Solution Brief
Network and Edge Group

Intel® Ethernet 800 Series - Application Device Queues (ADQ) in a Kubernetes Environment

ADQ is a technology designed to improve application specific queuing and steering. ADQ addresses issues of predictability, latency, and throughput for scaling containerized applications in a Kubernetes environment.

Authors

Dave Anderson
Dave Cremins
Niamh Hennigan
Brian Johnson
Stefan Peters
Sridhar Samudrala
Anil Vasudevan

Executive Summary

- A microservices architecture approach delivers many benefits and a few new challenges. Containerized application workloads bring new latency, predictability, and scalability issues that must be addressed to realize the full benefit of cloud-native applications running in a Kubernetes environment.
- Organizations can improve containerized workload performance by adopting Application Device Queues (ADQ), a technology designed to reduce latency, improve predictability and throughput when used in concert with Intel® Ethernet 800 Series Network Adapters.
- For this solution brief, we created a container workload scenario, measured latency, predictability (tail latency), and throughput. To examine how containerized application workloads would benefit from ADQ, we ran tests with and without ADQ.
- Test results confirm that ADQ accelerates performance on containerized workloads by reducing latency, improving response time predictability and throughput.

This document is part of the Network Transformation Experience Kit.

Latency, Predictability, and Challenges at Scale

In a single data center server, serving multiple network applications, the queues responsible for network traffic may be shared between multiple applications and network resource contention may occur. The case is the same for cloud-native applications delivered through containers. As application utilization in Kubernetes orchestrated clusters increases and becomes more complex, latency response time delays can impact an organization's ability to consistently meet service-level agreements (SLAs).

Context switching and stack interrupts are other examples of Linux networking inefficiencies that can have a negative effect on performance.

The behavior of the latency is an important characteristic to examine. Containerized applications tend to have latency, which is variable and unpredictable. Variable and unpredictable container response times can be felt by end customers and may impact an organization's ability to quality of service (QoS) commitments.

Containers rely heavily on the network in order to communicate. As containerized applications scale from hundreds running on a single cluster to thousands running on distributed clusters, they place higher demands on the network. As network traffic increases, jitter can occur, resulting in unpredictable containerized application response times.

How ADQ Works to Improve the Performance of Containerized Workloads

Ethernet is like a freeway system for data traveling between different places in the data center and the time taken to complete each packet's journey depends on the traffic conditions. Comparable to a freeway during rush hour when there are many cars with different origins and destinations all sharing the same lanes, there is increased unpredictability. The situation is the same for web servers and data caching servers that operate with a big data backend.

By dedicating queues for critical containerized workloads, ADQ accelerates performance by reducing latency, improving response time predictability and throughput.

With ADQ
Application traffic filtered into a dedicated set of queues

Without ADQ
Application traffic intermixed with other traffic types

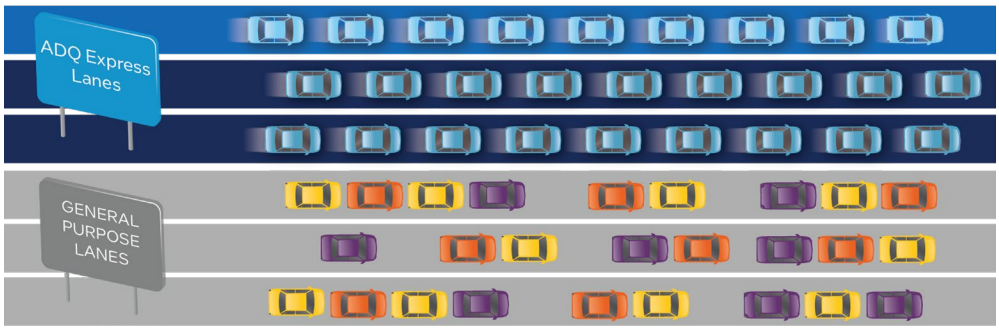

Figure 1. ADQ acts like an express lane for your containerized application workloads

At the core of ADQ is an application-specific queuing and steering technology that dedicates and isolates application-specific hardware NIC queues. These queues are then connected optimally to application-specific threads of execution. By preventing other traffic from contesting for resources with a chosen application's traffic, performance becomes more predictable and less prone to jitter. Additionally, application-specific outgoing network bandwidth can be rate-limited, which can be used to divide or prioritize network bandwidth for specific applications. Unlike other receive queuing and steering technologies, pinning CPU cores is not required with ADQ.

