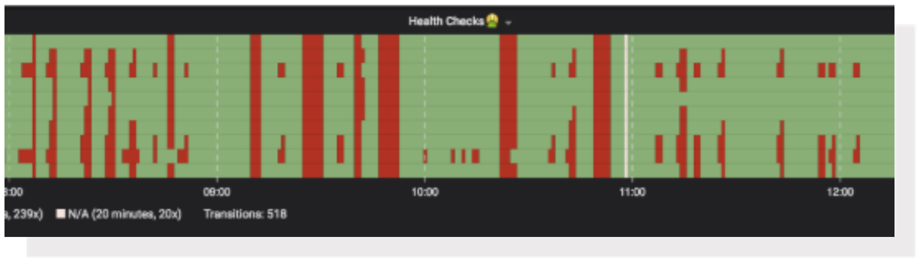
January 8, 2019

Alert Fatigue: Finding What's Wrong when



the Helpers are Hurting...

Lauren Billington

Here at Aspen Mesh, we're committed to using our own application in production. We can't expect you to use it if we're not, right? Plus, we kind of needed a service mesh!

Besides using the platform, we needed to be sure that our platform (composed of a bunch of microservices) was operating properly and helping our users be successful. The best idea to meet our basic needs (for now) was to make another microservice that impersonated a user and sent us alerts when parts of our platform were unavailable or failed in some defined way. When implemented, it rarely fired, and if it did, there was a good reason, and it helped us to quickly address the reason for the alert.

Fast-forward to KubeCon in December 2018. The interest in Aspen Mesh was almost overwhelming, and though we had no user complaints, and no real observations of trouble, our Slack alerts were going crazy. We muted the threads, some people left the channel. Still, we figured there had to be something going on and we needed to look into it.

We'll take you through the questions we had, and how we answered them, as we figured out what was going on. No spoilers, but the answer was unexpected, and the set of unexpected events showed us that there is a new alert we need to write!

Measuring Application Health: Be a User

How did we settle on impersonating users as a way to address application health? The idea of "application health" can be measured many different ways.

At the most granular level, you can ask "is any application emitting a log message? Are any requests not getting through?" But in the real world, we've found the answer is always "yes", which makes this level of granularity not terribly helpful. Even in the best case, users may disconnect their laptops in the middle of a request; no human can handle the deluge at this level. This level may be very noisy, but it's also the most actionable.

At the other end, you could just check and see if your users are calling support and screaming that the site is down. Low noise, but also low actionability – when "the site is down" you don't know when to start, and it's way too late to start intervening on behalf of your users. Plus, any customer intervention at that point requires much more investment (time, personal attention, baked goods…) to restore that lost trust. We needed a middle ground.

We know what our product is supposed to do. So a solid middle ground is to "be a user". We implemented a health check service that actively pretends to be a user of

Aspen Mesh and does the things a user would normally do: log in to the GUI, check the service graph, tweak an Aspen Mesh Experiment, check for some Istio Vet notes, etc.

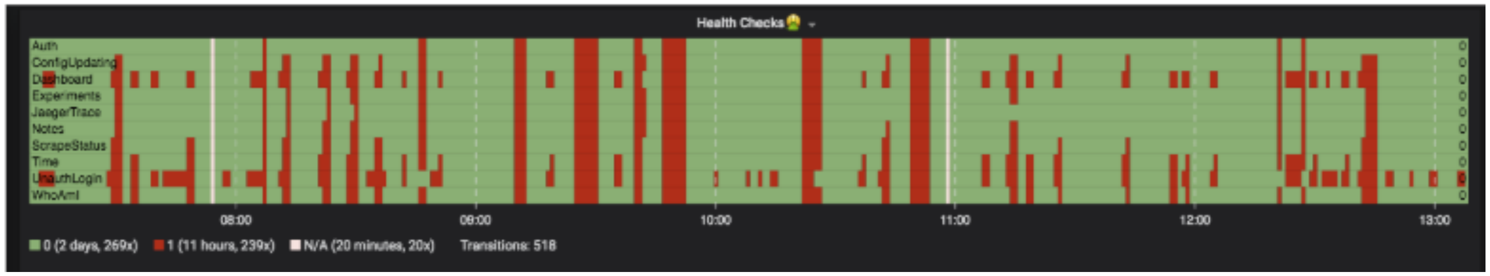
If this checker fails, we hope this is a high-fidelity signal that a user would experience the same failure. It's not as actionable as a log message in a particular microservice — if "the service graph fails", the fault could be in any number of components. But it's a good place to start.

