SDC
STORAGE DEVELOPER CONFERENCE
SNIA SANTA CLARA, 2016

NVMe over Fabrics support in Linux

Christoph Hellwig
Sagi Grimberg

NVMe over Fabrics: the beginning

- Early 2014 demo apparently involved Linux (neither of us were involved)
- During early spec development were at least two implementations: Intel (+ a few others) and HGST.
- Various Linux developers were / are involved with the NVMe and NVMe over Fabrics specification.

HGST prototype

- Initially just tunneled NVMe commands over the existing SRP protocol structures
- Then tried to accommodate the existing draft spec where possible
- Where not possible, change the spec.

NVMe Linux Fabrics Driver WG

- In 2015 a new working group of the NVM Express organization was created to merge the different Linux development streams.
- Multiple members contribution to the code and even more testing the code.
- Tried to follow Linux-style development as much as possible:
  - Private git repository
  - Mailing list

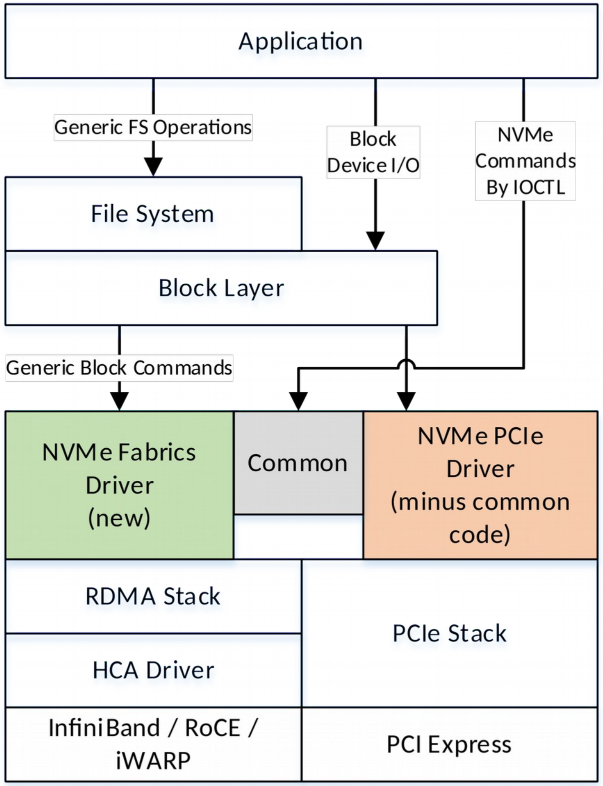
NVMe Linux Host Driver

- Even before the release of the spec we started splitting the existing Linux NVMe driver into a common and a PCIe specific part:
  - Added block layer pass through support for internal NVMe commands (similar to what Linux does for SCSI)
  - Separated data structures into common and PCIe
  - Added struct nvme_ctrl_ops
  - And moved the code around (new **`drivers/nvme/`** directory)

NVMe Linux Host Driver

- The existing NVMe host driver was split into a layer subsystem.
- Core components:
  - ***nvme-core***: Implements the native nvme specification in a fabric independent fashion.
  - ***nvme-fabrics***: implements the transport-independent parts of the fabrics specification.
- Drivers:
  - ***nvme-pci***: PCIe specific host driver.
  - ***nvme-rdma***: RDMA specific host driver.

NVMe over Fabrics Host Driver



The new Fabric drivers uses the existing common code

The transport driver is in control of the actual I/O path (no additional indirections for the fast path)

