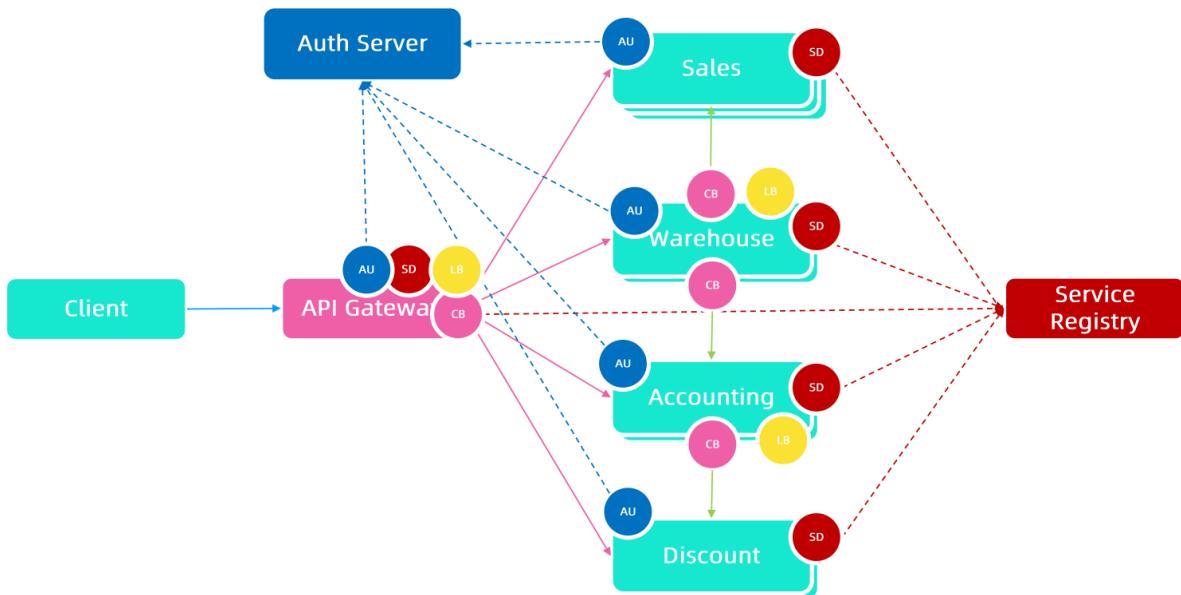


模块12-Istio

微服务架构到服务网格

微服务架构

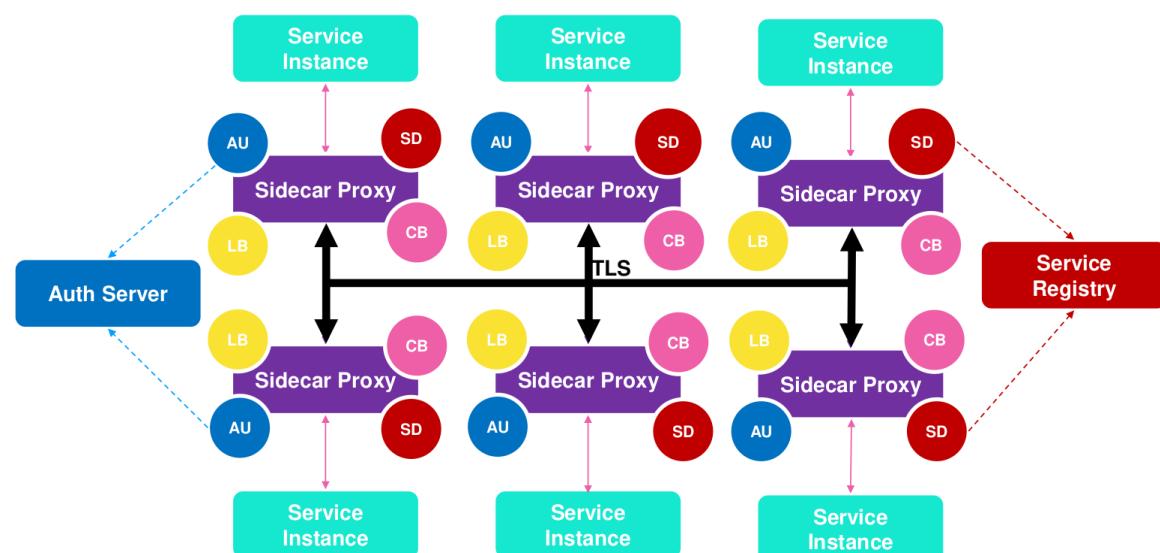


- AU-Auth Server
- SD-Service Discovery
- CB-Circuit Breaker
- LB-Load Balancer

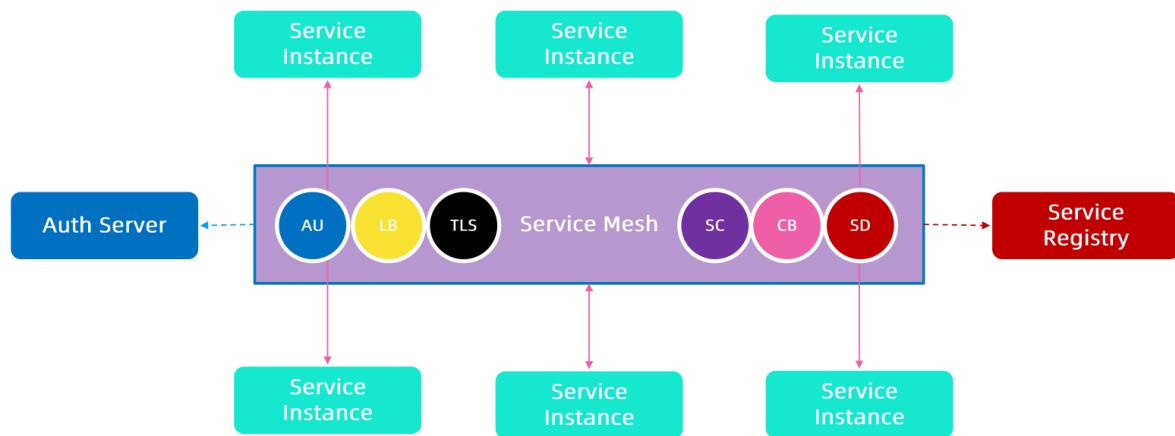
将基础设施和应用分离——服务网格

Sidecar工作原理

承载了微服务中和服务治理相关的设施



Service Mesh



上图中SD指Sidecar。

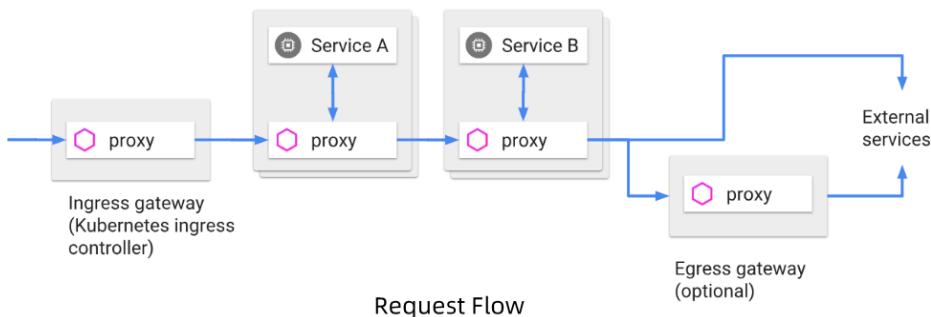
Service Mesh承载的微服务基础设施的功能有：

- **适应性**
 - 熔断
 - 重试
 - 超时处理
 - 失败处理
 - 负载均衡
 - Failover
- **服务发现**
 - 路由
- **安全和访问控制**
 - TLS 和证书管理
 - 黑白名单
- **可观察行**
 - Metrics
 - 监控
 - 分布式日志
 - 分布式 tracing
- **部署**
 - 容器
- **通讯**
 - HTTP
 - WS
 - gRPC
 - TCP

可选方案

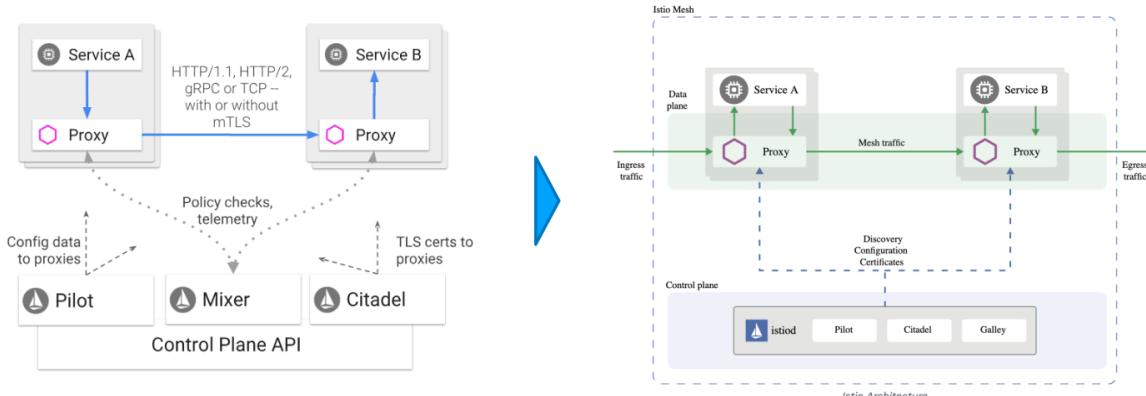
	Istio	Linkerd	Linkerd2	Consul Connect
Model	Sidecar	Node Agent	Sidecar	Sidecar
Platform	Kubernetes	Any	Kubernetes	Any
language	Go	JVM	Go / Rust	Go
Protocol	HTTP1.1 / HTTP2 / gRPC / TCP	HTTP1.1 / HTTP2 / gRPC	HTTP1.1 / HTTP2 / gRPC / TCP	TCP
Default Data Plane	Envoy (supports others)	Native	Native	Native (or Envoy)
Sidecar Injection	Yes	No	No	No
Encryption	Yes	Yes	Experimental	Yes
Traffic Control	label/content based routing, traffic shifting	Dynamic request routing, traffic shifting, per request routing	?	static upstream, prepared query, http api / dns with native integration
Resilience	timeouts, retries, connection pools, outlier detection	timeouts, retries, deadlines, circuit breaking	?	Pluggable
Prometheus Integration	Yes	Yes	Yes	No
Tracing Integration	Jaeger	Zipkin	None	Pluggable
Host to Host auth	Service Accounts	TLS Mutual Auth	No	Consul ACL
Agent Caching	Yes	No	No	Yes
Secure connection outside cluster	No	Yes	No	Yes
Complexity	High	High	Low	Low
Paid Support	No	Yes	Yes	Yes

