected by a monitoring service such as a heartbeat.
- The non-failing server identifies all of the PM relevant to the application based on configuration information and takes measures to insure that the failing server no longer has access to the surviving copy or copies of the data.
- If the non-failing server does not have local access to all of the PM relevant to the application, data is migrated to local PM from networked PM on other non-failing servers.
- The application restarts on the non-failing server as described in section 4.7.3 except that it uses an image of a consistency point that does not depend on any PM that is contained within the failing server and is fresh enough to adhere to the RPO.
- During or after application restart, data that lost redundancy due to the server failure is rebuilt provided that PM resources are available for that purpose.
- After the server is repaired or replaced it can resume participation in the HA system running the application once it has regained access to a complete local PM image.

Note that this scenario involves logistics of application failover that go beyond PM. These logistics are generally provided by additional failover services related to the OS or hypervisor that integrates a failover cluster.

4.8 Flush synchronization, ordering and scope

Version 1 of this white paper motivated additional work on the "NVM.PM.FILE.OPTIMIZED_FLUSH" action that appears in the "NVM Programming Model". OPTIMIZED_FLUSH comprises a series of flushes followed by a store barrier. As a result there is no way specified in the programming model to initiate flushes concurrently with ongoing activity. Such concurrency would be desirable so as to reduce the time spent completing the store barrier, especially when RPM is involved. For this reason the TWG is acting on a proposal to optionally split OPTIMIZED_FLUSH into two actions.

- NVM.PM.FILE.ASYNC_FLUSH – Initiate flushes for one or more memory ranges but do not wait for flushes to complete
- NVM.PM.FILE.ASYNC_DRAIN – Wait for all flushes within the scope of the drain to complete

ASYNC_FLUSH is useful for accelerating RPMA by enabling more concurrent data flow leading up to a synchronization point, such as the end of a transaction, that triggers

ASYNC_DRAIN. The split also benefits some local PM in some scenarios as well. Note that the use of the “ASYNC” prefix is intended to connote asynchrony between flush and drain. It does not denote non-blocking actions as ASYNC_DRAIN implementations, in particular, may block. Although OPTIMIZED_FLUSH will continue to exist in the Programming Model the remainder of this section describes ordering in terms of ASYNC_FLUSH and ASYNC_DRAIN. OPTIMIZED_FLUSH is simply a hard coded sequence of the two.

Figure 6 provides a context for reasoning about RPMA ordering. Here we see an application comprised of multiple application components such as threads or processes running on Node 1. Each application component creates a stream of RPM operations such as Load, Store and Move instructions as well as ASYNC_FLUSH and ASYNC_DRAIN calls. An RPMA library translates the stream into operations that can be interpreted by underlying network adapters and switches.

The ordering of operations enforced by the network is represented by an Order Nexus, the precise nature of which is implementation specific. At Node 2, Network hardware and software applies the operations in each stream to PM regions. Even though no overlapping PM regions are shown, certain ordering constraints apply the all of the regions even across streams. If regions were to overlap each other, applications would need to implement additional explicit ordering constraints that are beyond the scope of this document. The node and network labels overlap intentionally to indicate that networking may involve both hardware and software. The order nexus overlaps the application to indicate that application access to networks must be configured in a way that accounts for the network’s order nexus implementation. The order nexus overlaps RPM to indicate that RPM implementations may also need to account for ordering.

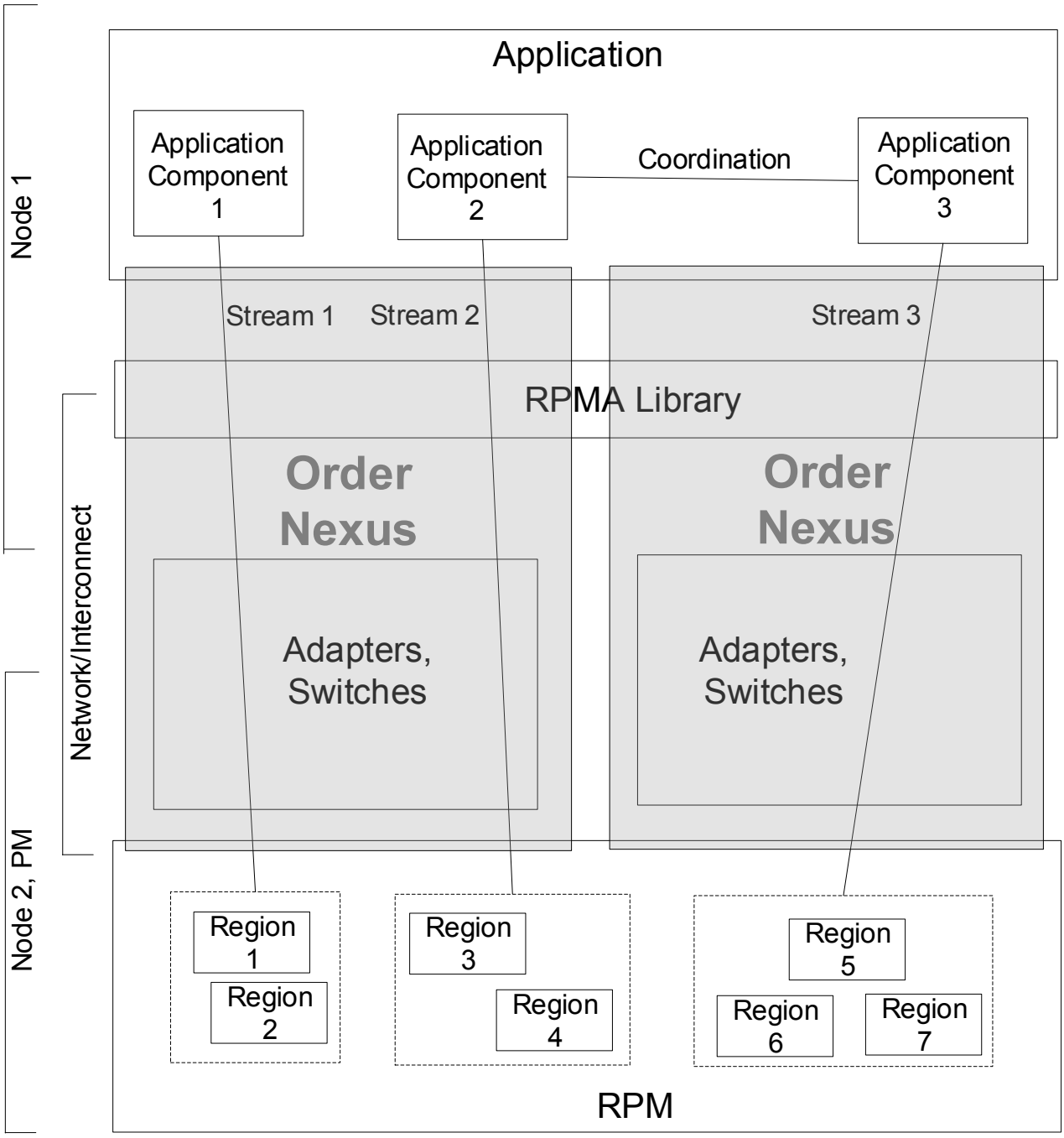

Figure 6 – Operation Streams and Ordering

The only point of store or move ordering required by the NVM Programming Model that applies to RPMA is a type of store barrier that is represented by ASYNC_DRAIN. Store barrier semantics state that all stores or moves to ranges where flush has been requested will reach persistence in a way that enforces a happens-before order relationship. Specifically, the completion of all ASYNC_FLUSH actions that apply to memory regions in the same scope as an ASYNC_DRAIN must happen before any actions that follow the ASYNC_DRAIN. The order nexus boxes in Figure 6 represent the scope of happens-before relationships.