ADQ can work transparently to the application. It does so through the use of application independent pollers. Advanced steering modes provide flexibility, depending on the deployment. For more details on the poller and steering mode configurations used please reference the E810 ADQ Configuration Guide (Table 1 Reference 1). This enables ADQ acceleration to be applied to a larger set of containerized applications.

ADQ Requirements for Kubernetes environments

- Container Networking Interface with VETH functionality (e.g., Cilium)
- Linux based OS only
- 5.12+ kernel version recommended for best supportability

ADQ in a Kubernetes Environment

Container orchestration is all about managing the lifecycles of containers to control and automate tasks like hardware abstraction, container provisioning and deployment, resource allocation and scaling, availability management, load balancing and external exposure of services running in a container with the outside world.

Kubernetes was built as a framework to handle large workloads with many nodes in an efficient manner. While this is true, as the cluster size increases, so too does the complexity of the networking. The increased density of applications across the cluster as scaling occurs leads to decreased visibility and the need for increased control on the networking protocols to maintain the QoS. Multi-tenancy issues can also occur when multiple workloads share resources. Communication and resources must be carefully monitored and managed to ensure that optimal service is provided to the users, further highlighting the need for advanced steering in a Kubernetes environment. On this basis, it can be concluded that Kubernetes will benefit from ADQ features. The figure below illustrates how ADQ works in a Kubernetes environment.

