What You Didn’t Know About Ingress Controllers’ Performance

Ismo Puustinen & Mikko Ylinen, Intel

Agenda

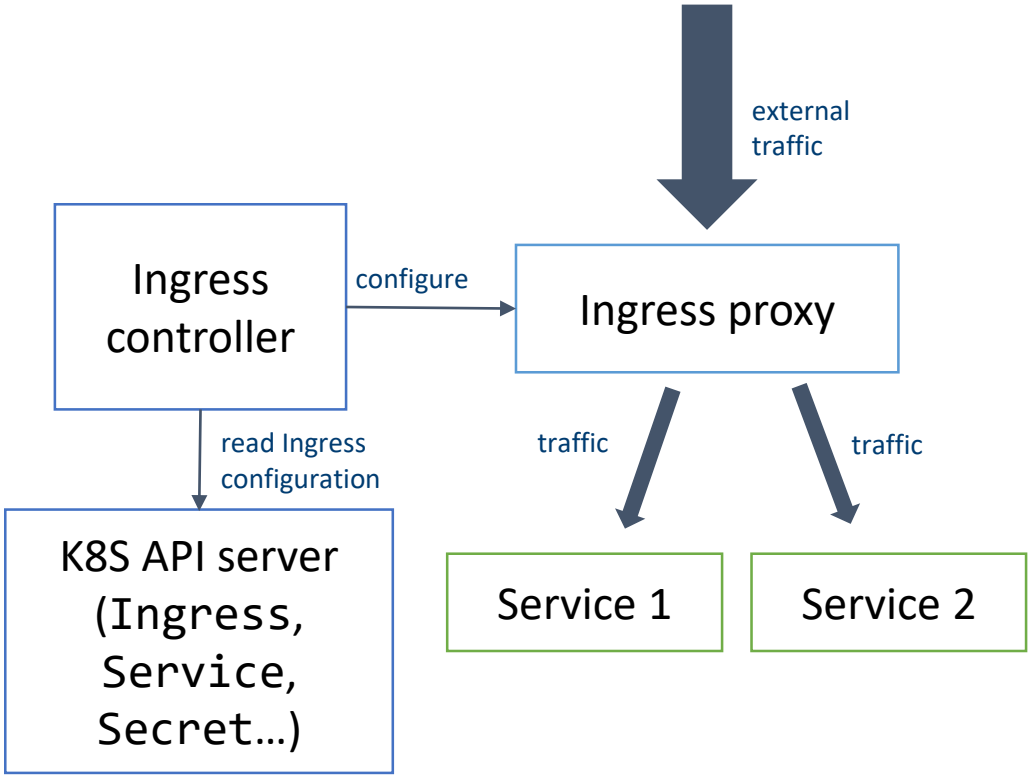
- Ingress overview and performance metrics
- Ingress controller optimization using
  - TLS offloading and asynchonous processing
  - Kubernetes CPU manager and alignment with CPU resources
- Call to Action

