Intel® Ethernet Controller E810 Application Device Queues (ADQ)

Configuration Guide

NEX Cloud Networking Group (NCNG)

Rev. 2.8

March 2023

Doc. No.: 609008, Rev.: 2.8

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

This document (and any related software) is Intel copyrighted material, and your use is governed by the express license under which it is provided to you. Unless the license provides otherwise, you may not use, modify, copy, publish, distribute, disclose or transmit this document (and related materials) without Intel's prior written permission. This document (and related materials) is provided as is, with no express or implied warranties, other than those that are expressly stated in the license.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors which may cause deviations from published specifications.

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

Other names and brands may be claimed as the property of others.

Copyright © 2020–2023, Intel Corporation. All rights reserved.

Did this document help answer your questions?

Contents

Revision History.....5

1.0 Fixed and Known Issues - Read First..... 10

1.1 Fixed Issues.....10

1.1.1 ADQ PF Issues.....10

1.1.2 ADQ Setup Script Issues.....10

1.1.3 ADQ VF Issues.....11

1.2 Known Issues.....11

1.2.1 ADQ PF Known Issues.....11

1.2.2 ADQ Setup Script Known Issues.....13

1.2.3 ADQ VF Known Issues..... 13

2.0 Introduction.....14

2.1 ADQ Overview.....14

2.2 ADQ Reference..... 16

3.0 ADQ Requirements.....17

3.1 Hardware Requirements.....17

3.2 Software Requirements.....17

4.0 ADQ System Under Test (SUT) Installation..... 19

4.1 Install OS..... 19

4.2 Update Kernel (If Needed)..... 21

4.3 Install *ice* Driver with ADQ Flags..... 23

4.4 Install iproute2 (Update If Needed)..... 24

4.5 Install cgroup Packages (If Needed)..... 25

5.0 General System Tuning..... 26

6.0 ADQ Setup Using ADQ Setup Script..... 31

6.1 ADQ Setup Script Prerequisites.....31

6.2 ADQ Setup Script Installation..... 33

6.3 ADQ Setup Script Usage..... 33

7.0 ADQ Configuration on SR-IOV Virtual Functions.....37

7.1 General SUT Configuration for VFs.....37

7.2 SUT Configuration Bare Metal VF.....39

7.3 SUT Configuration VF Inside a VM..... 41

7.4 ADQ VF Configuration.....43

8.0 Troubleshooting.....45

9.0 Testing ADQ with Netperf..... 46

9.1 Installation and Configuration – Both Systems..... 46

9.2 netperf Server.....48

9.3 netperf Client..... 50

10.0 Testing ADQ with Redis.....53

10.1 Redis Server Setup.....53

10.2 Redis Clients (non-ADQ) Configuration..... 56

Did this document help answer your questions?

11.0 Testing ADQ with Memcached/rpc-perf 60
11.1 Memcached Server Setup 60
11.2 Memcached/rpc-perf Clients (non-ADQ) Configuration 65
12.0 Testing ADQ with NGINX 67
12.1 NGINX Server Setup 67
12.2 wrk Clients Configuration 74
13.0 Testing ADQ with AF_XDP 76
13.1 AF_XDP Server Configuration 77
13.2 Traffic Generator Example Configuration (Non-ADQ) 81
13.2.1 Traffic Pattern to Be Generated 81
13.2.2 Example: Running DPDK-pktgen Traffic 81
14.0 Testing ADQ with Apache Traffic Server (ATS) 86
14.1 ATS Server Setup 86
14.2 Vegeta Clients Configuration 94
15.0 Testing ADQ with VirtIO 96
15.1 Installation and Configuration - Both Systems 96
15.2 Run the Traffic 103
Appendix A ADQ Configuration Reference (Manual) 104
A.1 ADQ Configuration on SUT 104
A.1.1 Adapter Preparation 104
A.1.2 Configure ADQ Traffic Class (TC) on SUT 107
A.1.3 Independent Pollers (Optional) 115
A.1.4 Apply Adapter Tuning 121
A.1.5 Configure Symmetric Queues 122
A.1.6 Configure Intel® Ethernet Flow Director Settings 123
A.1.7 Set cgroup Priority for Application Network Traffic (If Needed) 125
A.1.8 Verify ADQ Application Traffic and Independent Pollers (If Applicable) 129
A.1.9 Clear the ADQ Configuration 130

Did this document help answer your questions?

Revision History

Revision	Date	Comments
2.8	March 20, 2023	Updates include the following: • Renamed Fixed and Known Issues - Read First. • Added Fixed Issues. • Added ADQ PF Known Issues. • Renamed ADQ Setup Script Known Issues. • Renamed and updated ADQ VF Known Issues. • Updated Introduction. • Updated Software Requirements. • Updated Install OS. • Updated Update Kernel (If Needed). • Updated Install cgroup Packages (If Needed). • Renamed and moved ADQ Setup Using ADQ Setup Script to chapter level. Updated content including new subsections. • Updated ADQ Configuration on SR-IOV Virtual Functions. • Updated ADQ VF Configuration. • Updated Testing ADQ with Netperf. • Updated Installation and Configuration – Both Systems. • Updated netperf Server. • Updated netperf Client. • Updated Testing ADQ with Redis. • Updated Redis Server Setup. • Updated Redis Clients (non-ADQ) Configuration. • Updated Testing ADQ with Memcached/rpc-perf. • Updated Memcached Server Setup. • Updated Memcached/rpc-perf Clients (non-ADQ) Configuration. • Updated Testing ADQ with NGINX. • Updated NGINX Server Setup. • Updated Testing ADQ with AF_XDP. • Updated NGINX Server Setup. • Updated Example: Running DPDK-pktgen Traffic. • Updated ATS Server Setup. • Moved ADQ Configuration on SUT and subsections to Appendix. • Removed Appendix "TC Reference".
2.7	November 2, 2022	Updates include the following: • Updated "Known Issues ADQ Configuration Script". • Updated Introduction. • Updated Hardware Requirements. • Updated Software Requirements. • Updated Install OS. • Updated Update Kernel (If Needed). • Updated Install ice Driver with ADQ Flags. • Updated General System Tuning. • Updated ADQ Configuration on SUT. • Updated Adapter Preparation.

continued...

Did this document help answer your questions?

Revision	Date	Comments
		• Updated Create TCs. • Updated TC Filtering. • Updated Configure Independent Pollers. • Updated Set cgroup Priority for Application Network Traffic (If Needed). • Updated Clear the ADQ Configuration. • Updated ADQ Setup Script Usage. • Updated ADQ VF Configuration. • Updated Troubleshooting. • Updated netperf Server. • Updated netperf Client. • Updated Redis Server Setup. • Updated Redis Clients (non-ADQ) Configuration. • Updated Memcached Server Setup. • Updated Memcached/rpc-perf Clients (non-ADQ) Configuration. • Updated NGINX Server Setup. • Updated wrk Clients Configuration. • Updated AF_XDP Server Configuration. • Added section, Testing ADQ with Apache Traffic Server (ATS). • Updated appendix, "Clear the ADQ Configuration".
2.6	July 29, 2022	Updates include the following: • Updated "Known Issues". • Added new section, "Known Issues ADQ Configuration Script. • Updated "ADQ Overview". • Updated "Software Requirements". • Updated "Install OS". • Updated "Update Kernel (If Needed)". • Updated "Install iproute2 (Update If Needed)". • Updated "Install cgroup Packages (If Needed)". • Moved "General System Tuning". • Updated "ADQ Configuration on SUT". • Moved "Adapter Preparation". • Updated "Configure ADQ Traffic Class (TC) on SUT". • Updated "Create TCs". • Updated "TC Filtering". • Updated "Confirm TC Configuration". • Added new section, "Independent Pollers (Optional)". • Updated "Apply Adapter Tuning". • Updated "Configure Intel Ethernet Flow Director Settings". • Updated "Set cgroup Priority for Application Network Traffic (If Needed)". • Updated "Verify ADQ Application Traffic and Independent Pollers (If Applicable)". • Updated "Clear the ADQ Configuration". • Added new section, "ADQ Setup Using ADQ Configuration Script". • Updated "Testing ADQ with Netperf". • Updated "Installation and Configuration – Both Systems". • Updated "netperf Server". • Updated "netperf Client". • Updated "Testing ADQ with Redis". • Updated "Redis Server Setup". • Updated "Testing ADQ with Memcached/rpc-perf". • Updated "Memcached Server Setup".

continued...

Did this document help answer your questions?

Revision	Date	Comments
		• Updated "Testing ADQ with NGINX". • Updated "Testing ADQ with AF_XDP". • Updated "AF_XDP Server Configuration". • Updated "TC Creation on Interface". • Added new section, "TCP: Source IP + L4 Source Port". • Updated "View All Filters". • Updated "Clear the ADQ Configuration".
2.5	June 15, 2022	Updates include the following: • Updated Section 1.0, “Known Issues - Read First”. • Updated Section 4.1, “Install OS”. • Updated Section 4.2, “Update Kernel (If Needed)”. • Updated Section 5.1, “General System Tuning”. • Updated Section 5.2, “Adapter Preparation”. • Updated Section 6.1, “Configure ADQ Traffic Class (TC) on SUT”. • Updated Section 6.1.2, “TC Filtering” • Updated Section 6.4, “Configure Intel® Ethernet Flow Director Settings”. • Updated Section 6.5, “Set cgroup Priority for Application Network Traffic”. • Updated Section 6.7, “Clear the ADQ Configuration”. • Updated Section 7.2, “SUT Configuration Bare Metal VF”. • Updated Section 7.3, “SUT Configuration VF Inside a VM”. • Updated Section 9.2, “netperf Server”. • Updated Section 9.3, “netperf Client”. • Updated Section 10.1, “Redis Server Setup”. • Updated Section 11.1, “Memcached Server Setup”. • Updated Section 12.1, “NGINX Server Setup”. • Updated Section 13.1, “AF_XDP Server Configuration”. • Updated Section A.1, “TC Creation on Interface”. • Updated Section A.2.1, “Definitions”. • Updated Section A.3.1, “Clear the ADQ Configuration”.
2.4	December 20, 2021	Updates include the following: • Updated Section 1.0, “Known Issues - Read First”. • Updated Section 3.2, “Software Requirements”. • Updated Section 4.2, “Update Kernel (If Needed)”. • Updated Section 5.2, “Adapter Preparation”. • Updated Section 6.7, “Clear the ADQ Configuration”. • Updated Section 7.2, “SUT Configuration Bare Metal VF”. • Updated Section 7.3, “SUT Configuration VF Inside a VM”. • Updated Section 8.0, “Troubleshooting”. • Updated Section 9.2, “netperf Server”. • Updated Section 9.3, “netperf Client”. • Updated Section 12.2, “wrk Clients Configuration”. • Added Section 13.0, “Testing ADQ with AF_XDP”. • Updated Section A.3.1, “Clear the ADQ Configuration”.
2.3	September 2, 2021	Updates include the following: • Updated Section 1.0, “Known Issues - Read First”. • Added Section 2.2, “ADQ Reference”. • Updated Section 4.2, “Update Kernel (If Needed)”. • Updated Section 4.3, “Install *ice* Driver with ADQ Flags”. • Updated Section 5.2, “Adapter Preparation”. • Updated Section 6.2, “Apply Adapter Tuning”.

continued...

Did this document help answer your questions?

Revision	Date	Comments
		• Updated Section 6.3, “Configure Symmetric Queues”. • Updated Section 6.4, “Configure Intel® Ethernet Flow Director Settings”. • Updated Section 6.5, “Set cgroup Priority for Application Network Traffic”. • Updated Section 6.6, “Verify ADQ Application Traffic”. • Updated Section 6.7, “Clear the ADQ Configuration”. • Updated Section 7.2, “SUT Configuration Bare Metal VF”. • Updated Section 7.3, “SUT Configuration VF Inside a VM”. • Added Section 8.0, “Troubleshooting”. • Updated Section 9.0, “Testing ADQ with Netperf”. • Updated Section 10.0, “Testing ADQ with Redis”. • Updated Section 11.0, “Testing ADQ with Memcached/rpc-perf”. • Updated Section 12.0, “Testing ADQ with NGINX”. • Updated Appendix A, “TC Reference”.
2.2	March 19, 2021	Updates include the following: • Updated Section 1.0, “Known Issues - Read First”. • Updated Section 3.2, “Software Requirements”. • Updated Section 4.1, “Install OS”. • Updated Section 4.2, “Update Kernel (If Needed)”. • Updated Section 4.4, “Install iproute2 (Update If Needed)”. • Updated Section 5.2, “Adapter Preparation”. • Updated Section 6.1.1, “Create TCs”. • Updated Section 6.1.2, “TC Filtering”. • Updated Section 6.1.2.1, “TC Filtering Requirements”. • Updated Section 6.1.3, “Confirm TC Configuration”. • Updated Section 6.4, “Configure Intel® Ethernet Flow Director Settings”. • Updated Section 6.5, “Set cgroup Priority for Application Network Traffic”. • Added Section 6.6, “Verify ADQ Application Traffic”. • Added Section 6.7, “Clear the ADQ Configuration”. • Updated Section 7.4, “ADQ VF Configuration”. • Updated Section 9.1, “Installation and Configuration - Both Systems”. • Updated Section 9.2, “netperf Server”. • Updated Section 9.3, “netperf Client”. • Updated Section 10.1, “Redis Server Setup”. • Updated Section 10.2, “Redis Clients (non-ADQ) Configuration”. • Updated Section 11.1, “Memcached Server Setup”. • Updated Section 11.2, “Memcached/rpc-perf Clients (non-ADQ) Configuration”. • Updated Section 12.1, “NGINX Server Setup”. • Updated Section A.1, “TC Creation on Interface”. • Updated Section A.1.1, “TC Minimum Rate Creation”. • Updated Section A.1.2, “TC Maximum Rate Creation”. • Updated Section A.1.3, “TC Minimum and Maximum Rate Creation”. • Updated Section A.2.2, “TCP: Dest IP + L4 Dest Port”. • Updated Section A.2.3, “TCP:DEST IP + L4 Source Port”. • Updated Section A.2.5, “View All Filters”. • Updated Section A.3.1, “Clear the ADQ Configuration”. • Added Section A.3, “Post-Testing Steps”.

continued...

Did this document help answer your questions?

Revision	Date	Comments
2.1	October 8, 2020	Updates include the following: • Updated Section 2.1, "ADQ Overview". • Updated Section 4.2, "Update Kernel (If Needed)". • Updated Section 5.2, "Adapter Preparation". • Updated Section 6.1.1, "Create TCs". • Updated Section 9.2, "netperf Server". • Updated Section 9.3, "netperf Client". • Updated Section 10.1, "Redis Server Setup". • Updated Section 10.2, "Redis Clients (non-ADQ) Configuration". • Updated Section 11.1, "Memcached Server Setup". • Updated Section 11.2, "Memcached/rpc-perf Clients (non-ADQ) Configuration". • Updated Section 12.1, "NGINX Server Setup".
2.0[1]	July 28, 2020	Initial public release.
Note: 1. There are no previous publicly-available versions of this document.		

Did this document help answer your questions?

1.0 Fixed and Known Issues - Read First

The code contains the following fixed and known issues.

1.1 Fixed Issues

1.1.1 ADQ PF Issues

- ice driver version 1.1.x can create unnecessary Linux kernel (kworker) threads, under certain workloads with an uncabled port on an E810 Network Interface Card (NIC) port. This can increase CPU utilization, which might impact ADQ performance.

  Mitigation: Use ice driver version 1.2.1 or later.
- VXLAN stateless offloads (checksum, TSO) and TC filters directing traffic to a VXLAN interface are not supported with Linux v5.9 or later, using older versions of the ice driver.

  Mitigation: Use ice driver version 1.9.7 or later.
- ADQ configuration commands that update settings on the interface (i.e., `tc qdisc add and ethtool -L`) cause the driver to close the associated RDMA interface and reopen it. This disrupts RDMA traffic on the interface. RDMA and ADQ configurations are not supported on the same interface simultaneously.
- When TCs are created, deleted, then recreated, Intel® Ethernet Flow Director might not work as expected. To recover, reload ice driver and reapply ADQ configuration.

  Mitigation: Use ice driver version 1.5.8 or later.
- The ice driver can return an error when trying to create TC filters using ipv6 filter rules.

  Mitigation: Use ice driver version 1.9.7 or later.

1.1.2 ADQ Setup Script Issues

None

Did this document help answer your questions?

1.1.3 ADQ VF Issues

- When ADQ is enabled on VFs, TC filters on the VF TC0 (default TC) are not supported and do not pass traffic. It is not expected to add TC filters to TC0 since it is reserved for non-filtered default traffic.
- When TC offload is disabled on a VF interface, the *iavf* driver allows ADQ configuration steps to add TCs and TC filters to the VF interface. Expected behavior is for the iavf driver to return an error message if hw-tc-offload is set to **off**.

  Mitigation: Enable the hw-tc-offload flag on the VF interface:

  ```
  ethtool -K $vf_iface hw-tc-offload on
  ```

- When ADQ is enabled on a VF interface, the **tc qdisc add** command causes the VF connection (ping) to drop when using *ice-1.8.x* and *iavf-4.4.x*

  Mitigation: Use previous versions of the *ice* and *iavf* drivers for ADQ VF (*ice-1.7.16* with *iavf-4.3.19*).
  .
- Configuring ADQ traffic classes with an odd number of HW queues on a VF interface might result in system hang in the *iavf* driver.

  Mitigation: Configure even number of queues in the **tc qdisc add dev** command.
- TC VF filters do not correctly handle ADQ filter queues, causing ADQ SR-IOV application traffic to be directed to hardware queue(s) in the default traffic class (TC0).

  Mitigation: Use *ice* driver version 1.6.4 or later.

1.2 Known Issues

1.2.1 ADQ PF Known Issues

- Latest RHEL* and SLES* distributions have kernels with back-ported support for Application Device Queues (ADQ). For all other OS distributions, you must use LTS Linux kernel v4.19.58 or higher to use ADQ. The latest out-of-tree driver is required for ADQ on all operating systems. For more details, see Software Requirements and Install OS.
- ADQ configuration must be cleared following the steps in Clear the ADQ Configuration. The following issues might result if steps are not executed in the correct order:
  - Removing a Traffic Class (TC) qdisc prior to deleting a TC filter will cause the qdisc to be deleted from hardware and leave an unusable TC filter in software.
  - Deleting a ntuple rule after deleting the TC qdisc, then re-enabling ntuple, might leave the system in an unusable state that requires a forced reboot to clear.

  Mitigation: Follow the steps in Clear the ADQ Configuration.
- ADQ configuration is not supported on a bonded or teamed Intel® E810 Network adapter interface. Issuing the **ethtool** or **tc** commands to a bonded E810 interface will result in error messages from the *ice* driver to indicate the operation is not supported.

Did this document help answer your questions?

- If the application should stall for some reason when using application dependent polling, this can cause a queue stall for application-specific queues for up to two seconds.

  Mitigation: Recommend configuration of only one application per TC channel.

- DCB and ADQ are mutually exclusive and cannot coexist. A switch with DCB enabled might remove the ADQ configuration from the device.

  Mitigation: Do not enable DCB on the switch ports being used for ADQ. Disable LLDP on the interface by turning off firmware LLDP agent using:

  ```
  ethtool --set-priv-flags $iface fw-lldp-agent off
  ```

  Alternatively, use independent poller configuration that is not dependent on the application returning to drain the queues.

  Note (unrelated to Intel drivers): The 5.8.0 Linux kernel introduced a bug that broke the interrupt affinity setting mechanism.

  Mitigation: Use an earlier or later version of the kernel to avoid this error.

- Core-level reset of an ADQ-configured PF port (rare events usually triggered by other failures in the NIC/*ice* driver) results in loss of ADQ configuration. To recover, reapply ADQ configuration to the PF interface.
- When the number of queues is increased using **ethtool -L**, the new queues will have the same interrupt moderation settings as queue 0 (i.e., Tx queue 0 for new Tx queues and Rx queue 0 for new Rx queues). This can be changed using the **ethtool** per-queue coalesce commands.
- To fully release hardware resources and have all supported filter type combinations available, the *ice* driver must be unloaded and re-loaded.
- When using multiple TCs for ADQ application traffic, adding ntuple rules to the first queue of TC2 or higher does not work as expected. The **ethtool** ntuple rule fails for the first queue. This does not affect ntuple rules on TC1.
- TC filters can unexpectedly match packets which use IP protocols other than what is specified as the ip_proto argument in the **tc filter add** command. For example, UDP packets may be matched on a TCP TC filter created with ip_proto tcp without any L4 port matches.
- Creating more than 10k TC filters on an interface can result in errors talking to the kernel and the filters fail to get created. (The maximum number of supported TC filters is 32k.)
- TC filters do not function correctly when GTP encryption is enabled in the TC filter add command (enc_key_id 1).This causes traffic to go to default TC 0.

  Mitigation: Use ice driver version 1.11.x and later

Did this document help answer your questions?

1.2.2 ADQ Setup Script Known Issues

- ADQ independent pollers timeout argument passed to ADQ setup script (`poller-timeout`) does not get processed correctly by the script.

  Mitigation: Use **adqsetup** from ice driver version 1.10.x or later, or use `poller_timeout` (underscore instead of hyphen) as a workaround.

1.2.3 ADQ VF Known Issues

NOTE

Several of the issues listed above also apply to both PF and VF. In addition, the issues below are specific to VF.

- Core-level reset of an ADQ-configured VF port (rare events usually triggered by other failures in the NIC/*iavf* driver) results in loss of ADQ configuration. To recover, reapply ADQ configuration to the VF interface.
- *iavf* driver must use Trusted mode with ADQ: Trusted mode must be enabled for ADQ inside a VF. If TC filters are created on a VF interface with trusted mode off, the filters are added to the software table but are not offloaded to the hardware.
- VF supports Max Transmit Rate only: *iavf* driver only supports setting maximum transmit rate (max_rate) for Tx traffic. Minimum transmit rate (min_rate) setting is not supported with a VF.
- VF Max Transmit Rate: The **tc qdisc add** command on a VF interface does not verify that max_rate value(s) for the TCs are specified in increments of 500 Kb/s. TC max_rate is expected to be a multiple of (or equal to) 500 Kb/s.
- VF errors when deleting TCs or unloading *iavf* driver in a VF: *ice* and *iavf* driver error messages might get triggered in a VF when TCs are configured, and TCs are either manually deleted or the *iavf* driver is unloaded. Reloading the *ice* driver recovers the driver states.
- VF Max Transmit Rate: When a maximum TX transmit rate is specified in the **tc qdisc add** command on a VF interface, the max rate does not get applied correctly causing inconsistent TX rate limit for some applications.
- The system reboots/crashes when the number of queues on the VF interface is changed using the ethtool command while traffic is flowing from client to SUT, using ice driver version 1.10.X and iavf version 4.6.X.

  Mitigation: After the TCs have been created, add at least 2 seconds of sleep time.

Did this document help answer your questions?

2.0 Introduction

This document is designed to provide instructions for configuring and testing Intel® Ethernet Controller E810 with Application Device Queues (ADQ).

NOTE

This package contains licenses and patches to enable ADQ with applications listed in the Software Requirements section. If you are viewing this document in HTML, click the **Download PDF** button to download the zip package.

2.1 ADQ Overview

Intel® Ethernet 800 Series Network Adapters feature Application Device Queues (ADQ), an Intel technology designed to prioritize application traffic to deliver the performance required for high-priority, network intensive workloads. By performing application-specific queuing and steering, ADQ can improve response time predictability, reduce latency, and improve throughput for critical applications.

ADQ improves performance in the following ways:

- **Dedicated Resources = Increased Predictability**

  ADQ leverages multiple Intel hardware technologies to steer, load balance, and isolate incoming application traffic to a set of dedicated application queues. By preventing other traffic from contending for resources with a chosen application's traffic, performance becomes more predictable and less prone to jitter.

- **Efficient Packet Processing = Higher Throughput and Scalability**

  ADQ ensures that all packets from a queue are processed in one thread and are for the same application, which enables polling for data to be more performant. For example, the application might accept a few packets, choosing to process them sooner, rather than spending more time accepting many packets, and so taking longer before any processing starts. Queuing attributes can be customized based on an application's traffic profile, for example, significantly reducing the number of interrupts per second.

- **Customizable Traffic Shaping = Application Level QoS Control**

  ADQ allows for dedicated application traffic queues, to have custom QoS (for example: rate limits) applied to them, allowing traffic to be shaped per application instead of per class or VM.

- **Optimized Polling = Lower Latency**

  Application Dependent Polling Method: ADQ provides hints to the application for each application thread to service incoming and outgoing traffic from a single device queue. By doing so, a queue is drained in order by one thread, minimizing context switching, cache misses, and sub-optimal thread scheduling.

  Application Independent Polling Method (ADQ 2.0): In this mode, an independent poller drains the packets from the device queues and pushes them to socket queues associated with application threads. This avoids the need for

Did this document help answer your questions?

application threads to poll the device queues directly. As independent pollers can be used to poll multiple queues, a smaller number of pollers can service a large number of application threads and is more scalable. The poller threads minimize interrupts by polling the device queues, minimize context switches by pre-filling socket queues and avoiding the application threads from going through sleep/wakeup cycles. Although an application need not be modified to use the hints provided by ADQ to align the application threads to device queues in this mode, doing so can further improve the performance.

ADQ Versions 1.0 and 2.0

ADQ version 1.0 was released in 2019. It uses an application dependent (triggered) polling approach. ADQ version 2.0 offers an alternative to application-dependent polling where polling is triggered independent of the application.

1. ADQ 1.0 (ice driver version 1.8.x and earlier) contains all of the functional support and enhancements for ADQ acceleration. It uses application dependent (triggered) polling which normally requires the application vendor to make a small code change to provide a hint to the application for optimal application thread to NIC queue alignment. In addition to establishing the application threads with queue connections, busy polling is also triggered and managed by the application threads. This document will refer to this method as **application dependent polling**.
2. ADQ 2.0 (ice driver version 1.9.x and later) supports all of the ADQ enhancements included in previous driver versions and introduces a new *independent* polling method that does not require the application vendor to make a code change. ADQ 2.0 introduces a new *independent* polling method which does not require the application vendor to make a code change. ADQ 2.0 enables application agnostic ADQ acceleration, where the queueing and steering for ADQ workloads is done independently from the application threads. This approach enables ADQ acceleration to be configured for broader range of cloud to edge applications and environments. This document will refer to this method as **application independent polling**. See Independent Pollers (Optional) for more details.
3. Other new capabilities released in ADQ 2.0 include:

ADQ 2.0 Feature	Description
Application Independent Polling	Polling triggered independent of the application. Queueing and steering is done independently from the application threads.
ADQ Setup Script (applies to ADQ 1.0 and 2.0)	Provides a user-friendly method to input configuration details. Script generates and executes commands in the back end. Debug mode and command logging options, and option to make ADQ configuration persistent across reboots.
New and enhanced steering modes: TC ingress & egress forward to queue TC egress set priority Large TC (256 queues), shared queue set	Enables ADQ in more environments (TC limited, KVM based VMs, K8s pods)
Kubernetes Plugins (Applies to ADQ2.0)	Allows ADQ acceleration for applications running in K8s pods. ADQ container orchestration with device plugins and CNI chaining.
Acceleration for VirtIO-net based VMs (Applies to ADQ2.0)	ADQ provides dedicated HW queues and enhanced polling capability to VMs without direct SR-IOV passthrough interfaces.

Did this document help answer your questions?

