Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería Telemática

Trabajo Fin de Grado

Despliegue y gestión de microservicios usando Istio

Autor: Cristian Camilo Morales Tapias
Tutor/es: Óscar García Población

2021/2022

UNIVERSIDAD DE ALCALÁ
Escuela Politécnica Superior

Grado en Ingeniería Telemática

Despliegue y gestión de microservicios usando Istio

Autor: *Cristian Camilo Morales Tapias*
Tutor: *Óscar García Población*

Comite:

Presidente:	*Julia María Clemente Párraga*
Vocal 1st:	*Concepción Batanero Ochaíta*
Tutor:	*Óscar García Población*

2021/2022

Resumen

El objetivo del proyecto es realizar una prueba de concepto de las ventajas que aporta la herramienta `Istio` en el despliegue de una aplicación web usando una arquitectura basada en microservicios sobre un cluster de `kubernetes`. Se utilizará `GCP(Google Cloud Platform)` para aprovecharnos del servicio `GKE` el cual proporciona un entorno administrado para implementar, administrar y escalar las aplicaciones en contenedores mediante la infraestructura de Google con el sistema de administración de código abierto denominado `Kubernetes`. Dado que se analizará las características de `Istio` en entornos cloud se utilizará una herramienta de automatización de infraestructura llamado `Terraform`.

Se ha implementado una arquitecturas basada en microservicios las cuales se basaran en imágenes `docker` para poder analizar las distintas características que aporta `Istio` por separado. Los microservicios se comunicarán bajo la supervisión de `Istio` añadiendo una capa de abstracción a `Kubernetes` y así tener mas información y control sobre las comunicaciones.

Se estudiará y se pondrá a prueba las características principales de `Istio` que son:

- Administración de tráfico.
- Observabilidad a través de la monitorización y recopilación de métricas de red.
- Seguridad en la comunicación entre los microservicios.

Summary

The objective of the project is to carry out a proof of concept of the advantages provided by Istio tool in the deployment of a web application using an architecture based on in microservices on a Kubernetes cluster. GCP(Google Cloud Platform) will be used to take advantage of the GKE service which provides a managed environment to deploy, manage, and scale containerized applications using Google's infrastructure with the open source management system called Kubernetes. Since Istio's features will be analyzed in cloud environments, an infrastructure as a code automation tool called Terraform will be used.

An architecture based on microservices has been implemented, which will be based on in docker images to be able to analyze the different characteristics that Istio provides. The microservices will communicate under the supervision of Istio by adding a abstraction layer to Kubernetes and thus have more information and control over the communications.

The main features of Istio will be studied and tested, which are:

- Traffic management.
- Observability through monitoring and collection of network metrics.
- Security in communication between microservices.

Tabla de contenidos

Listado de Figuras

Listado de Tablas

Chapter 1

Introducción

Hoy en día se exige que las aplicaciones sean rápidas y eficientes. Si el cliente dispone de una red de alta velocidad queda en manos de la aplicación de proporcionar estas características. Para poder otorgar un buen servicio es imprescindible elegir la arquitectura de software más apropiada. Las arquitecturas más populares son, arquitectura monolítica y arquitectura basada en microservicios.

1.1 Arquitectura monolítica

La tendencia en el diseño de aplicaciones es seguir una arquitectura *monolítica*[1], esto quiere decir que la aplicación entera se desarrollaba como una sola. Todo el software que comprende la aplicación está en una única pieza por lo que todos los componentes de una arquitectura monolítica están interconectadas y son interdependientes. La figura 1.1 muestra una aplicación estilo `ecommerce` la cuál sigue una arquitectura monolítica, aunque las funcionalidades de la aplicación (autenticación, carrito, pedidos, pagos etc) estén separadas en el diagrama, estas se desplegarían como un único programa.

Los sistemas monolíticos están compuestos de procesos acoplamiento muy dependiente entre ellos, esto quiere decir que algún cambio sobre un proceso podría poner en peligro el rendimiento de la aplicación entera, en el mercado actual esto es una gran desventaja ya que la demanda exige una constante innovación y adaptación.

