Storage Networking Industry Association
Technical White Paper

NVM PM Remote Access for High Availability

Version 1.0

Feb 22, 2016

ABSTRACT: This paper explores the requirements that High Availability extensions to the NVM.PM.FILE mode of the SNIA NVM Programming Model might place on high speed networking, such as RDMA.

USAGE

The SNIA hereby grants permission for individuals to use this document for personal use only, and for corporations and other business entities to use this document for internal use only (including internal copying, distribution, and display) provided that:

1. Any text, diagram, chart, table or definition reproduced shall be reproduced in its entirety with no alteration, and,

2. Any document, printed or electronic, in which material from this document (or any portion hereof) is reproduced shall acknowledge the SNIA copyright on that material, and shall credit the SNIA for granting permission for its reuse.

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 white paper, 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 tcmd@snia.org.

Revision History

Revision	Date	Sections	Originator:	Comments
V1.0	Feb 22, 2016		D. Voigt	Initial Publication

Contents

1 Purpose

The purpose of this document is to establish the context and requirements for the use of RDMA as a transport for remote access to persistent memory (PM) in high availability implementations of the SNIA NVM Programming model.

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. While there are many ways to implement that communication it is thought that Remote Direct Memory Access (RDMA) may be the transport of choice.

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. In this context a CPU complex comprises the CPU, memory and support chips for a single or multi-socket server.

In referencing RDMA this document is not referring to any particular RDMA implementation. The intent is to enable a range of implementations while describing characteristics of RDMA that could reduce the overhead of correct HA operation.

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 addresses requirements 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.

This document describes security measures applicable to RDMA, such as techniques to encrypt data in flight and implementation guidelines to reduce exposure to attacks. It does not address security of data at rest (i.e., encryption on the storage media), as that is independent of the RDMA transport and is determined by the storage device model implemented.

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 section 3.4. This is because a loosely coupled server environment using high speed networking 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.

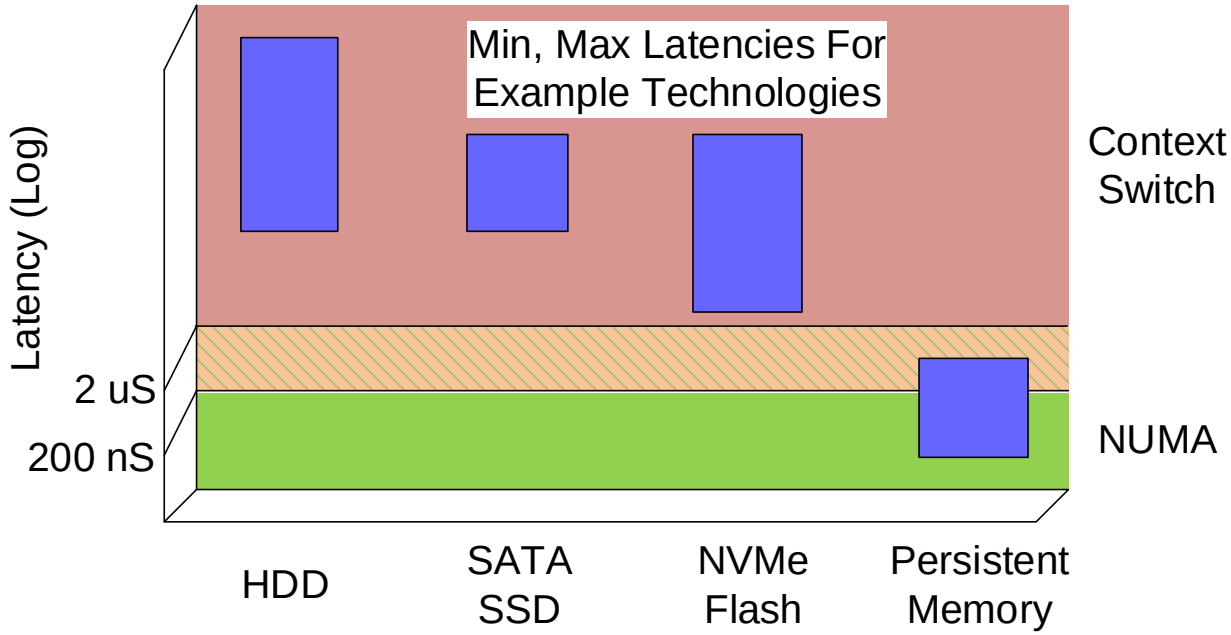

Figure 1 – Storage Latency Ranges Impact Software

Local or disaggregated persistent memory (see sections 1.1 and 3.3) can fall into the NUMA range of Figure 5. Networked persistent memory and virtual shared memory (sections 3.4 and 3.5) do 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.

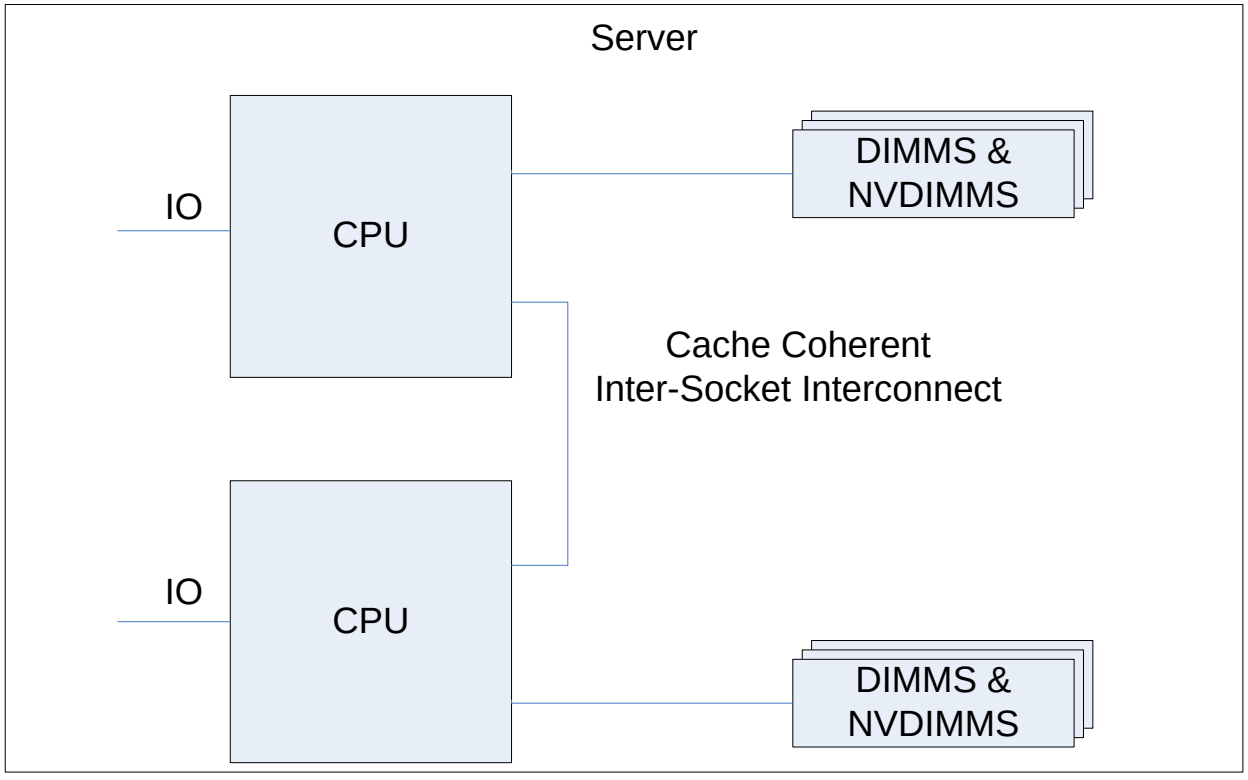

Figure 2 - Local Memory

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. Disaggregated memory is not necessarily 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 is subject to distance limitations to insure sufficiently low latency. Disaggregated memory is made feasible 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. Since disaggregated memory is not necessarily cache coherent, distributed programming techniques such as those used in clusters must be applied rather than the symmetric multi-processing techniques that apply within a single server.

