ISOVALENT

Better Bandwidth Management with eBPF

Daniel Borkmann, Cilium Team & eBPF co-maintainer

Christopher M. Luciano, Cilium Team

Nikolay Aleksandrov, Cilium Team

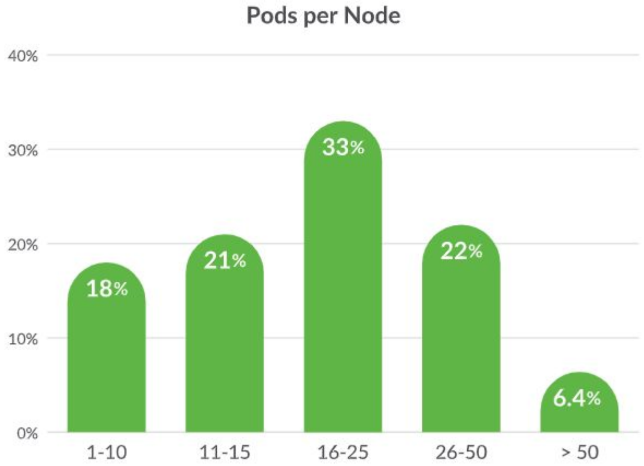
Problem Statement

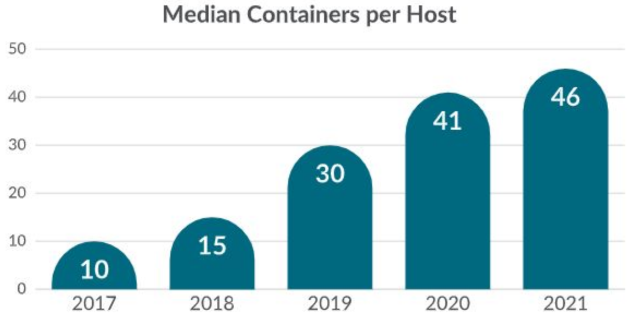
- Increasing Pod density per node
- Competition for node resources e.g. CPU and memory
- Optimization problem for operators: Resource allocation and efficient use, achieving SLOs, etc

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: app
    image: images.my-company.example/app:v4
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

What the container is guaranteed to get, e.g. kubelet will only schedule the Pod on a node which can provide this resource.

Hard upper limit, ensures that container never goes above this threshold.





Source: Sysdig 2022 Cloud Native Security and Usage Report

Problem Statement

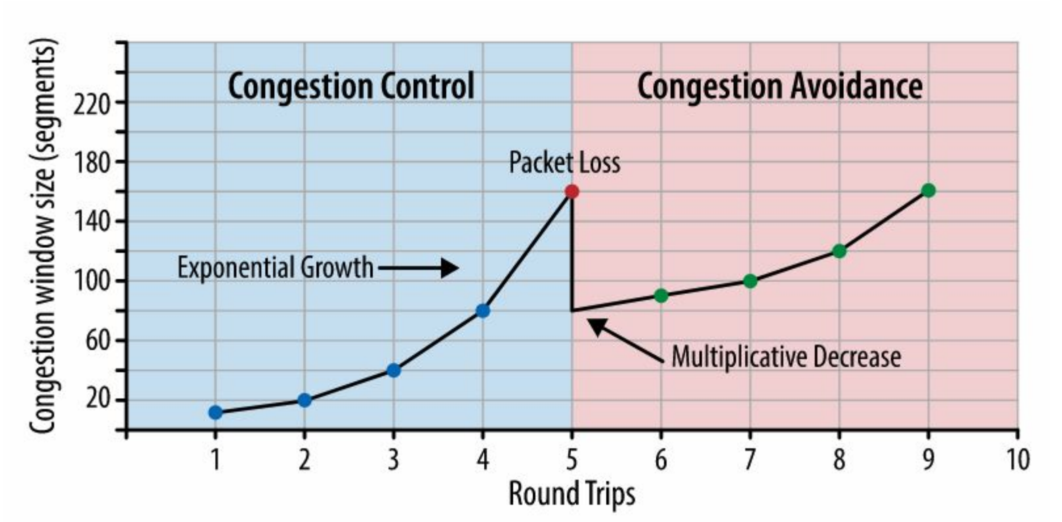
➔ But what about networking?

➔ TCP sends AFAP
(as fast as possible)

➔ AFAP output contract, shaping typically implemented by device output queues

➔ Queue length limit & receive window determines in-flight rate

➔ “How fast” implicit in queue drain rate



Source: https://hpbn.co/building-blocks-of-tcp/

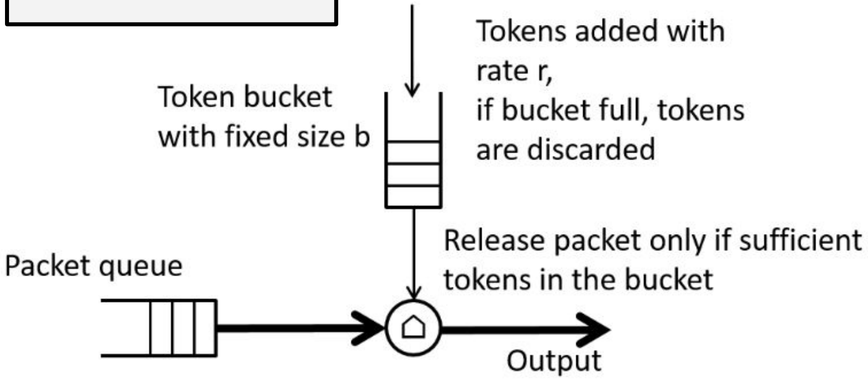
➔ Who limits a Pod’s network usage in Kubernetes?

Problem Statement

- ➔ Kubernetes bandwidth enforcement has only been experimental so far :-(
- ➔ Support for Pod annotations with ‘outsourced’ bandwidth meta plugin

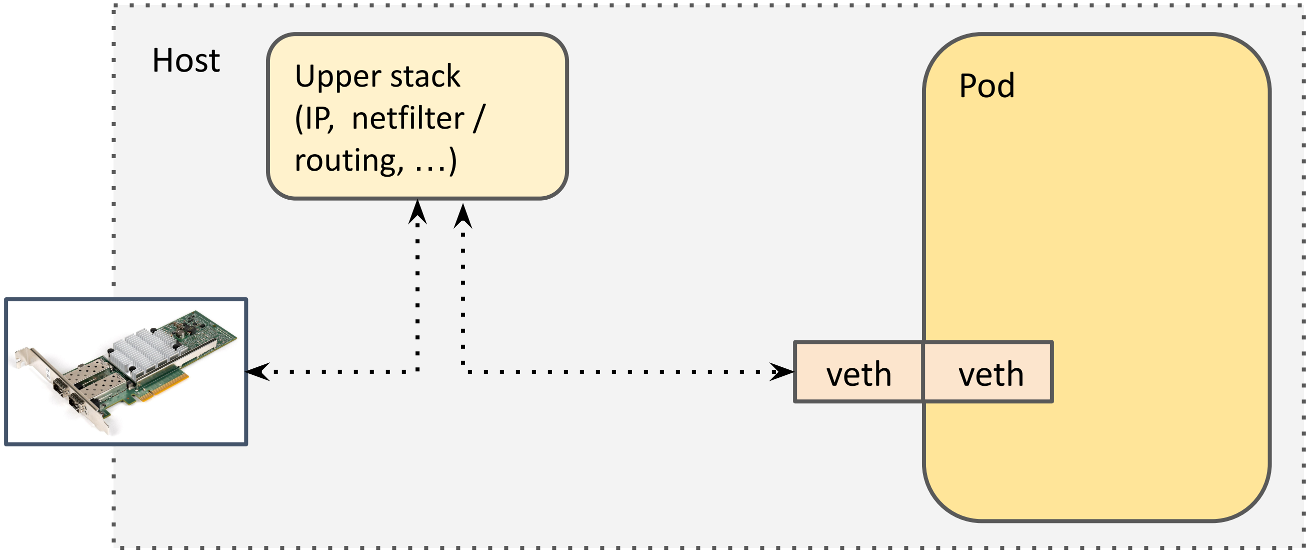
```yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubernetes.io/ingress-bandwidth: 1M
    kubernetes.io/egress-bandwidth: 1M
...
```

Adds rudimentary **token bucket filters (TBF)** to implement enforcement of annotations.



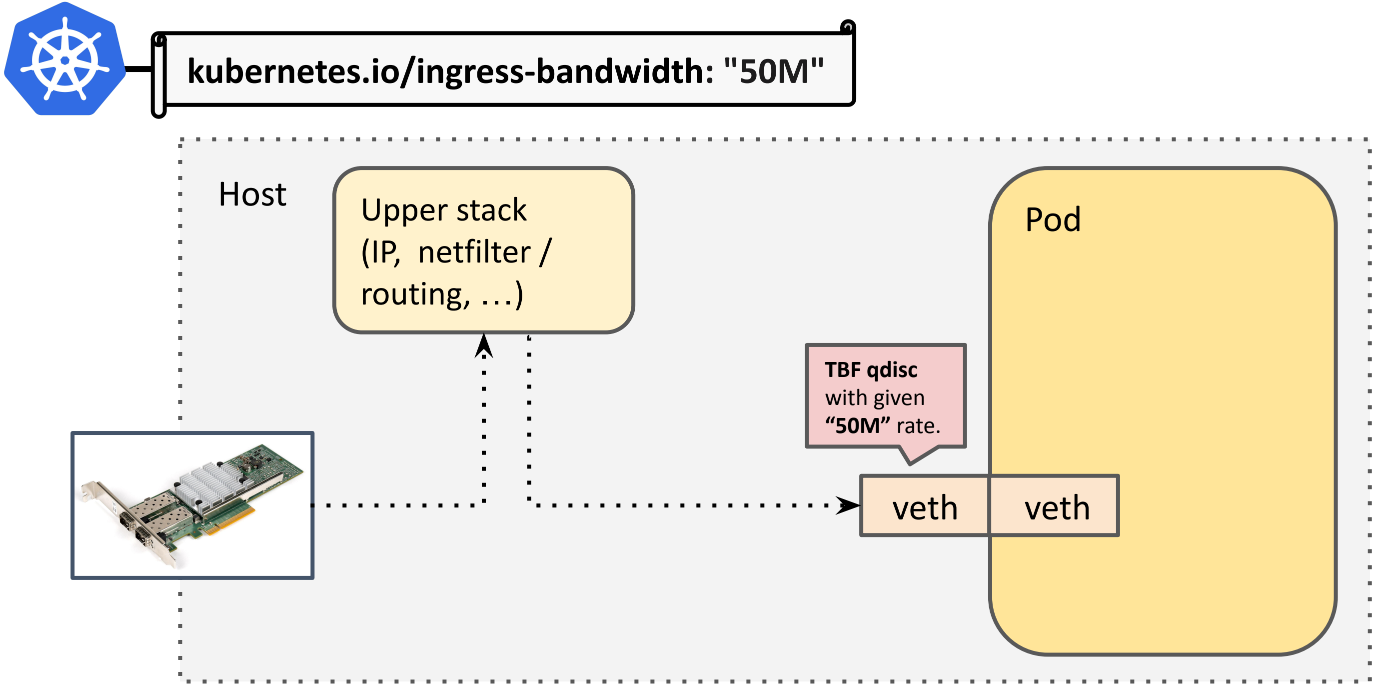
Problem Statement

➔ bandwidth meta plugin not scalable for production use
➔ TBFs are attached to the Pod's veth devices



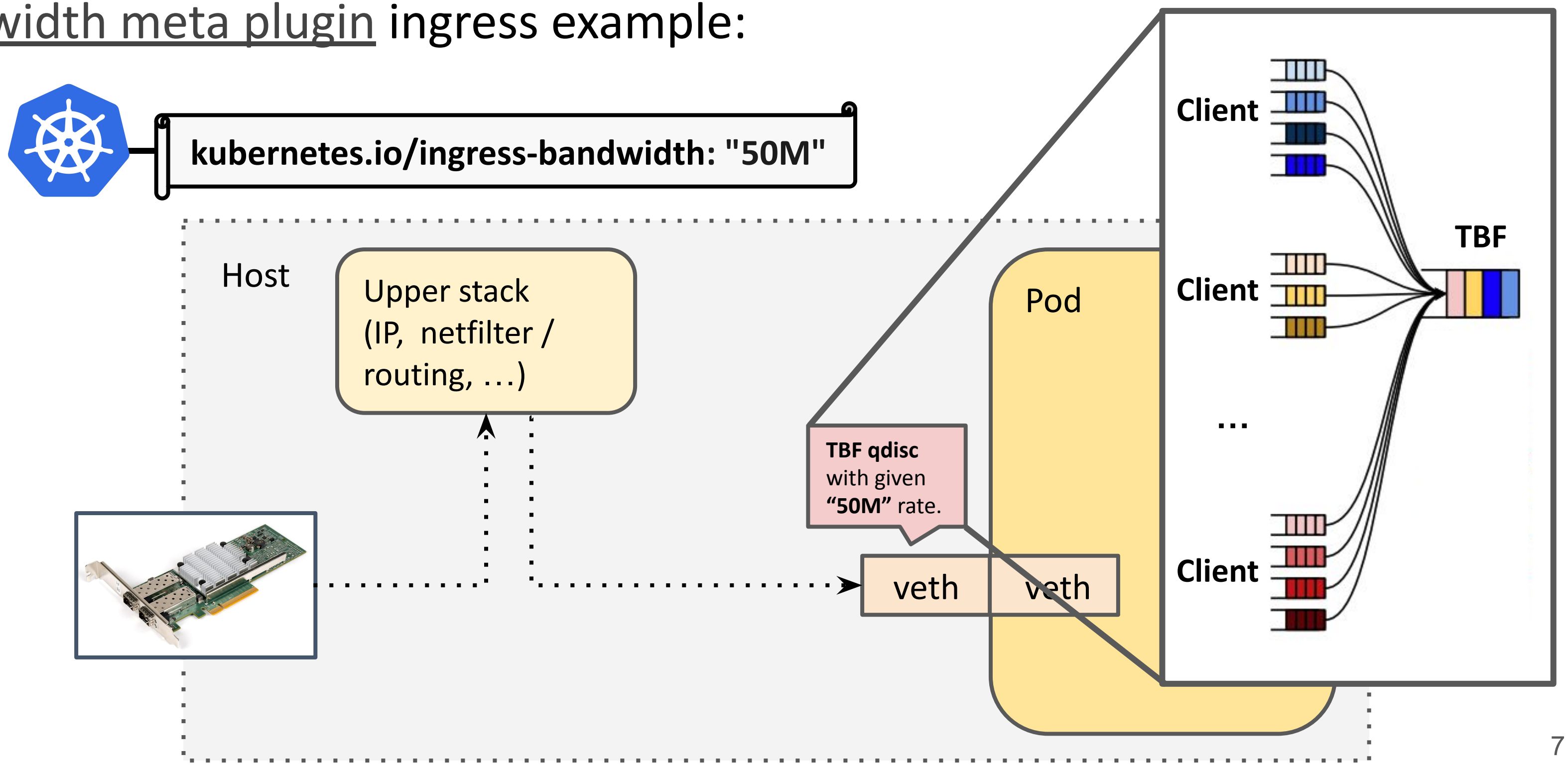
Problem Statement

➔ bandwidth meta plugin ingress example:



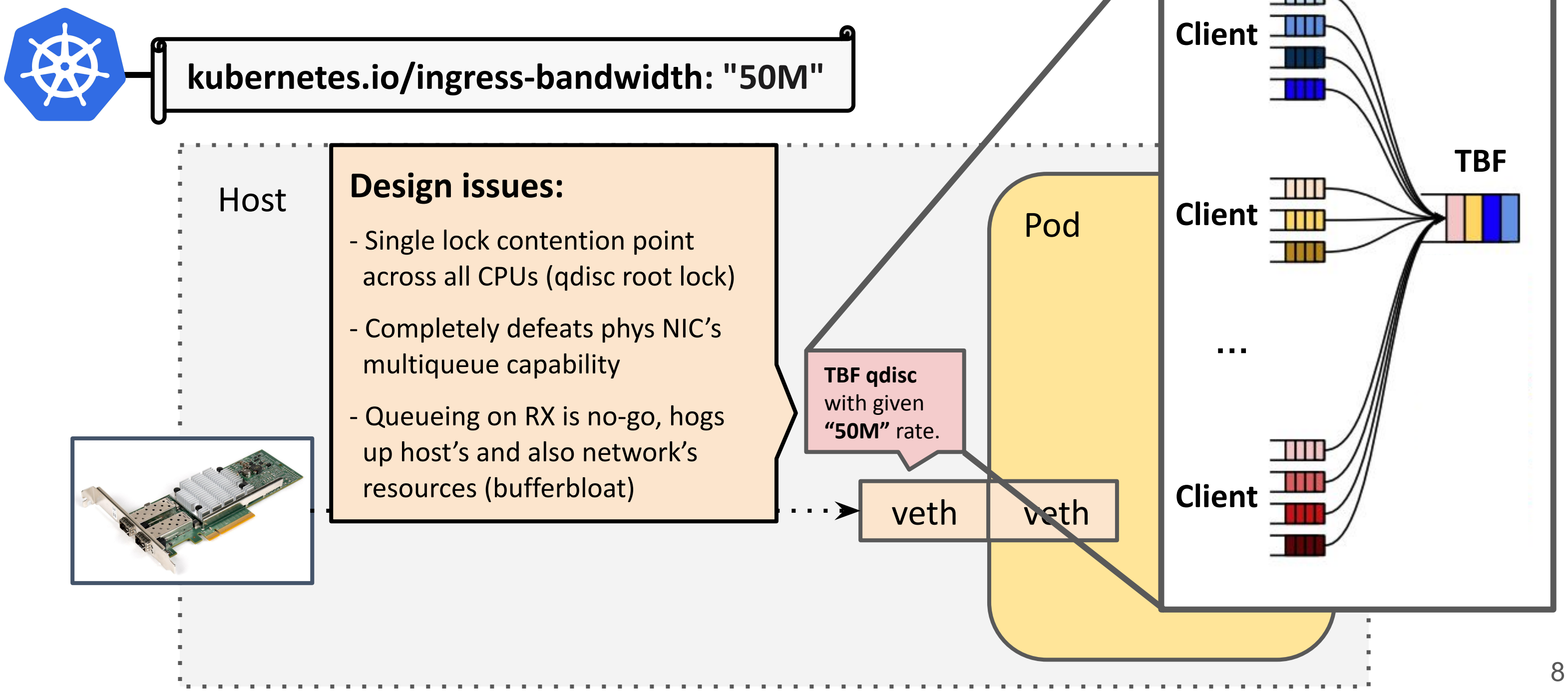
Problem Statement

➔ bandwidth meta plugin ingress example:



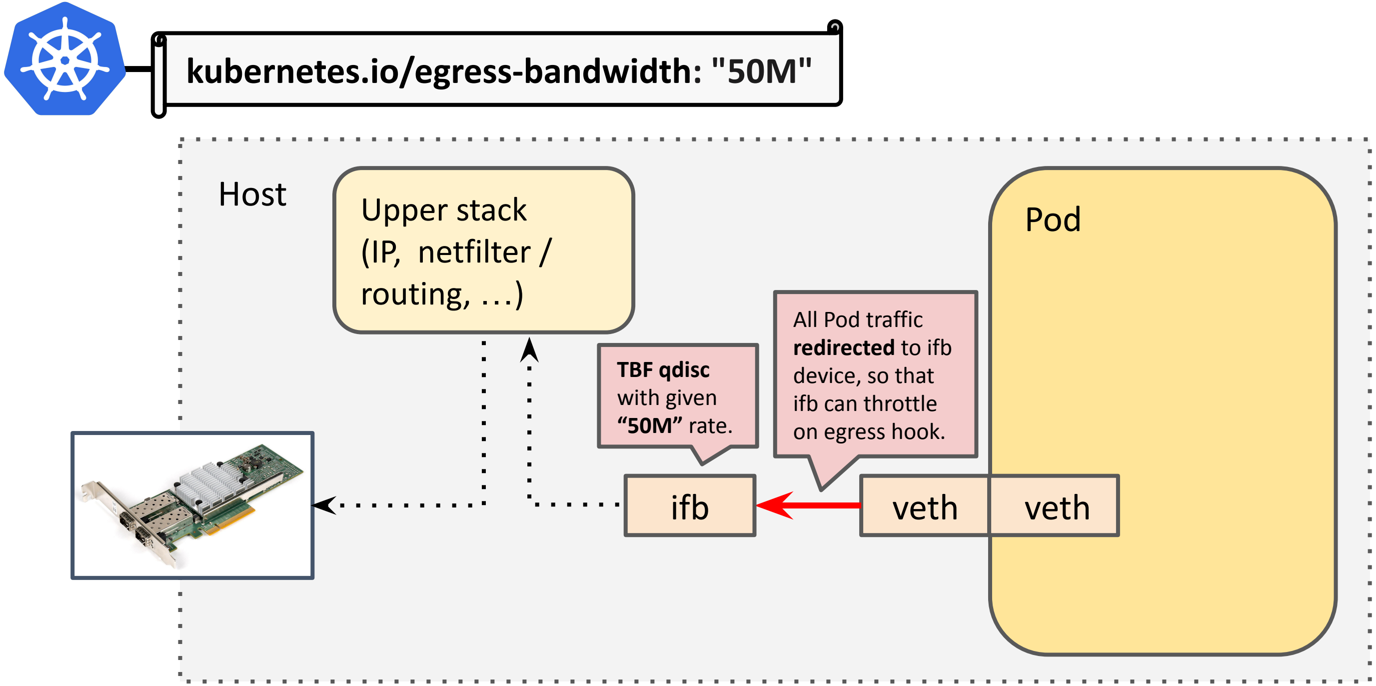
Problem Statement

➔ bandwidth meta plugin ingress example:



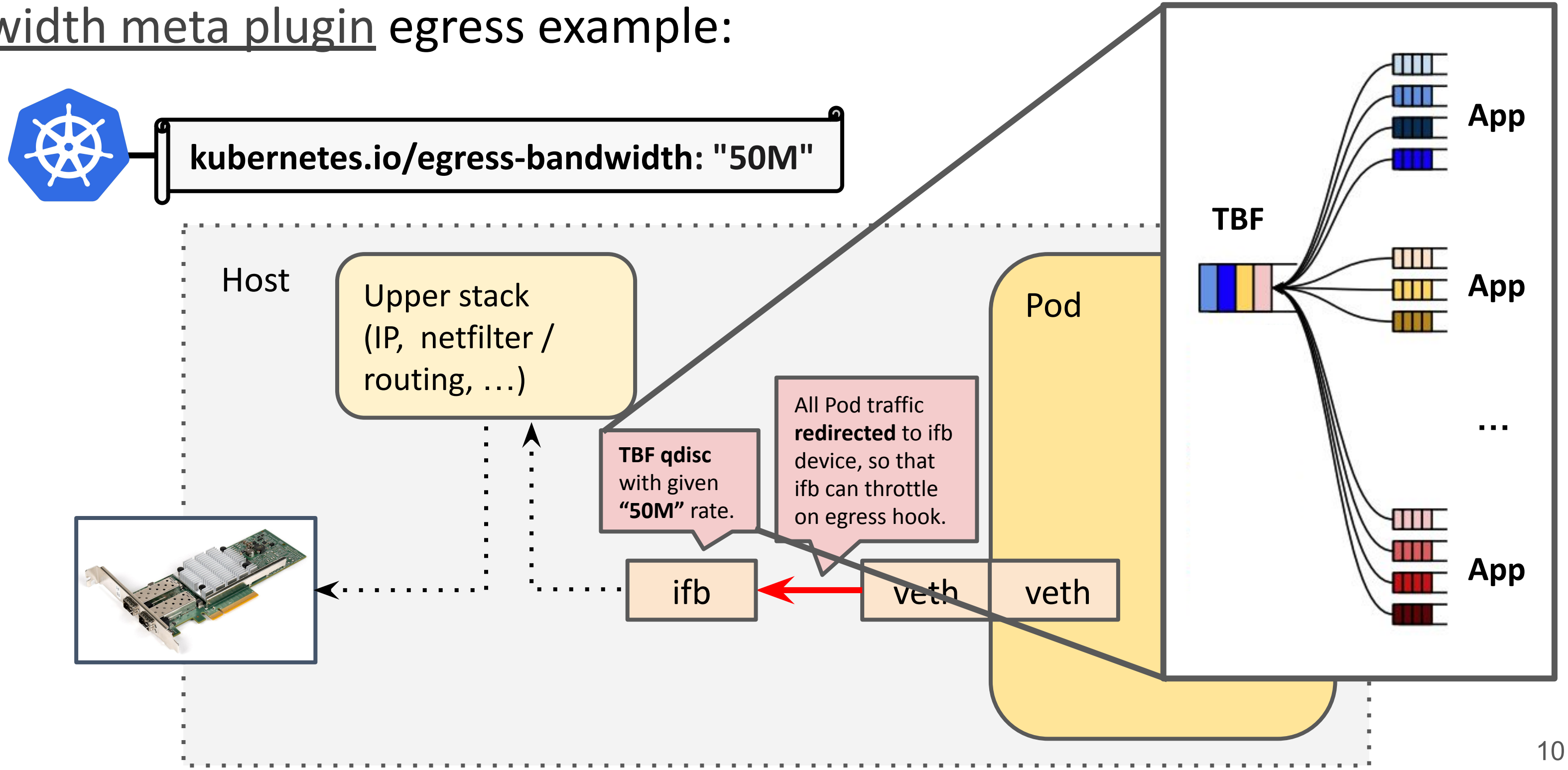
Problem Statement

➔ bandwidth meta plugin egress example:



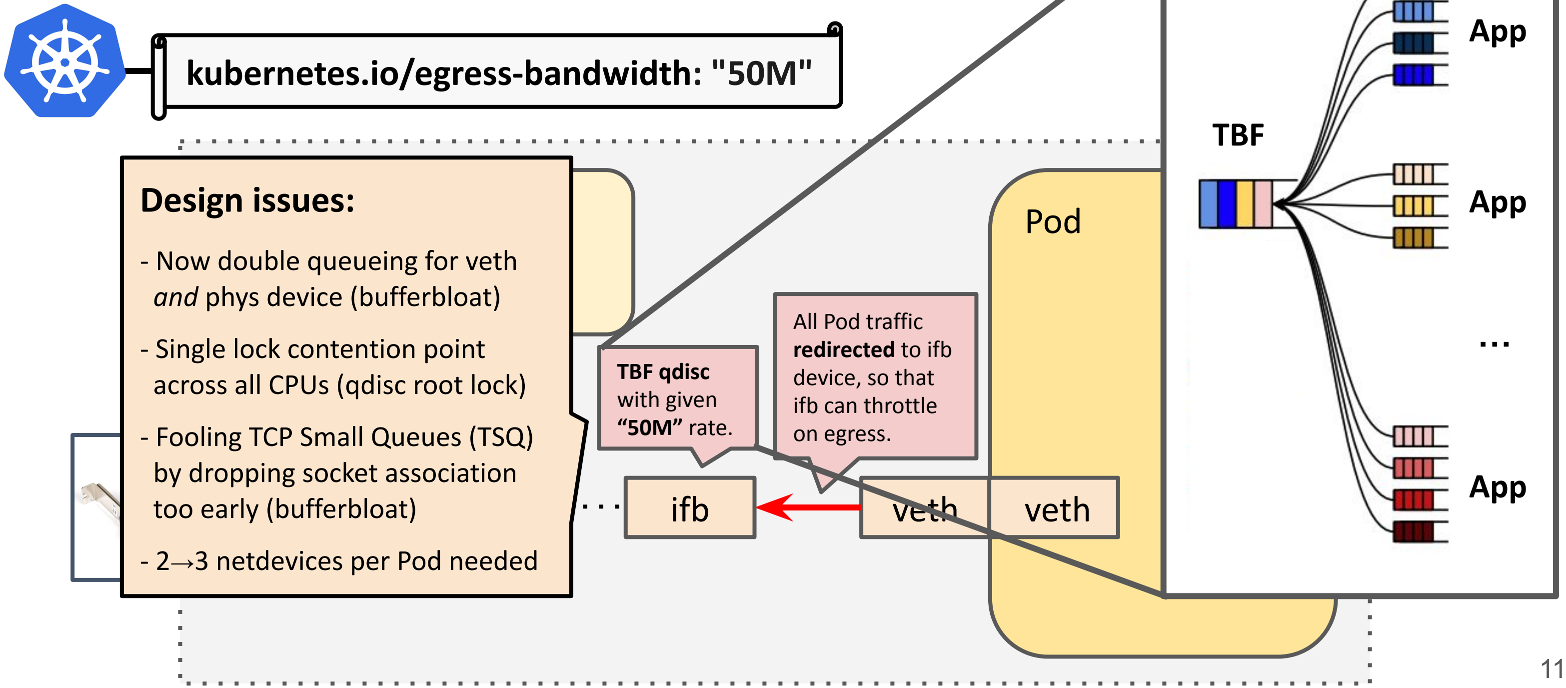
Problem Statement

➔ bandwidth meta plugin egress example:



Problem Statement

➔ bandwidth meta plugin egress example:



Problem Statement

➔ bandwidth meta plugin aka "latency killer"