Con el paso del tiempo el mantenimiento de la aplicación se transforma en un desafío técnico dado que el código crece y se convierte cada vez más complejo y teniendo en cuenta que todo esta estrechamente interconectado puede llegar a ser una pesadilla. Esto provoca que el desarrollo sea cada vez mas lento y reduce la frecuencia de actualización y

[1] De acuerdo con la RAE el adjetivo monolítico significa inconmovible, rígido, inflexible.

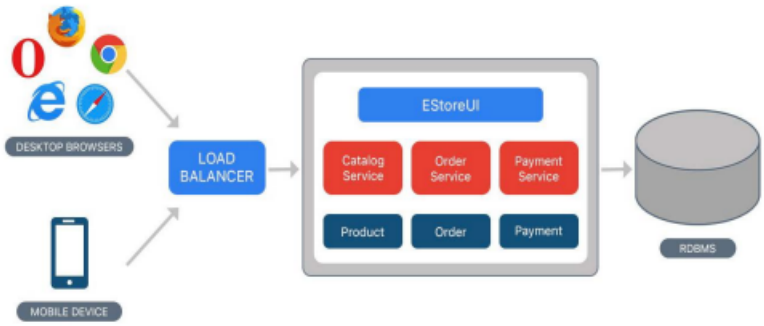


Figure 1.1: Ecommerce: Arquitectura monolítica.

creación de nuevas funcionalidades, ya que para actualizar un único componente hay que re-desplegar la aplicación entera. La escalabilidad de una arquitectura monolítica puede realizarse de una manera muy sencilla, simplemente habría que instalar la aplicación en varios servidores y repartir el tráfico de entrada entre ellos. Pero esto es una manera de escalar poco eficiente porque por ejemplo, si la demanda aumenta y únicamente fuera necesario escalar el servicio de carrito de compra usando esta arquitectura estaríamos escalando todos los servicios.

No todo son desventajas en las arquitecturas monolíticas, sus principales ventajas son:

- **Simples de desarrollar**: Como todo esta desarrollado en el mismo lugar esto facilita a la comunicación entre las funcionalidades lógicas de la aplicación por lo que es muy fácil, mediante un grupo pequeño de desarrolladores crear una aplicación y ponerla en producción.
- **Fáciles de escalar**. A priori es muy fácil[2] de escalar ya que lo único que habría que hacer es instalar la aplicación en varios servidores y balancear el tráfico que les llegue.
- **Sencillas de depurar**. Ya que todo el código se encuentra en el mismo sitio por lo que se pueden realizar la mayoría de pruebas sin depender de nada mas.

[2]Es una manera ineficiente de escalar porque puede que sea solo un componente el que necesite recursos y siguiendo este método escalamos todo mal gastando recursos.

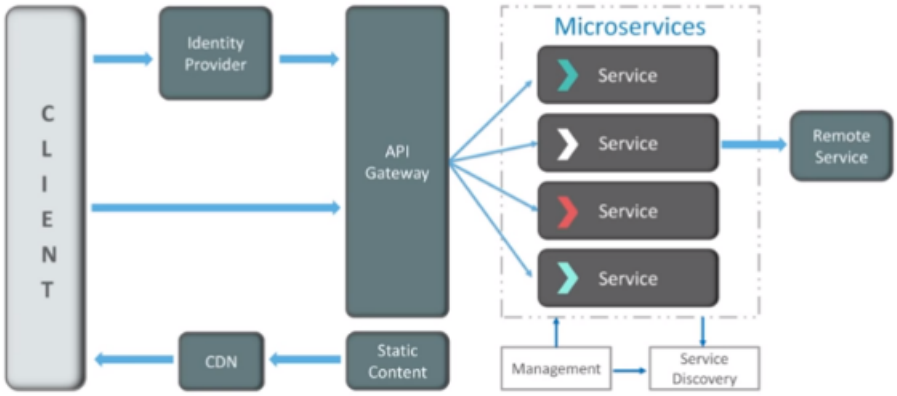


Figure 1.2: Ecommerce: Arquitectura basada en microservicios.

