Sailing with the Istio through the shallow water

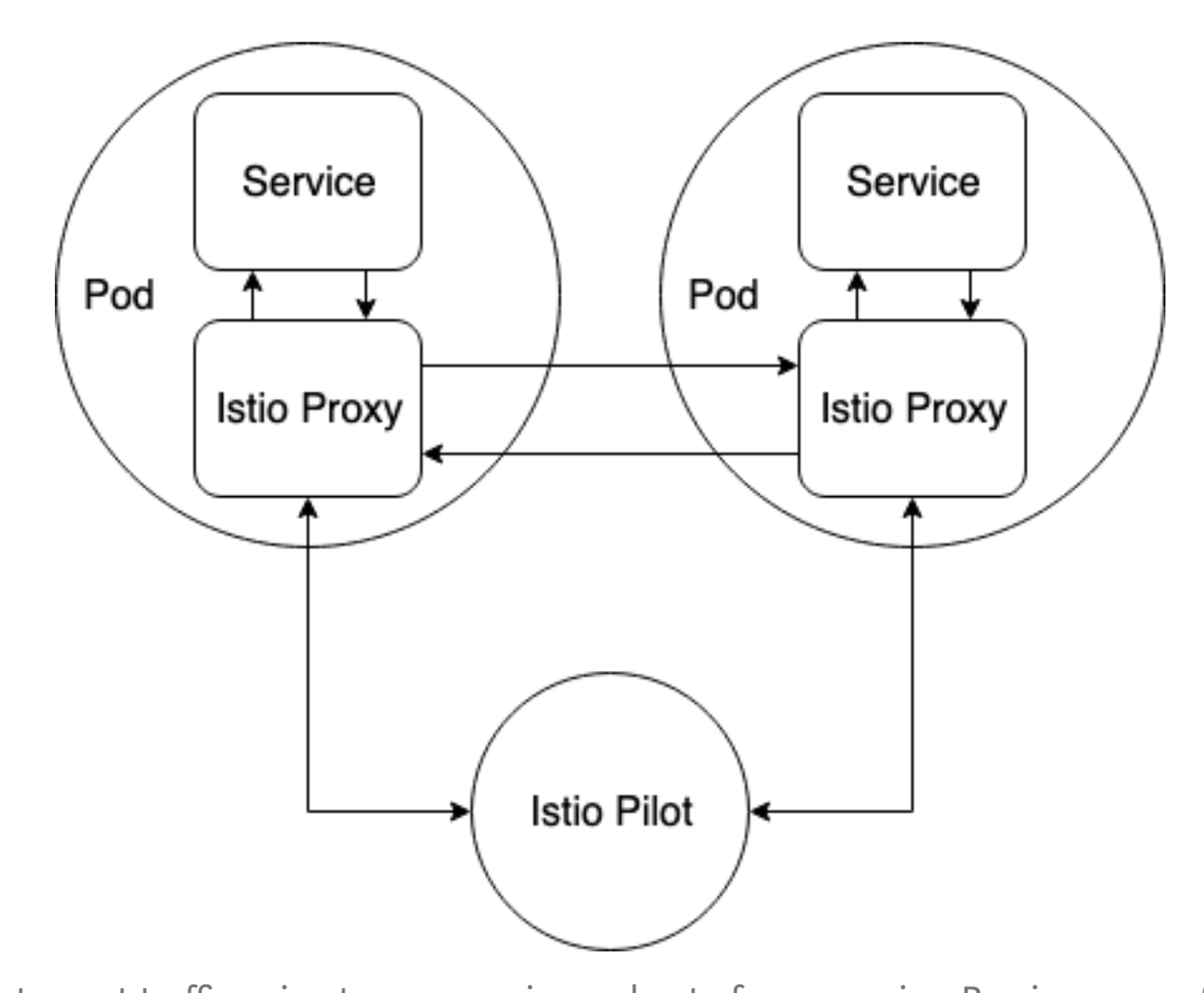
Jakub Kulich Follow
Oct 29, 2019 · 7 min read

When someone talks about Istio, it's just bells and whistles, but nobody talks about difficulties that may arise during the integration into the existing project. I'm a software engineer @Exponea, and I was responsible for integration of the Istio to our infrastructure. I've postponed the integration because we weren't able to take the last step that was deploying the Istio to the production. Why? Let me describe the whole journey from the beginning.