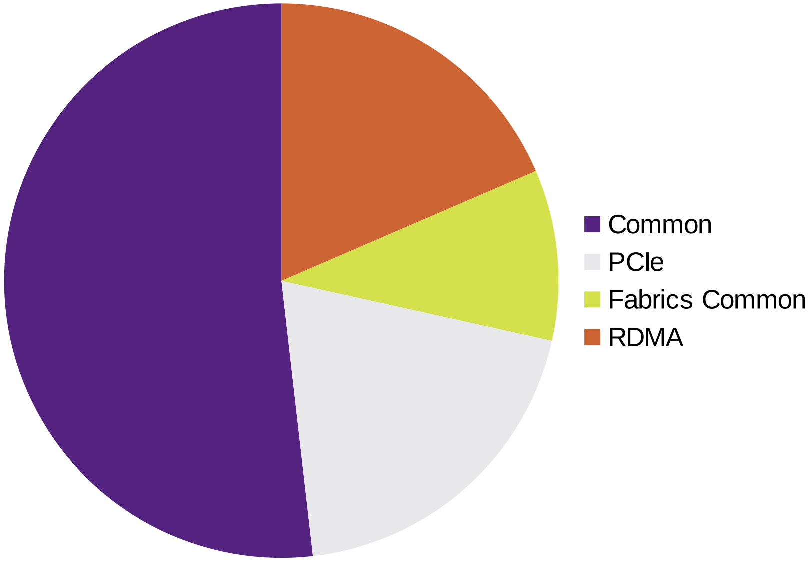
Existing user space APIs of the PCIe driver are all also supported when using Fabrics

Uses new sub-command of the existing **nvme-cli** tool to connect to remote fabrics controllers

NVMe over Fabrics Host Driver

- Features supported:
  - Various RDMA transports (Infiniband/RoCE/iWARP)
  - Dynamic connect/disconnect to/from multiple controllers
  - Discovery support (using **nvme-cli**)
  - Keep-alive
  - Basic multi-pathing (using **dm-multipath**)
- Features not supported (yet):
  - Authentication (spec not done yet)
  - Fibre Channel transport

NVMe Linux Host Driver now



- Most code is shared for the different transports
- Transport drivers are fairly small (~2000 lines of code)

NVMe Target

- Supports implementing NVMe controllers in the Linux kernel
  - Initially just NVMe over Fabrics
  - Adding PCIe support (e.g. using vhost) could be done later
- Split into a generic target and transport drivers:
  - RDMA
  - Loop (for local testing)
  - Fibre Channel support is work in progress

NVMe Target

- The NVMe target can use any Linux block device:
  - e.g. NVMe, SAS, SATA, ramdisk, virtio
- Uses the block layer to communicate with the device
- Early experiments with NVMe command pass through not continued

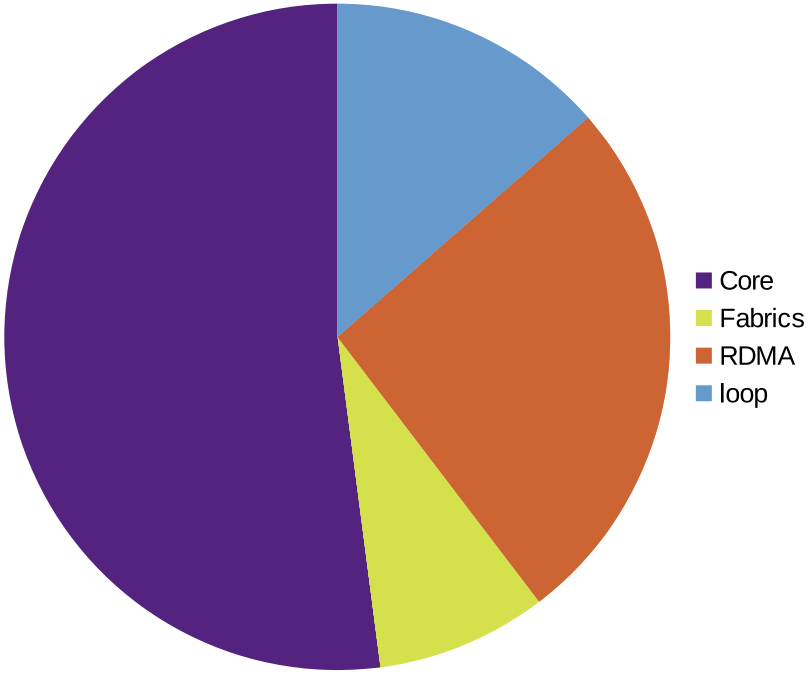
NVMe target

- Supported features:
  - All mandatory NVMe I/O commands
  - Full set of NVMe (over Fabrics) admin commands
  - Data Set Management (DSM) – for deallocate
  - Dynamic subsystem/controller/namespace provisioning
  - Discovery service support (including referrals)