Several months ago, we implemented this health checker that we could monitor via Prometheus and Grafana, and then send alerts to a Slack channel when a threshold is exceeded. It was exciting when we first launched it as we knew it would give us a deeper understanding of our microservices. Luckily, we have a few geniuses on staff and were quite pleased to find that everything was just peachy. We rarely got alerts, and when we did, we were able to trace the issues back to oddities like a dependency that restarted every Wednesday at 11 PM...

...But then we went to Kubecon and our alerts started firing every few minutes. Our sense of urgency quickly waned towards mild annoyance. Still, we wrote the code, and were pretty sure that the health checker wasn't sending us bogus alerts, so we needed to figure out what it was trying to tell us.

Round I: Initial Questions

Grafana showed this:



Not great.

We created a goal:

"*As an Aspen Mesh engineer, I only want to get alerts in #prod-alerts for real issues, and those alerts should occur infrequently so that I don't get alert fatigue.*"

One of our senior devs started a shared-doc with a list of his initial thoughts since I wasn't sure where to start:

Andrew's List of Questions:

- *Is the problem that the platform is actually having these errors, or is it on the platform monitoring side? For example, maybe the platform monitor can't connect "out" to the platform sometimes, but our users are happy as can be.*
- This hypothesis can be falsified by running the platform-health locally and confirming that we see similar errors.
- Anything really bad in the platform health logs? Is the platform-health pod restarting?
- If we curl the /metrics endpoint on the platform-health pod ourselves, do we see errors? Or is it "synthetic" (like Prometheus can't scrape the pod)?

- Is the platform-health pod running with an Istio sidecar?
- Is the problem correlated in time with any other issues?
- Some of these are really simple – the “unauth login” page is just getting back a static webpage (after going through some of our platform components). Why are these affected?

This doc became a sort of journal of things we tried, questions answered, screenshots. It was actually really helpful.

Initial Answers

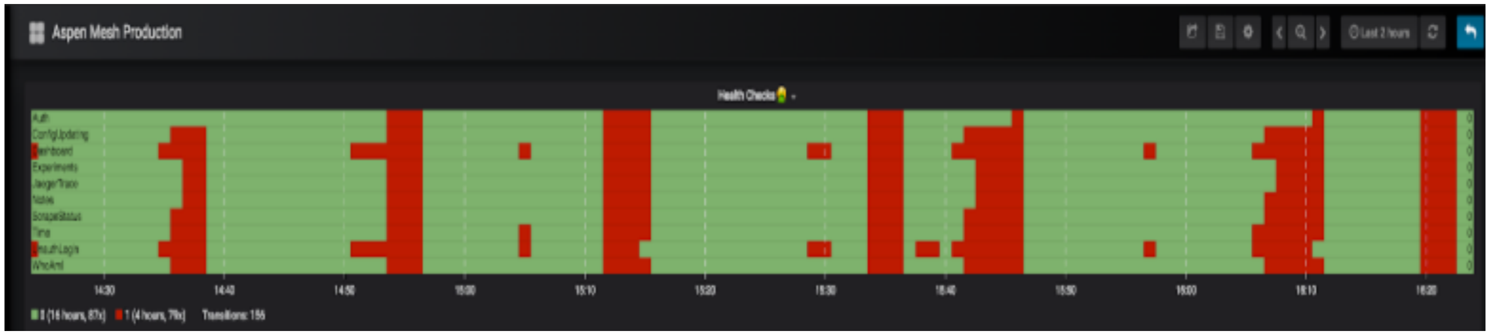
The important results were that the Health pod was healthy, had been live since the last time we updated production. It (correctly) had no sidecar. Running the service locally produced no errors. There weren’t any obvious issues with the pods.

Next we checked the logs for the pod. We have CloudWatch set up for this service, so I did some spot-checking to cover a few days worth of alerting. Remember that question of granularity? We implemented helpful error and logs messages in our code, and let Cloudwatch collect our logs so that we can dig in for info when needed (it does some other cool stuff I’ll get to in a moment).

