Debugging K8s pod throughput with Calico CNI

Thomas Hartland

User report: “Cluster is slow to create pods”

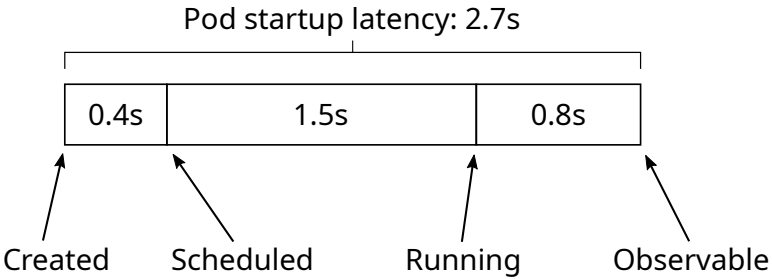
Me:

Clusterloader2

- Kubernetes benchmark tool [1].
- In this case, using it to
  - Create many pods in the cluster at once (N pods per node).
  - Measure the pod startup latencies.

[1] https://github.com/kubernetes/perf-tests/tree/master/clusterloader2

Pod startup latency



Measuring against

- Kubernetes pod startup latency SLI:

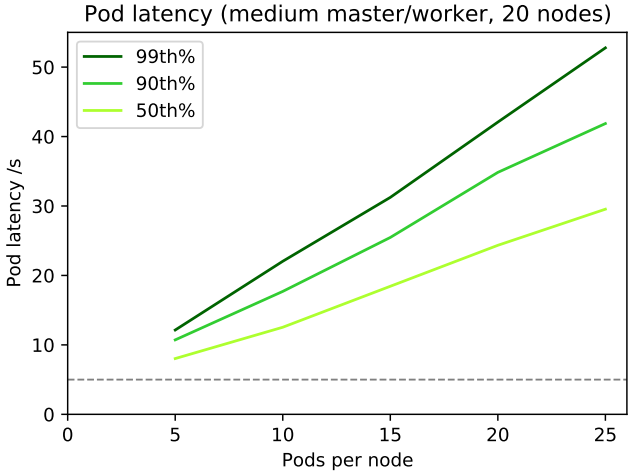
  Startup latency of schedulable stateless pods, excluding time to pull images and run init containers, measured from pod creation timestamp to when all its containers are reported as started and observed via watch, measured as 99th percentile over last 5 minutes. [2]

- The target: < 5 seconds

[2] https://github.com/kubernetes/community/blob/master/sig-scalability/slos/pod_startup_latency.md

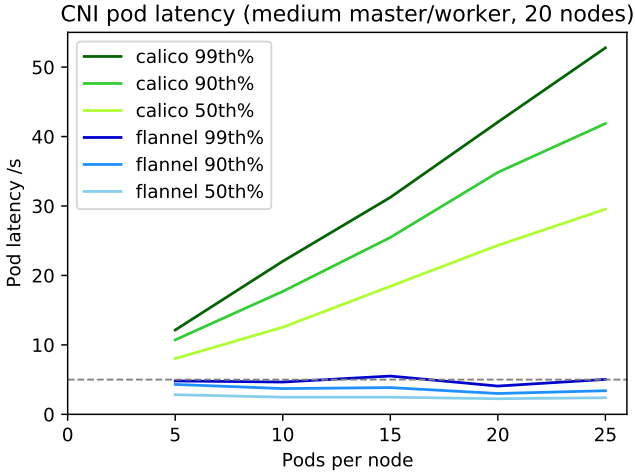
Measuring this myself

- Clusterloader2 measures the 50th, 90th and 99th percentiles of pod startup latency.
- I created a cluster with 20 nodes, to match the user's cluster.
- I started testing with clusterloader2, first with 5 pods per node and then increasing.


Pod latency (medium master/worker, 20 nodes)
99th%
90th%
50th%
Pod latency /s
0
10
20
30
40
50
0
5
10
15
20
25
Pods per node

- I looked in the kubelet logs for one node to check for anything unusual.
- I found a lot of errors like this:

```
[ERROR][1278786] customresource.go 136: Error updating resource
Key=IPAMBlock(10-100-145-192-26)
error=Operation cannot be fulfilled on ipamblocks.crd.projectcalico.org:
      the object has been modified; please apply your
      changes to the latest version and try again
```


CNI pod latency (medium master/worker, 20 nodes)
calico 99th%
calico 90th%
calico 50th%
flannel 99th%
flannel 90th%
flannel 50th%
Pod latency /s
0
10
20
30
40
50
0
5
10
15
20
25
Pods per node

Is the API server ok?

