# Istio Service Mesh on Multi-Cluster Kubernetes Environment

Manage microservices running on multiple Kubernetes clusters in a single service mesh

Gaurav Agarwal Follow

May 25 · 6 min read ★

Photo by Tom Chen on Unsplash

Imagine you work for a typical enterprise, with multiple teams working together, delivering separate pieces of software that make up a single application. Your teams follow a microservices architecture and have a widespread infrastructure made of multiple Kubernetes clusters.

As the microservices spread across multiple clusters, you need to architect a solution to manage all microservices centrally. Fortunately, you're using Istio, and providing this solution is just another configuration change.

A service mesh technology like Istio helps you securely discover and connect microservices spread across multiple clusters and environments. This article is a follow-up to "How to Authorise Non-Kubernetes Clients With Istio on Your K8s Cluster." Today let's discuss managing microservices hosted in multiple Kubernetes clusters using Istio.

. . .

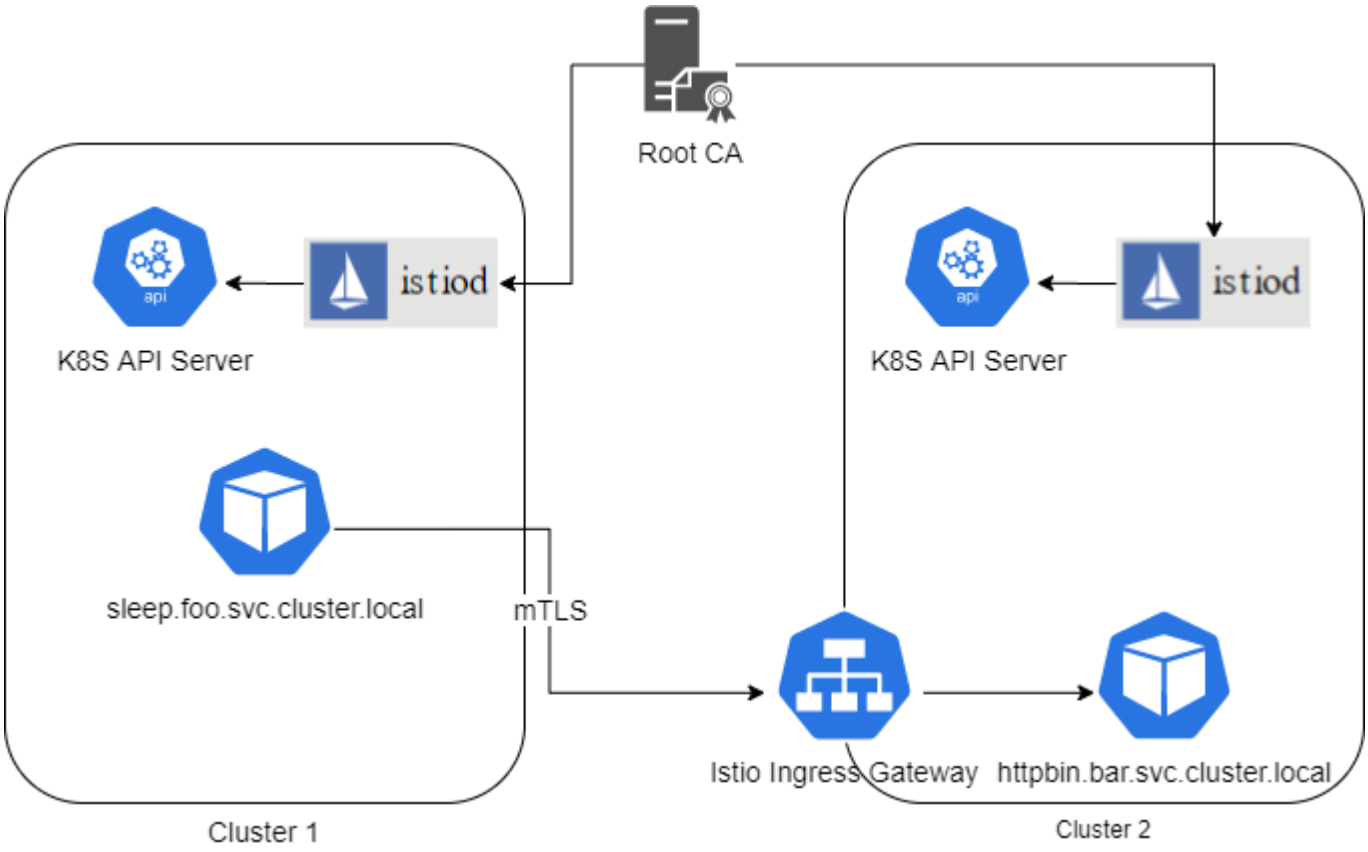
## Architecture

Istio provides cross-cluster service discovery with the following components:

- Istio Core DNS: Every Istio control plane comes with a Core DNS. Istio uses this to discover services defined on the global scope. For example, if a microservice hosted on cluster-1 needs to connect to another microservice hosted on cluster-2, you need to make a global entry for the microservice running on cluster-2 on the Istio Core DNS.
- A Root CA: As Istio requires an mTLS connection between services running on separate clusters, you need to use a shared Root CA to generate intermediate CA

certs for both clusters. That establishes trust between microservices running on different clusters as the intermediate certs share the same Root CA.