By spot-checking logs for different times and dates, I found that the majority of the failed requests were timeouts, and occasionally our client was totally unavailable. But there didn’t seem to be a pattern of when the client was missing and when the check failed due to timeouts. So at least we now knew that the Alerts were really reporting alerts. But why was it happening so often, for such a short period of time, and why was it that our users weren’t reporting any issues?

Round II: Maybe Patterns?

Next, I decided to zoom in on a 2-hour section of the alert graph to see what patterns I could find, and made some notes for Andrew.



Lauren's List of Possibly interesting things:

- *The Health service runs 1x/min.*
- There are times when there are only failures from the login page and dashboard that last about 1-2 minutes, then they succeed for anywhere from 2 to 10+ minutes.
- There are times when everything fails for about 2-5 minutes at a time.
- There are times when some things fail and then pass in a staggered way — this could indicate an order in which failures occur or resolve.
- There are some points where there are Auth failures (a failure means ctx.HttpClient() == nil), but it's not every time. There are times when the other pings fail but we have a valid working client. (See ~14:35pm, ~15:40, ~16:05)
- There are 2 cases where we did not have the ability to reach the login page, all the other things failed, then the first thing to recover is the login page. (see ~15:15 and

~16:10ish)

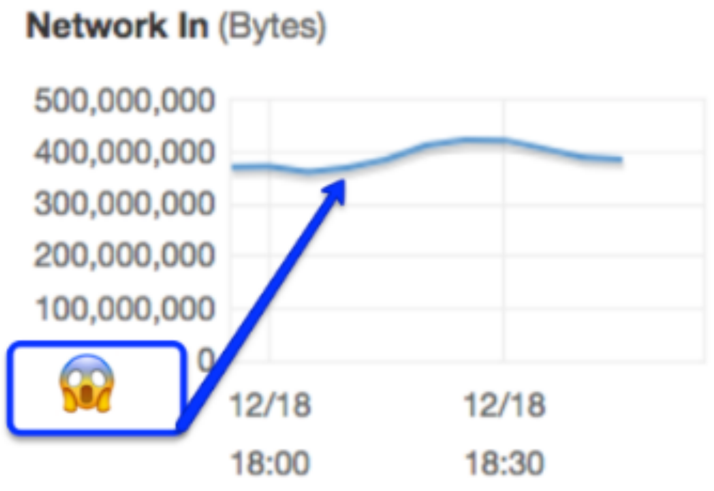
Andrew’s thoughts on this were:

- *Around 14:25, 15:05,15:28, 15:56 it looks like specifically Dashboard and UnauthLogin fail but everything else succeeds. That’s interesting because Dashboard and UnauthLogin are both pretty “boring” services, and they both go through an apiserver.*
- In a dashboard/unauth failure case, is the request getting all the way to the dashboard?
- Can we change the HTTP UserAgent for the health monitor, which may make it easier to tell in logs if a request is from the health monitor?