- No.

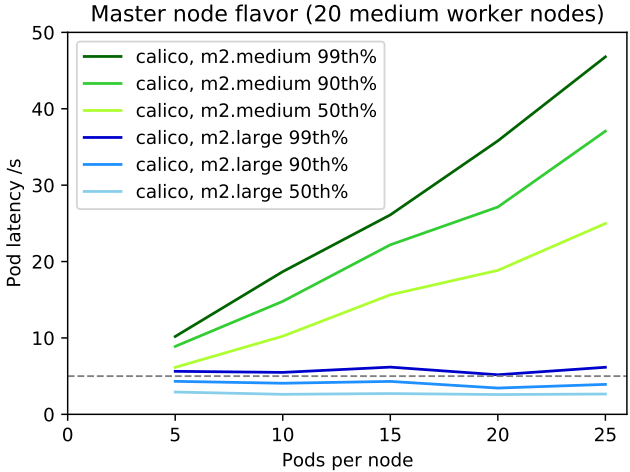
Is the API server ok?

- No.
- On the master node, the API server and etcd were fully using the available CPU.

Is the API server ok?

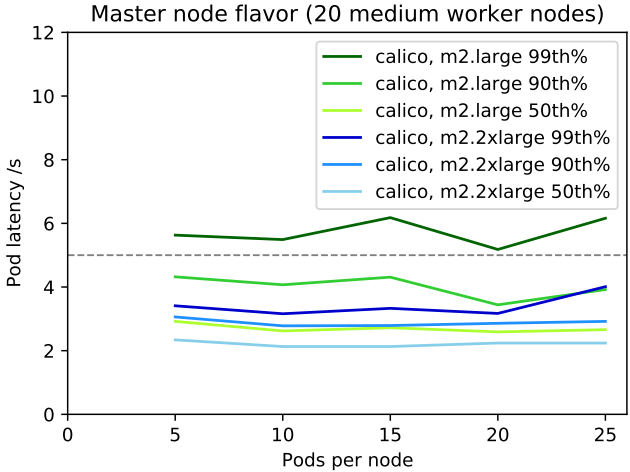
- No.

- On the master node, the API server and etcd were fully using the available CPU.
- I didn't change any defaults when creating the cluster, so it was only a 2-core m2.medium master node.


Master node flavor (20 medium worker nodes)
calico, m2.medium 99th%
calico, m2.medium 90th%
calico, m2.medium 50th%
calico, m2.large 99th%
calico, m2.large 90th%
calico, m2.large 50th%
Pod latency /s
Pods per node
0
10
20
30
40
50
0
5
10
15
20
25

Bigger is better

- Changing the master node to be a 4-core m2.large flavor brought things mostly under control.
- But, the 99th percentile is still not meeting the target.
- Let's go bigger! 16 cores!


Master node flavor (20 medium worker nodes)
Pod latency /s
Pods per node
0
2
4
6
8
10
12
0
5
10
15
20
25
calico, m2.large 99th%
calico, m2.large 90th%
calico, m2.large 50th%
calico, m2.2xlarge 99th%
calico, m2.2xlarge 90th%
calico, m2.2xlarge 50th%

So we're done?

- Yes, just make you sure choose a large enough master node to handle the expected amount of load in your cluster.

So we're done?

- Yes, just make you sure choose a large enough master node to handle the expected amount of load in your cluster.
- But.....

So we're done?

- Yes, just make you sure choose a large enough master node to handle the expected amount of load in your cluster.

- But.....
- I'm not satisfied yet!

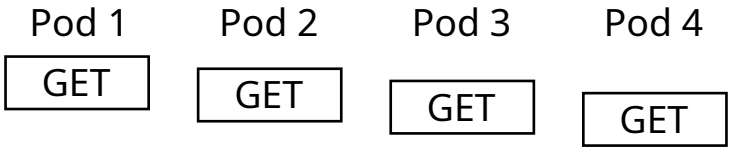
What's an IPAMBlock?