Our situation is that we are running a big part of our infrastructure on the Google Kubernetes Engine (GKE). We expect from Istio to simplify our application layer, give us more insights into our traffic, and increase the security of our application. We want to spend as little time as possible with managing Istio, so we are heading to try out the managed Istio on GKE. After a few experiments, we've found out that we're not able to use the managed Istio on GKE. It was a big disappointment because it would solve some troubles (like the most managed services) for us. Managed Istio on GKE didn't support any configuration besides the mTLS mode. We didn't want anything super-advanced, just options available in the Helm chart provided by Istio.

Brace yourselves, the adventure begins. We've deployed Istio to the development environment. I explain the production deployment at the end of this article.

An application cannot start, nearly all pods are in the crash loop back-off state. Without debugging, I suspect that port names don't have the right prefix. Istio knows what type of traffic flows through port based on its name (HTTP, gRPC, TCP, …). Therefore, if port names aren't named correctly, your traffic may not work in some cases — for example, if you tag port as an HTTP, but it's gRPC. I went through all the services and fixed all namings. Finally, most of the pods were able to start and run. If you are not familiar how Istio works, and you are not sure why this happens, check the diagram below.



Istio proxies intercept traffic going to your service and out of your service. Proxies are configured by Istio Pilot.

After a few days of debugging, new issues start to pop up. Our jobs are not finishing because Istio sidecars keep running after the job has finished. Reason? As you may know, Kubernetes Jobs run one-shot programs or scripts. When the program in the job exits, then Kubernetes considers the job as finished. On the contrary, Istio proxy is a server that runs forever. Istio proxy running causes the job to hang.

Our solution to this problem is disabling Istio proxy injection. Jobs without proxies aren't a problem for us, as we don't mutual TLS in the first stage of integration, and sidecar in jobs wouldn't bring us an advantage. Istio provided a workaround for this problem in the latest release, and you can shutdown proxy sidecar by calling its API after your job finishes.

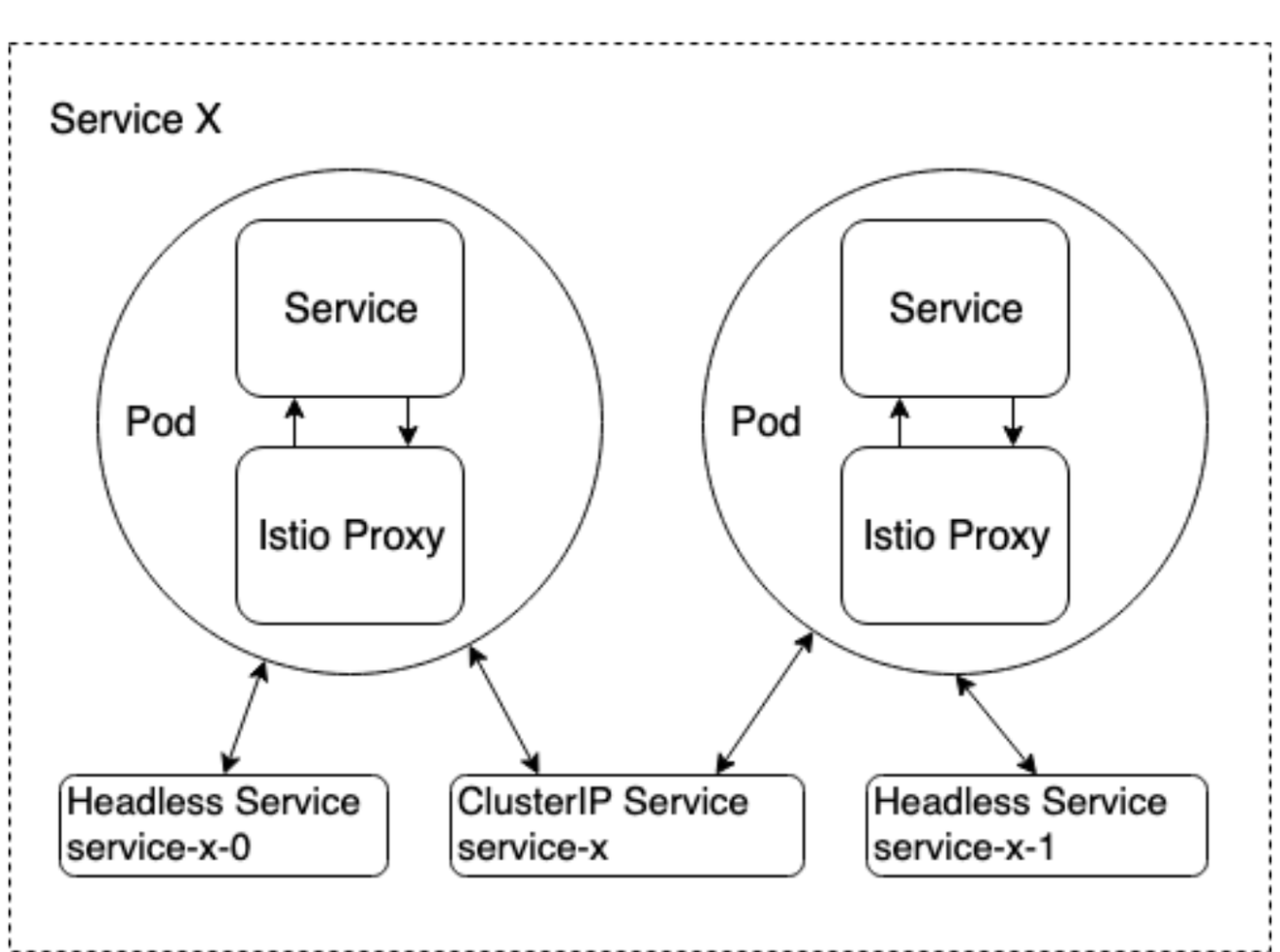
Now, things are getting exciting. Problems hide in-depth, and it's not easy to debug them. The connections between services were braking, and we had no idea why. We observed that this happened when services were connecting to services setup as stateful sets. Istio doesn't support stateful sets, but they can be attached to the service mesh using ServiceEntry resources.

```yaml
kind: ServiceEntry
apiVersion: networking.istio.io/v1alpha3
metadata:
  name: my-service
spec:
  hosts:
  - "my-service-0.my-service"
  - "my-service-1.my-service"
  ports:
  - number: 80
    name: http
    protocol: http
```

