go-chassis Documentation

xiaoliang

Jan 14, 2019

User Documentations

CHAPTER 1

Introductions

1.1 What is mesher

Mesher is a service mesh implementation based on go chassis. So it has all the features of go chassis like service discovery, load balancing, fault tolerance, route management,distributed tracing etc. it makes your service become resilience and observable

1.2 Concepts

1.2.1 Sidecar

Mesher leverage distributed design pattern, sidecar to work along with service.

1.2.2 go chassis

Mesher is a light weight sidecar proxy developed on top of go-chassis, so it has the same concepts with it and it has all features of go chassis

1.2.3 DestinationResolver

Destination Resolver parse request into a service name

1.2.4 Source Resolver

source resolver get remote IP and based on remote IP, it

1.2.5 Admin API

Listen on isolated port, it gives a way to interact with mesher

CHAPTER 2

Get started

2.1 Before you start

Before you start, you must know what you gonna do if you use mesher as your sidecar proxy.

Assume you launched 2 services, each of service has a dedicated mesher as sidecar proxy.

The network traffic will be: ServiceA->mesherA->mesherB->ServiceB.

To run mesher along with your services, you need to set minimum configurations as below:

1. Give mesher your service name in microservice.yaml file
2. Set service discovery service(service center, Istio etc) configurations in chassis.yaml
3. export HTTP_PROXY=http://127.0.0.1:30101 as your service runtime environment
4. (optional)Give mesher your service port list by ENV SERVICE_PORTS or CLI –service-ports

After the configurations, assume you serviceB is listening at 127.0.0.1:8080

the serviceA must use http://ServiceB:8080/{api_path} to access ServiceB

Now you can launch as many as serviceA and serviceB to make this system become a distributed system

Notice:

> consumer need to use http://provider_name:provider_port/ to access provider, instead of http://provider_ip:provider_port/. if you choose to set step4, then you can simply use http://provider_name/ to access provider

2.2 Quick start

2.2.1 Local

In this case, you will launch one mesher sidecar proxy and one service developed based on go-chassis as provider and use curl as a dummy consumer to access this service

the network traffic: curl->mesher->service

1.Install ServiceComb service-center

2.Install go-chassis and run rest server

1. Build and run, use go mod(go 1.11+, experimental but a recommended way)

```
cd mesher
GO111MODULE=on go mod download
#optional
GO111MODULE=on go mod vendor
go build mesher.go
./mesher
```

4.verify, in this case curl command is the consumer, mesher is consumer's sidecar, and rest server is provider

```
export http_proxy=http://127.0.0.1:30101
curl http://RESTServer:8083/sayhello/peter
```

Notice:

> You don't need to set service registry in chassis.yaml, because by default registry address is 127.0.0.1:30100, just same service center default listen address.

2.2.2 Run on different infrastructure

Mesher does not bind to any platform or infrastructures, plz refer to https://github.com/go-mesh/mesher-examples/tree/master/Infrastructure to know how to run mesher on different infra

2.2.3 Sidecar injector

Mesher supply a way to automatically inject mesher configurations in kubernetes

See detail https://github.com/go-chassis/sidecar-injector

CHAPTER 3

User guides

3.1 Mesher command Line

when you start mesher process, you can use mesher command line to specify configurations like below

```
mesher --config=mesher.yaml --service-ports=rest:8080
```

3.1.1 Options

–config

(optional, string) the path to mesher configuration file, default value is {current_bin_work_dir}/conf/mesher.yaml

–mode

(optional, string) mesher has 2 work mode, sidecar and per-host, default is sidecar

–service-ports

(optional, string) running as sidecar, mesher need to know local service ports, this is to tell mesher service port list, The value format is {protocol}-{suffix} or {protocol} if service has multiple protocol, you can separate with comma "rest-admin:8080,grpc:9000". default is empty, in that case mesher will use header X-Forwarded-Port as local service port, if it is empty also mesher can not communicate to your local service

3.2 Profile Mesher

Mesher has a convenience way to enable go pprof, so that you can easily analyze the performance of mesher

3.2.1 Configurations

```
pprof:
  enable: true
  listen: 127.0.0.0.1:6060
```

enable

(optional, bool) default is false

listen

(optional, string) Listen IP and port

3.3 Admin API

3.3.1 Configurations

admin api server leverage protocol server, it listens on isolated port, by default admin is enabled, and default value of goRuntimeMetrics is false.