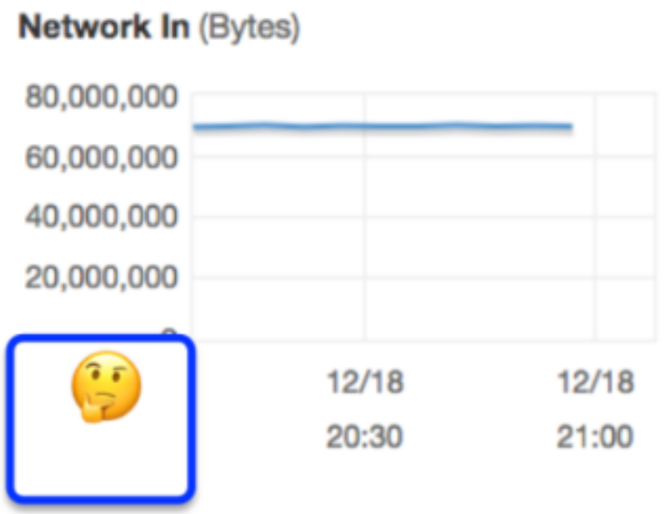
Round III: Jimmy the Jamboard

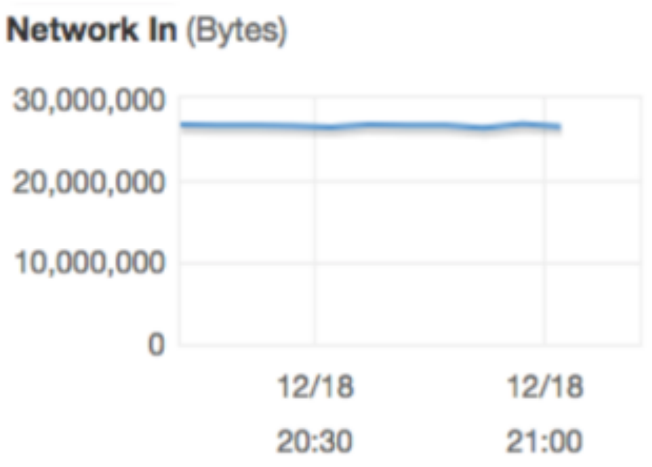
We met up the next day to Jamboard the problem (I dubbed the Jamboard “Jimmy”… we’ll see if it sticks), and Andrew hypothesized that there might be something wrong with the node the Health pod was in — that it might be hitting a limit, but it could be a resource limit, or a traffic limit.

Finding the nodeName for a pod took a bit. When using AWS, the pod’s nodeName is the AWS Private DNS Name. But once I had that, I was back in the AWS platform. By selecting the Private DNS Name, I checked the instance’s Monitoring tab for CloudWatch data. In addition to graphs of CPU usage, read/write data, it provides graphs of Network In/Out bytes per minute.   This node was handling about 380MB/min In and 20MB/min Out, which seemed pretty high.



I checked some other nodes running our main application and the traffic was surprisingly lower.





One bummer with Cloudwatch for checking individual nodes was the inability to compare multiple nodes in one window, or even multiple browser windows at once (I could only have 1 selected at a time, regardless of window) — hence the screenshots. Last, I double-checked all the nodes in question for a matching 2 hour time frame to be sure I hadn't been making assumptions based on an isolated incident.

We knew that the Health service itself was tiny, so what was going on with that node? And, if we were hitting a throughput limit, what was that limit? AWS doesn't do a great job of making that info available, so Andrew found an article where the author was running his own tests on AWS Instance Types. Ours capped at 0.45GB in the author's tests, so this node could certainly be hitting a network limit every few minutes.

Finally Getting Somewhere!

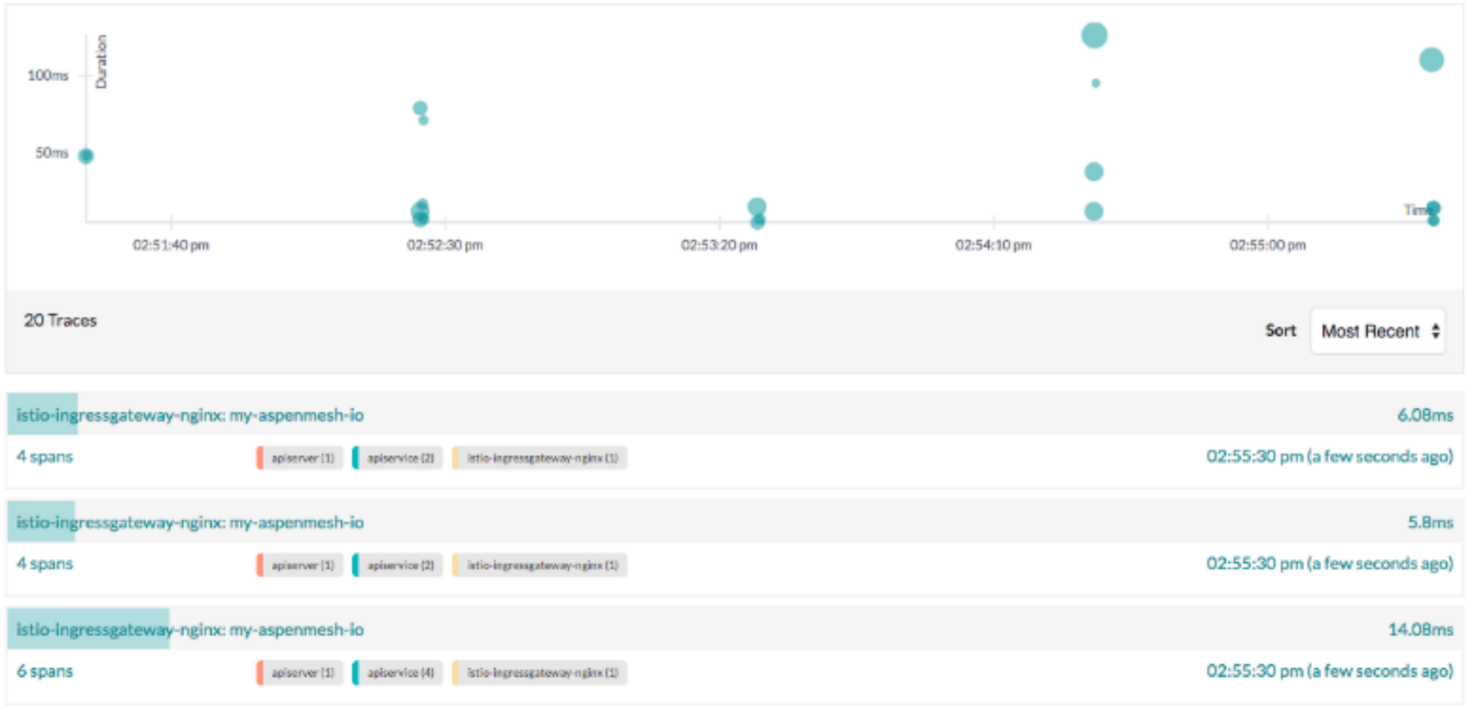
Next, it was back to the terminal to see what was going on in that node. I learned another new thing: printing out a list of pods by filtering for node name! (Thanks again, Andrew!)

```
kubectl get pods --all-namespaces --field-selector spec.nodeName=
<nodenamehere> -o wide
```

