SDC 19

September 23-26, 2019
Santa Clara, CA

Remote Persistent Memory

SNIA Nonvolatile Memory Programming TWG

Tom Talpey (Microsoft)

Alan Bumgarner (Intel)

SNIA NVMP TWG co-chairs

Outline

- SNIA NVMP TWG background
- NVMP Interface concepts
- RDMA requirements and extensions
- Current and future TWG focus

NVM Programming Model TWG - Mission

- Accelerate the availability of software that enables Persistent Memory (PM) hardware.
  - Hardware includes SSD's and PM
  - Software spans applications and OS's

- Create the NVM Programming Model
  - Describes application-visible behaviors
  - Allows API's to align with OS's
  - Describes opportunities in networks and processors

SNIA NVM Programming Model

- Version 1.2 approved by SNIA in June 2017
- Expose new block and file features to applications
  - Atomicity capability and granularity
  - Thin provisioning management
- Use of memory mapped files for persistent memory
  - Existing abstraction that can act as a bridge
  - Limits the scope of application re-invention
  - Open source implementations available
- Programming Model, not API
  - Described in terms of attributes, actions and use cases
  - Implementations map actions and attributes to API's

NVM Programming Model modes

	Block Mode Innovation	Emerging PM Technologies
User View	NVM.FILE	NVM.PM.FILE
Kernel Protected	NVM.BLOCK	NVM.PM.VOLUME
Media Type	Disk Drive	Persistent Memory
NVDIMM	Disk-Like	Memory-Like

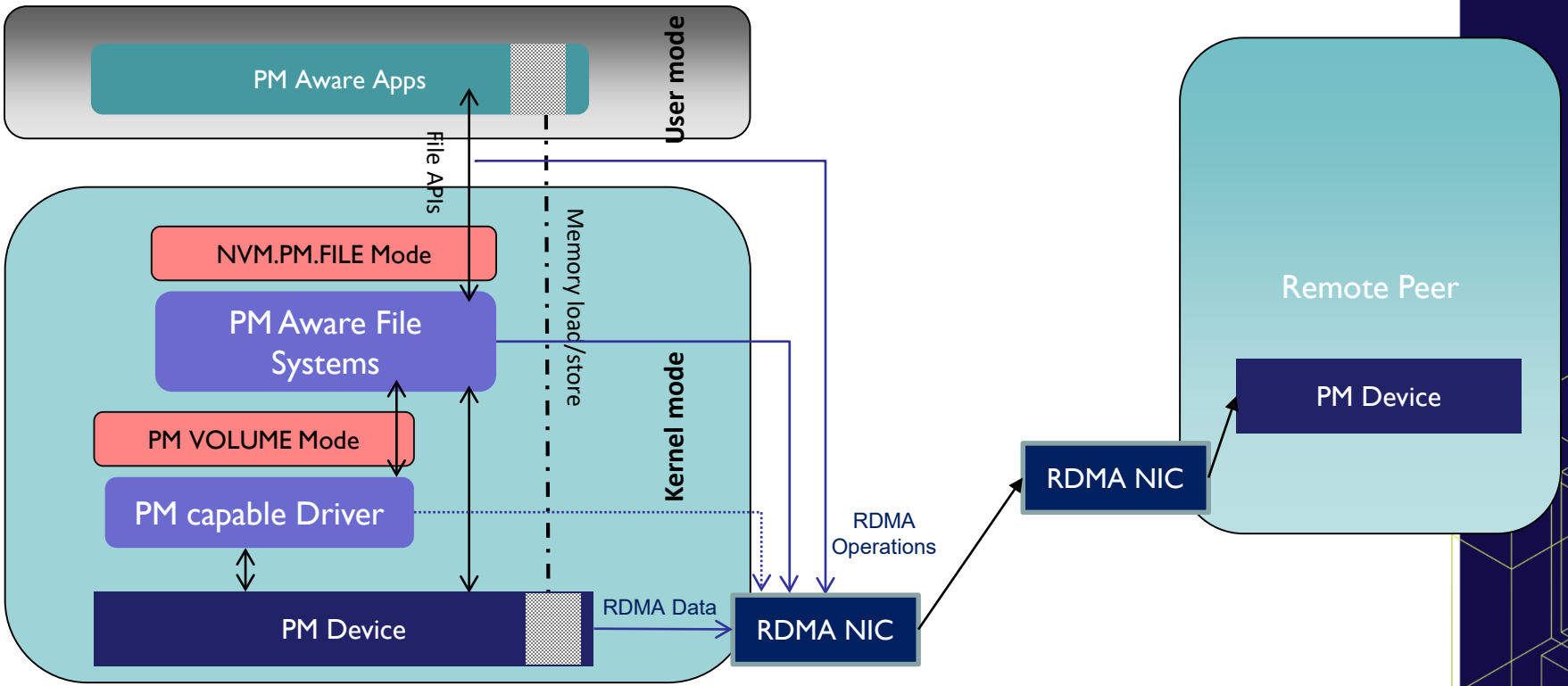
- Block and File modes use IO
  - Data is read or written using RAM buffers
  - Software controls how to wait (context switch or poll)
  - Status is explicitly checked by software
- Volume and PM modes enable Load/Store/Move
  - Data is loaded into or stored from processor registers
  - Processor waits for data during instruction
  - No status returned – errors generate exceptions