To start api server, set protocol server config in chassis.yaml

```
cse:
   protocols:
     rest-admin:
       listenAddress: 0.0.0.0:30102  # listen addr for adminAPI
```

tune admin api in mesher.yaml

```
admin:
  enable: true
  goRuntimeMetrics : true # enable metrics
```

admin.enable

(optional, bool) default is false

admin.goRuntimeMetrics

(optional, bool) default is false, enable to expose go runtime metrics in /v1/mesher/metrics

3.4 Local Health check

you can use health checker to check local service health, when service instance is not healthy, mesher will update the instance status in registry service to "DOWN" so that other service can not discover this instance. If the service is healthy again, mesher will update status to "UP", then other instance can discover it again. currently this function works only when you use service center as registry

examples:

Check local http service

```
localHealthCheck:
  - port: 8080
    protocol: rest
    uri: /health
    interval: 30s
    match:
      status: 200
      body: ok
```

3.4.1 Options

port

(require, string) must be a port number, mesher is only responsible to check local service, it use 127.0.0.1:{port} to check service

protocol

(optional, string) mesher has a built-in checker "rest",for other protocol, will use default TCP checker unless you implement your own checker

uri

(optional, string) uri start with /.

interval

(optional, string) check interval, you can use number with unit: 1m, 10s.

match.status

(optional, string) the http response status must match status code

match.body

(optional, string) the http response body must match body

3.5 Destination Resolver

Destination Resolver is a module to parse each protocol request to get a target service name. you can write your own resolver implementation for different protocol.

3.5.1 Configurations

example

```
plugin:
  destinationResolver:
    http: host # host is a build-in and default resolver, it uses host name as␣
↪service name
    grpc: ip
```

plugin.destinationResolver

(optional, map) here you can define what kind of resolver, a protocol should use

CHAPTER 4

Protocols

4.1 gRPC Protocol

Mesher support gRPC protocol

4.1.1 Configurations

To enable gRPC proxy you must set the protocol config

```
cse:
  protocols:
    grpc:
      listenAddress: 127.0.0.1:40101 # or internalIP:port
```

4.1.2 How to use mesher as sidecar proxy

Assume you original client is

```
    conn, err := grpc.Dial("10.0.1.1:50051",
        grpc.WithInsecure(),
        )
```

set http_proxy

```
export http_proxy=http://127.0.0.1:40100
```

4.1.3 example

A gRPC example is here

CHAPTER 5

Use Istio as control plane

5.1 Get started

Istio Pilot can be configured as the service discovery component for mesher. By default the Pilot plugin is not compiled into mesher binary. To make mesher work with Pilot, import the plugin in mesher's entrypoint source code:

```
import _ "github.com/go-mesh/mesher/plugins/registry/istiov2"
```

Then the Pilot plugin will be installed when mesher starts. Next step, configure Pilot as service discovery in `chassis.yaml`:

```
cse:
  service:
    registry:
      registrator:
        disabled: true
      serviceDiscovery:
        type: pilotv2
        address: grpc://istio-pilot.istio-system:15010
```

Since mesher doesn't have to register the service to Pilot, the registrator config item should be disabled. Make serviceDiscovery.type to be pilotv2, to get service information by xDS v2 API(the v1 API is deprecated).

5.1.1 The routing tags in Istio

In the original mesher configuration, user can specify tag based route rules, as described below:

```
## router.yaml
router:
  infra: cse
routeRule:
  targetService:
```

(continues on next page)

(continued from previous page)

```
    - precedence: 2
      route:
      - tags:
          version: v1
        weight: 40
      - tags:
          version: v2
          debug: true
        weight: 40
      - tags:
          version: v3
        weight: 20
```