NVMe target

- Features not supported (yet):
  - Smart/Error log pages
  - Authentication (spec not done yet)
  - Persistent reservations
  - Fused commands
  - NVMe security protocols (Security Send / Receive)
  - PCIe Peer to Peer I/O (e.g. CMB in a backend NVMe device)

NVMe Target



- Again most code is in the core
- The whole core (~ 3000 lines of code) is smaller than many SCSI target transport drivers
- We aggressively tried to offload work to common code:
  - RDMA subsystem
  - configfs subsystem

  and will continue to do so for new features, e.g. Persistent Reservations

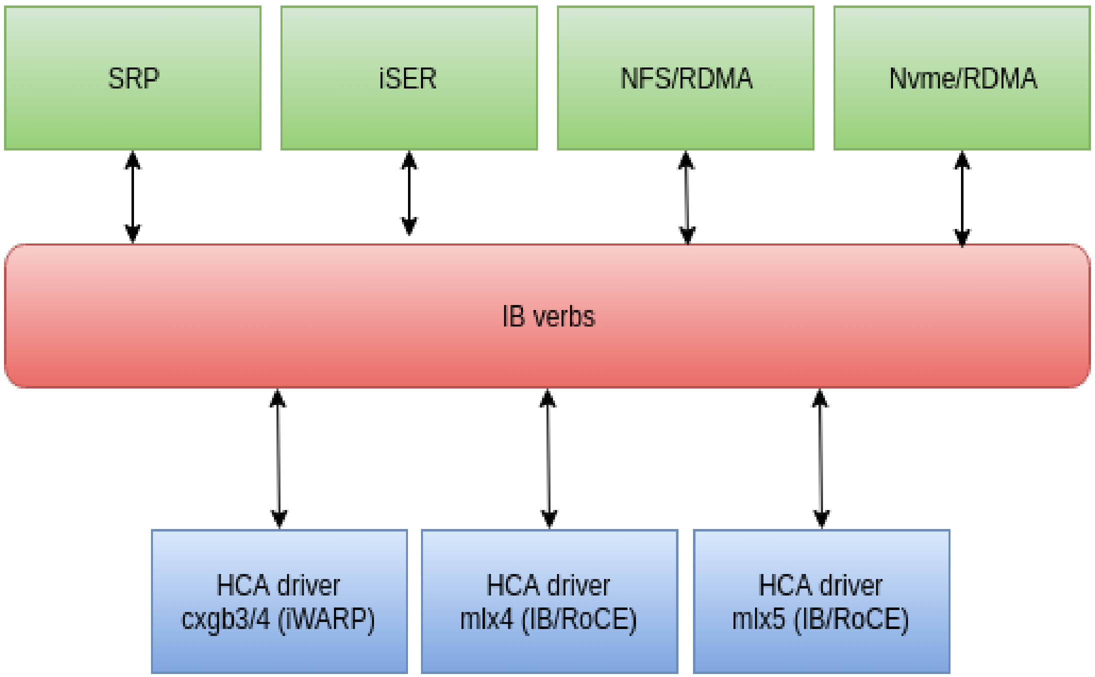
NVMe Target – configuration

- Uses a configfs interface to let user space tools configure the tool.
  - Simpler and more integrated than the SCSI target
- The prime user space tool is called nvmetcli and is written in python
  - Allows for interactive configuration using a console interface
  - Allows saving configurations into json format files and restoring them

