Solution Brief

Application Device Queues (ADQ) - Plugins for Kubernetes

Application Device Queues (ADQ) is an open technology designed to improve application specific queuing and steering. ADQ addresses issues of predictability, latency, and throughput for scale out of applications in a cloud native environment.

Authors

Eoghan Russell

Niamh Hennigan

Keith Cullen

Richard Walsh

Gershon Schatzberg

Executive Summary

ADQ is a workload optimization feature that provides access to hardware Quality-of-Service (QoS) in terms of queuing and steering for traffic flows belonging to containerized applications in a cloud orchestration environment. This document describes Application Device Queues (ADQ) plugins for Kubernetes.

This document is related to [Intel® Ethernet 800 Series Application Device Queues (ADQ) in a Kubernetes Environment Solution Brief](), which is part of the [Network Transformation Experience Kits]().

Overview

Kubernetes and Application Device Queues

Container orchestration manages the lifecycle of containers. Kubernetes provides the following features:

- Organizational primitives to query and group containers
- Scheduling to assign containers to run on hosts
- Automated health checks to relaunch containers if necessary
- Autoscaling to increase or decrease the number of containers handling a workload to meet demand
- Upgrade strategies to perform rolling updates
- Service discovery to determine which host a scheduled container is running on
- Abstracts hardware
- Provisioning and deployment of containers
- Allocation of resources
- Manage redundancy and availability of containers
    - Health monitoring of containers and hosts
    - Movement of containers from one host to another if there is a shortage of resources in a host, or if a host dies
- Load balancing and external exposure of services running in a container with the outside world

Kubernetes is designed to handle large workloads with many nodes in an efficient manner. As the cluster size increases, so too does the complexity of the networking configuration. Complex networking configuration leads to decreased visibility and the need for increased control on the networking protocols to maintain consistent quality of service. Multi-tenancy issues can also occur when multiple workloads share resources. Communication resources must be carefully monitored and managed to ensure that optimal service is provided to the users. This further highlights the need for advanced packet steering in the Kubernetes environment such as that provided by ADQ.