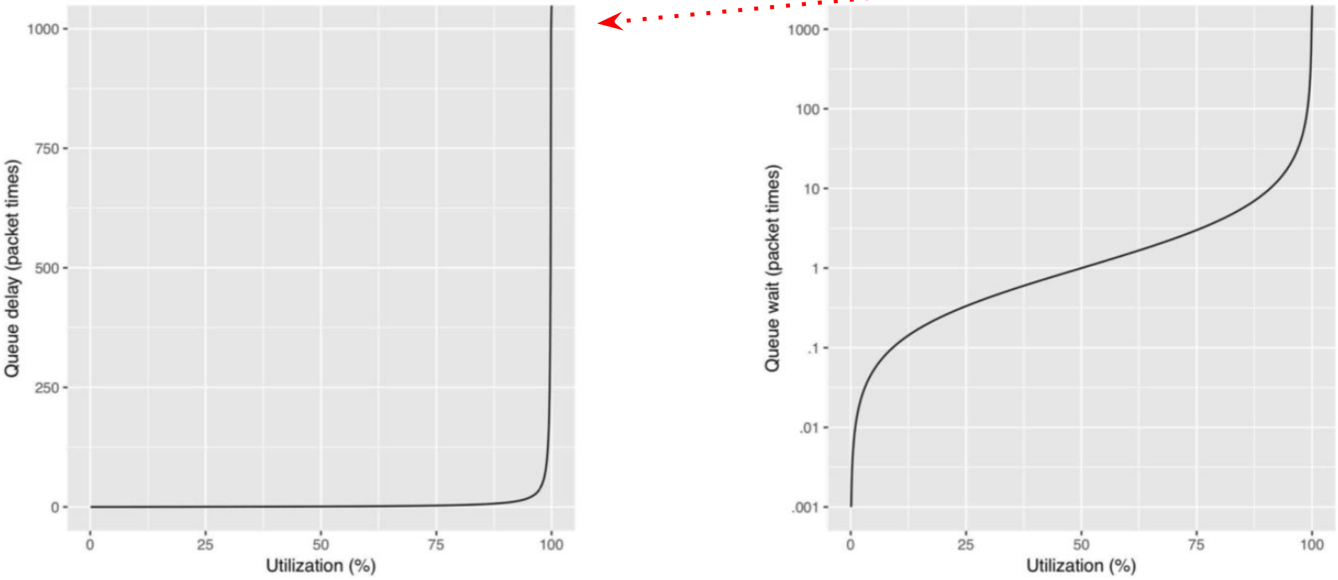
➔ tl;dr summary:

Problem Statement

… but AFAP makes bottleneck run at 100%

Queuing theory says this is fragile. E.g., for M/D/1:

Packet wait-time in queue skyrockets when bottleneck link gets utilized close to 100%.

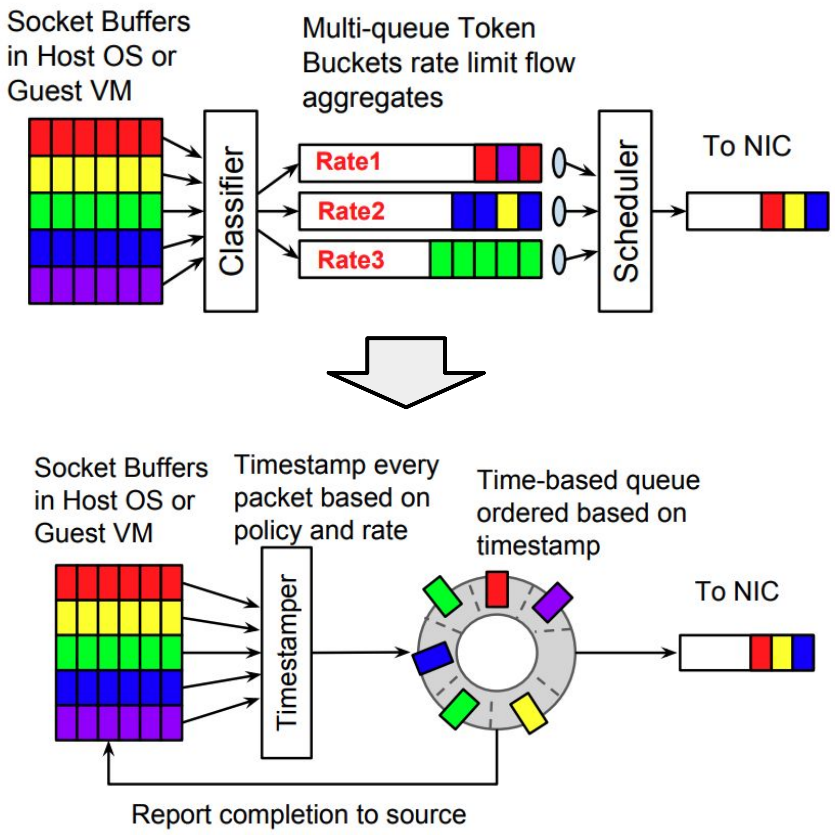


Source: Van Jacobson, “Evolving from AFAP Teaching NICs about time“

From Queues to EDT model

Core Idea to replace queues with two simple pieces:

- ➔ Earliest Departure Time (EDT) time stamp in every packet
- ➔ Timing-wheel scheduler which replaces the queue



Source: A. Saeed et al., "Carousel: Scalable Traffic Shaping at End Hosts" (SIGCOMM'17)

How can the EDT model be applied to Kubernetes?

eBPF

Programmable and **performant** in-kernel "virtual machine" that **safely** executes native code on certain events/hooks (aka "JavaScript for the kernel").

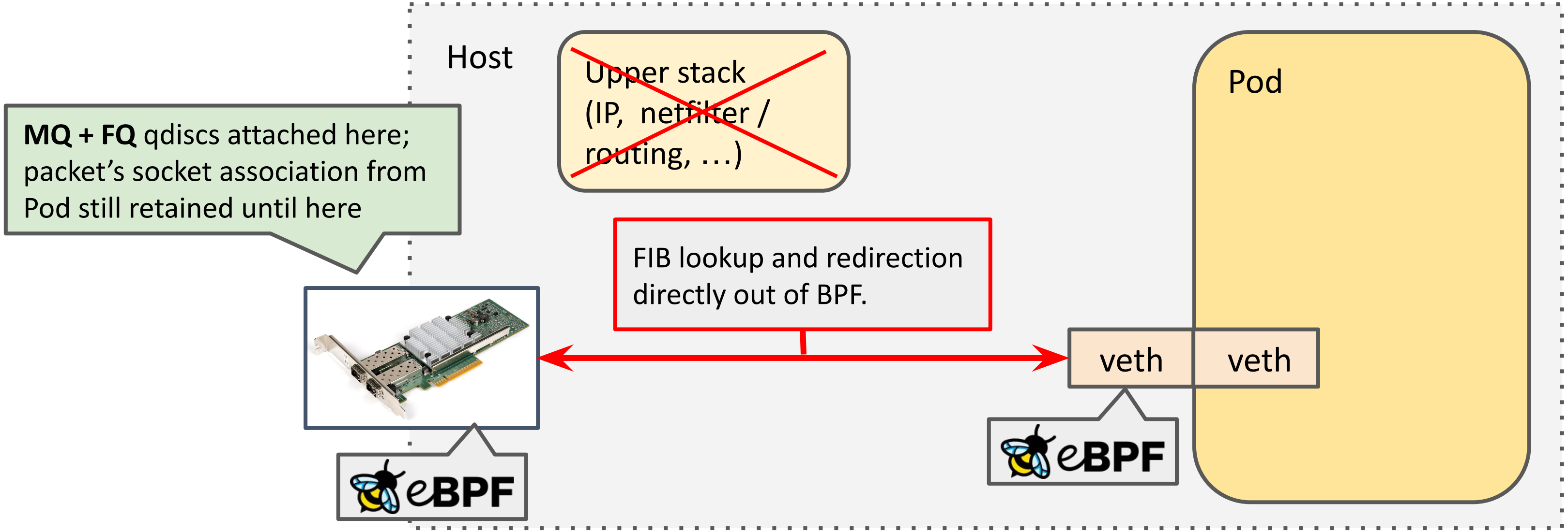
How can the EDT model be applied to Kubernetes?

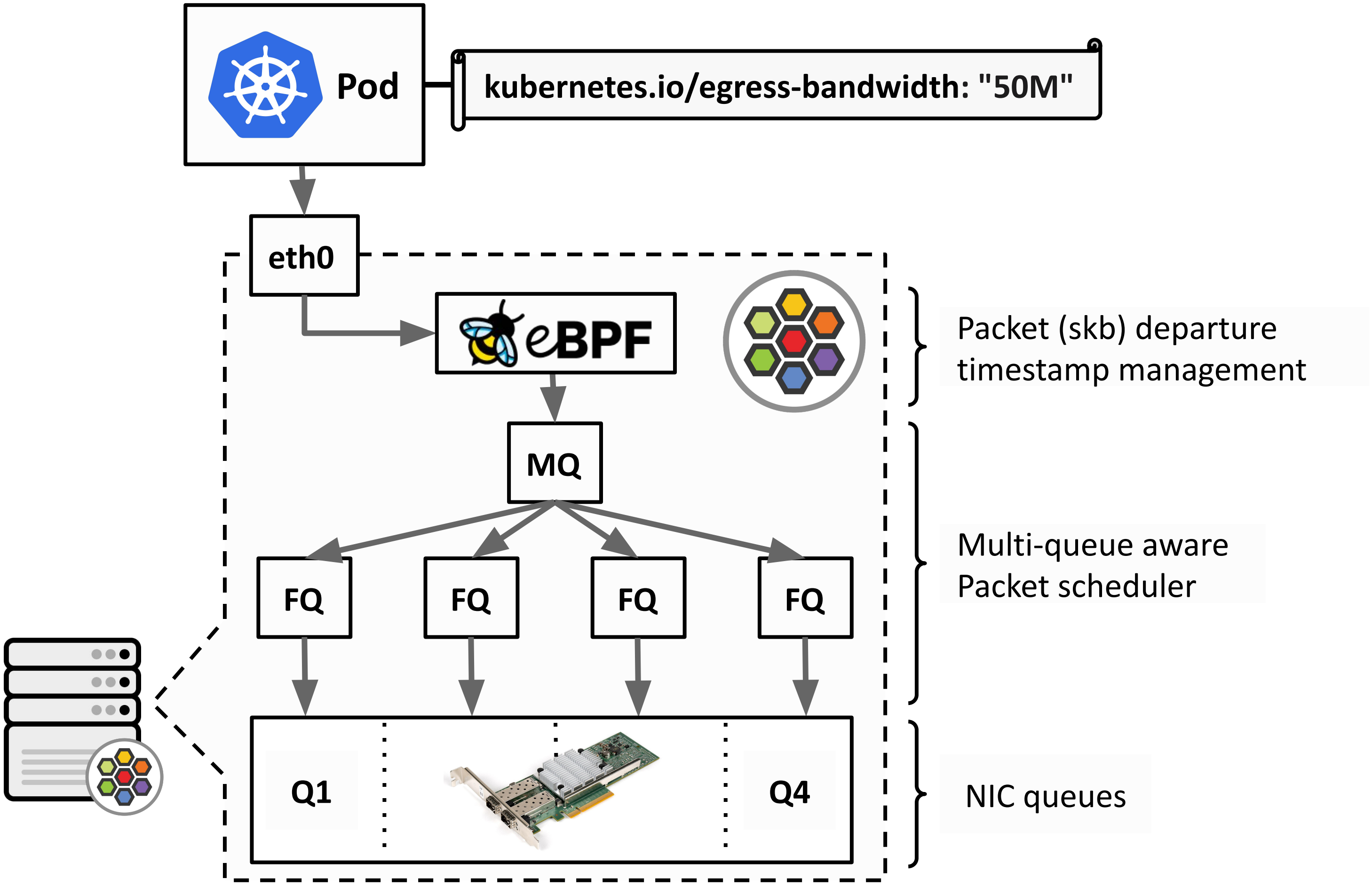
cilium

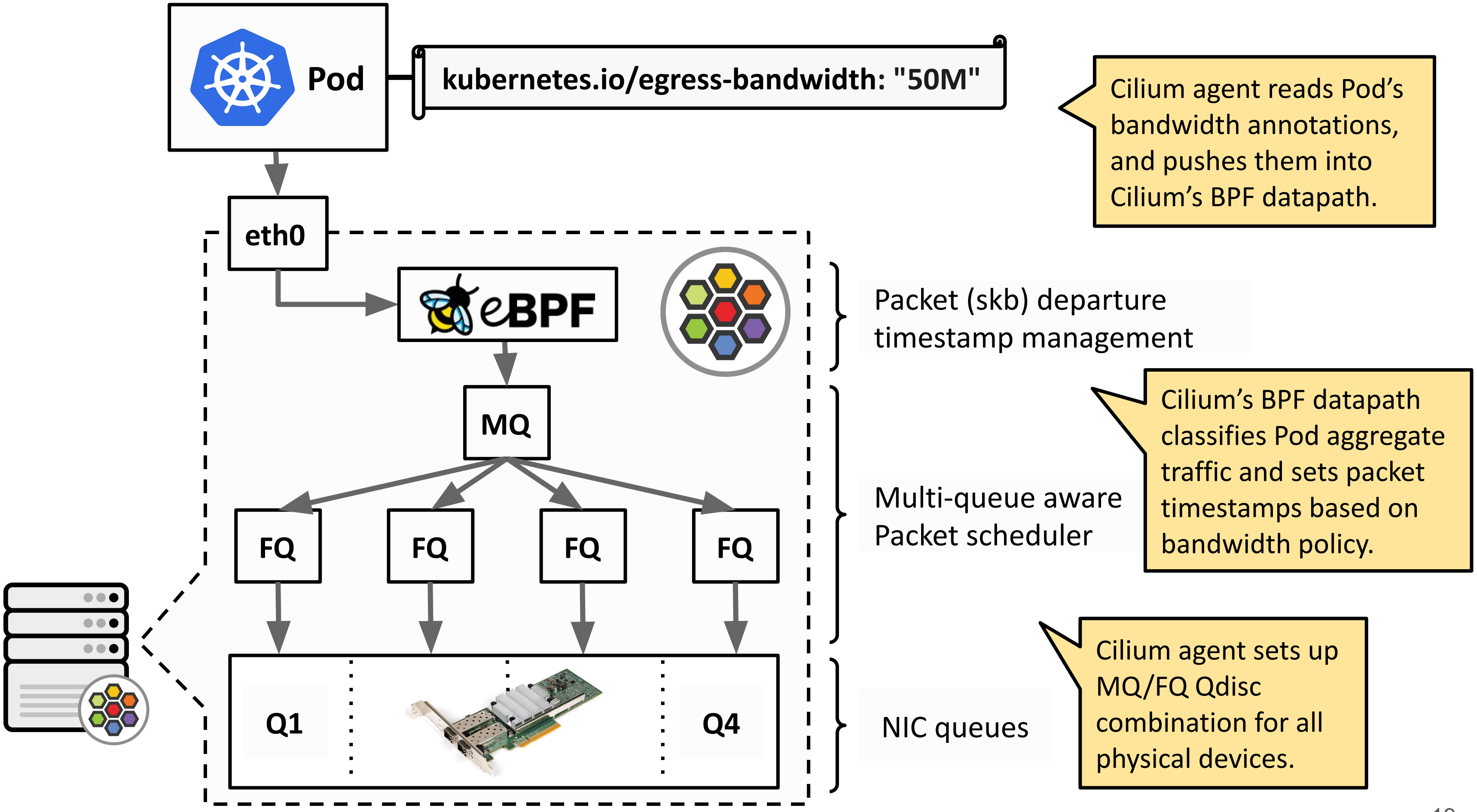
eBPF-based CNI / platform which provides Pod connectivity, service load-balancing, network policies, **bandwidth management**, transparent encryption and more.