Nvmetcli

```
root@testvm:~/nvmetcli# ./nvmetcli
/> ls
o- / ......................................................... [...]
  o- hosts ................................................... [...]
  o- ports ................................................... [...]
  | o- 2 ..................................................... [...]
  |   o- referrals ........................................... [...]
  |   o- subsystems .......................................... [...]
  |     o- nqn.2014-08.org.nvmexpress:NVMf:uuid:77dca664-0d3e-4f67-b8b2-04c70e3f991d  [...]
  o- subsystems .............................................. [...]
    o- nqn.2014-08.org.nvmexpress:NVMf:uuid:77dca664-0d3e-4f67-b8b2-04c70e3f991d  [...]
      o- allowed_hosts ....................................... [...]
      o- namespaces .......................................... [...]
        o- 1 ................................................. [...]
        o- 2 ................................................. [...]
/>
```

Linux RDMA stack



Linux RDMA stack

- The Linux RDMA stack has three layers:
  - Consumers (called ULPs)
  - RDMA core layer (API and management)
  - Device Drivers
- As part of the NVMe RDMA development a lot of shared logic and code duplication moved to the core
- Both ULP drivers and device drivers benefited from code simplification and optimization

RDMA memory registration

- In order to allow remote access a driver needs to register memory buffers with remote access permissions:
  - The stack offered 6 different methods to register memory
  - A lot of effort to support different methods
  - Results in code duplication in both ULPs and device drivers

RDMA memory registration

- Converged on a single memory registration method: Fast Registration Work Requests (**FRWR**):
  - New, simpler memory registration API that leaves most work to the core
  - Provides a simple interface for both ULPs and providers (uses the common *scatterlist* structure).
- Migrated existing ULPs to the new API

RDMA CQ API

- RDMA completion queue polling implementation was duplicated across the ULPs
- Each got some right (and some wrong).
  - Several had CQ re-arming, fairness and context abuse issues
- We converged on a higher-level API that gets it right once and provides a simple interface to the ULPs

RDMA CQ API

- Exposes a simple interface for ULPs to attach a “done” handler to each work request (similar to the Linux block-layer *bio* interface)
- Offers several CQ polling contexts:
  - Software-IRQ (using a new ”*irq-poll*” library)
  - Workqueue (Process-like context)
  - Direct polling (for application driven polling)
- Migrated existing ULPs to the new API

RDMA CQ Pooling API

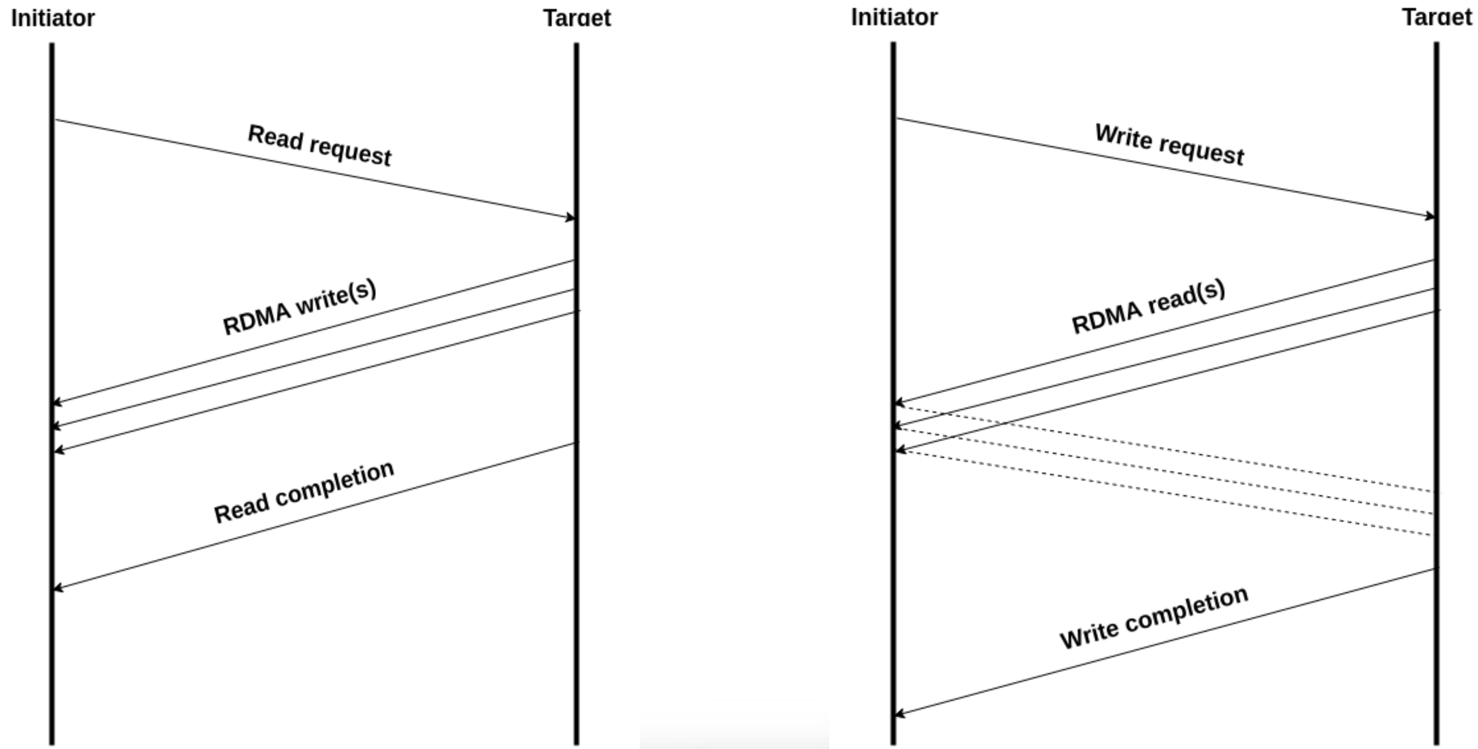
- RDMA applications can achieve better performance scalability by sharing RDMA completion queues:
  - Better completion aggregation
  - Less interrupts
- Smart and fair completion queue spreading by affinity requirements provides better parallelism in completion processing