ServiceEntry can be used for adding StatefulSet into the Istio service mesh

Unfortunately, there is a bug in a service discovery of Istio, which doesn't update IP addresses of headless services with ServiceEntry. As a result, if you recreate pod published with headless service, then services connecting to this service would connect to the old IP address, which doesn't exist. I've tried to fix this issue, but unfortunately, I wasn't successful, so instead, I've opened the issue. Our temporary fix for this is disabling sidecar injection for stateful services.

Another related problem to headless services were gRPC services. Some of them were using headless services to implement client-side load balancing. As you know, gRPC handles load balancing for you, so we had to add the normal (clusterIP) services besides the headless services to ensure the backward compatibility. If you ask yourself why you need this kind of backward compatibility, then the answer is that we've enabled Istio just for the part of our services. Therefore, some services would use client-side load balancing, and some services would leverage envoy load balancing.



Setup that enables support of both — client-side and Istio gRPC load balancing. Services connecting to service X running without Istio connect using client-side gRPC load balancing with backends "service-x-0" and "service-x-1". Services with Istio connect just to "service-x" backend and Istio ensures that traffic will be load balanced across all the pods.

The last problem (being optimistic here) that we've found in the development environment is the handling of graceful shutdowns by proxy sidecars. The sidecar exits gracefully upon receiving the signal from Kubernetes. However, this becomes the obstacle when your service has a more prolonged grace shutdown period and needs to finish its work (for example, send a batch of emails). Consequently, your service won't be able to do so because the proxy sidecar is already shutdown.

Again, we've encountered a known issue. We've taken inspiration from the comments and edited a sidecar injection template. The injection template adds preStop hook to the proxy container when pod has specific annotation. PreStop hook checks liveness probe endpoint and allow to exit the sidecar proxy when the liveness probe is no longer working. The solution worked better for cases where service made several HTTP calls, and the number of outgoing connections could fall to 0 during the graceful shutdown period.

```yaml
...
  ports:
  - containerPort: 15090
    protocol: TCP
    name: http-envoy-prom
  # Patch of sidecar injection template starts here
  {{ if (isset .ObjectMeta.Annotations `sidecar.istio.io/graceShutdownHandler`) -}}
  lifecycle:
    preStop:
      exec:
        command:
        - /bin/sh
        - -c
        - curl -f {{ index .ObjectMeta.Annotations `sidecar.istio.io/graceShutdownHandler` }}; while [ $? -eq 0 ]; do sleep 1; curl -f {{ index .ObjectMeta.Annotations `sidecar.istio.io/graceShutdownHandler` }}; done;
  {{ end -}}
  # Patch ends here
  args:
  - proxy
 ...
```

