Kubernetes made my latency 10x higher

Oct 22, 2019

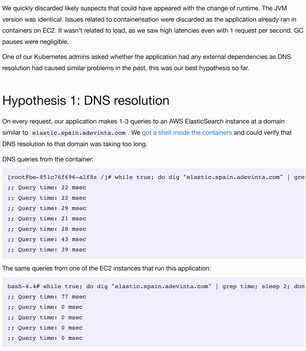
***Update**: it looks this post has gotten way more attention than I anticipated. I've seen / received feedback that the title is misleading and some people get dissapointed. I see why, so at the risk of spoiling the surprise, let me clarify what this is about before starting. As we migrate teams over to Kubernetes, I'm observing that every time someone has an issue, like "latency went up after migrating", there is a knee-jerk reaction that "Kubernetes is to blame", and investigations almost invariably result in "well, not really". This post exemplifies one of these cases. The title is what our dev said, by the end you'll see it wasn't Kubernetes at all. There are no break through revelations to be learned about Kubernetes here, but I think there are good lessons about complex systems.*

Last week my team was busy with the migration of one microservice to our central platform, which bundles CI/CD, a Kubernetes based runtime, metrics and other goodies. This exercise is meant to provide a blueprint to accelerate moving a fleet of ~150 more in the coming months, all of which power some of the main online marketplaces in Spain (Infojobs, Fotocasa, etc.)

After we had the application deployed in Kubernetes and routed some production traffic to it, things started looking worrisome. Request latencies in the Kubernetes deployment were up to x10 higher than on EC2. Unless we found a solution this would not only become a hard blocker to migrate this microservice, but could also jettison the entire project.

Why is latency so much higher in Kubernetes than EC2?

To pinpoint the bottleneck we collected metrics for the entire request path. The architecture is simple, an API Gateway (Zuul) that proxies requests to the microservice instances in EC2 or Kubernetes. In Kubernetes, we use the NGINX Ingress controller and backends are ordinary Deployment objects running a JVM application based in Spring.



The problem seemed to be upstream latency at the backend (I marked the connection with "xx" in the graph). When the application as deployed in EC2 it took ~20ms to respond. In Kubernetes it was taking 100-200 ms.

We quickly discarded likely suspects that could have appeared with the change of runtime. The JVM version was identical. Issues related to containerisation were discarded as the application already ran in containers on EC2. It wasn't related to load, as we saw high latencies even with 1 request per second. GC pauses were negligible.

One of our Kubernetes admins asked whether the application had any external dependencies as DNS resolution had caused similar problems in the past, this was our best hypothesis so far.

Hypothesis 1: DNS resolution

On every request, our application makes 1-3 queries to an AWS ElasticSearch instance at a domain similar to `elastic.spain.adevinta.com`. We got a shell inside the containers and could verify that DNS resolution to that domain was taking too long.

DNS queries from the container:

```
[root@be-851c76f696-alf8z /]# while true; do dig "elastic.spain.adevinta.com" | gre
;; Query time: 22 msec
;; Query time: 22 msec
;; Query time: 29 msec
;; Query time: 21 msec
;; Query time: 28 msec
;; Query time: 43 msec
;; Query time: 39 msec
```

The same queries from one of the EC2 instances that run this application:

```
bash-4.4# while true; do dig "elastic.spain.adevinta.com" | grep time; sleep 2; don
;; Query time: 77 msec
;; Query time: 0 msec
;; Query time: 0 msec
;; Query time: 0 msec
;; Query time: 0 msec
```

Given the ~30ms resolution time, it seemed clear that our application was adding DNS resolution overhead talking to its ElasticSearch.

But this was strange for two reasons:

- We already have many applications in Kubernetes that communicate with AWS resources and don't suffer such high latencies. Whatever the cause it had to be be specific to this one.
- We know that the JVM implements in-memory DNS caching. Looking at the configuration in these images, the TTL configured at `$JAVA_HOME/jre/lib/security/java.security` and it was set to `networkaddress.cache.ttl = 10`. The JVM should be caching all DNS queries for 10 seconds.

To confirm the DNS hypothesis we decided to avoid DNS resolution and see if the problem disappeared. Our first attempt was to have the application talk directly to the Elasticsearch IP, rather than the domain name. This required changing code and a new deploy so instead we simply added a line mapping the domain to its IP in `/etc/hosts`:

```
34.55.5.111 elastic.spain.adevinta.com
```

This way the container would resolve the IP almost instantly. We did observe a relative improvement, but nowhere near our target latency. Even though DNS resolution was too high, there real cause was still hidden.

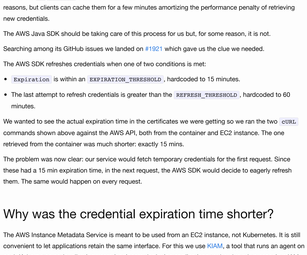
Network plumbing

We decided to `tcpdump` from the container so we could see exactly what was going on with the network.

```
[root@be-851c76f696-alf8z /]# tcpdump -leni any -w capture.pcap
```

We then sent a few requests and downloaded the capture (`kubectl cp my-service:/capture.pcap capture.pcap`) to inspect it with Wireshark.

There was nothing suspicious with DNS queries (except a detail I'll mention later) But something in the way our service handled each request seemed strange. Below is a screenshot of the capture, showing the reception of a request until the start of the response.