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

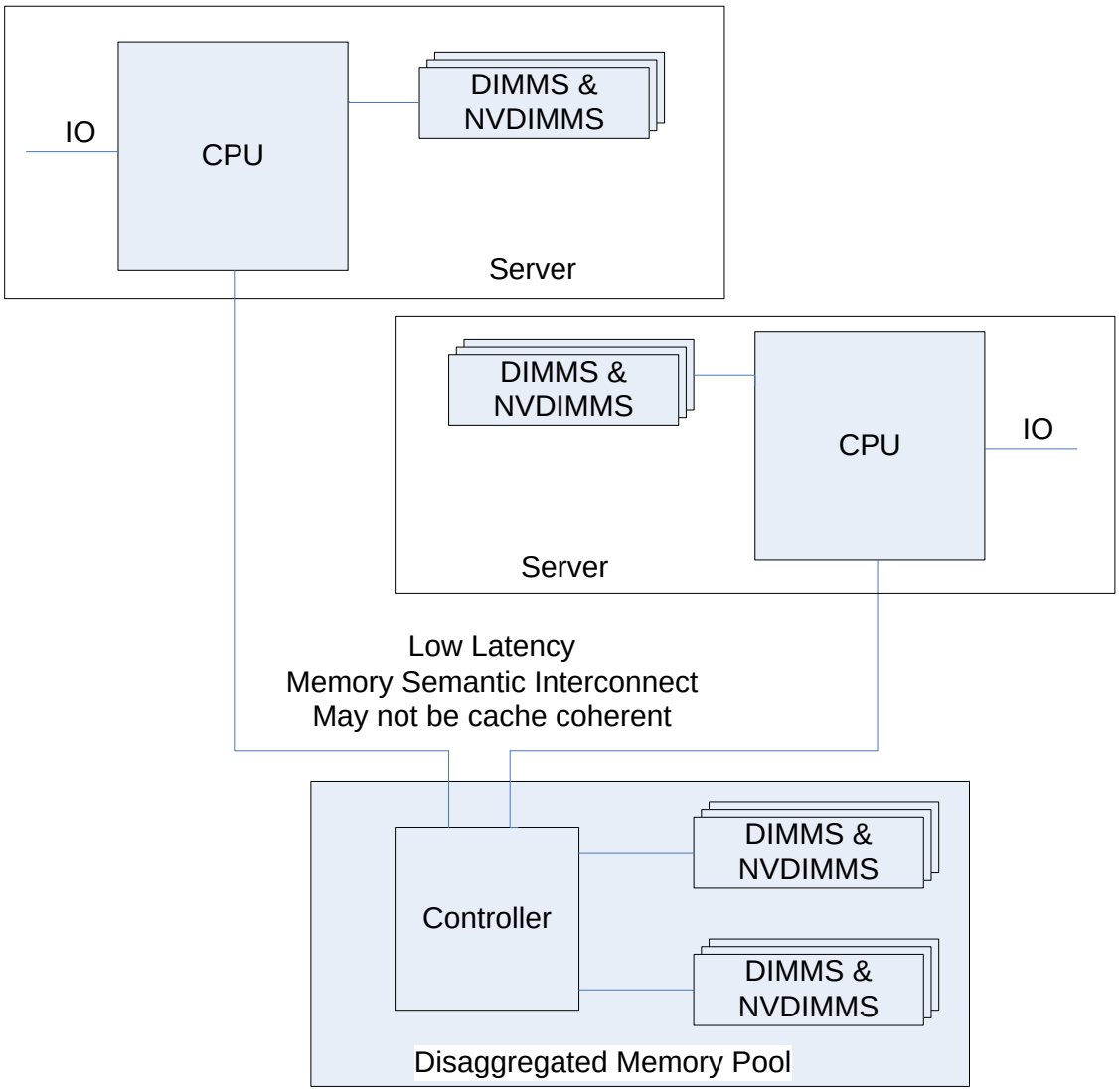

Figure 3 - Disaggregated Memory

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 either message passing or 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.

