Predictable Network Traffic in K8s

Dave Cremins – Cloud Software Architect (xPSD)

Abdul Halim – Senior Software Engineer (xPSD)

Agenda

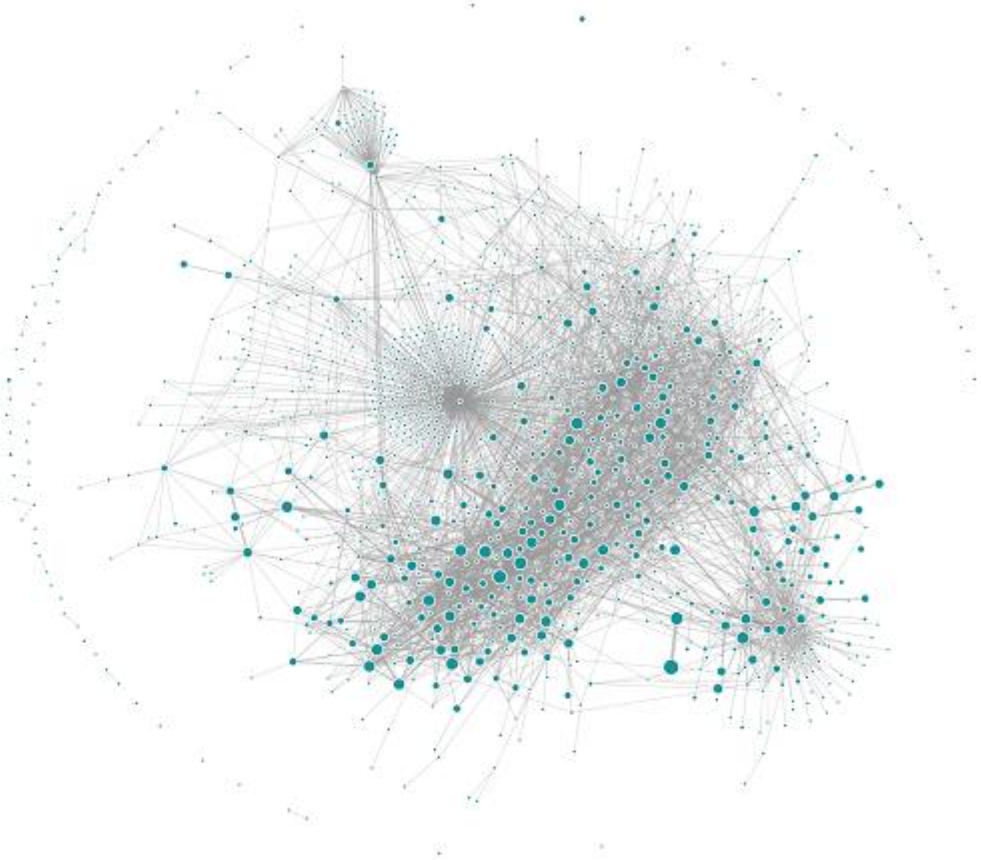
- Current state of microservices from a networking perspective
- Network contention in Cloud Native deployments
- Application Device Queues
- Orchestration flow
- Some preliminary results

Microservices

- Software pattern that promotes the decomposition of an application into small operating pieces with well-defined boundaries of functionality
- The individual pieces (i.e. services) are integrated together via API interfaces in a loosely coupled environment
- Commonly packaged as containers and deployed into orchestration platforms
- These API interfaces rely heavily on the network in order to communicate with each other
    - Huge increase in East-West traffic
    - 100s of services/containers running on a single compute platform
    - Potentially 1000s running on distributed platforms
    - Full request/response cycle will pass through many microservices
- What about multi-tenant deployments in the context of Cloud Native & Hyperscalers?