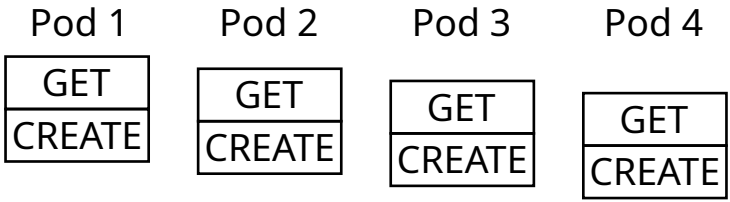
```
apiVersion: crd.projectcalico.org/v1
kind: IPAMBlock
metadata:
  name: 10-100-32-128-26
  resourceVersion: "1500"
spec:
  affinity: host:tgh-clusterloader-118-dz7jvsoaiqcc-node-0
  cidr: 10.100.32.128/26
  allocations:
  - 0
  - 1
  - 2
  - 3
```

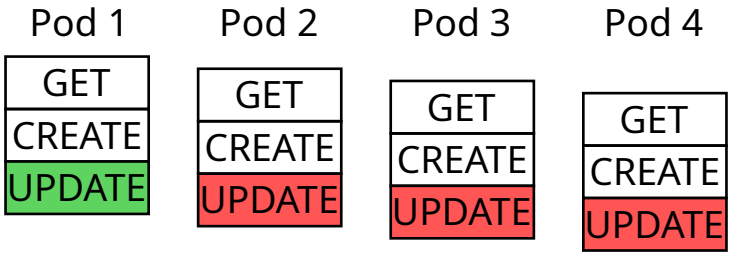
In the code

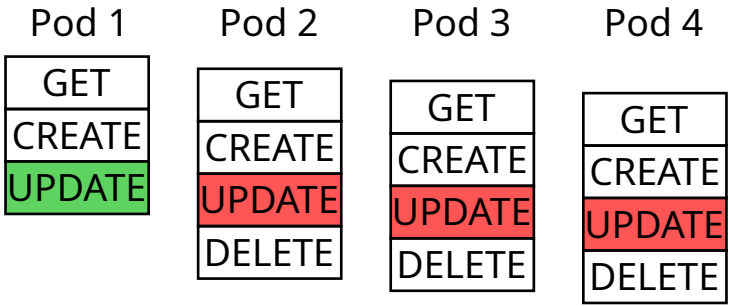
- When the calico-ipam CNI plugin is assigning an IP for a pod on a node, there is a section where it

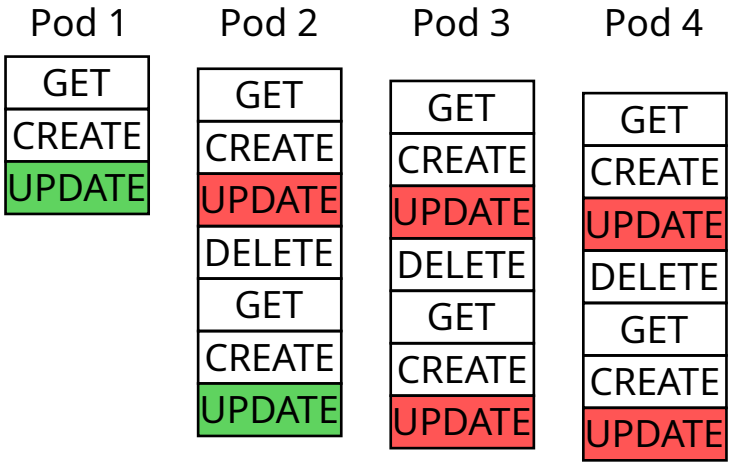
- Gets the IPAMBlock for the node
  - Picks an unused IP from the block
- Creates an IPAMHandle
- Tries to update the IPAMBlock to mark the IP as in use