Istio功能概览



Istio就是一个从K8S中拉取各种Service、pod、Endpoint的配置，然后控制面根据自己的策略生成配置下发到数据面的一个组件。

Istio是管理微服务的，但其自身架构却经历了从微服务回归单体的演变过程。



设计目标

最大化透明度

- Istio 将自身自动注入到服务间所有的网络路径中，运维和开发人员只需付出很少的代价就可以从中受益。
- Istio 使用 Sidecar 代理来捕获流量，并且在尽可能的地方自动编程网络层，以路由流量通过这些代理，而无需对已部署的应用程序代码进行任何改动。
- 在 Kubernetes 中，代理被注入到 Pod 中，通过编写 iptables 规则来捕获流量。注入 Sidecar 代理到 Pod 中并且修改路由规则后，Istio 就能够调解所有流量。
- 所有组件和 API 在设计时都必须考虑性能和规模。

增量

- 预计最大的需求是扩展策略系统，集成其他策略和控制来源，并将网格行为信号传播到其他系统进行分析。策略运行时支持标准扩展机制以便插入到其他服务中。

可移植性

- 将基于 Istio 的服务移植到新环境应该是轻而易举的，而使用 Istio 将一个服务同时部署到多个环境中也是可行的（例如，在多个云上进行冗余部署）。

策略一致性

- 在服务间的 API 调用中，策略的应用使得可以对网格间行为进行全面的控制，但对于无需在 API 级别表达的资源来说，对资源应用策略也同样重要。
- 因此，策略系统作为独特的服务来维护，具有自己的 API，而不是将其放到代理 /sidecar 中，这容许服务根据需要直接与其集成。

数据平面Envoy

主流七层代理的比较

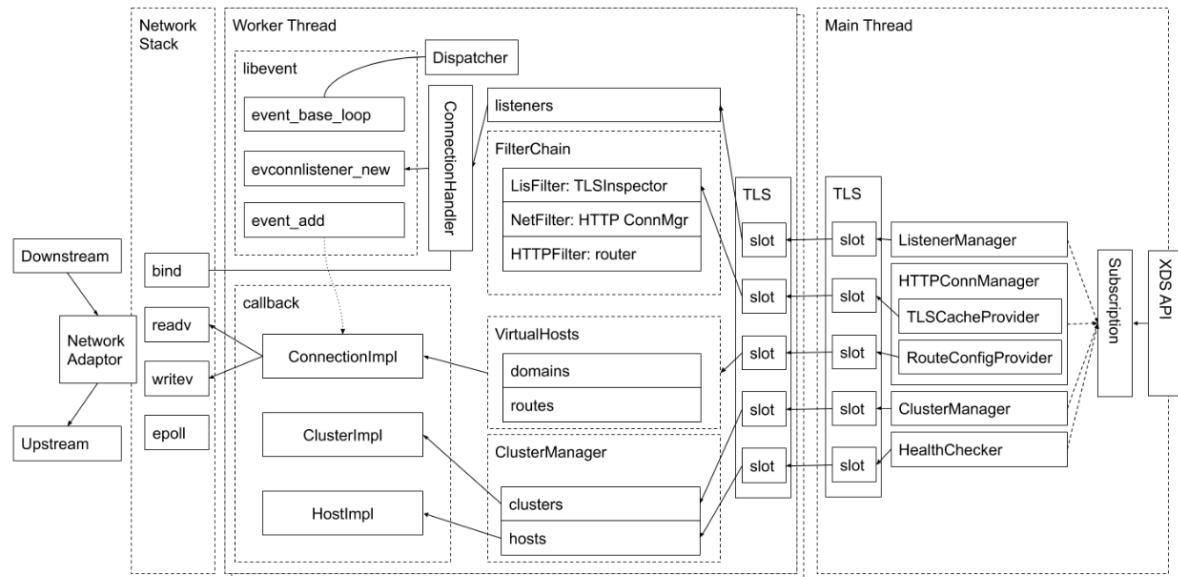
	Envoy	Nginx	HA Proxy
HTTP/2	对 HTTP/2 有最完整的支持，同时支持 upstream 和 downstream HTTP/2.	从 1.9.5 开始有限支持 HTTP/2 只在 upstream server 支持 HTTP/2, downstream 依然是 1.1.	不支持 HTTP/2.
Rate Limit	通过插件支持限流	支持基于配置的限流，只支持基于源IP的限流	
ACL	基于插件实现四层 ACL	基于源 / 目标地址实现 ACL	
Connection draining	支持 hot reload，并且通过 share memory 实现 connection draining 的功能	Nginx Plus 收费版支持 connection draining	支持热启动，但不保证丢弃连接

上图中的 Connection draining 指服务升级的时候，不中断现有连接，而是给一定的时间窗口让其把现有请求处理完再断开的过程。

优势

- 在L4和L7都通过可插拔过滤器提高了可扩展性
- API可配置性，即配置可以通过网络端点进行，使得Envoy不需要重新启动。

架构



Envoy的发现机制——xDS

xDS即架构图的最右侧的那个方块。

- 配置信息
 - LDS, Listener Discovery Service, 不严谨地对应一个端口。
 - RDS, Route Discovery Service, 路由规则, 比如 /a。RDS指向CDS, 从而配置和状态发生了关联。
- 状态信息——K8S集群的状态
 - CDS, Cluster Discovery Service, 简单的理解为对应K8S中的Service
 - EDS, Endpoint Discovery Service, 简单的理解为对应K8S中的Service里的POD

Secret Discovery Service (SDS):

- 一个专用的 API 来传递 TLS 密钥材料。这将解耦通过 LDS/CDS 发送主要监听器、集群配置和通过专用密钥管理系统发送秘钥素材。

Health Discovery Service (HDS):

- 该 API 将允许 Envoy 成为分布式健康检查网络的成员。中央健康检查服务可以使用一组 Envoy 作为健康检查终点并将状态报告回来, 从而缓解 N^2 健康检查问题, 这个问题指的是其间的每个 Envoy 都可能需要对每个其他 Envoy 进行健康检查。

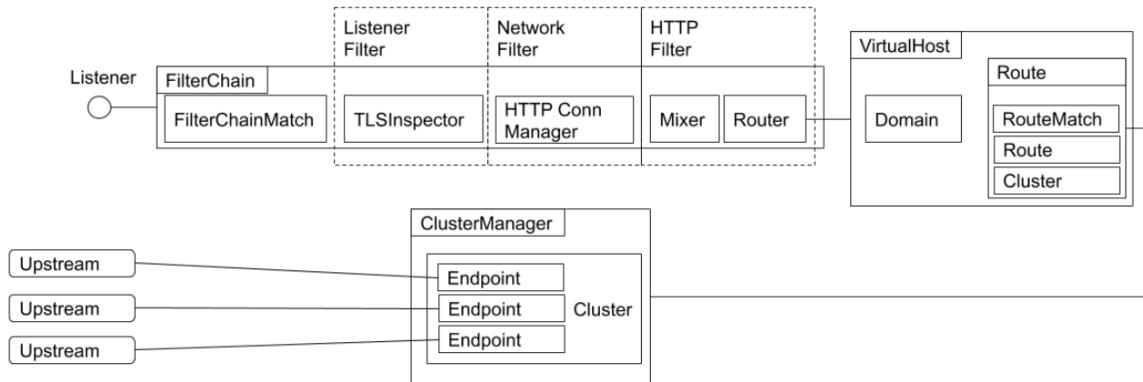
Aggregated Discovery Service (ADS):

- 总的来说, Envoy 的设计是最终一致的。这意味着默认情况下, 每个管理 API 都并发运行, 并且不会相互交互。在某些情况下, 一次一个管理服务器处理单个 Envoy 的所有更新是有益的(例如, 如果需要对更新进行排序以避免流量下降)。此 API 允许通过单个管理服务器的单个 gRPC 双向流对所有其他 API 进行编组, 从而实现确定性排序。
- SDS去K8S里发现secret, 得到key和cert。
- HDS。Envoy作为类似于NGINX的代理, 也需要高可用, 但是当实例数量太多的时候, 对后端服务主动做健康检查时, 会对后端服务产生过大压力; 为了减轻后端服务器的压力, 将健康检查委托给

其中一个实例，其余实例去问这个特殊实例要后端服务的健康状态。

- ADS处理LDS、RDS、CDS之间的先后关系（比如RDS指向CDS），处理后一起下发。而EDS是独立下发的，因为它本身是增量的（集群中的大部分ED是稳定的，只有少部分的变化频繁），所以对变化的部分做增量下发。