The current version (1.2) of the specification is available at https://www.snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf

Remote Access for HA

- History
  - Remote Access for High Availability white paper
    - Originally published 2016
    - Update published May 2019
- NVM Programming Model Specification update in development
  - Update specification to reflect learning from implementations
  - Incorporate learning from remote access white paper
    - Asynchronous Flush
    - Remote persistence ordering, error handling
- Remote Access Collaboration with Open Fabrics Alliance OFIWG
  - PMEM Remote Access for HA
  - Expand remote access use case enumeration

Persistent Memory (PM) Modes, +Remote



NVM Programming Interface

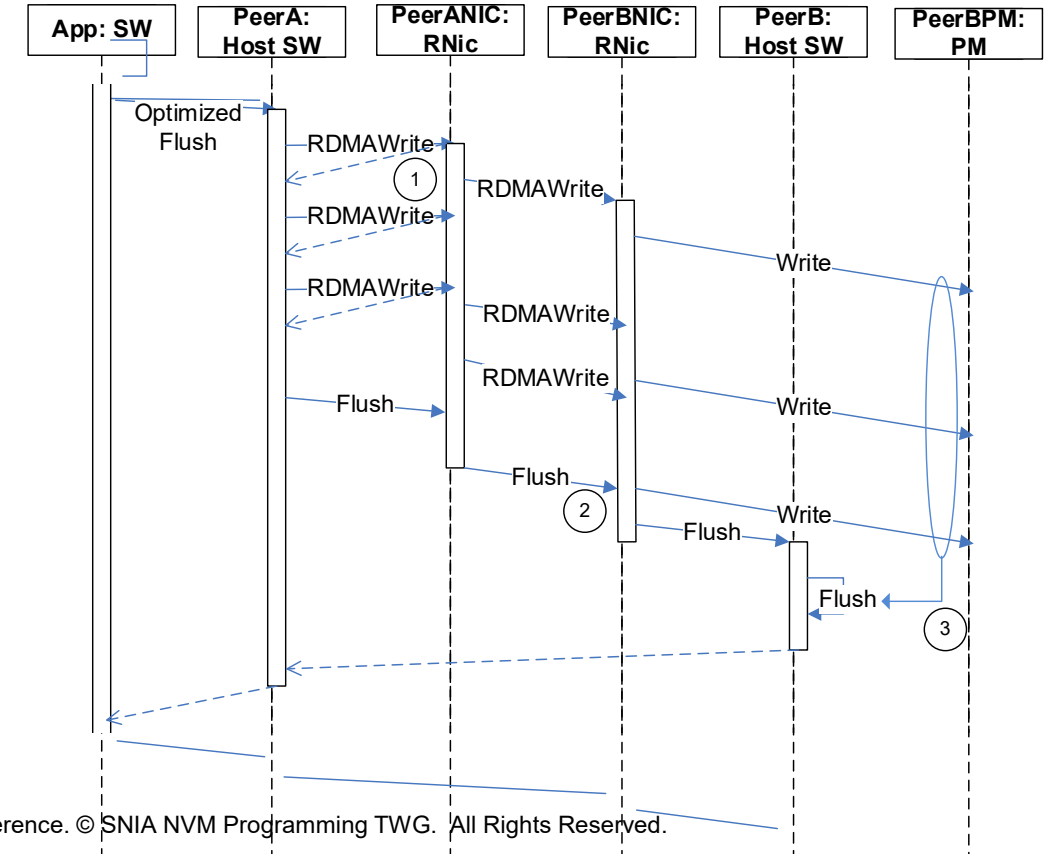
NVMP Interface: Map and Sync

- Map (local PMEM)
  - Associates memory addresses with open file
  - Caller may request specific address
- Sync (local PMEM)
  - Flush CPU cache for indicated range
  - Additional Sync types
  - Optimized Flush – multiple ranges from user space
  - Optimized Flush and Verify – Optimized flush with verification from media