1.2 Arquitectura basada en microservicios

La idea principal detrás de los microservicios es que hay tipos de aplicaciones como por ejemplo, páginas web que son más fáciles de desarrollarlas y mantenerlas si dividimos la aplicación en pequeñas piezas que trabajen entre si. Cada pieza lleva su propio continuo desarrollo y mantenimiento, la aplicación final se convierte en la suma de todos los componentes, al contrario que las aplicaciones monolíticas que son desarrolladas todo en una pieza. Estos servicios se comunican entre si mediante mecanismos ligeros como podría ser `APIs` de recursos `HTTP`.

1.2.1 Componentes

Para el correcto funcionamiento de las arquitecturas basadas en microservicios es necesario los siguientes componentes: servicios, administración, descubridor de servicios, `API gateway`, estos elementos se pueden observar en la figura 1.2.

Servicios

Los servicios son independientes entre ellos y débilmente acoplados, cada uno de estos microservicios tiene una función única que generalmente es gestionada por un equipo pequeño de desarrollo, los servicios se despliegan de manera independiente.

Administración

El componente de administración es responsable de desplegar los servicios en nodos, identificando errores, balanceando los servicios entre los nodos disponibles. Este componente se asegura de que los servicios estén en los estados deseados, si un servicio no se encuentra disponible lo eliminará y lo desplegará de nuevo automáticamente.

Descubridor de servicios

Este componente se encarga de mantener almacenado la lista de los servicios activos y los nodos en los que actualmente estén. Este componente es importantísimo ya que los servicios están continuamente siendo balanceados y escalados entre los nodos según la demanda de que haya por lo que hay momentos en los que no debe recibir tráfico.

`API Gateway`

El `API Gateway` es la puerta de acceso que utiliza el cliente para poder comunicarse con los servicios. El cliente no se comunicará directamente con los servicios, seria muy tedioso que tuviera que tener almacenado la dirección IP de todos los servicios, además que esta va cambiando ya que son efímeras. Este componente redirige la petición del cliente al servicio apropiado. Las principales ventajas de la arquitectura basada en microservicios son:

- **Independencia de desarrollo**: Los equipos pequeños son más eficientes trabajando en paralelo que equipos grandes y gracias a este enfoque los desarrolladores de un servicio no tiene porque entender el funcionamiento de los demás servicios pudiendo así centrarse únicamente en el servicio que tenga delegado. Además cada servicio puede implementarse en el lenguaje de programación que más sea apropiado para su funcionamiento ya que las comunicaciones entre ellos serán mediante protocolos estandarizados como `HTTP`.

- **Aislamiento y Resistencia**:Si un servicio se detiene por cualquier motivo el resto de la aplicación seguirá funcionando y solo se tendrá que relanzar el servicio que esta mal funcionando.

- **Escalabilidad**. Los servicios pequeños necesitan pocos recursos y pueden ser escalables fácilmente dependiendo de la demanda que haya en el momento la cuál puede ir cambiando a lo largo del tiempo y a diferencia que en las arquitecturas

monolíticas con este escalado no se malgastaran recursos.

1.2.2 Nuevos retos

A primera vista puede parecer que la arquitectura basada en microservicios es ideal y perfecta pero aunque aporte muchos beneficios como todo en esta vida, también introduce nuevos desafíos. Cuando se migra a una arquitectura basada en microservicios se introducen los siguientes retos que no existían en las arquitecturas monolíticas.

Comunicación

Los servicios necesitan comunicarse entre ellos para que la aplicación funcione por lo que tienen que conocer los `endpoints` de todos los servicios, como sabemos, continuamente los servicios son balanceados entre los nodos por lo que sus IPs son efímeras. Esto significa, que además de la lógica del funcionamiento del servicio habría que configurar como desarrollar la comunicación entre ellos. Las comunicaciones se realizaran a través de una red por lo que habrá nuevos puntos de fallo.