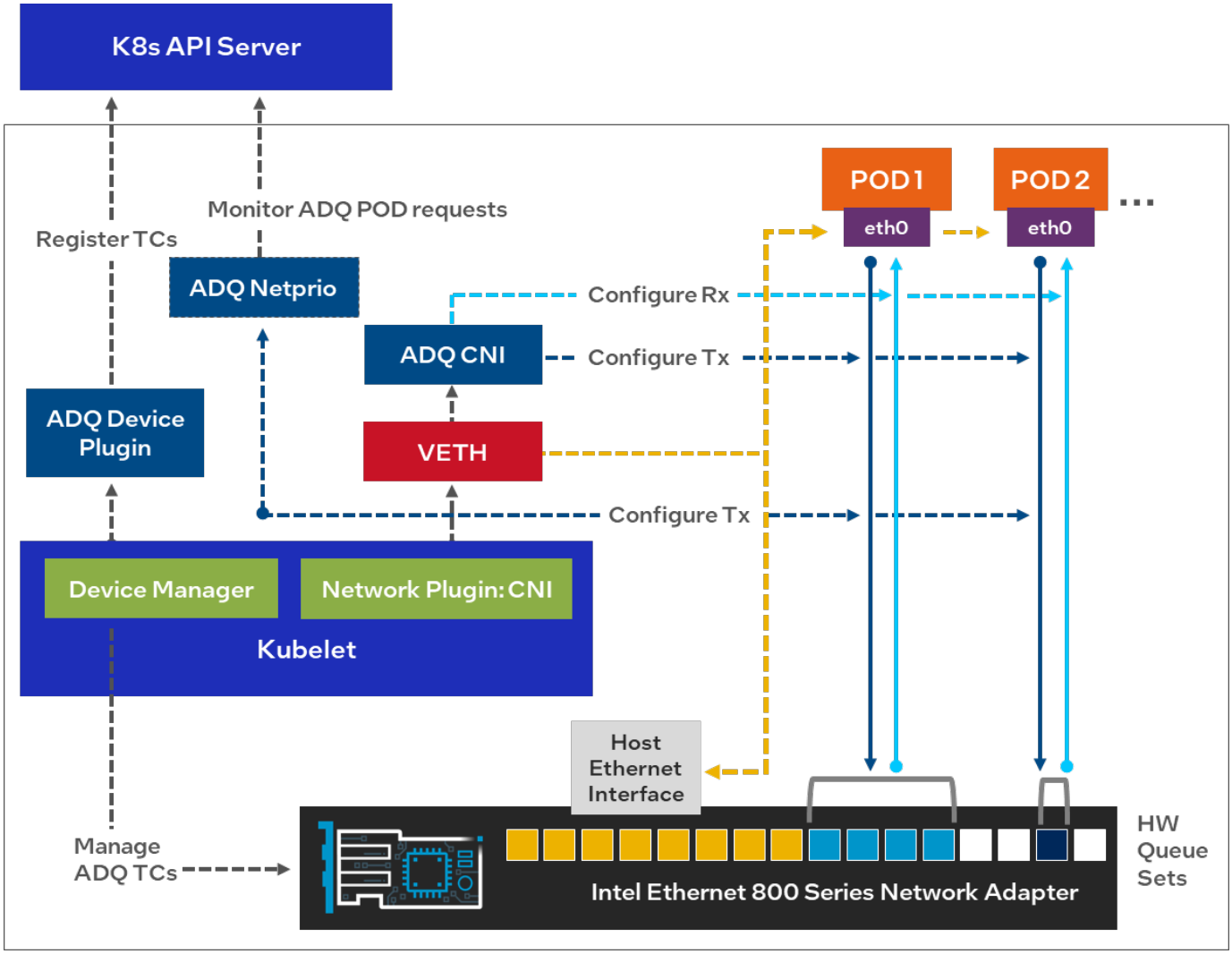

Figure 2. Application Device Queues in a Kubernetes Environment

The ADQ solution for containers running in a Kubernetes environment consists of three main elements: device plugin, container network interface, and Netprio (net_prio.ifpriomap functionality). Netprio is not essential but can be used as an optional component dependent on the CNI capabilities.

ADQ Device Plugin is responsible for resource management. It advertises the resources available on the system to the Kubernetes API server and makes the device available to the container. As shown in Figure 2, the ADQ Device Plugin advertises queue sets formed with hardware queue pairs, available on the Intel Ethernet 800 Series Network Adapter. A queue pair consists of a separate transmit (TX) and receive (RX) queue on the network adapter.

ADQ Container Network Interface (CNI) is responsible for creating the network namespace for the pod. It is invoked on the creation and deletion of a pod and handles the connectivity to the pod and configures the network interface to the pod. Linux Traffic Control (TC) infrastructure enables the creation of rules that we can put in place to control traffic flowing in an out of the Linux kernel. ADQ utilizes these TC rules to map hardware queues to software. It also implements filters that allow the packets for different applications reach their intended queue set. In our case, the ADQ CNI configures the container interface eth0 to be connected to the relevant queue set and sets the associated filters for that queue set. This allows the queue set to be connected directly to the workload pod.

Egress queue set steering is accomplished by setting network priority and/or queue mapping for outgoing traffic. To set the network priority for outgoing traffic, we must know the container ID. When a new pod is being created, a unique cgroup subdirectory is created. The ADQ Netprio application has a watcher on this directory to inspect any new subdirectories created and see if any are requesting an ADQ resource. If they are, the ADQ Netprio can then set a new pod watcher that will wait until it sees a 'Ready' status on the newly created container. When the container is up and running the ADQ Netprio will get the container ID and set the priority of egress traffic on a given interface. If Netprio is not used, the CNI will utilize rtnetlink API to set the egress queue set/queue steering configuration.

Test Scenario

Our goal was to measure the performance impact of applying ADQ on containerized application workloads.

An example container workload scenario was created to measure the latency, predictability (tail latency), and throughput of container workloads. To examine how containerized application workloads would benefit from ADQ, testing was performed without ADQ (non-ADQ) and with ADQ.