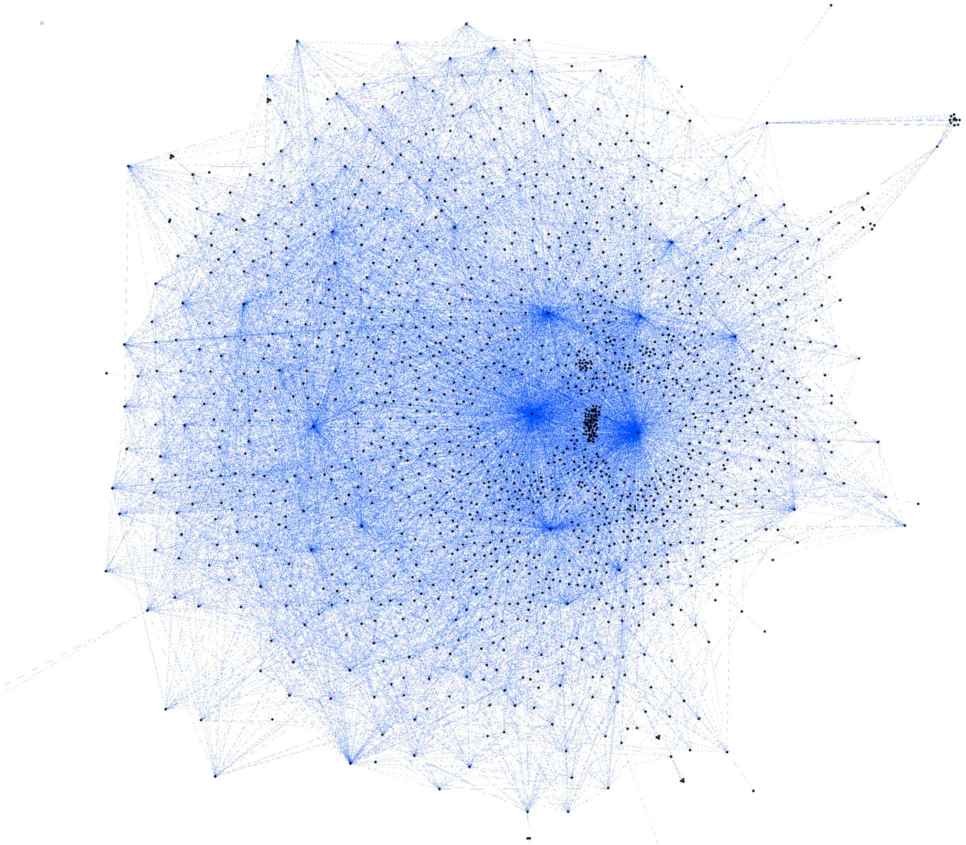
Cloud Native Microservices Examples

Uber graph 2018



~2200 Microservices

Monzo graph 2019



1500 Microservices

Let’s revisit the networking aspect

- More microservices means more EW traffic => results in more demand/dependency on the network
- As traffic increases, extra jitter can result in unpredictable response times for services
- Net affect of all of these concerns is degradation of SLAs via reduced performance and increased latency
- We need a way to prioritize communication for specific services i.e. More predictability for higher priority applications

Ethernet is like a freeway system for data travelling between different systems in a distributed environment

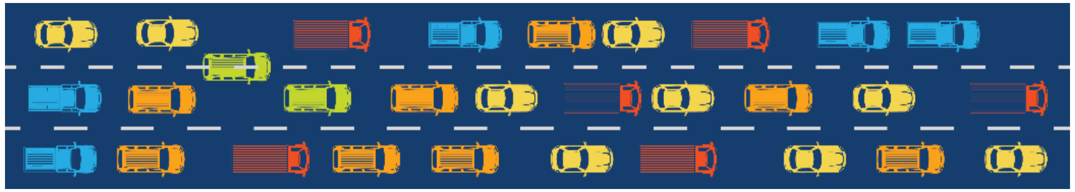
Application Device Queues (ADQ)