Seguridad

Por lo general en una estructura basada en microservicios, estos están protegidos a través de un cortafuegos y/o un `proxy` como punto de entrada por lo que alrededor de los microservicios hay una capa de seguridad pero una vez dentro los servicios se comunican usando protocolos no seguros como `HTTP`, por lo que si, un atacante consigue entrar puede llegar a ser muy peligroso. Puede que para pequeñas aplicaciones que no tengan datos sensibles de usuarios no sea mayor problema pero para aplicaciones como banca `online`, comercio electrónico, farmacéuticas etc. Es muy importante asegurarse que todas las comunicaciones se hagan de bajo protocolos seguros y así reducir el riesgo de posibles ciberataques.

Automatización

Hay que invertir muchos recursos en herramientas de automatización ya que para lo que antes era una simple máquina con toda la lógica dentro de ella, ahora se ha podido convertir en cientos de servicios que hay que desplegar y obviamente no debemos ir desplegando cada uno manualmente e ir verificando continuamente su estado para saber si hay que desplegar otra vez. Es necesario mantener una monitorización del estado y métricas de cada servicio para así controlar aspectos como la tolerancia a fallos, la latencia, el reparto

de tráfico etc.

1.2.3 Kubernetes

`Kubernetes` es una herramienta de código abierto destina a la orquestación de contenedores la cuál fue originalmente desarrollada por `Google`. Su principal función es gestionar aplicaciones que estén compuesta de un gran número de contenedores `Docker` en distintos entornos como máquina físicas, máquinas virtuales, entornos `cloud` o híbridos.

Tras el incremento de las arquitecturas basadas en microservicios se ha tomado como estándar utilizar la herramienta `Docker` para aislar los microservicios en contenedores. A medida que las aplicaciones crecen, se crean más y más microservicios, lo que supone gestionar cargas de trabajo compuesta por cientos o por miles de contenedores. La gestión de estas unidades de trabajo en distintos entornos mediante herramientas propias de automatización puede llegar a ser una tarea muy tediosa por lo que se desarrollo `Kubernetes` la cual se encarga de esta gestión.

`Kubernetes` como herramienta orquestadora de contenedores ofrece:

1. Alta disponibilidad: Los nodos pueden estar es distintas zonas geográficas, en el caso del servicio GKE que Google ofrece se puede configurar para que sea zonal y todos los nodos estén bajo la misma zona geográfica por ejemplo si se configura en la zona europe-west1 significara que los nodos se repartirán en unos `Datacenters` que hay Belgica en cambio si se configura regionalmente los nodos se repartirán por distintos `Datacenters` por lo que si uno de ellos falla el cluster de `Kubernetes` seguirá funcionando correctamente.

2. Rápida escalabilidad: Según la carga a la que el cluster este sometida automáticamente aumentara o disminuirá el numero de nodos que lo componen para hacer frente a los picos de carga.

3. Plan de restauración: El `cluster` en todo momento guardara un estado de todos los elementos que lo componen por si fuera necesario restaurarlo en otro `cluster`.

Estructura de Kubernetes

Un cluster de `Kubernetes` esta compuesto de al menos un nodo `máster` conectado como mínimo a dos nodos `workers` en los cuales se ejecuta un proceso llamado `kubelet`, este proceso hace posible la comunicación entre los nodos y ejecuta tareas sobre los `workers`. En cada nodo `worker` se ejecutan contenedores `Docker` de distintas aplicaciones y dependiendo de cómo la carga de trabajo esté distribuida entre los nodos habrá distintos

números de contenedores `Docker` en ejecución. En el nodo `máster` se ejecutan varios procesos de `Kubernetes` para la gestión del cluster, estos procesos son:

1. API server: Es el punto de entrada al cluster de `Kubernetes` por el cual se definirá distintas configuraciones a través de una interfaz gráfica, `API` o `CLI`.

2. Controller Manager: Este proceso se encarga de que el cluster de `Kubernetes` este en el estado deseado revisando continuamente si hay algún proceso que este mal funcionando y necesite ser reparado por ejemplo, si un contenedor muere y necesita ser reiniciado.

3. scheduler: Este proceso se encarga de repartir la carga entre los nodos por lo que continuamente va decidiendo sobre que nodos deben de arrancar nuevos contenedores y si hace falta moverlos a un nodo que tenga mas recursos disponibles.