2.2 ADQ Reference

- ADQ Resource Center

  https://www.intel.com/content/www/us/en/architecture-and-technology/ethernet/adq-resource-center.html

- ADQ Training Videos
  - Intel® Ethernet 800 Series Training (1 of 3):

    Introduction to Application Device Queues (ADQ) Video

    https://www.intel.com/content/www/us/en/design/products-and-solutions/networking-and-io/ethernet-800-series/introduction-to-application-device-queues.html

  - Intel® Ethernet 800 Series Training (2 of 3):

    How ADQ Improves Performance with Application Device Queues (ADQ) Video

    https://www.intel.com/content/www/us/en/design/products-and-solutions/networking-and-io/ethernet-800-series/ada-improves-performance-application-device-queues.html

  - Intel® Ethernet 800 Series Training (3 of 3):

    Requirements and Capabilities of Application Device Queues (ADQ) Video

    https://www.intel.com/content/www/us/en/design/products-and-solutions/networking-and-io/ethernet-800-series/requirements-capabilities-application-device-queues.html

- *NVM Express over TCP with SPDK for Intel® Ethernet Products with ADQ Configuration Guide*

  https://cdrdv2.intel.com/v1/dl/getContent/633368

- *NVM Express over TCP with Linux Kernel for Intel® Ethernet Products with ADQ Configuration Guide*

  https://cdrdv2.intel.com/v1/dl/getContent/691440

- *Principled Technologies Aerospike 3-node Scaling Performance Report*

  https://www.principledtechnologies.com/Dell/Intel-Ethernet-800-ADQ-PowerEdge-R740xd-0421.pdf

Did this document help answer your questions?

3.0 ADQ Requirements

3.1 Hardware Requirements

- System Under Test (SUT) – A server platform with an available PCI Express (PCIe) slot.
- Clients – Any comparable server platform to the SUT
- Device Under Test (DUT) is equipped with one or more Intel retail or OEM generic adapters such as; Intel® Ethernet 800 Series Network Adapters.

3.2 Software Requirements

- Operating System:
  - Refer to the Intel® Ethernet 800 Series *ice* driver documentation for the list of supported Linux-based operating systems for your driver version.
  - This document has examples in RHEL 8.0+, but others can be used with slight modifications.

    Refer to Install OS for a list of OS distributions with back-ported kernel support for ADQ.
- Kernel:
  - ADQ technology might require a kernel update. Refer to Install OS and Update Kernel (If Needed) for more details on OS and kernel requirements for ADQ. Refer to the OS vendor's documentation for the base kernel version of an OS.
- **iproute2-ss180813** or later, which is included in operating systems with inbox base kernels 4.17.19 or newer. If the OS has an older inbox base kernel (even if back-ported as above), an **iproute2** update may be required:

  https://git.kernel.org/pub/scm/network/iproute2/iproute2.git/

  NOTE

  This step is not required if the ADQ setup script is used to create an ADQ setup.

- The latest Linux *ice* driver (out of tree only - inbox drivers not supported):

  https://sourceforge.net/projects/e1000/files/ice%20stable/
- Linux *iavf* driver version 4.0.1 or later (if using Virtual Functions):

  https://sourceforge.net/projects/e1000/files/iavf%20stable/
- Applications (Note that other versions may be used but are out of scope of this document):
  - Redis version 4.0.11:

    http://download.redis.io/releases/redis-4.0.11.tar.gz

Did this document help answer your questions?

- The following Redis benchmark patch (included with this document): *redis-benchmark.patch*
- Memcached version 1.6.9: https://memcached.org/files/memcached-1.6.9.tar.gz
- Netperf version 2.7.0
- NGINX version 1.16.1 http://nginx.org/download/nginx-1.16.1.tar.gz
- The following NGINX ADQ enablement patch (included with this document): *nginx_reuseport_bpf.patch*
- ATS version 9.1.2 https://dlcdn.apache.org/trafficserver/trafficserver-9.1.2.tar.bz2

NOTE

If you are viewing this document in HTML, click the **Download PDF** button to download the zip package for the application specific patches listed above.

Did this document help answer your questions?

4.0 ADQ System Under Test (SUT) Installation

4.1 Install OS

Install any Intel® Ethernet 800 Series supported Linux-based operating system with software development packages. This document has examples for RHEL 8.0+, but others can be used.

ADQ technology support might require a kernel update. ADQ requires capabilities in the Linux kernel available in LTS Linux kernel. Refer to the OS vendor's documentation for the base kernel version of an OS. Refer to kernel and operating system requirements in this section for ADQ-specific kernel support.

NOTE

The *kernel-modules-extra* package is required for RHEL 8.0 or higher:

```
yum install -y kernel-modules-extra
```

ice-1.7.x and Earlier

- Linux kernel v4.19.x or higher.
- Inline Flow Director is configured using the *channel-inline-flow-director* **ethtool** private flag (set globally on the interface for all TCs).
- ADQ kernel support has also been back-ported to the following operating systems, which are based on earlier base kernel versions. These operating systems do not require a kernel update even though their base kernel version is lower than 4.19.58:
  - RHEL 8.0 or higher (inbox kernel)
  - SLES 12 SP5 or higher (inbox kernel)
  - SLES 15 SP1 or higher (inbox kernel)
  - Ubuntu 20.04 or higher (inbox kernel)

ice-1.8.x and Later

- Linux kernel v5.6 or higher, for devlink parameter support:

  https://www.kernel.org/doc/html/v5.6/networking/devlink/devlink-params.html
- Inline Flow Director is configured using per-TC inline flow director.

Did this document help answer your questions?

- The following operating systems contain kernels with devlink parameter support and do not require a kernel update even though their base kernel version may be lower than 5.6:
  - RHEL 8.6 or higher (inbox kernel)
  - RHEL 9.0 or higher (inbox kernel)
  - SLES 15 SP4 or higher (inbox kernel)
  - Ubuntu 20.10 or higher (inbox kernel)

NOTE

To check if a kernel has the required devlink parameter support:

```
devlink dev param show
```

If the kernel supports devlink params, output is displayed for each E810 device interface. Otherwise, an error message is returned by the kernel, similar to: `Command "param" not found`.

In addition to the kernel requirements listed for ice-1.8.x and later, ice-1.9.x and later requires Linux kernel **v5.12** or higher if using independent pollers with kthread based napi poll (earlier kernel versions use ksoftirqd threads to act as independent pollers). See Independent Pollers (Optional) for more details.

The following operating systems include kernel support for kthread-based napi poll and do not require a kernel update even though their base kernel version may be lower than 5.12:

- RHEL 8.6 or higher (inbox kernel)
- RHEL 9.0 or higher (inbox kernel)
- SLES 15 SP4 or higher (inbox kernel)

NOTE

Some applications or use cases might require a newer version of the kernel than what is listed in this section. The kernel version listed here is required for ADQ feature support in the kernel. See application-specific sections in this document to verify if there are any additional kernel requirements for the application.

For operating systems that require a kernel update, proceed with updating the kernel following the instructions in Update Kernel (If Needed).

For operating systems that do not require a kernel update, move directly on to Install ice Driver with ADQ Flags.

Did this document help answer your questions?

4.2 Update Kernel (If Needed)

NOTE

Kernel versions required for full ADQ support with older operating systems, unless the OS includes back-ported ADQ kernel support (and devlink param support).

- *ice* version 1.7.x and earlier requires LTS 4.19.x or later.
- *ice* version 1.8.x and later requires LTS 5.6 or later, or kernel with devlink param support for per-TC flow director.
- *ice* version 1.9.x and later requires LTS 5.6 or later, or kernel with devlink param support for per-TC flow director and independent poller. For independent poller using kthread-based napi poll, LTS 5.12 or later is required.

Refer to Install OS for determining if your OS includes ADQ kernel support or if a kernel update is required.

If kernel update is required, follow these steps:

1. Clone the kernel using **git**.

```
git clone http://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
```

2. Checkout to use long term stable branch (based off v5.12).

```
git checkout -b local_5.12 v5.12
```

OR

```
git reset --hard v5.12
```

3. Install the necessary packages to compile the kernel.

   Package installation can be done through several methods that are relevant to the base distribution, such as yum/dnf/apt-get. Packages needed are `gcc`, `bc`, `make`, `git`, `ncurses-devel`, `kernel-devel`, `kernel-headers`, `openssl`, `openssl-devel`, `binutils`, `bison`, `flex`, and `network-scripts`.

   Some kernel/arch combinations might require a package with the development version of *libelf* (for example, *libelf-dev*, *libelf-devel*, or *elfutils-libelf-devel*).

   Example (RHEL):

```
yum install -y gcc bc make git ncurses-devel kernel-devel kernel-headers openssl \
openssl-devel binutils bison flex network-scripts elfutils-libelf-devel
```

4. Compile and install the kernel.
   a. **cd** to the kernel directory.
   b. For operating systems based on older kernels (for example, RHEL 7.x is based on the kernel 3.10.x series), it might be best to use a config file that is based off of a closer kernel series (for example, from https://rpmfind.net/linux/RPM/fedora/29/x86_64/k/kernel-4.18.16-300.fc29.x86_64.html or newer).

Did this document help answer your questions?

Example:

```
rpm2cpio kernel-core-4.18.16-300.fc29.x86_64.rpm | cpio --extract \
--to-stdout ./lib/modules/4.18.16-300.fc29.x86_64/config > .config
make olddefconf
```

For operating systems based on later kernel versions (for example, RHEL 8.x), using the inbox kernel *.config* as a starting point should be sufficient.

Example:

```
make olddefconfig
```

c. Ensure these lines are in the *.config* file (note the inbox *ice* driver should NOT be built in the kernel, as the inbox driver can cause issues).

```
CONFIG_NET_SCH_MQPRIO=m
CONFIG_NET_ACT_MIRRED=m
CONFIG_NET_CLS_FLOWER=m
CONFIG_CGROUP_NET_PRIO=y
CONFIG_NET_RX_BUSY_POLL=y
CONFIG_SMP=y
CONFIG_ICE=n
CONFIG_NR_CPUS_DEFAULT=8192
```

NOTE

Additional config flags that might be required for specific applications:

```
CONFIG_BPF=y
CONFIG_EPOLL=y
CONFIG_NET_DEVLINK=y
CONFIG_XDP_SOCKETS=y
```

Use Cases:

- `CONFIG_BPF` is needed for applications that use the SO_ATTACH_REUSEPORT_CBPF socket option (such as NGINX).
- `CONFIG_EPOLL` is needed for applications that use epoll based syscall from user-space (example: using NGINX without ADQ and busy_poll, for performance comparisons to ADQ).
- `CONFIG_NET_DEVLINK` is needed for devlink param commands to configure per-TC inline flow director and independent poller
- `CONFIG_XDP_SOCKETS` is needed for AF_XDP.

NOTE

Note that **sed** can also be used to make these changes. For example:

```
sed -i -r 's/^(CONFIG_NR_CPUS=.*|# CONFIG_NR_CPUS is not set)/
CONFIG_NR_CPUS=8192/' .config sed -i -r 's/^(CONFIG_ICE=.*|# CONFIG_ICE
is not set)/CONFIG_ICE=n/' .config
```


Did this document help answer your questions?

d. Compile.

```
make -j"$(nproc)"
make -j"$(nproc)" modules
make -j"$(nproc)" modules_install
make -j"$(nproc)" install
make -j"$(nproc)" INSTALL_HDR_PATH=/usr/local headers_install
```

NOTE

`$(nproc)` = number of logical cores.

NOTE

`CONFIG_NR_CPUS_DEFAULT` might get set back to the default value of 64 on 4.x kernels after comping the kernel. On 5.x and later kernels, the value of 8192 should remain applied after the kernel is compiled.

5. Reboot to the new kernel.

4.3 Install *ice* Driver with ADQ Flags

The following variables are used in the examples in this section:

$driverver	The *ice* driver version.

1. Download the *ice* and untar.

```
tar -xzvf ice-${driverver}.tar.gz
cd ice-${driverver}/src
```

2. Compile and install the *ice* driver.

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

NOTE

To compile the *ice* driver on some kernel/arch combinations, you might need to install a package with the development version of libelf (for example, libelf-dev, libelf-devel, or elfutilsl-libelf-devel).

NOTE

The pkt_busy_poll counters are enabled by the –DADQ_PERF_COUNTERS CFLAG during *ice* driver compile. If the counters are not listed in `ethtool -S` output, verify that *ice* was built using the compile command in this step.

Example (RHEL):

```
yum install elfutils-libelf-devel
```


Did this document help answer your questions?

3. Load the *ice* driver.

```
modprobe ice
```

4.4 Install iproute2 (Update If Needed)

NOTE

iproute2-ss180813 or later is required. It is included in operating systems with inbox base kernels 4.17.19 or newer. If the OS has an older inbox base kernel (even if back-ported for ADQ support), an **iproute2** update may be required:

https://git.kernel.org/pub/scm/network/iproute2/iproute2.git/

Note that this step is not required if the ADQ setup script is used to create an ADQ setup.

1. Ensure that the necessary kernel modules are loaded.

```
modprobe sch_mqprio
modprobe act_mirred
modprobe cls_flower
```

2. Install **iproute2**.

```
clone the public iproute2:
git clone https://git.kernel.org/pub/scm/network/iproute2/iproute2.git
yum -y install libdb-devel
cd iproute2
./configure
make DESTDIR=/opt/iproute2 install
```

NOTE

If using RHEL 8.1 or later, **iproute2** is already at required version for ADQ and can be installed with the *iproute-tc* package rather than the above install procedure.

Example on RHEL 8.1:

```
yum install -y iproute-tc
```


Did this document help answer your questions?

4.5 Install cgroup Packages (If Needed)

This step is only needed if using Linux cgroups to align the TX traffic to the RX filter(s), using net_prio ID. If using egress tc filter(s) with clsact and skbedit, instead of cgroups, this step is not needed.

Install the **libcgroup** and **libcgroup-tools** packages on the system.

Example: RHEL 8.x and earlier:

```
yum install -y libcgroup libcgroup-tools
```

Example: RHEL 9.0 and later:

NOTE

The **libcgroup** and **libcgroup-tools** packages have been removed in RHEL 9.0 repository. We must manually download and install them using wget.

Wget https://kojihub.stream.centos.org/kojifiles/packages/libcgroup/0.42.2/5.el9/x86_64/libcgroup-0.42.2-5.el9.x86_64.rpm

Wget https://kojihub.stream.centos.org/kojifiles/packages/libcgroup/0.42.2/5.el9/x86_64/libcgroup-tools-0.42.2-5.el9.x86_64.rpm

NOTE

- RHEL 9.0 uses cgroup-v2 by default, which differs from RHEL 8.x using cgroup-v1. ADQ technology supports both versions of cgroup .

Did this document help answer your questions?

5.0 General System Tuning

NOTE

These settings are not persistent between reboots.

NOTE

System tuning is subjective and dependent on many factors in your environment, including platform specifications and software workload. These tunings have shown improved performance of heavily loaded systems using ADQ with fast IO workloads, but have only been tested on a limited number of system configurations and hardware. Experimentation is recommended to find the best settings for a particular workload.

1. Disable firewalls.

```
service firewalld stop; systemctl mask firewalld
```

2. Disable Security-Enhanced Linux (SELinux) (requires reboot to take affect).

   Change `SELINUX=enforcing` to `SELINUX=disabled` in */etc/selinux/config*.

3. Enable latency-performance tuned profile, if it makes sense for the application and workload. Refer to https://github.com/redhat-performance/tuned/blob/master/profiles/latency-performance/tuned.conf for a full list of what is expected to be included in the profile.

   NOTE

   This performance mode (and the performance governor in the next step) limit the CPU power management settings in the OS to provide more consistent results on a highly loaded system with many application threads. If using fewer queues than the total number of CPUs or on lightly loaded systems (for example netperf with only a few processes), skipping this step might increase performance, since under certain circumstances allowing Turbo mode on more cores can give a greater benefit than the avoidance of CPU power management sleep states (C States). Similar tuning choices should be made for CPU power management and Turbo settings in the BIOS.

```
tuned-adm profile latency-performance
```

   NOTE

   The *tuned-adm* daemon is not installed by default in RHEL9.0 systems. Install it with the command `yum install tuned`.

Did this document help answer your questions?

Check that settings are applied correctly:

```
cat /etc/tuned/active_profile
```

Output: `latency-performance`

```
cat /etc/tuned/profile_mode
```

Output: `manual`

NOTE

The default performance tuned setting is `tuned-adm profile throughput-performance`, which works well with many applications and workloads. Test both modes for optimal performance. Refer to https://github.com/redhat-performance/tuned/blob/master/profiles/throughput-performance/tuned.conf for a full list of what is expected to be included in the profile.

4. Set the CPU scaling governor to performance mode.

```
x86_energy_perf_policy performance
```

5. Increase the hard/soft nofile limits.

```
vi /etc/security/limits.conf
*  hard       nofile           16384
*  soft       nofile           16384
```

NOTE

Some OS distros do not allow a non-root user to set limits >64, even when listed in */etc/security/limits.conf*. A workaround is to set `DefaultLimitMEMLOCK=` in */etc/systemd/system.conf* and */etc/systemd/user.conf*, where the limit is defined in bytes.

Example:

The following output is on a system that has the default limit set to 1GB:

```
# grep ^DefaultLimitMEMLOCK /etc/systemd/system.conf
DefaultLimitMEMLOCK=1073741824

# grep ^DefaultLimitMEMLOCK /etc/systemd/user.conf
DefaultLimitMEMLOCK=1073741824
```

6. Stop the **irqbalance** service. (Needed for interface interrupt affinity settings.)

```
systemctl stop irqbalance
```

7. Stop LLDP agent (if installed and running).

```
systemctl stop lldpad.service
systemctl stop lldpad.socket
```


Did this document help answer your questions?

8. Set system network polling parameters. For the application testing sections listed in this document, follow the recommended parameters.

   In general, whether to use `busy_poll` alone or `busy_poll` and `busy_read` depends on the architecture of the application being used.

   For a majority of applications, which use epoll_wait/epoll_pwait followed by read/recv, it is recommended to set only `net.core.busy_poll` to the number of microseconds to wait for packets on the device queue for socket polling. Recommend 50K µs (50 ms) for a starting point to stay in poll mode as much as possible. This gives time for up to thousands of connections to be formed before timing out and dropping back to interrupt mode.

```
sysctl -w net.core.busy_poll=50000
```

   Some applications, such as **netperf**, which uses select/poll followed by read/recv, require `net.core.busy_read` to also be set along with `net.core.busy_poll`:

   Example:

```
sysctl -w net.core.busy_poll=50000
sysctl -w net.core.busy_read=50000
```

   NOTES

   - If setting system-wide `busy_read` is not acceptable, `busy_read` can be set per socket in the application using socket option `SO_BUSY_LOOP`.
   - The system-wide `busy_poll` and `busy_read` settings are only required when using application dependent polling mode; they are not required when using independent pollers.
   - When using the ADQ setup script to configure ADQ for application dependent polling (ADQ1.0), global `busy_poll` and `busy_read` settings are not required to be set manually. The script can handle these as well.

9. Configure OS tuning parameters to handle increased packet rate (note that some of these parameters will increase memory usage on the system):

   a. Increase the maximum number of connection requests queued for any listening socket.

```
sysctl -w net.core.somaxconn=4096
```

   b. Increase the maximum number of packets allowed to queue when a particular interface receives packets faster than the kernel can process them.

```
sysctl -w net.core.netdev_max_backlog=8192
```

   NOTE

   Set `netdev_max_backlog` to `250000` to improve ADQ performance with Apache Traffic Server (ATS).

Did this document help answer your questions?

c. Increase the maximum length of the queue for incomplete TCP sockets (SYN RECEIVED state).

```
sysctl -w net.ipv4.tcp_max_syn_backlog=16384
```

NOTE

Set `tcp_max_syn_backlog` to `250000` to improve ADQ performance with ATS.

NOTE

Applications might have their own backlog settings that could need to be adjusted. If backlog is set too low, performance can be impacted due to uneven queue use.

NOTE

If the TCP backlog setting is too low, you might see *dmesg* logs such as: `TCP: request_sock_TCP: Possible SYN flooding.`

d. Increase the Rx and Tx socket buffer maximum size (in bytes).

```
sysctl -w net.core.rmem_max=16777216
sysctl -w net.core.wmem_max=16777216
```

e. Adjust TCP memory usage auto-tuning for the stack (low, pressure, and max - in pages).

```
sysctl -w net.ipv4.tcp_mem="764688 1019584 16777216"
```

f. Adjust TCP buffer auto-tuning for each connection (min, default, and max - in bytes).

```
sysctl -w net.ipv4.tcp_rmem="8192 87380 16777216"
sysctl -w net.ipv4.tcp_wmem="8192 65536 16777216"
```

g. Apply immediately.

```
sysctl -w net.ipv4.route.flush=1
```

For convenience, the commands in this section (except for disabling SELinux, which requires a reboot) are condensed below:

NOTE

When using independent pollers, global `busy_poll` and `busy_read` are not required, these are only required for application dependent polling.

Did this document help answer your questions?

NOTE

For improved performance, the ATS-required backlog settings should be increased to 250000.

```
service firewalld stop; systemctl mask firewalld
tuned-adm profile latency-performance
cat /etc/tuned/active_profile
cat /etc/tuned/profile_mode
x86_energy_perf_policy performance
systemctl stop irqbalance
systemctl stop lldpad.service
systemctl stop lldpad.socket

# Application dependent polling
sysctl -w net.core.busy_poll=50000
sysctl -w net.core.busy_read=50000

sysctl -w net.core.somaxconn=4096
sysctl -w net.core.netdev_max_backlog=8192
sysctl -w net.ipv4.tcp_max_syn_backlog=16384
sysctl -w net.core.rmem_max=16777216
sysctl -w net.core.wmem_max=16777216
sysctl -w net.ipv4.tcp_mem="764688 1019584 16777216"
sysctl -w net.ipv4.tcp_rmem="8192 87380 16777216"
sysctl -w net.ipv4.tcp_wmem="8192 65536 16777216"
sysctl -w net.ipv4.route.flush=1
```


Did this document help answer your questions?

6.0 ADQ Setup Using ADQ Setup Script

There are two methods to configure ADQ:

- Method-1: Using ADQ setup script
- Method-2: Using manual ADQ configuration

This section is meant to be a generic walk-through of configuring ADQ with the ADQ setup script. If testing one of the enabled applications, skip this section and instead follow the instructions in the relevant testing application sections:

- Testing ADQ with Netperf
- Testing ADQ with Redis
- Testing ADQ with Memcached/rpc-perf
- Testing ADQ with NGINX
- Testing ADQ with AF_XDP

NOTE

This section contains the steps for configuring ADQ using setup script, for manual ADQ configuration, see Appendix: ADQ Configuration Reference (Manual)

6.1 ADQ Setup Script Prerequisites

The ADQ setup script can be used to configure ADQ parameters such as traffic classes, priority, filters, poller settings, and ethtool parameters through a python script.

The script is called *adqsetup.py* and it can be found in the *scripts* directory of the *ice* driver version 1.9.7 tar file (or later). A major benefit of the script is that it can also maintain persistent ADQ settings across system reboots.

Refer to the README file located in the *scripts/adqsetup* subdirectory of the ice driver package for more information on the script's usage, support, and configuration settings.

Dependencies: Python 2.7 or Python 3.5 (or later)

Did this document help answer your questions?

NOTES

1. If you get a */usr/bin/env: 'python': No such file or directory* error when you run the script, please install Python. If you have already installed Python, then try `whereis python` and you should see a message like *python: /usr/bin/python2.7 /usr/bin/python3.6 /usr/bin/python3.6m /usr/bin/python3.9* on the first line of the output.

   Either run the version you wish to use manually: `python3.6 adqsetup.py help`; or,

   Create a *python* symbolic link on the path: `ln -s /usr/bin/python3.6 /usr/local/bin/python`

2. When using the pollers feature, if your driver does **not** have independent poller support (as in *ice* driver versions before 1.9.7) , you will see the following error:

```
** Setting independent poller **
devlink answers: Invalid argument
```


Did this document help answer your questions?

6.2 ADQ Setup Script Installation

NOTE

Before the script installation and ADQ system configuration, perform the general system setup as in ADQ System Under Test (SUT) Installation and General System Tuning.

The script can be installed in one of two ways:

- Using Python Package Index (pip):

```
python -m pip install adqsetup
```

- Including it with *ice* driver version 1.9.7 or later:

```
cd $pathtoicepackage
python scripts/adqsetup/adqsetup.py install
```

6.3 ADQ Setup Script Usage

The following variables are used in the examples in this section:

$iface	The interface in use (PF).
$pathtoicepackage	The path to the ice driver package.
$port	Destination port of traffic
$portrange	Destination port range (ex: 11211-11218)
$addr	Destination IP address of traffic
$remote_addr	Source IP address of traffic
$remote_port	Source port of traffic
$num_queues_tc0	The number of queues for default traffic class (default queues are 2)
$num_queues_tc1	The number of queues for application traffic class 1
$protocol	IP Protocol of the traffic (tcp/udp)
$conf_file	Config file name to apply the settings to create the ADQ setup(ex: Memcached.cong, redis.conf and nginx.conf)
$driverpath	Device driver path to .ko file (ex: src/ice.ko)
$file_name	Creates a *quick-config* consisting of one application traffic class (it is a user defined name ex: netperf, redis, nginx etc)
$poller_number	The number of queue pairs per independent poller for a given TC (max value is #queues in the TC).
$poller_timeout	The timeout value for the independent pollers for a given TC (nonzero integer value in jiffies, default value 10000).
$mask	The bit mask used to select which queues to run the subcommand on (ex: for ADQ queues 2-9 : 0x3fc (0011 1111 1100)).
$cpus	Assigns pollers to specific CPU cores (ex: 4, 5, 6, 7)

Did this document help answer your questions?

NOTE

The script can be run in multiple ways:

1. Command line (V1.x only supports single TC configuration. V2.0 or later is required to configure multiple TCs via command line.)
2. Configuration file (Single/Multiple TC configuration supported from V1.x and later)
3. JSON (Supported from version V2.x and later)

Using Command Line Parameters

- Create ADQ filters in *exclusive* mode for multithreaded applications (uses inline flow director to distribute the connections to queues).

```
adqsetup --dev=$iface --priority=skbedit create [$file name] queues \
$num_queues_tc1 ports $port
```

- Create ADQ filters in *shared* mode for single threaded applications (creates perfect filters using Ntuple rules to distribute connections to queues).

```
adqsetup --dev=$iface --priority=skbedit create [$file name] mode shared \
queues $num_queues_tc1 protocol $protocol addrs $addr ports $portrange
```

- Create ADQ setup with interrupt moderation rate values for Tx/RX.

```
adqsetup --dev=$iface --priority=skbedit --rxusecs=0 \
--txadapt=off --txusecs=500 create [$file name] mode shared queues \
$num_queues_tc1 ports $portrange
```

  — To show all the per-queue coalesce settings:

```
ethtool --per-queue $iface --show-coalesce
```

  — To show the queues of ADQ traffic class alone:

```
ethtool --per-queue $iface queue_mask $mask --show-coalesce
```

- Create an ADQ setup by using various device drivers in the syntax.

```
adqsetup --dev=$iface --priority=skbedit create [$file name] queues \
 $num_queues_tc1 protocol $protocol ports $port --driver $driverpath \
--reload
```

- Create ADQ setup using independent pollers configuration with default timeout.

```
adqsetup --dev=$iface --priority=skbedit create [$file name] queues \
 $num_queues_tc1 ports $port addrs $addr pollers $poller_number
```

- Create ADQ setup using independent pollers configuration with pollers timeout in jiffies.

```
adqsetup --dev=$iface --priority=skbedit create [$file name] queues \
 $num_queues_tc1 ports $port addrs $addr pollers \ $poller_number \
pollers_timeout $poller_timeout
```


Did this document help answer your questions?