Introduction 1/2

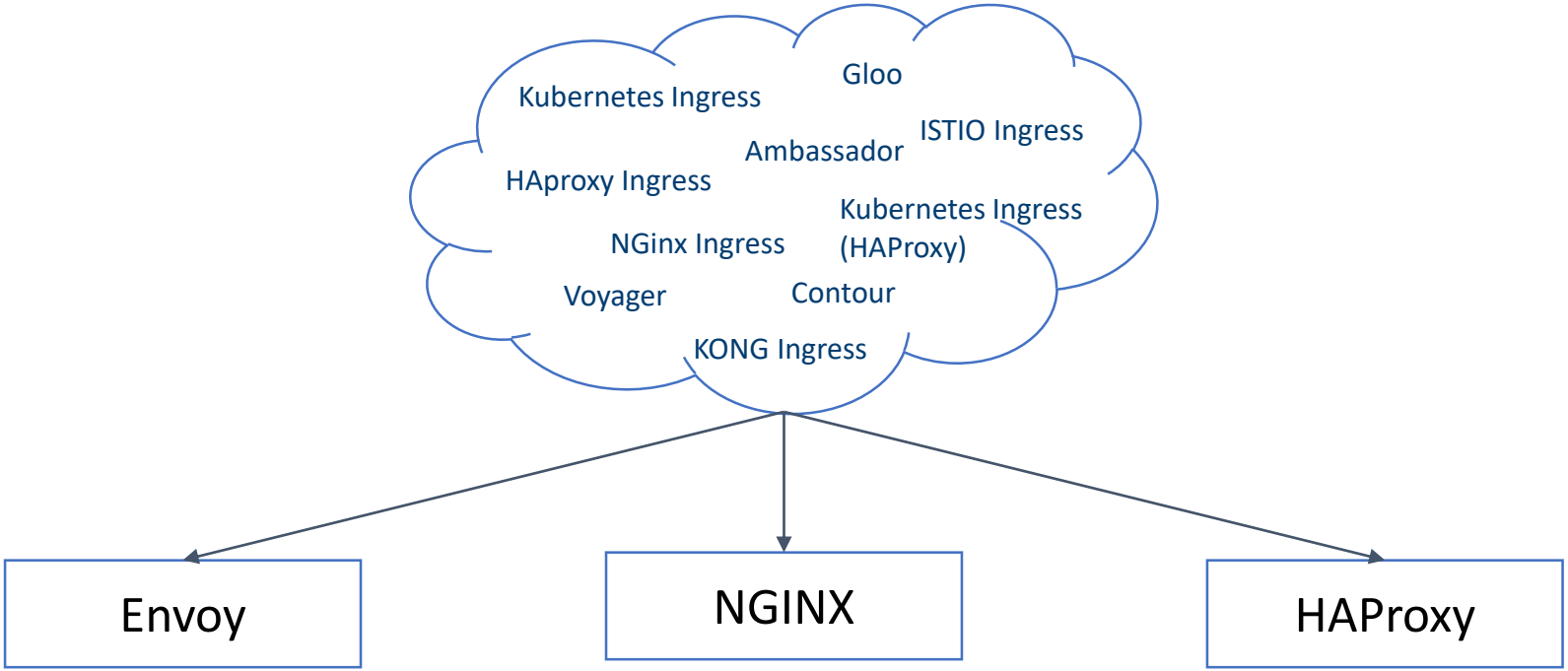
- What is Ingress?
  - Ingress exposes HTTP(s) routes to services within the cluster
    - “Route traffic coming to `https://example.com/foo/bar` to service `foobar`”
  - An Ingress controller is the implementation, at least one is required

Introduction 2/2

- Typical Ingress setup
  - Ingress controller follows the Kubernetes objects and creates the route configuration
  - Ingress proxy reads the configuration and handles the actual traffic routing



Kubernetes Ingress Controllers



https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/#additional-controllers

Performance Metrics

- Ingress proxy processing as performance bottleneck
  - Bandwidth (requests/sec)
  - Latency (ms/request, especially for tail)

```
time="2020-07-09T05:55:47Z" level=info msg="Test finished" i=43932 t=10.000065893s

    data_received..............: 60 MB 6.0 MB/s
    data_sent..................: 20 MB 2.0 MB/s
    http_req_blocked...........: avg=13.25ms  min=1.56ms   med=10.65ms  max=112.89ms p(90)=25.55ms  p(95)=31.43ms
    http_req_connecting........: avg=[illegible] min=45.26µs  med=157.05µs max=15.03ms  p(90)=206.51µ [illegible]
    http_req_duration..........: avg=13.6ms   min=217.88µs med=11.26ms  max=76.48ms  p(90)=27.64ms  p(95)=33.39ms
    http_req_receiving.........: avg=[illegible] min=15.84µs  med=100.4µs  max=41.77ms  p(90)=149.96µ [illegible]
    http_req_sending...........: avg=137.3µs  min=18µs     med=66.64µs  max=24.05ms  p(90)=104.57µs p(95)=153.98µs
    http_req_tls_handshaking...: avg=12.72ms  min=1.43ms   med=10.31ms  max=75.83ms  p(90)=24.87ms  p(95)=30.42ms
    http_req_waiting...........: avg=[illegible] min=[illegible]57µs med=10.97ms  max=76.33ms  p(90)=27.37ms  p(95)=33.14ms
    http_reqs..................: 4394  4394.171045/s
```