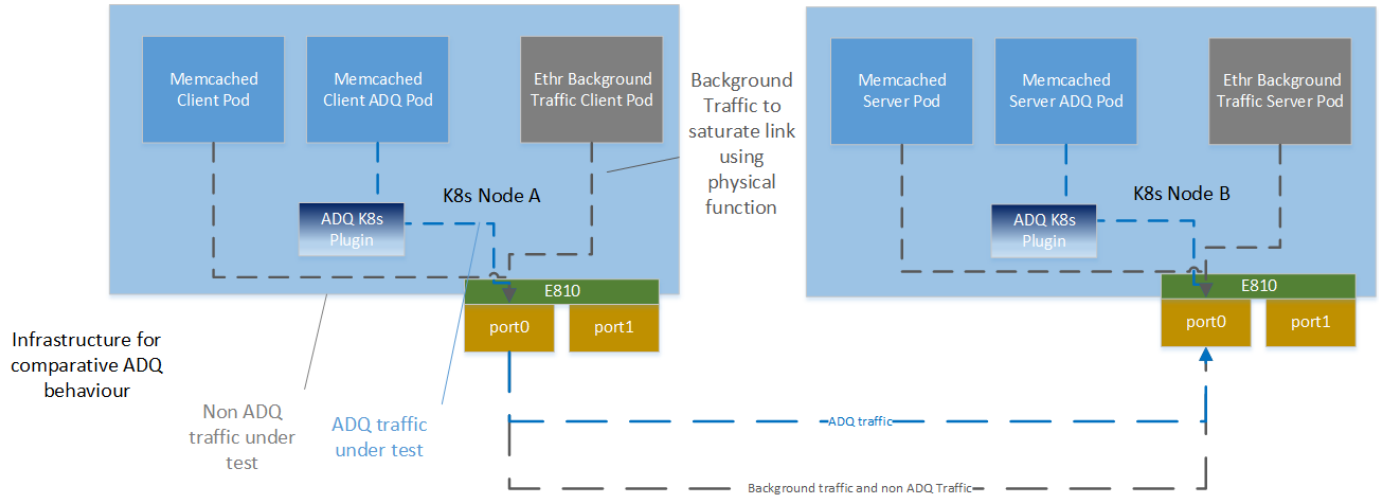

Figure 3. Infrastructure used for comparative ADQ and non ADQ traffic observation

The intention of the example was to illustrate ADQ and non-ADQ traffic co-existing on a network link in a Kubernetes cluster (non-ADQ meaning without any dedicated hardware queues assigned or established TCs being used). This was an example of ADQ behavior on a link and was not intended as an exhaustive performance evaluation.

A set of Memcached servers and clients were deployed for measurement. The testing environment consisted of two nodes where Intel Ethernet Network Adapters were used for back-to-back connectivity. Background traffic was generated using the opensource Ethr tool to simulate a live network scenario (verified to 100 Gbps link capacity). Subsequently non-ADQ traffic packets for that application were generated and latency and max throughput was measured. This was then repeated with traffic control configured, to give a ratio of non-ADQ based traffic and ADQ based traffic queue pools. The latency and max throughput were measured across three runs. Max throughput was determined when the actual number of requests per second handled by the Memcached server fell below a requests per second threshold of 5%.

These scenarios were run on two 3rd Gen Intel® Xeon® Scalable processor-based platforms connected back-to-back with a 100Gb cable. The system under test was the worker node, which was 1 node, 2x Intel® Xeon® Platinum 8358 CPU @ 2.60GHz with configuration found in Appendix 1. Refer to Appendix 1 for additional set-up information.

Test Results

To measure the network connectivity performance, we focused on throughput and latency using the P999 and P9999 standards to examine the tail latency. Throughput refers to how much data can be moved from one location to another in set amount of time, measured by requests per second (RPS). Latency is the delay in communication response time, measured in microseconds (µsec). The P999 standard is the latency that 99.90 percent of the requests were completed. The P9999 standard is the latency that 99.99 percent of the requests were completed, and it is the best measurement of how well an organization is doing to meet the service level agreement (SLA) and quality of service (QoS) commitments.

Testing was performed with and without ADQ optimized (labelled as "non- ADQ"). The SLA was placed at 1000 microseconds for this example scenario. Results are shown in the next two charts.

Throughput and P999 Latency

The combination chart found in Figure 4 shows throughput and P999 latency. Throughput (bars) illustrate requested versus achieved requests per second (RPS). The ADQ optimized use case (blue bar) scales, achieving ~100K RPS, three times greater than non-ADQ. The non-ADQ use case (gold bars) plateaus at ~35K RPS (grey bars indicate less than desired RPS).

The P999 latency measurements show both non-ADQ (gold line) and ADQ optimized (navy blue line) use cases meet the

SLA (red line) and show only minor variation as the requests per second increase.

The test findings show that an organization could use fewer resources by using ADQ to save on infrastructure cost for the same amount of throughput RPS compared to non-ADQ. The non-ADQ use cases need to use more resources as the application scales (>35K - 100K RPS).

Also given the predictability of the latencies, software developers could get the added benefit of fast profiling characterization of the application stack.