  - Warning! Sync does not guarantee order
- All the above are true remotely, but
  - Remote addresses are not “mapped”
    - Stores do not magically become RDMA Writes
  - Flushing applies to RDMA, network and i/o pipeline (not simply CPU)
  - An asynchronous flush is needed for efficiency!

Key Remotable NVMP interfaces

- Directly mappable to RDMA (with extensions):
  - In NVMP 1.2:
    - OPTIMIZED_FLUSH
    - OPTIMIZED_FLUSH_AND_VERIFY
  - Under discussion for updated NVMP:
    - ASYNC_FLUSH (initiates flushing)
    - ASYNC_DRAIN (waits for flush completion, persist fence)
    - Ordering (write-after-flush)
- Other NVM PM methods remotable via upper layer(s)

PMEM Remote Access ladder diagram



Asynchronous Flush

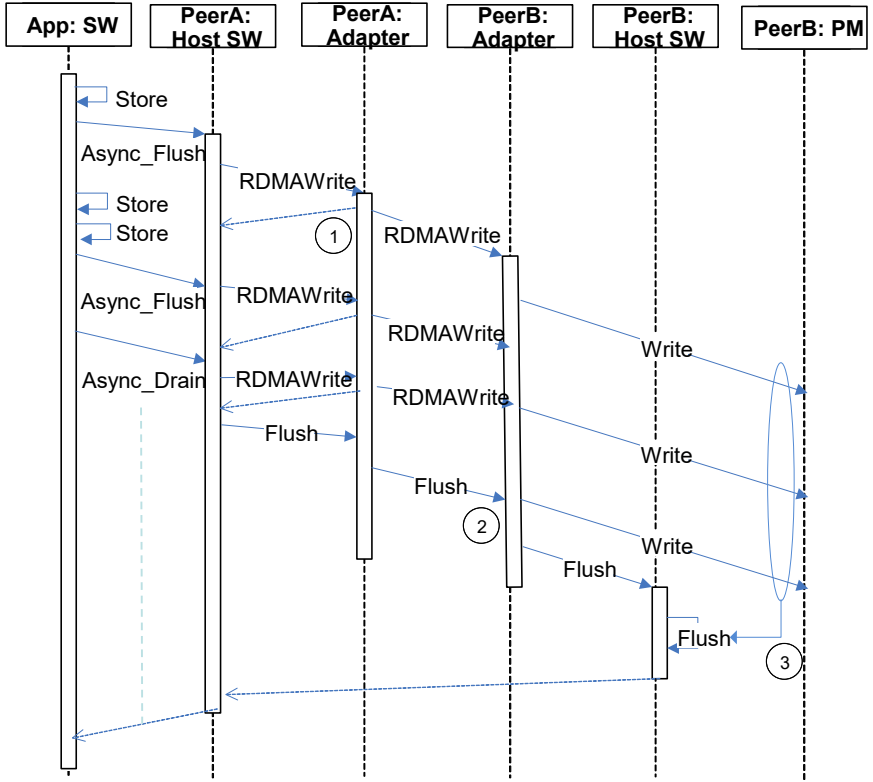
- Separates “Flush” and “Drain” stages
- Allows early scheduling of Writes without blocking
  - “Giddy-up”
  - Important for efficient concurrent processing
  - For both applications and middleware libs
- Drain allows application to ensure persistence
  - Less data remaining to flush: less wait latency
- Error conditions require careful analysis
  - Subject of NVMP TWG current work