Cilium's Bandwidth Manager

- ➔ Implements lock-less, EDT-based Pod rate-limiting with eBPF
- ➔ Enforcement points on phys devices instead of veths to avoid bufferbloat and improve TCP TSQ feedback

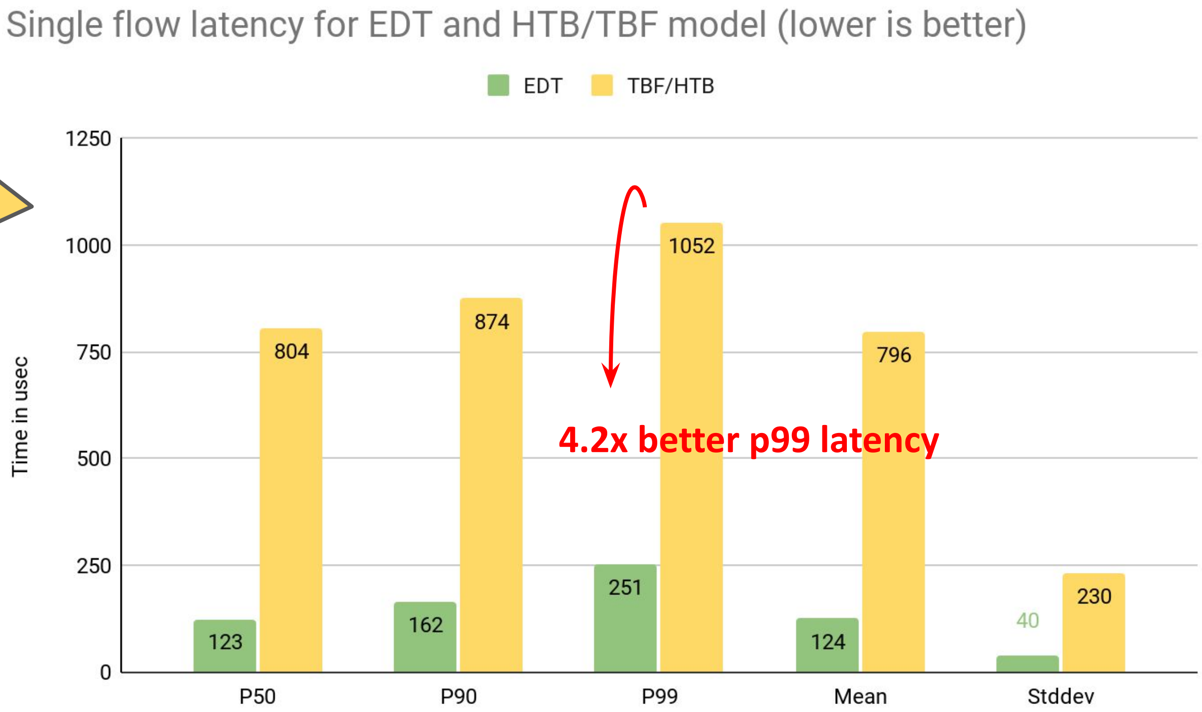



Pod
kubernetes.io/egress-bandwidth: "50M"
eth0
eBPF
MQ
FQ
FQ
FQ
FQ
Q1
Q4
Packet (skb) departure
timestamp management
Multi-queue aware
Packet scheduler
NIC queues


Pod
kubernetes.io/egress-bandwidth: "50M"
Cilium agent reads Pod's bandwidth annotations, and pushes them into Cilium's BPF datapath.
eth0
eBPF
Packet (skb) departure timestamp management
MQ
FQ
FQ
FQ
FQ
Multi-queue aware Packet scheduler
Cilium's BPF datapath classifies Pod aggregate traffic and sets packet timestamps based on bandwidth policy.
Q1
Q4
NIC queues
Cilium agent sets up MQ/FQ Qdisc combination for all physical devices.

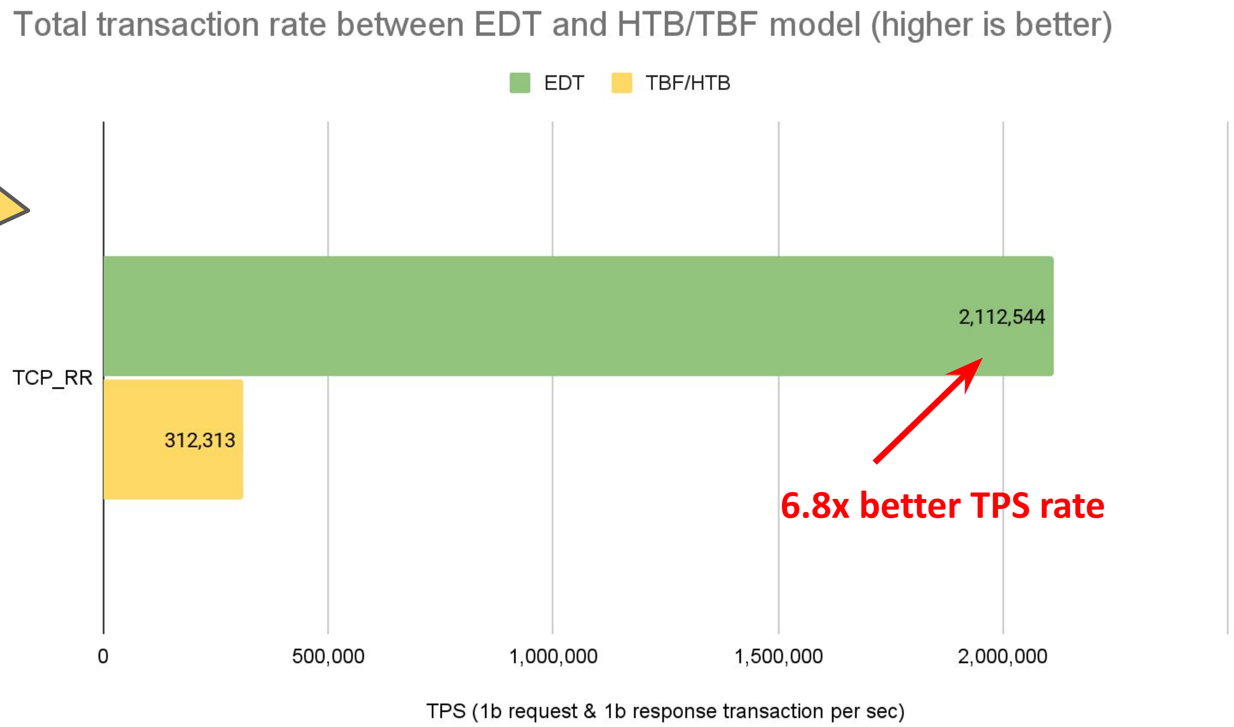
Comparison of Cilium’s EDT implementation vs TBF

Env: 256 concurrent request/response type flows (TCP_RR), 100M rate per flow



Comparison of Cilium’s EDT implementation vs TBF

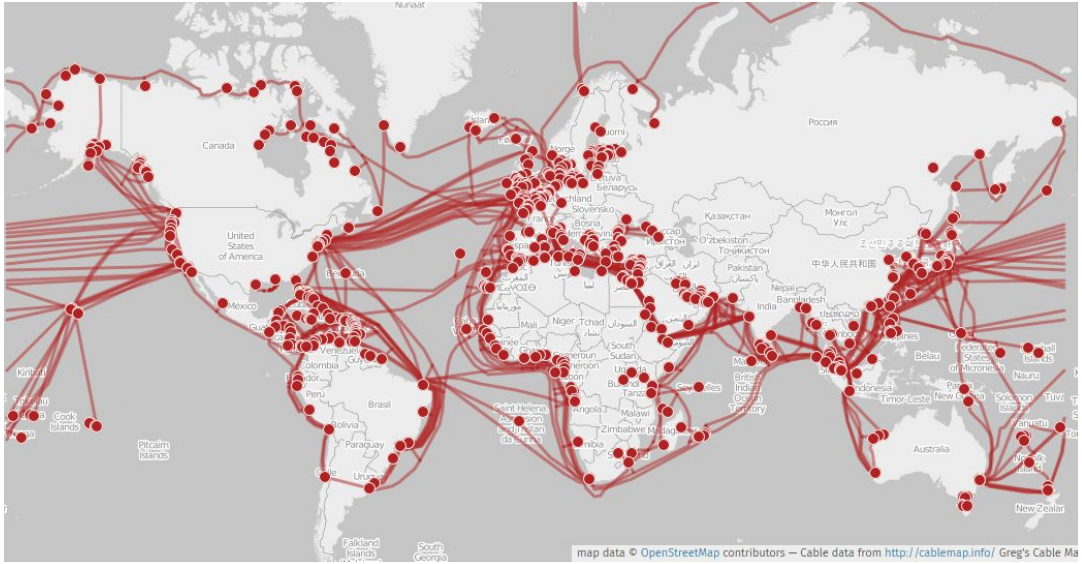
Env: 256 concurrent request/response type flows (TCP_RR), 100M rate per flow



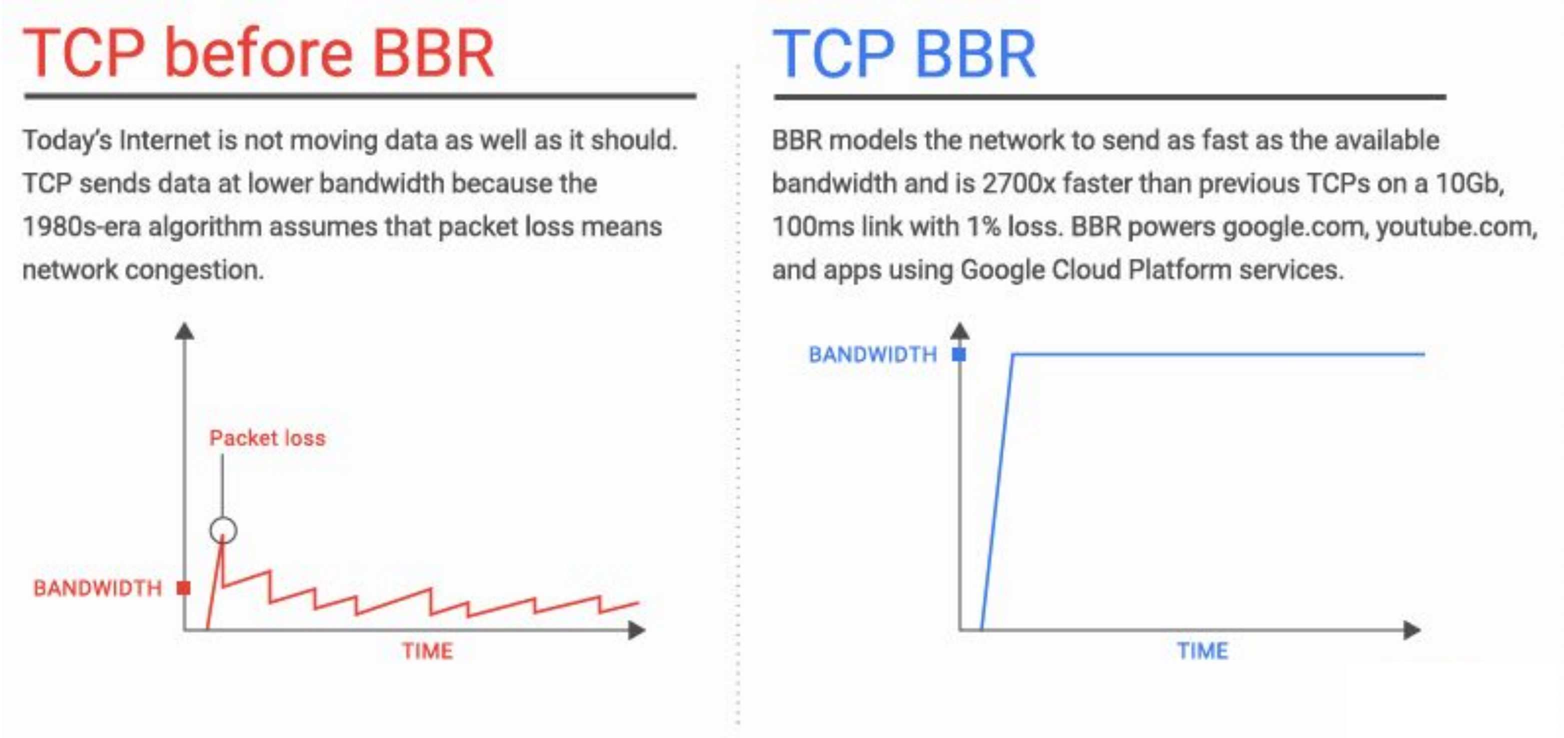
Cilium’s Bandwidth Manager: recap for now

➔ So far: Cilium’s EDT approach allows for scalable bandwidth enforcement

➔ What about more broadly Internet-level bandwidth management?