- ADQ is designed to improve application specific queuing and steering

- ADQ works by:
  - Filtering application traffic to a dedicated set of queues
  - Application threads of execution are connected to specific queues within the ADQ queue set
  - Bandwidth control of application egress (Tx) network traffic

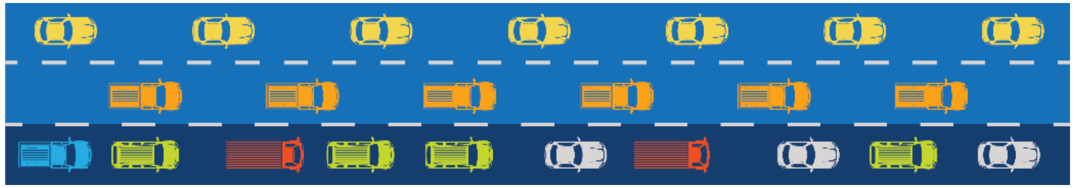
Without ADQ

Application traffic intermixed with other traffic types



With ADQ

Application traffic to a dedicated set of queues



- ADQ Benefits:

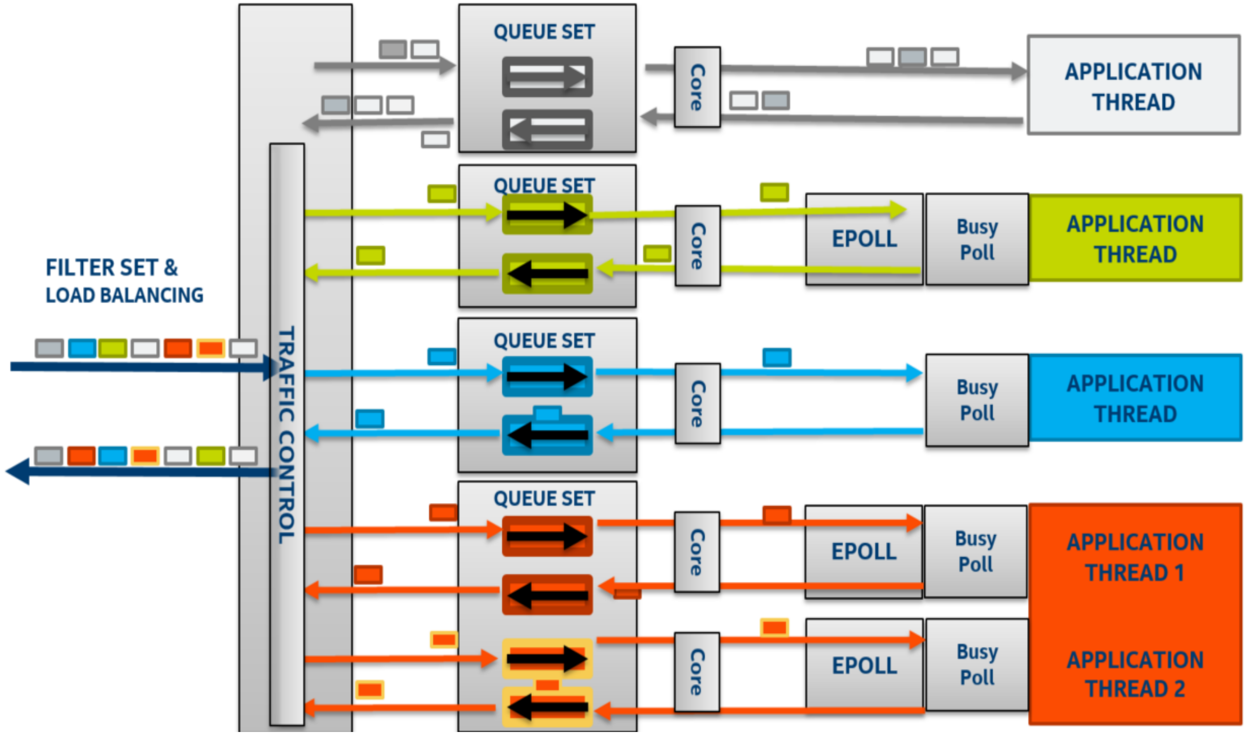
INCREASES APPLICATION PREDICTABILITY

REDUCES APPLICATION LATENCY

IMPROVES APPLICATION THROUGHPUT