Asyncronous Flush (NVMP 2.0 Proposal)

- Application overlapped processing
- Async_Flush invokes library
  - Library initiates RDMA Write(s) to RPM
  - Pipelined - does not wait, immediately returns
- Additional application processing…
- Async_Flush initiates more RDMA Write(s)
  - Pipelined - does not wait
- Async_Drain initiates Remote Flush
  - Library queues RDMA Flush after all prior RDMA Writes
  - Async_Drain completes only after all Writes Flush to PM
    - Note: application may also continue during this processing

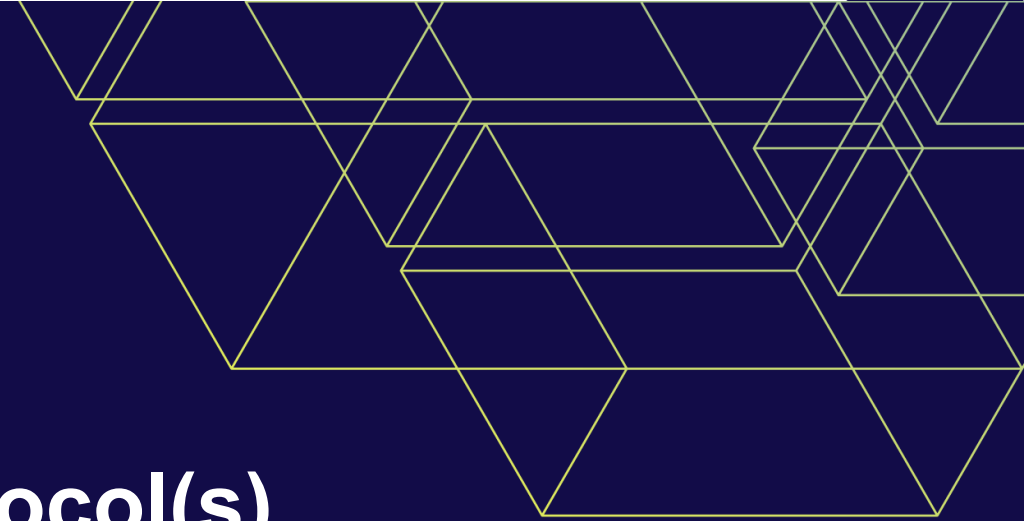
- Tricky bits (1,2,3):
  - Same as in prior example!
  - But note subtlety:
    - Application **Flush** -> RDMA **Write**
    - Application **Drain** -> RDMA **Flush**

- RDMA protocol:
  - ***Same as in prior example!***
  - "Ordering Nexus" is simply the Queue Pair



Additional NVMP Concepts

- "Consumers of visibility" vs "Consumers of persistence"
  - Visibility – e.g. network shared memory
  - Persistence – e.g. storage
- Differing semantic
  - E.g.: Compare and Swap in PMEM does not necessarily yield a persistent lock!
  - Because visibility is **not** achieved atomically with persistence
- Assurance of persistence integrity ("Verify")
  - Explicit integrity, as opposed to current Best-effort
- Scope of flush
  - Global, per-stream, or per-region?
    - Conceptual "store barrier" or "order nexus" abstraction
    - Streams of **stores**, which are later flushed to ensure persistence
  - Flush hints (ASYNC_FLUSH)
  - Modeling these in programming interface, and protocol
  - Understanding, and guiding, platform implementation



Protocol(s)

RDMA Protocols

- "RDMA Flush" proposed
  - Broad agreement in IBTA, IETF
  - RDMA Protocols being extended to support remote persistence guarantee
- Requires additional platform support
  - Possible PCIe extension
- Progressing slowly, but surely

RDMA Flush

- New RDMA transport operation
- Existing RDMA memory operations remain unchanged
- Flush executes like RDMA Read, and like local Flush/Drain
  - Ordered, acknowledged
    - And, flow-controlled
  - Requestor specifies byte ranges to be made durable
    - Memory Region range-based {region handle, offset, length}
      - Responder response guarantees specified range is persisted
      - Responder may flush additional bytes based on implementation
- Scope considerations
  - Per-connection, per-region or per-region-range possible
  - Single Flush may act upon many prior Writes
  - Responder acknowledges only when persistence complete
    - Connection breaks if error occurs
- Selectivity considerations
  - Flush to “Visibility” or “Persistence”