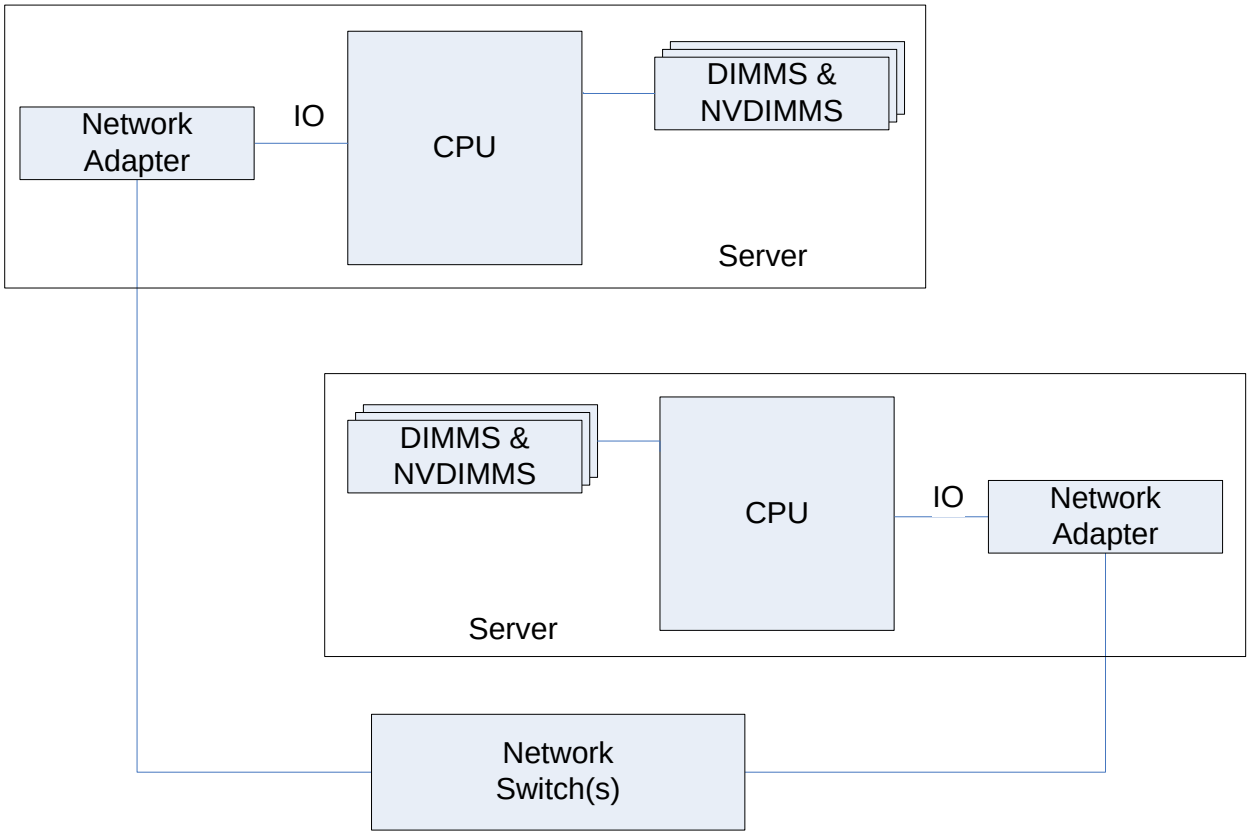

Figure 4 - Remote Memory

3.5 Virtual Shared Memory

Virtual shared memory is a means of emulating cache coherent shared memory across networked memory using software as shown in Figure 5. A uniform view of memory is presented to a number of servers, often using their processors memory management units. The servers are configured such that each write to a shared memory page causes a page fault which in turn causes invalidation of data on the other servers. Reads of invalidated data on one server also cause page faults so as to transfer that data over the network from a server with an up to date copy. A number of optimizations can be added to this basic principle to create the illusion of a symmetric multi-processing environment across the servers.

This approach to sharing is not favored for persistent memory because it adds considerable software overhead to many workloads.

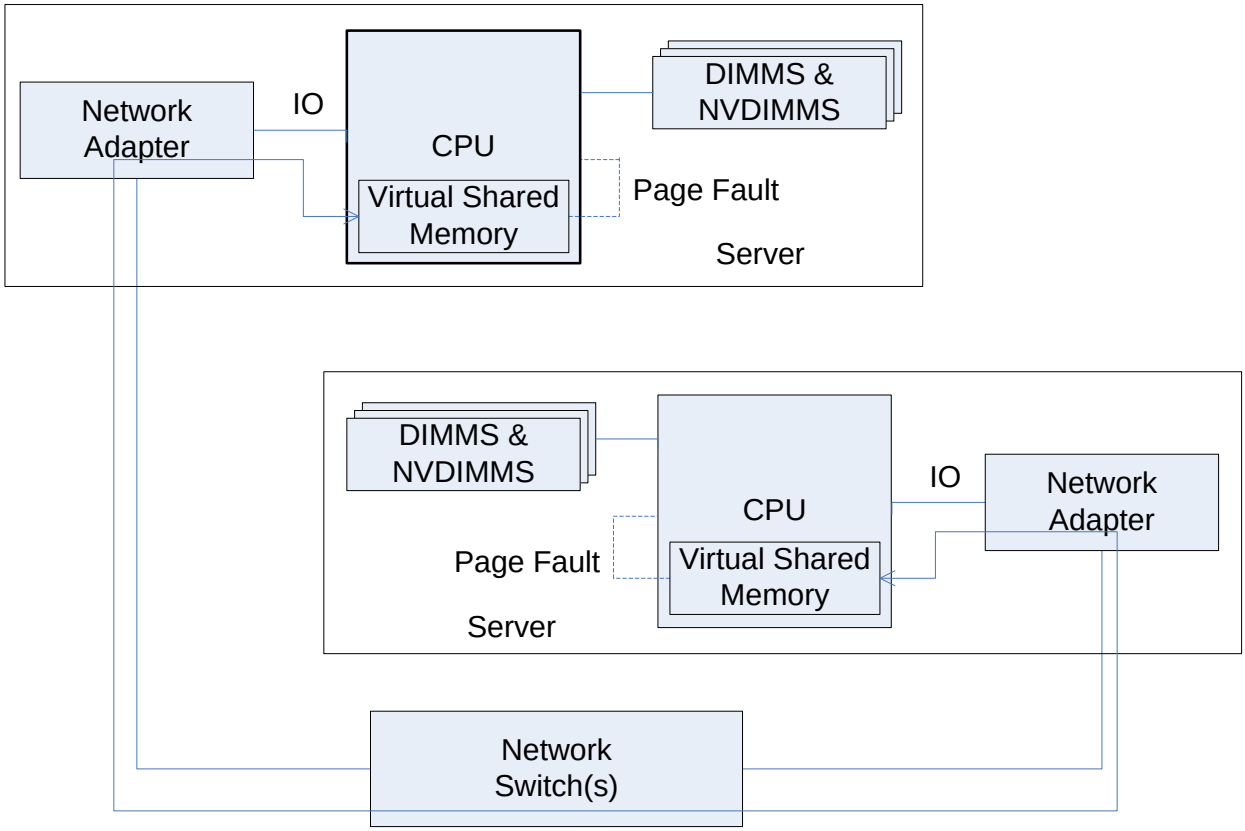

Figure 5 - Virtual Shared Memory

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 is considerable precedent 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 6 wherein the local and remote memory of Figure 4 are overlayed with red lines indicating the data flow of a store (ST) operation.

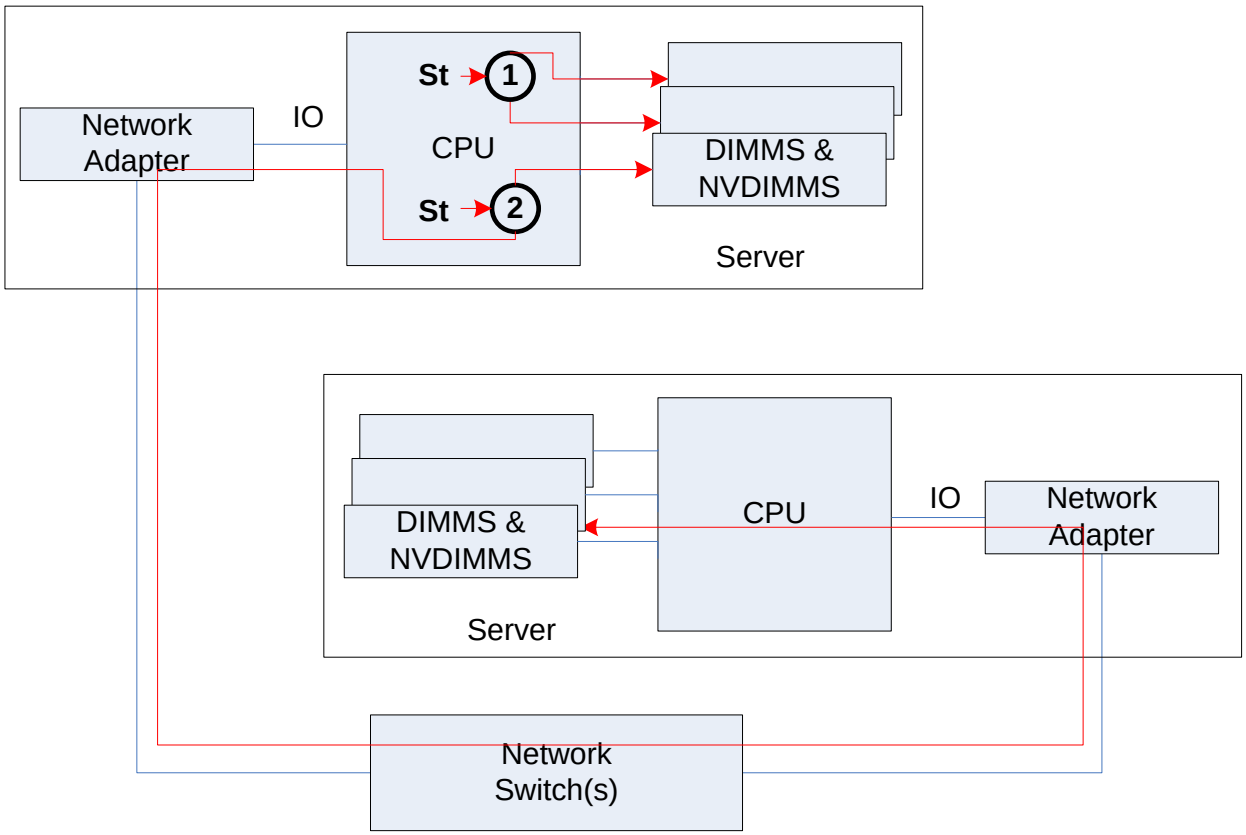

Figure 6 - High Durability vs High Availability

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 6 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 6 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 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 if a hardware failure occurs while Fred is transferring money from his account to Barney's, recovery from the failure cannot result in a state where both Fred and Barney have the money.