Performance Tuning Areas

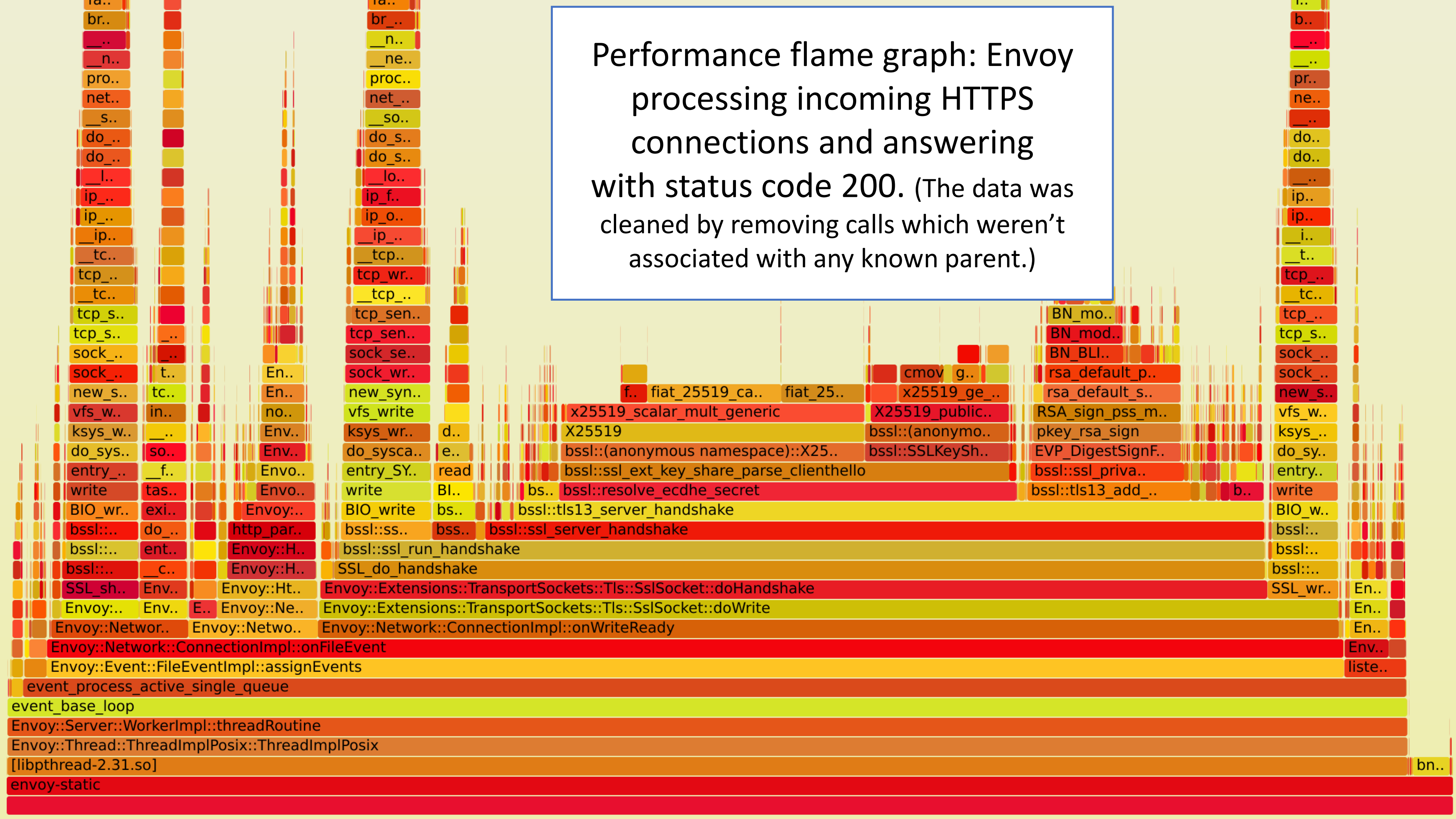
Kubernetes

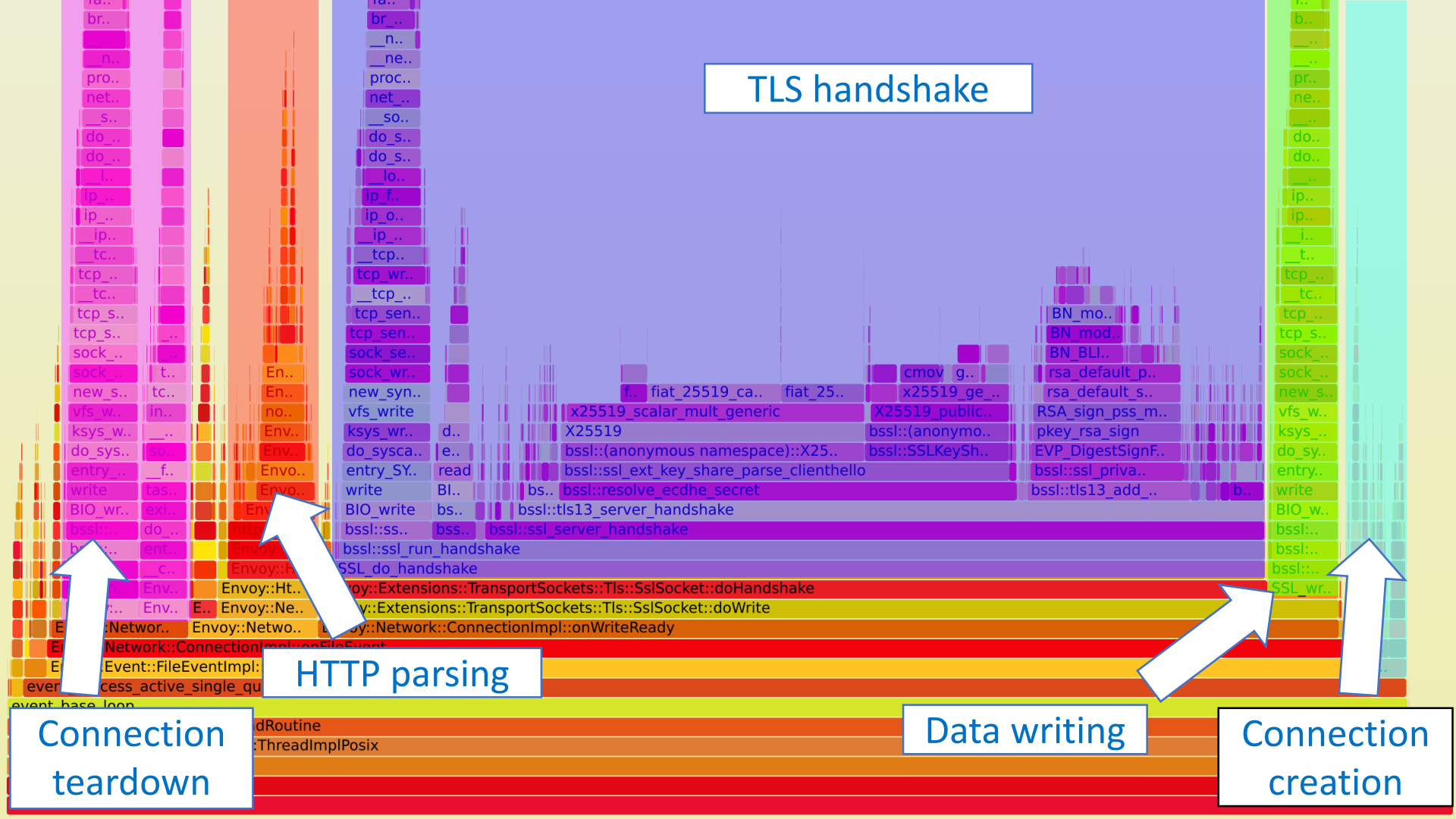
- Linux scheduling (QoS)
  - CFS, NUMA
- CPU manager policy
- Assignment of HW resources
- Ingress proxy configuration
  - Number of routes
  - Replicas
- Cluster topology (network)

Proxy

- Proxy threads and memory
  - In line with HW resources
  - Buffers
- Request content processing
  - TLS handshake
- Ingress proxy configuration
  - Upstream (cluster) load balancing
  - Timeouts, connection limits, …

Performance flame graph: Envoy processing incoming HTTPS connections and answering with status code 200. (The data was cleaned by removing calls which weren’t associated with any known parent.)


envoy-static
[libpthread-2.31.so]
bn..
Envoy::Thread::ThreadImplPosix::ThreadImplPosix
Envoy::Server::WorkerImpl::threadRoutine
event_base_loop
event_process_active_single_queue
Envoy::Event::FileEventImpl::assignEvents
liste..
Envoy::Network::ConnectionImpl::onFileEvent
Env..
Envoy::Networ..
Envoy::Netwo..
Envoy::Network::ConnectionImpl::onWriteReady
En..
Envoy:..
Env..
E..
Envoy::Ne..
Envoy::Extensions::TransportSockets::Tls::SslSocket::doWrite
SSL_sh..
Env..
Envoy::Ht..
Envoy::Extensions::TransportSockets::Tls::SslSocket::doHandshake