RDMA CQ Pooling API

- Moves completion queue allocation and assignments to the RDMA core.
  - Completion queue pools are allocated in a lazy fashion on demand
  - ULPs can pass an optional affinity hint
  - Moves topology knowledge from ULPs into the core
- Work in progress, expected to make kernel 4.9

Storage RDMA data transfer model

- All the RDMA storage ULPs share the classic RPC model:



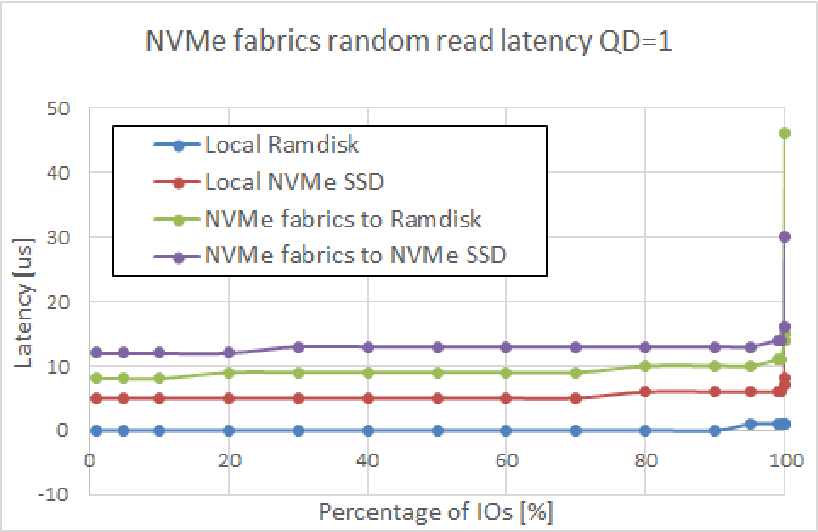
RDMA Read/Write implementations

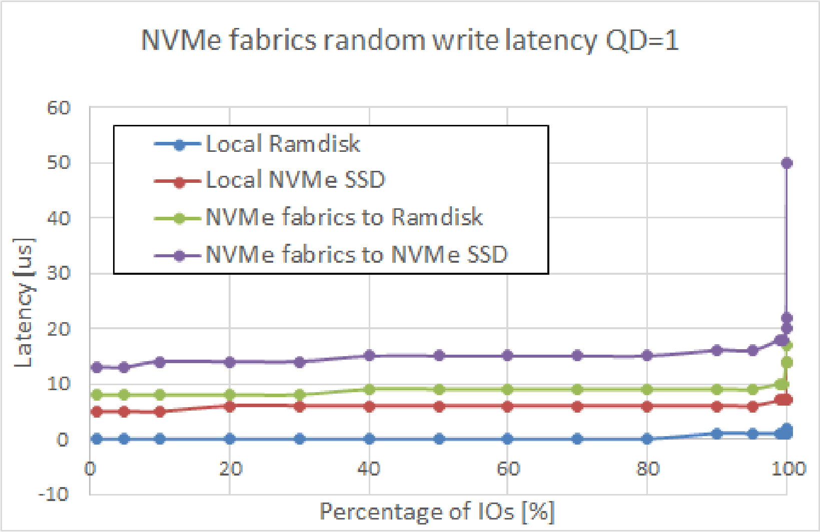
- Every single ULP implemented the same sequence of operations to transfer data over RDMA:
  - Again, lots of code duplication
  - All the logic lived in the ULPs, no logic in the core or HCA drivers
  - Some got it right (and most got it wrong somewhere)
- ULPs struggled to support iWARP which is slightly different from IB/RoCE

RDMA Read/Write API

- A new core library that implements generic data-transfer using RDMA READ/WRITE.
  - Exposes a simple and intuitive interface for ULPs (uses the common *scatterlist* structure)
  - Masks RDMA transport differences (iWARP vs IB/RoCE)