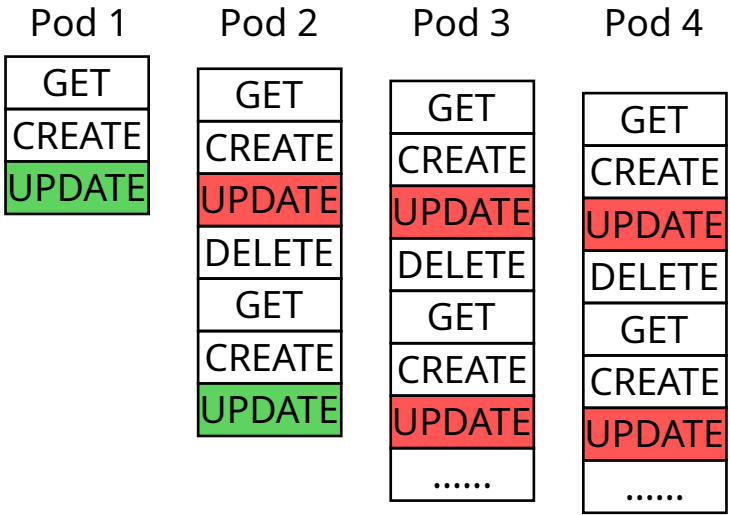

Pod 1
GET
Pod 2
GET
Pod 3
GET
Pod 4
GET


Pod 1
GET
CREATE
Pod 2
GET
CREATE
Pod 3
GET
CREATE
Pod 4
GET
CREATE


Pod 1
GET
CREATE
UPDATE
Pod 2
GET
CREATE
UPDATE
Pod 3
GET
CREATE
UPDATE
Pod 4
GET
CREATE
UPDATE


Pod 1
GET
CREATE
UPDATE
Pod 2
GET
CREATE
UPDATE
DELETE
Pod 3
GET
CREATE
UPDATE
DELETE
Pod 4
GET
CREATE
UPDATE
DELETE


Pod 1
GET
CREATE
UPDATE
Pod 2
GET
CREATE
UPDATE
DELETE
GET
CREATE
UPDATE
Pod 3
GET
CREATE
UPDATE
DELETE
GET
CREATE
UPDATE
Pod 4
GET
CREATE
UPDATE
DELETE
GET
CREATE
UPDATE


Pod 1
GET
CREATE
UPDATE
Pod 2
GET
CREATE
UPDATE
DELETE
GET
CREATE
UPDATE
Pod 3
GET
CREATE
UPDATE
DELETE
GET
CREATE
UPDATE
.....
Pod 4
GET
CREATE
UPDATE
DELETE
GET
CREATE
UPDATE
.....

Feedback loop

- The load on the API server causes calico to make conflicting updates to IPAMBlocks.
- Calico retries the updates which failed.
- Putting more load on the API server!

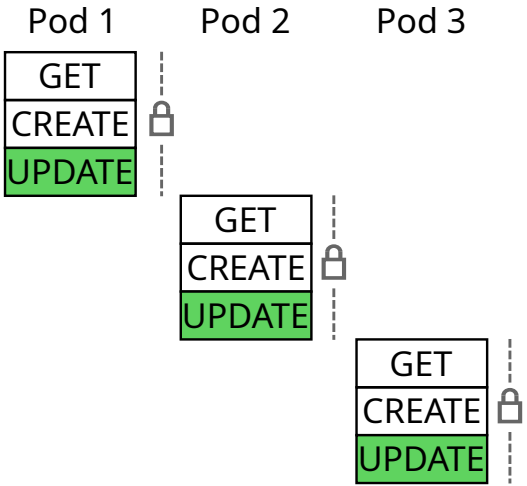
So what's the solution?

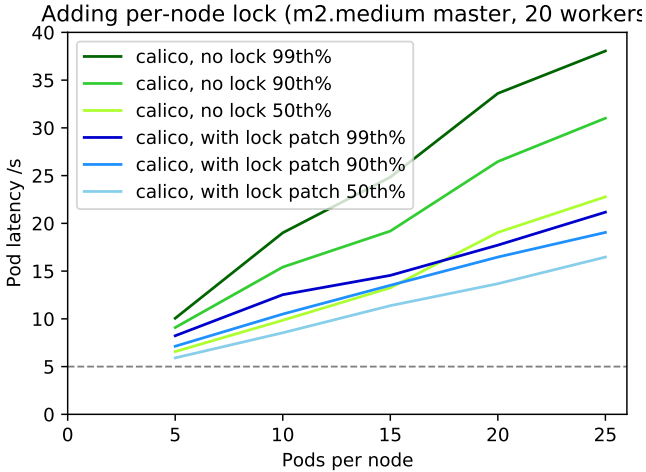
- The solution is still to use a larger master node.
- But at this point I went to the Calico #kubernetes Slack channel to speak to the developers.

So what's the solution?

- There are improvements that can be made in the worst case.
- Using a shared (file based) lock on each node to guard the IPAMBlock API, contention can be eliminated[3].