- Create ADQ setup with independent pollers by pinning pollers to specific CPU cores.

```
adqsetup --dev=$iface --priority=skbedit create [$file_name] mode shared
queues
$num_queues ports $portrange cpus $cpus pollers $poller_number
```

NOTES

1. Adaptive transmit and receive interrupt coalescing is enabled by default. Coalesce settings must be turned off (`-- txadapt=off -rxadapt=off`) to set the Tx interrupt moderation rate to a static value and turn off Rx interrupt moderation (ex. `-- txusec=500 --rxusec=0`).
2. In ADQ setup script, the independent pollers argument (`pollers`) is not directly related to qps_per_poller arguments passed to devlink. The script allows the user to specify how many pollers a particular TC should have, rather than specifying qps_per_poller (queue pairs per poller).
3. In the above examples, the driver is not reloaded, but can be reloaded by using the `--reload` parameter while creating the ADQ setup.
4. To load/use a different device driver while creating the setup, the `--driver` parameter may be used. Device driver path is the full path to the .ko file (ex: `ice-1.9.x/src/ice.ko`). Interface *must* be set to come up automatically with an ip address (via NetworkManager or other). adqsetup will wait up to three seconds for this to occur before erroring out. Conversely, you can load the driver and setup the interface manually before running the adqsetup.

   NOTE

   To load/use the various device drivers in adqsetup V1.x, use the `--reload` parameter in conjunction with the `--drivers` parameter.

5. If the `--priority=skbedit` parameter is not specified in the command syntax, a Cgroup must be created to align ingress and egress traffic. The performance of the two methods (cgroup priority or TC filters with skbedit priority) may differ slightly based on the application or workload.
6. The `--debug` parameter is optional, but it is useful for obtaining complete configuration details.
7. adqsetup 1.x required updated versions of the `tc`, `ethtool`, and `devlink` commands to be installed on the system. With adqsetup 2. x and onward, this requirement has been removed.
8. Pinning pollers to specific CPU cores, as well as 256 queue pair support, were added in adqsetup V2.x and later.

Using Configuration File

- Apply a configuration file to create ADQ setup.

```
adqsetup --dev=$iface apply $conf_file
```


Did this document help answer your questions?

The adqsetup script can automatically generate an `examples` subdirectory containing a set of example config files using the following command that can be customized as needed.

```
adqsetup examples
```

- To apply a configuration file for persistent ADQ settings across system reboots, use the following commands

```
adqsetup --dev=$iface persist $conf_file
systemctl enable --now adqsetup@$iface
```

  — To reapply the settings without system reboot:

```
systemctl restart adqsetup@$iface
```

  — To disable the persistence across system reboots:

```
systemctl disable adqsetup@$iface
```

Limitations

The ADQ setup script does not support the following configurations currently:

- ADQ configuration with VF
- VLAN
- IPv6
- GTP Filters
- VXLAN Filters
- Any D-MAC Filters
- Forward to Queue based filtering
- Rate limit option

Did this document help answer your questions?

7.0 ADQ Configuration on SR-IOV Virtual Functions

Single Root I/O Virtualization (SR-IOV) is a specification that allows a single physical PCIe device to present multiple separate PCIe devices in a virtual environment. The Intel® Ethernet Adaptive Virtual Function (Intel® Ethernet AVF) *iavf* driver for Linux can be configured with ADQ on Intel® Ethernet 800 Series Network Adapters to provide ADQ performance enhancements in a virtual environment. In this section, the *iavf* driver is referred to as VF for Virtual Function, and the *ice* driver is referred to as PF for Physical Function.

NOTE

This document is not intended to be a complete SR-IOV configuration guide. Refer to SR-IOV documentation, also *ice* and *iavf* driver README documents for more details on usage.

NOTE

The ADQ setup script does not support configuring ADQ with the VF interface.

The following variables are used in the examples in this section:

$vfdriverver	The *iavf* driver version.
$numvfs	The number of VFs to spawn on the SUT.
$iface_pf	The Physical Function (PF) interface name on the SUT.
$iface_vf	The Virtual Function (VF) interface name on the SUT.
$ipaddr_pf	The IP Address assigned to the PF interface on the SUT.
$ipaddr_vf	The IP Address assigned to the VF interface on the SUT.
$ipaddrclient	The IP Address assigned to the PF interface on the client where application is being run.
$app_port	The TCP port on the SUT where the application is being run.

7.1 General SUT Configuration for VFs

This section describes general System Under Test (SUT) configuration for either a bare metal VF environment, or for running a VF inside a Virtual Machine (VM) on the SUT.

1. Enable virtualization in the BIOS on the SUT. The following BIOS settings must be turned on to enable virtualization capability. These settings vary based on the platform, however most Intel BIOS should have similar settings.

```
[BIOS::Advanced::Processor Configuration]
Intel(R) Virtualization Technology=Enabled

[BIOS::Advanced::Integrated IO Configuration]
Intel(R) VT for Directed I/O=Enabled
```


Did this document help answer your questions?

```
[BIOS::Advanced::Integrated IO Configuration]
SR-IOV Support=Enabled
```

2. Perform general system OS install and setup. Complete the ADQ install and setup in ADQ System Under Test (SUT) Installation and General System Tuning.

NOTE

Many settings in Section 5.0 do not persist between reboots and might need to be reapplied.

NOTE

The OS install and setup steps on the SUT are only needed when spawning VFs on the local host. If using VFs inside a Virtual Machine (VM), the OS is installed and configured inside the VM's guest OS and this step can be skipped on the host OS.

3. Compile and install the *iavf* driver:

   a. Download the*iavf* driver and untar:

```
tar -xzvf iavf-${vfdriverver}.tar.gz
cd iavf-${vfdriverver}/src
```

   b. Compile and install the *iavf* driver:

```
make -j$(nproc) install
```

   c. Load *iavf* driver:

```
modprobe iavf
```

NOTE

The *iavf* driver only needs to be compiled and installed on the SUT when spawning VFs on the local host. If using a VM, *iavf* is compiled and installed on the VM's guest OS and this step can be skipped on the host OS.

4. Enable Intel® IOMMU kernel boot parameters. To enable full virtualization in the kernel, add `intel_iommu=on` and `iommu=pt` to the kernel boot parameters, regenerate GRUB configuration, and reboot the host.

   Use Linux admin tools such as **grubby** or **grub2-editenv** to set the kernel parameters and reboot. For example:

```
grubby --args="intel_iommu=on iommu=pt" --update-kernel=DEFAULT
reboot
```

   or

```
grub2-editenv - set "$(grub2-editenv - list | grep kernelopts) intel_iommu=on \
iommu=pt"
reboot
```


Did this document help answer your questions?

If **grubby** and **grub2-editenv** are not available on your system, you can alternatively append `intel_iommu=on iommu=pt` to GRUB_CMDLINE_LINUX line in */etc/default/grub*, followed by `grub2-mkconfig -o /boot/grub2/grub.cfg` and reboot.

The **iommu** kernel parameters can be verified after reboot using `cat /proc/cmdline`.

5. Depending on your VF environment, follow either SUT Configuration Bare Metal VF or SUT Configuration VF Inside a VM.

7.2 SUT Configuration Bare Metal VF

This section describes the SUT configuration using ADQ with a bare metal VF, spawning one or more VFs on the physical server itself, not from within a Virtual Machine (VM).

1. Unload and reload the PF and VF drivers and bring up the PF interface.

```
rmmod iavf
sleep 1
rmmod ice
sleep 2
modprobe ice
sleep 2
modprobe iavf
sleep 2
ifup $iface_pf
```

2. Spawn VFs. Use the following command to create virtual function(s).

```
echo $numvfs > /sys/class/net/$iface_pf/device/sriov_numvfs
sleep 2
```

The following example creates 1 VF on the p1p2 PF network interface:

```
echo 1 > /sys/class/net/p1p2/device/sriov_numvfs
```

NOTE

Note:To find the maximum number of VFs supported by the PF network device:

```
cat /sys/class/net/$iface_pf/device/sriov_totalvfs
```

a. Verify the creation of the VF device(s) using the **lspci** utility. For example:

```
lspci | grep -i virtual
18:11.0 Ethernet controller: Intel Corporation Ethernet Adaptive Virtual
Function (rev 01)
```

b. Verify the creation of the VF interface(s) using the **ip addr** command. The VF interface is the PF interface name with a *_0* or *v0* suffix (*_1*, *_2*, and so on or *v1*, *v2*, and so on for multiple VFs). The VF interface is represented by **$iface_vf** in this section.

Did this document help answer your questions?

For example, if the PF interface is p1p2, the VF spawned above would create VF interface `p1p2_0`:

```
12: p1p2_0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP
group default qlen 1000 link/ether 7a:79:3c:b0:8f:da brd ff:ff:ff:ff:ff:ff
inet 13.100.4.21/24 scope global p1p2_0 valid_lft forever preferred_lft
forever
```

3. Disable spoof checking and enable trust on each of the VF interfaces.

```
ip link set dev $iface_pf vf 0 spoofchk off
ip link set dev $iface_pf vf 0 trust on
```

Verify spoof checking and trust settings using **ip link** command:

```
11: p1p2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode
DEFAULT group default qlen 1000 link/ether 68:05:ca:a3:70:71 brd
ff:ff:ff:ff:ff:ff vf 0 MAC 7a:79:3c:b0:8f:da, spoof checking off, link-state
auto, trust on

12: p1p2_0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode
DEFAULT group default qlen 1000 link/ether 7a:79:3c:b0:8f:da brd
ff:ff:ff:ff:ff:ff
```

NOTE

Trusted mode must be enabled for TC filters to be created in the hardware on the NIC. If the *iavf* driver is used without trusted mode enabled, TC filters are added to the software table but not offloaded to hardware, as documented in Fixed and Known Issues - Read First.

4. Bring up the VF interface and assign an IP Address.

```
ip addr add $ipaddr_vf dev $iface_vf
ifup $iface_vf
```

Verify connectivity with link partner.

```
ping $ipaddrclient
```

NOTE

Some applications expect the IP Address of the VF interface to be on a separate subnet than the PF interface. If ADQ application traffic configured for a VF interface is directed to the PF interface, check the subnets.

Did this document help answer your questions?

7.3 SUT Configuration VF Inside a VM

This section describes system configuration using ADQ with a VF that is spawned inside a Virtual Machine (VM) on the SUT.

On the SUT Host OS

1. Unload the *iavf* (VF) and *ice* (PF) drivers.

```
rmmod iavf
sleep 1
rmmod ice
sleep 2
```

2. Reload the *ice* driver and bring up the PF interface.

```
modprobe ice
sleep 2
ifup $iface_pf
```

3. Spawn VFs. Use the following command to create virtual function(s):

```
echo $numvfs > /sys/class/net/$iface_pf/device/sriov_numvfs
```

The following example creates 1 VF on the p1p2 PF network interface:

```
echo 1 > /sys/class/net/p1p2/device/sriov_numvfs
```

NOTE

To find the maximum number of VFs the supported by the PF network device:

```
cat /sys/class/net/$iface_pf/device/sriov_totalvfs
```

a. Verify that the VF was created using **lspci** utility. For example:

```
lspci | grep -i virtual
18:11.0 Ethernet controller: Intel Corporation Ethernet Adaptive Virtual
Function (rev 01)
```

b. Verify that the VF interface was created using the **ip addr** command. The VF interface is the PF interface name with *a _0* suffix (*_1*, *_2*, and so on for multiple VFs). The VF interface is represented by **$iface_vf** in this section.

For example, if the PF interface is p1p2. the VF spawned above would create VF interface p1p2_0:

```
12: p1p2_0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP
group default qlen 1000 link/ether 7a:79:3c:b0:8f:da brd
ff:ff:ff:ff:ff:ff inet 13.100.4.21/24 scope global p1p2_0 valid_lft
forever preferred_lft forever
```

4. Disable spoof checking and enable trust on the VF interface.

```
ip link set dev $iface_pf vf 0 spoofchk off
ip link set dev $iface_pf vf 0 trust on
```


Did this document help answer your questions?

Verify spoof checking and trust settings using **ip link** command:

```
11: p1p2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode
DEFAULT group default qlen 1000 link/ether 68:05:ca:a3:70:71 brd
ff:ff:ff:ff:ff:ff vf 0 MAC 7a:79:3c:b0:8f:da, spoof checking off, link-state
auto, trust on

12: p1p2_0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode
DEFAULT group default qlen 1000 link/ether 7a:79:3c:b0:8f:da brd
ff:ff:ff:ff:ff:ff
```

NOTE

Trusted mode must be enabled for TC filters to be created in the hardware on the NIC. If the *iavf* driver is used without trusted mode enabled, TC filters are added to the software table but not offloaded to hardware, as documented in Fixed and Known Issues - Read First.

5. Install packages required for VM creation and VM management. Example, on RHEL:

```
yum install qemu-kvm qemu-img libvirt virt-manager
```

6. Create a Virtual Machine (VM).
   a. Create a VM using **virt-manager** Graphical User Interface (GUI) or **qemu** Command Line Interface (CLI).
   b. Select guest OS during VM creation for VM OS install.
   c. Assign a VF interface to the VM using **virt-manager** or **qemu**.

NOTE

This document does not include details for creating a VM or assigning a VF to the VM. Refer to SR-IOV documentation for instructions on creating VMs and assigning VFs.

On the SUT Guest OS (inside VM)

1. Configure VM guest OS environment. Apply all system tunings to the VM from ADQ System Under Test (SUT) Installation and General System Tuning. This include the required kernel with ADQ flags, *ice* driver with ADQ flags, **iproute2**, cgroup packages, and various system tunings.

NOTE

Many settings in General System Tuning do not persist between reboots and might need to be reapplied.

2. Compile and install the *iavf* driver.
   a. Download the *iavf* driver and untar:

```
tar -xzvf iavf-${vfdriverver}.tar.gz
cd iavf-${vfdriverver}/src
```


Did this document help answer your questions?

b. Compile and install the *iavf* driver:

```
make -j$(nproc) install
```

c. Load *iavf* driver:

```
modprobe iavf
sleep 2
```

3. Bring up the VF interface inside the VM and assign an IP Address.

```
ip addr add $ipaddr_vf dev $iface_vf
ifup $iface_vf
```

Verify connectivity with link partner.

```
ping $ipaddrclient
```

7.4 ADQ VF Configuration

The instructions in this section show how to configure ADQ on the SUT using the VF interface created in either SUT Configuration Bare Metal VF or SUT Configuration VF Inside a VM. If using a VM, the ADQ configuration should be done inside the VM.

1. Enable `channel-pkt-inspect-optimize` on the VF interface(s):

```
ethtool --set-priv-flags $iface_vf channel-pkt-inspect-optimize on
```

2. Follow the instructions in ADQ Configuration on SUT, and also the tuning steps listed in the applicable sections for netperf, Memcached, Redis, and NGINX for application-specific system tuning for the VF interface.

   When creating Traffic Classes (TCs), TC Filter(s), and issuing any **ethtool** commands, replace **$iface** with **$iface_vf**, and **$ipaddr** with **$ipaddr_vf** to use the VF interface and IP Address.

   For example, when creating a TC for the VF, specify the **$iface_vf** interface.

```
ethtool -K $iface_vf hw-tc-offload on

${pathtotc}/tc qdisc add dev $iface_vf root mqprio num_tc 2 \ map 0 1 queues
$num_queues_tc0@0 $num_queues_tc1@$num_queues_tc0 hw 1 \ mode channel
```

NOTE

Due to timing issues, applying TC filters immediately after the `tc qdisc add` command might result system reboot/crash. It is recommended to add two seconds of sleep time after the `tc qdisc add` command before adding TC filters.

NOTE

The total number of queues in the Traffic Classes (combined) is limited to the total number of queues in the VF. Refer to *iavf* documentation to determine total number of queues available for the VF before creating TCs.

Did this document help answer your questions?

The following example creates one TC filter on the VF interface under test:

```
${pathtotc}/tc qdisc add dev $iface_vf clsact

${pathtotc}/tc filter add dev $iface_vf protocol ip ingress\ prio 1 flower
dst_ip $ipaddr_vf/32 ip_proto tcp dst_port $app_port skip_sw hw_tc 1
```

3. Confirm TC configuration:
   a. Check that TCs were created correctly.

   ```
   ${pathtotc}/tc qdisc show dev $iface_vf
   ```

   b. Check TC filters.

   ```
   ${pathtotc}/tc filter show dev $iface_vf ingress
   ```

4. After the test finishes, remove the ADQ VF configuration following the steps in Clear the ADQ Configuration.

   To remove VF interfaces:

   ```
   echo 0 > /sys/class/net/$iface_pf/device/sriov_numvfs
   ```

All other ADQ configuration follows section ADQ Configuration on SUT and the application-specific sections that follow.

NOTE

For VF, use the **set_irq_affinity** and **set_xps_rxqs** scripts from the *ice* driver package and specify the **$iface_vf** interface.

NOTE

Add at least 2 seconds of sleep time to restore all queues before reconfiguring the ADQ VF traffic classes.

Did this document help answer your questions?

8.0 Troubleshooting

1. During or after a test, check that `pkt_busy_poll` counters on the ADQ queues for the interface are incrementing:

```
watch -d "ethtool -S ${iface} | grep poll | grep -v ': 0' | column
```

NOTE

The `pkt_busy_poll` counters are enabled by the `-DADQ_PERF_COUNTERS CFLAG` during ice driver compile. If the counters are not listed in **ethtool -S** output, verify ice was built using the compile command listed in Install ice Driver with ADQ Flags.

   a. If ADQ Rx queues are being used, but not Tx, check if the priority flags on the server and client are set and using the same priority. Priority is first set by the **map** parameter in the **tc qdisc add** command when creating TCs (example: `map 0 1 2 queues`). Priority is next set by the **prio** parameter in the **tc filter add** command (example: `prio 1`). Priority is also assigned to the net_prio subsystem of the Control Group for applications that use cgroups (example: `cgset -r net_prio.ifpriomap="$iface $prio" ${cgroup_name}`). For applications using the SO_PRIORITY socket option, priority is set by the application (check application command-line for option to specify priority).

   b. If ADQ Tx queues are being used, but not Rx, check that the TC filter on both sides were applied properly.

   c. If none of the ADQ queues are being used, check that the port value matches in all of the commands on the client and server.

   d. If `pkt_not_busy_poll` counters are incrementing on ADQ queues, check the `busy_poll` and `busy_read` settings are applied correctly. For applications that require a ntuple (perfect filter) rule for default traffic, verify configuration of the default queue.

   e. If ADQ application traffic is going to the default queues when using VF interface(s), verify network subnets. Some applications require VF interfaces to be configured on a separate subnet than the PF interfaces.

2. Check versions and settings:

```
ip addr show dev $iface | grep -i -e "mtu" -e "inet"
ethtool -i $iface | grep -e "driver" -e "^version" -e "firmware"
ethtool --show-channels $iface
ethtool --show-coalesce $iface
ethtool --show-priv-flags $iface
${pathtotc}/tc filter show dev $iface ingress
${pathtotc}/tc qdisc show dev $iface | grep -i -e "queue" -e "mode" -e "mqprio"
cat /proc/sys/net/core/busy_poll
cat /proc/sys/net/core/busy_read
```


Did this document help answer your questions?

9.0 Testing ADQ with Netperf

Netperf is an easily available, widely-used open source benchmark that can showcase the benefits of ADQ on latency, jitter, and transactions per second (tps) in a simple request/response test (TCP_RR). TCP_RR is used due to its similarity to database workloads common in datacenters and cloud environments.

netperf is usually tested with an identical server and client connected back-to-back or through a switch. In this case, ADQ is configured on both on the server and on the client, though the configuration is slightly different between the two.

NOTE

This document is not intended to be a complete **netperf** configuration guide. Refer to **netperf** documentation for more details on usage.

There are two options to configure ADQ in both netperf server and client.

- Option 1: Using ADQ Setup Script
- Option 2: Using Linux Command Line (Manual Configuration)

Both configuration methods require the completion of the system setup steps in Installation and Configuration – Both Systems prior to configuring the netperf server and client for ADQ in netperf Server and netperf Client.

NOTE

This section contains the steps for configuring ADQ using the setup script. For manual ADQ configuration, see Appendix: ADQ Configuration Reference (Manual)

9.1 Installation and Configuration – Both Systems

The following variable is used in the examples in this section:

$iface	The interface in use (PF).

1. Perform general system setup.

   Complete the ADQ installation and setup in ADQ System Under Test (SUT) Installation and in General System Tuning on both the server and client systems.

   NOTE

   Many settings in General System Tuning and Adapter Preparation do not persist between reboots and might have to be reapplied.

Did this document help answer your questions?

2. Install **netperf**:

   Method 1: Install **netperf** package from OS distribution.

   If your OS distribution has a **netperf** binary rpm available, the package can be installed using the package manager for the OS. Example on RHEL 8.x:

```
yum install -y https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm
yum install -y netperf
```

   Method 2: Build and install **netperf** from source code.

   NOTE

   Depending on the distro being used, libmnl dependency might be required before building **netperf** 2.7.0 or higher.

   Download **netperf** (2.7.0 recommended): https://github.com/HewlettPackard/netperf.git

   a. Install and build according to standard **netperf** directions.

```
yum -y install automake texinfo
cd netperf
./autogen.sh
./configure
make && make install
```

   NOTE

   netperf does not compile cleanly on RHEL 9.0 with gcc version 11. This is due to the default CFLAG changing from `-fcommon` to `-fno-common` in gcc 10. More details on the flag change can be found here:https://gcc.gnu.org/gcc-10/

Did this document help answer your questions?

porting_to.html. To compile **netperf** on RHEL 9.0 with gcc 11, the `-fcommon` flag can be used as a workaround. Replace the `make && make install` line above with:

```
make CFLAGS=-fcommon && make install
```

9.2 netperf Server

NOTE

This section uses application dependent polling (ADQ1.0) for application and testing and does not use an independent poller (ADQ2.0).

The following variables are used in the examples in this section:

$iface	The interface in use (PF).
$ipaddrserver	The IP Address of the **netperf** server interface under test.
$app_port	The starting port of the **netperf** application. *Note:* This should be available on both sever and client. Usually a high number (>2000) is best, but check that it is available (for example: sudo `netstat -tulpn | grep LISTEN`).
ADQ Setup Script Variables	
$bp	Global busy_poll value to be set by ADQ Setup Script (system-wide setting, not to be used with independent poller).
$br	Global busy_read value to be set by ADQ Setup Script (system-wide setting, not to be used with independent poller).
$file_name	Application name to be set and used by ADQ Setup Script (any descriptive string, i.e., *netperf*).
$portrange	TCP ports to be configured by ADQ Setup Script.
$addr	Destination IP address of traffic to be configured by ADQ Setup Script.

Using ADQ Setup Script

The ADQ Setup script allows you to quickly and easily configure required ADQ parameters such as traffic classes, priority, filters, and ethtool parameters etc.

1. To configure ADQ, run the following command:

```
adqsetup --dev=$iface --priority=skbedit --busypoll=$bp --busyread=$br create
$file_name mode shared \
queues $num_queues_tc1 ports $portrange addrs $addr
```

See **Notes** below for customizing the ADQ configuration. Once ADQ is configured by **adqsetup**, start the netperf server.

Did this document help answer your questions?

NOTES

a. The example command above creates both ingress (Rx) and egress (Tx) filters, and Linux cgroups are not needed to be created and can be skipped (cgroup is only needed if `--priority=skbedit` was NOT specified in **adqsetup** command).

b. ADQ Setup script handles symmetric queues and affinity.

c. The setup script sets per-queue coalesce by default.

d. Set the transmit and receive interrupt coalescing values to `--rxadapt=off --rxusecs=0 --txadapt=off --txusecs=500` for improved ADQ performance.

e. To configure independent pollers, add the `--pollers=$pollers` parameter in the **adqsetup** command (and optionally `--pollers_timeout`), and **remove** the flags to set global `--busypoll=`**`$bp`** `--busyread=`**`$br`**.

f. Use the `cpu` parameter in the command to bind the independent pollers to specific CPU cores. Refer to ADQ Setup Using ADQ Setup Script for more information on pinning pollers to specific CPU cores.

g. The `--debug` parameter is optional, but it is useful for obtaining complete configuration details.

h. For more information on how to use the script, refer to ADQ Setup Using ADQ Setup Script.

2. Start **netperf** server (**netserver**).

   If egress TC filters(s) created with skbedit priority:

```
netserver -4 -L ${ipaddrserver} -Dd
```

NOTE

If cgroup priority is set for Application Network traffic (`--priority=skbedit` is not specified in the command), see Set cgroup Priority for Application Network Traffic (If Needed) for instructions on configuring cgroup and running tasks with cgroup.

NOTE

Use **netserver -h** for parameter details.

NOTE

The ADQ Setup script clears the existing configuration before proceeding with the new ADQ configuration. To clear manually, follow the steps in Clear the ADQ Configuration.

Did this document help answer your questions?

9.3 netperf Client

NOTE

This section uses application dependent polling (ADQ1.0) for application and testing and does not use an independent poller (ADQ2.0).

The following variables are used in the examples in this section:

$iface	The interface in use (PF).
$ipaddrserver	The IP Address of the **netperf** server interface under test.
$ipaddrclient	The IP of the **netperf** client interface under test.
$app_port	The starting port of the **netperf** application. *Note:* This should be available on both sever and client. Usually a high number (>2000) is best, but check that it is available (for example: `sudo netstat -tulpn | grep LISTEN`).
$app_threads	The number of **netperf** client instances to be run (must be less than or equal to $num_queues_tc1).
$testtime	The length of time for each test.
ADQ Setup Script variables	
$bp	Global busy_poll value to be set by ADQ Setup Script (system-wide setting, not to be used with independent poller).
$br	Global busy_read value to be set by ADQ Setup Script (system-wide setting, not to be used with independent poller).
$file_name	Application name to be set and used by ADQ Setup Script (any descriptive string, i.e., *netperf*).
$portrange	TCP ports to be configured by ADQ Setup Script.
$addr	Destination IP address of traffic to be configured by ADQ Setup Script.

Using ADQ Setup Script

The ADQ Setup script allows you to quickly and easily configure required ADQ parameters such as traffic classes, priority, filters, and ethtool parameters etc.

1. To configure ADQ, run the following command:

```
adqsetup --dev=$iface --priority=skbedit --busypoll=$bp --busyread=$br create
$file_name mode shared \
queues $num_queues_tc1 ports $portrange addrs $addr
```

   See **Notes** below for customizing the ADQ configuration. Once ADQ is configured by **adqsetup**, start the netperf client.

Did this document help answer your questions?

NOTES

a. The example command above creates both ingress (Rx) and egress (Tx) filters, and Linux cgroups are not needed to be created and can be skipped (cgroup is only needed if `--priority=skbedit` was NOT specified in **adqsetup** command).

b. ADQ setup script handles symmetric queues and affinity.

c. Set the transmit and receive interrupt coalescing values to `--rxadapt=off --rxusecs=0 --txadapt=off --txusecs=500` for improved ADQ performance.

d. To configure independent pollers, add the `-pollers=$pollers` parameter in the **adqsetup** command (and optionally `--pollers_timeout`), and **remove** the flags to set global `--busypoll=`**$bp** `--busyread=`**$br**.

e. Use the `cpu` parameter in the command to bind the independent pollers to specific CPU cores. Refer to ADQ Setup Using ADQ Setup Script for more information on pinning pollers to specific CPU cores.

f. The `--debug` parameter is optional, but it is useful for obtaining complete stack trace.

g. For more information on how to use the script, refer to ADQ Setup Using ADQ Setup Script.

2. Run traffic.

   Start simultaneous **netperf** client instances.