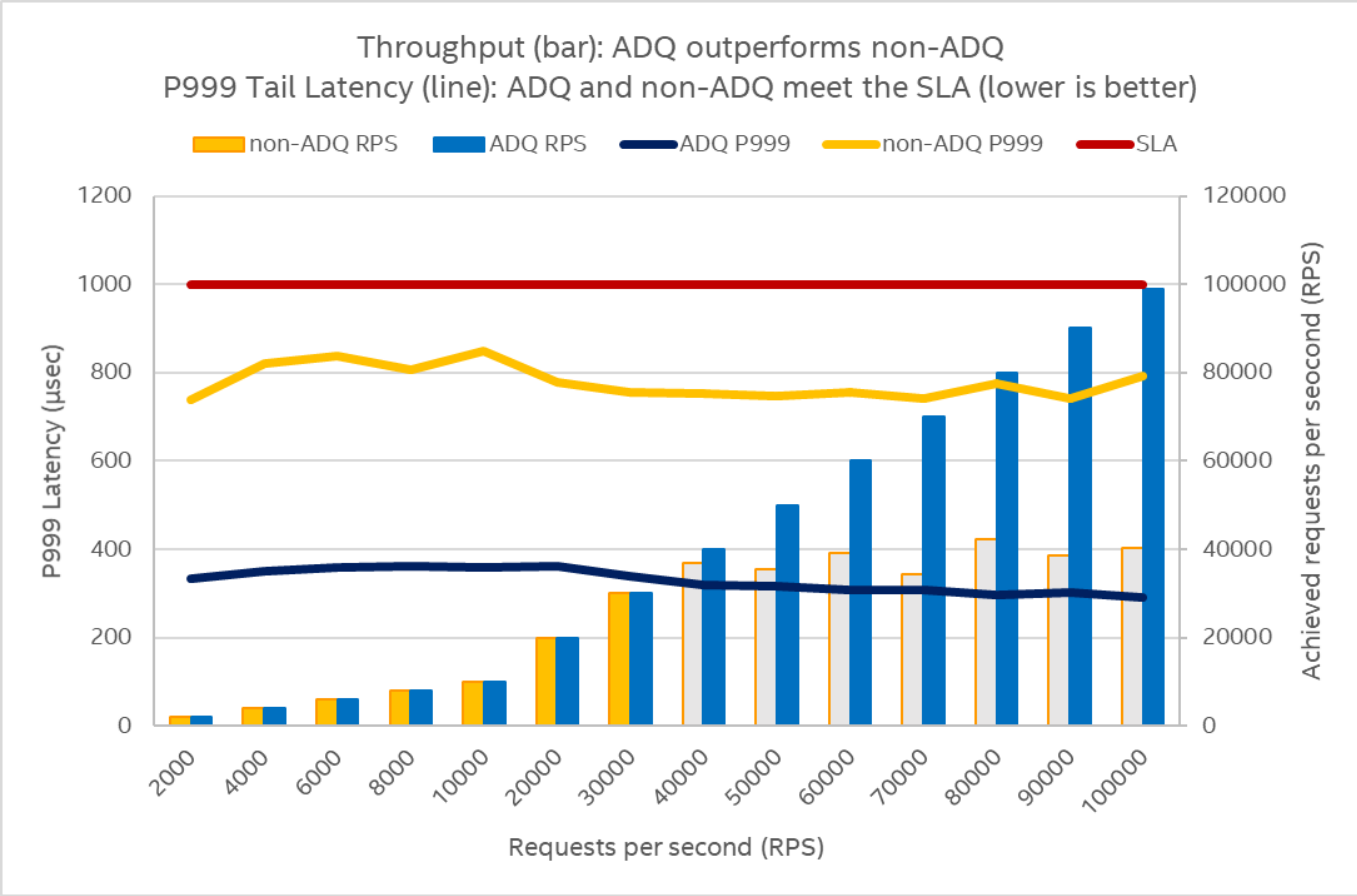

Figure 4. Throughput and P999 Tail Latency. Throughput (bars) illustrate requested vs. achieved requests per second (RPS). Latency (lines) illustrate the SLA compared to non-ADQ and ADQ. [RPC-PERF Memcached with 4 Clients 2 Pools, 3-run average with 75Gbps of background traffic].