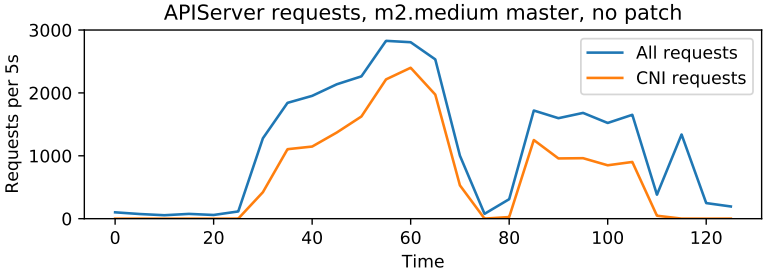
[3] https://github.com/projectcalico/cni-plugin/pull/985

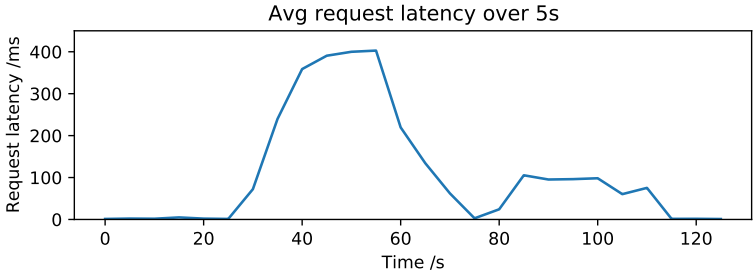

Pod 1
Pod 2
Pod 3
GET
CREATE
UPDATE
GET
CREATE
UPDATE
GET
CREATE
UPDATE

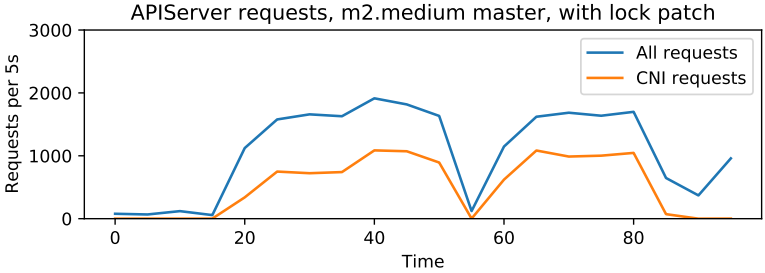

Adding per-node lock (m2.medium master, 20 workers
calico, no lock 99th%
calico, no lock 90th%
calico, no lock 50th%
calico, with lock patch 99th%
calico, with lock patch 90th%
calico, with lock patch 50th%
Pod latency /s
Pods per node
0
5
10
15
20
25
30
35
40

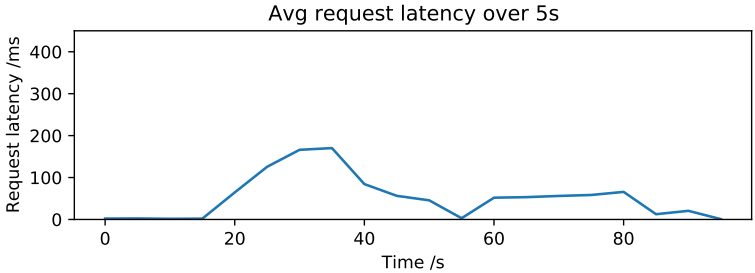
Checking the API server

- The API server logs contain request latencies, the time from receiving a request to returning a response.
- We can look at those, and the raw request rate, to see how this change impacts the API.


APIServer requests, m2.medium master, no patch
Requests per 5s
Time
All requests
CNI requests
0
1000
2000
3000
0
20
40
60
80
100
120


Avg request latency over 5s
Request latency /ms
Time /s
0
100
200
300
400
0
20
40
60
80
100
120


APIServer requests, m2.medium master, with lock patch
Requests per 5s
3000
2000
1000
0
All requests
CNI requests
0
20
40
60
80
Time


Avg request latency over 5s
Request latency /ms
400
300
200
100
0
0
20
40
60
80
Time /s

The result

- In total, number of requests from the CNI is reduced from 20k to 10k over the duration of the test.
- The peak API response latency is cut in half as well.

And on larger master nodes

- Possible 60ms -> 30ms peak response time on m2.large master node with the patch applied.
- m2.2xlarge master node maintains ~3ms response time even without patch.

Finally

- And in the end...

Finally

- And in the end...
- The problem in the user report was unrelated.