4. etcd: Este proceso se encarga de guardar continuamente el estado del cluster de `Kubernetes` por lo que contiene toda la configuración y estado de que cada nodo y contenedor en caso de que sea necesario restaurar este cluster de `Kubernetes`.

En la figura 1.3 podremos observar como están distribuidos los componentes previamente descritos.

Conceptos básicos de Kubernetes

Es importante tener claro algunos conceptos básicos de `Kubernetes` como los pods, servicios etc para así entender como Istio coopera con `Kubernetes`. Un `Pod` es un grupo de uno o más contenedores, con almacenamiento compartido y recursos de red, y una especificación sobre cómo ejecutar los contenedores que residen dentro de el. En estos `pods` estarán los contenedores de los cuales se componen las arquitecturas basadas en microservicios donde cada contenedor/pod corresponderá con un microservicio.

Cada pod tendrá una IP asociada pero dado que estos recursos son efímeros al ser eliminados y recreados en otro nodo se les asociara otra IP, aquí es donde entre en juego los `services` de `Kubernetes` ya que son los encargados de tener en todo momento las IPs de los `pods` a los que este configurado para que otros `Pods` pendan enviarle peticiones.

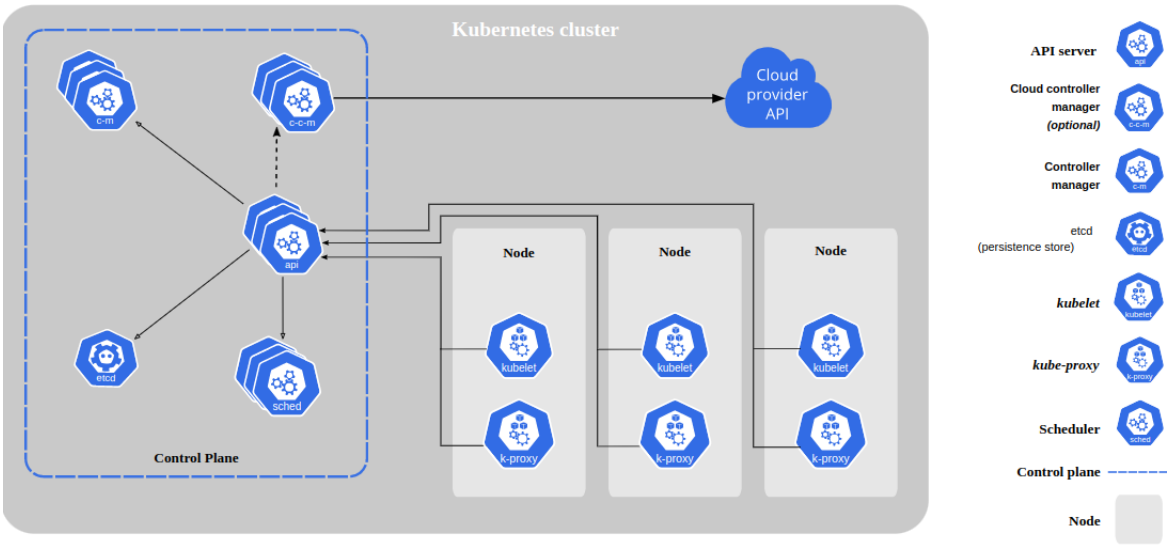


Figure 1.3: Diagrama de los componentes que componen Kubernetes.

Estructura de archivos de configuración

Hay una gran variedad de recursos en `Kubernetes` para los distintos requisitos que necesitemos en nuestras arquitecturas pero la configuración de estos recursos siguen una misma estructura. Esta compuesto de tres elementos:

1. `Kind`: Define que tipo de recurso de kubernetes se va a configurar este puede ser un `pod, service, deployment, statefulset etc...`
2. `Metadata`: Define metadatos como el nombre y etiquetas.
3. `Specification`: Define los parámetros de configuración dependiendo del tipo de recurso que se haya definido en el primer elemento `Kind`.

A continuación se mostrara la configuración de un `Pod`.