P9999 Latency

The combination chart found in Figure 5 shows throughput and P9999 tail latency. Throughput (bars) illustrate requested versus achieved requests per second (RPS). Similar to the throughput results shown in Figure 4, the non-ADQ use case (gold bar) plateaus at ~35K RPS (grey bar). But the ADQ optimized use case (blue bar) scales, achieving ~100K RPS, three times greater than non-ADQ.

The P9999 latency measurements show non-ADQ use case (gold line) variation as the requests per sec increase and does not consistently meet the SLA. Conversely, the ADQ optimized use case (navy blue line) consistently meets the SLA even as the requests per second increase.

Based on these test findings, by using ADQ an organization could use fewer resources to save on infrastructure costs for the same amount of throughput RPS compared to non-ADQ. Non-ADQ cost could also increase due to the unpredictability of the tail latencies as they breach the SLA. The organization would need to add more resources at the early stage (~3K RPS).

Also, software developers could get the added benefit of fast profiling characterization of the application stack due to the predictability of the latencies.

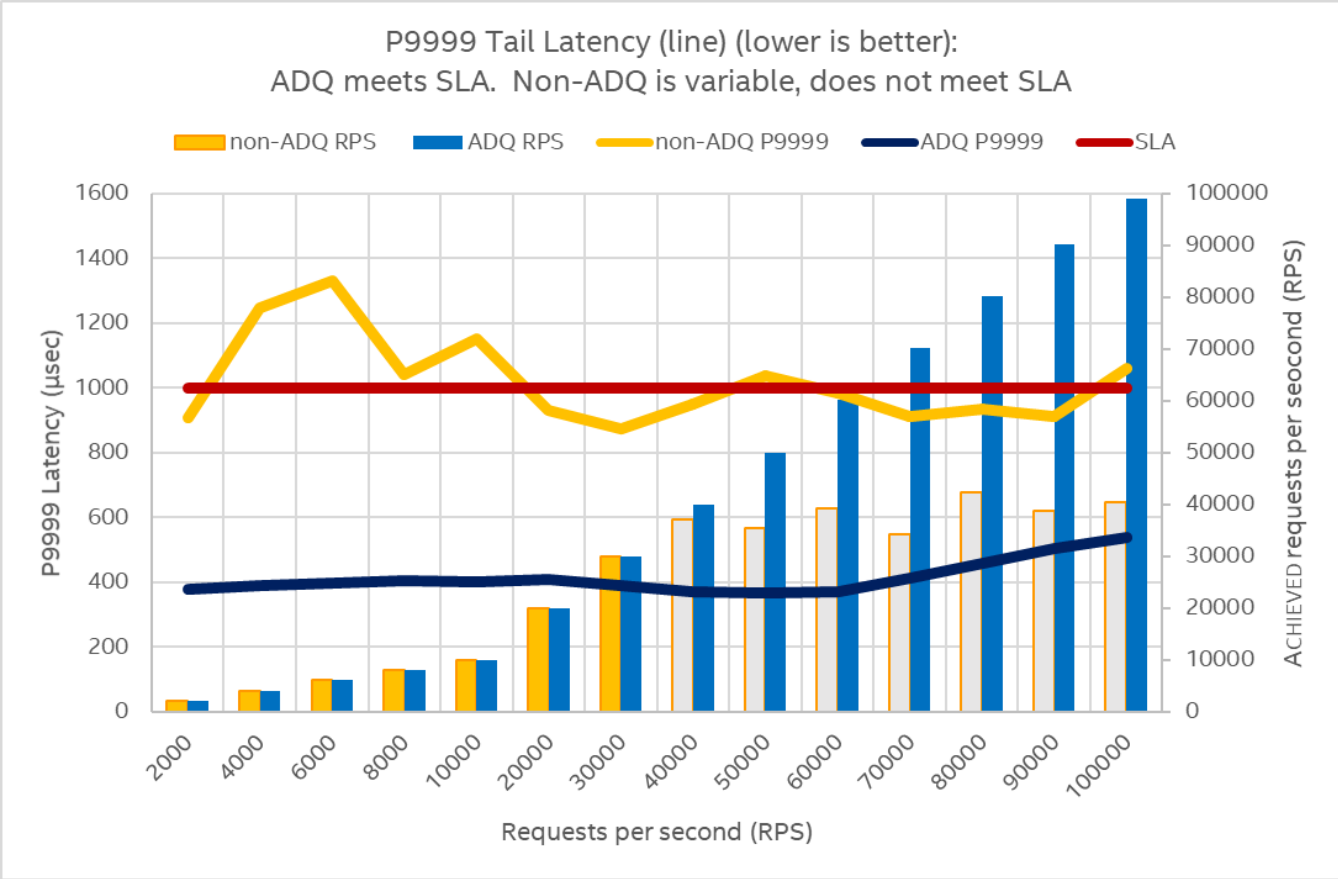

Figure 5. Throughput and P999 Tail Latency. Throughput (bars) illustrate requested vs. achieved requests per second (RPS). Latency (lines) illustrate the SLA compared to non-ADQ and ADQ. [RPC-PERF Memcached with 4 Clients 2 Pools, 3-run average with 75Gbps of background traffic].

Summary

- Managing latency, predictability and throughput on containerized workloads can be challenging.
- In this example scenario, testing confirms that ADQ improves containerized workload performance by reducing latency, improving predictability and throughput. By using ADQ, an organization could achieve P9999 standard and scale to achieve a 3x response per second (RPS):
  - Throughput: non-ADQ plateaus at ~35K RPS. ADQ scales, achieving ~100K RPS
  - P999 latency: both non-ADQ and ADQ meet the SLA and show only minor variation as the requests per second increase
  - P9999 latency: non-ADQ shows variation and does not consistently meet the SLA. Conversely, ADQ consistently meets the SLA even as the requests per second increase.