In this model, an Order Nexus is assigned to each application component during its initialization. We will refer to this initialization as session creation. When multiple application components share an order nexus, ASYNC_DRAIN actions in any of their streams apply to the memory regions named in ASYNC_FLUSH actions across all of their streams. On the other hand if an application has components assigned to multiple Order Nexus's it is up to the application to coordinate ASYNC_DRAINS across its components.

The NVM Programming Model specifies that data must be within a persistence domain prior to the completion of OPTIMIZED_FLUSH. The implementation of this requirement depends on various aspects of systems including ordering implementation. Some implementations may apply this same requirement to RPM. This does not necessarily require network services to wait for store barriers to complete as long as ASYNC_DRAIN and OPTIMIZED_FLUSH have some means of implementing the NVM Programming Model requirement.

ASYNC_DRAIN allows implementations to force additional data to RPM over and above the ranges described in ASYNC_FLUSHes. This becomes a tradeoff between implementation complexity in the form of tracking metadata for individual ranges, and data pipeline efficiency in the form of minimized data movement. The order nexus abstraction is intended to allow implementations make this tradeoff while leveraging native constructs such as queue pairs and enforcing order in a well-defined manner with respect to the Programming Model.

The native constructs that implement an order nexus may also be relevant to system initialization and failure recovery such as the following.

- Although the means of associating applications with network resources are network and protocol specific, they become associated with an order nexus implementation. The application must be configured during session creation in such a way that the order nexus associations implied by the network configuration properly insure data consistency.
- In the event of network failure, sessions may need to be created over again so that native constructs that form an order nexus are re-initialized. Should this occur, recovery actions must assure application data consistency. This may involve multiple layers of RPM stack implementation over and above hardware and order nexus implementations. Additional detail on error recovery is covered in section 7.

ASYNC_FLUSH and ASYNC_DRAIN expose another application level tradeoff regarding how long to wait between Stores (or Moves) and ASYNC_FLUSHes. Likewise, there may be application choices regarding the accumulation of ASYNC_FLUSHes before an ASYNC_DRAIN. If the system implements an RPO as described in section 4.6, these decisions are likely to be thereby constrained. Since these decisions may be influenced by the presence of RPMA in the stack so it would be useful for the application to be able to detect that presence. Beyond that, these considerations are application specific and beyond the scope of this document.

4.9 Integrity Checking

The NVM programming model version 1.0 included an “Optimized Flush and Verify” action that was initially thought by some to be analogous to “Write and Verify” SCSI command for disk drives. Further investigation proved that verification is at best expensive and often impractical. In RPM scenarios, given various network, CPU and memory component implementations, it can be so difficult to assure that data became persistent that applications should consider explicit integrity checking of data after it reaches persistent media. This could be done using actions that enable a hash computation that is known to both the consumer and the provider of the RPM. Implementation could include the following.

- Upper layers of an RPMA stack negotiate common hash algorithms implemented in storage, network or software infrastructure.
- RPM providers could compute hashes for each region designated in a new “Integrity Hash Calculation” option or action. In this scenario RPM consumers would compare hashes with known correct values.
- -or- RPM consumers could compute hashes and send them to RPM providers where they are re-computed using stored data and compared. Verification could occur during writes and/or in response to a separate action or option.

Additional benefits of integrity checking include the following.

- Faster detection and isolation of data corruption, giving corrupted data less time to propagate.
- Greater assurance of correct and timely error recovery, thus reducing the negative impact of exceptions.
- More rigorous and efficient background data scrubbing to detect data issues sooner, reduce repair time, increase availability and decrease probability of data loss in redundant systems.

Those familiar with SCSI can see how this generalized description could include a range of implementations, some of which align with DIF/DIX features.

5 HA Extensions to NVM.PM.FILE

Figure 7 illustrates a layering of software modules that includes the following features.

- User space NVM.PM.FILE implementation represented as libraries to the application
- User space based replication via RPMA (e.g. RDMA or Memory Interconnect protocol) to persistent memory in separate hardware
- Local and remote file systems. The local file system is PM aware and supports memory mapping. The remote file system stores data in PM and allows it to be accessed using an RPMA protocol such as RDMA.
- User space optimization to access remote memory

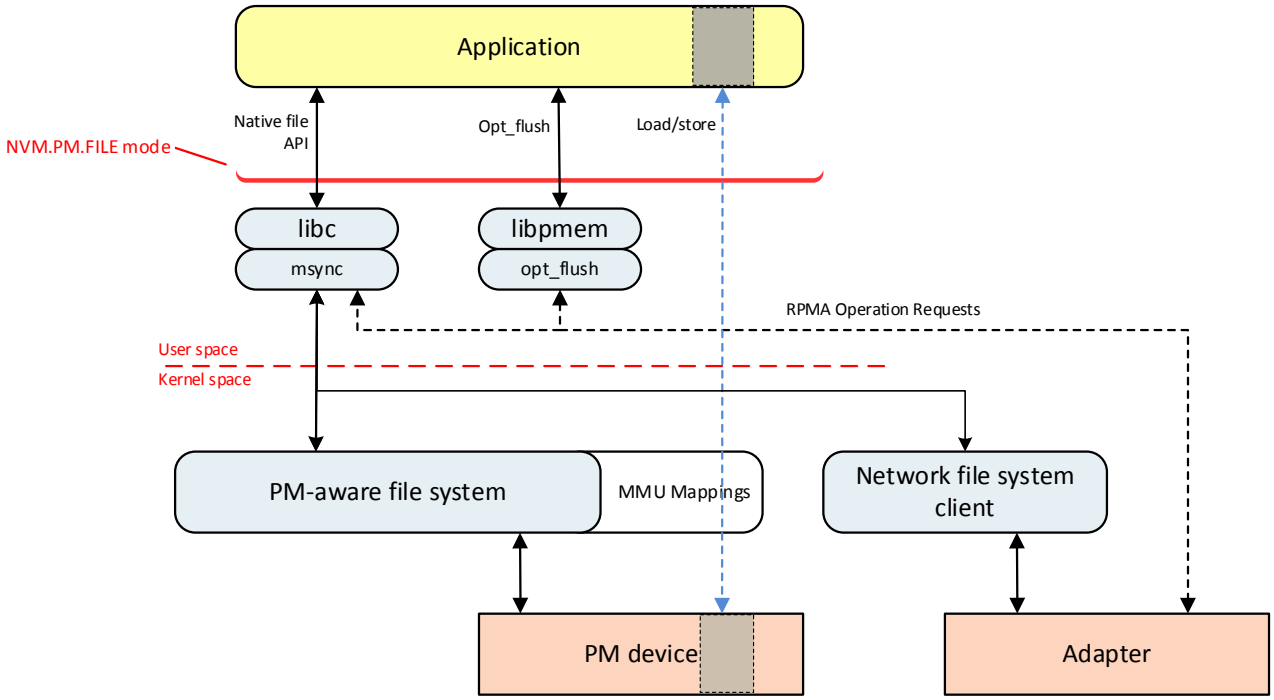

Figure 7 – HA Extension to NVM.PM.FILE

The application is presented with an implementation of NVM.PM.FILE with the assistance of user space libraries. One of these consists of the standard file system API while the other implements NVM.PM.FILE.OPTIMIZED_FLUSH. The load/store capability of the application is shown in the center of the diagram as it is enabled once files are memory mapped.