ADQ in a Kubernetes Environment

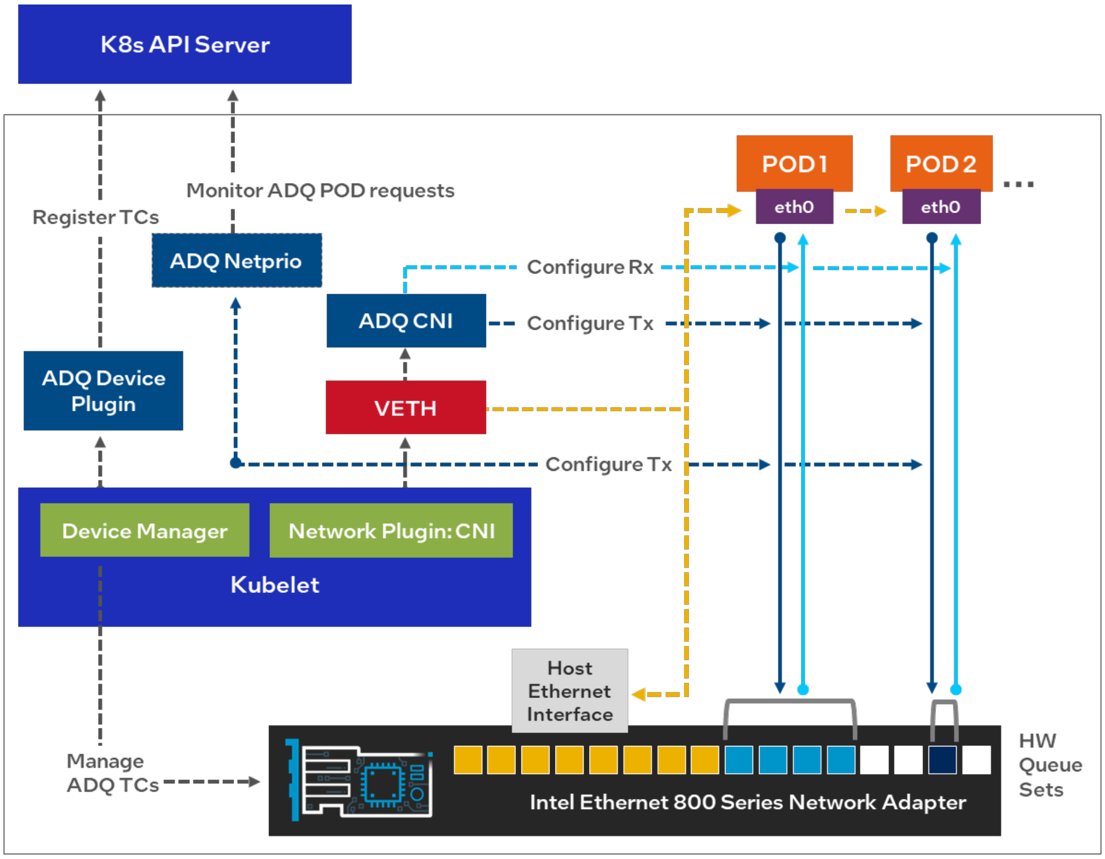

Figure 1. ADQ in a Kubernetes Environment

Devices supported by ADQ Plugins

The ADQ solution is supported on the following series of network adapters:

- Intel® Ethernet Network Adapter E810-CQDA1/CQDA2
- Intel® Ethernet Network Adapter E810-XXVDA4
- Intel® Ethernet Network Adapter E810-XXVDA2

Technologies Implemented

ADQ Plugins

Device Plugin

The 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 containers. The ADQ Device Plugin advertises the 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.

CNI Plugin

The ADQ Container Network Interface (CNI) plugin, is a chained plugin that is invoked on the creation and deletion of a pod. It leverages functionality of existing (deployed) CNIs for connectivity to the pod and configures the network interface to the pod. Linux Traffic Control (TC) infrastructure enables the creation of rules to control traffic flowing into and out of the Linux kernel. ADQ utilizes these TC rules to map hardware queues to software. It also implements filters that allow the packets from different applications to reach their intended queue set. The ADQ CNI configures the container network interface to be connected to the relevant queue set and assigns the associated filters for that queue set.

Netprio Plugin

In the Netprio plugin approach, to set the network priority for outgoing traffic, the container ID must be known. 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 determine if any are requesting an ADQ resource. If they are, 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, Netprio will get the container ID and set the priority of egress traffic on a given interface.

The ADQ Netprio plugin functionality is required if the CNI in use does not support netlink tc to set egress traffic priority. Otherwise, the ADQ CNI can set the egress traffic priority using skbedit actions as well as setting the ingress rules. Skbedit is the default mode of setting the priority for egress traffic. ADQ Netprio was included to allow for a wider range of support for CNI versions. To switch between the egress traffic configuration modes, the parameter EgressMode can be changed in the adq-cluster-config.yaml.

TC/Queue Configuration

ADQ allows the user to configure the number of traffic classes and the number of queues per traffic class. These details are passed to Kubernetes on ADQ pod creation. The default configuration has six traffic classes (TC0 – TC5). This first traffic class handles non ADQ, best-effort traffic and has 16 queues. The next four traffic classes can be assigned exclusively to individual containers and have four queues each. The final traffic class has 32 queues. It handles ADQ shared traffic and can be shared between containers. This configuration is set in the adq-cluster-config.yaml file. An ADQ setup Python tool then reads these values and sets up the hardware queues accordingly.

To show the filters set on your system run the following command. Replace `ens801f0` with the interface name used in the ADQ setup. This shows the filters set when no ADQ enabled workload is deployed.

```
tc filter show dev ens801f0 ingress && tc qdisc show dev ens801f0 | head
filter protocol all pref 99 bpf chain 0
filter protocol all pref 99 bpf chain 0 handle 0x1 bpf_netdev_ens801f0.o:[from-netdev] direct-
action not_in_hw id 7678 tag d4ea9e65878013b3 jited
qdisc mqprio 8004: root tc 6 map 0 1 2 3 4 5 0 0 0 0 0 0 0 0 0 0
             queues:(0:15) (16:19) (20:23) (24:27) (28:31) (32:63)
             mode:channel
             shaper:dcb
qdisc fq_codel 0: parent 8004:40 limit 10240p flows 1024 quantum 1514 target 5ms interval 100ms
memory_limit 32Mb ecn drop_batch 64
qdisc fq_codel 0: parent 8004:3f limit 10240p flows 1024 quantum 1514 target 5ms interval 100ms
memory_limit 32Mb ecn drop_batch 64
```

Independent poller Configuration

Independent poller (ipoller) is a driver-level operating mode. It allows a single NAPI poller running in a kthread context to service multiple queues. One independent poller can handle traffic for multiple containers. The ipoller configuration consists of the number of pollers per traffic class and a set of timeout values. This information is passed to Kubernetes upon ADQ pod creation. This is also set in the adq-cluster-config.yaml file.