Then in a typical Istio environment, which is likely to be Kubernetes cluster, user can specify the DestinationRules for targetService with the same tags:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: targetService
spec:
  host: targetService
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
      debug: "true"
  - name: v3
    labels:
      version: v3
```

Notice that the subsets' tags are the same with those in `router.yaml`, then mesher's tag based load balancing strategy works as it originally does.

5.2 Discovery

5.2.1 Introduction

Istio Pilot can be integrated with Mesher, working as the Service Discovery component.

5.2.2 Configuration

edit chassis.yaml.

registrator.disabled

> Must disable registrator, because registrator is is used in client side discovery. mesher leverage server side discovery which is supported by kubernetes

serviceDiscovery.type

specify the discovery plugin type to "pilotv2", since Istio removes the xDS v1 API support from version 0.7.1, type "pilot" is deprecated.

serviceDiscovery.address

the pilot address, in a typical Istio environment, pilot usually listens on the grpc port 15010.

5.2.3 examples

```
cse: # Using xDS v2 API
  service:
    Registry:
      registrator:
        disabled: true
      serviceDiscovery:
        type: pilotv2
        address: grpc://istio-pilot.istio-system:15010
```

5.3 Route Rule

Instead of using CSE and route config to manage route, mesher supports Istio as a control plane to set route rule and follows the envoy API reference to manage route. This page gives the examples to show how requests are routed between micro services.

5.3.1 Mesher Configurations

In **Consumer** router.yaml, you can set router.infra to define which router plugin mesher fetches from. The default router.infra is cse, which means the routerule comes from route config in CSE config-center. If router.infra is set to be pilotv2, the router.address is necessary, such as the in-cluster istio-pilot grpc address.

Notice that `infra: pilot` is deprecated since Istio removes the xDS v1 API from 0.7.1

```
router:
  infra: pilotv2 # pilotv2 or cse
  address: grpc://istio-pilot.istio-system:15010
```

In **Both** consumer and provider registry configurations, the recommended one shows below.

```
cse:
  service:
    registry:
      registrator:
        disabled: true
      serviceDiscovery:
        type: pilotv2
        address: grpc://istio-pilot.istio-system:15010
```

5.3.2 Kubernetes Configurations

The provider applications of v1, v2 and v3 version could be deployed in kubernetes cluster as **Deployment** with differenent labels. The labels of version is necessary now, and you need to set env to generate nodeID in Istio system,

such as **POD_NAMESPACE, POD_NAME** and **INSTANCE_IP**.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    version: v1
    app: pilot
    name: istioserver
  name: istioserver-v1
  namespace: default
spec:
  progressDeadlineSeconds: 600
  replicas: 1
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      app: pilot
      version: v1
      name: istioserver
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: pilot
        version: v1
        name: istioserver
    spec:
      containers:
      - image: gosdk-istio-server:latest
        imagePullPolicy: Always
        name: istioserver-v1
        ports:
        - containerPort: 8084
          protocol: TCP
        resources: {}
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
        env:
        - name: CSE_SERVICE_CENTER
          value: grpc://istio-pilot.istio-system:15010
        - name: POD_NAME
          valueFrom:
            fieldRef:
              apiVersion: v1
              fieldPath: metadata.name
        - name: POD_NAMESPACE
          valueFrom:
            fieldRef:
              apiVersion: v1
              fieldPath: metadata.namespace
        - name: INSTANCE_IP
          valueFrom:
            fieldRef:
```

(continues on next page)

(continued from previous page)

```
            apiVersion: v1
            fieldPath: status.podIP
      volumeMounts:
      - mountPath: /etc/certs/
        name: istio-certs
        readOnly: true
    dnsPolicy: ClusterFirst
    initContainers:
    restartPolicy: Always
    schedulerName: default-scheduler
    securityContext: {}
    terminationGracePeriodSeconds: 30
    volumes:
    - name: istio-certs
      secret:
        defaultMode: 420
        optional: true
        secretName: istio.default
```

5.3.3 Istio v1alpha3 router configurations

Traffic-management gives references and examples of Istio new router rule schema. First, subsets is defined according to labels. Then you can set route rule of differenent weight for virtual services.