过滤器模式



- TLSInspector。在一个端口上绑定了上千个域名，每个域名都有一个证书，这样Listener要和这些证书做关联。当加密请求过来时，如何选择一个证书呢？客户端在握手请求中把FQDN（全限定名）带过来，Listener根据这个域名信息找证书。
- HTTP Filter
 - Mixer：限流、指标上报
 - Router：路由转发。
- 转发规则中的 `virtualHost` 做域名匹配，然后根据 `RouteMatch` 做路由匹配，寻找到 Cluster。

静态配置举例

```
admin: #配置管理端口
address:
  socket_address: { address: 127.0.0.1, port_value: 9901 }

static_resources: #静态配置，和XDS没有任何关系
listeners: #最上层对象1-->listener
- name: listener_0
  address: #监听地址
  socket_address: { address: 0.0.0.0, port_value: 10000 }
  filter_chains: #filter机制
    - filters:
      - name: envoy.filters.network.http_connection_manager
        typed_config:
          "@type": type.googleapis.com/envoy.extensions.filters.network.http_connection_manager.v3.HttpConnectionManager
          stat_prefix: ingress_http
          codec_type: AUTO
          route_config:
            name: local_route
            virtual_hosts: #虚拟主机
              - name: local_service #匹配任何域名
                domains: ["*"]
                routes: #路由规则
                  - match: { prefix: "/" } #将以"/"开头的请求转发到
                    route: { cluster: some_service }
```

```

        http_filters:
          - name: envoy.filters.http.router

clusters:
  - name: some_service
    connect_timeout: 0.25s
    type: LOGICAL_DNS
    #LOGICAL_DNS表示会解析域名（即下面address对应的属性）的地址
    lb_policy: ROUND_ROBIN
    load_assignment:
      cluster_name: some_service
      endpoints:
        - lb_endpoints:
          - endpoint:
            address:
              socket_address:
                address: simple
                port_value: 80
                #后端服务的名字

```

上面cluster对应的是一个服务simple，这个simple定义如下：

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: simple
spec:
  replicas: 1
  selector:
    matchLabels:
      app: simple
  template:
    metadata:
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/port: "80"
      labels:
        app: simple
    spec:
      containers:
        - name: simple
          imagePullPolicy: Always
          image: cncamp/httpserver:v1.0-metrics
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: simple
spec:
  ports:
    - name: http
      port: 80
      protocol: TCP
      targetPort: 80
  selector:
    app: simple

```

针对上述的Envoy配置和后端服务simple，启动一个Envoy

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    run: envoy
  name: envoy
spec:
  replicas: 1
  selector:
    matchLabels:
      run: envoy
  template:
    metadata:
      labels:
        run: envoy
    spec:
      containers:
        - image: envoyproxy/envoy-dev
          name: envoy
          volumeMounts:
            - name: envoy-config
              mountPath: "/etc/envoy"
              readOnly: true
      volumes:
        - name: envoy-config
          configMap:
            name: envoy-config
```

Envoy启动后会有一个对应的pod

```
$ k get po -owide
NAME           READY   STATUS    RESTARTS   AGE     IP          NODE
             NOMINATED NODE   READINESS GATES
envoy-fb5d77cc9-vfv9n   1/1     Running   0          28m   10.0.0.13
xiaokai-thinkpad-x13-gen-1 <none>       <none>
```

那么就可以对Envoy的地址和端口即 10.0.0.13 + Envoy配置文件中指定的listener端口10000发起请求：

```
$ curl 10.0.0.13:10000/hello
hello [stranger]
=====Details of the http request header=====
Accept=[*/*]
X-Forwarded-Proto=[http]
X-Request-Id=[252144ce-97fd-470e-8b51-ef3145eafb24]
X-Envoy-Expected-Rq-Timeout-Ms=[15000]
User-Agent=[curl/7.68.0]
```

Istio

以K8S为配置中心，拉取service和pod的信息，加上自己的配置信息，Istio将它们转换成Envoy的配置下发下去，从而形成mesh。

安装

```
# install istio
curl -L https://istio.io/downloadIstio | sh -
cd istio-1.12.0
cp bin/istioctl /usr/local/bin
istioctl install --set profile=demo -y

# istio monitoring
grafana dashboard 7639
```

安装完Istio后，会在K8S集群中增加一个MutatingWebhook，监听打了`istio-injection=enabled`标的命名空间中的POD创建事件，当这类事件发生后，会修改Pod的定义，放入一个初始化容器（修改iptables）和另外一个Envoy容器。

实验

安装好Istio后，可以执行如下命令

```
# 创建一个命名空间
kubectl create ns sidecar
# 给命名空间打上标签“istio-injection=enabled”，这样Istio就会为其中的Pod注入sidecar
kubectl label ns sidecar istio-injection=enabled
# 创建两个Deployment
kubectl apply -f nginx.yaml -n sidecar
kubectl apply -f toolbox.yaml -n sidecar
```

其中服务端nginx.yaml内容如下（无任何特殊之处）：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
---
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  ports:
    - name: http
      port: 80
```

```
protocol: TCP
targetPort: 80
selector:
app: nginx
```

客户端toolbox.yaml内容如下 (就是一个普通的CENTOS)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: toolbox
spec:
  replicas: 1
  selector:
    matchLabels:
      app: toolbox
  template:
    metadata:
      labels:
        app: toolbox
        access: "true"
  spec:
    containers:
      - name: toolbox
        image: centos
        command:
          - tail
          - -f
          - /dev/null
```

查看上面创建的pod会发现每个pod里镜像的数目由一个变成了两个

```
root@node1:~/go/src/github.com/cncamp/101# k get po -n sidecar
NAME                           READY   STATUS    RESTARTS   AGE
nginx-deployment-6799fc88d8-xchqt   1/2     Running   0          8s
```

查看POD的细节，会看到：

- MutatingWebhook创建的Envoy容器，部分定义如下

```
image: docker.io/istio/proxyv2:1.12.0
imagePullPolicy: IfNotPresent
name: istio-proxy
ports:
- containerPort: 15090
  name: http-envoy-prom
  protocol: TCP
readinessProbe:
  failureThreshold: 30
  httpGet:
    path: /healthz/ready
    port: 15021
    scheme: HTTP
  initialDelaySeconds: 1
  periodSeconds: 2
  successThreshold: 1
  timeoutSeconds: 3
resources:
  limits:
    cpu: "2"
    memory: 1Gi
  requests:
    cpu: 10m
```

- MutatingWebhook插入的init容器，部分定义如下

```
initContainers:
- args:
  - istio-iptables
  - -p
  - "15001"
  - -Z
  - "15006"
  - -u
  - "1337"
  - -m
  - REDIRECT
  - -i
  - '*'
  - -X
  - ""
  - -b
  - '*'
  - -d
  - 15090,15021,15020
image: docker.io/istio/proxyv2:1.12.0
imagePullPolicy: IfNotPresent
name: istio-init
resources:
  limits:
    cpu: "2"
    memory: 1Gi
  requests:
    cpu: 10m
    memory: 40Mi
```

命令行参数不清楚，但貌似是插入了一些iptables规则。

此时如果进入客户端POD toolbox内部，curl服务端服务的服务名nginx，可以看到响应结果。

规则配置原理

首先查看客户端toolbox的容器ID

```
crictl ps
```

然后找到其对应的进程ID

```
crictl inspect [contain id] | grep pid
```

进入容器进程对应的网络空间，执行一些操作

```
nsenter -t [pid of container] -n ip a
```

输出如下：

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
3: eth0@if26: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
    link/ether c6:1d:45:9f:28:d2 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 192.168.166.135/32 scope global eth0
        valid_lft forever preferred_lft forever
    inet6 fe80::c41d:45ff:fe9f:28d2/64 scope link
        valid_lft forever preferred_lft forever
```

客户端

查看被Istio注入容器内部的iptables规则

```
nsenter -t 244941 -n iptables-save
```

```
root@node1:~/go/src/github.com/cncamp/101# nsenter -t 101072 -n iptables-save -t nat
# Generated by iptables-save v1.8.4 on Sat Dec 18 18:44:44 2021
*nat
:PREROUTING ACCEPT [103:6180]
:INPUT ACCEPT [103:6180]
:OUTPUT ACCEPT [39:3171]
:POSTROUTING ACCEPT [41:3291]
:ISTIO_INBOUND - [0:0]
:ISTIO_IN_REDIRECT - [0:0]
:ISTIO_OUTPUT - [0:0]
:ISTIO_REDIRECT - [0:0]
-A PREROUTING -p tcp -j ISTIO_INBOUND
-A OUTPUT -p tcp -j ISTIO_OUTPUT
-A ISTIO_INBOUND -p tcp -m tcp --dport 15008 -j RETURN
-A ISTIO_INBOUND -p tcp -m tcp --dport 22 -j RETURN
-A ISTIO_INBOUND -p tcp -m tcp --dport 15090 -j RETURN
-A ISTIO_INBOUND -p tcp -m tcp --dport 15021 -j RETURN
-A ISTIO_INBOUND -p tcp -m tcp --dport 15020 -j RETURN
-A ISTIO_INBOUND -p tcp -j ISTIO_IN_REDIRECT
-A ISTIO_IN_REDIRECT -p tcp -j REDIRECT --to-ports 15006
-A ISTIO_OUTPUT -s 127.0.0.6/32 -o lo -j RETURN
-A ISTIO_OUTPUT ! -d 127.0.0.1/32 -o lo -m owner --uid-owner 1337 -j ISTIO_IN_REDIRECT
-A ISTIO_OUTPUT -o lo -m owner ! --uid-owner 1337 -j RETURN
-A ISTIO_OUTPUT -m owner --uid-owner 1337 -j RETURN
-A ISTIO_OUTPUT ! -d 127.0.0.1/32 -o lo -m owner --gid-owner 1337 -j ISTIO_IN_REDIRECT
-A ISTIO_OUTPUT -o lo -m owner ! --gid-owner 1337 -j RETURN
-A ISTIO_OUTPUT -m owner --gid-owner 1337 -j RETURN
-A ISTIO_OUTPUT -d 127.0.0.1/32 -j RETURN
-A ISTIO_OUTPUT -j ISTIO_REDIRECT
-A ISTIO_REDIRECT -p tcp -j REDIRECT --to-ports 15001
COMMIT
# Completed on Sat Dec 18 18:44:44 2021
```

作为客户端，命中的iptables如下：

- OUTPUT: 所有的TCP规则跳转到ISTIO_OUTPUT规则链
- ISTIO_OUTPUT链有多条，最终命中ISTIO_REDIRECT规则，被重定向到15001端口

那么，15001是谁监听呢？