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4   name: example-pod
5   labels:
6      role: myrole
7  spec:
8    containers:
9      - name: web
10     image: nginx
```

```
11      ports:
12          - name: web
13          containerPort: 80
14          protocol: TCP
```

Chapter 2

Introducion a Istio

Istio es un `Service Mesh`, es decir, una herramienta para gestionar comunicación entre microservicios y conseguir que esta sea segura, flexible, observable y controlable. En un entorno de desarrollo, lo ideal, es que por cada microservicio los programadores solo se centre en la lógica de funcionamiento del mismo y así simplificar el trabajo, pero estos servicios se comunican entre ellos a través de la red. Esto añade complejidad en la configuración de cada microservicio.

Para poder desplegar una aplicación usando una arquitectura de microservicios hay que prestar atención a muchas más cosas además de la funcionalidad por lo que Istio como herramienta `Service Mesh` reduce la carga de trabajo.

2.1 Componentes de Istio

Los componentes que forman Istio se dividen en dos áreas. Como se observa en la figura 2.1 estos componentes son el plano de datos y el plano de control los cuales se describen a continuación.

2.1.1 Plano de datos

El plano de datos esta implementado de manera que intercepta todo todo el tráfico de entrada y salida de cada microservicio sin que este necesite saber de la existencia de Istio. Esto es gracias al `istio-proxy` que se despliega en cada `pod` ejecutándose como un `sidecar container`. Cada vez que un microservicio quiera comunicarse lo hará a través del `proxy` que actúa como intermediario con el cual se pueden añadir funcionalidades como reintentos automáticos, `circuit breaker`, descubridor de servicios, seguridad y mucho

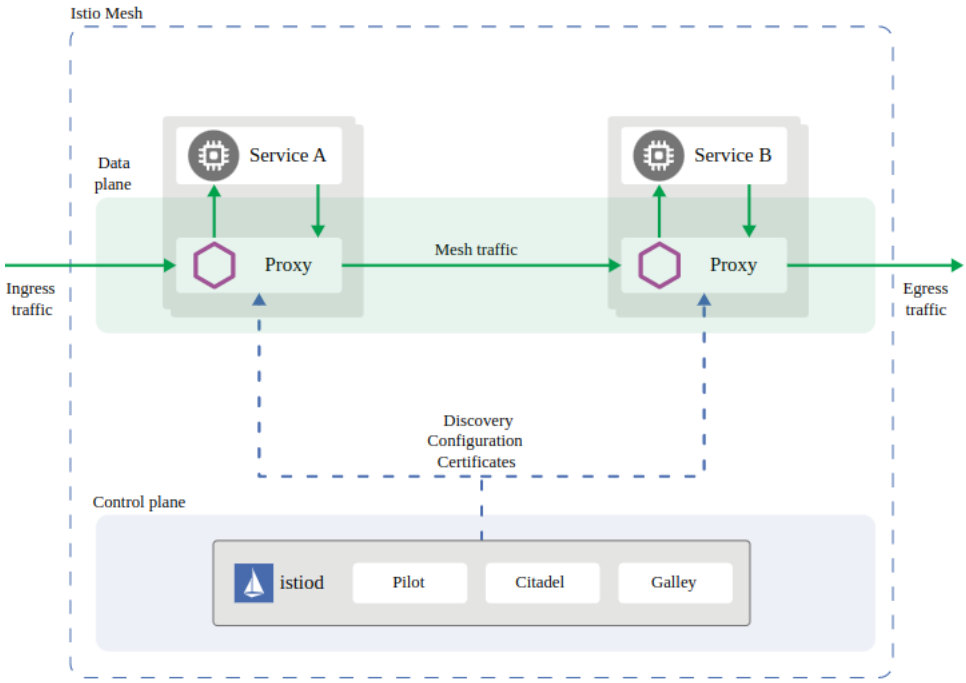


Figure 2.1: Estructura de Istio, plano de control y plano de datos [5].