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: istioserver
spec:
  host: istioserver
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
  - name: v3
    labels:
      version: v3
```

NOTICE: The subsets only support labels of version to distinguish differenent virtual services, this constrains will canceled later.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: istioserver
spec:
  hosts:
    - istioserver
  http:
  - route:
    - destination:
        host: istioserver
```

(continues on next page)

(continued from previous page)

```
        subset: v1
      weight: 25
    - destination:
        host: istioserver
        subset: v2
      weight: 25
    - destination:
        host: istioserver
        subset: v3
      weight: 50
```

5.4 Egress

5.4.1 Introduction

Mesher support Egress for your service, so that you can access any publicly accessible service from your microservice.

5.4.2 Configuration

The egress related configurations is all in egress.yaml.

infra

(optional, string) specifies from where the egress configuration need to be taken supports two values cse or pilot , cse means the egress configuration from egress.yaml file, pilot means egress configuaration are taken from pilot of istio, default is *cse*

address

(optional, string) The end point of pilot from which configuration need to be fetched.

hosts

(optional, []string) host associated with external service, could be a DNS name with wildcard prefix

ports.port

(optional, int) The port associated with the external service, default is *80*

ports.protocol

(optional, int) The protocol associated with the external service,supports only http default is *HTTP*

5.4.3 example

edit egress.yaml

```
egress:
  infra: cse  # pilot or cse
  address: http://istio-pilot.istio-system:15010
egressRule:
  google-ext:
    - hosts:
        - "www.google.com"
```

(continues on next page)

(continued from previous page)

```
      - "*.yahoo.com"
    ports:
      - port: 80
        protocol: HTTP
```

CHAPTER 6

Sidcar-injector Deployment and Usage

6.1 Introduction

Sidecar is a way to run alongside your service as a second process. The role of the sidecar is to augment and improve the application container, often without the application container's knowledge.

sidecar is a pattern of "Single-node, multi container application".

This pattern is particularly useful when using kubernetes as container orchestration platform. Kubernetes uses pods. A pod is composed of one or more application containers. A sidecar is a utility container in the pod and its purpose is to support the main container. It is important to note that standalone sidecar doesnot serve any purpose, it must be paired with one or more main containers. Generally, sidecar container is reusable and can be paired with numerous type of main containers.

For design pattern please refer

Container Design Pattern

Example: The main container might be a web server, and it might be paired with a "logsaver" sidecar container that collects the web server's logs from local disk and streams them to a cluster.

6.2 Injection

Two types

1. Manual sidecar injection
2. Automatic sidecar injection

6.3 Manual sidecar injection

In manual sidecar injection user has to provide sidecar information in deployment.

```
kubectl get deployment client -o wide
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE       CONTAINERS
client    1         1         1            1           13s       client,mesher
```