Snippet shows how we patched the sidecar injection template. If pod is annotated with "sidecar.istio.io/graceShutdownHandler", then we inject preStop hook to the Istio sidecar. The URL in the annotation's value is checked until it stops to respond. Thereafter, istio proxy is stopped.

It seems that all issues were solved, and we're going to figure out how to deploy Istio to the production cluster. We're getting to the most critical part of the integration because Istio is the single point of the failure. As I've mentioned above, we had to exclude the managed Istio on the GKE from the consideration. The other two options were deploying the Istio manually or deploy Istio using the operator from Banzai Cloud.

Our challenges were upgrades of Istio, its stability, and security. Let me start with the upgrade. Upgrade of the control plane works without problems in the latest releases, but the upgrade of sidecars is much worse. Some time ago, official Istio documentation had just link to the gist with a bash script, which looks like a joke. The process of upgrade isn't better now, but Istio replaced the link to the bash script by command introduced to the recent kubectl release, which can make a rolling restart of the deployment. Unfortunately, the Istio operator couldn't handle upgrades of the data plane as well. Thus we had to find out how to upgrade the data plane.

Our final solution for upgrading the data plane is an injection of the annotation "istio.io/version" which holds a version of the Istio control plane. As a result, we need to change the version of the Istio in our deployment tool, and it'll reinject new annotation to all pods at once.

The last task is to set a deployment strategy for Istio and both the control plane and proxies. We knew that we want to divide the control plane for each tenant, so Istio isn't a single point of the failure. The second reason why we wanted to divide the control plane is the Istio upgrade — recreating all the pods in the cluster would kill the Kubernetes masters. That's not what we want.

We've relied on the blog post from the official Istio blog from 2018. However, we were not able to deploy Istio in this setup, and we've again found out that we aren't alone. Istio made some changes that made Istio undeployable in the multitenant setup. The next checkpoint is the Istio operator from the Banzai cloud, but they have an issue about multitenancy open as well. Guys from Banzai Cloud recommended us to create multiple clusters rather than multitenant setup because they were not successful with this kind of setup.

With all the problems mentioned and the inability to split up the control plane to multiple namespaces, our only remaining strategy for deployment is run Istio just for services that are not mission-critical. Then, integrate other services when issues are solved, and when we are more confident that Istio doesn't cause many troubles in production. Sadly, we didn't find added value in having Istio just in few services. Thus we postponed the integration altogether. The Istio is a great project, and we hope that soon, it will be mature enough even for us.

This is my first blog post, and I appreciate all the feedback you have. Also, if you have any questions, feel free to leave them in a comment.

Kubernetes · Istio · Software Development · Microservices · Distributed Systems

512 claps

WRITTEN BY
Jakub Kulich Follow
DevOps engineer (or kid with cloud Lego)

See responses (10)

More From Medium

The Loading Shimmer! — Dhilip Kumar
A Gentle Introduction to SQL Basics in Python — Randy Macaraeg in Better Programming
How we wrote the Fastest JavaScript UI Frameworks — Ryan Carniato in JavaScript In Plain English
Java Concurrency — TyAnthoney Morrell in TyAnthoney Morrell
One SSL, One Domain with Multiple Services: HAproxy on Kubernetes — Bugra Ozturk
Coding Bedtime Stories: ft. The Spaceship Operator — Theresa Garcia
Streaming Data with Spring Boot RESTful Web Service — Swathi Prasad in The Startup
Left-recursive PEG grammars — Guido van Rossum

Discover Medium
Welcome to a place where words matter. On Medium, smart voices and original ideas take center stage - with no ads in sight. Watch

Make Medium yours
Follow all the topics you care about, and we'll deliver the best stories for you to your homepage and inbox. Explore

Become a member
Get unlimited access to the best stories on Medium — and support writers while you're at it. Just $5/month. Upgrade

Medium About Help Legal