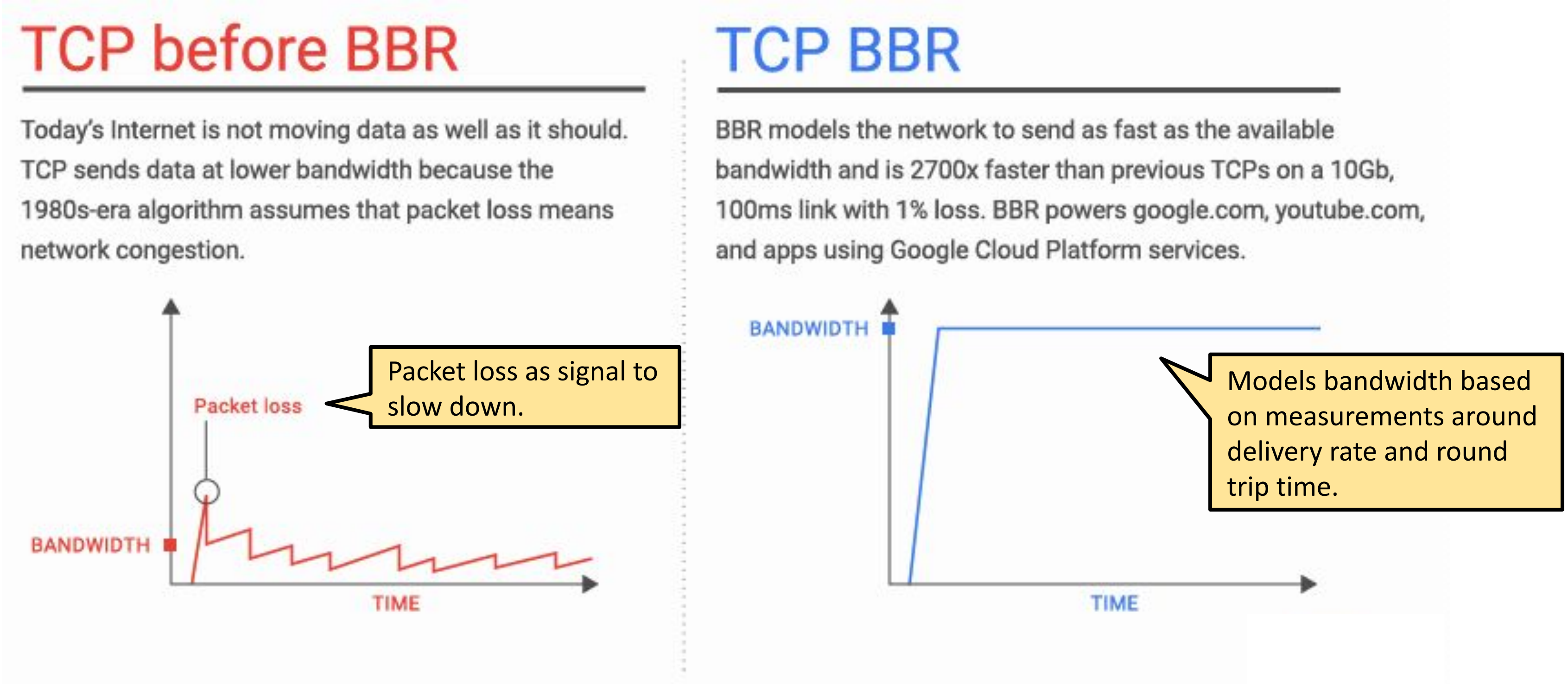


What else does EDT model enable? Enter: BBR



Source: https://cloud.google.com/blog/products/networking/tcp-bbr-congestion-control-comes-to-gcp-your-internet-just-got-faster

What else does EDT model enable? Enter: BBR



Source: https://cloud.google.com/blog/products/networking/tcp-bbr-congestion-control-comes-to-gcp-your-internet-just-got-faster

When is it useful to consider BBR?

- ➔ Kubernetes cluster exposing services to clients over the Internet
  - ◆ Significant latency improvements for low-end last-mile networks
  - ◆ Significant throughput improvements for high-speed long-haul links

	CUBIC (default)	**BBR (v1)**
Model parameters to the state machine	N/A	Throughput, RTT
Loss	Reduce cwnd by 30% on window with any loss	N/A
ECN	RFC3168 (classic ECN)	N/A
Startup	Slow-start until RTT rises (Hystart) or any loss	Slow-start until throughput plateaus

Source: Alexey Ivanov, “Evaluating BBRv2 On the Edge”

Example: **New York** (packet.net) -> **Zurich**

darkstar@linux:~/trees/bpf-... × root@zh-lab-node-1: ~ × darkstar@linux:~/trees/bpf × root@ny-c3-small-x86-01: ~ × root

```
root@zh-lab-node-1:~# while [ 1 ]; do iperf3 -c 147.75.66.15 -t 55 -R -i 5 -O 5 ; done
Connecting to host 147.75.66.15, port 5201
Reverse mode, remote host 147.75.66.15 is sending
[  5] local 192.168.178.91 port 52148 connected to 147.75.66.15 port 5201
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-5.00   sec  78.4 MBytes   132 Mbits/sec                  (omitted)
[  5]   0.00-5.00   sec  98.2 MBytes   165 Mbits/sec
[  5]   5.00-10.00  sec  99.1 MBytes   166 Mbits/sec
[  5]  10.00-15.00  sec   113 MBytes   189 Mbits/sec
[  5]  15.00-20.00  sec   159 MBytes   267 Mbits/sec
[  5]  20.00-25.00  sec   257 MBytes   431 Mbits/sec
[  5]  25.00-30.00  sec   153 MBytes   256 Mbits/sec
[  5]  30.00-35.00  sec   146 MBytes   245 Mbits/sec
[  5]  35.00-40.00  sec   148 MBytes   248 Mbits/sec
[  5]  40.00-45.00  sec   157 MBytes   264 Mbits/sec
[  5]  45.00-50.00  sec   193 MBytes   323 Mbits/sec
[  5]  50.00-55.00  sec   272 MBytes   457 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-55.10  sec  1.76 GBytes   274 Mbits/sec  1501             sender
[  5]   0.00-55.00  sec  1.75 GBytes   274 Mbits/sec                  receiver
```

Bandwidth probing, overreaction to loss!
(Sawtooth pattern nicely visible)

Default, server runs:

- TCP CUBIC
- fq_codel Qdisc

Example: **New York** (packet.net) -> **Zurich**

```
darkstar@linux:~/trees/bpf-...   root@zh-lab-node-1: ~   darkstar@linux:~/trees/bpf   root@ny-c3-small-x86-01: ~   root
root@zh-lab-node-1:~# while [ 1 ]; do iperf3 -c 147.75.66.15 -t 55 -R -i 5 -O 5 ; done
Connecting to host 147.75.66.15, port 5201
Reverse mode, remote host 147.75.66.15 is sending
[  5] local 192.168.178.91 port 52254 connected to 147.75.66.15 port 5201
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-5.00   sec   152 MBytes   254 Mbits/sec                  (omitted)
[  5]   0.00-5.00   sec   258 MBytes   433 Mbits/sec
[  5]   5.00-10.00  sec   240 MBytes   403 Mbits/sec
[  5]  10.00-15.00  sec   255 MBytes   427 Mbits/sec
[  5]  15.00-20.00  sec   247 MBytes   414 Mbits/sec
[  5]  20.00-25.00  sec   255 MBytes   428 Mbits/sec
[  5]  25.00-30.00  sec   255 MBytes   428 Mbits/sec
[  5]  30.00-35.00  sec   238 MBytes   400 Mbits/sec
[  5]  35.00-40.00  sec   255 MBytes   428 Mbits/sec
[  5]  40.00-45.00  sec   239 MBytes   401 Mbits/sec
[  5]  45.00-50.00  sec   253 MBytes   425 Mbits/sec
[  5]  50.00-55.00  sec   242 MBytes   407 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-55.10  sec  2.68 GBytes   418 Mbits/sec  58812             sender
[  5]   0.00-55.00  sec  2.67 GBytes   418 Mbits/sec                  receiver
```

No overreaction, stable.

Updated, server runs:

- TCP BBR
- FQ Qdisc (for EDT)

(results also reproduce with netperf)

Example: **New York** (packet.net) -> **Zurich**


```
darkstar@linux:~/trees/bpf-...   root@zh-lab-node-1: ~   darkstar@linux:~/trees/bpf   root@ny-c3-small-x86-01: ~   root
root@zh-lab-node-1:~# while [ 1 ]; do iperf3 -c 147.75.66.15 -t 55 -R -i 5 -O 5 ; done
Connecting to host 147.75.66.15, port 5201
Reverse mode, remote host 147.75.66.15 is sending
[  5] local 192.168.178.91 port 52254 connected to 147.75.66.15 port 5201
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-5.00   sec   152 MBytes   254 Mbits/sec                  (omitted)
[  5]   0.00-5.00   sec   258 MBytes   433 Mbits/sec
[  5]   5.00-10.00  sec   240 MBytes   403 Mbits/sec
[  5]  10.00-15.00  sec   255 MBytes   427 Mbits/sec
[  5]  15.00-20.00  sec   247 MBytes   414 Mbits/sec
[  5]  20.00-25.00  sec   255 MBytes   428 Mbits/sec
[  5]  25.00-30.00  sec   255 MBytes   428 Mbits/sec
[  5]  30.00-35.00  sec   238 MBytes   400 Mbits/sec
[  5]  35.00-40.00  sec   255 MBytes   428 Mbits/sec
[  5]  40.00-45.00  sec   239 MBytes   401 Mbits/sec
[  5]  45.00-50.00  sec   253 MBytes   425 Mbits/sec
[  5]  50.00-55.00  sec   242 MBytes   407 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-55.10  sec  2.68 GBytes   418 Mbits/sec  58812
[  5]   0.00-55.00  sec  2.67 GBytes   418 Mbits/sec
```

No overreaction, stable.

Updated, server runs:

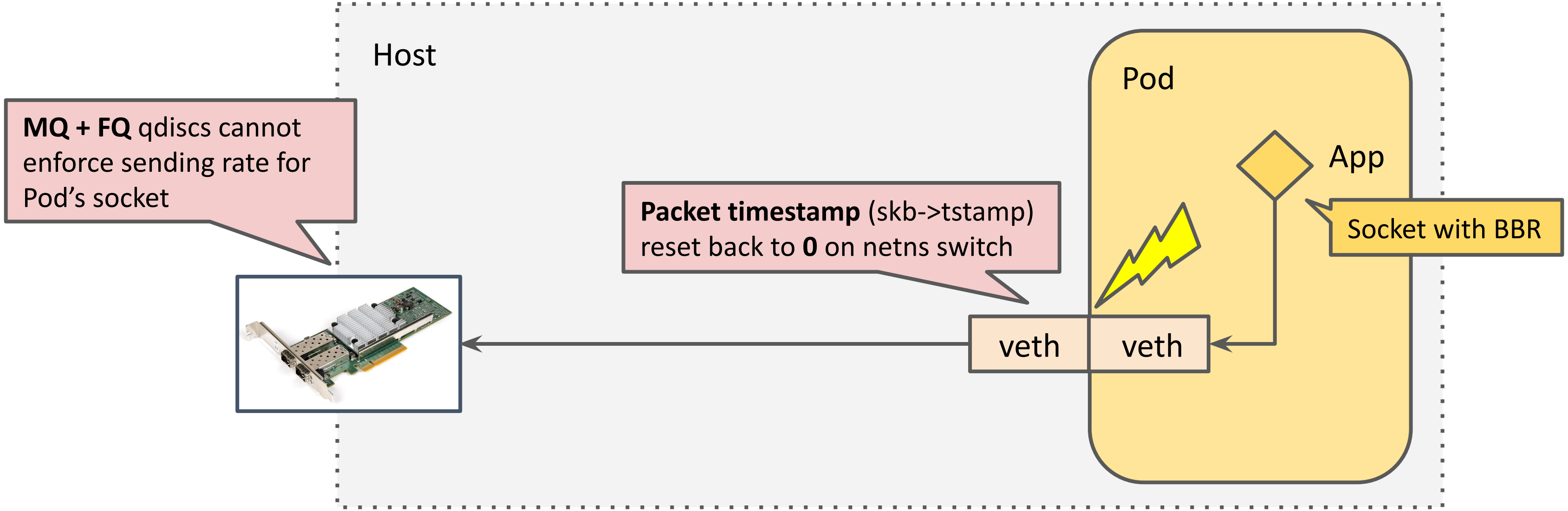
- TCP BBR
- FQ Qdisc (for EDT)

CUBIC ⟶ **BBR** bumps:

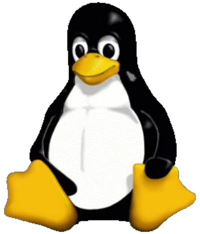
274 ⟶ **418 Mbit/s**

Can BBR be used for Kubernetes Pods?

- ➔ BBR works in conjunction with FQ and sets packet delivery timestamps
- ➔ Kernel clears timestamp for packets leaving Pods (== netns)
  - ◆ Usage of BBR for Pods not possible/broken in general today



Rationale on today's timestamp reset (see our LPC talk)

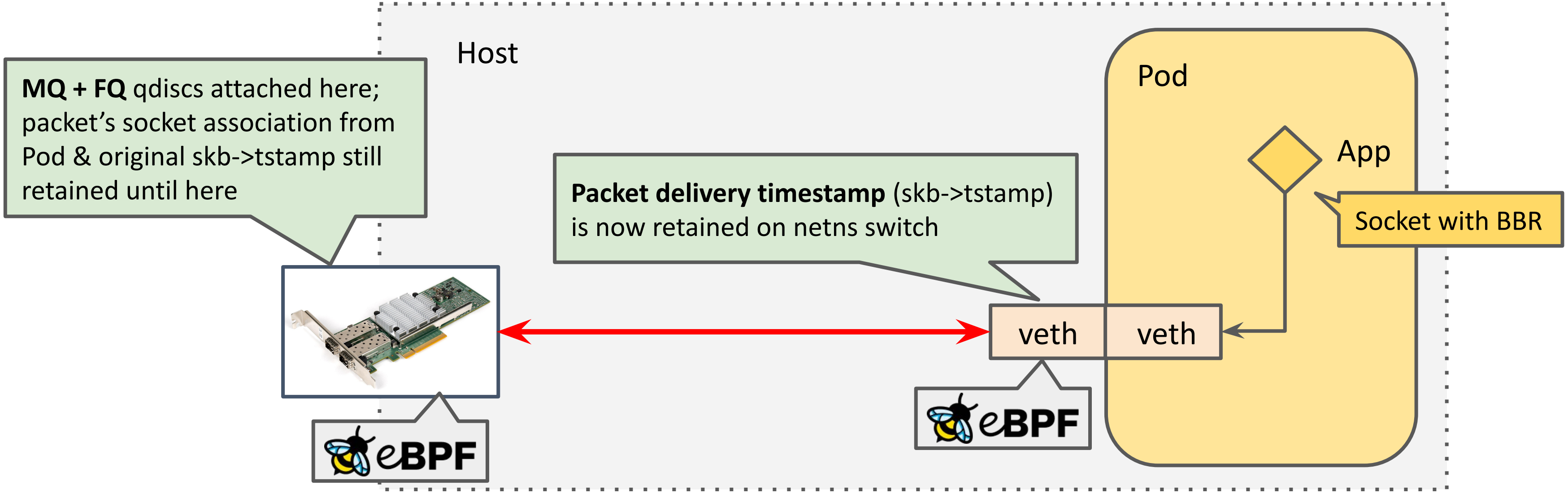


Kernel uses different clock bases for skb->tstamp:

- ➔ Ingress is CLOCK_TAI, egress is CLOCK_MONOTONIC (as is FQ)
- ➔ Forwarding from RX to TX would cause drop in FQ due to overreaching FQ's drop horizon given different clock's offsets
- ➔ No means to figure out clock base from skb->tstamp, hence reset

