Providing QoS Guarantees in the LightOS™ NVMe/TCP SDS System

Alex Shpiner
Lightbits Labs
NVMe-oF: from DAS to a Disaggregated Cloud

- Efficient scalability
- Maximal utilization - support more users
- Easy maintenance and operation

Direct-Attached Architecture

Lightbits Cloud Architecture

Networking

Storage

Compute

Storage

Compute
NVMe/TCP Overview
NVMe-oF: Short Recap

- 2014: Initial NVMe/RDMA pre-standard prototypes, NVMe-oF 1.0 standardization started
- 2015: NVMe.org formed Fabrics Linux Driver Task Force
  - Converged on the starting point prototype
  - Heavy lifting of NVMe stack reorg contributed even before NVMe-oF support
- Jun 2016: NVMe-oF 1.0 ratified
- Oct 2016: NVMe over Fabrics support (host and target) merged into the Linux kernel (v4.8)
Why NVMe/TCP?

- Ubiquitous - TCP is probably the most common transport, it runs on everything everywhere
- Well understood - both in terms of theory (academia) and in terms of implementations (industry)
- High performance - TCP delivers excellent thread scaling
- Well suited for large scale deployments and longer distances
- Actively developed - maintenance and enhancements are developed by major players
Ratification Status

- NVMe/TCP specification (NVMe TP-8000) ratified in Nov 2018
- Integral part of NVMe-oF 1.1 specification, on-track to be published Sep 2019
- Multi-vendor joint effort to ensure interoperability
  - Nov 2018: first UNH-IOL NVMe-oF Plugfest to include NVMe/TCP testing (PoC-level)
  - Regular NVMe-oF Plugfest feature ever since
NVMe/TCP - Linux Support

- 2017: Fabrics Linux Driver Task Force took on developing the NVMe/TCP driver
  - Two prototypes existed (Lightbits, SolarFlare)
  - Converged on a single code-base moving forward

- As the spec evolved, code adjustments followed providing feedback to the standardization

- Mar 2019: NVMe/TCP merged into the Linux kernel (v5.0), soon to be - or already! - available out of the box in your favorite distro

- Already running in production environments (backported)
NVMe/TCP in a Nutshell

- TCP is the transport layer below NVMe layer
- NVMe commands are sent over standard TCP/IP sockets
- TCP provides a reliable transport layer and congestion control
Association Model

- Controller association maps NVMe queues to TCP connections 1:1
  - No controller-wide sequencing
  - No controller-wide reassembly constraints
- Connection binding is performed at NVMe-oF connect time (binding queue to controller)
PDU: Protocol Data Unit

- NVMe-oF Capsules and Data are encapsulated in PDUs
- PDU structure varies per PDU type
  - 8-byte Common Header
  - Variable length PDU specific header
- PDUs optionally contain Header and/or Data digest protection
- PDUs contain optional PAD used for alignment enhancements
PDU Types

- **ICReq/ICResp**: Connection Initialization PDUs
- **H2CTermReq/C2HTermReq**: Connection Termination PDUs (only for error flow)
- **CapsuleCmd/CapsuleResp**: NVMe-oF Command and Response Capsule PDUs
- **H2CData/C2HData**: Unidirectional Data PDUs
- **R2T**: Ready to Transfer PDU (solicit **H2CData**)
I/O Flow

- Host-to-Controller data transfers ("writes") can be:
  - *In-capsule* - if the controller supports it (optional)
  - In a solicited H2CData PDU (R2T PDU)
Linux NVMe Host Stack

- TCP transport driver naturally plugs into the existing stack
- Almost no special additions for TCP (to this point)
- Control plane very similar to RDMA
  - Still have plenty of room for code reuse
Linux NVMe/TCP Host Features