   Pinning the application to a CPU core on the local NUMA node provides more consistent results for ADQ. The following example uses the **netperf -T** option to select which core to use for each instance of the application.

```
numa=0 cpus_allowed=(`lscpu | grep node${numa} | cut -d: -f2 | sed -e 's/ //g' | sed -
e 's/,/ /g'`)
start_core=${cpus_allowed[$app_threads]} for ((i = 0; i < app_threads; i++)); do core=$
(( $i * 2 + $start_core ))
netperf -j -H $ipaddrserver -t \
TCP_RR -T $core -l ${testtime} -- -k \
THROUGHPUT,MIN_LATENCY,MAX_LATENCY,P50_LATENCY,P90_LATENCY,\
P99_LATENCY,STDDEV_LATENCY,MEAN_LATENCY -P $((app_port + i)),$((app_port + i)) & done
```

NOTE

Use **netperf -h** for parameter details. Reference here:

https://hewlettpackard.github.io/netperf/doc/netperf.html#TCP_005fRR

Did this document help answer your questions?

3. Verify ADQ traffic is on the correct queues.

   While ADQ application traffic is running, watch **ethtool** statistics to check that only the ADQ queues are being used (have significant traffic) with busy poll (pkt_busy_poll) for ADQ traffic. If non busy poll (pkt_not_busy_poll) have significant counts and/or if traffic is not confined to ADQ queues, recheck the configuration steps carefully.

   Example:

```
watch -d -n 0.5 "ethtool -S $iface | grep busy | column"
```

   See Verify ADQ Application Traffic and Independent Pollers (If Applicable) for example **watch** output.

NOTE

The ADQ Setup script clears the existing configuration before proceeding with the new ADQ configuration. To clear manually, follow the steps inClear the ADQ Configuration.

Did this document help answer your questions?

10.0 Testing ADQ with Redis

Redis is an open source in-memory data structure store. For this section, Redis is used as an example of a real-world application with ADQ. ADQ enables Redis clusters to support more users at a higher request rate with more scalability.

NOTE

This document is not intended to be a complete Redis configuration guide. Refer to Redis documentation for more details on usage.

There are two options to configure ADQ in Redis server.

- Option 1: Using ADQ Setup Script
- Option 2: Using Linux Command Line (Manual Configuration)

Both configuration methods require the completion of the System Setup steps first.

NOTE

This section contains the steps for configuring ADQ using the setup script. For manual ADQ configuration, see Appendix: ADQ Configuration Reference (Manual)

10.1 Redis Server Setup

NOTE

This section uses application dependent polling (ADQ1.0) for application and testing and does not use an independent poller (ADQ2.0).

The following variables are used in the examples in this section:

$iface	The interface in use (PF).
$ipaddr	The IP Address of the Redis server interface under test.
$app_port	The starting port of the application (for example, 7001). *Note:* Port number greater than 5000 is best, but check that it is available (for example: sudo netstat -tulpn | grep LISTEN)
$app_threads	The number of Redis server instances to be run (must be less than or equal to **$num_queues_tc1**).
$redispath	The path to Redis install (for example, */opt/redis-4.0.11/bin*).
$clients	The array of hostnames or IPs of the physical Redis client systems (for example, declare -a clients=("server1" "server2")).
$redisbench_keysize	The Redis benchmark key size in bytes.
$redisbench_conn	The number of connections per Redis benchmark thread.

continued...

Did this document help answer your questions?

ADQ Setup Script variables	
$bp	Global busy_poll value to be set by ADQ Setup Script (system-wide setting, not to be used with independent poller).
$br	Global busy_read value to be set by ADQ Setup Script (system-wide setting, not to be used with independent poller).
$file_name	Application name to be set and used by ADQ Setup Script (any descriptive string, i.e., *redis*).
$portrange	TCP ports to be configured by ADQ Setup Script.
$addr	Destination IP address of traffic to be configured by ADQ Setup Script.

System Setup

1. Perform general system setup.

   Complete the ADQ install and setup in ADQ System Under Test (SUT) Installation and General System Tuning on just the server under test system.

   NOTE

   Many settings in General System Tuning do not persist between reboots and might have to be reapplied.

2. Perform Redis build:

   a. Download the Redis release.

   ```
   wget http://download.redis.io/releases/redis-4.0.11.tar.gz
   ```

   b. Untar the package.

   ```
   tar -xvzf redis-4.0.11.tar.gz
   ```

   c. Install Redis.

   ```
   cd redis-4.0.11
   make PREFIX=/opt/redis-4.0.11 install
   ```

   d. After this, the redis-server and redis-benchmark are found under */opt/redis-4.0.11/bin*.

3. Perform Redis-specific tuning.

   Enable overcommit and disable transparent hugepages as per Redis recommendations.

   ```
   sysctl -w vm.overcommit_memory=1
   echo never > /sys/kernel/mm/transparent_hugepage/enabled
   ```


Did this document help answer your questions?

Using ADQ Setup Script

The ADQ Setup script allows you to quickly and easily configure required ADQ parameters such as traffic classes, priority, filters, and ethtool parameters etc.

1. To configure ADQ, run the following command:

```
adqsetup --dev=$iface --priority=skbedit --busypoll=$bp --busyread=$br create
$file_name mode shared \
queues $num_queues_tc1 ports $portrange
```

See **Notes** below for customizing the ADQ configuration. Once ADQ is configured by **adqsetup**, start the Redis server.

NOTES

a. The example command above creates both ingress (Rx) and egress (Tx) filters, and a Linux cgroups are not needed to be created and can be skipped (cgroups are only needed if `--priority=skbedit` was NOT specified in **adqsetup** command).

b. ADQ setup script handles symmetric queues and affinity.

c. The setup script sets per-queue coalesce by default.

d. Set the transmit and receive interrupt coalescing values to `--rxadapt=off --rxusecs=0 --txadapt=off --txusecs=500` for improved ADQ performance.

e. To configure independent pollers, add the `-pollers=$pollers` parameter in the **adqsetup** command (and optionally `--pollers_timeout`), and **remove** the flags to set global `--busypoll=`**$bp** `--busyread=`**$br**.

f. Use the `cpu` parameter in the command to bind the independent pollers to specific CPU cores. Refer to ADQ Setup Using ADQ Setup Script for more information on pinning pollers to specific CPU cores.

g. The `--debug` parameter is optional, but it is useful for obtaining complete stack trace.

h. For more information on how to use the script, refer ADQ Setup Using ADQ Setup Script.

2. Start Redis clones and seed initial data.

Example:

```
server="${redispath}/redis-server --protected-mode no --loglevel warning \
--tcp-backlog 1023"
trafficgen="${redispath}/redis-benchmark"
ports=($(seq $app_port $((app_port + app_threads - 1))))
# start clones in the background then seed database
(
for p in ${ports[@]:1}; do
   sleep 1
   $server --port $p --slaveof \ 127.0.0.1 $app_port &
done

sleep 1

#seed initial data
$trafficgen -q -t SET,GET -n 5000 -r 1000 -d 100 -c 1 -P 1 -p $app_port
echo "** Ready!"
```


Did this document help answer your questions?

```
) &
# start master server in the foreground and wait for benchmark to start
$server --port ${ports[0]}
```

NOTE

Pinning the application processes to CPU cores using **numactl** or similar might provide more consistent results.

NOTE

The ADQ Setup script clears the existing configuration before proceeding with the new ADQ configuration. To clear manually, follow the steps in Clear the ADQ Configuration.

10.2 Redis Clients (non-ADQ) Configuration

The following variables are used in the examples in this section:

$iface	The interface in use.
$ipaddr	The IP Address of the Redis server interface under test.
$app_port	The starting port of the application (for example, 7001). *Note:* Port number greater than 5000 is best, but check that it is available (for example: sudo netstat -tulpn | grep LISTEN)
$app_threads	The number of Redis server instances to be run (must be less than or equal to **$num_queues_tc1**).
$pathtoicepackage	The path to the *ice* driver package.
$redispath	The path to Redis install (for example, */opt/redis-4.0.11/bin*).
$clients	The array of hostnames or IPs of the physical Redis client systems (for example, declare -a clients=("server1" "server2")).
$redisbench_keysize	The Redis benchmark key size in bytes.
$redisbench_conn	The number of connections per Redis benchmark thread.

1. Perform Client configuration.
   a. Enable throughput-performance tuned profile.

```
tuned-adm profile throughput-performance
```

NOTE

The *tuned-adm* daemon is not installed by default in RHEL 9.0 systems. Install it with the command `yum install tuned`.

Check settings are applied correctly:

```
cat /etc/tuned/active_profile
```


Did this document help answer your questions?

Output: `throughput-performance`

```
cat /etc/tuned/profile_mode
```

Output: `manual`

b. Stop the **irqbalance** service.

```
systemctl stop irqbalance
```

c. Run the **set_irq_affinity** script for all interfaces (included in the *scripts* folder of the *ice* package).

```
${pathtoicepackage}/scripts/set_irq_affinity -X all $iface
```

d. Set interrupt moderation rate to a static value for Tx and turn off interrupt moderation for Rx.

```
ethtool --coalesce $iface adaptive-rx off rx-usecs 0
ethtool --coalesce $iface adaptive-tx off tx-usecs 25
```

2. Perform Redis build.

a. Download the Redis release.

```
wget http://download.redis.io/releases/redis-4.0.11.tar.gz
```

b. Untar the package.

```
tar -xvzf redis-4.0.11.tar.gz
```

c. Apply *redis-benchmark.patch* for more detailed benchmark statistics and a duration (-D) command flag.

NOTE

If you are viewing this document in HTML, click the **Download PDF** button to download the zip package containing the *redis-benchmark.patch* patch.

```
cd redis-4.0.11/src
patch -p1 < redis-benchmark.patch
```

d. Install Redis.

```
cd redis-4.0.11
make PREFIX=/opt/redis-4.0.11 install
```

e. After this, the redis-server and redis-benchmark are found under */opt/redis-4.0.11/bin*.

3. Perform Redis benchmarking.

A typical test run would be one redis-benchmark instance per SUT queue/Redis instance, distributing the benchmark instances evenly across all the client systems. The request size (-d) can be scaled for testing as well as the connection rate per benchmark instances (-c).

Did this document help answer your questions?

For example, in the case of 90 queues and 10 client systems, there would be 90 instances of redis-server on the SUT listening on 90 different TCP ports, and 9 instances of redis-benchmark on each client, each connecting to a different port on the SUT. If a total of 900 connections/users were desired, then each of the 10 client benchmark threads would have a connection rate of -c 10.

To run the automated benchmarking script below, the server that the script is run on must have passwordless **ssh** access to all clients.

To set this up, run the following from the server that the script is going to be run on:

```
ssh-keygen
for client in ${clients[@]}; do ssh-copy-id $client; done
```

NOTE

Refer to https://redis.io/topics/benchmarks for more detail on Redis benchmarking.

Example:

```
trafficgen="${redispath}/redis-benchmark"
num_clients=${#clients[@]}
for ((i = 0; i < app_threads; i++)); do
   (ssh ${clients[$((i % num_clients))]} "$trafficgen -t GET -n 100000 -r
1000 \
-d $redisbench_keysize -c $redisbench_conn -P 1 -h $ipaddr -p $((app_port + \
i))") & done
```

NOTE

Pinning the application processes to CPU cores using **numactl** or similar might provide more consistent results.

Parameter Definitions:

```
'-t {tests}',
'-c {threads}',
'-n {requests}',
'-r {keys}',
'-P {pipeline}',
'-d {size}',
'-h {server}',
'-p {port}'
```


Did this document help answer your questions?

4. Verify ADQ traffic is on the correct queues.

   While ADQ application traffic is running, watch **ethtool** statistics to check that only the ADQ queues are being used (have significant traffic) with busy poll (pkt_busy_poll) for ADQ traffic. If non busy poll (pkt_not_busy_poll) have significant counts and/or if traffic is not confined to ADQ queues, recheck the configuration steps carefully.

   ```
   watch -d -n 0.5 "ethtool -S $iface | grep busy | column"
   ```

   See Verify ADQ Application Traffic and Independent Pollers (If Applicable) for example **watch** output.

NOTE

The ADQ Setup script clears the existing configuration before proceeding with the new ADQ configuration. To clear manually, follow the steps in Clear the ADQ Configuration.

Did this document help answer your questions?

11.0 Testing ADQ with Memcached/rpc-perf

Memcached is an in-memory key-value store for data. **rpc-perf** is a benchmark meant to help measure the performance particularly throughput (request rate) and Latency (percentile). For this section, Memcached is used as an example of a real-world application with ADQ.

NOTE

This document is not intended to be a complete Memcached/rpc-perf configuration guide. Refer to Memcached/rpc-perf documentation for more details on usage.

There are two options to configure ADQ in Memcached server.

- Option 1: Using ADQ Setup Script
- Option 2: Using Linux Command Line (Manual Configuration)

Both configuration methods require the completion of the System Setup first.

NOTE

This section contains the steps for configuring ADQ using the setup script. For manual ADQ configuration, see Appendix: ADQ Configuration Reference (Manual)

11.1 Memcached Server Setup

NOTE

This section uses application dependent polling (ADQ1.0) for application and testing and does not use an independent poller (ADQ2.0).

The following variables are used in the examples in this section:

$iface	The interface in use (PF).
$num_queues_tc0	The number of queues for default traffic (for example, 2).
$num_queues_tc1	The number of queues for Memcached traffic class.
$ipaddr	The IP Address of the interface under test.
$app_port	The port of the Memcached Server (for example, 11211 for Memcached server)
$mem	Memory block "Use <*num*> MB memory max to use for object storage; the default is 64 megabytes." (https://linux.die.net/man/1/memcached) So the total memory in megabytes used for storage.
$memcachedpath	The path to Memcached install (for example, */opt/memcached-1.6.9-<REV>*).

continued...

Did this document help answer your questions?

$libeventdevelpath	The path to libevent-devl library (for example, */usr/lib64/*).
ADQ Setup Script Variables	
$bp	Global busy_poll value to be set by ADQ Setup Script (system-wide setting, not to be used with independent poller).
br	Global busy_read value to be set by ADQ Setup Script (system-wide setting, not to be used with independent poller).
$file_name	Application name to be set and used by ADQ Setup Script (any descriptive string, i.e., *memcached*).
$port	TCP port to be configured by ADQ Setup Script.
$addr	Destination IP address of traffic to be configured by ADQ Setup Script

System Setup

1. Perform general system OS install and setup.

   Complete the ADQ install and setup in ADQ System Under Test (SUT) Installation and General System Tuning on just the SUT system.

   NOTE

   Many settings in General System Tuning do not persist between reboots and might need to be reapplied.

2. Install `libevent-devel` prerequisite package.

   Example on RHEL 8.x:

```
yum install -y libevent-devel
```

   NOTE

   For RHEL 7.x, libevent-devel is not packaged with the OS iso, it is available in the Server-optional package for the release.

3. Perform Memcached/rpc-perf build:

   a. Download the latest **Memcached** release.

      Example:

```
wget https://memcached.org/files/memcached-1.6.9.tar.gz
```

   b. Untar the package.

```
tar -xvzf memcached-1.6.9.tar.gz
```

   c. Install **Memcached**.

```
i =${memcachedpath}

./configure -prefix=${memcachedpath}\  --with-libevent=$
{libeventdevelpath}  make install
```

      After this, the Memcached server is found under ***${memcachedpath}***/*bin*.

Did this document help answer your questions?

d. Download the **rpc-perf** release.

```
cd /opt
git clone https://github.com/twitter/rpc-perf.git
```

e. Install **rpc-perf**.

```
yum install cargo
cd rpc-perf/
cargo build --release
cp ../target/release/rpc-perf /usr/local/bin
chmod +x /usr/local/bin/rpc-perf
```

Using ADQ Setup Script

The ADQ Setup script allows you to quickly and easily configure required ADQ parameters such as traffic classes, priority, filters, and ethtool parameters etc.

1. To configure ADQ, run the following command:

```
adqsetup --dev=$iface --priority=skbedit --busypoll=$bp --busyread=$br create
$file_name queues \
$num_queues_tc1 ports $app_port addrs $addr
```

See **Notes** below for customizing the ADQ configuration. Once ADQ is configured by **adqsetup**, start the Memcached server.

NOTES

a. The example command above creates both ingress (Rx) and egress (Tx) filters, and Linux cgroups are not needed to be created and can be skipped (cgroups are only needed if `--priority=skbedit` was NOT specified in **adqsetup** command).

b. ADQ setup script handles symmetric queues and affinity.

c. The setup script sets per-queue coalesce by default.

d. The setup script sets per-TC inline flow director, for ice-1.8.x and later. For earlier versions of the ice driver, the global *channel-inline-flow-director* flag is used by default.

e. Set the transmit and receive interrupt coalescing values to `--rxadapt=off --rxusecs=0 --txadapt=off --txusecs=500` for improved ADQ performance.

f. To configure independent pollers, add the `-pollers=$pollers` parameter in the **adqsetup** command (and optionally `--pollers_timeout`), and **remove** the flags to set `global --busypoll=`**`$bp`** `--busyread=`**`$br`**.

g. Use the `cpu` parameter in the command to bind the independent pollers to specific CPU cores. Refer to ADQ Setup Using ADQ Setup Script for more information on pinning pollers to specific CPU cores.

h. The `--debug` parameter is optional, but it is useful for obtaining complete stack trace.

i. For more information on how to use the script, refer to ADQ Setup Using ADQ Setup Script.

Did this document help answer your questions?

2. Seeding Memcached data prior to beginning benchmarking is best done with a python library for Memcached called *pymemcache*. Download and install from:

   https://github.com/pinterest/pymemcache

   This requires pip, which can be available when installing Python 3 setup tools.

3. Start **Memcached** server and seed data to prepare database.

   From the SUT, prepare the Memcached database for benchmarking by seeding the database, using a Python script to generate large number of keys and values. For simplicity, this example shows a single threaded pymemcache client, but this could be expanded easily to multiple threads.

   NOTE

   The goal of seeding the database is just to write the key value pairs, so the number of threads used in the seeding process does not matter.

   For this example, key counts are 20000000, with key size of 8 bytes and value size of 64 bytes. To ensure 100% hit-rate so that every request has a value returned, each key has the format '00000001','00000002'...N. with N being the total number of keys. Each key value is the random string of length valsize.

   Example Memcached start script (calls pymemcache Python script):

```
simple_memcached_start.sh
iface="p1p2"
num_queues_tc1=32
ipaddr=" 13.100.2.22"
port=11211
mem=30000
keycount=20000000
keysize=8
valsize=64
server="${memcachedpath}/bin/memcached"

client="./singlethread_seed.py -s ${ipaddr} -p ${port} -c ${keycount} -k
${keysize} -v ${valsize}"

# seed database in the background

(sleep 2; ${client}; echo "*** Done seeding, ready to start client side
benchmark!") &

# start server in the foreground

${server} -u root -p ${port} -c 8192 --
napi-ids=${num_queues_tc1} -t ${num_queues_tc1} -m ${mem} --disable-evictions
```

   NOTE

   The `--napi-ids` option was added in **Memcached** version 1.6.9 to provide another thread selection option in the *dispatch_conn_new* code path. The default mechanism for thread selection is round-robin. The flag enables NAPI_ID based thread selection that creates an association between socket(s) traffic that arrives on a given NIC (hardware) queue and the Memcached thread that consumes this traffic. This mapping between a Memcached thread and a HW NIC queue streamlines the flow of data from the NIC to the application. In addition, an optimal path with reduced context switches is possible, if epoll-based busy polling (`sysctl -w net.core.busy_poll = <non-zero value>`) is also enabled.

Did this document help answer your questions?

Example Python3 pymemcache script:

```
singlethread_seed.py
#!/usr/bin/env python3
import os
import sys
import argparse
import random
import string
import psutil
from pymemcache.client import base

def set_key_range(ipaddr, port, keycount, keysize, valsize):
        client = base.PooledClient((ipaddr, port), connect_timeout=2)
        value = random_string(string_length=valsize)
        for i in range(1, keycount):
                key = str(i).zfill(keysize) #zero padding, '0000001'
                        client.set(key, value)

def random_string(string_length=64):
        letters = string.ascii_lowercase
        return ''.join(random.choice(letters) for i in range(string_length))

def main(argv):
    parser = argparse.ArgumentParser()
    parser.add_argument('-s', '--server', type=str,
                        default="13.100.2.22", help="Server hostname or IP Address")
    parser.add_argument('-p', '--port', type=int, default=11211,
                         help="TCP port server is listening on. Default is 11211.")
    parser.add_argument('-k', '--keysize', type=int, default=8,
                         help="The keysize in bytes.")
    parser.add_argument('-c', '--keycount', type=int, default=20000000,
                         help="Number of keys to produce.")
    parser.add_argument('-v', '--valsize', type=int, default=64,
                         help="The value size in bytes.")
    parser.add_argument('-t', '--processs', type=int, default=44,
                         help="Number of parallel thread creating keys and values.")
    args = parser.parse_args()
    set_key_range(args.server, args.port, args.keycount, args.keysize,
args.valsize)

if __name__ == "__main__":
    main(sys.argv)
```

NOTE

The ADQ Setup script clears the existing configuration before proceeding with the new ADQ configuration. To clear manually, follow the steps in Clear the ADQ Configuration.

Did this document help answer your questions?

11.2 Memcached/rpc-perf Clients (non-ADQ) Configuration

The following variables are used in the examples in this section:

$ipaddr	The IP Address of the interface under test.
$threads	The number of connections per **rpc-perf** instance.
$duration	Test time.

1. Perform general system OS install and setup.

   Minimum required kernel for Memcached/rpc-perf (non-ADQ) clients is v4.19.18.

   Complete Install OS and Update Kernel (If Needed). Update the Kernel on each of the rpc-perf client systems, using kernel v4.19.18 or later.

2. Perform **rpc-perf** build:

   a. Download the **rpc-perf** release on each client system.

   ```
   git clone https://github.com/twitter/rpc-perf.git
   ```

   b. Install **rpc-perf** on each client system.

   ```
   cargo build --release
   cp rpc-perf/target/release/rpc-perf /usr/local/bin
   chmod +x /usr/local/bin/rpc-perf
   ```

3. Perform Memcached/rpc-perf benchmarking.

   A test run would be one **rpc-perf** instance per physical client. The sum of the connections that can be spawned for all clients/instances should be greater than or equal to the total number of TC1 queues available at the SUT/Memcached server.

   For example, in the case of a Memcached server with 110 TC1 queues and 10 physical client systems, there would be 10 instances of **rpc-perf** (one running on each physical client) and 11 threads for each instance.

   Other parameters would include 8-byte key, 64-byte value, 100% read, and 20 million total keys.

   We can start a **rpc-perf** instance on each client via a command line parameter.

   Example:

   ```
   rpc-perf --config rpc-perf-memcached.toml --endpoint $ipaddr:$port --interval
   $duration --clients $threads
   ```

   This needs to be concurrently started on all clients. We can also use the TOML file to configure these values instead of command line.

   Example TOML file `rpc-perf-memcached.toml` (replace *<threads>* with the number of client threads per instance and *<interval>* with the length of time of each test):

   ```
   [general]
   protocol = "memcache"
   interval = <interval> # seconds
   windows = 1 # run for 1 intervals
   clients = <threads>
   poolsize = 1 # each client has 1 connection per endpoint
   ```


Did this document help answer your questions?

```
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 = 20000000 # limit to 20M keys
weight = 1 # this keyspace has a weight of 1
commands = [ # get:set ratio is 1
   {action = "get", weight = 0},
   {action = "set", weight = 1},
]
values = [ # value length will always be 64 bytes
   {length = 64, weight = 1},
]
```

4. Verify that ADQ traffic is on the correct queues.

   While ADQ application traffic is running, watch **ethtool** statistics to check that only the ADQ queues are being used (have significant traffic) with busy poll (pkt_busy_poll) for ADQ traffic. If non busy poll (pkt_not_busy_poll) have significant counts and/or if traffic is not confined to ADQ queues, recheck the configuration steps carefully.

   ```
   watch -d -n 0.5 "ethtool -S $iface | grep busy | column"
   ```

   See Verify ADQ Application Traffic and Independent Pollers (If Applicable) for example **watch** output.

NOTE

The ADQ Setup script clears the existing configuration before proceeding with the new ADQ configuration. To clear manually, follow the steps in Clear the ADQ Configuration.

Did this document help answer your questions?

12.0 Testing ADQ with NGINX

NGINX is a commonly-used, high-performance web server. wrk is a benchmark meant to help measure the performance of web servers, particularly throughput (request rate) and latency (percentile). For this section, NGINX is used as an example of a real-world application with ADQ.

NOTE

This document is not intended to be a complete NGINX/wrk configuration guide. Refer to NGINX/wrk documentation for more details on usage.

Throughout this section, the steps that are required for ADQ only are highlighted with an [ADQ only] flag. If benchmarking a baseline configuration, follow all other steps and make sure those [ADQ only] steps are omitted/reverted.

There are two options to configure ADQ in NGINX server.

- Option 1: Using ADQ Setup Script
- Option 2: Using Linux Command Line (Manual Configuration)

Both configuration methods require the completion of the System Setup first.

NOTE

This section contains the steps for configuring ADQ using the setup script. For manual ADQ configuration, see Appendix: ADQ Configuration Reference (Manual)

12.1 NGINX Server Setup

NOTE

This section uses application dependent polling (ADQ1.0) for application and testing and does not use an independent poller (ADQ2.0).

The following variables are used in the examples in this section:

$iface	The interface in use (PF).
$num_queues_tc0	The number of queues for default traffic (note that this should be a power of 2, for example, 2).
$num_queues_tc1	The number of queues for application traffic class (this is the number of application threads).
$ipaddr	The IP Address of the interface under test.
$app_port	The TCP port of the **nginx** application being run on the SUT (for example, 80 to listen NGINX HTTP server).

continued...

Did this document help answer your questions?

$application	The path to application install (for example, */opt/NGINX*).
ADQ Setup Script Variables	
$bp	Global busy_poll value to be set by ADQ Setup Script (system-wide setting, not to be used with independent poller).
$br	Global busy_read value to be set by ADQ Setup Script (system-wide setting, not to be used with independent poller).
$file_name	Application name to be set and used by ADQ Setup Script (any descriptive string, i.e., *nginx*).
$port	TCP port to be configured by ADQ Setup Script.
$addr	Destination IP address of traffic to be configured by ADQ Setup Script

System Setup

1. Perform general system OS install and setup.

   Complete the ADQ install and setup in ADQ System Under Test (SUT) Installation and General System Tuning on just the SUT system.

   NOTE

   For best NGINX performance with ADQ, the following default *ice* driver settings should be used on the NGINX Server:

   ```
   ethtool --set-priv-flags $iface channel-pkt-inspect-optimize on
   ethtool --set-priv-flags $iface channel-pkt-clean-bp-stop-cfg off
   ethtool --set-priv-flags $iface channel-pkt-clean-bp-stop off
   ```

   The above settings can also be set using an ADQ Setup script; for more information, see Using ADQ Setup Script.

   One change to the system tuning is required. Instead of tuned-adm profile latency-performance, we recommend using the throughput-performance profile.

   ```
   tuned-adm profile throughput-performance
   ```

   NOTE

   The *tuned-adm* daemon is not installed by default in RHEL9.0 systems. Install it with the command `yum install tuned`.

   Check that the settings are applied correctly:

   ```
   cat /etc/tuned/active_profile
   ```

   Output: `throughput-performance`

   ```
   cat /etc/tuned/profile_mode
   ```

   Output: `manual`

Did this document help answer your questions?

NOTE

Many settings in General System Tuning do not persist between reboots and might need to be reapplied.

2. Perform NGINX build:
   a. Install prerequisite NGINX package(s):

```
yum install -y pcre-devel
```

   b. Download the latest echo-nginx-module. The echo module is a testing utility that allows for multi-worker content access without file contention, improving simulated benchmark performance. Under a real workload (not simulated for benchmarking) this module would not be necessary for full ADQ performance benefit.