The packet numbers are shown in the first column. I coloured the different TCP streams for clarity.

The green stream starting at packet 328 shows the client (172.17.22.150) opened a TCP connection to our container (172.17.36.147). After the initial handshake (328-330), packet 331 brought an `HTTP GET /v1/..` the incoming request to our service. The whole process was over in 1ms.

The grey stream from packet 339 shows that our service sent an HTTP request to the Elasticsearch instance (you don't see the TCP handshake because it was using an existing TCP connection.) This took 18ms.

So far this makes sense, and the times roughly fit in the overall response latencies we expected (~20-30ms measured from the client).

But inbetween both exchanges, the blue section consumes 86ms. What was going on there? At packet 333, our service sent an HTTP GET request to `/latest/meta-data/iam/security-credentials`, and right after, on the same TCP connection, another GET to `/latest/meta-data/iam/security-credentials/arn:...`.

We verified that this was happening on every single request for the entire trace. DNS resolution is indeed a bit slower in our containers (the explanation is interesting, I will leave that for another post). But the actual cause for the high latencies were queries to the AWS Instance Metadata service on every single request.

Hypothesis 2: rogue calls to AWS

Both endpoints are part of the AWS Instance Metadata API. This service is used from our microservice during reads from Elasticsearch. The two calls are a basic authorisation workflow. The endpoint queried in the first request yields the IAM role associated to the instance.

```
/ # curl http://169.254.169.254/latest/meta-data/iam/security-credentials/
arn:aws:iam::<account_id>:role/some_role
```

The second request queries the second endpoint for temporary credentials for that instance:

```
/ # curl http://169.254.169.254/latest/meta-data/iam/security-credentials/arn:aws:i
{
    "Code" : "Success",
    "LastUpdated" : "2012-04-26T16:39:16Z",
    "Type" : "AWS-HMAC",
    "AccessKeyId" : "ASIAI05FONS7EXAMPLE",
    "SecretAccessKey" : "wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY",
    "Token" : "token",
    "Expiration" : "2017-05-17T15:09:54Z"
}
```

The client is able to use them for a short period of time, and is expected to retrieve new ones periodically (before the `Expiration` deadline). The model is simple: AWS rotates temporary keys often for security reasons, but clients can cache them for a few minutes amortizing the performance penalty of retrieving new credentials.

The AWS Java SDK should be taking care of this process for us but, for some reason, it is not.

Searching among its GitHub issues we landed on #1921 which gave us the clue we needed.

The AWS SDK refreshes credentials when one of two conditions is met:

- `Expiration` is within an `EXPIRATION_THRESHOLD`, hardcoded to 15 minutes.
- The last attempt to refresh credentials is greater than the `REFRESH_THRESHOLD`, hardcoded to 60 minutes.

We wanted to see the actual expiration time in the certificates we were getting so we ran the two `cURL` commands shown above against the AWS API, both from the container and EC2 instance. The one retrieved from the container was much shorter: exactly 15 mins.

The problem was now clear: our service would fetch temporary credentials for the first request. Since these had a 15 min expiration time, in the next request, the AWS SDK would decide to eagerly refresh them. The same would happen on every request.

Why was the credential expiration time shorter?

The AWS Instance Metadata Service is meant to be used from an EC2 instance, not Kubernetes. It is still convenient to let applications retain the same interface. For this we use KIAM, a tool that runs an agent on each Kubernetes node, allowing users (engineers deploying applications to the cluster) to associate IAM roles to Pod containers as if they were EC2 instances. It works by intercepting calls to the AWS Instance Metadata service and serving them from its own cache, pre-fetched from AWS. From the point of view of the application, there is no difference with running in EC2.

KIAM happens to provide short-lived credentials to Pods, which makes sense as it's fair to assume that the average lifetime of a Pod is shorter than EC2 instances. The default is precisely 15 min.

But if you put both defaults together, you have a problem. Each certificate provided to the application has a 15 min expiration time. The AWS Java SDK will force refreshing any certificate with less than 15 min expiration time left.

The result is that every request will be forced to refresh the temporary certificate, which requires two calls to the AWS API that add a huge latency penalty to each request. We later found a feature request in the AWS Java SDK that mentions this same issue.

The fix was easy. We reconfigured KIAM to request credentials with a longer expiration period. Once this change was applied, requests started being served without involving the AWS Metadata service and returned to an even lower latency than in EC2.

Takeaways

In our experience with these migrations, one of the most frequent sources of problems is not bugs in Kubernetes or other pieces of the platform. It isn't either about fundamental flaws in the microservices we're migrating. **Problems often appear just because we put some pieces together in the first place**.

We are blending complex systems that had never interacted together before with the expectation that they collaborate forming a single, larger system. More moving pieces, more surface for failures, more entropy.

In this case, our high latency wasn't the result of bugs or bad decisions in Kubernetes, KIAM, the AWS Java SDK, or our microservice. It was a behaviour resulting from the combination of two independent defaults, in KIAM and the AWS Java SDK. Both choices make sense in isolation: an eager credential refresh policy in the AWS Java SDK, and the lower default expiration in KIAM. It is when they come together that the results are pathological. **Two independently sound decisions don't necessarily make sense together**.

To get notifications for new posts, subscribe to the RSS feed or follow me on Twitter. Here is the log of older posts.