ADQ: Hardware View

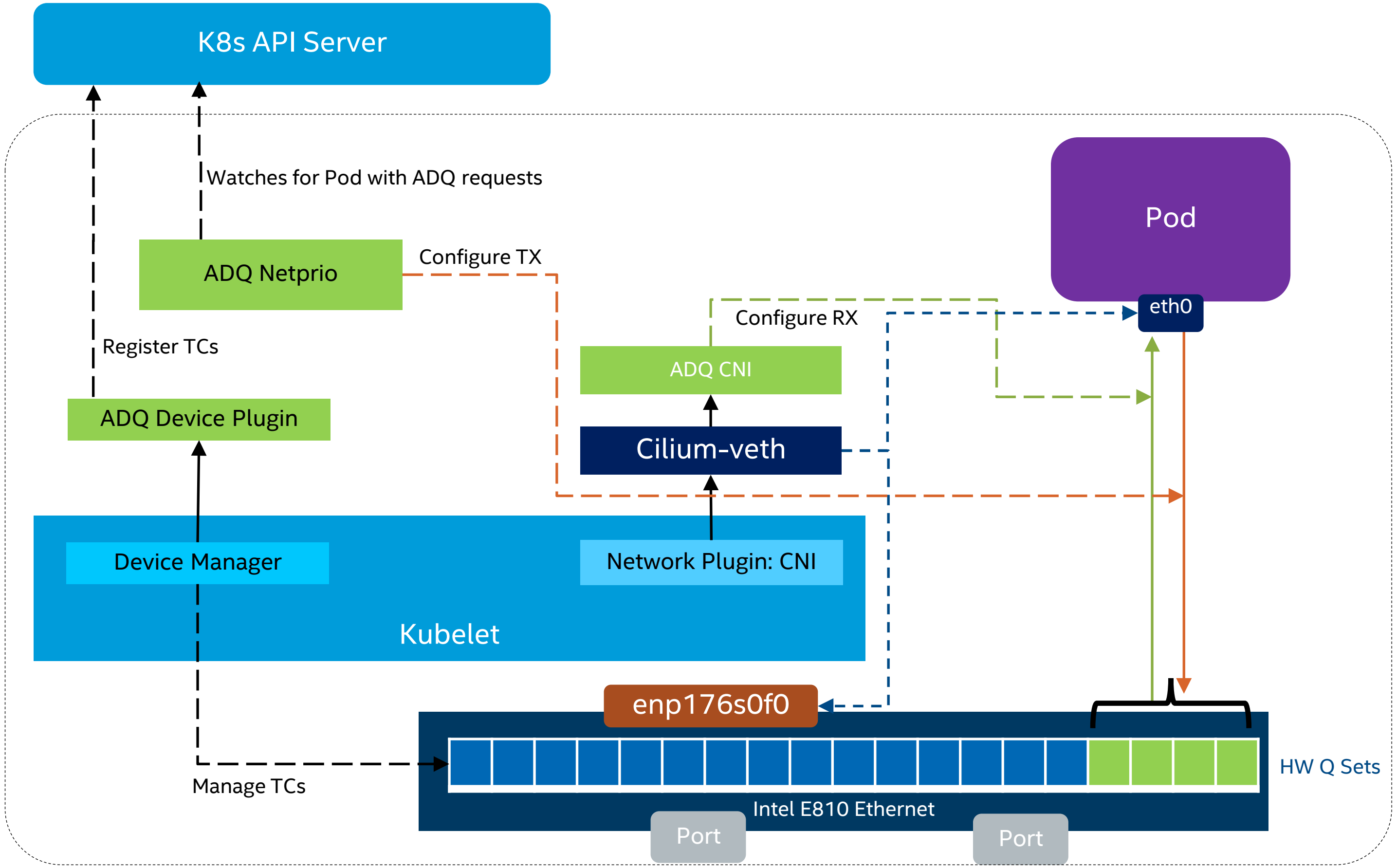
- Enables a path from Epoll that leverages BPS (Busy Polling Sockets), polling is configured on the platform
- Provides a "hint" for application threads to monitor specific sockets and align
  - e.g., sockets on the same queue, handled by the same thread
  - Single producer-consumer per queue affinitization
- Configures an application identifier on the NIC to steer traffic to dedicated load balanced queues
- Configures TX rate limiting on the NIC per application identifier
- Performance optimizations in the NIC driver
  - interrupts and load balancing optimizations