      Example:

```
git clone https://github.com/openresty/echo-nginx-module.git
```

   c. Download the latest stable NGINX release.

      Example:

```
wget http://nginx.org/download/nginx-1.16.1.tar.gz
```

   d. Untar the package.

```
tar -xf nginx-1.16.1.tar.gz
```

   e. **[ADQ only]** Apply the patch *nginx_reuseport_bpf.patch*.

      NOTE

      If you are viewing this document in HTML, click the **Download PDF** button to download the zip package containing the *nginx_reuseport_bpf* patch.

```
cd nginx-${VER}
patch -p1 < <pathtopatch>/nginx_reuseport_bpf.patch
```

      NOTE

      This *nginx_reuseport_bpf* patch is required in an ADQ environment, but might provide benefit even without other ADQ configuration (such as TCs and cgroups). The *nginx_reuseport_bpf* patch establishes a 1:1 model of application thread to hardware queue use in NGINX by adding the socket option SO_ATTACH_REUSEPORT_CBPF to the application's listening socket code. This socket option allows the application to attach a small classic Berkley Packet Filter (cBPF) program to perform hardware NIC queue-aware load balancing among the application threads.

Did this document help answer your questions?

f. Configure NGINX with applied patch.

```
make clean
./configure --prefix=$application --add-module=<pathtoecho-nginx>/echonginxmodule
```

g. Copy NGINX configuration into build so that it is deployed in the install.

```
cp -f nginx.conf conf/
```

NOTE

NGINX has its own set of configuration parameters to tune. Each application deployment might be different according to its individual use case. For full ADQ performance, the `use epoll` statement is required in the events {} section. To enable the *nginx_reuseport_pbf* patch, `listen $app_port reuseport` is required in the server {} section. An example nginx.conf file is included below.

Example nginx.conf (note the $app_port variable needs to match with the application $app_port in the rest of the section):

```
daemon off;
error_log stderr;

events {
    use epoll;
    worker_connections  1024;
}

http {
    include       mime.types;
    default_type  application/octet-stream;
    sendfile       on;
    tcp_nopush     on;
    tcp_nodelay    on;
    open_file_cache max=1000 inactive=10m;

    keepalive_timeout  120;
    keepalive_requests 1000000;

    server {
        listen       $app_port reuseport;
        server_name  localhost;

        access_log off;

        location / {
            default_type text/html;
            echo "<html><body><h2>Test Server</h2></body></html>";
        }

        location /test256.dat {
            echo_duplicate 256 "*";
        }

        location /test512.dat {
            echo_duplicate 512 "*";
        }


        location /test1024.dat {
            echo_duplicate 1024 "*";
        }
```


Did this document help answer your questions?

```
        location /test2048.dat {
            echo_duplicate 2048 "*";
        }

        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   html;
        }
    }
}
```

h. Install NGINX.

```
make install
```

After this, the NGINX server is found under ***$application***/*sbin*.

Using ADQ Setup Script

The ADQ Setup script allows you to quickly and easily configure required ADQ parameters such as traffic classes, priority, filters, and ethtool parameters etc.

1. To configure ADQ, run the following command:

```
adqsetup --dev=$iface --priority=skbedit --busypoll=$bp create $file_name queues
$num_queues_tc1 ports \
$port addrs $addr
```

See **Notes** below for customizing the ADQ configuration. Once ADQ is configured by **adqsetup**, start the nginx server.

Did this document help answer your questions?

NOTES

a. The example command above creates both ingress (Rx) and egress (Tx) filters, and Linux cgroups are not needed to be created and can be skipped (cgroups are only needed if `--priority=skbedit` was NOT specified in **adqsetup** command).

b. ADQ Setup script handles symmetric queues and affinity.

c. The Setup script sets per-queue coalesce by default.

d. The Setup script sets per-TC inline flow director, for ice-1.8.x and later. For earlier versions of the ice driver, the global *channel-inline-flow-director* flag is used by default.

e. Set the transmit and receive interrupt coalescing values to `--rxadapt=off --rxusecs=0 --txadapt=off --txusecs=500` for improved ADQ performance.

f. Set the ice driver settings to `--optimize on --bpstop off --bpstop-cfg off` for better NGINX performance with ADQ.

g. To configure independent pollers, add the `-pollers=$pollers` parameter in the **adqsetup** command (and optionally `--pollers_timeout`), and **remove** the flags to set global `--busypoll=`**`$bp`** `--busyread=`**`$br`**.

h. Use the `cpu` parameter in the command to bind the independent pollers to specific CPU cores. Refer to ADQ Setup Using ADQ Setup Script for more information on pinning pollers to specific CPU cores.

i. The `--debug` parameter is optional, but it is useful for obtaining complete stack trace.

j. For more information on how to use the script, refer to ADQ Setup Using ADQ Setup Script.

Did this document help answer your questions?

2. Start NGINX either in the cgroup for ADQ, or without the cgroup if testing baseline performance.
   a. **[ADQ only]** Start NGINX in the cgroup with the same number of threads as were configured for TC1.

      Example:

      ```
      server="${application}/sbin/nginx"
      ${server} -g "worker_processes ${num_queues_tc1};error_log stderr info;"
      ```

   b. **[Baseline]** Start NGINX.

      Example:

      ```
      server="${application}/sbin/nginx"
      ${server} -g "worker_processes ${num_queues_tc1};error_log stderr info;"
      ```

3. **[ADQ only]** Troubleshooting — While test is running, verify that ADQ traffic is on the correct queues.

   While ADQ application traffic is running, watch **ethtool** statistics to check that only the ADQ queues are being used (have significant traffic) with busy poll (pkt_busy_poll) for ADQ traffic. If non busy poll (pkt_not_busy_poll) have significant counts and/or if traffic is not confined to ADQ queues, recheck the configuration steps carefully.

   ```
   watch -d -n 0.5 "ethtool -S $iface | grep busy | column"
   ```

   See Verify ADQ Application Traffic and Independent Pollers (If Applicable) for example **watch** output.

   NOTE

   Note:If the benefits of ADQ enabled vs. disabled are not as dramatic as expected even though the queues appear to be well aligned, it is possible that the performance is limited by the processing power of the client systems. If client CPU shows greater than an average of 80% CPU utilization on the CPU cores in use, it is probable that the client is becoming overloaded. To achieve maximum performance benefits, try increasing the number of or processing power of the client systems.

NOTE

The ADQ Setup script clears the existing configuration before proceeding with the new ADQ configuration. To clear manually, follow the steps in Clear the ADQ Configuration.

Did this document help answer your questions?

12.2 wrk Clients Configuration

NOTE

ADQ is not required for the clients. The following example sets up clients without ADQ for use in benchmarking the ADQ or non-ADQ NGINX SUT server performance using the wrk benchmark.

The following variables are used in the examples in this section:

$iface	The interface in use.
$duration	Test time in format <value><unit> (example: 60s for 60 seconds, 3m for 3 minutes).
$client_num	The total number of physical systems being used as clients.
$pathtoicepackage	The path to the *ice* driver package.
$connection_scale	The number of connections per client thread.
$num_queues_tc1	From the SUT, the number of queues for application traffic class on the SUT.
$ipaddr	From the SUT, the IP Address of the application server's interface under test.
$app_port	From the SUT, the TCP port of the **nginx** application being run on the SUT.

1. Perform **wrk** build:
   a. Download **wrk**.

```
git clone --branch 4.1.0 https://github.com/wg/wrk.git wrk
```

   b. Install **wrk**.

```
yum install -y openssl-devel
cd wrk
make
cp wrk /usr/local/bin
```

2. Perform Client configuration.
   a. Enable throughput-performance tuned profile.

```
tuned-adm profile throughput-performance
```

NOTE

The *tuned-adm* daemon is not installed by default in RHEL9.0 systems. Install it with the command `yum install tuned`.

Check settings are applied correctly:

```
cat /etc/tuned/active_profile
```

Output: `throughput-performance`

```
cat /etc/tuned/profile_mode
```


Did this document help answer your questions?

Output: `manual`

b. Stop the **irqbalance** service.

```
systemctl stop irqbalance
```

c. Run the **set_irq_affinity** script for all interfaces (included in the scripts folder of the *ice* package).

```
${pathtoicepackage}/scripts/set_irq_affinity -x all $iface
```

d. Set limits file in */etc/security/limits.conf* to include:

```
hard       nofile       32768
soft       nofile       32768
```

3. Perform **wrk** benchmarking.

A test run would be one **wrk** instance per physical client system, each with multiple application threads. Due to the design of **wrk** itself, it is recommended to keep the number of threads of the client benchmark per client system less than or equal to the number of physical local CPU cores on the client systems. For more predictable results, it is also recommended to run the client benchmark with the total number of threads from all client systems equal to an even multiple of the number of application threads on the SUT. This ensures an even distribution of client load.

For example, in the case of an ADQ enabled NGINX server with 110 TC1 queues and 10 physical client systems each with 24 physical CPU cores per socket, a recommended configuration would be 10 instances of **wrk** (one running on each physical client), each with 22 threads. The number of connections can then be linearly scaled to increase the workload evenly across the SUT.

NOTE

Average latency is just a measure of how fast the SUT can respond.

If the SUT is not overloaded then it does not *need* ADQ to have a decent latency number.

ADQ shows benefit when you scale up the system beyond what it would normally be able to handle.

We can start a **wrk** instance on each client concurrently via a command line script (example below) and then the results added together.

Example:

NOTE

$num_queues_tc1, **$ipaddr**, and **$app_port** are the same numbers used from the SUT configuration

```
file="test${buffer_size}.dat" threads=$(( 2 * $num_queues_tc1 /
$client_num  )) conns=$(( $threads * $connection_scale ))
numactl --cpunodebind=netdev:$iface --membind=netdev:$iface wrk --latency -t
${threads} -c ${conns} -d ${duration} "http://${ipaddr}:${app_port}/${file}"
```


Did this document help answer your questions?

13.0 Testing ADQ with AF_XDP

XDP (eXpress Data Path) is a framework that uses eBPF, an extended version of the Berkley Packet Filter (BPF), to perform high-speed packet processing within the Linux kernel. XDP applications can be attached directly to a network interface, to create efficient packet processing directly from the network device, allowing the XDP application to bypass the network stack in the kernel. AF_XDP is an extension to the existing XDP support in the Linux kernel for XDP enabled network devices.

AF_XDP is an address family optimized for high performance, low latency, packet processing with XDP. AF_XDP was introduced in the 4.18 kernel and is a new socket type that permits raw packet data from the network adapter to be delivered straight to user space from XDP without any copying, and at significantly higher speeds. AF_XDP does not completely bypass the kernel but utilizes its functionality to create something similar to DPDK or the AF_Packet. AF_Packet is a socket in the Linux kernel, which allows applications to send and receive raw packets through the kernel.

AF_XDP uses Linux kernel drivers, it does not use any user-space drivers. This support is available for customer experimentation and testing but has not been fully tested with all possible AF_XDP workloads. For convenience, instructions for testing with kernel sample application XDPSOCK are included in this section.

There are two options to configure ADQ in AF_XDP server.

- Option 1: Using ADQ Setup Script
- Option 2: Using Linux Command Line (Manual Configuration)

Both configuration methods require the completion of the System Setup first.

NOTE

This section contains the steps for configuring ADQ using the setup script. For manual ADQ configuration, see Appendix: ADQ Configuration Reference (Manual)

Did this document help answer your questions?

13.1 AF_XDP Server Configuration

NOTE

This section uses application dependent polling (ADQ1.0) for application and testing and does not use an independent poller (ADQ2.0).

The following variables are used in the examples in this section:

$iface	The interface in use (PF or VF)
$num_queues_tc0	The number of queues for default traffic (for example, 2).
$num_queues_tc1	The number of queues for application traffic (also maximum number of threads to be run).
$ipaddr	The IP Address of the server interface under test.
$app_port	The starting port of the application (for example, 7000). *Note:* Port number greater than 5000 is best, but check that it is available (for example: sudo netstat -tulpn | grep LISTEN).
$dst_ip	The destination IP Address created on each ADQ TC filter.
$baseaddr	Holds the first 3 octants of IP Address.
$lsv	Holds the 4th octant of IP Address.
$def_app_port	The port for the default TC/queues.
$def_ipaddr	The destination IP Address for the default queues.
$kernelpath	The path to the installed kernel.
ADQ Setup Script Variables	
$bp	Global busy_poll value to be set by ADQ Setup Script (system-wide setting, not to be used with independent poller).
$br	Global busy_read value to be set by ADQ Setup Script (system-wide setting, not to be used with independent poller).
$file_name	Application name to be set and used by ADQ Setup Script (any descriptive string, i.e., *af_xdp*).
$portrange	TCP ports to be configured by ADQ Setup Script
$addr	Destination IP address of traffic to be configured by ADQ Setup Script

System Setup

1. Perform the system OS install and setup.
   a. Complete the OS install and kernel update inADQ System Under Test (SUT) Installation on the server under the test system.

      NOTE

      For busy polling, kernel version v5.11.0 or later is required.

Did this document help answer your questions?

NOTE

Ensure XDP socket flag is enabled in the kernel *.config* file in Step 4c in Update Kernel (If Needed)

Add `CONFIG_XDP_SOCKETS=y flag` in the *.config* file.

If the kernel is compiled without `CONFIG_XDP_SOCKETS=y`, the following error message is observed while running the XDPSOCK application: `Address family not supported by protocol`.

NOTE

Disable Hyper threading in BIOS for better performance with AF_XDP.

b. Complete the ADQ setup in General System Tuning on the SUT system.

One change to the system tuning is recommended for improved performance with ADQ. Set the following private flags on the interface under test:

```
ethtool --set-priv-flags $iface channel-pkt-inspect-optimize off
ethtool --set-priv-flags $iface channel-pkt-clean-bp-stop on
ethtool --set-priv-flags $iface channel-pkt-clean-bp-stop-cfg on
```

The above settings can also be set using an ADQ Setup script; for more information, see Using ADQ Setup Script below.

2. Install prerequisites: Packages needed for LLVM and BPF samples: `cmake`, `libcap-devel`.

```
yum install -y cmake libcap-devel
```

If *libpcap-devel* is not available in the OS distribution, download and install *libpcap* and *libpcap-devel*:

```
wget http://mirror.centos.org/centos/8/BaseOS/x86_64/os/Packages/
libpcap-1.9.1-5.el8.x86_64.rpm
wget http://mirror.centos.org/centos/8/PowerTools/x86_64/os/Packages/libpcap-
devel-1.9.1-5.el8.x86_64.rpm
rpm -i libpcap-devel*
```

NOTE

If installing packages using the **rpm** command, and if *libpcap* is already installed on the system, use **rpm -U** instead of **rpm -i** to upgrade the package. Version numbers might differ depending on OS being used.

3. Install LLVM tool.

a. Download LLVM.

```
wget https://github.com/llvm/llvm-project/releases/download/
llvmorg-10.0.1/llvm-project-10.0.1.tar.xz
```


Did this document help answer your questions?

b. Untar the package.

```
tar -xf llvm-project-10.0.1.tar.xz
```

c. Install LLVM.

```
cd to the directory "llvm-project-10.0.1" mkdir build;
cd build cmake -G "Unix Makefiles" -DLLVM_ENABLE_PROJECTS=clang ../llvm
make -j"$(nproc)" install
```

NOTE

`$(nproc)` = number of logical cores.

NOTE

LLVM requires a lot of disk space, be sure to have at least 65 GB of disk space available before building and installing LLVM.

4. Build BPF Samples.

a. **cd** to the Linux kernel source (for example, */kernels/5.11.0/*) and build tools.

```
cd $kernelpath
make -C tools clean
```

b. Build sample bpf:

```
make M=samples/bpf
```

or

```
cd $kernelpath/samples/bpf
make -j all
```

c. Verify that the **xdpsock** executable was created:

```
ls $kernelpath/samples/bpf | grep xdpsock
xdpsock
xdpsock_ctrl_proc
xdpsock_ctrl_proc.c
xdpsock_ctrl_proc.o
xdpsock.h
xdpsock_kern.c
xdpsock_kern.o
xdpsock_user.c
xdpsock_user.o
```


Did this document help answer your questions?

Using ADQ Setup Script

The ADQ Setup script allows you to quickly and easily configure required ADQ parameters such as traffic classes, priority, filters, and ethtool parameters etc.

1. To configure ADQ, run the following command:

```
adqsetup --dev=$iface --priority=skbedit  --busypoll=$bp --busyread=$br
create $file_name mode shared \
queues $num_queues_tc1 ports $portrange addrs $addr protocol udp
```

See **Notes** below for customizing the ADQ configuration. Once ADQ is configured by **adqsetup**, launch the XDPSOCK application with multiple parallel sockets in different modes `l2fwd/TX-only/RX-drop`.

NOTES

a. ADQ Setup script handles symmetric queues and affinity.

b. The Setup script sets per-queue coalesce by default.

c. Set the transmit and receive interrupt coalescing values to `--rxadapt=off --rxusecs=0 --txadapt=off --txusecs=500` for improved ADQ performance.

d. Set the ice driver settings to `--optimize off --bpstop on --bpstop-cfg on` for better performance with ADQ.

e. To configure independent pollers, add the `-pollers=$pollers` parameter in the **adqsetup** command (and optionally `--pollers_timeout`), and **remove** the flags to set global `--busypoll=`**$bp** `--busyread=`**$br**.

f. Use the `cpu` parameter in the command to bind the independent pollers to specific CPU cores. Refer to ADQ Setup Using ADQ Setup Script for more information on pinning pollers to specific CPU cores.

g. The `--debug` parameter is optional, but it is useful for obtaining complete stack trace.

h. For more information on how to use the script, refer ADQ Setup Using ADQ Setup Script.

2. Create and run a test script like the example below. The script opens multiple parallel sockets in the background with the selected mode: `l2fwd/TX-only/RX-drop`.

Example:

```
xdpsock.sh
mode=$1
if [ "x${mode}" == "x" ]
then
    echo -e "Enter mode in which the script should run: -l for L2FWD or -r
for RX-DROP or -t for TX-ONLY:"
    echo -e "Usage: ./xdpsock.sh -l or ./xdpsock.sh -r or ./xdpsock.sh -t"
    exit 1
fi
inst=${2:-<number_of_queues_to_test>}
iface=ens1f1
kernelpath=/kernels/5.11.0/
XDPSOCK="${kernelpath}/samples/bpf/xdpsock"
trap 'kill $(jobs -p)' EXIT
max=$(( inst + 2 ))
```


Did this document help answer your questions?

```
for (( i = 2; i < $max; i++ ))
do
    printf -v q "%d" $((i*1))
    taskset -c $i $XDPSOCK -i $iface ${mode} -q $i & #mode is -l for l2fwd, -
r for RX-DROP, -t for TX-ONLY
    sleep 0.1
done
while true
do
    sleep 1
done
```

3. Run the traffic from an external packet generator of choice. See Traffic Generator Example Configuration (Non-ADQ) for details.
4. After the test finishes, kill the XDPSOCK process.

```
killall xdpsock
```

NOTE

The ADQ Setup script clears the existing configuration before proceeding with the new ADQ configuration. To clear manually, follow the steps in Clear the ADQ Configuration.

13.2 Traffic Generator Example Configuration (Non-ADQ)

To test AF_XDP functionality with ADQ, it is recommended to run multiple streams of traffic to the ADQ-configured AF_XDP server. There are multiple ways to generate the traffic, some are hardware based (IXIA, Spirent, etc.) and others are software only.

13.2.1 Traffic Pattern to Be Generated

To test AF_XDP as in the example given in Example: Running DPDK-pktgen Traffic, a packet generator of choice can be used to generate test traffic to the ADQ server with the following characteristics:

- UDP protocol to connect to AF_XDP sockets on the server.
- Destination port must match the **$app_port** number used in the ADQ configuration on the AF_XDP server (TC filter and ntuple configuration commands in AF_XDP Server Configuration).
- To use all ADQ queues on the AF_XDP server, the number of traffic streams (ports) should be equal to or a multiple of the number of ADQ queues configured on the AF_XDP server (**$num_queues_tc1** in AF_XDP Server Configuration).
- Other variables, such as **$packet_size**, can be configured to test the desired traffic pattern.

13.2.2 Example: Running DPDK-pktgen Traffic

This section provides an example for how to configure a standard server with a DPDK-pktgen software packet generator to generate network traffic to exercise the AF_XDP server configured in AF_XDP Server Configuration. Note that the server generating the traffic does not have to be configured with ADQ.

Did this document help answer your questions?

NOTE

This is not intended to be a complete DPDK-pktgen configuration guide. Refer to the *Intel® Ethernet Controller E810 Data Plane Development Kit (DPDK) 21.08 Configuration Guide* for more details on DPDK and pktgen installation, configuration, and usage.

The following are the attributes used to generate the traffic:

$pci_id	PCIe address of the interface in use.
$cores	List of cores used to generate the traffic.
$mem_channels	Number of memory channels. *Note:* The `-n 2` is a minimum required argument for DPDK (Range 2-4).
$socket_mem	DPDK allocated memory from each CPU.
$log_level	Set the specific log level (example: 1-8).
$protocol	Traffic protocol (AF_XDP must use UDP).
$packet_size	Size of the packet (example: 64-1518).
$src_mac	Client interface MAC Address.
$dst_mac	AF_XDP server interface MAC Address.
$src_ip	IP Address configured on the client interface using **pktgen set** command.
$dst_ip	IP Address of the AF_XDP server interface.
$src_port	Port number configured on client interface using **pktgen set** command.
$dst_port	Port number configured in the traffic class filters on the AF_XDP server.
$min_value	Minimum range value configured using **pktgen range** command.
$max_value	Maximum range value configured using **pktgen range** command.
$increment	Increment value configured using **pktgen range** command.
$RTE_SDK	Points to the DPDK installation directory. *Note:* Refer to DPDK configuration guide for more details.
$pathtopktgen	Path to pktgen application. *Note:* Refer to DPDK configuration guide for more details.
$file_prefix	Hugepage filename prefix.
$port_num	Interface port number (example: port 0,1).
$rx_cores	Cores to handle specific port's Rx.
$tx_cores	Cores to handle specific port's Tx.
Notes: • **$protocol** must be set to UDP to connect to AF_XDP sockets on the server. • For best performance, **$cores** should be set to CPU cores on the same (*local*) NUMA node as the E810 network adapter. See Step are from NUMA node 0 inventory. The following command can be used to list inventory of available NUMA nodes on the system, and the respective CPU cores: numactl -hardware (or) lscpu | grep -i numa Example using five interleaved CPU cores (2,4,6,8,10): Core 2 - Used for pktgen command line and displaying the run time metrics. C □捐□ below for more details. • **$dst_port** must match the **$app_port** number used in the ADQ configuration on the AF_XDP server (TC filter and ntuple configuration commands in AF_XDP Server Configuration). • Other variables, such as **$packet_size**, can be configured to test the desired traffic pattern.	

Did this document help answer your questions?

To use all queues, the number of traffic streams (ports) should be equal to or a multiple of the number of ADQ queues configured on the AF_XDP server (**$num_queues_tc1** in AF_XDP Server Configuration).

1. **pktgen** configuration command.

```
cd $RTE_SDK
./$pathtopktgen/pktgen -l $cores -n $mem_channels --proc-type auto --log-
level $log_level --socket-mem=$socket_mem --file-prefix $file_prefix -a
$pci_id -- -N -P -T -m [$rx_cores:$tx_cores].$port_num
```

Following are the mandatory and optional command-line options:

Mandatory fields:

-l:	List of cores on which to run.
-n:	Number of memory channels (range is 2-4).
-m:	Matrix for mapping ports to logical cores.
-N:	Enable NUMA support.
-P:	Enable PROMISCUOUS mode on all ports.
--file-prefix:	Prefix for huge page filenames.

Optional fields:

-T:	Enable the color output.
--proc-type:	Type of the process (primary| secondary |auto).
--log-level:	Set specific log level.
--socket-mem:	Memory to allocate on specific sockets (comma separated values).

Example commands:

The following examples configure **pktgen** to use CPU cores specified by **-l $cores**, where the first core is used for the **pktgen** command line, for timers and for displaying the runtime metrics text on the terminal, and the remaining cores are used for the generated UDP packet streams. **$cores** are CPU cores on the same (local) NUMA node as the E810 network adapter. **$rx_cores** and **$tx_cores** are used to map Rx and Tx traffic from Port 0 of the network adapter to the CPU cores specified by **-m**.

NOTE

The PCIe device 12:00.0 that is used in **pktgen** is on NUMA node 0, so **-l <***CPUs***>** are from NUMA node 0 inventory.

The following command can be used to list inventory of available NUMA nodes on the system, and the respective CPU cores:

```
numactl -hardware (or) lscpu | grep -i numa
```

a. Example using five interleaved CPU cores (2,4,6,8,10):
   - Core 2 - Used for pktgen command line and displaying the run time metrics.

     Cores 4,6 - RX traffic.

Did this document help answer your questions?

Cores 8,10 - TX traffic.

```
./pktgen -l 2,4,6,8,10 -n 4 --proc-type auto --log-level 7 --socket-
mem=1024 --file-prefix pktgen1 -a 12:00.0 -- -N -P -T -m [4-6:8-10].0
```

b. Example using nine consecutive CPU cores (1-9):

- Core 1 - Used for pktgen command line and displaying the run time metrics.

  Cores 2-5 - RX traffic.

  Cores 6-9 - TX traffic.

```
./pktgen -l 1-9 -n 4 --socket-mem=256,256 --file-
prefix=pktgen2 -a 12:00.0 -- -N -T -P -m "[2-5:6-9].0"
```

2. Pktgen commands to generate the traffic.

```
Pktgen:/> set $port_num proto $protocol
Pktgen:/> set $port_num size $packet_size
Pktgen:/> set $port_num src mac $src_mac
Pktgen:/> set $port_num dst mac $dst_mac
Pktgen:/> set $port_num src ip $src_ip/mask
Pktgen:/> set $port_num dst ip $dst_ip
pktgen:/> set $port_num dport $dst_port
Pktgen:/> set $port_num sport $src_port
Pktgen:/> enable $port_num range
Pktgen:/> range $port_num proto $protocol
Pktgen:/> range $port_num size $packet_size $min_value $max_value $increment
Pktgen:/> range $port_num src mac $src_mac $min_value $max_value $increment
Pktgen:/> range $port_num dst mac $dst_mac $min_value $max_value $increment
Pktgen:/> range $port_num dst ip $dst_ip $min_value $max_value $increment
Pktgen:/> range $port_num src ip $src_ip $min_value $max_value $increment
Pktgen:/> range $port_num dst port $dst_port $min_value $max_value $increment
Pktgen:/> range $port_num src port $src_port $min_value $max_value $increment
Pktgen:/> start $port_num #To start the traffic
Pktgen:/> stop $port_num #To stop the traffic
Pktgen:/> quit   #To quit the application
```

Example commands:

The following commands configure 64-byte UDP packets from the pktgen host (src) to the AF_XDP server (dst), generating 8 traffic streams across 8 application ports. The destination port numbers (dport and dst port) must match the port numbers used in the ADQ configuration steps on the AF_XDP server in AF_XDP Server Configuration.

Must send UDP traffic to an AF_XDP socket.

```
Pktgen:/> set 0 proto udp
Pktgen:/> set 0 size 64
Pktgen:/> set 0 src mac b4:96:91:9c:f9:90
Pktgen:/> set 0 dst mac 40:a6:b7:19:08:d0
```

Tested with /16 network mask, other range of network masks could also be used (Default is /32 for IPv4). Subnet mask not required on the destination IP address.

```
Pktgen:/> set 0 src ip 61.1.1.50/16
Pktgen:/> set 0 dst ip 61.1.1.100
```


Did this document help answer your questions?