```
istioctl ps -n sidecar
```

上面命令中的ps表示 `policy status`，查看规则的同步情况。

使用 `istioctl ps` 命令查看pod内配置好的规则，此时在寻找15001端口谁在监听，即查看 `listener`

```
istioctl pc listener -n sidecar toolbox-68f79dd5f8-dcz5g --port 15001 -ojson
...
{
  "name": "virtualoutbound",
  "address": {
    "socketAddress": {
      "address": "0.0.0.0",
      "portValue": 15001
    }
  },
  "useOriginalDst": true
}
```

15001端口收到请求后，会将它们转到虚拟端口，虚拟端口和请求的端口一样（比如80，Istio监听的15001端口到虚拟端口80的转发只是处理配置的过程，并没有走网络）

```
"name": "0.0.0.0_80",
"address": {
  "socketAddress": {
    "address": "0.0.0.0",
    "portValue": 80
  }
},
"routeConfigName": "80"
```

可以使用 `istioctl ps route` 查看路由规则

```
istioctl pc route -n sidecar toolbox-68f79dd5f8-dcz5g --name=80 -ojson
```

输出

```
{
  "name": "nginx.sidecar.svc.cluster.local:80",
  "domains": [
    "nginx.sidecar.svc.cluster.local",
    "nginx.sidecar.svc.cluster.local:80",
    "nginx",
    "nginx:80",
    "nginx.sidecar.svc",
    "nginx.sidecar.svc:80",
    "nginx.sidecar",
    "nginx.sidecar:80",
    "10.109.253.161",
    "10.109.253.161:80"
  ],
  "routes": [
    {
      "name": "default",
      "match": {
```

```

        "prefix": "/"
    },
    "route": {
        "cluster":
"outbound|80||nginx.sidecar.svc.cluster.local",
    }
    ...

```

上述输出表明：then it is handed over to cluster outbound|80||nginx.sidecar.svc.cluster.local and the endpoint is the actual pod ip。

使用`istioctl ps cluster`查看cluster

```
istioctl pc cluster -n sidecar toolbox-68f79dd5f8-dcz5g --
fqdn=nginx.sidecar.svc.cluster.local
```

```
root@node1:~/go/src/github.com/cncamp/101# istioctl pc cluster -n sidecar toolbox-68f79dd5f8-rsrjk --fqdn=nginx.sidecar.svc.cluster.local
SERVICE FQDN          PORT      SUBSET   DIRECTION   TYPE      DESTINATION RULE
nginx.sidecar.svc.cluster.local  80        -         outbound     EDS
```

查看cluster对应的Endpoint

```
$ istioctl pc endpoint -n sidecar toolbox-68f79dd5f8-dcz5g
192.168.166.189:80           HEALTHY      OK
outbound|80||nginx.sidecar.svc.cluster.local
```

then the request is send to kernel, going to iptables again

```
-A OUTPUT -p tcp -j ISTIO_OUTPUT
-A ISTIO_OUTPUT -m owner --uid-owner 1337 -j RETURN
```

上面的第二条规则，判断如果是Envoy自己转发的（根据用户ID1337），则不在处理，iptables return and the traffic is send out!。

用户ID1337是Envoy镜像在制作时添加的命令，比如使用如下命令查看镜像

```
crictl inspecti [镜像id]
```

可以看到如下的添加用户的命令

```
{
  "created": "2021-10-06T19:03:32.921258572",
  "created_by": "RUN /bin/sh -c useradd -m --uid 1337 istio-proxy \u0026\u0026 echo \"istio-proxy ALL=NOPASSWD: ALL\" \u003e\u003e /etc/sudoers # build
kit",
  "comment": "buildkit.dockerfile.v0"
},
```

服务端

server side is almost same but easier

iptables first, hijacked to port 15006

```

root@node1:~/go/src/github.com/cncamp/101# nsenter -t 97626 -n iptables-save -t nat
# Generated by iptables-save v1.8.4 on Sat Dec 18 19:03:57 2021
*nat
:PREROUTING ACCEPT [792:47520]
:INPUT ACCEPT [794:47640]
:OUTPUT ACCEPT [83:7203]
:POSTROUTING ACCEPT [83:7203]
:ISTIO_INBOUND - [0:0]
:ISTIO_IN_REDIRECT - [0:0]
:ISTIO_OUTPUT - [0:0]
:ISTIO_REDIRECT - [0:0]
-A PREROUTING -p tcp -j ISTIO_INBOUND
-A OUTPUT -p tcp -j ISTIO_OUTPUT
-A ISTIO_INBOUND -p tcp -m tcp --dport 15008 -j RETURN
-A ISTIO_INBOUND -p tcp -m tcp --dport 22 -j RETURN
-A ISTIO_INBOUND -p tcp -m tcp --dport 15090 -j RETURN
-A ISTIO_INBOUND -p tcp -m tcp --dport 15021 -j RETURN
-A ISTIO_INBOUND -p tcp -m tcp --dport 15020 -j RETURN
-A ISTIO_INBOUND -p tcp -j ISTIO_IN_REDIRECT
-A ISTIO_IN_REDIRECT -p tcp -j REDIRECT --to-ports 15006
-A ISTIO_OUTPUT -s 127.0.0.6/32 -o lo -j RETURN
-A ISTIO_OUTPUT ! -d 127.0.0.1/32 -o lo -m owner --uid-owner 1337 -j ISTIO_IN_REDIRECT
-A ISTIO_OUTPUT -o lo -m owner ! --uid-owner 1337 -j RETURN
-A ISTIO_OUTPUT -m owner --uid-owner 1337 -j RETURN
-A ISTIO_OUTPUT ! -d 127.0.0.1/32 -o lo -m owner --gid-owner 1337 -j ISTIO_IN_REDIRECT
-A ISTIO_OUTPUT -o lo -m owner ! --gid-owner 1337 -j RETURN
-A ISTIO_OUTPUT -m owner --gid-owner 1337 -j RETURN
-A ISTIO_OUTPUT -d 127.0.0.1/32 -j RETURN
-A ISTIO_OUTPUT -j ISTIO_REDIRECT
-A ISTIO_REDIRECT -p tcp -j REDIRECT --to-ports 15001
COMMIT
# Completed on Sat Dec 18 19:03:57 2021

```

1. 进入流量首先走PREROUTING链，跳转到ISTIO_INBOUND规则。
2. ISTIO_INBOUND规则命中最后一条，跳转到ISTIO_IN_REDIRECT；
3. ISTIO_IN_REDIRECT规则重定向到15006端口，即Envoy实际监听的端口。

```

istioctl pc listener -n sidecar nginx-deployment-6799fc88d8-qq68n --port 15006
{
    "name": "virtualInbound",
    "address": {
        "socketAddress": {
            "address": "0.0.0.0",
            "portValue": 15006
        }
    },
    ---
    {
        "name": "0.0.0.0_80",
        "address": {
            "socketAddress": {
                "address": "0.0.0.0",
                "portValue": 80
            }
        },
        "filters": [
            "inbound_0.0.0.0_80"
        ]
    },
    "domains": [
        "*"
    ],
}

```

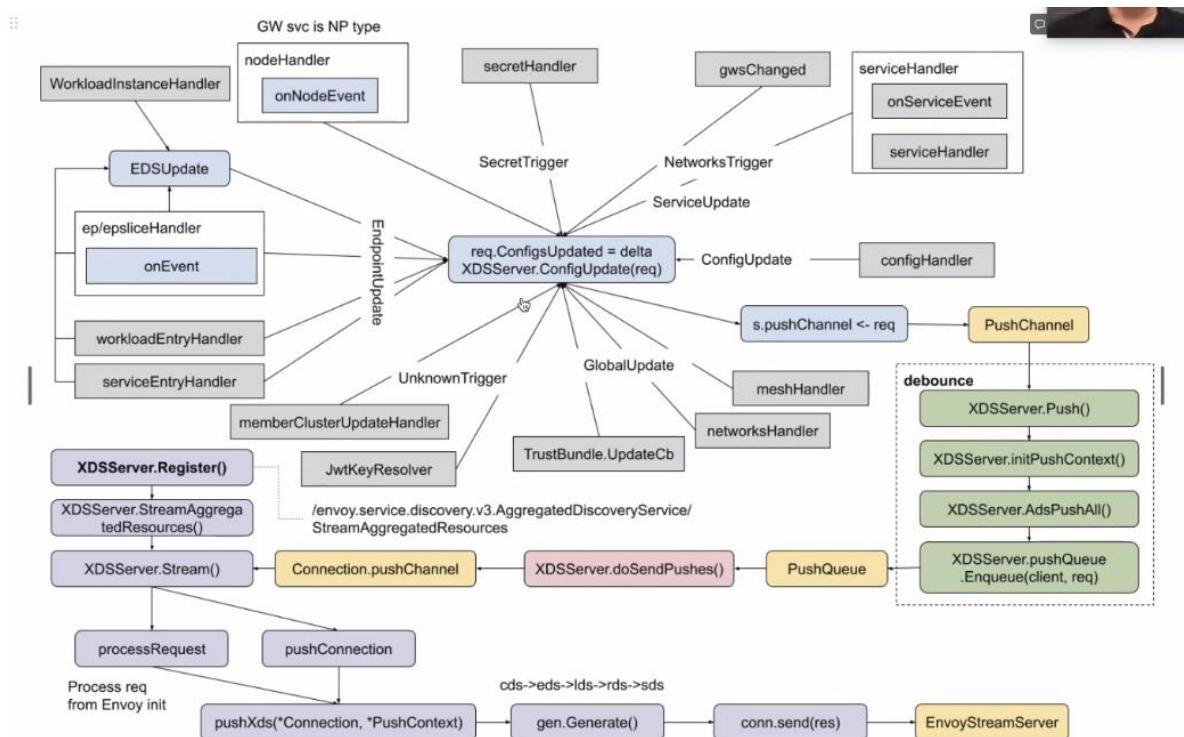
```