Using the NVM.PM.FILE mode we see that replication software (e.g. RAID or erasure coding) is implemented in the user space library. This software enables construction of a high availability solution by communicating with both the local file system and a remote file system via the network file system client and adapter illustrated to the right of the PM-aware file system and the PM device.

The user space library is capable of setting up a remote memory access session with the remote file system using well known means such as RDMA, OFA libfabric or an open memory interconnect. The session can then be accessed from user space to enable data to be written to networked PM for redundancy without context switching. The user space “msync” and “opt_flush” use the session for this purpose during sync and optimized flush actions respectively as represented by the black dotted arrow. Replication for HA is achieved when the remote write reaches the persistence domain in the remote system as a result of the remote memory access. The optimized flush and native API paths may use each other’s implementations should it be advantageous to do so.

6 RPMA for HA

This section provides additional detail on Remote Persistent Memory Access (RPMA) for HA in the context of the software model described in section 5.

6.1 Peer to Peer Deployment Model

The following figure illustrates two servers, each of which runs an NVM.PM.FILE implementation in cross-communicating client server file systems.

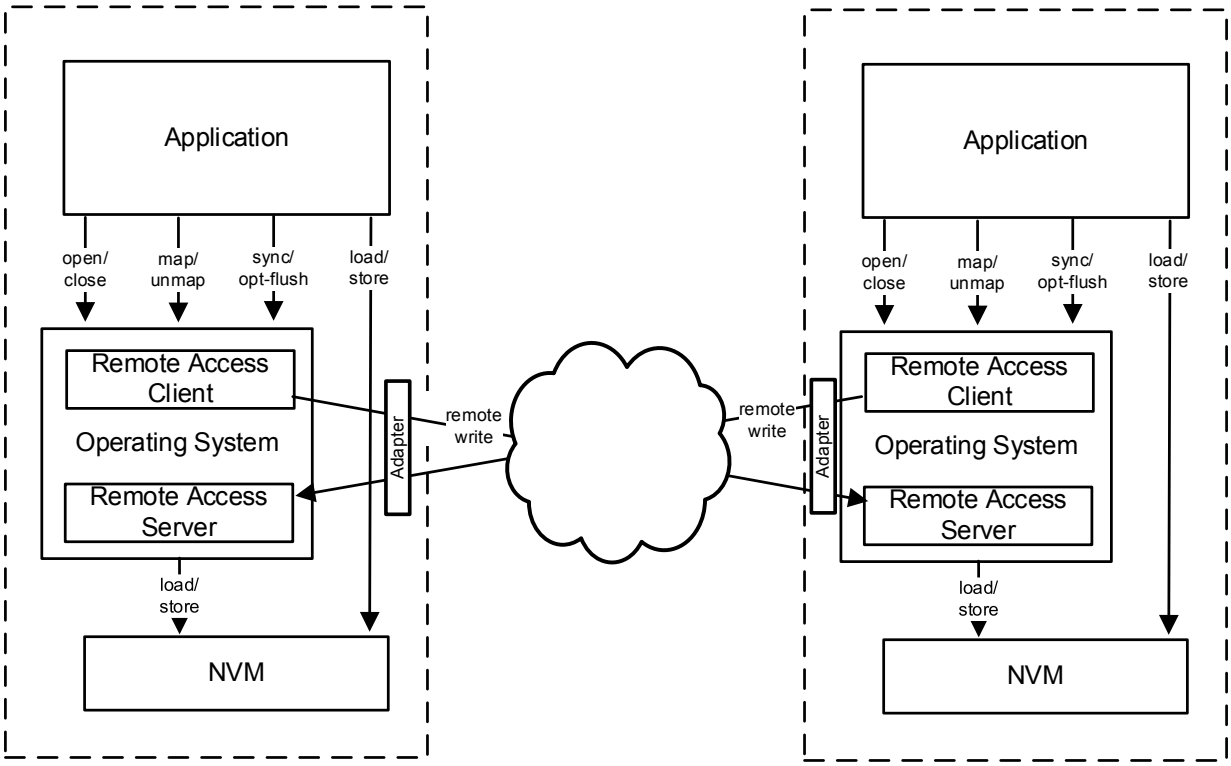

Figure 8 – NVM.PM Peer to Peer HA Replication Deployment Diagram

Peer A and Peer B are physically separate servers or server blades connected by a network or memory interconnect. Each server has access to the other's file system in a client/server configuration such as NFS or SMB. Both message passing and RPMA communication passes between the remote access clients and servers as indicated in Figure 7. Each peer only has memory mapped access to local NVM .

The adapters in this figure are specific to the services used to implement RPMA. If RDMA is used, the adapters may be, for example, HCAs or RNICs. If a memory interconnect is used the adapter may be part of an IO memory controller implementation for a protocol such as Gen-Z or OpenCAPI.

6.2 Address Spaces

The use of RPMA with memory mapped files introduces additional address spaces which must be correlated by various elements of the system. Figure 9 enumerates those address spaces. The vertical axis represents numerical address assignments. The placement of the arrows illustrates the fact that only the physical addresses used in the file system's view of the media coincide.