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 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 high availability is that they create opportunities to optimize networked persistent memory communication.

4.3 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.

4.4 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 V1 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.

4.5 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 4.3.

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.

4.6 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.

4.6.1 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 4.6.2).

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.

4.6.2 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.

4.6.3 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.

4.6.4 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 4.5.3 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.

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 RDMA (or similar 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 RDMA.
- User space optimization to access remote networked memory

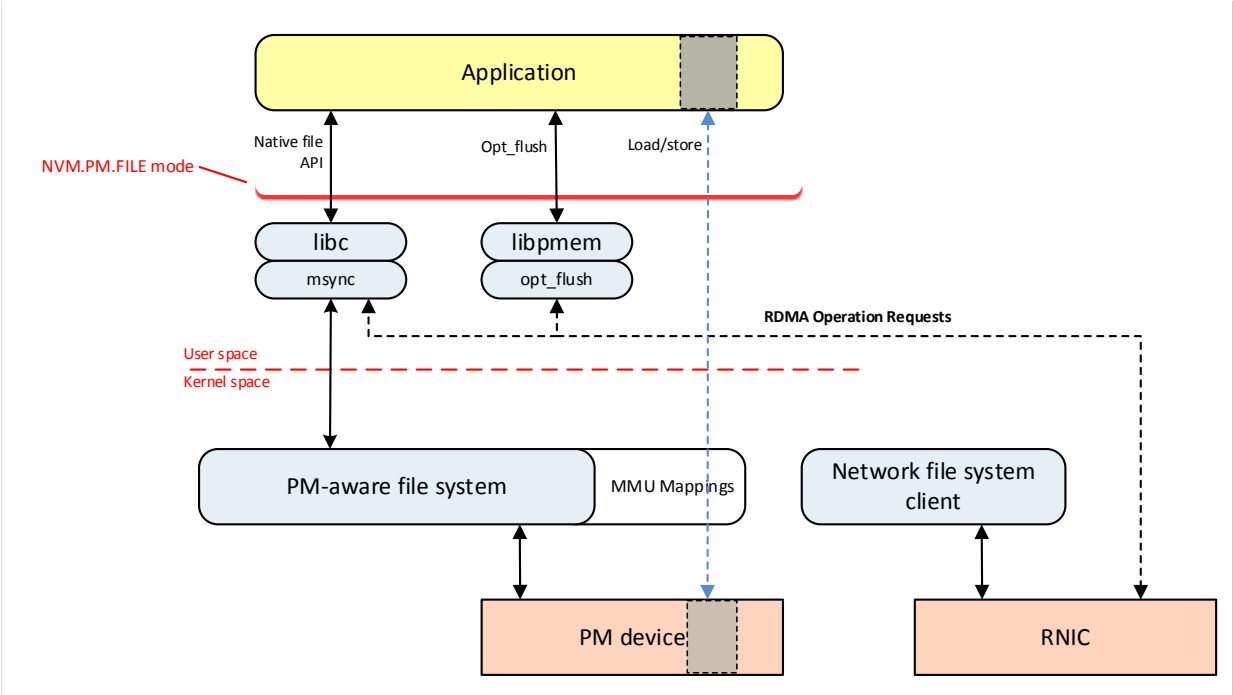

Figure 7 – HA Extension to NVM.PM.FILE

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 RNIC illustrated to the right of the PM-aware file system and the PM device.

The user space library is capable of setting up an RDMA session with the remote file system. The RDMA 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 RDMA 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 RDMA. The optimized flush and native API paths may use each other's implementations should it be advantageous to do so.

6 RDMA for HA

This section provides additional detail on RDMA 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.

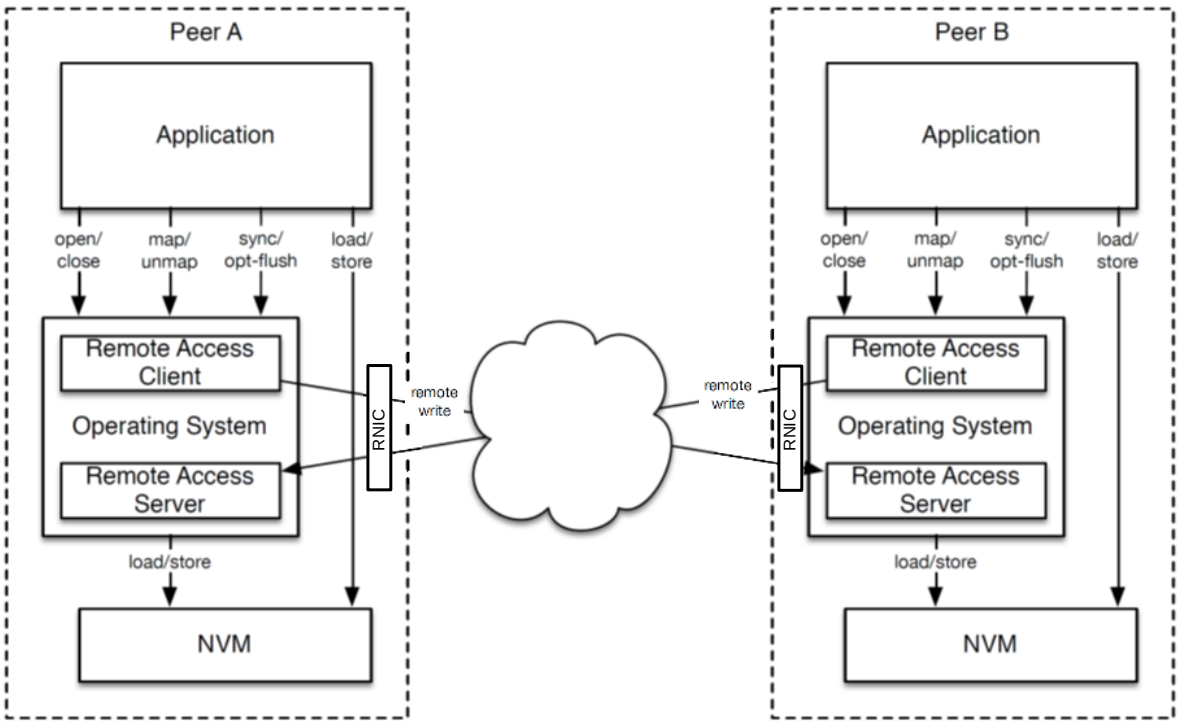

Figure 8 – NVM.PM Peer to Peer HA Replication Deployment Diagram

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

6.2 Address Spaces

The use of RDMA with memory mapped files introduces additional address spaces which must be correlated by various elements of the system. Figure 9enumerates 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.