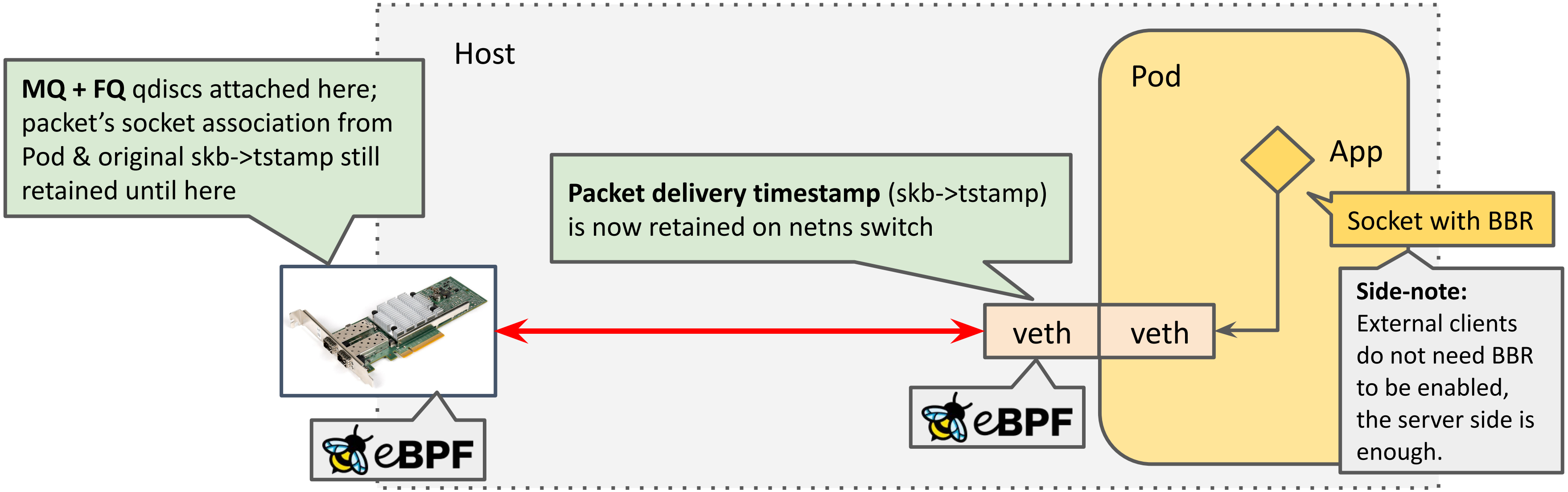
Cilium’s Bandwidth Manager: BBR

- ➔ We helped fixing networking stack in Linux v5.18+ to retain timestamps
- ➔ Bandwidth Manager plumbs the appropriate underlying infrastructure
  - Receives new knob for switching whole cluster over to BBR by default



Cilium’s Bandwidth Manager: BBR

- ➔ We helped fixing networking stack in Linux v5.18+ to retain timestamps
- ➔ Bandwidth Manager plumbs the appropriate underlying infrastructure
  - Receives new knob for switching whole cluster over to BBR by default

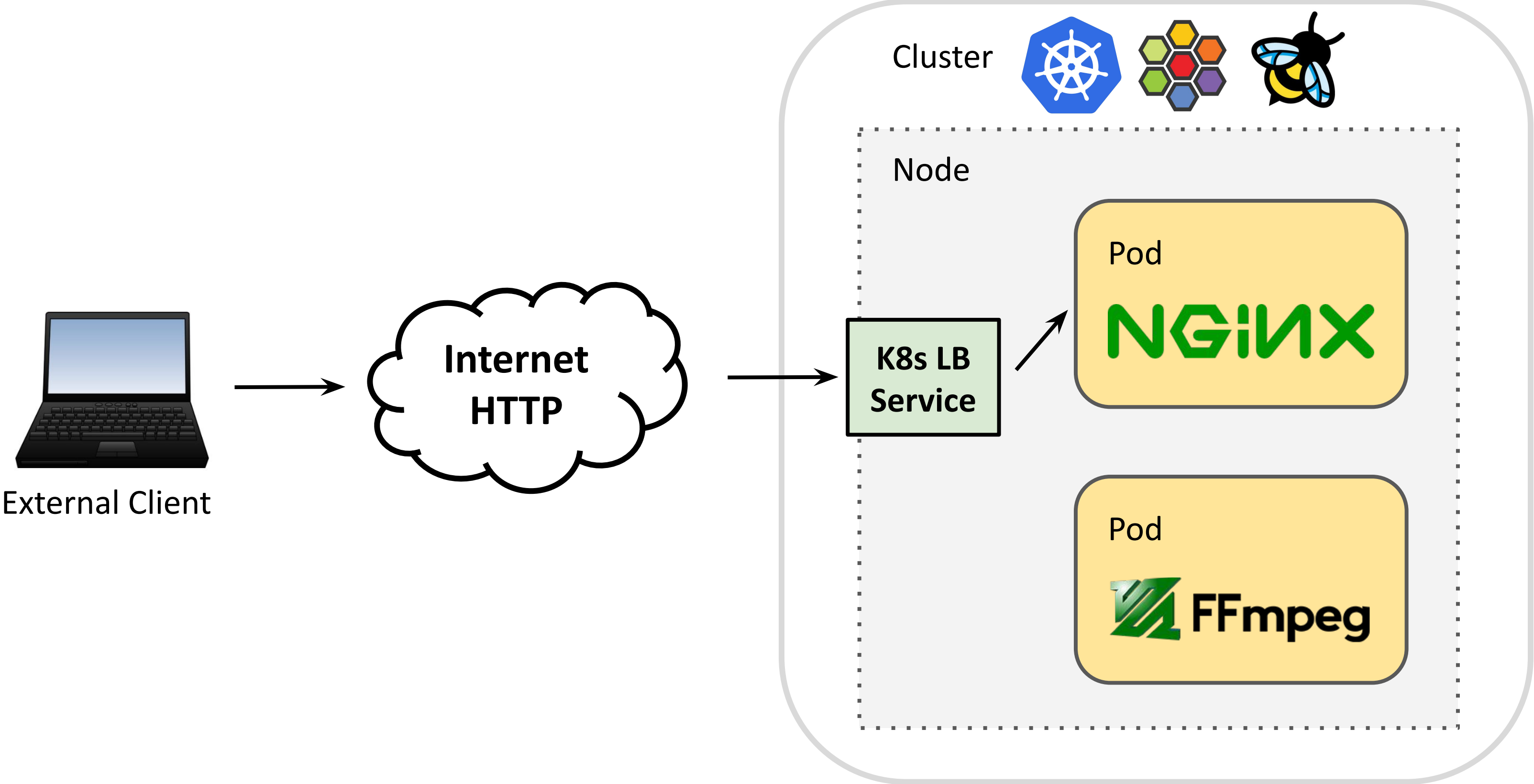


ISOVALENT

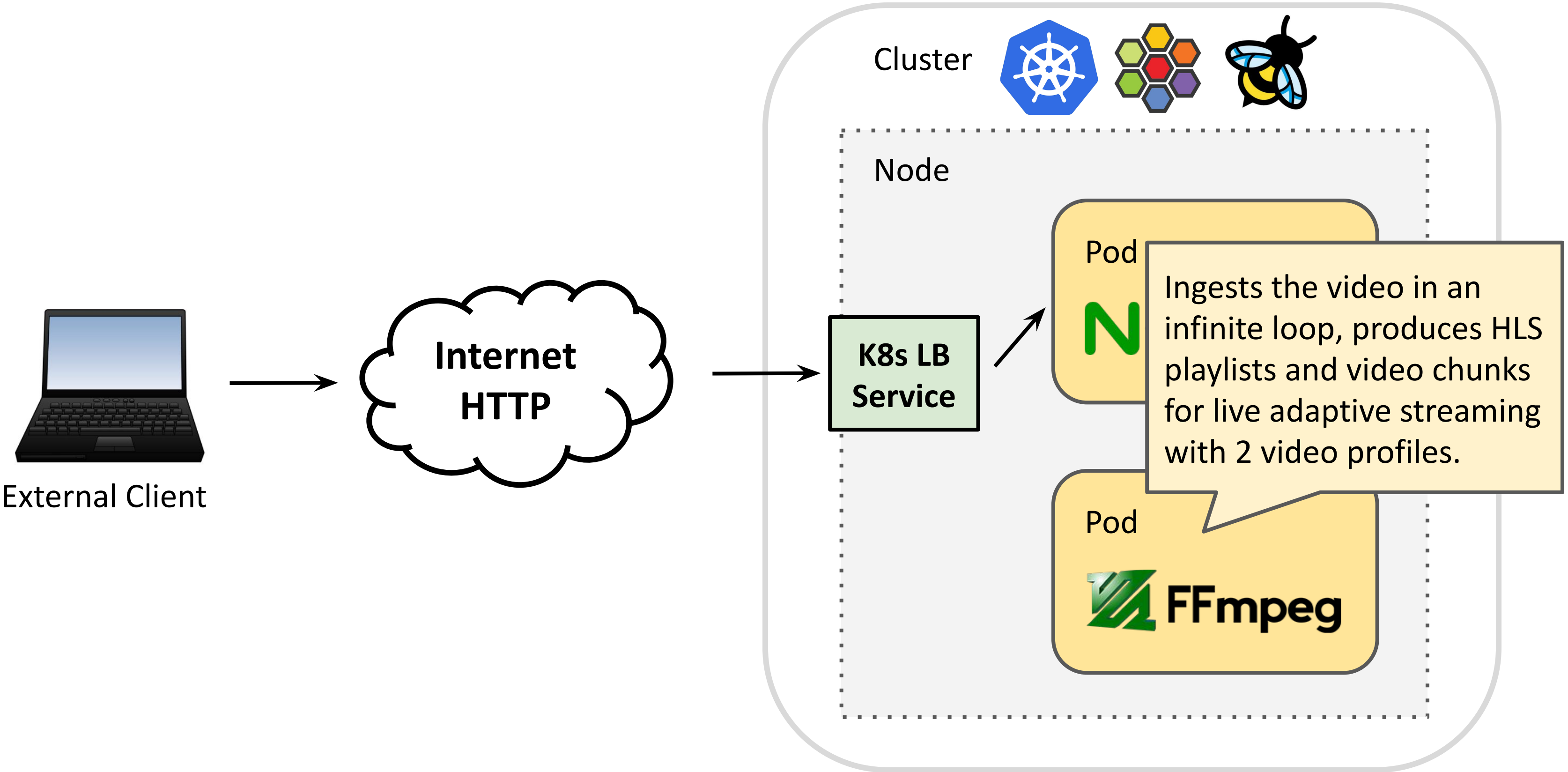
Demo: BBR for Pods

(K8s/Cilium-backed video streaming service: CUBIC versus BBR)

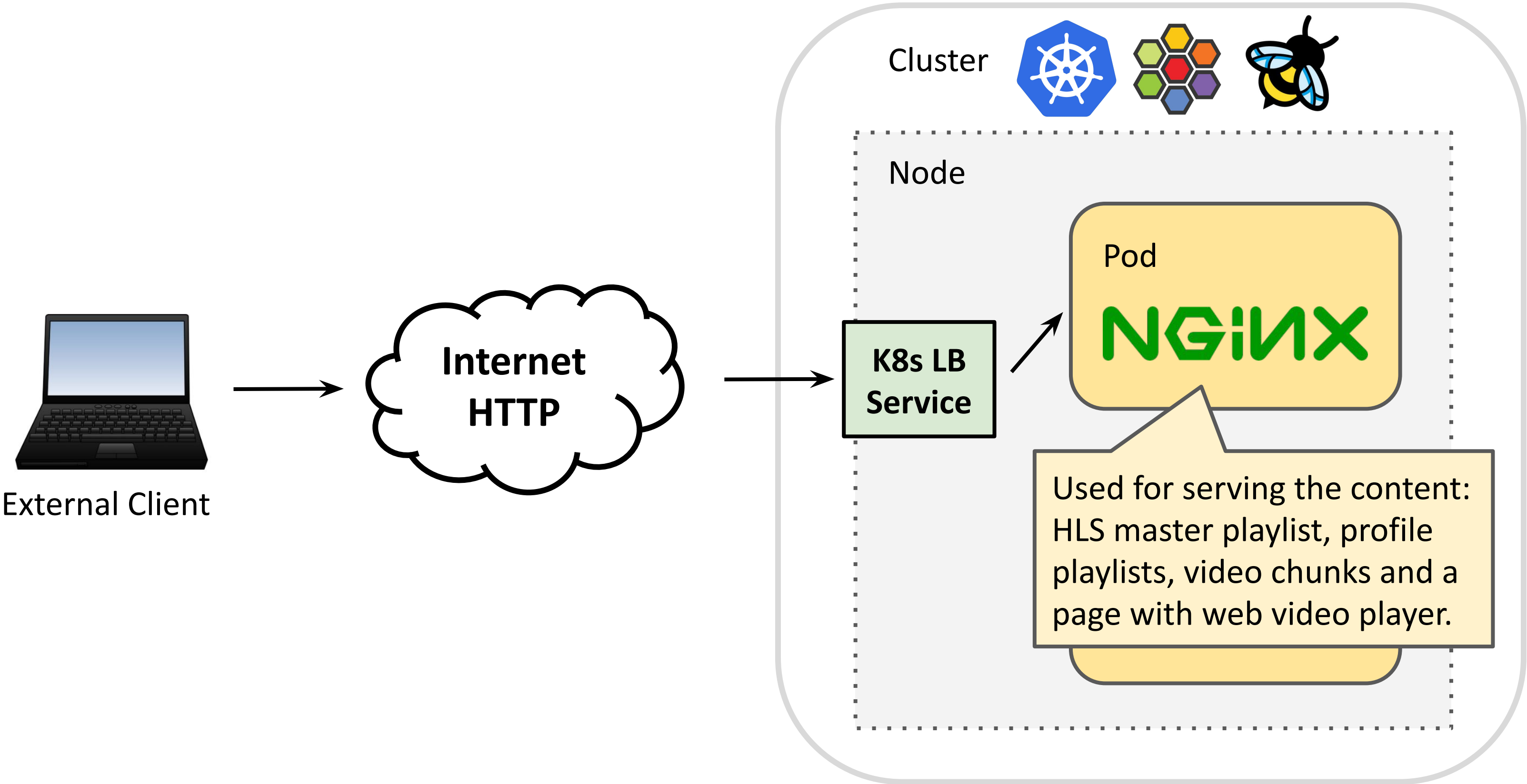
Video Streaming Service Demo Setup: CUBIC vs BBR