SSL_wr..
bssl::..
_c..
Envoy::H..
SSL_do_handshake
bssl::..
ent..
Envoy::H..
bssl::ssl_run_handshake
bssl::..
do_..
http_par..
bssl::ss..
bss..
bssl::ssl_server_handshake
BIO_wr..
exi..
Envoy:..
BIO_write
bs..
bssl::tls13_server_handshake
BIO_w..
write
tas..
Envo..
write
Bl..
bs..
bssl::resolve_ecdhe_secret
bssl::tls13_add_..
b..
entry_..
_f..
Envo..
entry_SY..
read
bssl::ssl_ext_key_share_parse_clienthello
bssl::ssl_priva..
entry..
do_sys..
so..
Env..
do_sysca..
e..
bssl::(anonymous namespace)::X25..
bssl::SSLKeySh..
EVP_DigestSignF..
do_sy..
ksys_w..
Env..
ksys_wr..
d..
X25519
bssl::(anonymo..
pkey_rsa_sign
ksys_..
vfs_w..
in..
no..
vfs_write
x25519_scalar_mult_generic
X25519_public..
RSA_sign_pss_m..
vfs_w..
new_s..
tc..
En..
new_syn..
f..
fiat_25519_ca..
fiat_25..
x25519_ge_..
rsa_default_s..
new_s..
sock_..
t..
En..
sock_wr..
cmov
g..
rsa_default_p..
sock_..
sock_..
sock_se..
BN_BLI..
sock_..
tcp_s..
tcp_sen..
BN_mod..
tcp_s..
tcp_s..
tcp_sen..
BN_mo..
tcp_..
_tc..
tcp..
_tc..
tcp_..
tcp_wr..
tcp_..
_tc..
_tcp..
_t..
_ip..
_ip_..
_i..
ip_..
ip_o..
ip..
ip_..
ip_f..
ip..
_l..
_lo..
do_..
do_s..
do..
do_..
do_s..
do..
_s..
_so..
net..
net_..
ne..
pro..
proc..
pr..
_n..
_ne..
br..
br_..
b..