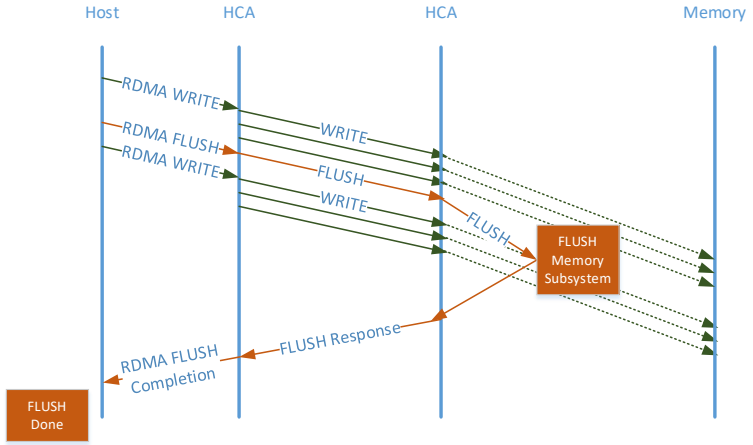
RDMA Flush - Overview

- Non-Posted/Queued
  - Non-deterministic execution time (PCIe, media type, media interface)

- Preserve RDMA Operation Model
  - Follow Existing RDMA Ordering Rules of Non-Posted/Queued operations
    - Posted operations (i.e. WRITE) can bypass non-posted operations (i.e. READ)
    - Non-posted (i.e. READ) operations can't bypass posted operations (i.e. WRITE)
  - RDMA transport operations remain unchanged

- Performance Requirements
  - Amortize Cost of the FLUSH Operation
  - FLUSH Selectiveness
  - FLUSH Pipelining

- Types
  - Global Visibility
  - Persistency



Persistency vs Visibility

- Persistency
  - FLUSH type persistency shall ensure the placement of preceding data accesses in a memory that persists the data across power cycle, response shall be sent only after successful completion in the responder.
- Global Visibility
  - FLUSH type consistency shall ensure the placement of the preceding data accesses in the memory domain to be visible for reading for the responder platform
- The responder shall respond after the FLUSH has been executed and completed according to its type

- N.B. Persistency also/always provides Visibility

Flush Scope

- Memory Region Range
  - FLUSH preceding data access within the range {Handle, VA, Length} within the QP
- Memory Region
  - FLUSH preceding data access within the Handle and within the QP
- All
  - FLUSH all preceding data accesses within the QP

- Implementation determined by the responder
  - Not explicit in the protocol

RDMA Atomic Write

- Also require a transactional write in support of two-phase commit
- Solution: Atomic Write operation
  - Second new transport function
  - Atomically updates 8-byte size, 8-byte aligned value
  - Non-posted/Queued, and ordered after Flush

PCI Protocol

- Efficient RDMA Flush requires PCIe extension
- Allow RDMA adapter to invoke without CPU
  - Minimal latency and overhead
- PCI SIG reportedly considering Flush semantic
  - To enable platform-independent RNIC behaviors
- PCI “Atomic Ops ECN” (August 2017)
  - May provide additional semantic guarantees for Atomic Write RDMA operation
- “Out of Band” (platform-specific) solutions are possible

Workloads

Remote PM Workloads

- High Availability (HA)
  - Resilience, recovery, “RAID-like” properties
  - Replication
  - Scaleout
- Transactions
  - Atomicity (failure atomicity)
- Networked Shared Memory
  - Including Publish/Subscribe model
- And others!

- Desire to maintain:
  - Ultra-low latency (~ +1 RTT: O(μseconds))
  - Programming model compatibility
    - Ideally, transparency!