    "routes": [
      {
        "name": "default",
        "match": {
          "prefix": "/"
        },
        "route": {
          "cluster": "inbound|80||"
        }
      }
    ]
  }
}

istioctl pc route -n sidecar nginx-deployment-6799fc88d8-qq68n --name="inbound|80||" -ojson
---
{
  "name": "inbound|80||",
  "decorator": {
    "operation": "nginx.sidecar.svc.cluster.local:80/*"
  }
}
---
istioctl pc cluster -n sidecar nginx-deployment-6799fc88d8-qq68n
nginx.sidecar.svc.cluster.local
  80      -      outbound      EDS
---
istioctl pc endpoint -n sidecar nginx-deployment-6799fc88d8-qq68n
---
192.168.166.189:80           HEALTHY      OK
outbound|80||nginx.sidecar.svc.cluster.local
---
then it is send to kernel, handled by OUTPUT chain
-A ISTIO_OUTPUT -m owner --uid-owner 1337 -j RETURN

```

Envoy中的规则是如何形成并下发的

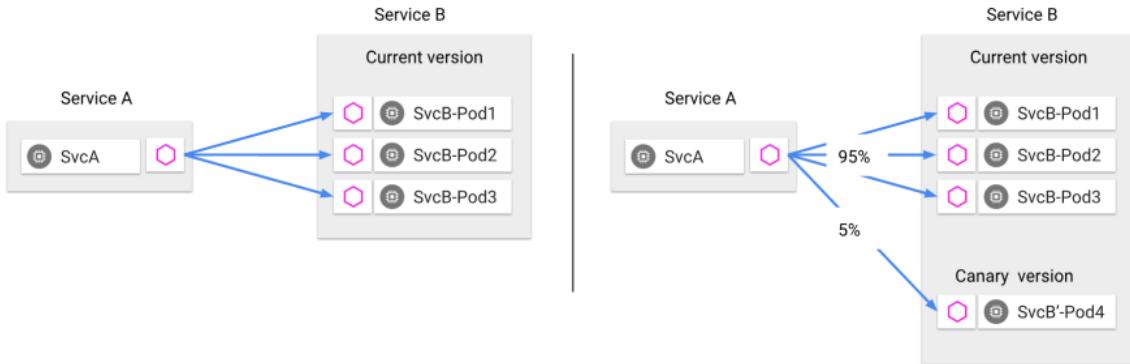


- 首先上图中的serviceHandler、secretHandler、epsliceHandler(关注POD状态的更新)、configHandler从K8S集群中拉去信息，加上自己的配置，通过PushChannel推送至Istio；

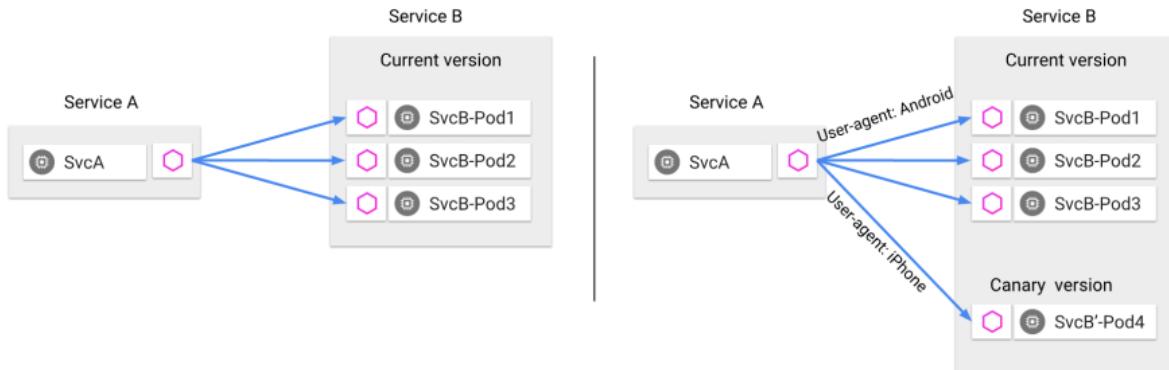
- 从PushChannel中获取配置（每100ms获取一次），合并发送至PushQueue，然后通过每个sidecar专享的Connection下发到XDSServer.Stream()；
- pushConnection→pushXds→gen.Generate，按照cds→eds→lds→rds→sds的顺序生成Envoy配置并下发
- Envoy启动的时候会向Istio拉取一次配置（拉模式）；后续如果集群状态有变更，配置会被推送下来（推模式）。

流量管理

- 金丝雀发布



- 基于内容的流量管理



- 服务之间的通讯

Istio不提供DNS，应用使用底层的kube-dns中的DNS服务来解析FQDN。

- 提供接入网关Ingress和接出网关Egress。上面的金丝雀和基于内容搞的流量管理都是基于Ingress做的。
- 提供服务发现和负载均衡

在做负载均衡时，会自动根据API调用的返回码来统计失败率，从而实现断路器模式——剔除失败的实例。服务可以通过HTTP 503响应健康检查来主动减轻负担。

- 故障处理

- 超时处理。如果客户端没有超时处理，sidecar可以植入此类规则，在超时时直接断开客户端链接。
- 对应用透明的重试机制
- 基于并发量和请求的流量控制
- 成员的健康检查
- 细粒度熔断机制，可以针对Load Balancing Pool中的每个成员设置规则

各种功能讲解

Gateway相关

这里演示的是Gateway，集群流量的入口，不是mesh之间的调用，所以不需要自动注入sidecar，就是名称空间不需要打`injection=enabled`这样的标签。

对于如下的HTTP服务

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: simple
spec:
  replicas: 1
  selector:
    matchLabels:
      app: simple
  template:
    metadata:
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/port: "80"
      labels:
        app: simple
    spec:
      containers:
        - name: simple
          imagePullPolicy: Always
          image: cncamp/httpserver:v1.0-metrics
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: simple
spec:
  ports:
    - name: http
      port: 80
      protocol: TCP
      targetPort: 80
  selector:
    app: simple
```

下面的任务是将服务simple发布出去。

VirtualService

使用Istio的**VirtualService**对其增加一条路由规则

```
apiVersion: networking.istio.io/v1beta1
kind: virtualService  #对应route
metadata:
  name: simple
spec:
```

```

gateways:
  - simple          #此路由规则于名为simple的Gateway产生关联关系
hosts:
  - simple.cncamp.io
http:
  - match:
    - port: 80
  route:
    - destination:
      host: simple.simple.svc.cluster.local
      port:
        number: 80

```

上述定义的含义是发送到simple.cncamp.io的80端口的请求都会被路由至
simple.simple.svc.cluster.local

Gateway

使用Istio的**Gateway**对其一个Listener

```

apiVersion: networking.istio.io/v1beta1
kind: Gateway          # 对应Envoy中的监听器
metadata:
  name: simple
spec:
  selector:
    istio: ingressgateway   #监听器起在带有标签ingressgateway的pod里
  servers:
    - hosts:
      - simple.cncamp.io  # 处理到这样的host主机的请求
      port:
        name: http-simple
        number: 80          #监听80端口，在上面起HTTP服务
        protocol: HTTP

```

上面的selector选择的标签是 `ingressgateway`。 Istio安装后，会在 `istio-system` 命名空间中运行如下三个pod

```

root@node1:~/go/src/github.com/cncamp/101# kubectl get po
NAME                               READY   STATUS    RESTARTS   AGE
istio-egressgateway-7f4864f59c-jl5nh   1/1     Running   2 (3h2m ago)   6d2h
istio-ingressgateway-55d9fb9f-7xnwv    1/1     Running   2 (3h2m ago)   6d2h
istiod-555d47cb65-rbtpf                1/1     Running   2 (3h2m ago)   6d2h