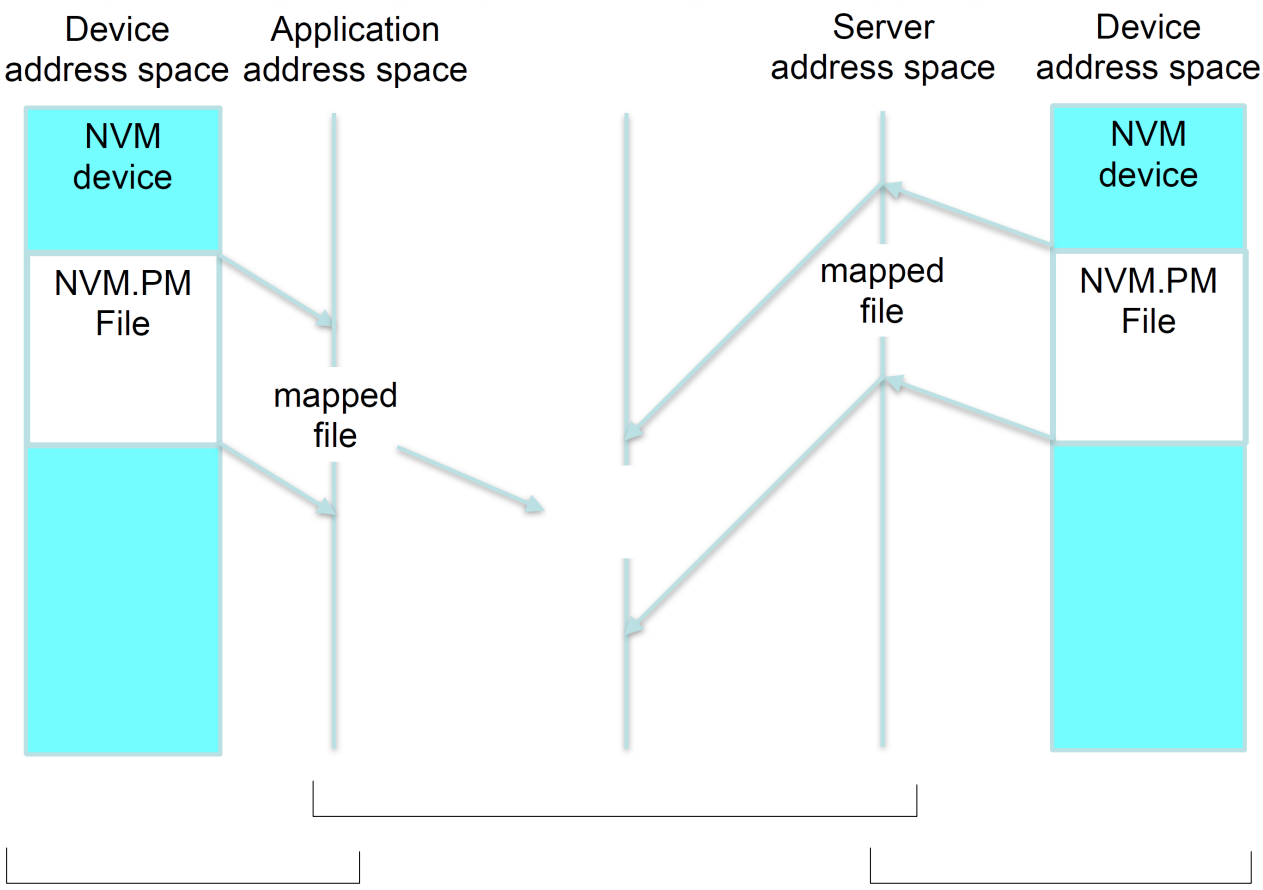

Figure 9 - Memory Mapping and Session Address Spaces

Starting at the left we see the physical PM address space as viewed by the CPU running the application. The notions of virtual and physical addressing are always relative to a point of view. In this case, the CPU observes contiguous ranges of physical memory addresses that represent a file resident in PM according to the file system's metadata and allocation policies. Media controllers closer to the actual physical media may introduce additional address virtualization for purposes such as defective media replacement.

The application address space column represents the CPU's memory mapping unit providing the application with virtual addresses for ranges of PM as part of the NVM.PM.FILE.MAP implementation. The mapping between the first and second columns of Figure 9 is typically maintained by operating systems using page tables. This virtual address space must align with the application's method of resolving pointers among persistent data structures. The alternatives for pointer resolution are described in the NVM Programming Model Appendix A.

RPMA is initialized by creating a session for exchanging data between servers. When the session is created using things like RDMA, libfabric or memory interconnect protocols, an additional address space is created to rapidly and securely correlate registered memory across the adapters in Peer A and Peer B. This session address

space has no numerical alignment with any of the other address spaces. The mapping between the session address space and the application and server address spaces is under the control of the session-aware layers in the adapters and related software on each peer. The RPMA session is used by the user space msync and opt_flush implementations shown in Figure 7. These implementations use remote write operations to copy portions of the application address space from the client to the server for replication. This copy process is represented in Figure 9 as a single arrow from the application address space to the session address space. The other arrows appear in pairs to represent address range mappings at multiple layers in the system.

The server address space column represents the virtual memory address space in the peer running the remote access server as shown in Figure 9. For HA purposes the application and server virtual address spaces do not necessarily need to align as long as the file system metadata reflects the byte-wise correlation of redundant data within files. As with the application, the mapping between the server address space and the device address space is maintained by the OS on the server.

As described in the scope of this document, sharing data in PM for purposes other than HA is not considered here. If real time sharing were a consideration, additional constraints might apply to the correlation of the virtual address spaces between the application and server columns.

6.3 Assurance of Remote Durability

In most of today's hardware implementations, completion of a write is not sufficient to guarantee that data has reached persistent memory. This is because the path from an adapter to an NVDIMM as shown in Figure 4 goes through several buffering stages as it traverses the peers, including I/O busses, networks and CPUs. Within the CPU there are generally buffers or caches that are not necessarily flushed by the CPU before the network adapter responds to the remote write. For example, in some CPU architectures there are several levels of volatile buffers or caches that may need to be flushed depending on system configuration. This may include PCI buffers, Memory controller buffers and possibly CPU caches. This creates hidden inconsistency between redundant PM images that could lead to inaccurate recovery from hardware failure after power loss.

This can be rectified if peer A signals peer B to trigger a flush of any buffers on the IO bus (generally PCI) to memory path. Unfortunately this creates significant overhead compared to the low latencies of local NVM.PM.FILE access. It would be highly desirable to avoid this overhead.

6.4 RDMA Example

Figure 10 illustrates the interaction between the two servers and the adapters that interface them to the network as illustrated in Figure 4 and Figure 8. RDMA terminology is used in this example although the same sequences of events can be orchestrated using other RPMA technologies.