Application Device Queues in Kubernetes

Application Device Queues in Kubernetes

- Resource management – K8s Device Plugin
  - Accountability of HW queues on host
  - HW queue allocations for containers/apps
  - Scheduling + on-node allocation
- RX configuration – CNI plugin
  - Configures HW queue filters using Pods IP and application port info
  - Deployed as a CNI chain with Cillium CNI with veth mode
  - Application information(port/protocol) via Pod spec
- TX configuration – cgroup net_prio
  - Watches for "readiness" of Pod with ADQ resources
  - Finds Pods cgroup information, and adds net_prio for its network interface



Requesting ADQ: Memcached server

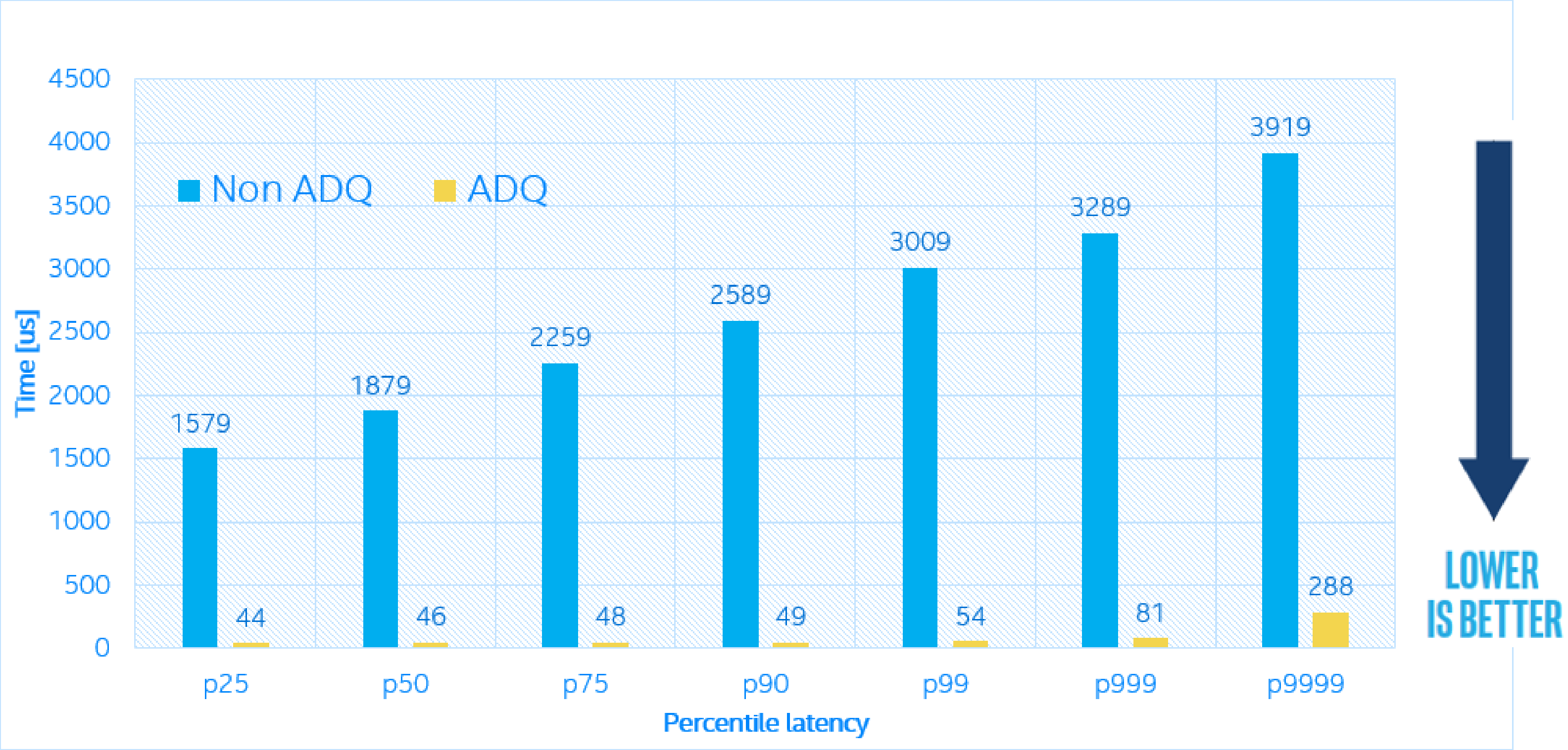
```
---
apiVersion: v1
kind: Pod
metadata:
  name: memcached-adq
  namespace: adqb
  labels:
    app: memcached-server
  annotations:
    net.v1.intel.com/adq-config: '[ { "name": "memcached", "ports": { "local": ["11211/TCP"] } } ]'
spec:
  nodeSelector:
    adq-benchmark: server
  hostname: memcached-adq
  subdomain: memcached-servers
  containers:
  - name: memcached
    image: memcached:1.6.10
    imagePullPolicy: IfNotPresent
    command: ["memcached"]
    args: ["-t", "4", "-N", "4", "-c", "5000", "-p", "11211", "-M", "-o", "lru_maintainer"]
    resources:
      limits:
        cpu: 4
        memory: 1Gi
        net.intel.com/adq: 1
    ports:
    - containerPort: 11211
    readinessProbe:
      tcpSocket:
        port: 11211
```

Requesting ADQ: Memcached client

```
---
apiVersion: v1
kind: Pod
metadata:
  name: memcached-bench-adq
  namespace: adqb
  annotations:
    net.v1.intel.com/adq-config: '[ { "name": "memcached-client", "ports": { "remote": ["11211/TCP"] } } ]'
spec:
  nodeSelector:
    adq-benchmark: client
  restartPolicy: Never
  containers:
  - name: memcached-client
    image: rpc-perf:v0.1
    command: ["sleep", "36000"]
    resources:
      limits:
        cpu: 4
        memory: 1Gi
        net.intel.com/adq: 1
    volumeMounts:
    - name: config
      mountPath: /etc/rpc-perf/config
  volumes:
  - name: config
    configMap:
      name: rpc-perf-cm
```

Latency comparison: no ADQ vs with ADQ

Latency – VETH



- Background traffic generated with iperf3
- CPU: Intel(R) Xeon(R) Gold 6238L @2.10GHz

Closing comments

- ADQ is a technology designed to improve application specific queuing and steering
- It allows filtering of application traffic to a dedicated set of queues
- It optimizes how data polling is performed
- ADQ addresses three important factors: **predictability, latency, and throughput**
- K8s orchestration code is in final stages to be open sourced
- Waiting for the following features to be up-streamed:
  - Tc flower forward to hw queue filters
  - Per-tc inline flow director
  - Per-tc qps_per_poller
  - Per-tc poller_timeout
- Kernel version 4.19+