```

分别是 `egress`、`ingress`、和 `istiod`，其中 `ingress` 具有 `istio=ingressgateway` 的标签

```

root@node1:~/go/src/github.com/cncamp/101# kubectl get po --show-labels
NAME                               READY   STATUS    RESTARTS   AGE   LABELS
istio-egressgateway-7f4864f59c-jl5nh   1/1     Running   2 (3h2m ago)   6d2h   app=istio-egressgateway,chart=gateways,heritage=Tiller,install.operator.istio.io/o
wning-resource=unknown,istio.io/rev=default,istio=egressgateway,operator.istio.io/component=EgressGateways,pod-template-hash=7f4864f59c,release=istio,service.ist
io.io/canonical-name=istio-egressgateway,service.istio.io/component=EgressGateways,pod-template-hash=7f4864f59c,release=istio,service.istio.io/revision=latest,sidecar.istio.io/inject=false
istio-ingressgateway-55d9fb9f-7xnwv    1/1     Running   2 (3h2m ago)   6d2h   app=istio-ingressgateway,chart=gateways,heritage=Tiller,install.operator.istio.io/o
wning-resource=unknown,istio.io/rev=default,istio=ingressgateway,operator.istio.io/component=IngressGateways,pod-template-hash=55d9fb9f,release=istio,service.ist
io.io/canonical-name=istio-ingressgateway,service.istio.io/component=IngressGateways,pod-template-hash=55d9fb9f,release=istio,service.istio.io/revision=latest,sidecar.istio.io/inject=false
istiod-555d47cb65-rbtpf                1/1     Running   2 (3h2m ago)   6d2h   app=istiod,install.operator.istio.io/owning-resource=unknown,istio.io/rev=default,
istio=pilot,operator.istio.io/component=Pilot,pod-template-hash=555d47cb65,sidecar.istio.io/inject=false

```

所以上面Gateway的定义会在ingress的pod中插入规则：

- 在80上侦听处理域名simple.cncamp.io的HTTP请求

同时VirtualService会在上面80的监听器下面加一个转发规则：

- 访问端口80，host为simple.cncamp.io的请求会被转发至服务simple.simple.svc.cluster.local的80端口。

URL重写

可以修改上面的VirtualService定义，增加URL重写

```
apiVersion: networking.istio.io/v1beta1
kind: virtualService
metadata:
  name: simple
spec:
  gateways:
    - simple
  hosts:
    - simple.cncamp.io
  http:
    - match:
        - uri:
            exact: "/simple/hello"
      rewrite:
        uri: "/hello"
      route:
        - destination:
            host: simple.simple.svc.cluster.local
            port:
              number: 80
    - match:
        - uri:
            prefix: "/nginx"
      rewrite:
        uri: "/"
      route:
        - destination:
            host: nginx.simple.svc.cluster.local
            port:
              number: 80
```

发布HTTPS服务

可以创建一个普通的HTTP服务，然后把它以HTTPS的形式暴露出去

1. 签发一对公私钥

```
openssl req -x509 -sha256 -nodes -days 365 -newkey rsa:2048 -subj '/O=cncamp
Inc./CN=*.cncamp.io' -keyout cncamp.io.key -out cncamp.io.crt
```

2. 用这个公私钥创建secret

```
kubectl create -n istio-system secret tls cncamp-credential --
key=cncamp.io.key --cert=cncamp.io.crt
```

3. 使用上面的secret发布HTTPS服务，仍然是涉及VirtualService和Gateway对象

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
```

```
metadata:
  name: httpsserver
spec:
  gateways:
    - httpsserver
  hosts:
    - httpsserver.cncamp.io
  http:
    - match:
        - port: 443          # 匹配的端口改成了443
      route:
        - destination:
            host: httpserver.securesvc.svc.cluster.local
            port:
              number: 80
---
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
  name: httpsserver
spec:
  selector:
    istio: ingressgateway
  servers:
    - hosts:
        - httpsserver.cncamp.io
      port:
        name: https-default
        number: 443
        protocol: HTTPS      # 指定为HTTPS
      tls:
        mode: SIMPLE         # 单向HTTPS
        credentialName: cncamp-credential  # 指定为创建的secret的名字
```

证书也可以不像上面的使用secret挂载，而是直接指定路径，但是**这个办法不推荐**：

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: bookinfo-gateway
spec:
  servers:
    - port:
        number: 443
        name: https
        protocol: HTTPS
      hosts:
        - bookinfo.com
      tls:
        mode: SIMPLE
        serverCertificate: /tmp/tls.crt
        privateKey: /tmp/tls.key
```

4. 可以用带有TLSInspector机制的curl命令访问HTTPS服务了

```
curl --resolve httpsserver.cncamp.io:443:$INGRESS_IP
https://httpsserver.cncamp.io/healthz -v -k
```

灰度发布

这里的例子是在mesh内部的。

1. 建立v1版的用户端服务

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: canary
spec:
  replicas: 1
  selector:
    matchLabels:
      app: canary
  template:
    metadata:
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/port: "80"
    labels:
      app: canary
      version: v1
```

```

spec:
  containers:
    - name: canary
      imagePullPolicy: Always
      image: cncamp/httpserver:v1.0-metrics
      ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: canary
spec:
  ports:
    - name: http
      port: 80
      protocol: TCP
      targetPort: 80
  selector:
    app: canary

```

注意，服务只选择了POD的名字，没有选择版本

2. 建立客户端

只是为了可以在里面执行curl命令。

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: toolbox
spec:
  replicas: 1
  selector:
    matchLabels:
      app: toolbox
  template:
    metadata:
      labels:
        app: toolbox
        access: "true"
  spec:
    containers:
      - name: toolbox
        image: centos
        command:
          - tail
          - -f
          - /dev/null

```

3. 在客户端中访问服务端

```
curl canary/hello
```

4. 发布V2版的服务(注意只有Deployment, **Service是复用V1版发布时的服务**)

```

apiVersion: apps/v1
kind: Deployment

```

```

metadata:
  name: canary-v2
spec:
  replicas: 1
  selector:
    matchLabels:
      app: canary
  template:
    metadata:
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/port: "80"
    labels:
      app: canary          # label和v1版的相同
      version: v2          # 此处版本号和v1版服务不同
  spec:
    containers:
      - name: canary
        imagePullPolicy: Always
        image: cncamp/httpserver:v2.0-metrics # 镜像不同
      ports:
        - containerPort: 80

```

可以看到这两个版本的同一套POD

```

jesse@JESSEMENG-MB0 5.canary % k get po -n canary --show-labels -l app=canary
NAME           READY   STATUS    RESTARTS   AGE   LABELS
canary-67bf875a59-tg5g8  2/2     Running   0          2m32s   app=canary,pod-template-hash=67bf875a59,security.istio.io/tlsMode=istio,service.istio.io/canonical-name=canary,service.istio.io/canonical-revision=v1,version=v1
canary-v2-c88d5b655-67g5m  2/2     Running   0          23s    app=canary,pod-template-hash=c88d5b655,security.istio.io/tlsMode=istio,service.istio.io/canonical-name=canary,service.istio.io/canonical-revision=v2,version=v2

```

注意服务没有变化，但只使用pod的app label而不是version label:

```

jesse@JESSEMENG-MB0 5.canary % k get svc -ncanary -owide
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE      SELECTOR
canary    ClusterIP  10.100.13.110  <none>          80/TCP      2m53s   app=canary

```

5. 使用Istio的DestinationRule根据version标签将pod切分到不同的子集

```

apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: canary
spec:
  host: canary #为服务canary而设
  trafficPolicy:
    loadBalancer:
      simple: RANDOM # 默认使用随机的负载均衡策略
  subsets:
    - name: v1
      labels:
        version: v1
    - name: v2
      labels:
        version: v2
      trafficPolicy:
        loadBalancer:
          simple: ROUND_ROBIN #不使用默认的负载均衡策略，而为V2子集指定R策略

```

ROUND_ROBIN策略

6. 不同子集的流量转发还是使用VirtualService

- 根据header做流量转发

```
apiVersion: networking.istio.io/v1beta1
kind: virtualservice
metadata:
  name: canary
spec:
  hosts:
    - canary      # 针对名为canary的K8S service
  http:
    - match:        # 根据header来做交易路由
      - headers:
          user:
            exact: jesse #user头的值为jesse的请求转发值v2版的服务
    route:
      - destination:
          host: canary
          subset: v2
      - route:      # 否则去v1版的服务。这是一个不带match标签的路由规则，即默认的
          路由规则
        - destination:
            host: canary
            subset: v1
```

- 根据权重转发流量

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
    - reviews
  http:
    - route:
        - destination:
            host: reviews
            subset: v1
            weight: 75
        - destination:
            host: reviews
            subset: v2
            weight: 25
```

超时

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
    - ratings
  http:
    - route:
        - destination:
            host: ratings
            subset: v1
  timeout: 10s
```

上面的timeout参数控制的是和服务端之间的超时时间。比如

```
apiVersion: networking.istio.io/v1beta1
kind: virtualservice
metadata:
  name: canary
spec:
  hosts:
    - canary
  http:
    - match:
        - headers:
            user:
              exact: jesse
      route:
        - destination:
            host: canary
            subset: v2
      # if upstream server response delay is greater than 1s, send timeout error
      # to client
      timeout: 1s
    - route:
        - destination:
            host: canary
            subset: v1
```

重试

返回5xx时重试

```
apiVersion: networking.istio.io/v1beta1
kind: virtualService
metadata:
  name: canary
spec:
  hosts:
    - canary
  http:
    - match:
        - headers:
            user:
              exact: jesse
      route:
        - destination:
            host: canary
            subset: v2
    ### retry if upstream server send 5xx
    retries:
      attempts: 3
      perTryTimeout: 2s
    - route:
        - destination:
            host: canary
            subset: v1
    ### send 500 to client in 80%
    fault:
      abort:
        httpStatus: 500
        percentage:
          value: 80
```

故障注入

还是上面的VirtualService，使用 `fault` 属性指定超过80%的概率返回500的错误码。

- 还可以指定延迟

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
    - ratings
  http:
    - fault:
        delay:
          percent: 10
          fixedDelay: 5s
    route:
      - destination:
          host: ratings
          subset: v1
```

- 指定返回400的错误码

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
    - ratings
  http:
    - fault:
        abort:
          percent: 10
          httpStatus: 400
  route:
    - destination:
        host: ratings
        subset: v1
```