Workload: Basic Replication

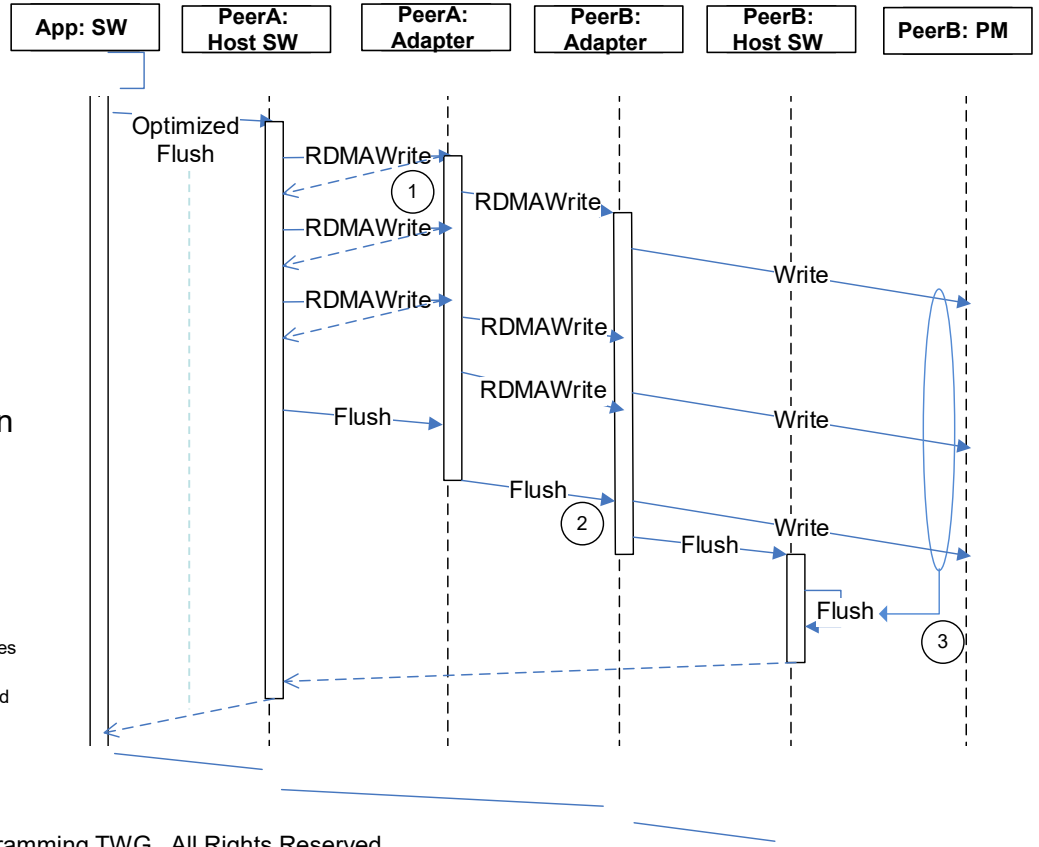
- Simple replication (mirroring) with writes and flushes

- Write, optionally more Writes, OPTIMIZED_FLUSH
  - No overwrite
  - No ordering dependency (but see following workloads)
  - No completions at data sink
  - Pipelined (no “bubbles”)

Remote Flush (Basic Semantic)

- Optimized Flush invokes library
  - Library initiates RDMA Write(s) to RPM
  - Library initiates Remote Flush
    - Ordered after prior Writes
    - And blocks for Write+Flush completion
    - Returns (only) when Flush is complete

- Tricky bits:
  1. RDMA Writes complete at requestor before stores to PM at responder
  2. Remote Flush arrives before Writes are executed at responder
  3. Remote Flush must wait at responder until all Writes are safely in PM



Workload: Log Writer
(Filesystem, Database)