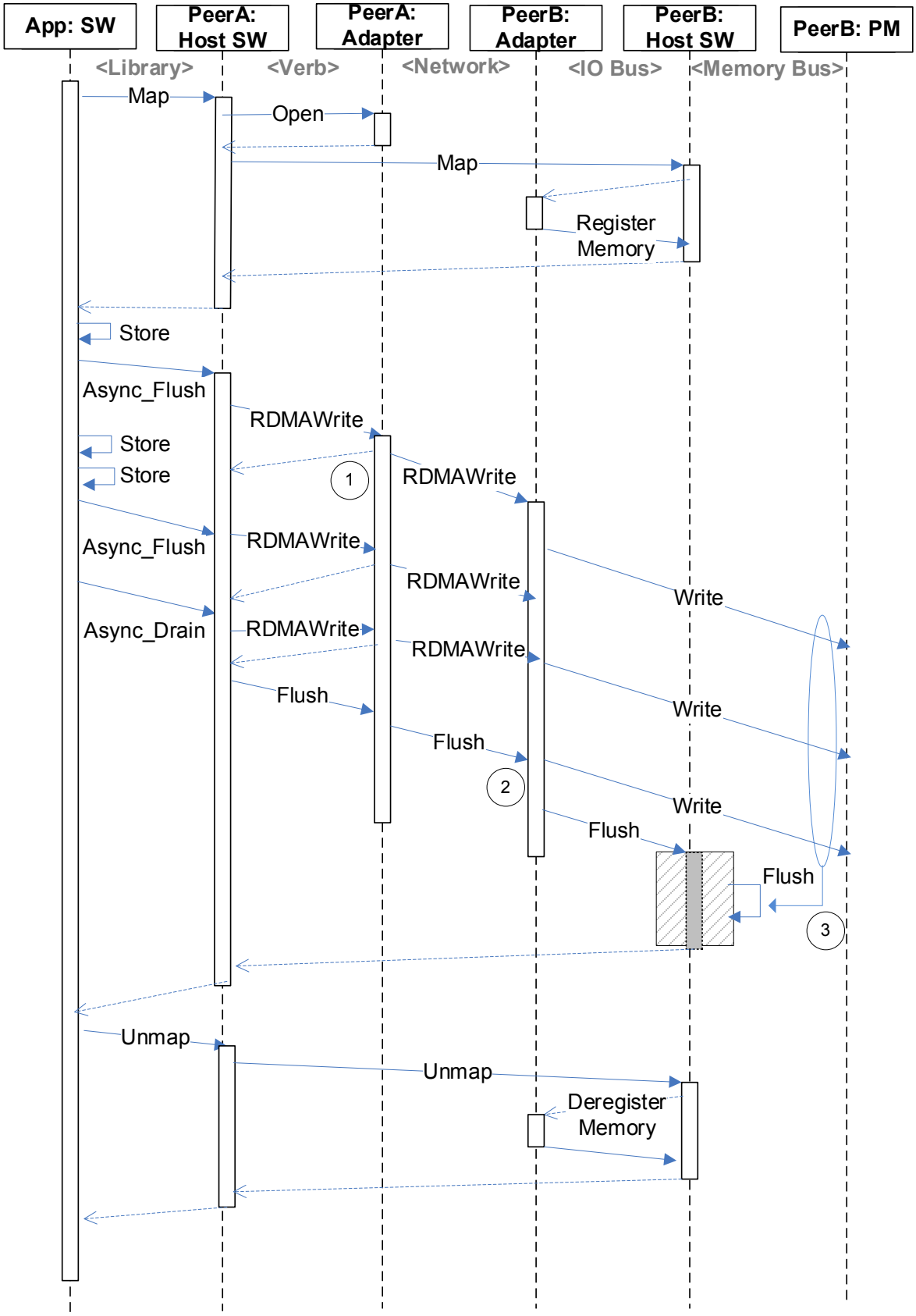

Figure 10 - Peer to Peer HA Replication using client initiated RPMA

This flow illustrates interaction between 6 actors; an application, Peer A, Peer B, the Adapters in peers A and B and the persistent memory in peer B. The annotations at the top of the diagram provide additional context for the interaction between the actors represented by adjacent columns. Communication between the first two columns generally takes the form of software library calls. The actions that result from these calls generally flow to adapters in the form of verbs. The adapters communicate with each other over the network, and with host CPU's over an IO bus such as PCIe. Finally, hosts use memory busses to communicate with memory. IO bus and memory bus interactions are called out specifically on Peer B to illustrate tradeoffs in that area that are described below.

The context provided by these annotations is necessary to differentiate the interpretation of terms such as "RDMAWrite" and "Flush" which, although they recur across columns in the diagram, have slightly different meanings based on the context. For example, "Flush" takes the form of a verb (ala InfiniBand) when it flows from a host to an adapter, but it takes the form of a network protocol when communicated between adapters. Similar distinctions occur in each context.

Some implementation specific session initialization must occur prior to the activity shown in this diagram. This includes initializing and opening adapters, creating queues, authenticating the Peers and querying for attributes. At the start of the diagram, Peer A opens an RDMA session with Peer B while memory mapping a file. It establishes a remote address mapping and registers it with the adapter. Within this session, application addresses on Peer A are used to access persistent memory devices on Peer B in a way that aligns with file system metadata. This is illustrated in detail by Figure 9.

The application then uses CPU instructions to store data to a number of possibly discontiguous memory ranges on Peer A. This is illustrated on the application thread right after the Map. In this example the application uses the Async_Flush action after the first store to trigger a copy to PeerB. The application may use Async_Flush multiple times. After all of the necessary stores have occurred the application uses the Async_Drain action to ensure that the stores are persistent on PeerB. Async_Flush actions causes the remaining Writes, if any, to be transmitted from Peer A to Adapter A. As represented near circled numeral 1, Host A can get a completion notification for each Write however this may not indicate that the data has progressed beyond Adapter A. This is analogous to the semantics of a local store within a CPU.

After Async_Flush Adapter A transmits data to adapter B which uses Peer B's IO bus (i.e. PCIe) to deliver data to PM on Peer B. It is up to Adapter A to determine how many Write transmissions occur between itself and RNIC B. Since no acknowledgement is required in that exchange this decision has miniscule effect on latency. As per section 6.3, at this point there is no guarantee that data has actually reached PM. Figure 10 illustrates this with the asynchronous write process in which writes reach the PM actor at unknown times after they are received by Adapter B.

RDMA implementations are required to ensure that the signal indicting receipt of a SEND (and certain other verbs) cannot be generated by Adapter B until all of the writes that precede it have been delivered by Adapter B. Therefore an upper layer can

implement the flush operation using a SEND at circled numeral 2 which is processed by the software in Peer B at circled numeral 3. The flush is required to insure that all of the writes that preceded the flush are in PM before Adapter B responds back to Adapter A indicating completion of the flush. The gray box near item 3 indicates that multiple flush implementations are possible that also reflect Async_Drain semantics. This may involve remote host software, remote host hardware, or other approaches such as a flush protocol operation on the wire that may involve extensions to existing protocols.

By this means writes and flushes are orchestrated in such a way that the net effect of the original Async_Flush and Async_Drain actions is the same on Peer A and Peer B, namely that all of the data referenced by the Async_Drain has reached PM before the completion of the Async_Drain. Used correctly by applications, this is sufficient to enable crash consistency with RPO=0 (relative to Async_Drain actions) in backtracking recovery scenarios as described in section 4.7.2. As described in section 4.8, this entire sequence can be implemented using Optimized_Flush instead of Async_Drain, in which cases the Async_Flush actions would not appear.

When the application is finished modifying the memory mapped file it cleans up by deregistering and closing the RDMA session.

Like local memory access, this scenario does not require that all Write's reach the PM in Peer B in the same order that they did in Peer A, as long as the memory state on both peers adheres to the definition of Optimized_Flush and Async_Drain. For RPO=0, Optimized_Flush and Async_Drain actions are executed in the same order on both Peer A and Peer B. Ordering constraints for RPO > 0 are implementation dependent so long as reordering does not corrupt a consistency point that may become visible to Peer A during recovery. Remember also that Optimized_Flush and Async_Drain do not make atomicity guarantees. This means that remote PM must account for the local atomicity that originates with the local CPU.

Figure 11 illustrates the use of redundancy on Peer B to recover from an unrecoverable ECC error on Peer A.