- As application utilization in Kubernetes orchestrated clusters increases and becomes more complex, latency (response time delays) can impact an organization's ability to consistently meet SLAs. By reducing latency and improving throughput, ADQ can help an organization to meet an SLA more easily.
- Containerized applications tend to have latency, which is variable and unpredictable. Variable and unpredictable container response times can be felt by end customers. ADQ is a technique that can be applied to improve predictability by reducing tail latency. With a higher level of predictability, an organization can meet QoS goals.
- Organizations can take advantage of ADQ benefits to help to improve their bottom line. Software developers can get the added benefit being able to optimize the application stack at a fast rate (better deterministic predictability on tail latencies).

References

Table 1. References

Reference	Source
E810 ADQ Configuration Guide	https://cdrdv2.intel.com/v1/dl/getContent/609008
Application Device Queues (ADQ) Resource Center	https://www.intel.com/content/www/us/en/architecture-and-technology/ethernet/adq-resource-center.html
Performance Testing Application Device Queues (ADQ) with Memcached	https://www.intel.com/content/www/us/en/architecture-and-technology/ethernet/performance-testing-application-device-queues-with-memcached.html
Intel Presents at Networking Field Day 21: Increasing Predictability at Scale Using Intel Ethernet 800 Series with ADQ	https://www.youtube.com/watch?v=E9Cu5oT04gE
Video Guide to Intel® Ethernet Application Device Queues (ADQ)	https://www.intel.com/content/www/us/en/support/articles/000088703/ethernet-products/800-series-network-adapters-up-to-100gbe.html
Faster, More Predictable Ethernet with the Intel® Ethernet 800 Series with Application Device Queues (ADQ)	https://www.intel.com/content/www/us/en/architecture-and-technology/ethernet/application-device-queues-technology-brief.html
tc(8) — Linux ref manual page for traffic control	https://man7.org/linux/man-pages/man8/tc.8.html#:~:text=qdisc%20is%20short%20for%20'queueing,qdisc%20configured%20for%20that%20interface
Linux Kernel Information for network priority cgroups	https://www.kernel.org/doc/Documentation/admin-guide/cgroup-v1/net_prio.rst

Learn more about ADQ

Download the ADQ Configuration Guide.

Visit the ADQ Resource Center at intel.com/ADQ.

Contact your Intel sales representative or distributor for more details about Intel® Ethernet 800 Series Network Adapters with ADQ.