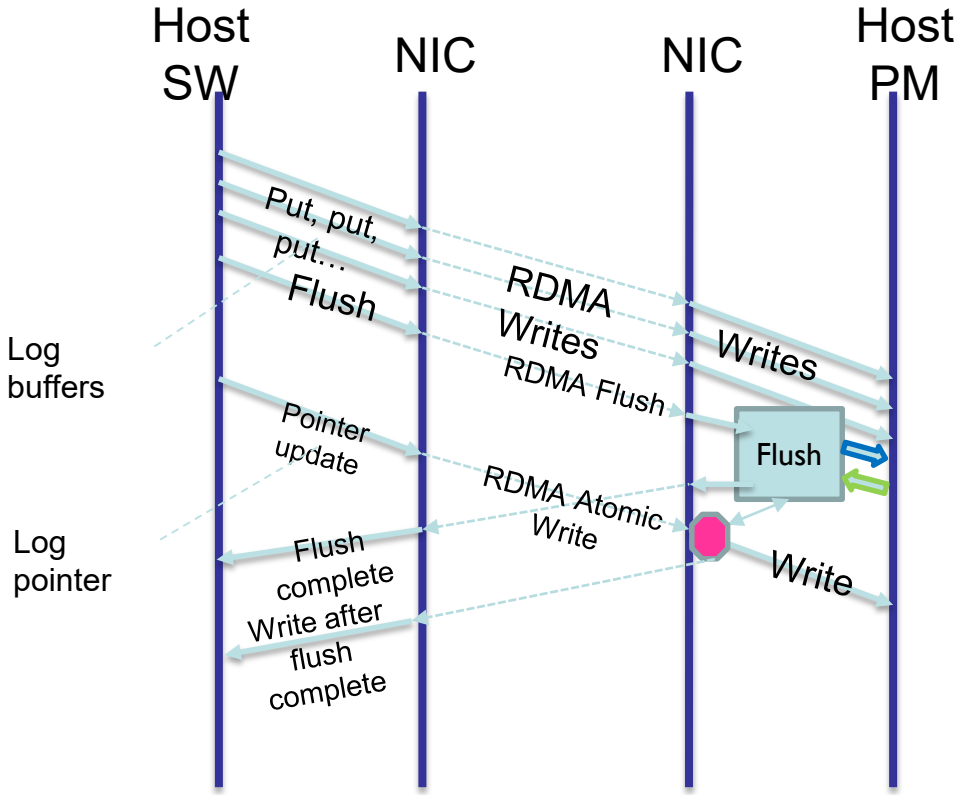
- For (ever)

  { Write log record, Commit }, { Write log pointer }

  - Latency is critical
  - Log pointer cannot be placed until log record is successfully made durable
    - Log pointer is the validity indicator for the log record
    - i.e., Transaction model
  - Log records are eventually retired, buffer is circular
- Protocol implications:
  - Must ensure successful commit (e.g. ASYNC_DRAIN)
  - Potentially introduces a pipeline bubble – which would be very bad for throughput and overall latency
  - Desire an ordering between Commit and second Write to avoid this
- Utilize RDMA “Atomic Write”

Write, Flush and Atomic Write

- RDMA Atomic Write
  - Additional new non-posted/queued operation
  - Executes at responder only after successful prior non-posted operations (i.e. Flush)
  - Implementable at responder with or without PCIe Atomic support
- Logwriter example shown
- Similarly able to support 2-phase commit