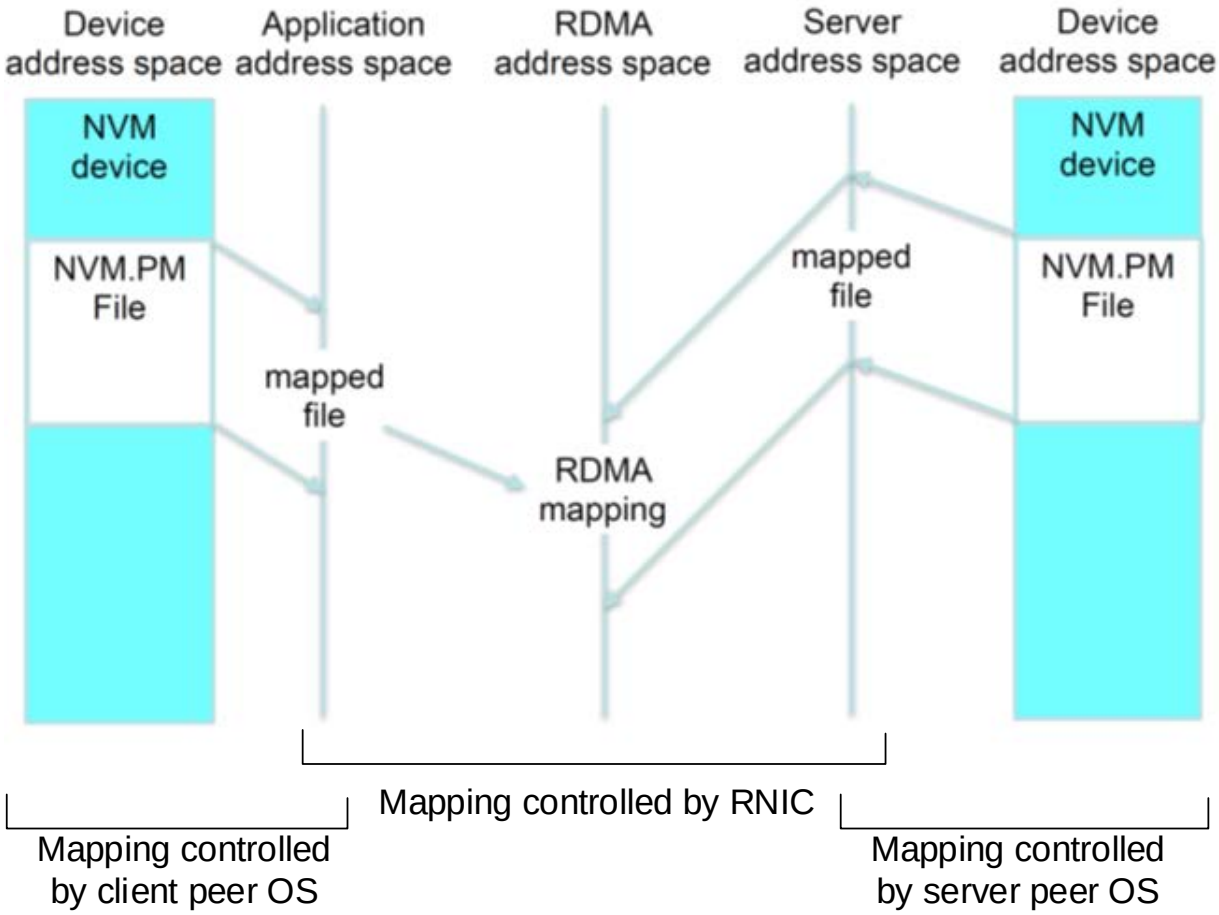

Figure 9 - Memory Mapping and RDMA Address Spaces

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.

When RDMA is initialized to establish the session for sending data to a server, an additional RDMA address space is created to rapidly and securely correlate registered memory across the RDMA NICs in Peer A and Peer B. This RDMA address space has no numerical alignment with any of the other address spaces. The mapping between the RDMA address space and the application and server address spaces is under the control of the RDMA-aware layers on each peer. The RDMA-aware layers include the user space msync and opt_flush implementations shown in Figure 7. These

implementations use RDMA 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 RDMA 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 an RDMA write is not sufficient to guarantee that data has reached persistent memory. This is because the path from a network 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 RDMA. 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 considering the low latencies of local NVM.PM.FILE access. It would be highly desirable to avoid this overhead.

6.4 Client initiated RDMA Protocol Flow

Figure 10 illustrates the interaction between the two servers and the RDMA NICs that interface them to the network as illustrated in Figure 4 and Figure 8.