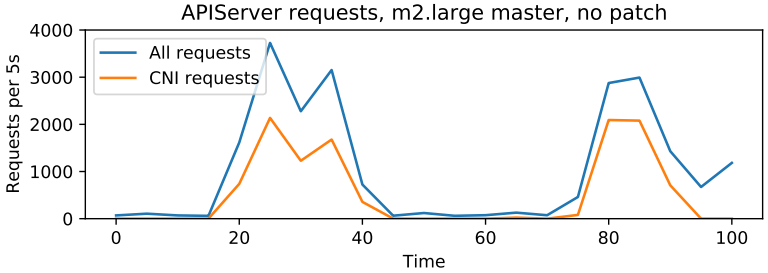
Finally

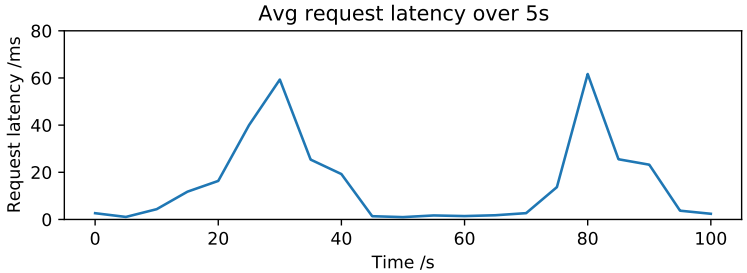
- And in the end...

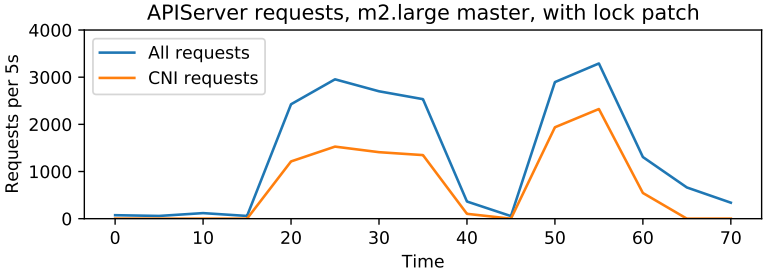
- The problem in the user report was unrelated.
- (Spinning disks on the nodes).

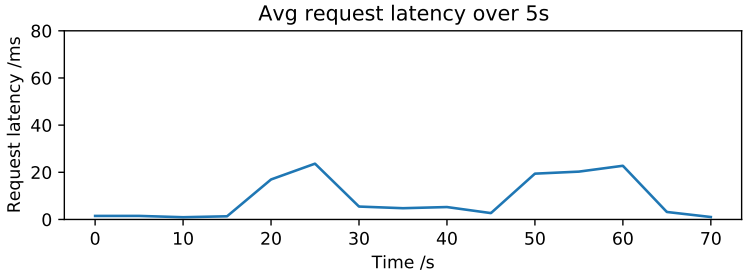
End

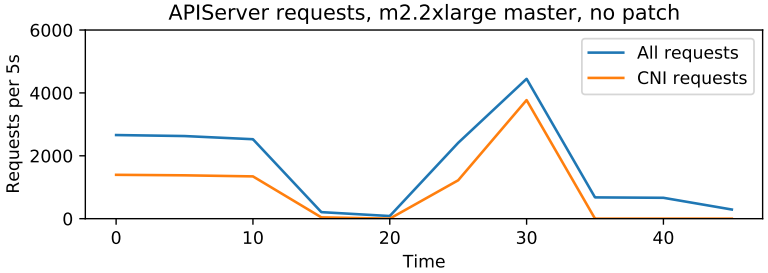
Extra slides

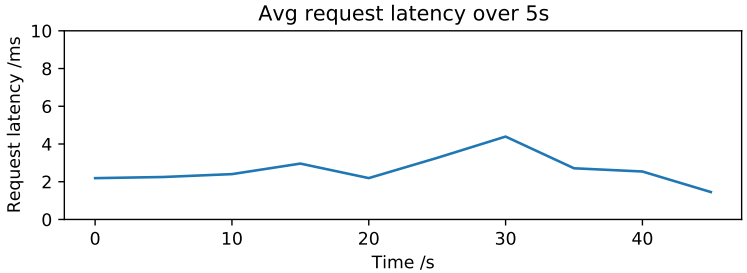

APIServer requests, m2.large master, no patch
All requests
CNI requests
Requests per 5s
4000
3000
2000
1000
0
0
20
40
60
80
100
Time


Avg request latency over 5s
Request latency /ms
80
60
40
20
0
0
20
40
60
80
100
Time /s


APIServer requests, m2.large master, with lock patch
All requests
CNI requests
Requests per 5s
4000
3000
2000
1000
0
0
10
20
30
40
50
60
70
Time


Avg request latency over 5s
Request latency /ms
80
60
40
20
0
0
10
20
30
40
50
60
70
Time /s


APIServer requests, m2.2xlarge master, no patch
Requests per 5s
6000
4000
2000
0
All requests
CNI requests
0
10
20
30
40
Time


Avg request latency over 5s
Request latency /ms
10
8
6
4
2
0
0
10
20
30
40
Time /s