6.4 Automatic sidecar injection

Sidecars can be automatically added to applicable Kubernetes pods using

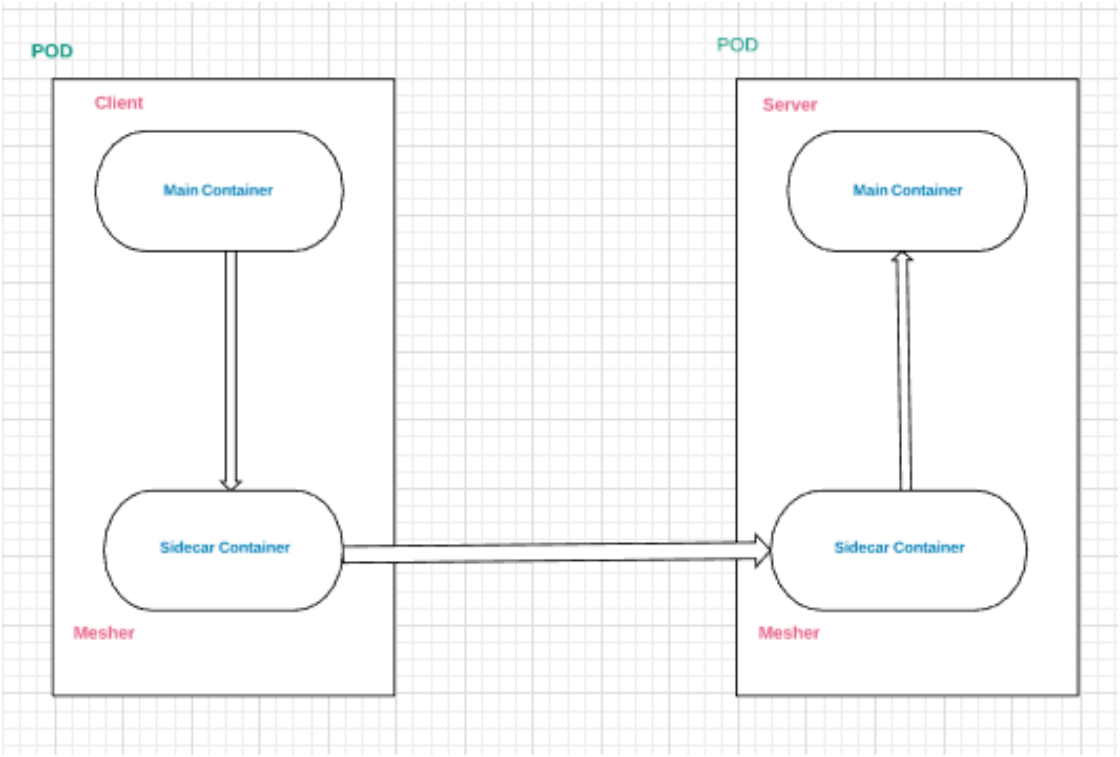
mutating webhook admission controller Note that unlike manual injection, automatic injection occurs at the pod-level. You won't see any change to the deployment itself.

Verify

6.5 How it works

sidecar will deploy along side with main container as shown below

The figure shows the client and server communication using mesher as a sidecar.



Explanation:

Mesher is deployed as a sidecar along with main container of server and client in a pod.

client and server will implement some rest api's and functionalities like loadbalance, circuit-breaker, fault-injection, routerule, discovery etc... will be provided by mesher(sidecar).

workflow:

user/curl——>client(main container)——>mesher(sidecar container)——>mesher(sidecar container)——>server(main container).

6.6 Deployment Of Sidecar-Injector

Prerequisites

Quick start

Use below links to build and Install sidecar

build install

6.7 Annotations

Refer k8s document

Annotation

6.8 Deployment of application

The Sidecar-injector will automatically inject mesher containers into your application pods.

Following are the annotations used to inject mesher sidecar into the user pod

1. sidecar.mesher.io/inject:

   The allowed values are “yes” or “y”

   If “yes” or “y” provided the sidecar will inject in the main container. If not, sidecar will not inject in the main container.

2. sidecar.mesher.io/discoveryType:

   The allowed values are “sc” and “pilot”

   If value is “sc” it will use serviecComb service-center as a registry and discovery. If value is “pilot” it will use the istio pilot as a discovery.

3. sidecar.mesher.io/servicePorts:

   serviceports are the port values of actual main server container append with “rest or grps”

   ex: sidecar.mesher.io/servicePorts: rest:9999

Required annotation for client and server sidecar.mesher.io/inject:

Optional annotation for client and server sidecar.mesher.io/discoveryType:

Optional annotation for server sidecar.mesher.io/servicePorts:

6.9 Prerequisites before deploying application

Label the chassis namespace with sidecar-injector=enabled

kubectl label namespace chassis sidecar-injector=enabled

kubectl get namespace -L sidecar-injector

```
NAME          STATUS    AGE       SIDECAR-INJECTOR
default       Active    18h
kube-public   Active    18h
kube-system   Active    18h
chassis       Active    3m        enabled
```

6.10 Usage of istio

To use istio following are the required annotation to be given in client and server yaml file sidecar.mesher.io/inject: “yes” and sidecar.mesher.io/discoveryType:”pilot”

Example to use pilot registry

deploy the examples using kubectl command line

kubectl create -f <filename.yaml> -n chassis

6.11 Usage of serviceComb

To use service-center following are the required annotation to be given in client and server yaml file sidecar.mesher.io/inject: “yes” and sidecar.mesher.io/discoveryType:”sc”

Example to use sc registry

deploy the examples using kubectl command line

kubectl create -f <filename.yaml> -n chassis

6.12 Verification

Follow