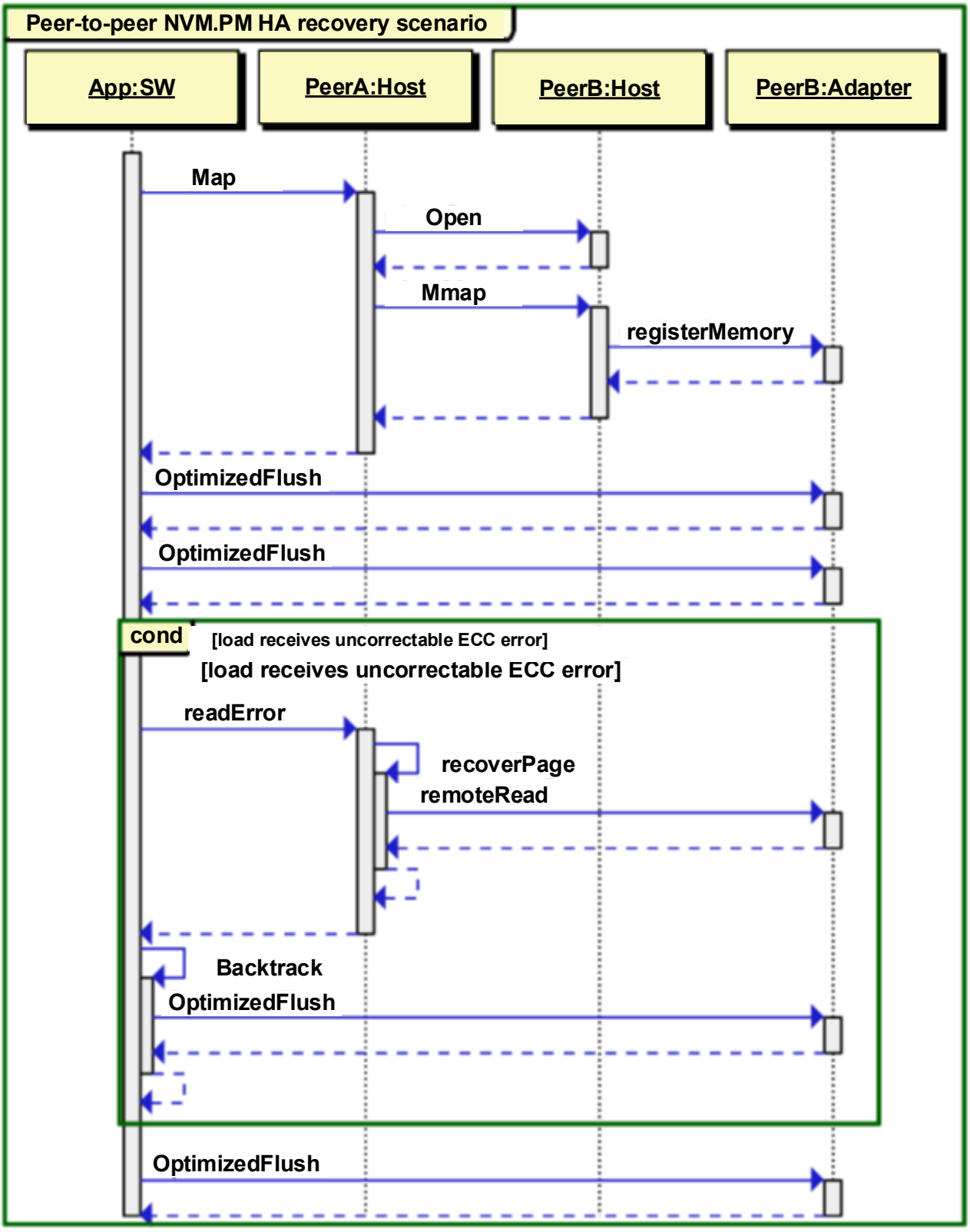

Figure 11 - Uncorrectable Error Recovery

For graphic simplicity the PM thread of this figure has been removed. It participates in OptimizedFlush (used here instead of Async_Drain) as described in Figure 10. This scenario proceeds as before until the error occurs, represented by the box labeled “load receives uncorrectable ECC error”. At that point the application is shown encountering a “read error” which represents an exception that is fielded by the file system. The resiliency function described in section 5 does a remote read to recover the lost data. Since the data is only as recent as the last time it was referenced by an Optimized_Flush, backtracking (such as a transaction abort) may be required on the application’s exception handling thread as described in section 4.7.2. Any aborts may require additional Write and Flush actions prior to the completion of the exception handling, after which the application resumes normal operation. As in Figure 10 the application eventually ends the RPMA session (not shown).

6.5 HA across multiple processor architectures

The use of RPMA for direct data transfer to PM in a remote node does not address any potential architectural incompatibilities between local and remote nodes. For example, with RDMA specifically the application is responsible for addressing data representation differences such as endian-ness or floating point number encoding. If remote access for HA is attempted across divergent processor architectures then portable data structures are required, especially in the event of failover from one processor architecture to another.

A similar issue arises with respect to atomicity of fundamental data-types (NVM Programming Model Version 1.2 section 10.1.1 – “Applications and PM Consistency”). It is common for PM optimized data structures to depend on atomic updates to fundamental data types such as integers and pointers. Such dependencies may not be conveyed across RPMA operations due to processor architecture differences or packetization of data within or below the transport layer of the network protocol stack.

Since there are no common specifications of failure atomicity related to either RPMA or processor architectures there is no way to guarantee correct handling of atomicity short of detailed end to end review of the component implementations involved in a given deployment. Some existing protocols include atomic operations however these do not address persistence. In the absence of a failure atomic store as a primitive for remote fundamental data type operations forces applications to fall back to checksum based atomicity.

At a minimum these considerations create a requirement that the architectural similarity of two nodes in an HA relationship be ascertainable by management software. This should provide a warning in conditions where access to data structures after failover may be in doubt. In addition, any applicable atomicity granularity attributes should account for remote atomicity. Finally, restrictions on component replacements or VM relocations that cross processor architecture boundaries may also apply.

Additional exploration of potential failure atomicity considerations appears in Appendix C.

7 Error Handling

There are numerous sources of errors in the processes described in section 6. Rather than attempting an exhaustive enumeration of these, this section describes a systematic approach to error detection, recovery and reporting in the context of Figure 12 below. In general, error recovery should occur at the lowest level possible. If recovery is not possible at a given layer, the error should be propagated upward to the next highest layer in the stack. With this in mind, error handling processes or actions can be categorized as follows.

- Detection – some piece of hardware or software detects an error.
- Local Recovery – the portions of the system affected by the error take action that allows them to continue operation, if possible, in spite of the error.
- Global Recovery – software at some level in the system ensures that the entire system has responded to the error in such a way that the system can continue operation without risk of data corruption or inconsistency. This may involve parts of the system that were not initially involved in error detection or local recovery.
- Reporting – software logs the error. Each error detected should be reported at the lowest layer possible. Error information should be propagated upward only to enable recovery. Recovery attempted and its success or failure should be reported by the layer that initiated recovery.

Figure 12 shows 3 layers of error handling operating across 2 nodes. Many implementations may involve more nodes or nested recovery in which complex recovery processes are embedded within a single layer. Even so, this model is useful to illustrate how different types of recovery are layered.