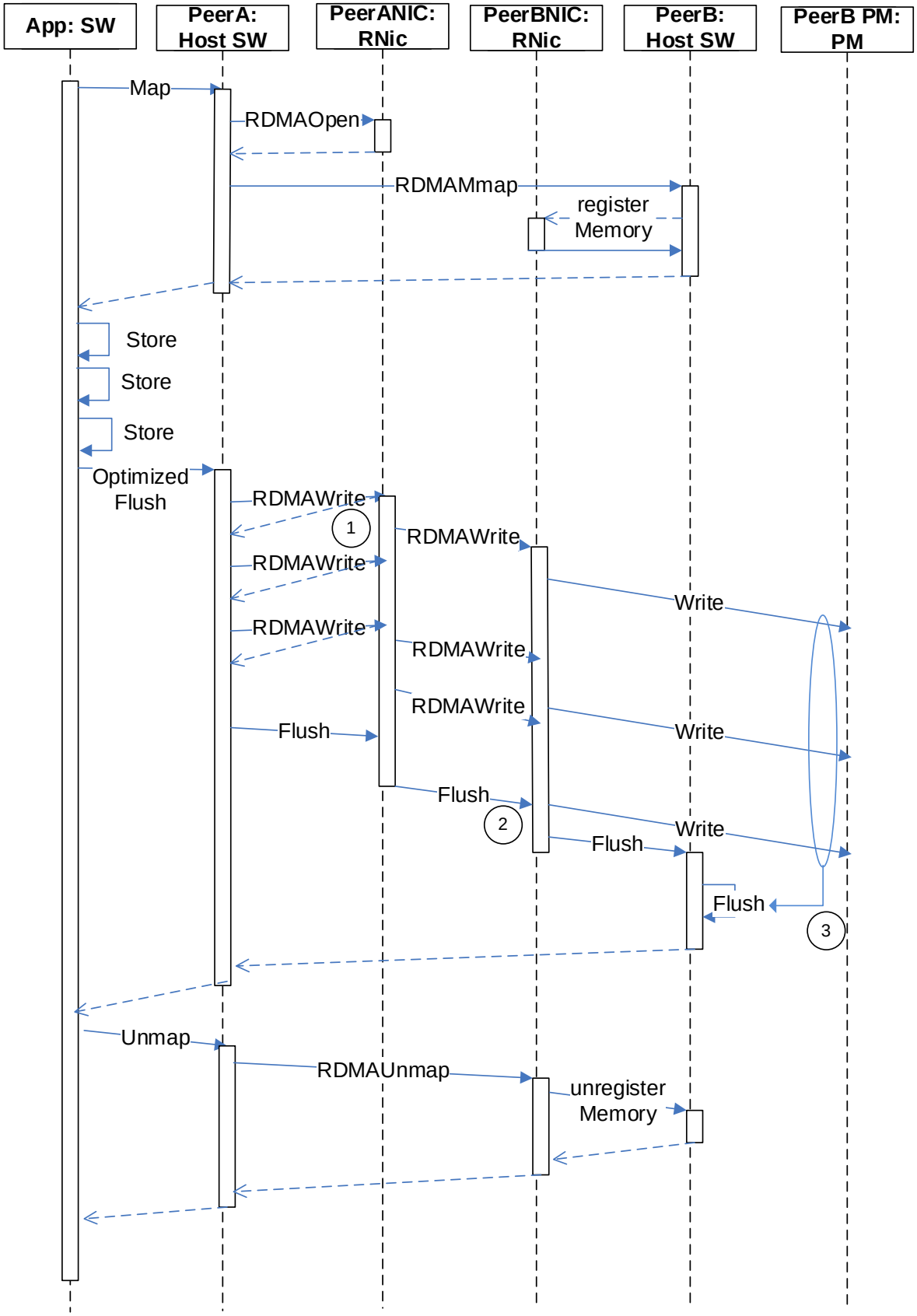

Figure 10 - Peer to Peer HA Replication using client initiated RDMA

This flow illustrates interaction between 6 actors; an application, Peer A, Peer B, the RDMA Network Interface Cards (RNICs) in peers A and B and the persistent memory in peer B. Some implementation specific RDMA session initialization must occur prior to the activity shown in this diagram. This includes initializing and opening network adapters, creating queue pairs, 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 an RDMA address mapping and registers it with the RNIC. 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. The application then uses the NVM.PM.FILE.OptimizedFlush action to insure that the stores are persistent. There may be some advantage to communicating stored data to Peer B prior to the OptimizedFlush. In the simplified example of the diagram, the OptimizedFlush action causes one or more RDMA Writes to be transmitted from Peer A to the RNIC A. As represented near circled numeral 1, Host A can get a completion notification to each RDMA write however this may not indicate that the data has progressed beyond RNIC A. This is analogous to the semantics of a local store within a CPU.

RNIC A then transmits the data to RNIC B which uses Peer B's IO bus (i.e. PCIe) to deliver data to PM on Peer B. It is up to RNIC A to determine how many RDMA 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 RNIC B.

The RDMA protocol is required to insure that the signal indicting receipt of a send cannot be generated by RNIC B until all of the writes that precede it have been delivered by RNIC 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 RNIC B responds back to RNIC A indicating completion of the flush. There are other ways of implementing flush that also reflect optimized flush semantics. One variation might involve an RDMA protocol that supports the piggybacking of additional flush semantics on the last RDMA write. Other variations might involve CPU architecture specific optimizations of the flush interaction between RNIC B, Peer B and PM.

By this means RDMA's and flushes are orchestrated in such a way that the net effect of the original OptimizedFlush is the same on Peer A and Peer B, namely that all of the data referenced in the OptimizedFlush has reached PM before the completion of the OptimizedFlush. Used correctly by applications this is sufficient to enable crash consistency with RPO=0 (relative to OptimizedFlush actions) in backtracking recovery scenarios as described in section 4.5.2.

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

Like local memory access, this scenario does not require that all RDMA'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. For RPO=0, Optimized Flush 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 that OptimizedFlush does not itself 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.

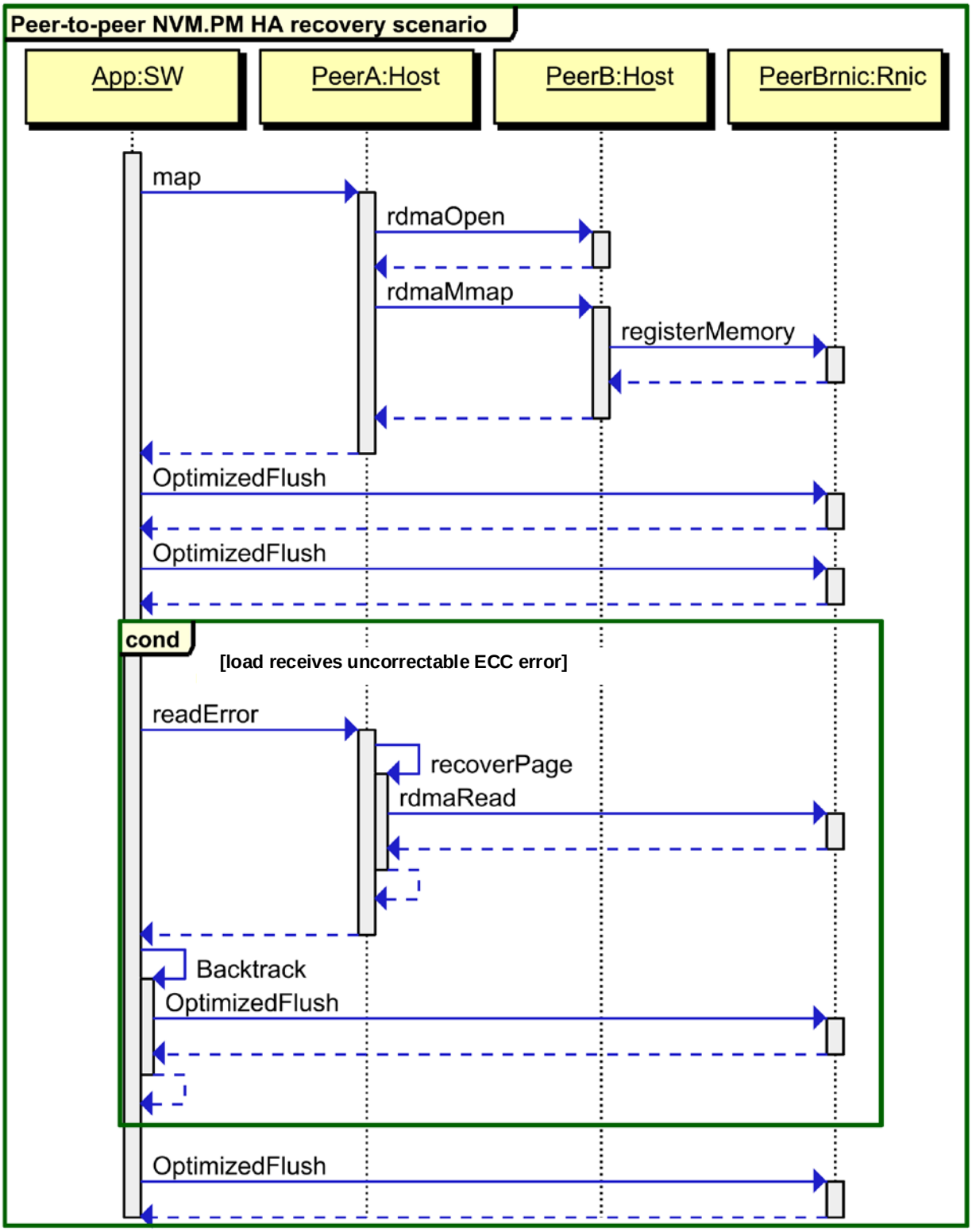

Figure 11 - Uncorrectable Error Recovery

For graphic simplicity the PM thread of this figure has been removed. It participates in OptimizedFlush 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 an RDMA read to recover the lost data. Since the data is only as recent as the last time it was referenced by an OptimizedFlush, backtracking (such as a transaction abort) may be required on the application’s exception handling thread as described in section 4.5.2. Any aborts may require additional rdmaScatterWriteAndFlush 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 RDMA session (not shown).