TLS handshake
HTTP parsing
Connection teardown
Data writing
Connection creation
x25519_scalar_mult_generic
X25519
bssl::resolve_ecdhe_secret
bssl::tls13_server_handshake
bssl::ssl_server_handshake
bssl::ssl_run_handshake
SSL_do_handshake
Envoy::Extensions::TransportSockets::Tls::SslSocket::doHandshake
Envoy::Extensions::TransportSockets::Tls::SslSocket::doWrite
Envoy::Network::ConnectionImpl::onWriteReady
event_base_loop
ThreadImplPosix

Anatomy of a TLS Connection

- Two separate parts: handshake (asymmetric crypto) and data transfer (symmetric crypto)
  - Symmetric crypto is pretty fast
    - OpenSSL, BoringSSL, GnuTLS support AES instructions which make it even faster
  - **Asymmetric crypto is slow (ECDHE-RSA cipher suites)**
    - Receiving new HTTPs connections is where the CPU cycles are spent!

Faster TLS Handshakes?

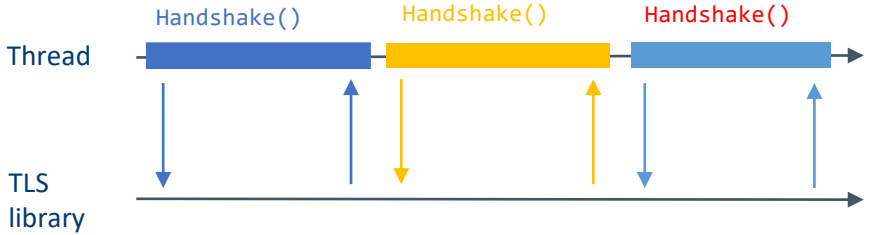
- Crypto parameters (TLS1.3, ECDSA vs. RSA, ECDHE vs. RSA…) – feasible?
- Crypto operations can be accelerated by offloading them to an external accelerator
  - Individual operations or vectorization (wait until several operations are received)
  - Different TLS libraries have different methods for offloading crypto operations to external providers:

OpenSSL	OpenSSL Engines
BoringSSL	Private key operations
GnuTLS	Cryptography provider layer

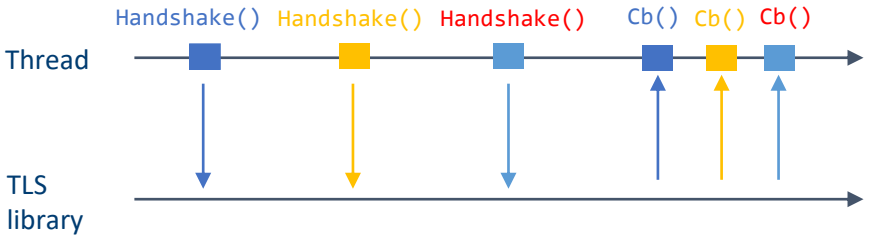
  - It's common that accelerating a single crypto operation isn't much faster, but many operations can be done in parallel: **asynchronous handshake processing**