CNIs

ADQ provides support for both Calico and Cilium as the container network interface of the cluster in a chained manner with the ADQ CNI. The supported networking modes include VXLAN and VETH. If necessary, there are installation scripts provided to allow for easy installation and swapping between these CNIs.

Prometheus/Grafana

Prometheus is a monitoring solution for collecting time series data. Grafana is a web-based tool for visualizing the data collected by Prometheus. ADQ plugins use these tools to gather and display metrics on throughput for traffic flows. It allows the user to verify that the traffic is in fact being filtered through the correct traffic class. This feature can be used to demonstrate and verify that the system is set up correctly and the traffic is in fact being filtered through the correct traffic class for ADQ enabled workloads.

Deployment

In order to deploy an ADQ enabled workload, the Kubernetes cluster must be configured as detailed below, specifically by deploying both the CNI and Device Plugin. With this implementation, there is no need to alter the logic of the application to be able to benefit from ADQ. The containerized application simply has to request an appropriate ADQ resource within the pod specification yaml.

Prerequisites

Linux based operating system with 5.12+ kernel recommended for best supportability

- Select Intel® Ethernet 800 Series Network Adapters with firmware 4.0+
- Intel ice driver version 1.9.11+ with ADQ flag set

make -j$(nproc) CFLAGS_EXTRA='-DADQ_PERF_COUNTERS' install

Applying Cluster Configuration

Once the configuration of the cluster in terms of the egress mode, tc/queue configuration and ipoller configuration has been completed, apply the adq-cluster-config.yaml.

```
kubectl apply -f deploy/k8s/adq-cluster-config.yaml
```

Building the ADQ Images

To deploy ADQ, the images must be built from source code.

- IMAGE_REGISTRY is the address of the registry where the images should be pushed, that is, my.private.registry.com
- IMAGE_VERSION is the version reference that you would like to apply to the image, that is, 22.06

```
make docker-build IMAGE_REGISTRY=<YOUR_REGISTRY>/ IMAGE_VERSION=<TAG>
make docker-push IMAGE_REGISTRY=<YOUR_REGISTRY>/ IMAGE_VERSION=<TAG>
```

Deploying ADQ

Edit the file adq-cni-dp-ds.yaml to reflect the updated values for IMAGE_REGISTRY, IMAGE_VERSION, and TAG.

```
kubectl apply -f deploy/k8s/adq-cni-dp-ds.yaml
```

Summary

Managing latency, predictability and throughput on containerized workloads can be challenging. Organizations can improve containerized workload performance by adopting Application Device Queues and Intel® Ethernet 800 Series Network Adapters. To learn more on the potential performance benefits associated with ADQ, refer to the two documents in the References section that carry out benchmarking. The ADQ solution for containers running in a Kubernetes environment uses the plugins detailed in this document.

Terminology

Table 1. Terminology

Abbreviation	Description
ADQ	Application Device Queues
CNI	Container Network Interface
CR	Custom Resource
CRD	Custom Resource Definition
NAPI	New Application Programming Interface
QoS	Quality-of-Service
TC	Traffic Class
VETH	Virtual Ethernet

References

Table 2. References

Reference	Source
Intel ADQ K8s Plugin GitHub	https://github.com/intel/adq-k8s-plugins
Intel® Ethernet 800 Series - Application Device Queues (ADQ) in a Kubernetes Environment Solution Brief	https://networkbuilders.intel.com/solutionslibrary/intel-ethernet-800-series-application-device-queues-kubernetes-env-solution-brief
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

Document Revision History

Revision	Date	Description
001	January 2023	Initial release.

0123/DN/WIT/PDF 764911-001US