6.5 HA across multiple processor architectures

The use of RDMA, or similar methods of 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 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 Revision 1 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 RDMA operations due to processor architecture differences or packetization of data within or below the RDMA transport layer of the network protocol stack.

Since there are no common specifications of failure atomicity related to either RDMA 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 absense 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 RDMA Security

This section provides an overview of security concepts and their application to RDMA-based transports. It includes a summary of the RDMA security model prescribed by

RFC 5042, lists various threat models, and describes the various RDMA transports and relevant security mechanisms.

This section does not address security of data at rest (e.g., encryption of user data), as that is independent of the RDMA transports and is provided by mechanisms specific to each storage device type.

7.1 Security Concepts

Data security has the objective of preventing the improper disclosure or alteration of data in storage devices. Many of the concepts are defined in the *SNIA Dictionary*. For purposes of this white paper, several areas of interest are described informally here:

data at rest
Data in a storage volume may be subject to disclosure if the volume can be stolen. This is often mitigated by encrypting the data before or during storage.

data in flight
Data being written to or read from a storage device may be subject to disclosure if the connection can be snooped. This is often mitigated by encrypting the connection. This usually requires hardware resources in the device and the host to perform the encryption/decryption without sacrificing transmission speed.

authentication
Authentication is the process by which a storage device determines the identity of a host attempting to access it. If a host is not authenticated, then it will not be allowed to access any data in the device.

authorization
Once a host is authenticated, then the storage device may determine whether the host is authorized to perform the particular operation it is requesting. For example, some hosts may be permitted to read data from a volume, but not to write data to the volume.

provisioning
Provisioning is the process of configuring a storage device for operation in a particular system. With respect to security, provisioning includes installing credentials which the device can use to authenticate remote hosts and specifying the operations which each host is authorized to perform.

channel binding
Channel binding (see RFC 5056) is the binding of a pair of end points mutually authenticated at a higher-level protocol to a secure channel in a lower-level protocol. This permits delegation of session protection to the transport layer, which can provide better performance than performing encryption at the application layer.

7.2 RDMA Security Model

7.2.1 Overview

RFC 5042, *Direct Data Placement Protocol (DDP) / Remote Direct Memory Access Protocol (RDMAP) Security*, analyzes security issues for DDP and RDMAP and presents countermeasures to protect systems. Figure 12 is reproduced from the RFC and shows the RDMA reference model and is used to analyze security threats and solutions. Detailed explanations of the concepts described in section 7.2 can be found in RFC 5042 and associated RDMA standards.

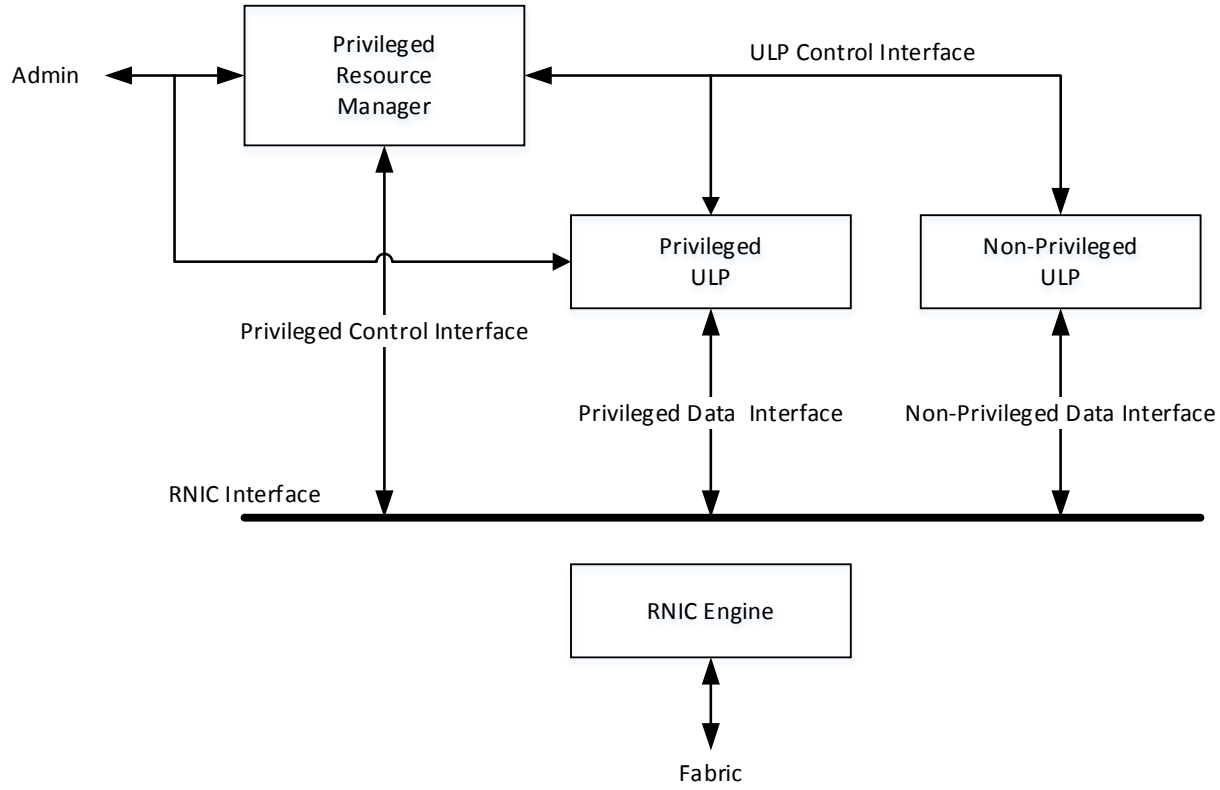

Figure 12 – RDMA Security Model

The elements shown in Figure 12 are:

a) The RDMA network interface controller (RNIC) implements the RDMA protocol to access the fabric (which implements a lower layer protocol, or LLP).
b) The privileged resource manager manages RNIC resources.
c) A privileged upper layer protocol (ULP; e.g., an application or middleware library) is trusted by the local system not to attack the operating environment.
d) A non-privileged ULP is not trusted, and its requests must be verified.

These are concepts that apply to all RDMA implementations, although individual implementation will differ.

7.2.2 Protection Domains

A protection domain (PD) is a collection of RDMA resources for purposes of isolation and security. It is a local construct and is never visible on the connection between nodes. When mutual authentication is successfully performed, a PD is created in the host and a PD is created in the storage device. The resources which may be assigned to a PD are:

- Connection endpoint (a queue pair consisting of a request queue and a send queue)
- Steering tag (STag – a scalar identifying the destination buffer for a tagged message, e.g., data to be transmitted)

The RNIC is responsible for ensuring that resources in one PD are not accessible by resources belonging to another PD. A node is required to check that the endpoint and the STag are in the same PD before permitting the operation to access the resource.

The use of these objects in the context of PM devices is explored by this document.

7.2.3 Partial Trust

Partial trust is a concept for defining which threats are addressed by specific security techniques; in other words, one party assumes that another will not use a particular set of attacks. There are three characteristics which may or may not be present in a particular trust model (i.e., sharing of local resources, local partial trust, and remote partial trust). These are used to define five trust models:

- NS-NT – Non-Shared Local Resources, no Local Trust, no Remote Trust
- NS-RT – Non-Shared Local Resources, no Local Trust, Remote Partial Trust
- S-NT – Shared Local Resources, no Local Trust, no Remote Trust
- S-LT – Shared Local Resources, Local Partial Trust, no Remote Trust
- S-T – Shared Local Resources, Local Partial Trust, Remote Partial Trust

7.2.4 Remote Partial Trust

Partial mutual trust among a set of RDMA streams (see RFC 5040) implies that one authentication can apply to all streams in the set. All may be in the same protection domain. Conversely, one protection domain must never contain streams among which partial mutual trust does not exist.