- Istio Ingress Gateways: Inter-cluster communications happen via the Ingress Gateways, and there is no direct connection between services. Therefore, you need to ensure that the Ingress Gateway is discoverable and that all clusters can connect to it.




Inter-cluster communication

. . .

## Service Discovery

Istio uses the following steps to foster service discovery:

1. There are identical control planes on all clusters to promote high-availability.
2. The Kube DNS is stubbed to the Istio Core DNS to provide Global Service discovery.

3. The user defines routes to remote services via ServiceEntries in the Istio Core DNS in the format name.namespace.global.
4. The source sidecar uses the global Core DNS entry to route traffic to the target Istio Ingress Gateway.
5. The target Istio Ingress Gateway routes the traffic to the correct microservice pod.

## Prerequisites

This article assumes that you have a basic understanding of how Kubernetes and Istio work. For an introduction to Istio, check out "How to Manage Microservices on Kubernetes With Istio." To follow along in the hands-on demonstration, ensure:

- You have at least two Kubernetes clusters running versions 1.14, 1.15, or 1.16.
- You have privileges to Install and configure Istio within the clusters.
- You have cluster admin access on both Kubernetes clusters.
- The Ingress Gateways are reachable to other clusters via a network load balancer or similar configuration. A flat network is not necessary.

. . .

## Install Istio

On both clusters, install Istio 1.5.2 using the following commands:

```
curl -L https://istio.io/downloadIstio | ISTIO_VERSION=1.5.2 sh -
cd istio-1.5.2
export PATH=$PWD/bin:$PATH
```

As we need to boot our Istio service mesh with intermediate certs generated by a shared root cert, create a secret using the intermediate certs.

For this illustration, let's use the sample certificates provided. However, I would not recommend you use these in production as they're generally available and well-known.

Its best if you use your organization's root CA to generate the intermediate CA certificates.

Run the following commands on both clusters to use the sample certificates. If you're using your certificates, replace the applicable file paths.

```
kubectl create namespace istio-system
kubectl create secret generic cacerts -n istio-system \
    --from-file=samples/certs/ca-cert.pem \
    --from-file=samples/certs/ca-key.pem \
    --from-file=samples/certs/root-cert.pem \
    --from-file=samples/certs/cert-chain.pem

secret/cacerts created
```

As we need to Install Istio for a multi-cluster setup, use the provided Istio multi-cluster gateway manifest file on both clusters.

```
$ istioctl manifest apply -f
install/kubernetes/operator/examples/multicluster/values-istio-
multicluster-gateways.yaml
- Applying manifest for component Base...
✓ Finished applying manifest for component Base.
- Applying manifest for component Pilot...
✓ Finished applying manifest for component Pilot.
  Waiting for resources to become ready...
- Applying manifest for component AddonComponents...
- Applying manifest for component IngressGateways...
- Applying manifest for component EgressGateways...
✓ Finished applying manifest for component EgressGateways.
✓ Finished applying manifest for component IngressGateways.
✓ Finished applying manifest for component AddonComponents.

✓ Installation complete
```

. . .

## Configure Kube DNS

The next step is to federate DNS resolution from the Kube DNS to the Istio Core DNS. Let's configure a stub domain by defining a `ConfigMap` for `kube-dns`. Apply the following manifest on both clusters:

```
$ kubectl apply -f - <<EOF
apiVersion: v1
kind: ConfigMap
metadata:
  name: kube-dns
  namespace: kube-system
data:
  stubDomains: |
    {"global": ["$(kubectl get svc -n istio-system istiocoredns -o jsonpath={.spec.clusterIP})"]
EOF

configmap/kube-dns configured
```

config-map.yaml hosted with ♡ by GitHub view raw

• • •

## Setup Context

As we need to connect with both clusters for different activities, it would make sense to get contexts and store them in environment variables. With that, we can run `kubectl` commands in the cluster of our choice by just including the context in the `kubectl` command.

Get the contexts:

```
$ kubectl config get-contexts
CURRENT   NAME        CLUSTER     AUTHINFO      NAMESPACE
          cluster-1   cluster-1   cluster-1
*         cluster-2   cluster-2   cluster-2
```

Set up environment variables to use the contexts:

```
$ export CTX_CLUSTER1=$(kubectl config view -o
jsonpath='{.contexts[0].name}')
$ export CTX_CLUSTER2=$(kubectl config view -o
jsonpath='{.contexts[1].name}')
$ echo CTX_CLUSTER1 = ${CTX_CLUSTER1}, CTX_CLUSTER2 = ${CTX_CLUSTER2}
CTX_CLUSTER1 = cluster-1, CTX_CLUSTER2 = cluster-2
```

. . .