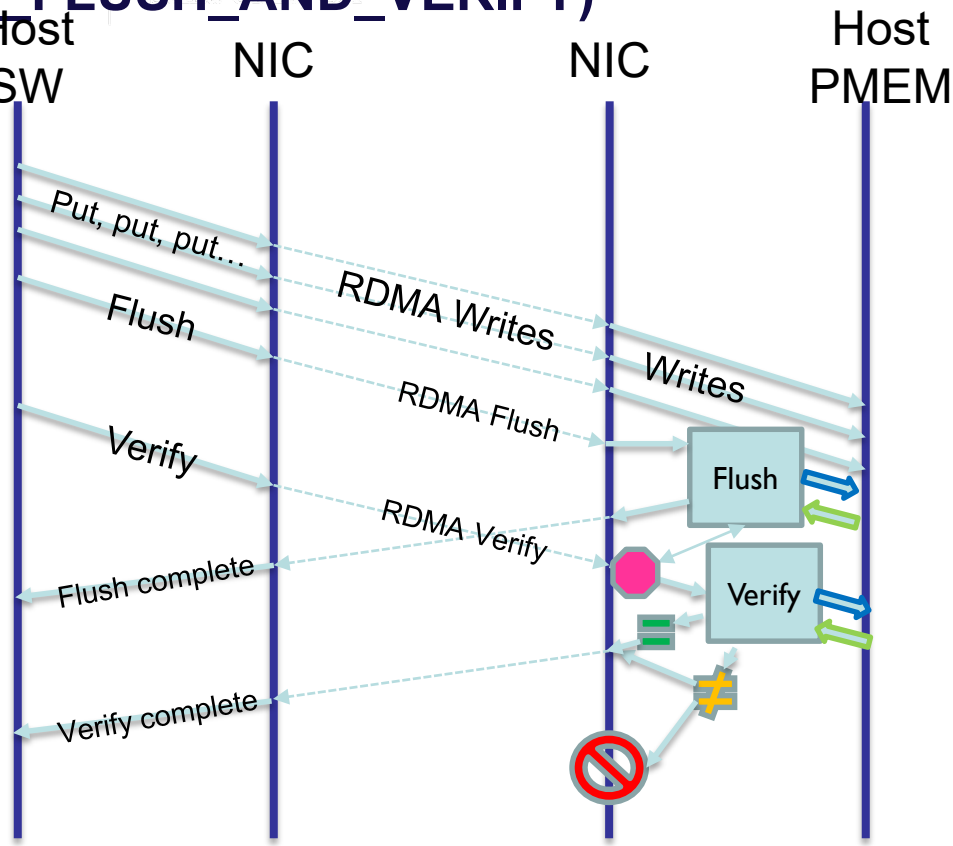


Workload: Paranoid Log Writer
(Remote Data Integrity)

- Assuming we have an RDMA Write + RDMA Commit
- And the Writes + Commit all complete (with success or failure)
- How does the initiator know the data is intact?
  - Or in case of failure, which data is not intact?
  - BEFORE completing the transaction (e.g. writing the log pointer)
- Possibilities:
  - Reading back
    - extremely undesirable (and possibly not actually reading media!)
  - Signaling upper layer
    - high overhead
    - Upper layer possibly unavailable (the “Memory-Only Appliance”!)
- Same question applies also to:
  - Array “scrub” (verifying existing data)
  - Storage management and recovery (rebuild)

Write, Flush and Verify
(OPTIMIZED_FLUSH_AND_VERIFY)

- RDMA “Verify”
  - *Under discussion*
- Computes and returns the hash of a region
  - Non-posted/queued to execute at responder only after prior Flush etc
  - Must read the actual persistence domain, not the visibility domain!
  - Optional behavior to return the hash, or break connection on mismatch
- In support of enhanced “Optimized Flush and Verify”
- Supports “paranoid log writer”
  - Using break-on-mismatch to fence a following Atomic Write
  - Without requiring a pipeline bubble!
- Also supports “scrub”
  - Using return-the-hash



RDMA Extension Implications on Programming Model

- Strengthens need for ASYNC_FLUSH
- Increased imprecision of errors
  - RDMA connection is simply broken
- Need for bubbling up AtomicWrite completion?
- Asynchronous Verify indication?
- Verify Fail imprecision in connection-break mode

RDMA PM Extensions Next Steps

- SNIA NVMP TWG specification work continues
  - OFIWG feedback on semantics
  - Integration of RDMA Protocol proposed extensions
- IBTA, IETF RDMA Standards specification proceed
- OFIWG and RDMA software implementation
  - In Open Source, commercial operating systems, etc
- RDMA vendor implementation
- PCI SIG specification and broad PCIe implementation

Ongoing NVMP TWG work

- Core NVM PM update
  - Current “v2.0” work in progress (in the NVMP TWG)
- Asynchronous Flush
- Incorporate implementation learnings
  - Optimized Flush, Deep Flush, Flush on Fail
  - Interaction between NVMP and C memory model (!)
  - Visibility versus Persistence
- Continue Remote Access for HA work
  - Greater RDMA mapping detail
  - Efficient remote programming interface models
  - RDMA and platform requirements clarified
  - Errors, error handling, error recovery in remote scenario
  - Collaboration with Open Fabrics Alliance OFI WG
- Scope of flush, flush barriers, analysis
- “Flush on Fail Fail” (failure of persistence) analysis

References

SNIA NVM Programming TWG:
https://www.snia.org/forums/sssi/nvmp

PM Programming Model:
https://www.snia.org/tech_activities/standards/curr_standards/npm

PM Remote Access for HA (and other papers):
https://www.snia.org/tech_activities/standards/whitepapers


Thank you!