For example, not all ULPs using a host's file system may be trusted. The PM device may not be able to trust the file system on a host, and all streams from the file system cannot be assumed to be in the same protection domain.

7.3 Threat Models

The need for particular protection mechanisms depends upon the security threats:

- If the data center is physically secure to the extent that pilfering of storage devices is very unlikely, then encryption of data at rest may not be necessary.
- If the storage area network is secure against snooping, then encryption of data in flight may not be necessary.
- If the storage area network is secure against introduction of a rogue host, then authentication may not be necessary.
- If there is a rogue process on a host (i.e., a non-privileged ULP in Figure 12), it may be possible to limit its access.

Many networks of interest provide better security than general purpose nodes on a general purpose network. Examples include:

- Distributed storage devices which are networked have special-purpose functionality and may lack commonly-attacked functions found in general purpose nodes. Moreover, these arrays often use dedicated private networks.
- Software defined storage (SDS) virtual machines usually utilize a private network.
- Virtual appliances implemented as SDS virtual machines that have prescribed functionality implemented on the VM may provide fewer points of attack.

A PM device can perform a number of actions to protect itself from unauthorized access:

- Authenticate each protection domain (see 7.2.2). Require that each protection domain is associated with a single fabric client node. Each node may have multiple protection domains.
- Use fabric-specific encrypted connections to client nodes (e.g., IPsec for IP-based fabrics, FC-SP for FC based fabrics) to prevent snooping. Enforce channel binding to ensure that authentication performed in-band is associated with the right connection.
- The privileged resource manager (see 7.2.1) can reduce the impact of denial of service (DoS) attacks by controlling all scarce resources (e.g., by reaping the resources of idle streams and not sharing RQs, CQs, STags across streams).
- Prevent buffer overflows to protect data of different streams.

7.4 Transport Security

This document addresses four of the fabric transports (i.e., LLPs) for which RDMA mappings are defined: iWARP, RoCE, RoCEv2, and InfiniBand.

7.4.1 iWARP

iWARP (RFC 5040) provides mappings of RDMA to TCP and to SCTP. Because these rest upon IP, IPsec could be used for authentication and encryption of data in flight.

7.4.2 InfiniBand™

InfiniBand (InfiniBand Trade Association specification) defines RDMA over the InfiniBand fabric.

7.4.3 RDMA over Converged Ethernet (RoCE)

RoCE (Annex A16 of volume 1 of the InfiniBand specification) is defined over L2 framing and does not address authentication, authorization, or encryption of data in flight.

7.4.4 RDMA over Converged Ethernet version 2 (RoCEv2)

RoCEv2 (Annex A17 of volume 1 of the InfiniBand specification) is defined over UDP (which utilizes IP), and thus can utilize IPsec for authentication and encryption of data in flight.

8 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 7. Error handling processes can generally be described in several parts:

- Detection – some piece of hardware or software gets the first indication that an error has occurred.

- 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 insures that the entire system has responded to the error in a comprehensive manner. This may involve parts of the system that were not initially involved in error detection or local recovery.
- Reporting – software logs the error, possibly at multiple levels.

In the software layering of Figure 7, 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 are grouped with 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 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.

8.1 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.

8.2 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.

8.3 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.

9 Requirements Summary

As is often the case it is difficult to isolate requirements from implementation examples. While this document frames HA for NVMP in terms of RDMA, RDMA per se is not the only way to address these requirements. 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 small 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. One could envision this as a set of small byte ranges that are packaged in one RDMA and piggybacked with remote flushing to persistent memory. This requirement also motivates a kind of scatter gather RDMA that operates at both the application and the remote access server as shown in Figure 9.
- Efficient large transfers

  Byte range transfer optimization cannot 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.
- 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. This should provide a warning in conditions where access to data structures after failover may be in doubt.
- Atomicity of fundamental data types

  It is not clear that this requirement is met by any current implementations without the use of CRC. 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.

- Resource recovery after failure

  Consideration must be given to the ease of recovering RDMA resources dedicated to failed components. This must be addressed in order to limit the scope of resets during failure recovery.

- Isolation/HW fencing after failure

  Correct failure recovery generally assumes fail stop behavior of failing nodes before remaining nodes resume activity. This 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.

One outstanding area of requirement investigation has to do with the security, RDMA resource management and flow control necessary to assure safe and correct operation with as much latency reduction as possible. There is a general notion that these areas can be simplified relative to the use of RDMA in today's non-PM file systems. In PM systems, RDMA 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 – Workload Generation and Measurement

While there are some benchmarking tools for memory mapping available (e.g. IOZONE) these tools offer little control over parameters such as the ratio of sync calls to stores in mmapped workloads. It would be useful for controlled testing and measurement purposes to have a new benchmark that offered fine grain control over syncs.

As always, it would be even more valuable to ascertain what benchmark settings best represent specific applications. Since the timescales involved make memory workloads harder to measure than IO workloads, memory workload characterization may require hardware instrumentation making it even more elusive than IO workload characterization.

Workload parameters

The following traditional IO benchmarking parameters should be included:

- Number of threads
- File size (or memory mapped region size)

There may be an option to determine whether each thread opens and mmaps the same file/region or a different one. When used in the file context, whole files should be mmapped.

Once a file is opened and memory mapped, the workload is not described in terms of IO's but rather syncs interspersed with loads and stores. The following new parameters should be included:

- Load record size (bytes)
- Store record size (bytes)
- Load/Store ratio
- Number of Store's before corresponding sync
- Number of syncs per sync group
- Total number of records to visit in a trial

Based on this set of parameters a given thread does a sequential series of Load or Store instructions until reaching the designated record size. It then uses the Load/Store ratio to decide whether to switch from Load's to Store's (or vice versa). Regardless of whether the access type is switched the thread chooses a new random address within the memory mapped region at which a new sequence of sequential Load's or Store's begins.

After some number of Store cycles in the above pattern, a sync is generated for the first Store record as indicated by the "Number of Store's before corresponding sync". At that point sync's are generated for a group of Store records as indicated by "Number of syncs per sync group". The pattern continues by triggering a sync group each time a Store record gets old enough.

This pattern of activity continues until the total number of records to visit is reached, at which point the remaining Store records are synced and region is unmapped.

With this starting point in mind, a number of variations can be created including the following:

- Use optimized sync for each sync group
- Apply statistical or patterned distributions to various parameters
- Vary the size of an individual Load or Store instruction within a record
- Repeated Store's to the same address before a sync
- Add a background workload that is never synced
- Augment or replace the sync with begin/end transactions

Measurements

It would be desirable to ascertain the following statistics from each trial

- Record access rate
- Sync response time

Since response times are difficult to measure in software we will probably have to settle for a timed run of a specific number of record accesses. This leaves the question of how to measure sync times, which may require the use of a sampled profiling approach that determines what percentage of the total run time elapsed during sync.

Appendix B – 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 C – 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 D – 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

"NVM Atomics" Paul Van Beheren et. Al – as of this writing this companion white paper is still under development.

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 E – 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.