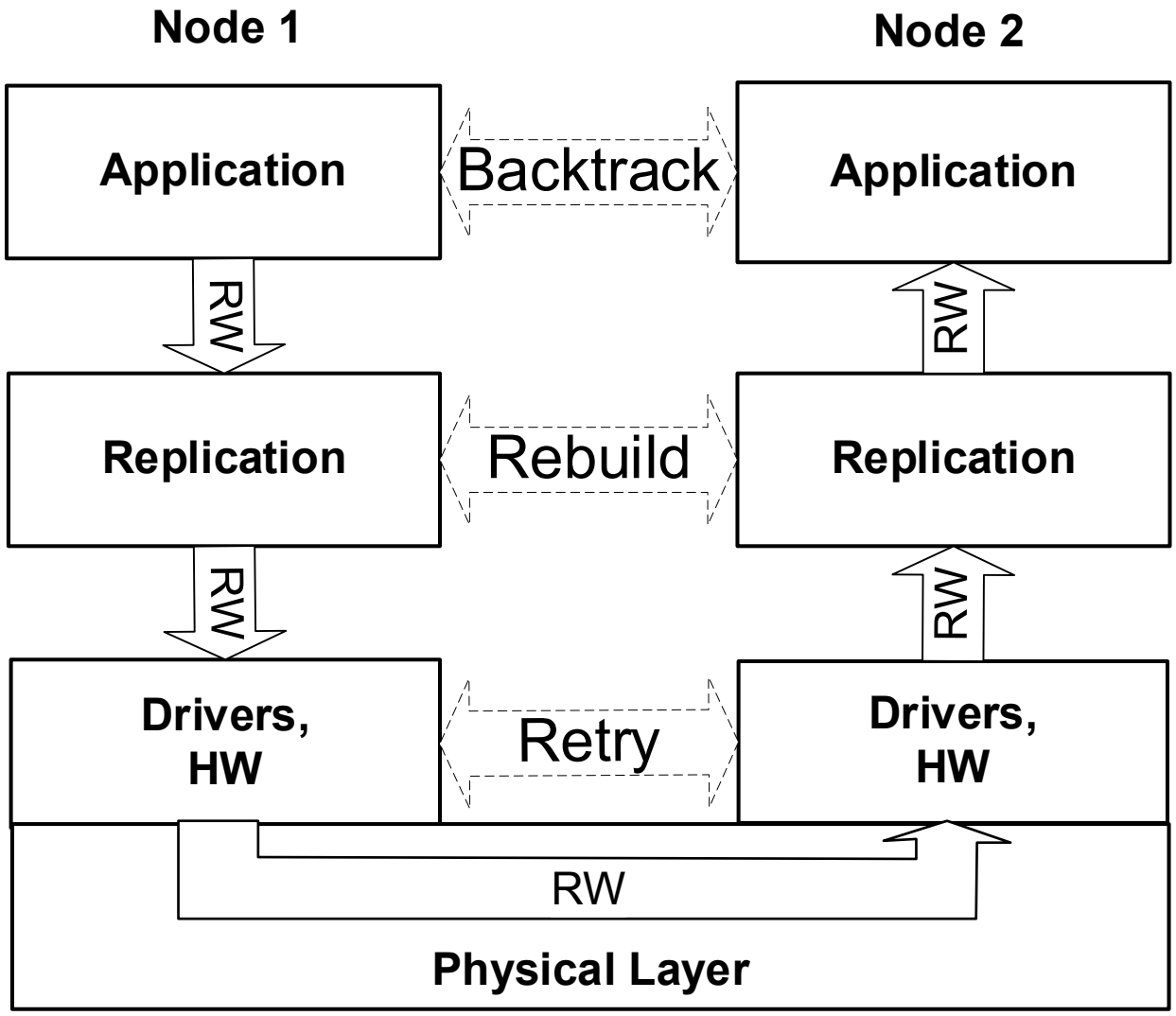

Figure 12 – Error Handling Layers

Here we see Application, Replication and Driver/HW layers stacked on 2 separate nodes. Although most layers may be bypassed in real time, the data path conceptually flows through all of the layers that may become involved in recovery. The arrows across the middle represent categories of recovery action that may be coordinated across networks between corresponding layers. They are dashed because they represent virtual communication that is actually tunneled through lower layers. They are bidirectional because commands flow to the right while status and error information flows to the left. The driver, hardware and physical layers are conjoined to illustrate that the placement of low level recovery actions depends on implementation.

In the software layering of Figure 12, most of these actions are performed at one of three layers in the system:

- Hardware and low level software such as drivers – NICs, PM devices or processors detect and possibly locally recover from errors. For soft errors, hardware may take all of the necessary action to globally recover from the error

without involving software beyond the associated drivers. Network fault tolerance techniques such as multi-pathing, forward error correction, etc. are grouped in this category. Note that local persistent memory error handing is not addressed here as it is covered in the Error Handling content of the NVM Programming Model specification.

- Storage Resiliency – Resiliency is implemented as a replication layer using techniques such as RAID or erasure coding. The replication layer is seldom the first to detect errors but it is often the locus of global recovery. In this case the replication layer is a user mode library as shown in Figure 7.
- Application – In some cases the application must respond to errors, especially if backtracking is involved. For this purpose, transaction functionality is considered to be part the application.

These layers are listed above from lowest to highest levels in an escalation hierarchy. Each layer performs best effort recovery within its scope. If that recovery completely resolves the error and no other recovery action is needed then that layer has achieved global recovery and higher layers are not involved. Otherwise the layer that detected the error performs whatever local recovery it can, and escalates the failure to the next layer up, where the process repeats. The application layer is the last resort for global recovery. Failure of global recovery at the application layer renders the system at least partially inoperable pending manual intervention.

For example, at the lowest layer shown in Figure 12 error recovery generally involves some combination of low-level error correction and operation retry. This is illustrated by the “Retry” arrow across the driver/HW layer of Figure 12. Initial error detection may occur at either end of a network connecting multiple nodes. Although the division of recovery responsibility is implementation specific the greater burden generally falls on the initiating end (Node 1 in this case) where the need to execute a read or write originated.

If the driver/HW layer cannot fully recover, the resiliency layer may attempt recovery from redundant data using such processes as the rebuilding of data in a RAID implementation. If the resiliency layer fails, or if there is question regarding the consistency of recovered data, the error is escalated to the application layer where processes such as transaction recovery or restoration from an earlier backup may occur. These are illustrated by the “Backtrack” arrow because they often involve some loss of work. Generally, failure of the resiliency or application layers is detected at the initiating node (Node 1 in this case) although exceptions may arise due to background processing such as data scrubbing or consistency checking ala Section 4.9.

7.1 Hardware

Networking hardware, drivers and/or protocol stack are expected to detect, report and locally recover from the following types of errors:

- Loss of network access
- Loss of remote server power
- Transient network errors – network is expected to achieve global recovery

- Unrecoverable transmission errors – For global recovery at the replication or application levels this is expected to be converted into a data loss or a loss of network access depending on the pervasiveness and type of errors.

7.2 Replication

The replication layer is expected to report and locally recover from the following types of errors. Additional expectations are listed case by case. Local recovery without detection is triggered by error reporting from hardware layers:

- Loss of network access – The application may proceed without redundancy. The replication layer may need to do local recovery. The replication layer is expected to report the failure and achieve global recovery by resynchronization local and remote data after network access is re-established so that both represent the same consistency point(s) as defined in section 4.
- Loss of remote server power – The replication layer is expected to detect and locally recover. The replication layer may also detect the error. In addition, the file system layer is expected to report the error and achieve global recovery as with loss of network access.
- Remote server or file service reset – The replication layer responsibilities are the same as with loss of remote server power, assuming no lapse in network accessibility.
- Loss of local data – The replication layer reports and locally recovers from this type of error. If global recovery can be achieved without backtracking then it may be accomplished by the file system layer. Otherwise the application layer must participate.
- Loss of local data with additional error such as loss of remote data or remote server access – Since this case involves multiple failures the replication layer may be unable to achieve global recovery. This can only be achieved at the application layer.
- Data corruption – The replication layer may need to participate in local recovery.

7.3 Application

The hardware and replication layers make every effort to detect and recover from errors without application assistance, however in backtracking and/or data loss scenarios the application layer must participate as follows. The application layer reports its involvement in any of these scenarios:

- Loss of remote server access – This represents a group of error conditions in which the replication layer orchestrates global recovery using backtracking to a prior consistency point. The application may need to participate in backtracking by, for example, aborting and/or retrying transactions.
- Loss of local data – The application or local PMFS detects this error, reports it, and may participate in global recovery.
- Loss of local data with additional error or loss of both local and remote data – the application must orchestrate global recovery by restoring data from backup (outside of the replication layer) and restarting.