Destination port number must match the starting $app_port number configured on the ADQ AF_XDP server (AF_XDP Server Configuration).

```
pktgen:/> set 0 dport 5002
Pktgen:/> set 0 sport 3002
Pktgen:/> enable 0 range
Pktgen:/> range 0 proto udp
Pktgen:/> range 0 size 64 64 64 1
Pktgen:/> range 0 src mac b4:96:91:9c:f9:90 b4:96:91:9c:f9:90
b4:96:91:9c:f9:90 00:00:00:00:00:01
Pktgen:/> range 0 dst mac 40:a6:b7:19:08:d0 40:a6:b7:19:08:d0
40:a6:b7:19:08:d0 00:00:00:00:00:01
Pktgen:/> range 0 dst ip 61.1.1.100 61.1.1.100 61.1.1.100 0.0.0.1
Pktgen:/> range 0 src ip 61.1.1.50 61.1.1.50 61.1.1.50 0.0.0.1
```

Number of traffic streams (ports) should be equal to or a multiple of the number of ADQ queues configured on the AF_XDP server (**$num_queues_tc1** in AF_XDP Server Configuration). **$dst_port** numbers must match the port numbers configured on the ADQ server.

```
Pktgen:/> range 0 dst port 5002 5002 5009 1 #Ports used to add TC filters in
ADQ server
 Pktgen:/> range 0 src port 3002 3002 3009 1 #8 source ports
 Pktgen:/> start 0
 Pktgen:/> stop 0
```

3. Verify ADQ traffic is on the correct queues.

   After traffic is started in the client machine, on the AF_XDP server start the XDPSOCK application and watch **ethtool** statistics to check that only the ADQ queues are being used. The pkt_busy_poll counter should have significantly higher traffic. If instead, pkt_not_busy_poll show significantly higher traffic, or if traffic is not confined to ADQ queues, re-check the configuration steps carefully and/or make sure XDPSOCK is running on the AF_XDP server.

```
watch -d -n 0.5 "ethtool -S $iface | grep busy | column"
```

   See Verify ADQ Application Traffic and Independent Pollers (If Applicable) for example ethtool watch output.

NOTE

The ADQ Setup script clears the existing configuration before proceeding with the new ADQ configuration. To clear manually, follow the steps in Clear the ADQ Configuration.

Did this document help answer your questions?

14.0 Testing ADQ with Apache Traffic Server (ATS)

ATS is a fast, scalable, and extensible HTTP/1.1 and HTTP/2 compliant caching proxy server. **Vegeta** is "a versatile HTTP load testing tool built out of a need to drill HTTP services with a constant request rate"[1]. It helps measure the performance of web servers, particularly throughput and latency (percentile). In this section, ATS is used as an example of a real-world application with ADQ.

NOTE

This document is not intended to be a complete ATS/Vegeta configuration guide. Refer to ATS/Vegeta documentation for more details on usage.

https://docs.trafficserver.apache.org/en/latest/

https://docs.trafficserver.apache.org/en/latest/getting-started/index.en.html

Throughout this section, the steps that are required for ADQ only are highlighted with an **[ADQ only]** flag. If benchmarking a baseline configuration, follow all other steps and make sure **[ADQ only]** steps are omitted/reverted.

14.1 ATS Server Setup

The following variables are used in the examples in this section:

$iface	The PF interface in use.
$num_queues_tc0	The number of queues for default traffic.
$num_queues_tc1	The number of queues for application traffic class.
$ip_addr	The IP Address of the interface under test.
$app_port	The TCP port of the **ATS** application being run on the SUT(for example, 80, 8080, 8888; default port for http traffic is 80).
$cgroup_name	The name for the application group.
$iface_bdf	The network interface BDF notation (Bus:Device.Function) used by devlink.
$cpu_sockets	The list of sockets on SUT (0, 1 for 2-socket system).
$tc1_qps_per_poller	The number of queues per one poller thread.
$tc1_timeout_value	The value of timeout in poller threads (nonzero integer value in jiffies, default value 10000).
$pathtoicepackage	The path to the ice driver package.
${pathtotc}	The path to the TC command.
$core_number	The specific core number.

continued...

1 From Vegeta readme on github: https://github.com/tsenart/vegeta#readme

Did this document help answer your questions?

$pid	The process identification number.
$prio	The priority of flow-based traffic control group.
$core_range_0-$core_range_1	The range of cpu cores assigned to cgroup.

NOTE

Use of independent poller feature in Apache Traffic Server is currently supported on PF only (not supported on VF).

1. Perform general system OS install and setup.
   a. Complete the ADQ install and setup in ADQ System Under Test (SUT) Installation.

      NOTE

      SUT Linux kernel later than v5.12 is strongly recommended for optimal performance, and ICE driver version 1.9.x is required for Independent Pollers feature of ADQ 2.0.

      NOTE

      For best ATS performance with ADQ, the following default *ice* driver settings should be used on the ATS Server:

      - Turn off packet optimizer:

        ```
        ethtool --set-priv-flags $iface channel-pkt-inspect-optimize off
        ```

      - Enable hardware tc offload:

        ```
        ethtool -K $iface hw-tc-offload on
        ```

        And for the interface that is being used, set:

        ```
        echo 1 > /sys/class/net/$iface/threaded
        ```

   b. Enable threaded mode napi poll.

      Two changes to the system tuning are required.

      i. Enable latency-performance tuned profile.

         ```
         tuned-adm profile latency-performance
         ```

         NOTE

         The *tuned-adm* daemon is not installed by default in RHEL9.0 systems. Install it with the command `yum install tuned`.

Did this document help answer your questions?

Check that the settings are applied correctly:

```
cat /etc/tuned/active_profile
```

Output: `latency-performance`

```
cat /etc/tuned/profile_mode
```

Output: `manual`

ii. Set the CPU scaling governor to performance mode

```
x86_energy_perf_policy performance
```

Check that the settings are applied correctly:

```
cat /etc/tuned/active_profile
```

Output: `latency-performance`

```
cat /etc/tuned/profile_mode
```

Output: `manual`

```
cat /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor
```

Output: `performance`

NOTE

For each CPU thread on server.

For best performance with Apache Traffic Server, set `net.core.netdev_max_backlog` and `net.ipv4.tcp_max_syn_backlog` as follows:

```
sysctl -w net.core.netdev_max_backlog=250000
sysctl -w net.ipv4.tcp_max_syn_backlog=250000
```

Also set:

```
sysctl -w net.core.busy_poll=0
```

NOTE

Many settings in General System Tuning and those listed above do not persist between reboots and might need to be reapplied.

Did this document help answer your questions?

2. Perform ATS build.
   a. Download the latest ATS version from one of the following locations:
      - Download the latest ATS version from GitHub:

        Example:

        ```
        git clone https://github.com/apache/trafficserver
        ```

      Or

      - Download the latest stable ATS release from apache.org:

        Example:

        ```
        wget https://dlcdn.apache.org/trafficserver/trafficserver-9.1.2.tar.bz2
        ```

        Untar the package

        ```
        tar xf trafficserver-9.1.2.tar.bz2
        ```

   b. Compile and install ATS.

      ```
      cd trafficserver-${VER}
      ```

      NOTE

      $VER is the trafficserver version that was downloaded.

      ```
      autoreconf-if
      ./configure
      make
      make install
      ```

   c. Set the number of ATS threads to be equal to **$num_queues_tc1** using `records.config`.

      The records.config file, by default located in `/usr/local/etc/trafficserver/`, is a list of configurable variables used by the Traffic Server software.

      Field:

      ```
      CONFIG proxy.config.exec_thread.autoconfig.scale FLOAT 1
      ```

      Set the multiplier according to total number of cores vs number of ADQ queues.

      Example:

      If the total number of cores is 144 and the number of queues (**$num_queues_tc1**) is 126, the multiplier would be 0.875 and the config line would be:

      ```
      CONFIG proxy.config.exec_thread.autoconfig.scale FLOAT 0.875
      ```


Did this document help answer your questions?

3. **[ADQ only]** Create Traffic Classes (TCs) on the interface under test. Note that TC1 should have the same number of queues as the number of threads that will be started by the ATS application.

```
${pathtotc}/tc qdisc add dev $iface root mqprio num_tc 2 map 0 1 queues
$num_queues_tc0@0 num_queues_tc1@$num_queues_tc0 hw 1 mode channel

${pathtotc}/tc qdisc add dev $iface clsact
```

NOTE

Due to timing issues, applying TC filters immediately after the `tc qdisc add` command might result in the filters not being offloaded in hardware. An error in dmesg is logged if the filter fails to add properly. It is recommended to wait five seconds after `tc qdisc add` before adding TC filters.

```
sleep 5
```

4. **[ADQ only]** Create one TC filter on the interface under test.

```
 ${pathtotc}/tc filter add dev $iface protocol ip ingress prio 1 flower
dst_ip $ip_addr/32 ip_proto tcp dst_port $app_port skip_sw hw_tc 1
```

NOTE

The /32 in `dst_ip` **`$ip_addr`**`/32` is not the subnet of the network being used, but the subnet of the filter you are creating. In other words, /32 indicates a single IP Address being filtered. It is recommended to use /32 when creating filters to limit the addresses being filtered.

NOTE

ATS by default sets HTTP on port 8080, HTTPS on port 8443. It might be changed in

```
 vi /usr/local/etc/trafficserver/records.config
```

5. **[ADQ only]** Confirm TC configuration:
   a. Check TC filter.

```
${pathtotc}/tc filter show dev $iface ingress
```

   b. Check that TCs were created correctly.

```
${pathtotc}/tc qdisc show dev $iface
```

6. For best performance with Apache Traffic Server set the interrupt moderation rate to this static value for Tx and Rx.

```
ethtool --coalesce ${iface} adaptive-rx off rx-usecs 100
ethtool --coalesce ${iface} adaptive-tx off tx-usecs 100
```


Did this document help answer your questions?

7. Configure Independent pollers.

   ice version 1.9.x (and later):

```
iface_bdf=$(ethtool -i ${iface} | grep bus-info | awk '{print $2}'
devlink dev param set pci/${iface_bdf} name tc1_qps_per_poller value
$tc1_qps_per_poller cmode runtime
devlink dev param set pci/${iface_bdf} name tc1_poller_timeout value
$tc1_timeout_value cmode runtime
```

   NOTE

   Kernel with devlink param support is required for ice-1.8.x and later. See Install OS and Update Kernel (If Needed) to determine OS and kernel requirements.

   NOTE

   Valid devlink param flags include `tc1_qps_per_poller` and `tc1_poller_timeout` through `tc15_qps_per_poller` and `tc15_poller_timeout`, to configure pollers on up to 16 TCs (max #TCs).

8. Configure Flow Director.

```
devlink dev param set pci/${iface_bdf} name tc1_inline_fd value true cmode
runtime
```

9. Run the set_irq_affinity script for the interface under test.

```
${pathtoicepackage}/scripts/set_irq_affinity -X7 all $iface
```

10. Configure symmetric queues on the interface using the script included in the scripts folder of the ice package.

```
${pathtoicepackage}/scripts/set_xps_rxqs $iface
```

11. **[ADQ only]**

    NOTE

    The system under test can be configured with a single NIC or multiple NICs; choose the option based on the configuration.

    a. **Single NIC configuration**

    Configure specific cores as being responsible for dedicated poller threads to optimize CPU usage and improve SUT stability. In case of 2-socket systems and one NIC use cores on the CPU located on the same NUMA node as the NIC.

```
#!/bin/bash
CHK () {
        "$@"
        if [ $? -ne 0 ]; then
                echo "Error with ${1}, stopping now!" >&2
                exit 1
        fi
}
```


Did this document help answer your questions?

```
pin_napi_threads () {
        num_threads=$(ps -aef |grep "\[napi\/" | wc -l)
        start_pid=$(ps -ae |grep napi |head -n 1 |cut -d ? -f 1)

        for (( i = 0; i < $num_threads; i++ ))
        do
                printf -v pid "%d" $((start_pid+i))
                printf -v cpu "%d" $((2*i+1))
                CHK taskset -p -c "${core_number}" $pid
        done
}
```

pin_napi_threads

NOTE

There is possibility to assign more than one cpu core for poller threads, it is required to be equal to the no. of configured poller threads. Each core number has to be separated by comma.

Example for 8 poller threads and therefore 8 cores:

```
CHK taskset -p -c "36,37,38,39,40,41,42,43" $pid
```

NOTE

In case of small object being transferred more poller threads usually yields better performance (example: 1 poller thread per 8-16 queues for 10kB objects), for larger objects (>1MB) a smaller number of poller threads (and therefore dedicated cores) can be used.

b. **Multiple NIC configuration**

If your system under test has multiple NIC, ensure, that the system is balanced (it has the same number of NICs per NUMA node). Queues should be divided to equal no. per every NIC, and poller threads should be placed on each NUMA node in equal number and assigned to their respective NICs residing on the same NUMA node as the poller threads.

Example:

```
num_threads=$(ps -aef |grep "\[napi\/$iface1" | wc -l)
 start_pid=$(ps -ae |grep napi\/$iface1 |head -n 1 |cut -d ? -f 1)

 for (( i = 0; i < $num_threads; i++ ))
 do
        printf -v pid "%d" $((start_pid+i))
        printf -v cpu "%d" $((2*i+1))
        CHK taskset -p -c "${core_number}" $pid
 done

 num_threads=$(ps -aef |grep "\[napi\/$iface2" | wc -l)
 start_pid=$(ps -ae |grep napi\/$iface2 |head -n 1 |cut -d ? -f 1)

 for (( i = 0; i < $num_threads; i++ ))
 do
        printf -v pid "%d" $((start_pid+i))
        printf -v cpu "%d" $((2*i+1))
        CHK taskset -p -c "${core_number}" $pid
 done
```


Did this document help answer your questions?

12. **[ADQ only]** In this example, we are using cgroups for pinning Apache Traffic Server threads and poller threads to specific cores. Other methods can be used to achieve the same effect.

    Create cgroup and map to interface under test. Set the priority for processes belonging to the cgroup. The **$prio** value should map to the position of the targeted TC in Step 3 to create traffic classes (TCs). See Create TCs for reference on TC priority mapping

```
cgcreate -g cpuset,memory,net_prio:${cgroup_name}
cgset -r net_prio.ifpriomap="$iface $prio" ${cgroup_name}
cgset -r cpuset.mems="${cpu_sockets}" ${cgroup_name}
cgset -r cpuset.cpus="${core_range_0}-${core_range_1}" ${cgroup_name}
```

13. Start ATS either in the cgroup for ADQ or without the ADQ if testing baseline performance.

    a. **[ADQ Only]** Start ATS in the cgroup with the same number of threads that were configured for TC1.

       Example:

```
/usr/local/bin/trafficserver start
cgclassify -g cpuset,net_prio:ats $(ps -e | grep TS_MAIN | awk \ '{ print
$1 }')
cgclassify -g cpuset,net_prio:ats $(ps -e | grep ET_NET | awk \ '{ print
$1 }')
```

    b. **[Baseline]** Start ATS.

       Example:

```
/usr/local/bin/trafficserver start
```

14. **[ADQ only]** Verify cgroup configuration:

    a. Show that the interface is mapped to the right interface.

```
cat /sys/fs/cgroup/net_prio/${cgroup_name}/net_prio.ifpriomap
```

    b. Show the Process IDs being run in the cgroup and match them to the application Process IDs.

```
cat /sys/fs/cgroup/net_prio/${cgroup_name}/tasks
```

15. **[ADQ only]** Troubleshooting: While test is running, verify that ADQ traffic is on the correct queues.

    While ADQ application traffic is running, watch ethtool statistics to check that only the ADQ queues are being used (that is, have significant traffic) with busy poll (`pkt_busy_poll`) for ADQ traffic. If non busy poll (`pkt_not_busy_poll`) has significant counts and/or if traffic is not confined to ADQ queues, recheck the configuration steps carefully.

```
watch -d -n 0.5 "ethtool -S $iface | grep busy | column"
```

    See Configure Intel® Ethernet Flow Director Settings for example watch output.

Did this document help answer your questions?

NOTE

If the benefits of ADQ enabled versus disabled are not as dramatic as expected even though the queues appear to be well aligned, it is possible that the performance is limited by the processing power of the client systems. If client CPU shows greater than an average of 80% CPU utilization on the CPU cores in use, it is probable that the client is becoming overloaded.

To achieve maximum performance benefits, try increasing the number of or the processing power of the client systems.

After the test finishes, remove the cgroup.

```
jobs -p | xargs kill &> /dev/null cgdelete -g net_prio: ${cgroup_name}
```

After the test finishes, remove the ADQ configuration following steps in Clear the ADQ Configuration.

14.2 Vegeta Clients Configuration

NOTE

ADQ is not required for the clients. The following example sets up clients without ADQ for use in benchmarking the ADQ or non-ADQ SUT performance using the Vegeta benchmark tool.

The following variables are used in the examples in this section:

$ipaddr	From the server, the IP Address of the application server's interface under test.
$app_port	From the server, the TCP port of the ATS application being run on the server.
$endpoint	The object or page that is tested on ATS.
$PATH	The ordered list of paths that Linux will search for executables when running a command
$pathtovegeta	The directory where Vegeta files are placed

1. Perform Vegeta build:

   Example:

   a. Download Vegeta.

   ```
   wget https://github.com/tsenart/vegeta/releases/download/v12.8.4/
   vegeta_12.8.4_linux_amd64.tar.gz
   ```

   b. Untar Vegeta files.

   ```
   tar xf vegeta_12.8.4_linux_amd64.tar.gz
   ```

   c. Add Vegeta to environment variables

   ```
   export PATH="${PATH}:${pathtovegeta}"
   ```


Did this document help answer your questions?

2. Performance Vegeta benchmarking.

   Vegeta is an open-source product used for benchmark HTTP/HTTPS. Tool has many different parameters to modify. For example, it allows you to specify the frequency of requests per second and connections that can be tested on the SUT. Additionally, it can be easily automated using scripts.

   Measurements given by Vegeta are by default *time to last byte*.

   Vegeta allows to test out many objects test case using targets parameter.

   NOTE

   Average latency is a measure of how fast SUT can respond

   NOTE

   $ipaddr and **$app_port** are the same variables used in SUT configuration

   Example of Vegeta usage:

```
echo GEThttp://${ipaddr}:${app_port}/${endpoint} | ./vegeta attack -rate
100/s \
-insecure -duration 300s -timeout 10s -output /root/results.bin -connections
100 -keepalive \
-max-body=0

sleep 300s

./vegeta report -type json -output /root/results.json /root/results.bin
```


Did this document help answer your questions?

15.0 Testing ADQ with VirtIO

VirtIO is an abstraction layer over a host's devices for virtual machines. It is one of I/O (block, NIC ,etc) virtualization techniques that are used to provide guests with high performance network and many of the performance benefits.

VirtIO divided into three parts:

1. Frontend virtio drivers: Implemented in the guest operating system as a device driver.
2. Backend virtio drivers: Implemented in the hypervisor, and accepts the I/O requests from front-end drivers and perform corresponding I/O operations via a PF.
3. Transport: Communication between front-end and back-end drivers is done through a queue called virtqueue.

Virtio-net is a virtual ethernet card running in the guest kernel space.

The main building blocks in virtio are KVM, QEMU and libvirt.

1. KVM used QEMU for I/O hardware emulation. QEMU is a user-space emulator that can emulate a variety of guest processors on host processors.
2. KVM is managed via the libvirt API and tools. libvirt tools include virsh,virt-install and virt-clone.
3. KVM provides the ability for a portion of the physical CPU to be directly mapped to the virtual CPU.

15.1 Installation and Configuration - Both Systems

The following variable is used in the examples in this section:

$iface	The interface in use (PF).
$bridge_iface	Bridge interface created on PF (ex: br0).
$bridge_ipadress	IP address of the bridge interface.
$num_queues_tc0	The number of queues for default traffic class (usually 2 or more but needs to be a power of 2 or it is not accepted).
$num_queues_tc1	The number of queues for application traffic class 1 (corresponds to the user selected number of application threads for application 1). Note power of 2 restrictions below.
$dst_mac	MAC address of the VM's virtio interface.
$ipaddress	VM IP address (ex: virtio-net interface IP configured in VM).
$queue_id	The specific rx queue number in hexadecimal (ex: 0x3 or 0xa or 0xb etc)
$Tx_queue_id	The specific Tx queue number (ex: 3 or 4 or 5 etc)
$qps_per_poller	The number of queue pairs per independent poller for a given TC (max value is #queues in the TC)

continued...

Did this document help answer your questions?

$iface_bdf	The network interface BDF notation (Bus:Device.Function) used by devlink (PF)
$poller_timeout	The timeout value for the independent pollers for a given TC (nonzero integer value in jiffies, default value 10000)
$pathtoicepackage	The path to the ice driver package

1. Enable Virtualization in the BIOS. The following settings must be enabled in the BIOS.

```
[BIOS::Advanced::Processor Configuration]  Intel(R) Virtualization
Technology=Enabled
[BIOS::Advanced::Integrated IO Configuration]  Intel(R) VT for Directed I/
O=Enabled
[BIOS::Advanced:: Processor Configuration ]  Intel(R) Hyper-Threading
Tech=Enabled
```

2. **[Optional]** Enable the following BIOS settings on both servers for better performance with VirtIO.

   Processor Settings ➢ Logical Processor ➢ Enabled

   System Profile Settings ➢ System Profile ➢ Performance

3. Verify the CPU Virtualization extensions are available.

```
grep -E 'svm|vmx' /proc/cpuinfo
```

   Where:

   `vmx`: Entry indicating an Intel processor with the Intel VT extensions.

   `svm`: Entry indicating an AMD processor with the AMD-V extensions.

4. Verify *kvm* and *kvm_intel* modules are loaded in the kernel, and if not load manually.

```
lsmod | grep -i kvm
```

5. Install pre-requisites.

```
yum install -y libvirt
yum install -y bridge-utils
```

6. Enable and start the libvirtd service

```
systemctl enable libvirtd; systemctl start libvirtd
```

NOTE

These settings are not persistent across system reboots.

Did this document help answer your questions?

7. Configure network settings.

NOTE

Remove the virtual bridge network interface *virbr0*, which is created by default.

```
ip link set virbr0 down
ip link delete dev virbr0
```

NOTE

You can use either the Linux bridge or the OVS bridge.

a. **Network Settings Using Linux Bridge:**

i. Create the Linux bridge.

NOTE

There are many methods to create bridge network using virsh, nmcli, or editing network scrips etc. Here, we used IP commands to configure the bridge.

NOTE

If the PF interface has an IP address, remove it.

```
ip link add name $bridge_iface type bridge
 ip link set $iface master $bridge_iface
 ip address add dev $bridge_iface $bridge_ipadress
 ip link set up $iface
 ip link set up $bridge_iface
```

ii. Check if the bridge interface is active.

```
virsh net-list --all
```

If **$bridge_iface** is not listed, the following commands create the bridge and make it active.

```
a. cd /tmp/
b.vi bridge.xml
    <network>
    <name>$bridge_iface </name>
    <forward mode="bridge"/>
    <bridge name="$bridge_iface" />
    </network>
c.  virsh net-define /tmp/bridge.xml
d. virsh net-start $bridge_iface
e. virsh net-autostart $bridge_iface
f. virsh net-list -all
```


Did this document help answer your questions?

b. **Network Settings Using Open Vswitch (OVS):**

i. Download and untar OVS source package.

```
mkdir /opt/ovs
wget -O /opt/ovs/ovs.tar.gz https://github.com/openvswitch/ovs/archive/
v2.12.0.tar.gz
cd /opt/ovs ; tar -xf ovs.tar.gz
cd ovs-2.12.0
```

ii. Start bootstraping OVS.

```
./boot.sh
```

iii. Configure Open Vswitch default database directory using options as shown here.

```
./configure --prefix=/usr --localstatedir=/usr/local/var --
sysconfdir=/etc
```

NOTE

Open vSwitch installed with .rpm (e.g., through `yum install` or `rpm -ivh`) and .deb (e.g., through `apt-get install` or `dpkg -i`) packages and use these configure options.

iv. Install Open Vswitch.

```
make
  make install
```

v. Configure a database for ovsdb-server to use.

```
mkdir -p /usr/local/var/run/openvswitch
mkdir -p /usr/local/etc/openvswitch
/opt/ovs/ovs-2.12.0/ovsdb/ovsdb-tool create /etc/openvswitch/conf.db
\
  /usr/local/share/openvswitch/vswitch.ovsschema
```

vi. Configure ovsdb-server to use database created above, to listen on a Unix domain socket, to connect to any managers specified in the database itself, and to use the SSL configuration in the database.

```
ovsdb-server --remote=punix:/usr/local/var/run/openvswitch/db.sock \
  --remote=db:Open_vSwitch,Open_vSwitch,manager_options \
  --private-key=db:Open_vSwitch,SSL,private_key \
  --pidfile -detach
```

vii. Initialize the database using ovs-vsctl and start the Open Vswitch daemon.

```
/usr/bin/ovs-vsctl --no-wait init
/usr/bin/ovs-vsctl --no-wait set Open_vSwitch . other_config:hw-
offload=true
/usr/bin/ovs-vsctl --no-wait set Open_vSwitch . other_config:tc-
policy=none
ovs-vswitchd --pidfile --detach --mlockall --log-file=/tmp/ovs-
vswitchd.log
```


Did this document help answer your questions?

viii. Set up bridge network using ovs-ctl.

```
ovs-vsctl add-br $bridge_iface
ovs-vsctl add-port $bridge_iface $iface
ip addr add $bridge_ipadress dev $bridge_iface
ip link set $bridge_iface up
```

8. Configure VM using GUI or CLI.

NOTE

VMs can be configured using either the *virt-manager* (GUI) or the Virsh commands (CLI).

VM-level System Tunings

a. Enable virtual-guest tuned profile.

This profile decreases virtual memory swappiness values and increases disk readahead values.

```
tuned-adm profile virtual-guest
```

b. Set busy_read setting for virtual guests.

Recommended 100 µs for a starting point to stay in poll mode.

```
sysctl -w net.core.busy_read=100
```

c. Disable firewalls.

```
service firewalld stop; systemctl mask firewalld
```

d. Disable Security-Enhanced Linux (SELinux) (requires reboot to take affect).

Change `SELINUX=enforcing` to `SELINUX=disabled` in */etc/selinux/config*

e. Stop the irqbalance service.

```
systemctl stop irqbalance
```

NOTE

When assigning the network to the VM, choose the network source as the bridge interface created and virtio as the device model.

NOTE

Check that the host bridge network and the VM's virtio-net interface are on the same subnet and ping each other to confirm connectivity between the host and the VM.

Did this document help answer your questions?

9. Perform general system setup at host level.

   Complete the ADQ installation and setup in ADQ System Under Test (SUT) Installation and in General System Tuning on both the server and client systems

   NOTE

   Many settings in General System Tuning and Adapter Preparation do not persist between reboots and might have to be reapplied.

   NOTE

   Recheck the bridge settings when unloading/loading the ICE driver.