Sync vs. Async TLS

Synchronous TLS



Asynchronous TLS

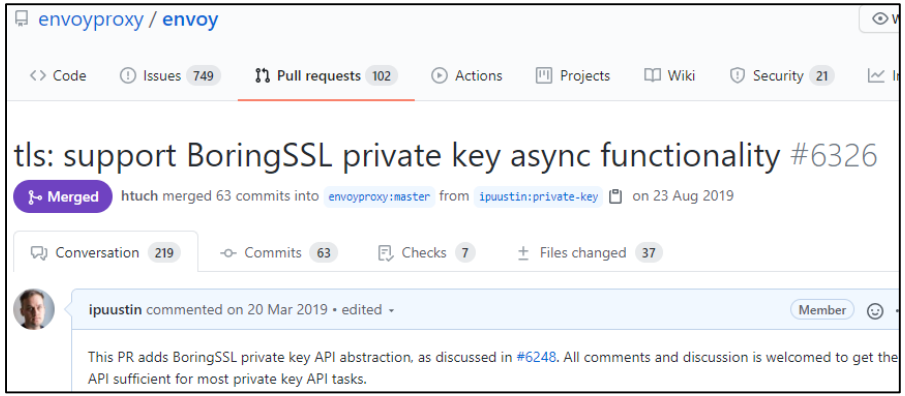


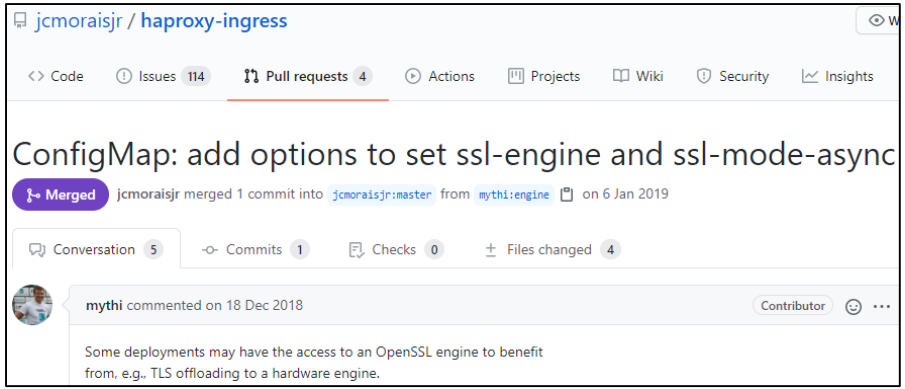
Asynchronous TLS

- Handshake function returns immediately, callback is called when the handshake is done
  - Crypto is accelerated in the background while Ingress proxy server can do other things
- Sub-problems:
  1. How to get the Ingress proxy servers to support asynchronous handshakes?
  2. How to get accelerator resources to Kubernetes Ingress Controllers?

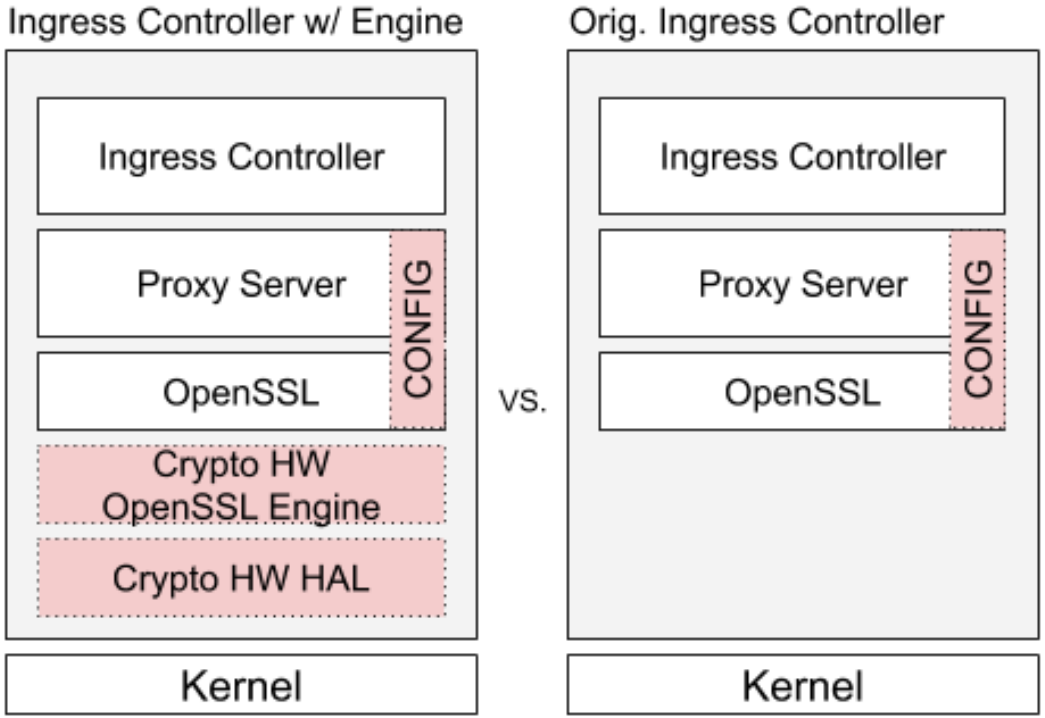
Async TLS Support In Ingress

1. Add asynchronous TLS support to underlying proxy servers
2. Add support to Ingress controllers' configuration for getting asynchronous TLS used in the proxy