- Data corruption – the application must detect this error and orchestrate global recovery. This may involve rolling back through backups until one is discovered that does not have the corruption.

8 Requirements Summary

This document describes some specific examples in terms of RDMA, but RDMA per se is not the only way to address these requirements, hence the pervasive references to RPMA throughout the document. In addition, the description in section 6 includes some implementation specific details in order to concisely communicate a desired outcome.

This requirements summary adds to the behaviors common to RDMA transports. In the interest of clarity each of the following items is framed as a general requirement with implementation specific examples to further illustrate the nature of the requirement:

- Assurance of durability
  This requirement motivates some protocol to force data into PM at the RDMA data sink (i.e. the remote peer in Figure 8) including confirmation of same back to the application. This could involve additional flow between client and server or it could be built into the transport as a latency reduction.
- Efficient byte range transfers
  This requirement represents a strong desire to reduce the latency of HA for Load/Store workloads to a much larger degree than can be achieved with today's RDMA implementations. For example, load/store access tends to create sets of small byte ranges that can be packaged in one RDMA and piggybacked with remote flushing to persistent memory. This requirement also motivates a kind of scatter gather RPMA that operates at both the application and the remote access server as shown in Figure 9. Of course byte range transfer optimization should not come at the expense of large transfer optimization. It should be reasonable to assume that the transport can self- optimize based on the expression of byte ranges in the application's call to optimized flush.
- Order Nexus
  Support an order nexus abstraction sufficient to meet the requirements of NVM.PM.ASYNC_DRAIN in coordination with the release of that feature in the NVM Programming Model specification.
- Discoverability of architectural incompatibilities
  Gaps in the ability to fail over to another node and recover data on that node should be discoverable. One known type of gap has to

do with data representation. This is described in more detail in section 6.5. As described there, the architectural similarity of two nodes in an HA relationship should be ascertainable by management software. Configuration software should provide at least a warning in conditions where access to data structures after failover may be in doubt.

- Atomicity of fundamental data types
  PM related software may depend on atomicity (all or nothing updates) of fundamental data types (integers and pointers) with respect to power failure. When available, this is a feature of processors. If this requirement cannot be met by a given network and processor implementation, then application level CRC will be required. Section 6.5 outlines the issues and alternatives related to remote failure atomicity. Some option validation, experimentation and most likely new implementation is needed. In any case the NVM Programming model specification should include atomicity granularity and/or other attributes that account for remote atomicity.
- Isolation/HW fencing after failure
  Correct failure recovery generally assumes fail stop behavior of failing nodes before remaining nodes resume activity. Fail stop means that a failing component cannot come back to life or otherwise behave in a way that disrupts the rest of the system during or after recovery. Fail stop behavior must include scenarios that involve concurrent power loss and hardware failure. Failing components are required to be isolated from the rest of the system even in those scenarios.
- Error handling
  Error recovery should occur at the lowest level of an implementation that is capable of recovery. When a layer cannot recover, errors should be reported to upper levels (e.g. replication or application in Figure 12) to enable data recovery techniques described in section 4.7, up to and including application level backtracking.

Several ongoing areas of requirement investigation include the following.

- Security – See the SNIA PM Hardware Threat Model White Paper
- Resource management – There is a general notion that buffer management can be simplified relative to the use of RDMA, for example, in today's non-PM file systems. In PM systems, remote access flows directly to and from persistent memory that is permanently allocated (or semi-permanently allocated for the duration of a memory mapping) for the purpose of mirroring specific client data. This is expected to eliminate buffer resource management considerations., thus potentially enabling the elimination of a network round trip in HA solutions for PM.

Appendix A – HA Protocol Flow Alternatives

As shown in Figure 10, since the PCIe bus doesn't have a persistence barrier transaction, and the memory systems on modern systems use multiple distributed memory controllers, the ordering of writes to the persistence domain is indeterminate if the writes end up on more than one memory controller (also assuming DDIO is disabled). It is not clear that systems will be able to implement this optimization any time soon so a CPU will have to be involved with the flush.

This leaves an open question that is subject to experimentation and analysis. Given that a CPU has to be involved for the flush, would it be just as well for it to parse the packet and place the data too? If so then perhaps a straight message-based protocol would be as good as, or better than an RDMA-based protocol. By expressing requirements in terms of an abstract networking protocol this document enables RDMA and bus protocols to evolve.

Appendix B – Remote Atomicity Considerations

Additional effort is needed to evaluate approaches to remote failure atomicity. This appendix contains some information that could form a framework for such investigation.

Since the desired atomicity property occurs when data is written to PM it must involve the implementation of the sink RNIC (i.e. RNIC B in Figure 10) and the way data is flushed. Given implementation of a failure atomic flush between an RNIC and PM, the RNIC can apply this primitive in several ways. The following table suggests a ranking of several options relative to each other (i.e. lower numbers are better) based on the following criteria:

- Overhead – how much additional latency occurs when failure atomicity is applied
- Selectiveness – to what degree can the overhead be applied only when needed
- Compatibility – how intrusive is the potential protocol impact of the option

Option	Over-head	Selective-ness	RDMA Compat-ibility	NVMP Compat-ibility
A - Apply to atomic actions surfaced by existing protocols	1	1	1	3
B - Apply to all RDMA writes	2	3	1	1
C - Apply to all RDMA writes in a given session based on a registration option	2	2	2	1
D - Apply to individual RDMA writes based on a flag in each RDMA write	1	1	2	3
E - Use checksum when atomicity is required	3	2	1	2

At this point there is no quantitative data on relative overheads of these options so it is difficult to draw conclusions from such a ranking. Any of these options other than CRC requires a failure atomic sink RNIC write implementation (i.e. at RNIC B in Figure 10). Options A and D may have “convoy” alternatives in which multiple atomic writes are

communicated at once. Various consistency and alignment considerations may come into play within each of these options.

Appendix C – References

"Memory consistency and event ordering in scalable shared-memory multiprocessors," Gharachorloo et al, ISCA, 1990, pp. 15–26

"NVM Programming Model" created by the SNIA NVMP TWG - http://www.snia.org/sites/default/files/NVMProgrammingModel_v1.pdf

"Persistent Memory Atomics and Transactions" – https://www.snia.org/sites/default/files/technical_work/Whitepapers/PM_Atomics_and_Transactions.pdf

"PM Hardware Threat Model" - https://www.snia.org/PM-Hardware-Threat-Model-Technical-WP

iWARP (RFC 5040) - http://tools.ietf.org/html/rfc5040

InfiniBand (InfiniBand Trade Association specification) including annexes A16 and A17 regarding ROCE and ROCE2 - http://www.infinibandta.org/content/pages.php?pg=technology_public_specification

IOZONE - http://iozone.org/

Appendix D – Glossary

NVM – Non-Volatile Memory – In the context of the SNIA NVM Programming TWG, NVM refers to all types of storage including storage class memory, persistent memory, SSD's and rotating media disk drives.

PM – Persistent Memory – In the context of the SNIA NVM Programming TWG, PM refers to durable media that operates at memory speed and enables byte or cache line access.

RDMA – Remote Direct Memory Access – A means of directly accessing memory in a remote location over a network. RDMA is a key feature of InfiniBand interface, among others.

RNIC – RDMA NIC – A Network Interface Card that supports Remote Direct Memory Access

RPO – Recovery Point Objective – A metric specifying the amount of work that might be lost in the event of a failure.