条件规则

这里指VirtualService匹配请求的规则

```
apiVersion:
networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: productpage
spec:
  hosts:
    - productpage
  http:
    - match:
        - uri:
            prefix: /api/v1
      ...
...
```

```
apiVersion:
networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
    - reviews
  http:
    - match:
        - headers:
            end-user:
              exact: jason
      ...
...
```

```
apiVersion:
networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
    - ratings
  http:
    - match:
        sourceLabels:
          app: reviews
      ...
...
```

流量镜像

mirror 规则可以使 Envoy 截取所有 request，并在转发请求的同时，将 request 转发至 Mirror 版本，同时在 Header 的 Host/Authority 加上 -shadow。

这些 mirror 请求会工作在 fire and forget 模式，所有的 response 都会被废弃。

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: httpbin
spec:
  hosts:
    - httpbin
  http:
    - route:
        - destination:
            host: httpbin
            subset: v1
            weight: 100
        - mirror:
            host: httpbin
            subset: v2
```

规则委托

就是VirtualService规则用类似C++的 `include` 语法把各个文件组合起来

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: bookinfo
spec:
  hosts:
    - "bookinfo.com"
  gateways:
    - mygateway
  http:
    - match:
        - uri:
            prefix: "/productpage"
      delegate:
        name: productpage
        namespace: nsA
    - match:
        - uri:
            prefix: "/reviews"
      delegate:
        name: reviews
        namespace: nsB
```

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: productpage
  namespace: nsA
spec:
  http:
    - match:
        - uri:
            prefix: "/productpage/v1/"
      route:
        - destination:
            host: productpage-v1.nsA.svc.cluster.local
    - route:
        - destination:
            host: productpage.nsA.svc.cluster.local
```

规则的优先级

同一个目标有多个VirtualService规则时，会按照定义的顺序从上往下依次匹配，一旦命中就不再往下进行，所以规则的排列应该和异常捕获一样，越精准的规则应该越靠上，反之越靠下。

目标规则

参加“灰度发布”相关的DestinationRule的描述，是数据包离开Envoy时的规则。

断路器

使用DestinationRule对象实现。

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: httpbin
spec:
  host: httpbin
  trafficPolicy:
    connectionPool:
      tcp:
        maxConnections: 1
      http:
        http1MaxPendingRequests: 1
        maxRequestsPerConnection: 1
    outlierDetection:
      consecutiveErrors: 1
      interval: 1s
      baseEjectionTime: 3m
      maxEjectionPercent: 100
```

上图中：

- tcp.maxConnections指定最多一个底层链接
- http.maxRequestsPerConnection指定每个底层链接只能有一个HTTP请求
- http.http1MaxPendingRequests指定最多有一个等待的请求

上面的规则意味者当服务器较慢时，第三个请求就被丢弃了。

outlier其实是一个被动的健康检查，会去判断500的错误和超时的错误：

- consecutiveError表示连续多少次500错误码活超时才会触发规则
- interval表示多久探测一次
- 一次踢出去多久
- maxEjectionPercent表示最多可以踢出去多少服务实例

ServiceEntry

用来为MESH外的服务建立对应的cluster，并被自动发现。ServiceEntry中使用hosts字段来指定目标，可以是一个完全限定名，也可以是个通配符域名。

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: foo-ext-svc
spec:
  hosts:
    - *.foo.com
  ports:
    - number: 80
      name: http
      protocol: HTTP
    - number: 443
      name: https
      protocol: HTTPS
```

WorkLoadEntry

还是一个ServiceEntry，只是用workloadSelector指定真正的后端服务——也即WorkLoadEntry。总体上，相当于一个外部服务的“指针”或者“引用”

```
apiVersion: networking.istio.io/v1beta1
kind: ServiceEntry
metadata:
  name: details-svc
spec:
  hosts:
    - details.bookinfo.com
  location: MESH_INTERNAL
  ports:
    - number: 80
      name: http
      protocol: HTTP
  resolution: STATIC
  workloadSelector:
    labels:
      app: details-legacy
```

```
apiVersion: networking.istio.io/v1beta1
kind: WorkloadEntry
metadata:
  name: details-svc
spec:
  serviceAccount: details-legacy
  address: 2.2.2.2
  labels:
    app: details-legacy
    instance-id: vm1
```

遥测Telemetry V2

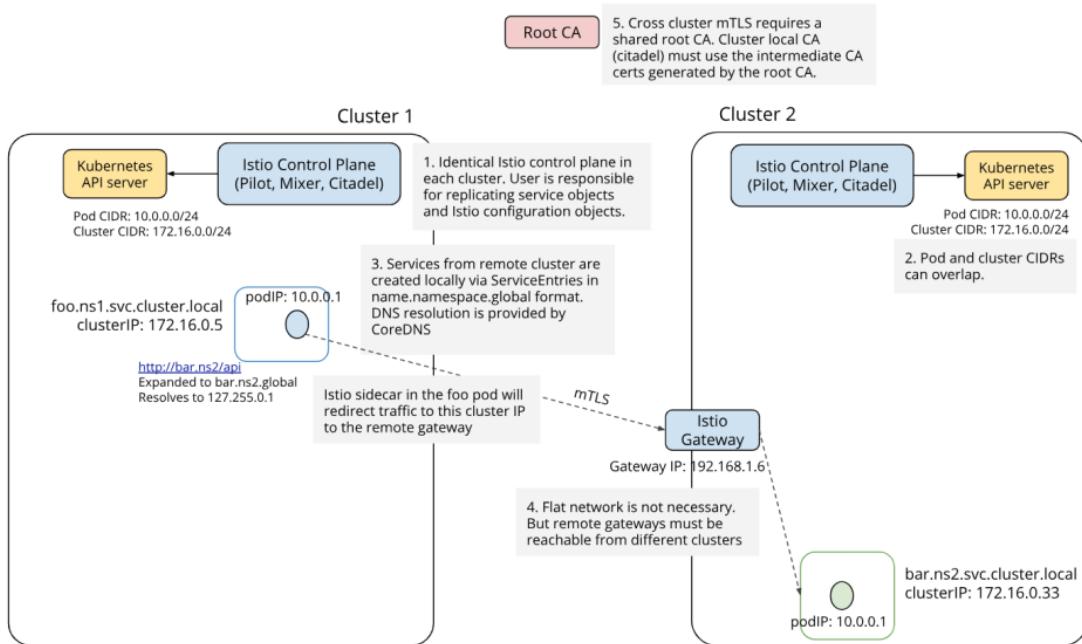
Envoy收集指标直接发给Prometheus. (没有重点讲解)

- 针对HTTP, HTTP/2和grpc协议，Istio收集以下指标

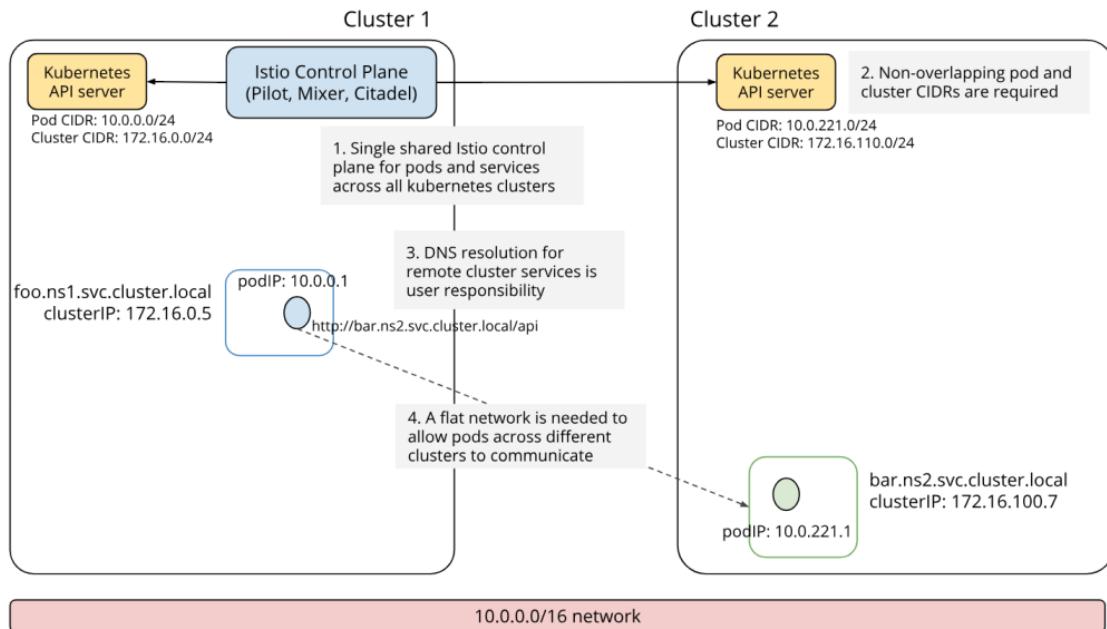
- Request Count (`istio_requests_total`)
 - 请总数
 - Request Duration (`istio_request_duration_milliseconds`)
 - 请求处理时长
 - Request Size (`istio_request_bytes`)
 - 请求包大小
 - Response Size (`istio_response_bytes`)
 - 响应包大小
- 针对TCP协议，Istio收集以下指标
- Tcp Byte Sent (`istio_tcp_sent_bytes_total`)
 - 发送的数据响应包的总大小
 - Tcp Byte Received (`istio_tcp_received_bytes_total`)
 - 接收到的数据包大小
 - Tcp Connections Opened (`istio_tcp_connections_opened_total`)
 - TCP 连接数
 - Tcp Connections Closed (`istio_tcp_connections_closed_total`)
 - 关闭的 TCP 连接数
 - 同时支持 WASM (Web Assembly) plugin 收集数据
 - 但启用 WASM 后资源开销显著上升

多集群

- 网络不互通时基于Gateway



- 基于VPN或者网络互通的多集群



如上图，左侧为Primary集群，右侧为Remote集群；

- Primary集群是完整的
- Remote集群只有Pilot，然后将自己的config交给Primary集群，这样Primary集群除了自己，还会watchRemote集群，然后建立各种xDS相关的对象。

跟踪采样tracing

1. 安装jaeger及对应服务

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: jaeger
  namespace: istio-system
```

```

labels:
  app: jaeger
spec:
  selector:
    matchLabels:
      app: jaeger
  template:
    metadata:
      labels:
        app: jaeger
      annotations:
        sidecar.istio.io/inject: "false"
        prometheus.io/scrape: "true"
        prometheus.io/port: "14269"
  spec:
    containers:
      - name: jaeger
        image: "docker.io/jaegertracing/all-in-one:1.23"
        env:
          - name: BADGER_Ephemeral
            value: "false"
          - name: SPAN_STORAGE_TYPE
            value: "badger"
          - name: BADGER_DIRECTORY_VALUE
            value: "/badger/data"
          - name: BADGER_DIRECTORY_KEY
            value: "/badger/key"
          - name: COLLECTOR_ZIPKIN_HOST_PORT
            value: ":9411"
          - name: MEMORY_MAX_TRACES
            value: "50000"
          - name: QUERY_BASE_PATH
            value: /jaeger
        livenessProbe:
          httpGet:
            path: /
            port: 14269
        readinessProbe:
          httpGet:
            path: /
            port: 14269
        volumeMounts:
          - name: data
            mountPath: /badger
        resources:
          requests:
            cpu: 10m
    volumes:
      - name: data
        emptyDir: {}
---
apiVersion: v1
kind: Service
metadata:
  name: tracing
  namespace: istio-system
  labels:
    app: jaeger