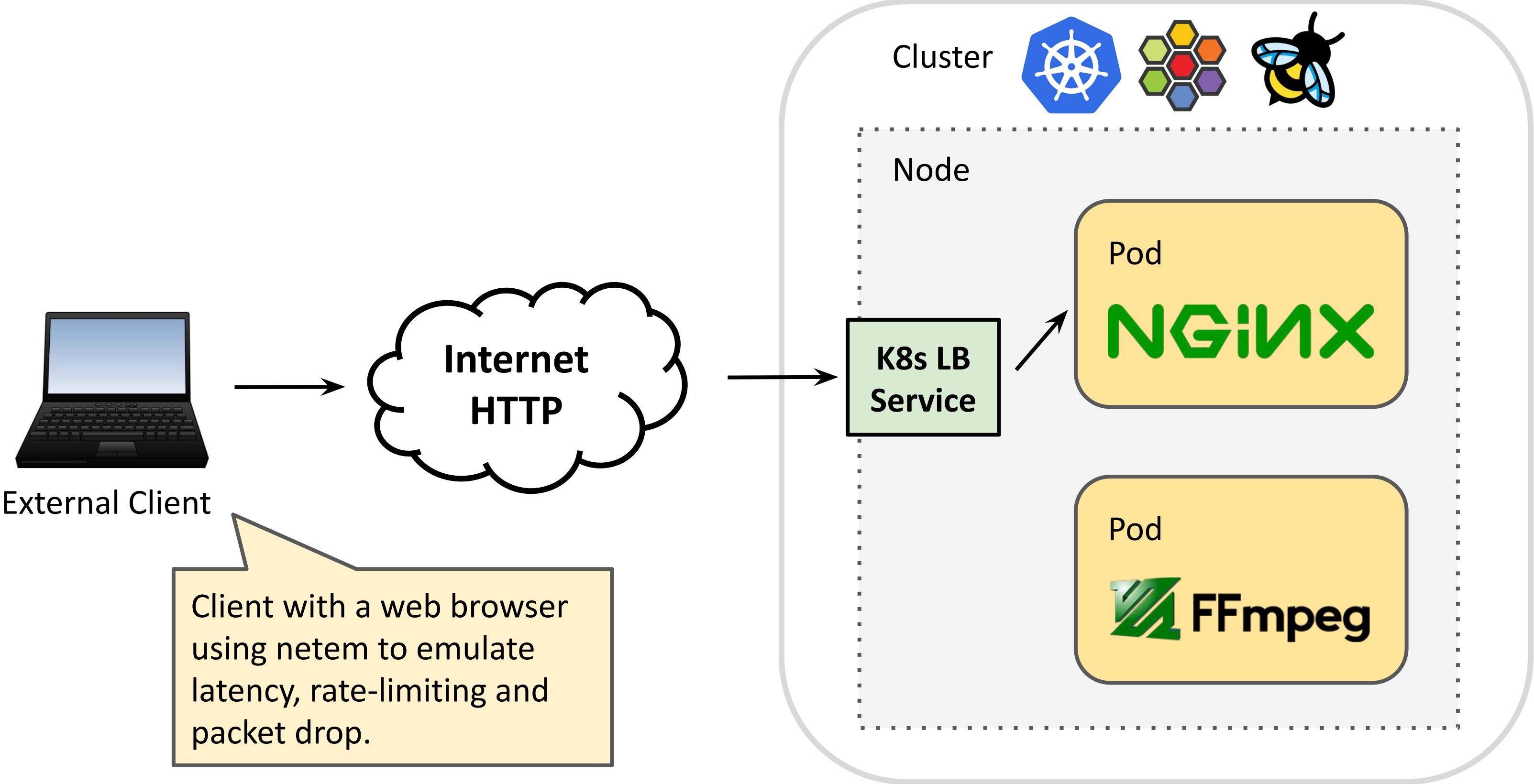
Video Streaming Service Demo Setup: CUBIC vs BBR



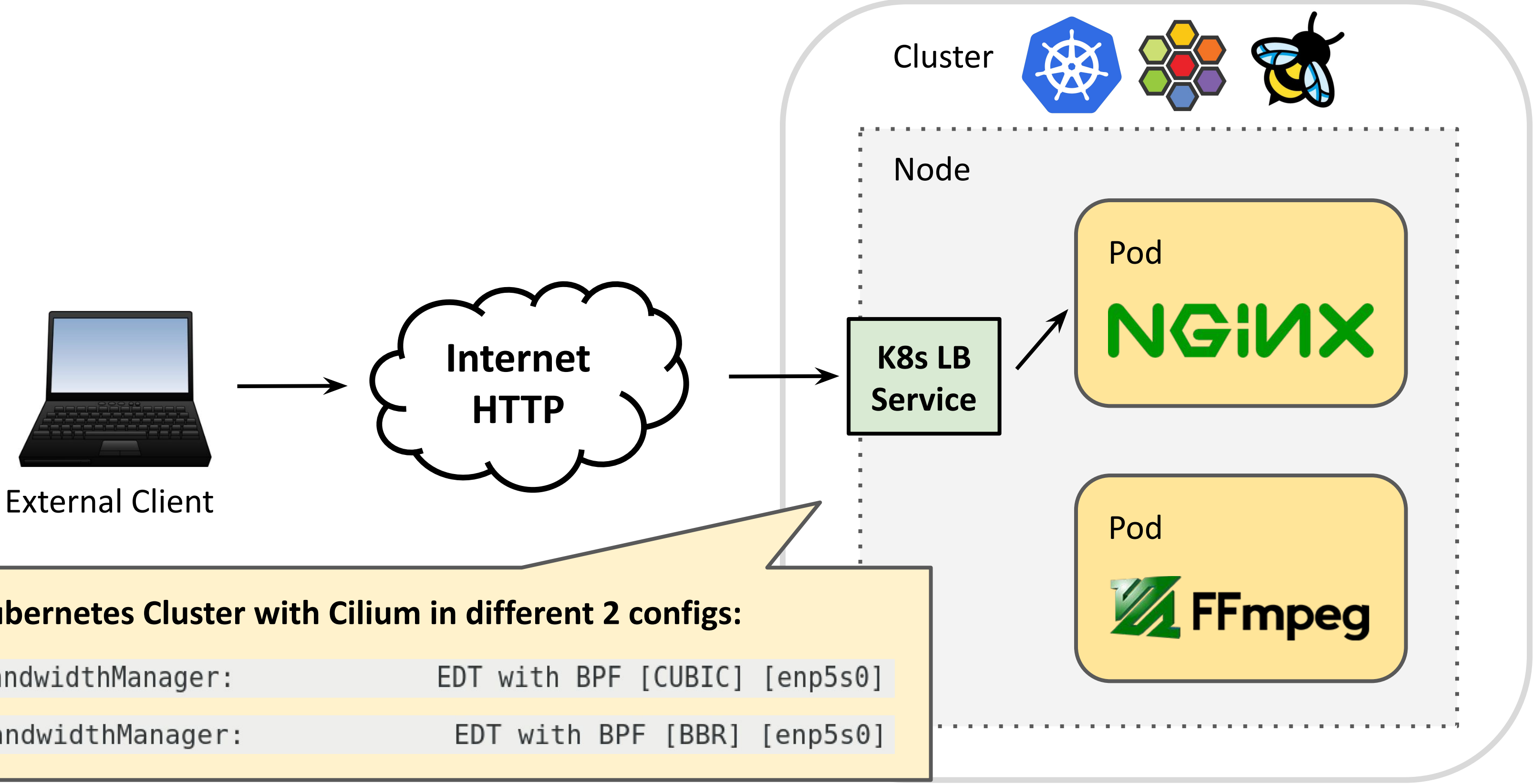
Video Streaming Service Demo Setup: CUBIC vs BBR



Video Streaming Service Demo Setup: CUBIC vs BBR



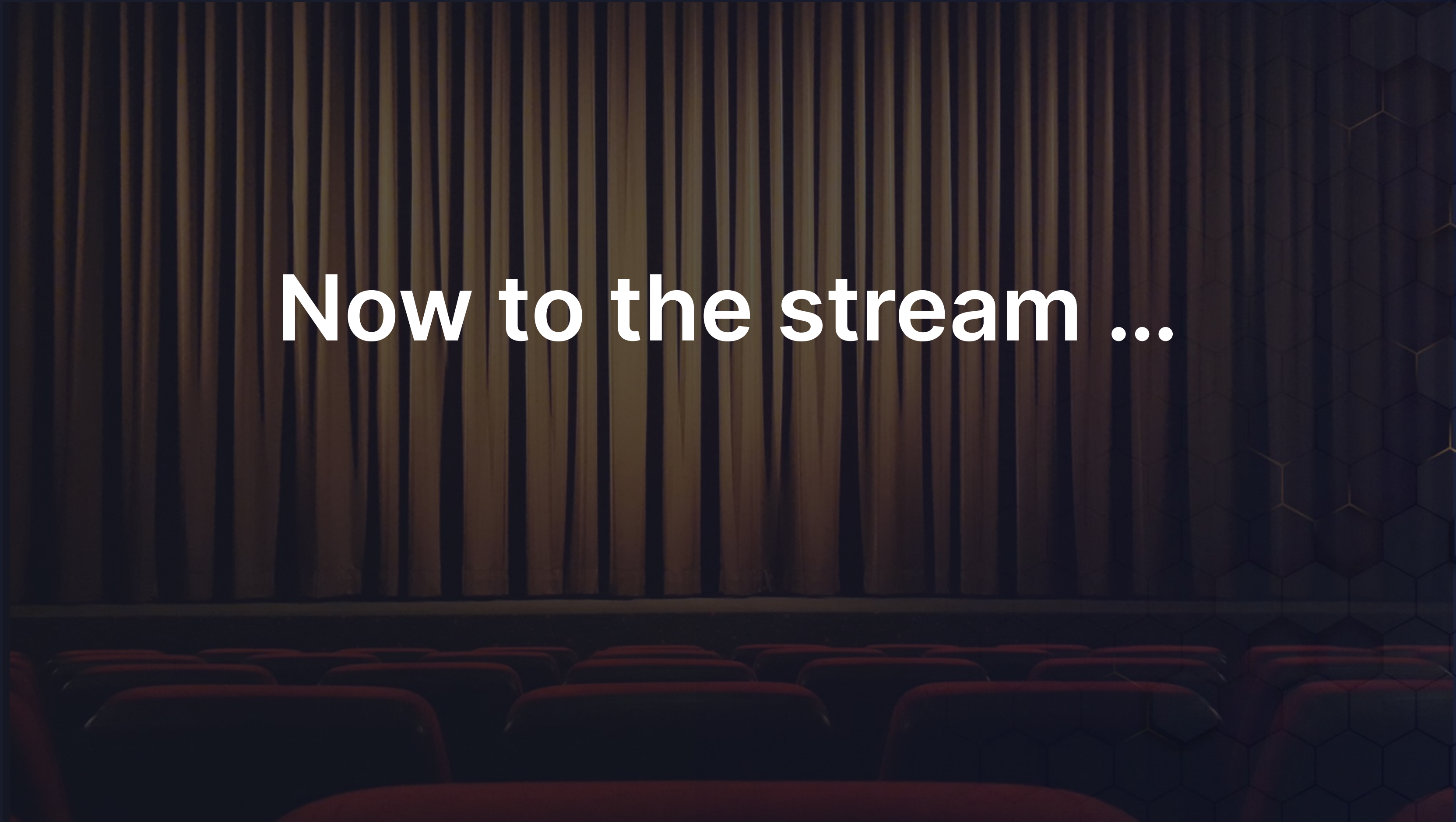
Video Streaming Service Demo Setup: CUBIC vs BBR



Kubernetes Cluster with Cilium in different 2 configs:

```
BandwidthManager:          EDT with BPF [CUBIC] [enp5s0]
BandwidthManager:          EDT with BPF [BBR] [enp5s0]
```

ISOVALENT



Now to the stream …

What needs to be considered with use of BBR?

- ➔ BBR has potential unfairness issues towards CUBIC when env uses both
- ➔ BBR will trigger a higher TCP retransmission rate (more aggressive probing)
- ➔ BBRv2 in the works to overcome them

Source: Alexey Ivanov, "Evaluating BBRv2 On the Edge"

Revisiting earlier Problem Statement

→ Kubernetes bandwidth enforcement does not need to be in a poor state

→ Native implementation via Cilium's Bandwidth Manager (GA since v1.12)

- Efficient, eBPF-based bandwidth enforcement via EDT model
- First CNI to support BBR (& socket pacing) for Pods
- Side-note: Realizing such architecture only possible with eBPF

Getting Started Guide for Bandwidth Manager:

```
helm upgrade cilium ./cilium \
  --namespace kube-system \
  --reuse-values \
  --set bandwidthManager.enabled=true \
  --set bandwidthManager.bbr=true
kubectl -n kube-system rollout restart ds/cilium
```

(needs Linux kernel v5.1+)

(needs Linux kernel v5.18+)

Acknowledgements

- ➔ Van Jacobson
- ➔ Eric Dumazet
- ➔ Vytautas Valancius
- ➔ Stanislav Fomichev
- ➔ Martin Lau
- ➔ John Fastabend
- ➔ Cilium, BPF & netdev kernel community

ISOVALENT

Thank you! Questions?

github.com/cilium/cilium

cilium.io

ebpf.io

Isovalent: booth S21

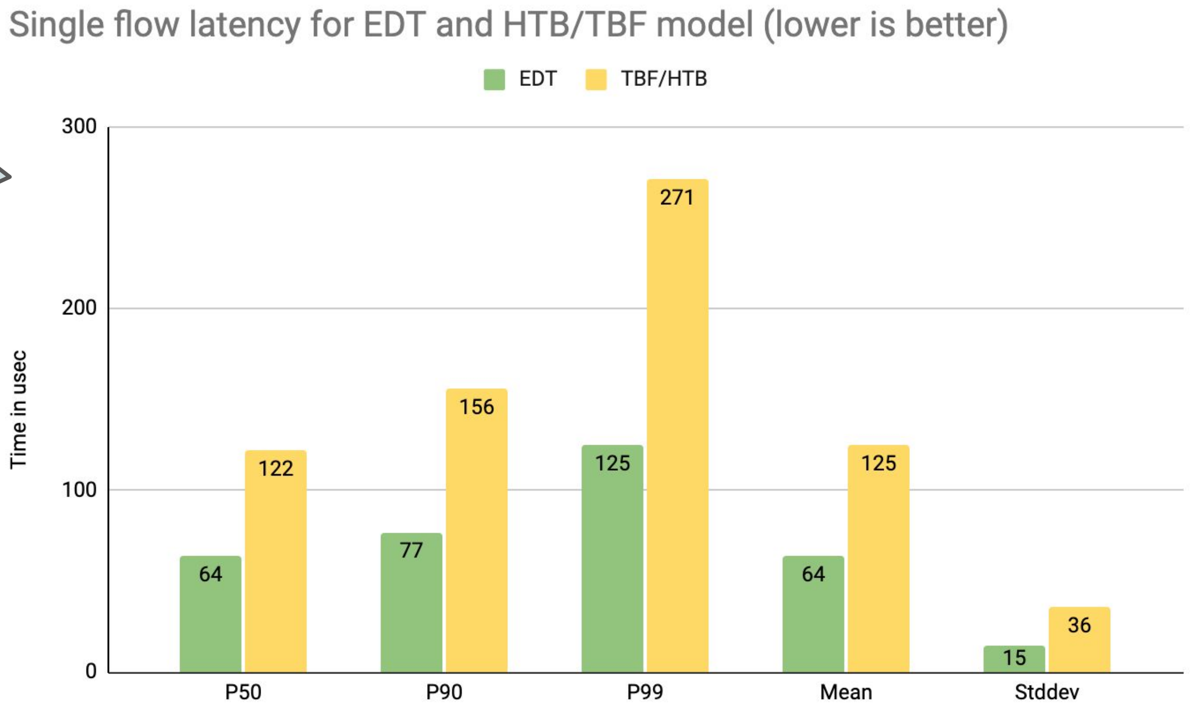
Cilium: kiosk 12 (@ project pavilion)