  - Provides support for Mellanox Signature extension (to implement T10 PI)
- Optimizes the implementation with correct work request pipelining, completion signalling and extra features
- Migrated most existing ULPs to the new API

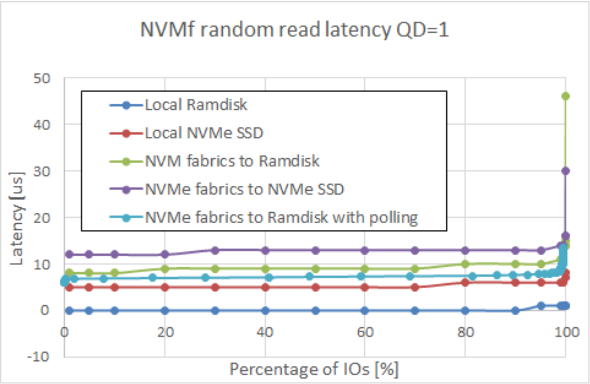
Initial Performance Measurements

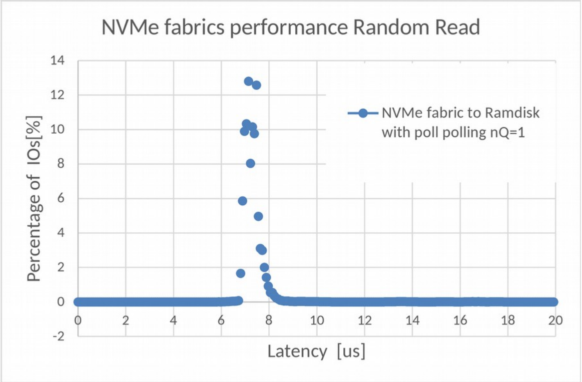




- 13us latency for QD=1 random reads
  - Sub-10us network contribution

More Performance Measurements





Polling allows for sub-7us added latency

Status

- All code mentioned has been merged into Linux 4.8 (only release candidates so far)
- Fibre Channel support for both the host and target will be submitted soon
- The updated nvme-cli with Fabrics support and nvmetcli need to get into the distributions

Links

- NVMe over Fabrics source code:

  https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git

  http://git.infradead.org/nvme-fabrics.git (for-next)

- Nvme-cli repository:

  https://github.com/linux-nvme/nvme-cli

- Nvmetcli repository:

  http://git.infradead.org/users/hch/nvmetcli.git

Questions?