   NOTE

   Virtio-net-based VMs do not require VF HW configuration, and ADQ acceleration is possible with kthread-based independent polling, which is supported starting with kernel 5.12 or higher. The global settings listed below should be disabled in Independent poller-based ADQ.

```
sysctl -w net.core.busy_poll=0
sysctl -w net.core.busy_read=0
ethtool --set-priv-flags $iface channel-pkt-inspect-optimize off
```

10. Create Traffic classes on the PF interface (the interface which is participated in the bridge configuration).

```
${pathtotc}/tc qdisc add dev $iface root mqprio num_tc 2 map 0 1 queues
$num_queues_tc0@0 $num_queues_tc1@$num_queues_tc0 hw 1 mode channel
${pathtotc}/tc qdisc add dev $iface clsact
```

NOTE

Due to timing issues, applying TC filters immediately after the `tc qdisc add` command might result in the filters not being offloaded in hardware. An error in dmesg is logged if the filter fails to add properly. It is recommended to wait five seconds after tc qdisc add before adding TC filters.

```
Sleep 5
```

11. Create TC filters (Both ingress and egress) on the PF interface.

    - **Ingress Filters (one Filter Per VM):**

```
${pathtotc}/tc filter add dev $iface protocol ip ingress prio 1
flower dst_mac $dst_mac dst_ip $ipaddress/32 skip_sw classid
ffff:$queue_id
```

Whereas:

`ffff`:Qdisc ID(Fixed value, can get using the command : tc qdisc show dev $iface clsact)

Did this document help answer your questions?

- **Egress Filters (one Filter per VM):**

```
${pathtotc}/ tc filter add dev $iface egress prio 1 protocol
ip flower src_ip $ipaddress action skbedit priority 0x1 pipe
action skbedit queue_mapping $Tx_queue_id
```

Whereas:

`src_IP`: VM IP address

`queue_mapping`: Aligns TX with RX queue ID

NOTE

Cgroup is not supported in virtio. Instead, per-TC skbedit socket priority option should be used for egress traffic.

12. Enabled Number of pollers, poller time out and threaded NAPI

```
iface_bdf=$(ethtool -i ${iface} | grep bus-info | awk '{print $2}')
devlink dev param set pci/$iface_bdf name tc1_qps_per_poller value
$qps_per_poller cmode runtime
devlink dev param set pci/$iface_bdf name tc1_poller_timeout value
$poller_timeout cmode runtime
echo 1 /sys/class/net/$iface/threaded
```

13. Set irq_affinity.

```
${pathtoicepackage} /set_irq_affinity -X all $iface
${pathtoicepackage} /scripts/set_xps_rxqs $iface
```


Did this document help answer your questions?

15.2 Run the Traffic

NOTE

Any application can be used to generate traffic; in this case, Netperf is used.

1. Execute the Netperf TCP RR test between the servers' VMs. To configure and run the Netperf , follow the steps in the subsections of Testing ADQ with Netperf.
2. Watch the busy_poll counters on the host machine.

```
watch -d -n 0.5 "ethtool -S $iface | grep busy | column"
```

3. Inspect the pollers on the host machine.

   When ADQ application traffic is running with one or more independent pollers configured, system monitoring tools such as top/htop will show kernel threads running (with kernel version 5.12+). Refer to Verify ADQ Application Traffic and Independent Pollers (If Applicable) for more information.

   NOTE

   Application traffic can be pinned either to the same CPU cores, or different cores, than the independent poller threads. There may be slight performance improvements using different CPU cores for the kernel pollers. However, if reducing the total number of CPU cores used by ADQ is important, overlapping application threads and independent poller threads is an option.

Known Issues

Egress traffic does not go from the last queue of the ADQ TC set; this is a kernel-specific issue that has been addressed in the 5.19* kernel.

- Linux 5.18 or lower: Skbedit priority is in hex 0 based and queue_mapping is in decimal 1 based.
- Linux 5.19 or later: Skbedit priority is in hex 0 based and queue_mapping is in decimal 0 based.

Did this document help answer your questions?

Appendix A ADQ Configuration Reference (Manual)

A.1 ADQ Configuration on SUT

This section is meant to be a generic walk-through of configuring ADQ on a system. If testing one of the enabled applications, skip this section and instead use the instructions in the relevant testing application appendices:

- Testing ADQ with Netperf
- Testing ADQ with Redis
- Testing ADQ with Memcached/rpc-perf
- Testing ADQ with NGINX
- Testing ADQ with AF_XDP
- Testing ADQ with Apache Traffic Server (ATS)
- Testing ADQ with VirtIO

There are two methods to configure ADQ on the SUT:

- Method-1: Using Manual ADQ Configuration (Adapter Preparation through Clear the ADQ Configuration)
- Method-2: Using ADQ Setup Script (ADQ Setup Using ADQ Setup Script)

Both methods require the completion of ADQ System Under Test (SUT) Installation and General System Tuning.

NOTE

The settings listed in this section do not persist between reboots. The ADQ Setup Script provides an option to make the configuration persistent across reboots using a configuration file, see ADQ Setup Using ADQ Setup Script for more details.

A.1.1 Adapter Preparation

The following variable is used in the examples in this section:

$iface	The interface in use (PF).

NOTE

$iface in this section refers to the Physical Function (PF) interface. If using VFs, the commands in this section must be applied to the PF interface first, before spawning VF interfaces (as explained in ADQ Configuration on SR-IOV Virtual Functions).

Did this document help answer your questions?

1. Reload the *ice* driver to remove any previous TC configuration.

```
rmmod ice
sleep 2
modprobe ice
```

2. Make sure the interface is up with an IP Address assigned (either through configuration files or ifconfig/ip commands).
3. Enable hardware TC offload on the interface.

```
ethtool -K $iface hw-tc-offload on
```

NOTE

There is a known issue that allows ADQ configuration to be applied on a VF interface when the `hw-tc-offload` flag is disabled (set to off) on the VF interface. For details, see Fixed and Known Issues - Read First.

4. Disable LLDP on the interface. (This is the default setting and the following command should not be necessary.)

```
ethtool --set-priv-flags $iface fw-lldp-agent off
```

5. Enable `channel-pkt-inspect-optimize` on the interface.

```
ethtool --set-priv-flags $iface channel-pkt-inspect-optimize on
```

NOTE

If the ADQ setup includes independent pollers, this flag must be set to `off`.

6. **[Optional]** Enable busy poll private flag(s) for large I/O block size.

   The *ice* driver version 1.1.0 or later includes two private flags that optimize performance of Rx packet processing during ADQ configuration with `busy_poll`. These flags should be enabled for applications using large I/O block size. Workloads that benefit from having these two flags turned *on* include AF_XDP and NVMe/TCP using large block I/O.

   The first flag is recommended for applications using 64K (or greater) I/O block size:

```
ethtool --set-priv-flags $iface channel-pkt-clean-bp-stop on
```

   The second flag is recommended for applications using 128K (or greater) I/O block size:

```
ethtool --set-priv-flags $iface channel-pkt-clean-bp-stop-cfg on
```

NOTE

Both flags must be set to on for the second flag to be enabled.

Did this document help answer your questions?

NOTE

If the ADQ setup includes independent pollers, both bp-stop flags must be set to off.

7. Verify settings.

```
ethtool -k $iface | grep "hw-tc"
ethtool --show-priv-flags $iface
```

Example output (*ice-1.8.x*):

```
hw-tc-offload: on
Private flags for ens8f0:
link-down-on-close           : off
fw-lldp-agent                : off
channel-inline-fd-mark       : off
channel-pkt-inspect-optimize : on
channel-pkt-clean-bp-stop    : off
channel-pkt-clean-bp-stop-cfg: off
vf-true-promisc-support      : off
mdd-auto-reset-vf            : off
vf-vlan-pruning              : off
legacy-rx                    : off
cgu_fast_lock                : off
dpll_monitor                 : off
extts_filter                 : off
```

Example:

For convenience, the commands in this section are condensed below.

NOTE

Most workloads do not require the activation of two bp-stop flags, which are only required for larger block I/O in ADQ1.0.

NOTE

If the ADQ setup includes independent pollers (ADQ2.0), two bp-stop flags must be set to off.

```
#Variables
iface= #The interfact in use
#Settings
rmmod ice
sleep 2
modprobe ice
ethtool -K $iface hw-tc-offload on
ethtool --set-priv-flags $iface fw-lldp-agent off
ethtool --set-priv-flags $iface channel-pkt-inspect-optimize on
ethtool --set-priv-flags $iface channel-pkt-clean-bp-stop off
ethtool --set-priv-flags $iface channel-pkt-clean-bp-stop-cfg off
ethtool -k $iface | grep "hw-tc"
ethtool --show-priv-flags $iface
```


Did this document help answer your questions?

NOTE

If the interface is not set to come back up automatically, it might be necessary to set the interface up and reset the IP Address.

A.1.2 Configure ADQ Traffic Class (TC) on SUT

NOTE

Any **ethtool** coalesce or IRQ affinity settings have to be re-applied after all TC and TC filter creation.

The following variables are used in the examples in Create TCs through Confirm TC Configuration:

$iface	The interface in use (PF or VF).
$ipaddr	The IP Address of the interface. For GTP protocol, it is inner destination IP. For VXLAN, it is VXLAN interface IP.
$macaddr	The MAC Address of the interface.
$app_port	The listening port of the application (usually user-selected in the application configuration). For GTP protocol, it is inner destination L4 port.
$num_queues_tc0	The number of queues for default traffic class (usually 2 or more, but needs to be a power of 2 or it is not accepted).
$num_queues_tc1	The number of queues for application traffic class 1 (corresponds to the user-selected number of application threads for application 1). Note power of 2 restrictions below.
$num_queues_tc2	The number of queues for application traffic class 2 (corresponds to the user-selected number of application threads for application 2). Note power of 2 restrictions below.
$tunnel_id	GTP Tunnel key ID. For VXLAN: Tunnel ID
$gtp_port	GTP port (ex: GTP-U user data tunneling port 2152).
$ipv6addr	The IPv6 address of the interface.
$queue_id	The specific rx queue number in hexadecimal (ex: 0xa,0xb etc).

NOTE

If using VFs, **$iface** in this section refers to the Virtual Function (VF) interface

Did this document help answer your questions?

A.1.2.1 Create TCs

The following example creates three TCs with a small number of queues for TC0 (default TC) and the remaining number of logical cores divided between TC1 and TC2 (application TC) queues. In general, two queues for TC0 and the remaining number of cores for the application TCs are sufficient.

For best performance, the number of application worker threads/processes should match with the number of queues configured for a given application-specific TC.

Traffic originating from an application with priority set to 1 is mapped to TC1, with priority 2 mapped to TC2, etc. All default traffic is mapped to TC0.

NOTE

The allocation of bandwidth between TCs is not guaranteed to be proportional to the number of queues assigned. By default, the bandwidth available is roughly evenly split among all TCs. If a certain minimum bandwidth is required for a particular TC, it is recommended to use rate limiting to set either maximum or minimum bandwidth limits with the TC shaper bw_rlimit parameter.

Due to resource allocation rules, the following rules apply to the number of TC queues:

- **For TC0:** The number of queues for the default traffic class (`$num_queues_tc0`) must always be a power of two.
- **For TC1 or higher:**
  - If all TCs are powers of two, they can be any number.
  - If the number of queues for any TC (for TC1 or above) is not a power of two, any subsequent TC must be either exactly that number or a power of two that is less than that number.
- *ice-1.8.x* and earlier: **The total number of queues from all TCs** cannot be greater than the max number of logical cores in the system.
- *ice-1.9.x* and later: **The total number of queues from all TCs** is increased to a max number of 256 (the number of MSI-X interrupts per PF).
- **The total number of TCs allowed 16.**

Example:

If num_queues_TC0=2, num_queues_TC1=16, and num_queues_TC2=7, then num_queues_TC3+ must be either exactly 7 or some power of 2 less than 7 (4,2,1).

Example:

```
${pathtotc}/tc qdisc add dev $iface root mqprio num_tc 3 map 0 1 2 queues \
$num_queues_tc0@0 $num_queues_tc1@$num_queues_tc0 $num_queues_tc2@$
((num_queues_tc0 \
+ num_queues_tc1)) hw 1 mode channel
```


Did this document help answer your questions?

For example, if there are 96 logical cores in a system, one possible configuration is TC0 with 2 queues, TC1 with 32 queues and TC2 with 62 queues.

```
${pathtotc}/tc qdisc add dev $iface root mqprio num_tc 3 map 0 1 2 queues \
2@0 32@2 62@34 hw 1 mode channel
```

With *ice* driver version 1.9.7 and later, large TCs with up to 256 queues can be used.

Example:

To create 8 TCs with 256 queues spread across all of the TCs:

```
${pathtotc}/tc qdisc add dev $iface root mqprio num_tc 8 map 0 1 2 3 4 5 6 7 \
 2@0 4@2 8@6 16@14 32@30 64@62 128@126 2@254 hw 1 mode channel
```

NOTE

E810 network adapters with 1 and 2 ports support up to 256 queue pairs (256 MSI-X interrupts) per-PF. The maximum number of queue pairs supported by 4 and 8 port E810 adapters is limited by the number of system cores.

Parameter Definitions

- **Map** — Priority mapping for up to 16 priorities to TCs. For example, map 0 0 0 0 1 1 1 1 sets priorities 0-3 to use tc0 and 4-7 to use tc1).
- **Queues** — For each TC, <*num queues*>@<*offset*>. For example, queues 2@0 16@2 assigns 2 queues to tc0 at offset 0 and 16 queues to tc1 at offset 2. A maximum total number of queues for all TCs is the number of cores.
- **hw 1 mode channel** — Channel with hardware set to 1 is a new hardware offload mode in **mqprio** that makes full use of the **mqprio** options, the TCs, the queue configurations, and the QoS parameters.

A.1.2.2 TC Filtering

NOTE

Due to timing issues, applying TC filters immediately after the `tc qdisc add` command might result in the filters not being offloaded in hardware. An error in *dmesg* is logged if the filter fails to add properly. It is recommended to wait five seconds after `tc qdisc add` before adding TC filters.

```
sleep 5
```

Rx and Tx filtering for ADQ application traffic can be configured with the **iproute2** utility. ADQ supports a wide variety of filter options and protocols, summarized here with examples listed below in this section:

IPv4 Protocol

- src/dst IP + port (Rx and Tx)
- DMAC (destination MAC)
- Forward to a queue using classid

Did this document help answer your questions?

- GTP Protocol
- VLAN Protocol
- VXLAN Protocol
- 256 queue pairs (256 MSI-X interrupts) per-PF

IPv6 Protocol

- src/dst IP + port

Queues from TC1 are used as a destination for the ADQ filter, which is a combination of the destination IP Address plus the destination port. This aligns both transmit and receive traffic on queues belonging to TC1.

NOTE

The /32 in `dst_ip` **`$ipaddr`**`/32` is not the subnet of the network being used, but the subnet of the filter you are creating. In other words, /32 indicates a single IP Address being filtered. It is recommended to use /32 when creating filters to limit the addresses being filtered.

Example Using IP Protocol

Method 1: ingress filter for RX (requires a cgroup to align TX):

```
${pathtotc}/tc qdisc add dev $iface clsact
${pathtotc}/tc filter add dev $iface protocol ip ingress prio 1 flower dst_ip \
$ipaddr/32 ip_proto tcp dst_port $app_port skip_sw hw_tc 1
${pathtotc}/tc filter show dev $iface ingress
```

Method 2: ingress/egress filters for RX/TX (cgroup not required):

```
${pathtotc}/tc qdisc add dev $iface clsact
${pathtotc}/tc filter add dev $iface protocol ip ingress prio 1 flower dst_ip \
$ipaddr/32 ip_proto tcp dst_port $app_port skip_sw hw_tc 1
${pathtotc}/tc filter add dev $iface protocol ip egress prio 1 flower src_ip \
$ipaddr/32 ip_proto tcp src_port $app_port action skbedit priority 1
${pathtotc}/tc filter show dev $iface ingress
${pathtotc}/tc filter show dev $iface egress
```

Example Using DMAC (Destination MAC)

NOTES

- This feature is supported by ice driver versions 1.9.7 and later.
- In bridged networking setups, the DMAC of the incoming packets will be the virtual interface's MAC, where PF will be added to OVS or bridge for external connectivity to VMs/Containers.

```
${pathtotc}/tc qdisc add dev $iface clsact
${pathtotc}/tc filter add dev $iface ingress protocol ip prio 1 flower skip_sw \
    dst_mac $macaddr dst_ip $ipaddr/32 hw_tc 1
${pathtotc}/tc filter show dev $iface ingress
```


Did this document help answer your questions?

Example of Forwarding to a Queue Using classid

NOTE

With *ice* driver version 1.9.7 and later, TC filtering can be used to directly forward ingress traffic to a specific queue rather than just a qdisc queue set. Forwarding packets to a specific device queue can be used as an alternative to ntuple perfect filter rules which direct each application TCP/UDP port to a specific queue within the TC range. See Configure Intel® Ethernet Flow Director Settings for more details.

```
${pathtotc}/tc qdisc add dev $iface clsact
${pathtotc}/tc filter add dev $iface protocol ip ingress prio 1 flower dst_ip \
$ipaddr/32 ip_proto tcp dst_port $app_port skip_sw classid ffff:$queue_id
```

Where:

- **$queue_id** is a hexadecimal queue id starting with an initial value of 0x1

  For example: 0xb is hexadecimal queue id and it corresponds to queue 10 [0xa + 0x1]

```
${pathtotc}/tc filter show dev $iface ingress
```

Example Using GTP Protocol

NOTES

1. The feature is supported by ice driver versions 1.7.x and later.
2. GTP filters require the Telecommunications DDP Package to be loaded on the PF port before TC configuration begins. For details on DDP and how to load the package, refer the *Intel® Ethernet Controller E810 Dynamic Device Personalization (DDP) Technology Guide*.

   The Device Dynamic Personalization (DDP) Telecommunication package can be downloaded here:

   https://downloadcenter.intel.com/download/29889/Intel-Ethernet-800-Series-Telecommunication-Comms-Dynamic-Device-Personalization-DDP-Package

```
${pathtotc}/tc qdisc add dev $iface clsact
${pathtotc}/tc filter add dev $iface protocol ip parent ffff: prio 1 flower dst_ip
$ipaddr/32 ip_proto udp dst_port $app_port enc_key_id $tunnel_id enc_dst_port
$gtp_port skip_sw hw_tc 1
${pathtotc}/tc filter show dev $iface ingress
```

Example Using VLAN Protocol

```
${pathtotc}/tc qdisc add dev $iface clsact
${pathtotc}/tc filter add dev $iface ingress protocol 802.1Q prio 1 flower vlan_id
$vlan_id vlan_ethtype $vlan_ethtype ip_proto tcp dst_ip $ipaddr/32 dst_port \
$app_port skip_sw hw_tc 1
${pathtotc}/tc filter show dev $iface ingress
```


Did this document help answer your questions?

Example Using VXLAN Protocol

NOTE

The feature is supported by ice driver versions 1.6.x and later.

```
${pathtotc}/tc qdisc add dev $vxlan clsact
${pathtotc}/tc filter add dev $vxlan protocol ip ingress flower dst_ip \
$ipaddr/32 ip_proto $protocol dst_port $app_port enc_key_id $tunnel_id hw_tc 1
${pathtotc}/ tc filter show dev $vxlan protocol ip ingress
```

Example Using IPv6 Protocol

NOTE

This feature is supported by ice driver versions 1.7.x and later.

```
${pathtotc}/tc qdisc add dev $iface clsact
${pathtotc}/tc filter add dev $iface protocol ip ingress prio 1 flower dst_ip \
$ipv6addr ip_proto tcp dst_port $app_port skip_sw hw_tc 1
${pathtotc}/tc filter show dev $iface ingress
```

NOTE

Unlike IPv4 filters, **dst ip $ipv6addr** does not require netmask to be defined.

Parameter Definitions

- **dst_ip** – IP V4/V6 Address of DUT (in VXLAN filters, it is VXLAN interface IP).
- **dst_mac** – MAC Address of DUT.
- **dst_port** – Destination port.
- **flower** – Flow-based traffic control filter.
- **ip_proto** – IP protocol to use (TCP or UDP).
- **prio** – Priority.
- **protocol** – Encapsulation protocol (valid options are IP, and 802.1Q).
- **skip_sw** – Flag to add the rule only in hardware.
- **hw_tc <*tc* >** – Route incoming traffic flow to this hardware TC. The TC count starts at 0. For example, hw_tc 1 indicates that the filter is on the second TC.
- **vlan_id** – VLAN ID.
- **vlan_ethtype** – Type of Ethernet (IPv4).
- **enc_key_ID** – GTP Tunnel Key ID (in VXLAN filters, it is network identifier).
- **enc_dst_port** – GTP port number.
- **classid** - Queue id specification
- **ffff** - qdisc ID (fixed value - `tc qdisc show dev $iface ingress/clsact`)

Did this document help answer your questions?

A.1.2.2.1 TC Filtering Requirements

Currently the resources of the Intel® Ethernet 800 Series Network Controllers support a deterministic number of TC filter types per packet *profile*, determined by the size of the fields used in the filter types.

For most implementations, filter requirements are not a restriction on implementation.

NOTE

Every filter automatically includes the destination MAC Address (3 words) even if is not explicitly specified, which does count towards the total number of matched words used by each recipe.

NOTE

Resources are per E810 device, not per port.

Resource Type	Available per E810 Device per Profile	Required Resources per Number of Matched Words in Filter Type
Chain	Depends on profile (see table below)	• If (Number of matched words) <= 4: 0 • If (Number of matched words) > 4: Floor[(Number of matched words - 1) / 4] + 1
Recipe	54	# chain resources + 1

Profiles

Packet processing profiles are defined by a set of protocols (for example, IPv4 outer IP + TCP port or IPv4 Outer IP, IP tunnel, IPV6 Inner IP). Each profile has a specific number of chain resources available globally per device (see table). Rules can be associated with more than one profile, but a rule must contain all the requested protocols in the profile to be associated. Therefore, having more specific rules narrows which profiles are associated. If any of the associated profiles do not have enough chain resources, the entire rule cannot be added.

For example, if you create a rule that looks only at outer IPv4 source address, every profile that has an outer IPV4 source field must contain enough *chain resources*, or the rule is not added.

The E810 supports the following packet processing profiles, with chain resources per profile per device in the following table.

	No Tunnel		UDP Tunnel (VXLAN), IPv4 Inner		IP Tunnel (GRE), IPv4 inner		UDP Tunnel (VXLAN), IPv6 Inner		IP Tunnel (GRE), IPv6 Inner	
IPv4 Outer	TCP	26	TCP	11	TCP	11	TCP	5	TCP	5
	UDP	26	UDP	11	UDP	11	UDP	5	UDP	5
	Other	28	Other	13	Other	13	Other	7	Other	7
IPv6 Outer	TCP	14	TCP	17	TCP	17	TCP	5	TCP	5
	UDP	14	UDP	17	UDP	17	UDP	5	UDP	5
	Other	16	Other	19	Other	19	Other	7	Other	7

Did this document help answer your questions?

TC Filter Resource Examples

A recipe is determined by the filter protocol fields, not the values of those fields. Multiple filters can be created with the same recipe, just varying values of the components.

Thus, multiple filters can be added on the same device that uses the same recipe (and requires no additional recipe resources), either on the same interface or on different interfaces on the device. Each filter uses the same fields for matching, but can have different match values.

Example: IPv4 Profile

Recipes = 54

Chain resources for TCP profile = 26

- `"${pathtotc}/tc filter add dev $iface protocol ip ingress prio 1 \ flower ip_proto tcp dst_port $app_port skip_sw hw_tc 1`

Dest MAC	...	...	TCP Dest Port

  Dest MAC (3 Words) + TCP Port (1 word) = 4 words (1 standard recipe slot)

  # chain resources = 0

  # recipes = 1

  Recipes remaining = 53

  Chain resources = 26

- `"${pathtotc}/tc filter add dev $iface protocol ip ingress prio 1 \ flower ip_proto tcp src_port $app_port skip_sw hw_tc 1`

Dest MAC	...	...	TCP Src Port

  Dest MAC (3 Words) + TCP Port (1 word) = 4 words (1 standard recipe slot)

  # chain resources = 0

  # recipes = 1

  Recipes remaining = 52

  Chain resources = 26

- `"${pathtotc}/tc filter add dev $iface protocol ip ingress prio 1 \ flower dst_ip $ipaddr/32 ip_proto tcp dst_port $app_port skip_sw hw_tc 1`

Dest MAC	...	...	TCP Dest Port	IPv4 Dest IP	...

  Dest MAC (3 Words) + TCP Port (1 word) + IPv4 IP (2 words) = 6 words

  # chain resources = floor[(6-1)/4]+1= 2

  # recipes = 3

  Recipes remaining = 49

  Chain resources = 24

If you try to add another IPv4 profile filter recipe, such as:

Did this document help answer your questions?

```
${pathtotc}/tc filter add dev $iface protocol ip ingress prio 1 \
flower src_ip $ipaddr/32 ip_proto tcp src_port $app_port skip_sw
hw_tc 1
```

Dest MAC	...	...	TCP Src Port	IPv4 Src IP	...

Dest MAC (3 Words) + TCP Port (1 word) + IPv4 IP (2 words) = 6 words

chain resources = floor[(6-1)/4]+1= 2

recipes = 3

Recipes remaining = 46

Chain resources remaining = 22

Result: As expected, there are enough resources for all desired filter combinations.

A.1.2.3 Confirm TC Configuration

1. Check TCs were created correctly.

```
${pathtotc}/tc qdisc show dev $iface
```

2. Check TC filters.

```
${pathtotc}/tc filter show dev $iface ingress
${pathtotc}/tc filter show dev $iface egress
```

NOTE

Egress filter(s) will only be listed if configured using clsact qdisc with egress tc filter(s) in TC Filtering.

A.1.3 Independent Pollers (Optional)

This must be done after TC creation.

[Optional - For some use cases] - See feature description in Independent Pollers Introduction.

A.1.3.1 Independent Pollers Introduction

ADQ 2.0 (ice driver version 1.9.x and later) adds the ability to support ADQ acceleration without application changes, allowing for a broader set of use cases for ADQ acceleration. With application dependent polling mode, a small code change in any multi-threaded application is usually required to provide a hint to the application for optimal application thread to NIC queue alignment. In addition to establishing the application threads with queue connections, busy polling is also triggered and managed by the application threads. ADQ 2.0 adds a feature referred to as *independent pollers*. Independent pollers builds on standard NAPI polling methods (kthread based NAPI polling) to allow selected ADQ queues to maintain polling for a longer period of time using a configurable timeout value.

Did this document help answer your questions?

Without the restriction for a 1:1 polling application thread to NIC queue alignment, resource grouping/aggregation can be done to optimize number of polling CPU cores. A single independent poller can poll multiple queues, and the ratio of the number of pollers to number of queues is configurable based on the application workload.

Note that the latency/throughput performance for independent poller based ADQ 2.0 may be different than an application dependent polling approach (better in some cases, worse in others). Application dependent polling is still supported for enabled use cases and is not considered deprecated. ADQ 2.0 simply expands the capability and flexibility of ADQ to enable many more environments/use cases.

This enhancement in ADQ 2.0 technology provides several benefits, including:

- ADQ acceleration without application changes
- ADQ acceleration of workloads running in containers/VMs
- Better system utilization through poller aggregation for some usages
- Per-TC value to configure poller timeout, rather than global busy_poll timeout
- Configurable number of RX/TX queue pairs per poller

Independent Poller Kernel Requirements:

- Linux kernel v5.6 or later for independent poller using ksoftirqd based polling
- Linux kernel v5.12 or later for independent poller using kthread based napi polling **(Recommended)**
- See Install OS and Update Kernel (If Needed) to determine OS and kernel requirements, and for a list of OS distributions with back-ported support for devlink params and kthread based polling.

Per-TC Configurable Parameters:

- Number of pollers (or number of queue pairs per poller)
- Poller Timeout

A.1.3.2 Configure Independent Pollers

The following variables are used in the examples in this section:

$iface	The interface in use (PF).
$iface_bdf	The network interface BDF notation (Bus:Device.Function) used by devlink (PF).
$queues	The number of queues in the TC.
$qps_per_poller	The number of queue pairs per independent poller for a given TC (max value is #queues in the TC).
$poller_timeout	The timeout value for the independent pollers for a given TC (nonzero integer value in jiffies, default value 10000).