```

```

spec:
  type: ClusterIP
  ports:
    - name: http-query
      port: 80
      protocol: TCP
      targetPort: 16686
      # Note: Change port name if you add '--query.grpc.tls.enabled=true'
    - name: grpc-query
      port: 16685
      protocol: TCP
      targetPort: 16685
  selector:
    app: jaeger

```

2. Jaeger implements the Zipkin API. To support swapping out the tracing backend, we use a Service named Zipkin.

```

apiVersion: v1
kind: Service
metadata:
  labels:
    name: zipkin
    name: zipkin
    namespace: istio-system
spec:
  ports:
    - port: 9411
      targetPort: 9411
      name: http-query
  selector:
    app: jaeger

```

3. 部署jaeger-collector Service

```

apiVersion: v1
kind: Service
metadata:
  name: jaeger-collector
  namespace: istio-system
  labels:
    app: jaeger
spec:
  type: ClusterIP
  ports:
    - name: jaeger-collector-http
      port: 14268
      targetPort: 14268
      protocol: TCP
    - name: jaeger-collector-grpc
      port: 14250
      targetPort: 14250
      protocol: TCP
    - port: 9411
      targetPort: 9411
      name: http-zipkin

```

```
selector:  
  app: jaeger
```

4. 为了测试效果，采样率设置为100%

```
kubectl edit configmap istio -n istio-system  
set tracing.sampling=100
```

5. 在tracing空间中部署3个HTTP服务，调用链为service0→service1→service2

```
kubectl apply ns tracing  
kubectl label ns tracing istio-injection=enabled # 开启自动注入sidecar  
kubectl -n tracing apply -f service0.yaml  
kubectl -n tracing apply -f service1.yaml  
kubectl -n tracing apply -f service2.yaml
```

6. 通过Istio发布服务

```
apiVersion: networking.istio.io/v1beta1  
kind: VirtualService  
metadata:  
  name: service0  
spec:  
  gateways:  
    - service0  
  hosts:  
    - '*'  
  http:  
    - match:  
        - uri:  
            exact: /service0  
    route:  
      - destination:  
          host: service0  
          port:  
            number: 80  
---  
apiVersion: networking.istio.io/v1beta1  
kind: Gateway  
metadata:  
  name: service0  
spec:  
  selector:  
    istio: ingressgateway  
  servers:  
    - hosts:  
      - '*'  
    port:  
      name: http-service0  
      number: 80  
      protocol: HTTP
```

7. 发起请求

check ingress ip

```
k get svc -nistio-system  
istio-ingressgateway LoadBalancer $INGRESS_IP
```

access the tracing via ingress for 100 times(default sampling rate is 1%)

```
curl $INGRESS_IP/service0
```

check tracing dashboard

```
istioctl dashboard jaeger
```

在上面启动的dashboard里可以看到如下的结果

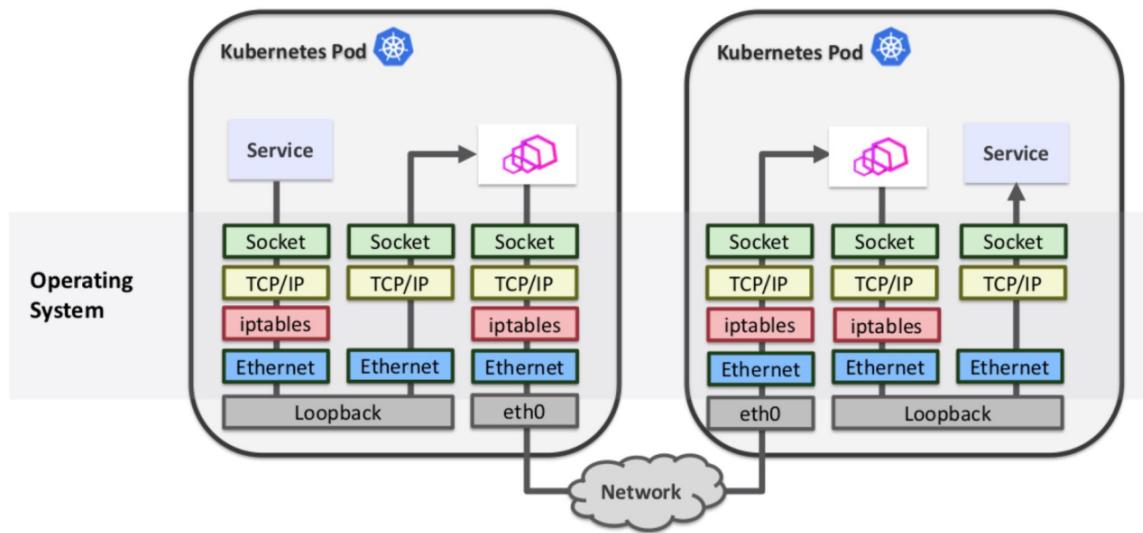


原理

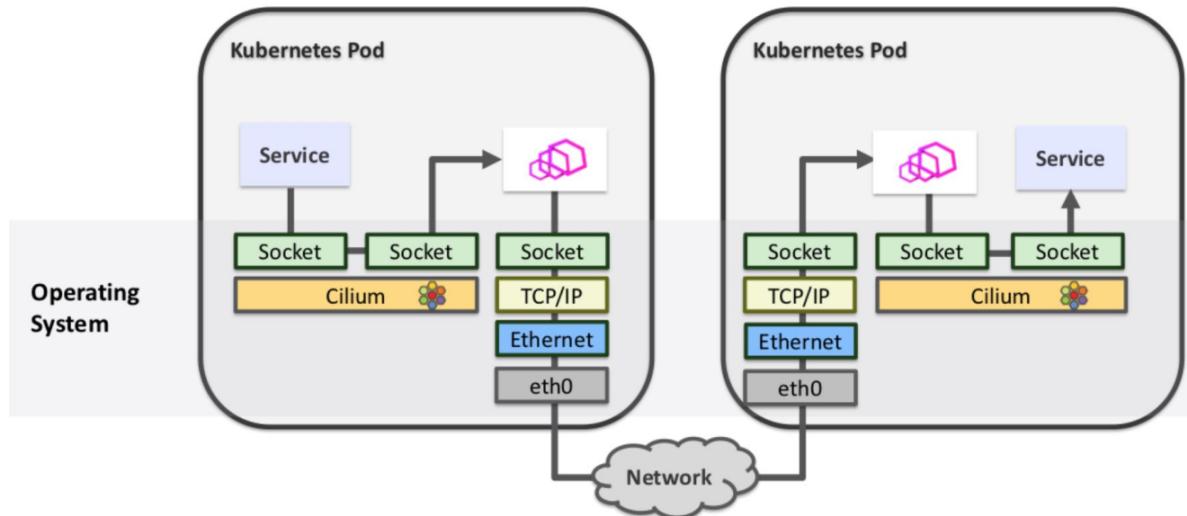
Istio自动生成了采样，但是需要链路上的所有服务把自动植入的HTTP头传递下去

- **x-request-id**
- **x-b3-traceid**
- **x-b3-spanid**
- **x-b3-parentspanid**
- **x-b3-sampled**
- **x-b3-flags**
- **x-ot-span-context**

Service Mesh涉及的网络栈及优化



可以看到多次出入内核网络协议栈，有一定的性能损失，可以在EBPF机制的hook点上做优化，缩短路径



生产部署的考虑

配置一致性检查

要考虑配置如果下发失败怎么检测以及处理。K8S的风格是最终一致性，如果应用对一致性有要求，就需要自己处理。

Endpoint健康检查

目前社区可直接使用的不多。

海量转发规则情况下的scalability