Async TLS Offloading



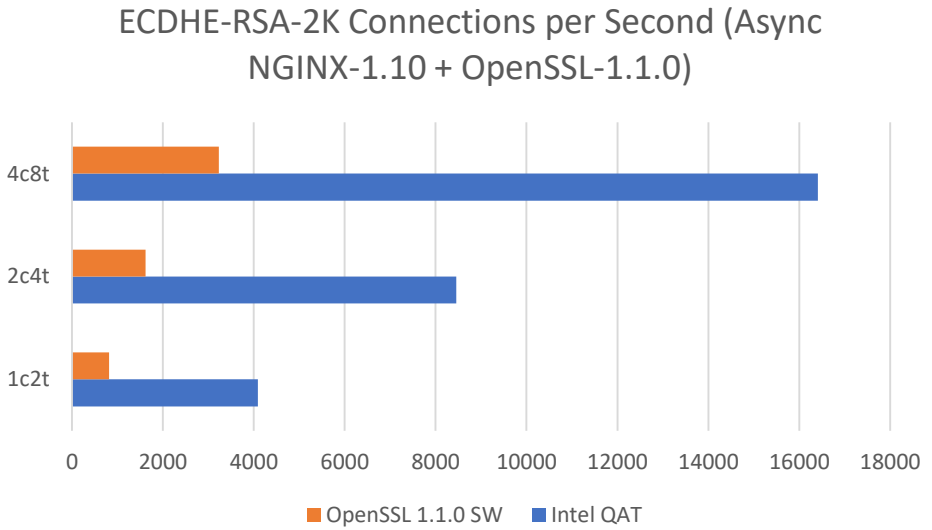
TLS Offloading Status

	NGINX	HAProxy	Envoy	Envoy-OpenSSL
Crypto offload	OpenSSL Engine	OpenSSL Engine	BoringSSL Private Key Provider	OpenSSL Engine
Asynchronous Processing	No*	Handshakes	Handshakes	Handshakes
Rebuild to use Offloading	No	No	Yes	No
Ingress Configuration Supported	(`OPENSSL_CONF` override)	Haproxy-ingress, Kubernetes (Haproxy) Ingress		(`OPENSSL_CONF` override)

* Asynch_mode_nginx: Handshakes and bulk crypto

When To Offload?

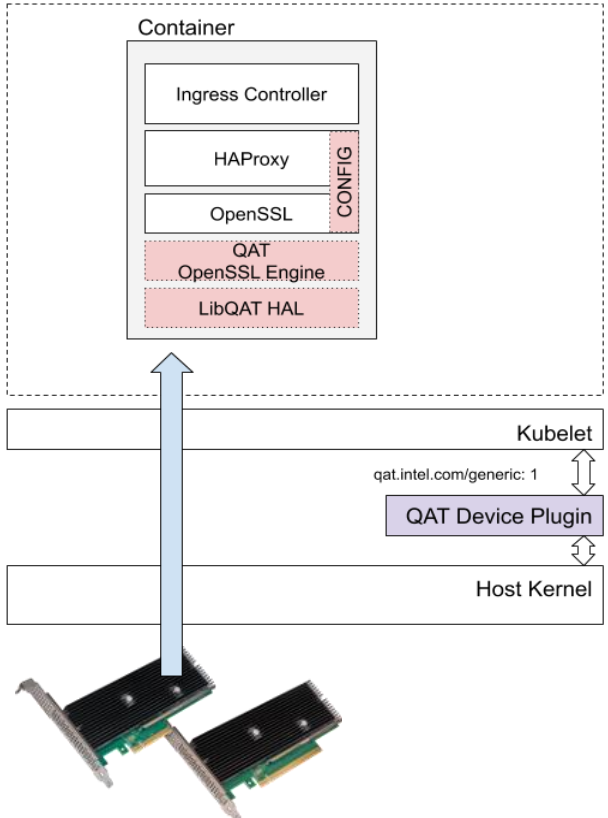
- Identify if the ingress controller proxy is **CPU-bound**
  - Adding more cores increases throughput and/or reduces latency
- Find out if you have a lot of incoming **new** HTTPs connections
  - Upstream connections don't matter, because the TLS connections are reused
- The biggest benefit from the acceleration is when the Ingress proxy is running on a **low CPU core count** (1-8 cores)