- Zero Copy Transmission
- Header/Data Digest
- CPU/NUMA affinity assignments for I/O threads
- TLS Support - *Future*
  - Will probably be trampolined to userspace for TLS handshake
- Polling Mode I/O - *Future*
  - Need to continue polling with an inherent context switch
- Automatic aRFS (*accelerated Receive Flow Steering*) support - *Future*
  - Need to figure out atomicity of NIC steering table updates
- Out-of-Order Data transfers - *Future*
  - Probably fabrics 1.2 material
QoS in LightOS™
- First commercially-available NVMe/TCP open storage platform
- Software-Defined Storage: runs on standard servers, with commodity SSDs
- Based on standard networks, without proprietary client hardware or software
- High throughput, consistent low latency, data services, QoS
- 100Gbps streaming compression/decompression and erasure coding
- Thin provisioning
- Storage server clustering (multi-server data protection)
Multi-tenant Storage Challenges

**Problem**: Unpredictable performance or behaviour of the application (service)

- Noisy neighbours
- Impact of writes on performance of reads
- Write imbalance across SSDs
- No performance (throughput, latency, etc.) guarantees per tenant

**Naive solution**: overprovision resources so there are always spare IOPs.

- But this is expensive...

**Better solution**: QoS (Quality of Service)
LightOS End-to-End QoS Value Proposition

- **Reads and writes separation**
- **Guaranteed Service: Per-tenant I/O capping**
- **Balance write I/Os and endurance across SSDs**
Problem: Head-of-queue (HOQ) blocking: read latency is affected by presence of writes.

- Read requests (few bytes) can be placed behind large write request (eg. 1MB)
- Read requests will not be processed before write request is consumed by application from the network
Read/Write Separation

**Solution**: Read/Write Separation

- Linux supports separate Queue Maps since v5.0 - leverage that
- Client side dedicated read queues and dedicated write queues (TCP connections)
- Target side dedicated NIC queues for read connections and write connections
Test the impact of Large Write I/O on Read Latency

- 32 Readers issuing synchronous 4KB Read I/O
- 1 Writer that issues 256KB Writes, QD=16
I/O Capping

**Problem**: noisy neighbours

**Solution**: IO capping per tenant

- Multi-queues system
- Arbiter - coordination between queues
  - ...and between parallel front-end cores

![Diagram of I/O Capping system with NIC, FE, Arbiter, and Global FTL connections]
I/O Capping: Multi-queues System

- Arriving requests are separated to queues by: (tenant, {write | read}).
- I/O capping per queue:
  - Queues are served (requests submitted to GFTL) according to quota allocated by the arbiter
  - Spare quota is spread equally among the queues (incl. best-effort queues)
- SLO-driven volume allocation (SLO - Service Level Objective)
  - New volumes are not allocated if combined SLO is not achieved by system capabilities
I/O Capping: Lab Results

- **Scenario:**
  - Two tenants sending read requests of 4KB from 4 clients each
    - A: queue depth 8
    - B: queue depth 128

Throughput

Each client gets fair share of BW

Throughput

Each client gets unfair share of BW proportional to its queue depth
Balance I/Os and SSD Endurance

**Problem**: Writes are not balanced across all SSDs

- Write amplification and garbage collection activity is different across SSDs
- Endurance of each SSD is different
- Read latency varies depending on which SSD is used to handle the read request
Balance I/Os and SSD Endurance

Solution: Writes are distributed evenly across all SSDs as they come
- Append only, no write-in-place
- SW-controlled garbage collection

Result:
- Same endurance for all SSDs
- Write amplification is balanced
- Read latency is predictable
  - Each SSD is serving the same write activity when a read arrives
Summary

- LightOS is a first commercial high-performance NVMe/TCP target with data services
- QoS is integral part of the system that copes with multi-tenant storage challenges
- Read-write separation provides low read latency by avoiding head-of-line blocking
- Per-tenant I/O capping provides guaranteed and isolated performance for every tenant
- Global FTL balances writes uniformly across SSDs for endurance and predictable read latency

Contact information:
www.lightbitslabs.com
alex@lightbitslabs.com