Appendix 1: Test Configuration

	ITEM	DESCRIPTION
	Product	3rd Generation Intel® Xeon® Platinum 8358 Processor
	Platform	Intel M50CYP2SBSTD Server
	Sockets	2
	Speed (MHz)	2600
	No of Cores	128
	Stepping	6
System DDR Memory	Vendor	Samsung
	Type	DDR4
	Part Number	M393A2K43DB3-CWE
	Total Memory	256 GB
	Slots	16
	Capacity	16 GB
	Run Speed	3200
Storage – Boot	Type	Intel SSD
	Size	480 GB
BIOS	Vendor	Intel Corporation
	Date	04/26/2021
	Version	SE5C620.86B.01.01.0003.2104260124
	Microcode	0xd000363
	Turbo	Off
	HT	On
Operating System	OS Version	CentOS Stream 8
	Kernel Version	5.13.13
Test Software	Memcached Docker image	1.6.10
	rpc-perf image	3.2.0
	Kubernetes	1.24.2
	Container Runtime: containerd	1.6.8
	Cilium	1.12.0
Ethernet Network	2 x Intel Ethernet Network Adapters	E810-CQDA2 8.0 GT/s PCIe x16 link
	Intel Ethernet Controller firmware	4.00 0x800117e9 1.3236.0
	Cable connectivity	100 Gb

	Ice driver	1.9.11 COMPILED with ADQ flag. ADQ statistics enabled.
	ADQ Script Parameters	EgressMode: skbedit FilterPrio 1 Default Queue Set 16 queues [globals] arpfilter = false busypoll = 0 busyread = 0 cpus = 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 dev = ens801f0 numa = all queues = 16 rxadapt = false rxusecs = 50 txadapt = false txusecs = 50 ADQ Queue Set 4 Queues [adqTC3] cpus = 19 mode = exclusive numa = all poller_timeout = 10000 pollers = 1 protocol = tcp queues = 4 tc qdisc show information for ens801f0 ens801f0_tc_qdisc_entry.1=qdisc mqprio 8004: root tc 6 map 0 1 2 3 4 5 0 0 0 0 0 0 0 0 0 0 ens801f0_tc_qdisc_entry.2=queues:(0:15) (16:19) (20:23) (24:27) (28:31) (32:63) ens801f0_tc_qdisc_entry.3=mode:channel ens801f0_tc_qdisc_entry.4=shaper:dcb tuned profile: throughput-performance irqbalance disabled

Test Parameters	rpc-perf and memcached	kubectl exec -it memcached-bench-adq -n adqb -- /bin/bash -c "rpc-perf --endpoint memcached-adq.memcached-servers:11211 --interval 30 --windows 4 --clients 4 --poolsize 2 --request-ratelimit ${request_rl} --connect-ratelimit 100 --config /etc/rpc-perf/config/memcached-benchmark.conf"
		
		kubectl exec -it memcached-bench-noadq -n adqb -- /bin/bash -c "rpc-perf --endpoint memcached-noadq.memcached-servers:11211 --interval 30 --windows 4 --clients 4 --poolsize 2 --request-ratelimit ${request_rl} --connect-ratelimit 100 --config /etc/rpc-perf/config/memcached-benchmark.conf”
		
		memcached-benchmark.conf:
		---- 	
		[general]
		protocol = "memcache"
		interval = 30 # seconds
		windows = 4 # run for 4 intervals
		clients = 4 # use 4 client thread
		poolsize = 2 # each client has 2 connection per endpoint
		tcp_nodelay = false # do not enable tcp_nodelay

```
request_timeout = 1_000_000 # microseconds
connect_timeout = 1_000_000 # microseconds

[[keyspace]]
length = 8 # 8 byte keys
count = 500_000 # limit to 100K keys
weight = 1 # this keyspace has a weight of 1
commands = [ # get:set ratio is 1:1
  {action = "get", weight = 4},
  {action = "set", weight = 1},
]
values = [ # value length will always be 64 bytes
  {length = 64, weight = 1},
]
```

Document Revision History

Revision	Date	Description
001	April 2022	Initial release.
002	June 2022	Updated charts and diagrams for clarity. Corrected links in references. Title changed to follow naming convention.
003	Sept. 2022	Text updated to reflect new ADQ version 2.0 functionality for containerized applications. Test configuration, results, conclusion updated to reflect new round of testing done with ADQ version 2.0 using ice driver 1.9.11.

0922/DN/WIPRO/PDF 715543-003US