## Deploy Sample Microservices

Let's start by deploying the sleep microservice on the `foo` namespace on cluster-1:

```
$ kubectl create --context=$CTX_CLUSTER1 namespace foo
namespace/foo created
$ kubectl label --context=$CTX_CLUSTER1 namespace foo istio-
injection=enabled
namespace/foo labeled
$ kubectl apply --context=$CTX_CLUSTER1 -n foo -f
samples/sleep/sleep.yaml
serviceaccount/sleep created
service/sleep created
deployment.apps/sleep created
$ export SLEEP_POD=$(kubectl get --context=$CTX_CLUSTER1 -n foo pod -
l app=sleep -o jsonpath={.items..metadata.name})
```

Now let's deploy the `httpbin` microservice on cluster-2 on the bar namespace:

```
$ kubectl create --context=$CTX_CLUSTER2 namespace bar
namespace/bar created
$ kubectl label --context=$CTX_CLUSTER2 namespace bar istio-
injection=enabled
namespace/bar labeled
$ kubectl apply --context=$CTX_CLUSTER2 -n bar -f
samples/httpbin/httpbin.yaml
serviceaccount/httpbin created
service/httpbin created
deployment.apps/httpbin created
```

. . .

## Creating a Service Entry

Now we need to make a service entry on the Istio Core DNS so that we can discover services on cluster-2 from cluster-1. As all communications occur via the Ingress Gateways, export the cluster-2 Ingress gateway address.

```
export CLUSTER2_GW_ADDR=$(kubectl get --context=$CTX_CLUSTER2 svc --
selector=app=istio-ingressgateway \
    -n istio-system -o
jsonpath='{.items[0].status.loadBalancer.ingress[0].ip}')
```

For services on cluster-1 to discover `httpbin` on cluster-2, we need to create a `ServiceEntry` for `httpbin.bar.global` on cluster-1. That ensures that the Istio Core DNS on cluster-1 can reach the cluster-2 ingress gateway if a service in cluster-1 hits the endpoint `httpbin.bar.global`. The below `yaml`:

- Defines the service domain name in the hosts section.
- Location is `MESH_INTERNAL`, as we need to treat the other service as part of the same service mesh.
- It exposes the service on port 8000.
- We need to provide a unique IP address to this service. The IP address need not be routable, and you can use any address on the 240.0.0.0/16 range.
- We define the cluster-2 ingress gateway address on the endpoint address section so that the requests can route to it. The port 15443 is the SNI-aware Envoy proxy port of the Ingress gateway for routing traffic among pods of the target cluster service.

Apply the `yaml` file:

```
$ kubectl apply --context=$CTX_CLUSTER1 -n foo -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
```

```
  metadata:
    name: httpbin-bar
  spec:
    hosts:
    - httpbin.bar.global
    location: MESH_INTERNAL
    ports:
    - name: http1
      number: 8000
      protocol: http
    resolution: DNS
    addresses:
    - 240.0.0.2
    endpoints:
    - address: ${CLUSTER2_GW_ADDR}
      ports:
        http1: 15443 # Do not change this port value
  EOF

  serviceentry.networking.istio.io/httpbin-bar created
```

httpbin-serviceentry.yaml hosted with ♡ by GitHub view raw

. . .

## Testing the Service

Now let's generate some traffic from the sleep microservice and see if it can reach the `httpbin` microservice running on cluster-2.

```
$ kubectl exec --context=$CTX_CLUSTER1 $SLEEP_POD -n foo -c sleep --
curl -I httpbin.bar.global:8000/headers
  % Total    % Received % Xferd  Average Speed   Time    Time
Time  Current
                                 Dload  Upload   Total   Spent
Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:-
-:--     0HTTP/1.1 200 OK
  0   519    0     0    0     0      0      0 --:--:-- --:--:-- --:-
-:--     0
server: envoy
date: Sat, 16 May 2020 23:03:22 GMT
content-type: application/json
```

```
content-length: 519
access-control-allow-origin: *
access-control-allow-credentials: true
x-envoy-upstream-service-time: 37
```

And we get a successful response! Congratulations, we've successfully configured service discovery between multiple Kubernetes clusters using Istio.

. . .

## Conclusion

Thanks for reading! I hope you enjoyed the article.

This was a demonstration of a highly available Istio service mesh configuration running on Multiple clusters. You can also have a shared control plane configuration, but that isn't recommended for production — if you lose one cluster due to an outage, you also lose control of the running cluster.



Programming Kubernetes Software Engineering Containers DevOps