It turns out that this node was running a ton of per-user microservices that monitor users' clusters so they can get real-time data. While that seemed the likely culprit, we had noticed a few other deficiencies. We focused on convincing ourselves we had the right root cause, and also took advantage of the downtime while running our experiment to fix some "nice-to-haves".

Nice-to-have #1: There were two places in our Health Checker code where the closing the request was missing, so we checked to see if there was an overwhelming number of sockets that were open causing an ephemeral source port starvation. That didn't seem to be the case, but since we found the bugs, we fixed them anyway.

Nice-to-have #2: We also changed the User-Agent string that is sent with requests. This makes it easy to select only the health monitor requests in log messages or in our distributed traces. I recommend setting the user-agent for your different services early on in a project so you're not staring at a sea of "Go-http-client/1.1"s when debugging.



20 Traces

istio-ingressgateway-nginx: my-aspenmesh-io 6.08ms
4 spans apiserver (1) apiservice (2) istio-ingressgateway-nginx (1) 02:55:30 pm (a few seconds ago)

istio-ingressgateway-nginx: my-aspenmesh-io 5.8ms
4 spans apiserver (1) apiservice (2) istio-ingressgateway-nginx (1) 02:55:30 pm (a few seconds ago)

istio-ingressgateway-nginx: my-aspenmesh-io 14.08ms
6 spans apiserver (1) apiservice (4) istio-ingressgateway-nginx (1) 02:55:30 pm (a few seconds ago)

After all of this, it seemed the issue was that our little Health pod was living in a node with a ton of hungry services causing the node to reach its network bandwidth limit. The easiest way to test this was to either kill the pod and allow "Chance" to start it on a new node (hoping it wasn't a busy node), or to go through setting taints or tolerations to exercise some control on where the pod restarted. It seemed faster to just go the route of Chance, and then kill it again if the gamble was bad.

With luck, it started on a mostly bare node and we went several hours without an alert. Using Jaeger, we were able to isolate traces with the new user-agent string (via Tags) to see that we weren't having the same kinds of issues. Just to check, we killed it again and let it restart (luck landed it on a moderately busy node this time) and the service continued to thrive. Jaeger still showed we were good.

Of course, relying on Chance to schedule favorably is not a solution, but it convinced us that we understood what was going on (intra-cluster network saturation), so now we can go off and get a handle on that…

I think we've all learned something today…

So the culprit ended up being a node with too much traffic that hit limits which blocked the little service from completing its checks, so the checker did its job and reported errors until we listened. We now know we need to address node traffic in production – maybe using taints and tolerations, maybe with a new service deployed to each node, or maybe something else. But our little Health checker alerted us to a bigger and unforeseen issue, and that's pretty cool. Also, now when we get alerts in Slack, we care because we've fought the alert fatigue and won (at least for today).

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

POST COMMENT

< Back to Blog

Let us take the burden out of managing microservices

GET ASPEN MESH

hello@aspenmesh.io