Reference: https://01.org/sites/default/files/downloads/intelr-quickassist-technology/337003-001-intelquickassisttechnologyandopenssl-110.pdf

Examples

HAProxy Hardware Acceleration



ConfigMap

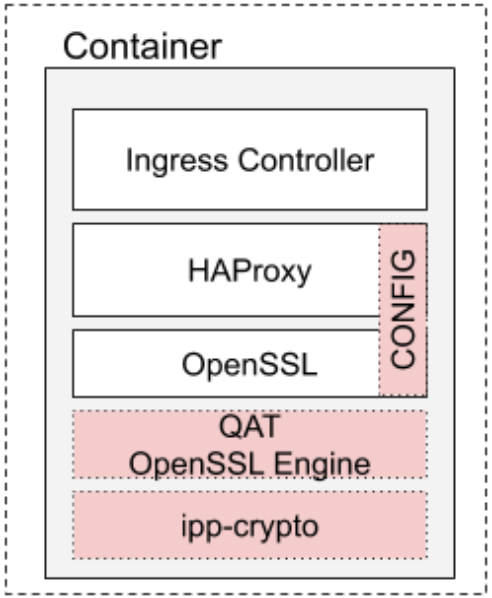
```
config-snippet: |
  ssl-engine qat
  ssl-mode-async
```

Custom Deployment

```
containers:
  ...
  - name: demo-container-1
    image: custom-haproxy-ingress:2.0
    resources:
      limits:
          qat.intel.com/generic: 1
```

HAProxy RSA Multibuffer

- RSA Multibuffer and asynchronous OpenSSL engine
  - queue up-to eight RSA operations and process them in parallel
  - ipp-crypto's multi-buffer
  - AVX512 IFMA (e.g., Ice Lake CPUs) enabled nodes



ConfigMap

```
config-snippet: |
  ssl-engine qat
  ssl-mode-async
```

Custom Deployment (`node-feature-discovery` labeled nodes):

```
...
spec:
  template:
    spec:
      nodeSelector:
        feature.node.kubernetes.io/cpu-cpuid.AVX512IFMA: "true"
```

Performance Tuning Areas

Kubernetes

- Linux scheduling (QoS)
  - CFS, NUMA
- CPU manager policy
- Assignment of HW resources
- Ingress proxy configuration
  - Number of routes
  - Replicas
- Cluster topology (network)

Proxy

- Proxy threads and memory
  - In line with HW resources
  - Buffers
- Request content processing
  - TLS handshake
- Ingress proxy configuration
  - Upstream (cluster) load balancing
  - Timeouts, connection limits, …

Kubernetes CPU Manager

- Ingress proxy performance scales almost linearly with CPU cores
- Kubelet CPU Manager “static” policy can be used to assign full cores for a container in a pod with Guaranteed QoS class
  - “Full cores” means that other containers will not be scheduled to the `cpuset` belonging to the container. Also, CPU manager selects cores which are topologically close, meaning threads on the same physical core, cores on the same socket, and so on.
- Problem: how to choose the hard limits? What if Ingress controller memory usage increases when changing the Ingress configuration?

QoS class is a pod-level property.

Guaranteed: all containers in the pod need to have both request and limit set to same value for all native resources.

Burstable: at least one container needs to have a native resource request or limit set.

Best-effort: everything else.

CPU Manager Performance

- The performance impact depends on other workloads running on the same node
  - Difficult to quantify! Test this in your own environment.
- Performance scales both to larger and smaller workloads
  - Large workloads will get CPU cores with good topology
  - Small workloads benefit from less processor cache misses and immunity to CFS throttling

Proxy Settings And Resources

- The Ingress proxies' resource use can be configured
  - Worker threads, memory buffers, connections per route, …
- Proxies typically try to autodetect the amount of resources available: for bare metal and virtual machines this works fine
  - Problem: when running in container, the CPU core count auto-detection doesn't necessarily work at all, because the proxies aren't cgroup-aware: the worker thread count is easily much bigger than intended
  - Solution: set the parameters manually or use more advanced configuration values
  - Example: Envoy option `--cpuset-threads` adjusts the worker thread count to match with the Envoy cpuset size and not the number of cores in the system

Call To Action

- Users: analyze your workloads to see where the bottlenecks are
- Ingress controller developers: ensure the following config knobs are there:
  - Easy customization of crypto offload and async processing modes
  - Easy customization of non-native resoures
  - Easy customization of nodeAffinity labels
- Ingress proxy developers: switch to async TLS and integrate mechanisms for enabling custom TLS handshake


KubeCon
CloudNativeCon
Europe 2020
Virtual
HELM
KEEP CLOUD NATIVE
CONNECTED