ISOVALENT

Appendix: Latency & TPS for 64/128/256 flows

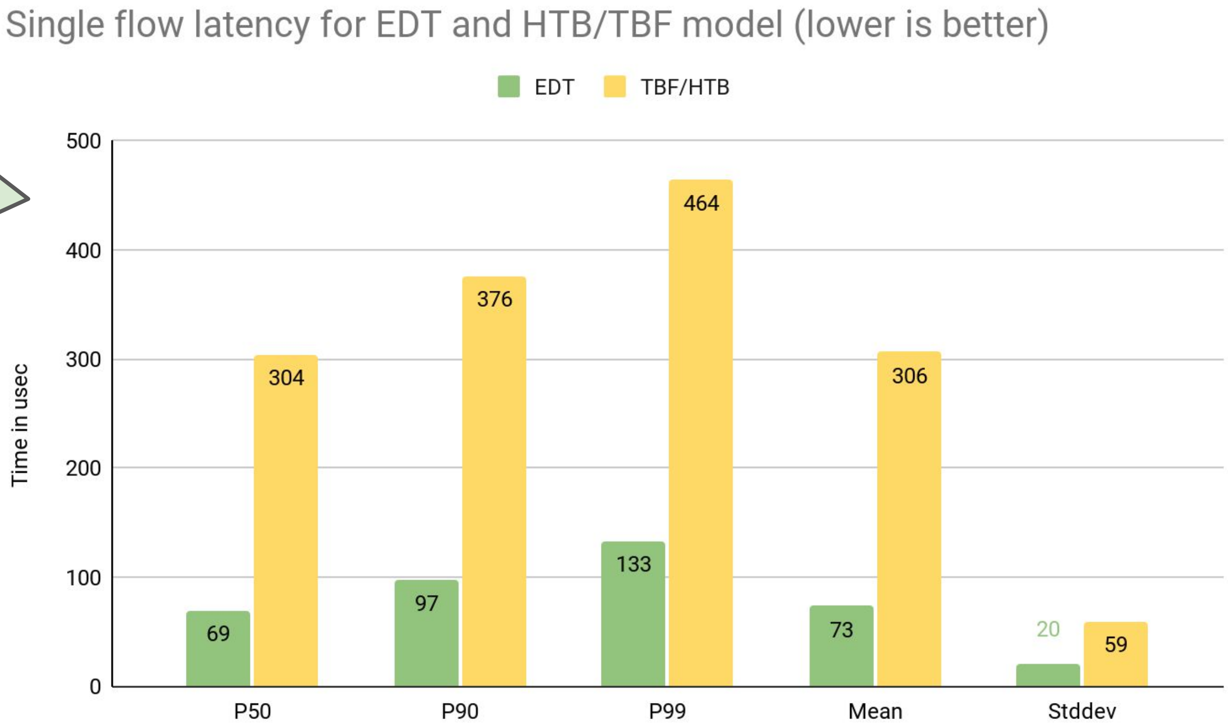
Comparison of Cilium's EDT implementation vs TBF

Env: 64 concurrent request/response type flows (TCP_RR), 100M rate per flow



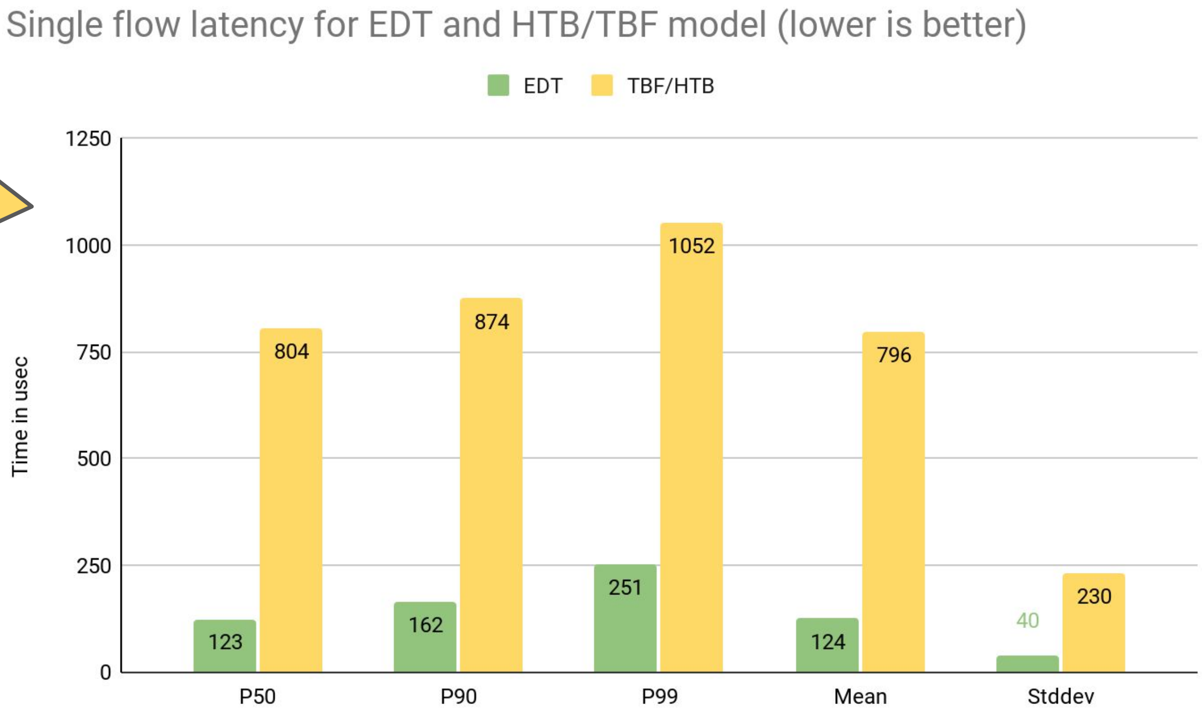
Comparison of Cilium's EDT implementation vs TBF

Env: 128 concurrent request/response type flows (TCP_RR), 100M rate per flow



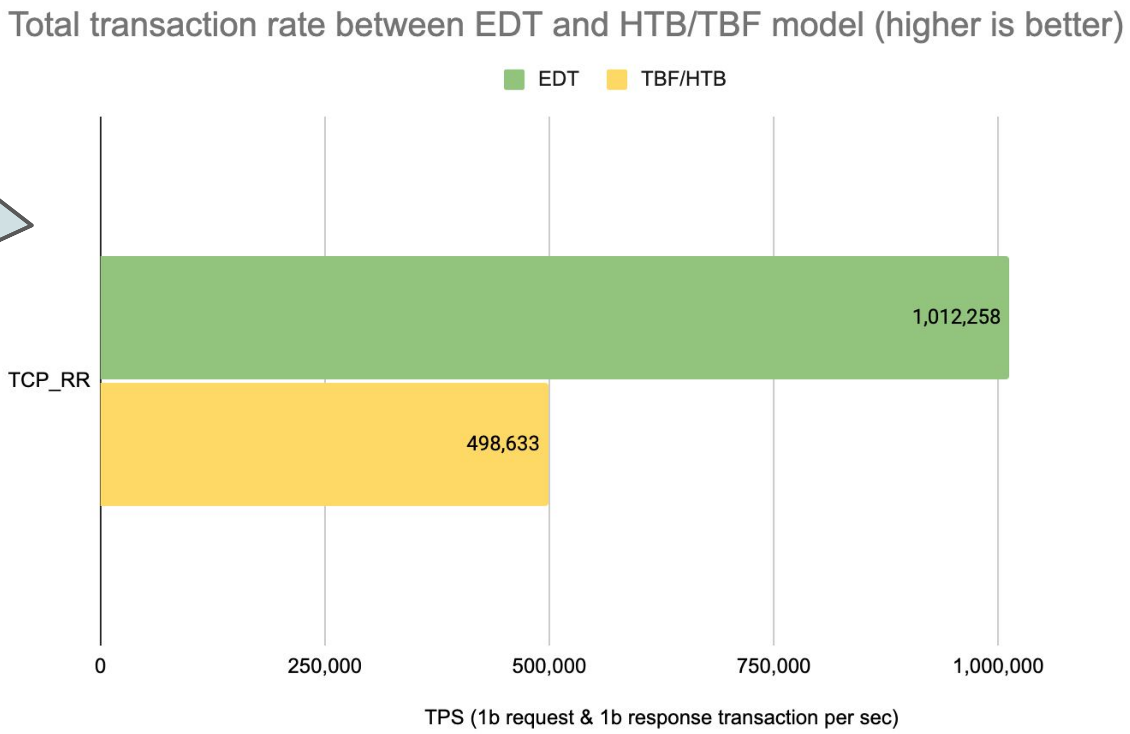
Comparison of Cilium’s EDT implementation vs TBF

Env: 256 concurrent request/response type flows (TCP_RR), 100M rate per flow



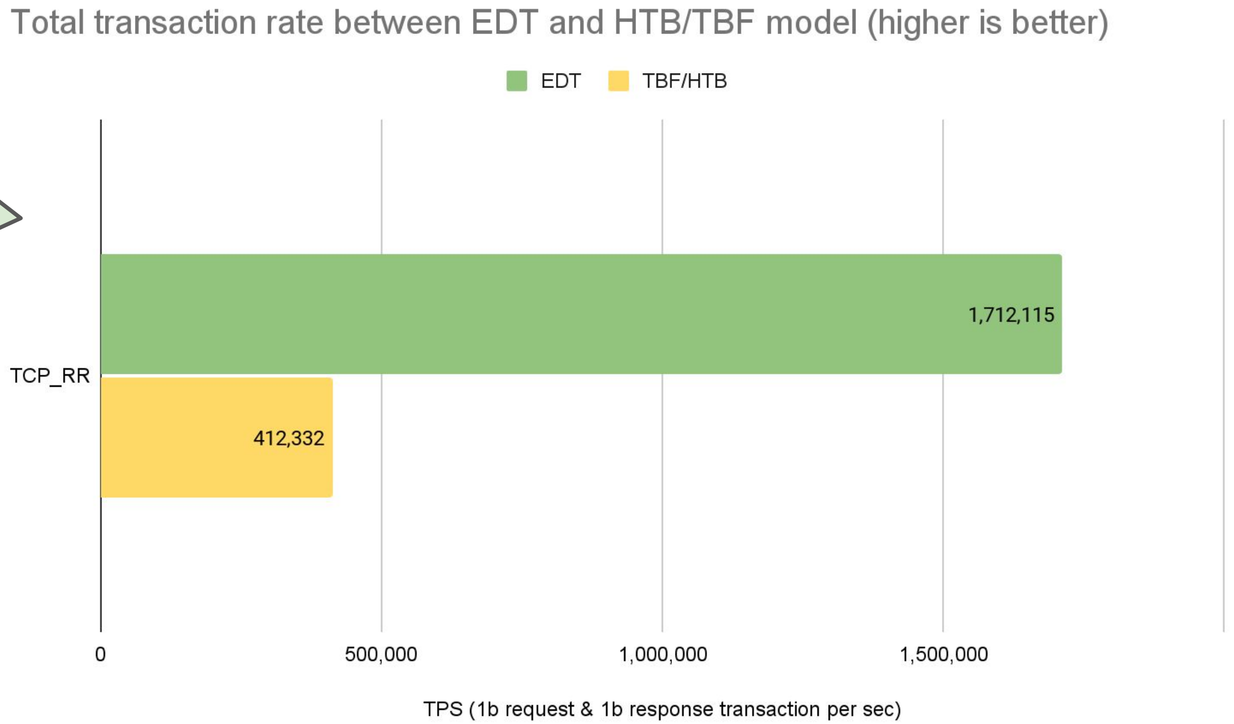
Comparison of Cilium's EDT implementation vs TBF

Env: 64 concurrent request/response type flows (TCP_RR), 100M rate per flow

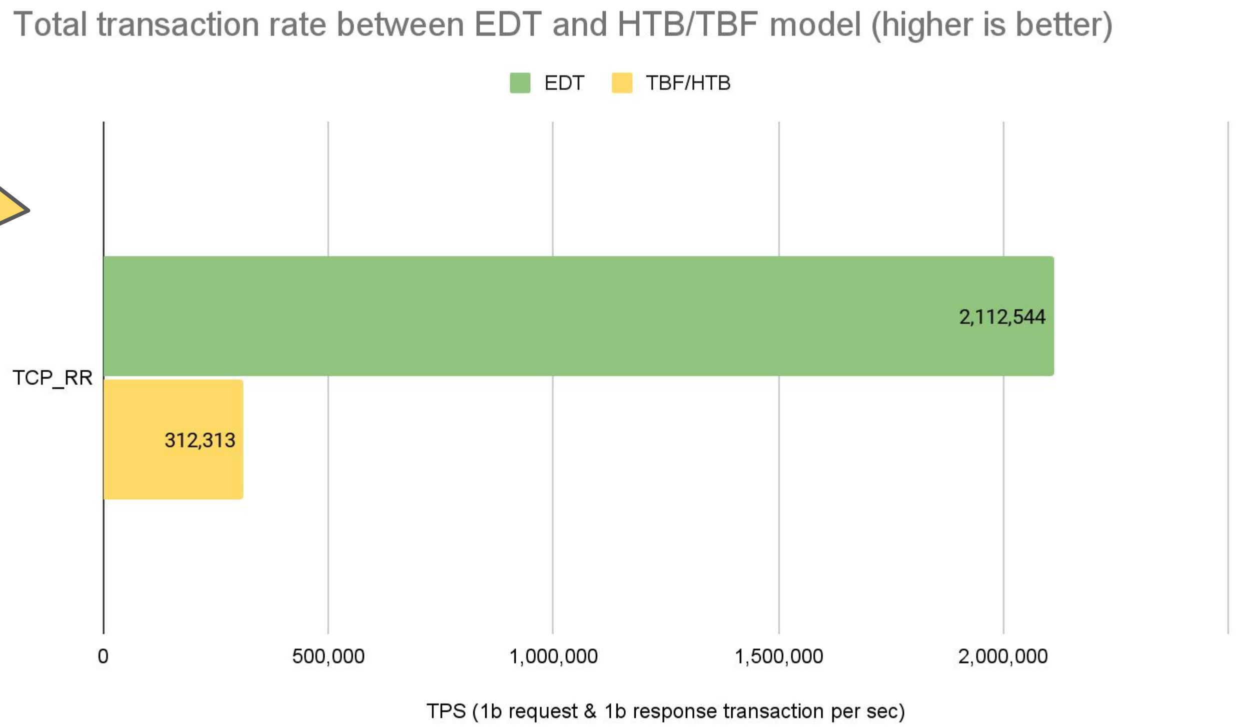


Comparison of Cilium's EDT implementation vs TBF

Env: 128 concurrent request/response type flows (TCP_RR), 100M rate per flow



Comparison of Cilium's EDT implementation vs TBF



Env: 256 concurrent request/response type flows (TCP_RR), 100M rate per flow