NOTE

Independent poller feature is currently supported on PF only (not supported on VF).

Did this document help answer your questions?

NOTE

ADQ devlink parameters are only exposed when ADQ traffic class is configured via mqprio in Create TCs.

NOTE

$poller_timeout is specified in jiffies, where the size of a jiffy is determined by the kernel software clock as a measure of HZ. This differs from the global system settings for *sysctl.net.core.busy_poll* and *sysctl.net.core.busy_read*, which controls the number of microseconds to wait for packages on the device queue for socket selects (busy_poll) and socket reads (busy_read). A more detailed description of jiffies can be found in the Linux time(7) man page: https://man7.org/linux/man-pages/man7/time.7.html.

Setting global *sysctl.net.core.busy_poll* and *sysctl.net.core.busy_read settings* is NOT needed with ADQ independent poller configuration.

After creating TCs (in Create TCs):

1. Set the interface BDF, needed for devlink param commands.

```
iface_bdf=$(ethtool -i ${iface} | grep bus-info | awk '{print $2}')
```

2. Set the number of queue pairs per poller for the TC.

```
devlink dev param set pci/${iface_bdf} name tc1_qps_per_poller value \
  $qps_per_poller cmode runtime
```

NOTE

The example above is for TC1. Valid **devlink param** flags include `tc1_qps_per_poller` through `tc15_qps_per_poller`, to configure independent pollers on up to 16 TCs (max #TCs). If the TC does not exist, the devlink command will return error:

```
devlink answers: Invalid argument
```

NOTE

The maximum allowed value for **$qps_per_poller** is the number of **$queues** in the TC. If the **$qps_per_poller** value specified is greater than **$queues**, the following error will be returned by the ice driver:

```
Error: ice: Value cannot be greater than number of queues in TC.
devlink answers: Invalid argument
```

3. Set the poller timeout value for the TC.

```
devlink dev param set pci/${iface_bdf} name tc1_poller_timeout value \
  $poller_timeout cmode runtime
```


Did this document help answer your questions?

NOTE

The example above is for TC1. Valid **devlink param** flags include `tc1_poller_timeout` through `tc15_poller_timeout`, to configure independent poller timeout values on up to 16 TCs (max #TCs). If not specified, the default **$poller_timout** is set to 10000 jiffies.

4. Enable threaded napi.

```
echo 1 > /sys/class/net/$iface/threaded
```

NOTE

Independent pollers work with both threaded napi as well as default napi that runs in softirq context and triggers ksoftirqd threads.

5. Pin Independent Poller kthreads to specific CPU cores.

NOTE

Threaded napi must be enabled in order to pin application threads/pollers to specific CPU cores.

a. Get the napi threads PIDs (order from highest to lowest).

```
ps -x | grep "$iface" | head -n -1 | tac | awk '{print $1}'
```

b. Affinitize all Poller threads to a specific CPU core (skipping the PIDs that are for non-polled queues).

```
taskset -c 2 -p <pid of thread 2>
taskset -c 2 -p <pid of thread 3>
taskset -c 2 -p <pid of thread 4>
taskset -c 2 -p <pid of thread 5> etc
```

NOTE

Depending on the application workload, it is preferable to pin application threads and pollers to different CPU cores. If the application is not overly busy, running both types of threads on the same cores may yield better performance results.

6. Verify independent pollers configuration.

To query all devlink parameters:

```
devlink dev param show
```

To query a specific devlink parameter for the pci device:

```
devlink dev param show pci/${iface_bdf} name tc1_qps_per_poller
```


Did this document help answer your questions?

Example Configuration Using devlink Commands

To configure 4 queues of TC1 to be polled by each independent poller:

```
# devlink dev param set pci/0000:3b:00.0 name tc1_qps_per_poller value 4 \
  cmode runtime
```

If TC1 has a total of 8 **$queues**, the above command will configure 2 independent pollers, where each poller is used to poll 4 queues. If TC1 has a total of 16 **$queues**, the above command will configure 4 independent pollers, each polling 4 queues.

To set the timeout value in jiffies for TC1 when no traffic is flowing (setting to 5000 jiffies in this example, default is 10000 jiffies):

```
# devlink dev param set pci/0000:3b:00.0 name tc1_poller_timeout value 5000 \
  cmode runtime
```

NOTE

Clear the ADQ Configuration provides more information on clearing the ADQ configuration.

Example Configuration Using ADQ Setup Script

NOTE

The ADQ Setup script uses the *pollers* parameter with a value that specifies the total number of independent pollers to be configured for the TC. This differs from the **devlink param** commands above, which use the number of RX/TX queue pairs per poller. See ADQ Setup Using ADQ Setup Script for details on **adqseup**.

1. To configure TC1 with 8 queues using 2 independent pollers (4 queue pairs per poller) with default poller timeout:

   ```
   # adqsetup --dev=ens8f0 create myapp queues 8 pollers 2 ports 7000
   ```

2. To configure TC1 with 16 queues using 4 independent pollers (4 queue pairs per poller) with default poller timeout:

   ```
   # adqsetup --dev=ens8f0 create myapp queues 16 pollers 4 ports 7000
   ```

3. To configure TC1 with 16 queues using 8 independent pollers (2 queue pairs per poller) with modified poller timeout of 5000 jiffies:

   ```
   # adqsetup --dev=ens8f0 create myapp queues 16 pollers 8 \
   poller_timeout 5000 ports 7000
   ```

   NOTE

   The last example uses modified `poller_timeout` argument (using underscore rather than hyphen). This is a known issue with adqsetup not recognizing `poller-timeout`, fixed in **adqsetup** v1.2.3.

Did this document help answer your questions?

NOTE

The **adqsetup** examples above specify **only** the device and port (required) and parameters to configure independent pollers. Additional recommended parameters, depending on the workload:

`--priority=skbedit` to configure clsact qdisc and create egress TC filters for Tx traffic (instead of using Linux cgroups to align Tx packets to queues)

`--rxadapt=off --rxusecs=0 --txadapt=off --txusecs=500` to set the interrupt moderation rate to a static value for Tx and turn off interrupt moderation for Rx. These values are recommended for most ADQ workloads. Setting rx-usecs to a non-zero value, such as 50 or 100, might improve interrupt handling and ADQ performance for certain applications and workloads. Experiment with `rxusecs=0, 50, 100` to determine best ADQ performance.

NOTE

Global system settings for `--busypoll`, `--busyread`, `--optimize`, `--bpstop`, and `--bpstop-cfg` are not used when `pollers` is specified. Independent pollers use the **$poller_timeout** for the TC, rather than global sysctl settings.

Did this document help answer your questions?

A.1.4 Apply Adapter Tuning

This must be done after TC creation.

The following variable is used in the examples in this section:

$iface	The interface in use (PF or VF).
$mask	The queue mask to configure per-queue coalesce.

1. Configure the adapter tuning parameters with **ethtool** (might vary per workload).

   To configure coalesce on all queues:

```
ethtool --coalesce $iface adaptive-rx off rx-usecs 0
ethtool --coalesce $iface adaptive-tx off tx-usecs 500
ethtool --show-coalesce $iface
```

   To configure per-queue coalesce:

```
ethtool --per-queue $iface queue_mask $mask --coalesce \
  adaptive-rx off rx-usecs 0 adaptive-tx off tx-usecs 500
ethtool --per-queue $iface queue_mask $mask --show-coalesce
```

   Example to configure coalesce settings on 8 queues of TC1, using queue_mask 0x3fc for queues 2-9 (0011 1111 1100):

```
ethtool --per-queue ens8f0 queue_mask 0x3fc -coalesce \
  adaptive-rx off rx-usecs 0 adaptive-tx off tx-usecs 500
ethtool --per-queue ens8f0 queue_mask 0x3fc --show-coalesce
```

   NOTE

   ethtool version 5.1 or later is required for per-queue coalesce support.

   NOTE

   Setting rx-usecs to a non-zero value, such as 50 or 100, might improve interrupt handling and ADQ performance for certain applications and workloads. Experiment with rx-usecs=0, 50, 100 to determine best ADQ performance.

   NOTE

   Coalesce settings must be re-applied after all TC and TC filter creation.

2. **[Optional]** Run the **set_irq_affinity** script for all interfaces (included in the *scripts* folder of the *ice* package) to pin interface IRQs to CPU cores.

```
${pathtoicepackage}/scripts/set_irq_affinity -X all $iface
```


Did this document help answer your questions?

NOTE

The command above sets IRQ affinity for all CPU cores in the system, on both local and remote NUMA nodes (CPU sockets). For best performance with applications using cores on one CPU, it is recommended to constrain the CPU cores used to the local NUMA node of the Intel® Ethernet 800 Series network adapter under test. To set IRQ affinity using only CPU cores on the local NUMA node:

```
${pathtoicepackage}/scripts/set_irq_affinity -X local $iface
```

NOTE

Affinity settings have to be re-applied after all TC and TC filter creation.

A.1.5 Configure Symmetric Queues

This must be done after TC creation.

Symmetric queuing method is used for aligning egress/ingress traffic on a pre-configured Tx/Rx queue pair. XPS using receive queues map is used to select transmit queue based on the receive queue(s) map configuration set by the administrator. A 1:1 mapping between receive queues and transmit queues enables sending packets on the same queue associations for transmit and receive.

XPS is only available if the **kconfig** symbol CONFIG_XPS is enabled. As specified in the kernel build section, the kernel needs to compiled with CONFIG_SMP=y for this. For symmetric transmit queue selection based on receive queues map, xps_rxqs map attribute for the transmit queue needs to be configured using the sysfs file entry.

The following variables are used in the examples in this section:

$iface	The interface in use (PF or VF).
tx-${i}	The transmit queue being configured.
$mask	A bitmap of receive queues that might use the transmit queue.

Automatic configuration can be done easily using the script included in the scripts folder of the *ice* driver package:

```
${pathtoicepackage}/scripts/set_xps_rxqs $iface
```

Or configuration can be done manually by echoing a CPU mask into the desired Tx queue's xps_rxqs location:

```
echo ${mask} > /sys/class/net/${iface}/queues/tx-${i}/xps_rxqs
```

Example:

```
echo 10 > /sys/class/net/eth0/queues/tx-4/xps_rxqs
```

This enables egress traffic to be sent on transmit queue 4 when ingress traffic is received on receive queue 4.

Did this document help answer your questions?

NOTE

An additional requirement for symmetric queuing to work correctly with ADQ, is 1) cgroup configuration, or 2) setsockopt with SO_PRIORITY. Both are outlined in Set cgroup Priority for Application Network Traffic (If Needed). Failure to configure cgroup or SO_PRIORITY will result in Tx traffic going to default TC queues.

A.1.6 Configure Intel® Ethernet Flow Director Settings

Linux traffic control filtering allows an application to be mapped to a set of specified queues on an interface, using dedicated queue sets (also called *traffic classes*) for application traffic with ADQ. Intel® Ethernet Flow Director allows for incoming connections to be distributed and load balanced across these application queues, within the queue set, in a more reliably defined way than the default RSS hash. This is important to the inner workings of ADQ because it is recommended that no two application threads are busy polling on the same queue.

There are two options for Intel® Ethernet Flow Director configuration in the *ice* driver:

- Option 1: *channel inline* or *per-TC inline*
- Option 2: *ntuple*

The following variables are used in the examples in this section:

$iface	The interface in use (PF).
$iface_bdf	The network interface BDF notation (Bus:Device.Function) used by devlink (PF).
$port	The starting port of the application.
$queues	The number of queues in the TC.
$tc1offset	The offset for the application TC (for example, if TC0 has 2 queues, TC1 offset would be 2).

NOTE

Commands in this section must be applied to the PF interface (not available on VF interfaces).

Option 1: channel inline or per-TC inline

Intel® Ethernet Flow Director distributes connections to queues in a round-robin fashion. This is recommended for ADQ-enabled, single-instance, multi-threaded applications that manage their own worker thread load distribution (such as **Memcached**).

***ice* version 1.7.x (and earlier):**

channel inline: Intel® Ethernet Flow Director is configured using a global private flag and gets applied to all TCs on the interface:

```
ethtool --set-priv-flags $iface channel-inline-flow-director on
```


Did this document help answer your questions?

***ice* version 1.8.x (and later):**

per-TC inline: Intel® Ethernet Flow Director is configured using a per-TC configuration and gets applied to only the queues within the TC specified for the interface. A different devlink command is required for each TC. Example for tc1:

```
iface_bdf=$(ethtool -i ${iface} | grep bus-info | awk '{print $2}')
devlink dev param set pci/${iface_bdf} name tc1_inline_fd value true cmode runtime
```

NOTE

Kernel with **devlink param** support is required for *ice-1.8.x* and later. See Install OS and Update Kernel (If Needed) to determine OS and kernel requirements.

NOTE

Valid **devlink param** flags include `tc1_inline_fd` through `tc15_inline_fd`, to configure inline flow director on up to 16 TCs (max #TCs).

NOTE

XPS using receive queues map must be configured for per-TC inline flow director to work correctly. See Configure Symmetric Queues to configure XPS.

Option 2: ntuple

Intel® Ethernet Flow Director assigns perfect filters to direct each application TCP port to a specific queue within the application TC range. This is useful for applications (such as Redis) where each instance of the application runs a single worker thread, and load balancing is done either manually or through a separate load balancer application.

```
ethtool --features $iface ntuple on
for ((i=0; i < queues; i++)) do
  ethtool --config-ntuple $iface flow-type tcp4 dst-port $((port + i)) action \
  $(((i %queues) + tc1offset))
  sleep 0.5
done
#verify set correctly
ethtool --show-ntuple $iface
```

NOTE

If `rmgr: Cannot insert Rx class rule: Invalid argument` error message appears, verify that no previous assignment is in place:

```
ethtool --show-ntuple $iface
```

Assignments as a whole can be deleted:

```
ethtool --features $iface ntuple off
```


Did this document help answer your questions?

NOTE

Linux forward to queue using class ID is an alternative to ntuple filter rules using the action *forward to queue* to forward packets to a specific device queue. Ntuple can set the specific packet flow types (i.e., tcp4, udp4) and depends on the ethtool version, RHEL 8.x, or upstream kernel to configure Ntuple flags, whereas forward to queues can be set per-filter using the class ID parameter. See TC Filtering, Example of Forwarding to a Queue Using classid as an alternative to ntuple filter rules in this section.

A.1.7 Set cgroup Priority for Application Network Traffic (If Needed)

Linux Control Groups (cgroups) are used to steer outgoing (Tx) traffic to a set of dedicated hardware queues on the NIC. If egress tc filter(s) were configured in TC Filtering, Linux cgroups are not needed and this step can be skipped. If only ingress tc filter(s) were created, follow the steps in this section to create a cgroup and set the priority of network traffic to align TX packets to the Rx filter queues.

NOTE

The performance of the two methods (cgroup priority or TC filters with skbedit priority) may differ slightly based on the application or workload.

The following variables are used in the examples this section:

$cgroup_name	The name for the application cgroup.
$app_pid	The process ID for the application.
$iface	The network interface that the application uses (PF or VF).
$prio	Defines the network priority for the applications belonging to cgroup ${cgroup_name}.
--sticky	An option before the command to keep any child processes in the same cgroup.
$command	The task command with arguments to be executed in the cgroup.

1. Configure cgroup.

   The Network Priority (net_prio) subsystem of the Control Groups (cgroups) Linux kernel feature provides a way to set the priority of network traffic per each network interface for applications within various cgroups. Typically, an application sets the priority of its traffic via the SO_PRIORITY socket option. However, if applications are not coded to set the priority value, the administrator can assign a process to a specific cgroup which defines the priority of outgoing traffic on a given network interface.

   NOTE

   Network Priority (net_prio) using cgroup configuration in this section, or the SO_PRIORITY socket option in the application, must be configured for symmetric queuing (Configure Symmetric Queues) to work correctly with ADQ. Failure to configure either 1) Network Priority (net_prio) using cgroups, or 2) SO_PRIORITY in the application, will result in Tx traffic going to the default TC queues.

Did this document help answer your questions?

There are two options for cgroup configuration: manually, or through the **cgconfig** service tools.

Option 1: Manual cgroup Configuration

NOTE

RHEL9.0 does not include the net_prio module by default. A net_prio directory must be created under `/sys/fs/cgroup/` before mounting it. For example:

```
mkdir /sys/fs/cgroup/net_prio
```

This step is not required in RHEL8.X or earlier versions.

a. Mount the cgroup filesystem.

```
mount -t cgroup -o net_prio none /sys/fs/cgroup/net_prio
```

b. Create a new cgroup.

```
mkdir /sys/fs/cgroup/net_prio/${cgroup_name}
```

c. Assign the application process to cgroup.

```
echo $app_pid > /sys/fs/cgroup/net_prio/${cgroup_name}/tasks
```

NOTE

Before running this command, start the server application (for example, redis server, netperf, etc.) and include the specific application PID as **$app_pid**.

d. Set priority to **$prio** for traffic originating from processes belonging to the `net_prio` cgroup.

```
echo "$iface $prio" > /sys/fs/cgroup/net_prio/${cgroup_name}/ \
net_prio.ifpriomap
```

Example:

```
echo "eth0 1" > /sys/fs/cgroup/net_prio/foo/net_prio.ifpriomap
```

Option 2 (Recommended): cgroup Configuration Using CGConfig Service Tools

a. Make a net_prio directory and mount it in the /sys/fs/cgroup directory.

```
mkdir /sys/fs/cgroup/net_prio
mount -t cgroup -o net_prio none /sys/fs/cgroup/net_prio
```

NOTE

This step is only applicable for RHEL9.0 and later operating systems because the net_prio module is not available in the OS repository.

Did this document help answer your questions?

b. Create a new cgroup.

```
cgcreate -g net_prio:${cgroup_name}
```

c. Set the priority for processes belonging to the cgroup.

The `$prio` value should map to the position of the targeted TC defined in Create TCs.

```
cgset -r net_prio.ifpriomap="$iface $prio" ${cgroup_name}
```

Example:

For example, in the following TC command, `map 0 0 1 2` sets priority 0-1 to use TC0, prio 2 to TC1, and prio 3 to TC2.

```
${pathtotc}/tc qdisc add dev $iface root mqprio num_tc 3 map \
0 0 1 2 queues 2@0 40@2 54@42 hw 1 mode channel
cgcreate -g net_prio:app_tc1
cgset -r net_prio.ifpriomap="$iface 2" app_tc1

cgcreate -g net_prio:app_tc2
cgset -r net_prio.ifpriomap="$iface 3" app_tc2
```

2. Run the task in the given cgroup (for example, `${cgroup_name}`)

```
cgexec -g net_prio:${cgroup_name} --sticky $command
```

Example:

```
cgexec -g net_prio:${cgroup_name} --sticky netserver
```

3. Verify cgroup configuration:

a. Verify that the interface is mapped to the right interface:

```
cat /sys/fs/cgroup/net_prio/${cgroup_name}/net_prio.ifpriomap
```

b. Show the Process IDs being run in this cgroup and match them to the application Process IDs:

```
cat /sys/fs/cgroup/net_prio/${cgroup_name}/tasks
```

Example:

For convenience, the commands in this section are condensed below with example values. Variables must be changed to match each environment.

Did this document help answer your questions?

NOTE

ADQ configuration might be application-specific depending on whether the application is single-threaded or multi-threaded. Different tunings or filtering methods might apply. See specific application sections in this document for further reference.

```
#Variables:
#The interface in use.
iface=ens8f1
#The path to TC installation (default /usr/sbin).
pathtotc="/usr/sbin"
#The path to the ice driver package.
pathtoicepackage=/opt/ice-1.8.8
#The number of queues for default traffic (for example, 2).
num_queues_tc0=2
#The number of queues for application traffic class (also maximum number of
application threads to be run).
num_queues_tc1=8
#The IP Address of the interface under test on the SUT.
ipaddrserver=13.100.2.13
#The starting port of the application (any high numbered port).
app_port=6000
#Name of Linux cgroup.
cgroup_name=app_tc1
#Command to run inside cgroup on SUT.
command=servercmd

#Commands (using prio=1 for tc1):
${pathtotc}/tc qdisc add dev $iface root mqprio num_tc 2 map 0 1 queues \
$num_queues_tc0@0 $num_queues_tc1@$num_queues_tc0 hw 1 mode channel
sleep 5
${pathtotc}/tc qdisc add dev $iface clsact
${pathtotc}/tc filter add dev $iface protocol ip ingress prio 1 flower dst_ip \
${ipaddrserver}/32 ip_proto tcp dst_port $app_port skip_sw hw_tc 1
${pathtotc}/tc qdisc show dev $iface
${pathtotc}/tc filter show dev $iface ingress
ethtool --coalesce ${iface} adaptive-rx off rx-usecs 0
ethtool --coalesce ${iface} adaptive-tx off tx-usecs 500
ethtool --show-coalesce $iface
${pathtoicepackage}/scripts/set_irq_affinity -X all $iface
${pathtoicepackage}/scripts/set_xps_rxqs $iface

#ice version 1.7.X (and earlier):
ethtool --set-priv-flags $iface channel-inline-flow-director on

#ice version 1.8.X (and later):
iface_bdf=$(ethtool -i ${iface} | grep bus-info | awk '{print $2}')
devlink dev param set pci/${iface_bdf} name tc1_inline_fd value true cmode runtime

cgcreate -g net_prio:${cgroup_name}
cgset -r net_prio.ifpriomap="$iface 1" ${cgroup_name}
cgexec -g net_prio:${cgroup_name} --sticky $command
cat /sys/fs/cgroup/net_prio/${cgroup_name}/net_prio.ifpriomap
cat /sys/fs/cgroup/net_prio/${cgroup_name}/tasks
```


Did this document help answer your questions?

A.1.8 Verify ADQ Application Traffic and Independent Pollers (If Applicable)

1. Verify that ADQ application traffic is on the correct queues.

   While ADQ application traffic is running, watch **ethtool** statistics to check that only the ADQ queues are being used (have significant traffic) with busy poll (pkt_busy_poll) for ADQ traffic. If non busy poll (pkt_not_busy_poll) have significant counts and/or if traffic is not confined to ADQ queues, recheck the configuration steps carefully.

   Example:

```
watch -d -n 0.5 "ethtool -S $iface | grep busy | column"
```

   The above **watch** command displays pkt_busy_poll and pkt_not_busy_poll counters for both Tx and Rx packets, refreshing the counters every half (0.5) second. For a configuration with 2 traffic classes, where TC0 has 2 queues configured for non-ADQ traffic and TC1 has 4 queues for ADQ traffic, the following is example **watch** output on the ADQ server when an application benchmark is initiated from client(s):

```
Every 0.5s: ethtool -S ens2f0 | grep busy | column
  tx_busy: 0                            rx_0.pkt_busy_poll: 0
  tx_0.pkt_busy_poll: 0                 rx_0.pkt_not_busy_poll: 7720
  tx_0.pkt_not_busy_poll: 529           rx_1.pkt_busy_poll: 0
  tx_1.pkt_busy_poll: 0                 rx_1.pkt_not_busy_poll: 3615
  tx_1.pkt_not_busy_poll: 4649          rx_2.pkt_busy_poll: 104353
  tx_2.pkt_busy_poll: 108492            rx_2.pkt_not_busy_poll: 2692
  tx_2.pkt_not_busy_poll: 1539          rx_3.pkt_busy_poll: 104219
  tx_3.pkt_busy_poll: 108593            rx_3.pkt_not_busy_poll: 2829
  tx_3.pkt_not_busy_poll: 1214          rx_4.pkt_busy_poll: 104227
  tx_4.pkt_busy_poll: 108393            rx_4.pkt_not_busy_poll: 2879
  tx_4.pkt_not_busy_poll: 1481          rx_5.pkt_busy_poll: 103973
  tx_5.pkt_busy_poll: 108413            rx_5.pkt_not_busy_poll: 2922
  tx_5.pkt_not_busy_poll: 1518          rx_6.pkt_busy_poll: 104332
```

NOTES

- The pkt_busy_poll counters are enabled by the `-DADQ_PERF_COUNTERS CFLAG` during ice driver compile. If the counters are not listed in **ethtool -S** output, verify ice was built using the compile command listed in Step 1 of Install ice Driver with ADQ Flags.
- The pkt_busy_poll counters are refreshed and incremented on the ADQ server while application traffic is actively running on the client(s).
- If pkt_not_busy_poll counters are being incremented on ADQ queues, refer to Troubleshooting for guidance.

Did this document help answer your questions?

2. Verify independent poller threads (if applicable).

   When ADQ application traffic is running with one or more independent pollers configured, system monitoring tools such as `top/htop` and `ps` will show N ksoftirqd kernel threads running (with kernel version 5.12+), where N is the number of kernel independent poller threads. For example, if TC1 is configured with 8 queues and `tc1_qps_per_poller` is set to 4, there will be 2 ksoftirqd poller threads in the `top` output, and `htop` output will show 2 CPU cores used @100% by kernel poller threads.

   Application traffic can be pinned either to the same CPU cores, or different cores, than the independent poller threads. There may be slight performance improvements using different CPU cores for the kernel pollers. However, if reducing the total number of CPU cores used by ADQ is important, overlapping application threads and independent poller threads is an option.

A.1.9 Clear the ADQ Configuration

To clear ADQ configuration after finishing testing, it is recommended to remove TC filters first before removing the TCs. Note that if the TCs are removed first, any existing TC filters will not function properly even if the TC is re-added. All TC filters must be added after the TC creation step.

NOTE

Use **$iface_vf** to clear the VF ADQ configuration, and use **$iface** to clear the PF ADQ configuration.

1. Remove ntuple rules (if applicable).

```
ethtool --features $iface ntuple off
```

2. Remove the cgroup (if applicable).

```
cgdelete -g net_prio:${cgroup_name}
```

3. Remove all Traffic Class (TC) filters on the interface under test. The following example removes an ingress and egress filter with priority 1:

   To remove PF-specific filters:

```
${pathtotc}/tc filter del dev $iface protocol ip ingress prio 1
${pathtotc}/tc filter del dev $iface protocol ip egress prio 1
```

   To remove VXLAN-specific filters:

```
${pathtotc}/tc filter del dev $vxlan ingress
```

4. Remove all TCs on the interface under test.

```
${pathtotc}/tc qdisc del dev $iface clsact
${pathtotc}/tc qdisc del dev $iface root mqprio
```


Did this document help answer your questions?

5. To remove VF interfaces:

```
echo 0 > /sys/class/net/$iface_pf/device/sriov_numvfs
```

NOTE

This step is applicable when the system is configured with ADQ VF.

6. Disable pollers.

```
iface_bdf=$(ethtool -i $iface | grep bus-info | awk '{print $2}')
devlink dev param set pci/$face_bdf name num_qps_per_poller value 0 cmode
runtime
```

7. Turn off napi threading.

```
echo 0 > /sys/class/net/$iface/threaded
```

NOTE

Steps 6 and 7 are applicable when the ADQ setup is configured with independent pollers (ADQ2.0) and kthread based napi polling.

8. Confirm configuration is cleared.

```
${pathtotc}/tc filter show dev $iface ingress
${pathtotc}/tc filter show dev $iface egress
${pathtotc}/tc qdisc show dev $iface
```

NOTE

It is recommended to remove ntuple rules, cgroup and TC filters first, before removing the TCs.

An alternative is to reload the *ice* driver to remove any (and all) TC configurations, system wide. Reloading the *ice* driver should **only** be done if there are no other E810 interfaces in use on the system, as this will disrupt **all** network traffic.

```
rmmod ice
sleep 2
modprobe ice
```

NOTE

Add at least 2 seconds of sleep time to restore all queues before reconfiguring the ADQ traffic classes.

Did this document help answer your questions?