más. Por defecto, el `istio-proxy` esta basado en un `envoy-proxy`. Hay una gran variedad de `proxys` en el mercado como `NGINX`, `HAProxy`, **Envoy**, Traefik, Amazon ALB etc. La herramienta Istio hace uso del **Envoy**. A continuación, se describe su funcionamiento.

Envoy proxy

Un `envoy` es un proxy basado en la capa 7(capa de aplicación del modelo OSI) desarrollado por la compañía `Lyft` la cual usa estos proxys para encargarse de millones de peticiones por segundo. Este `proxy` esta escrito en `C++`, ha sido altamente testeado proporcionando un alto rendimiento además de ser ligero. Proporciona características como el balanceo de carga para `HTTP1.1`, `HTTP2` y `gRCP`. Istio lleva las capacidades de `proxy` a lo más cercanas posible al código de la aplicación a través de una técnica de despliegue conocida como `sidecar` la cual consiste en desplegar el `envoy proxy` en el mismo `pod` en el cuál reside la aplicación. Es importante tener claro que es un `proxy` para poder entender el funcionamiento de Istio. Un `proxy` es un componente intermediario en una arquitectura de red en la que se sitúa en medio de las comunicaciones por lo que todo el tráfico que vaya a una aplicación tendrá que pasar por el `proxy` y este la procesará según esté configurado, posteriormente se la enviará al `backend` correspondiente, de la misma manera la respuesta del `backend` también pasará por el `proxy`. Los `proxys` simplifican lo

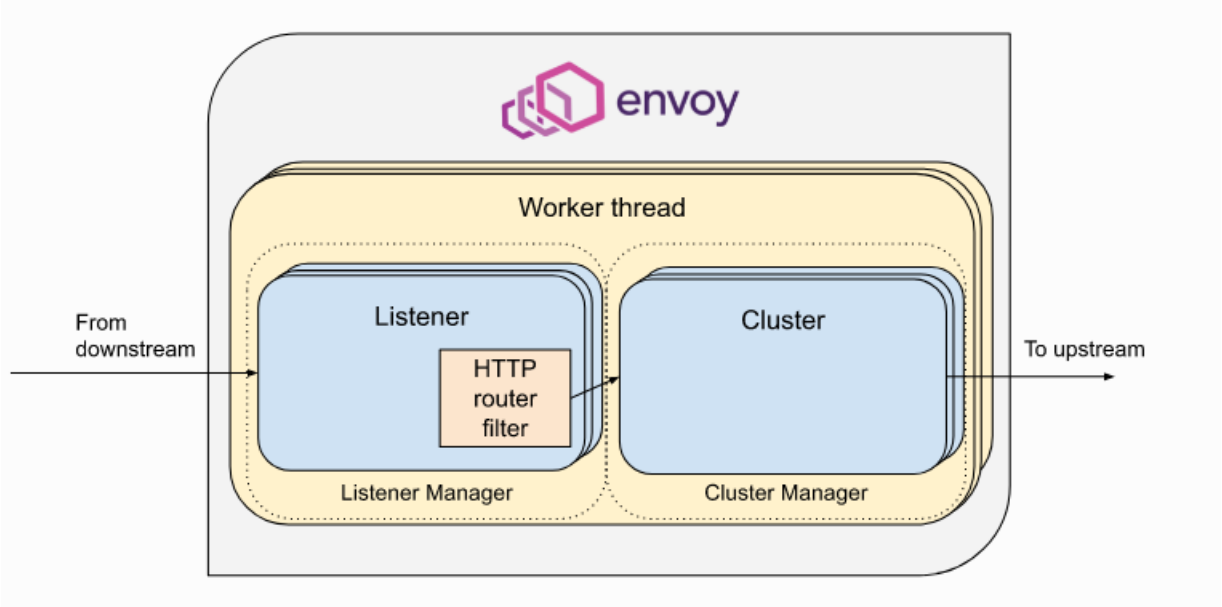


Figure 2.2: Estructura del proxy envoy [6].

que un cliente necesita saber para poder comunicarse con una aplicación y protegen los `backends` de ser sobrecargados y que así tengan un correcto funcionamiento

Un `envoy proxy` esta compuesto de 3 componentes lógicos:

1. *Listeners*: Expone un puerto por el cual acepta trafico de las aplicaciones del exterior.
2. *Routes* Defines las operaciones a realizar sobre el trafico que entra por los *listeners*. Con esta configuración obtenemos la gran mayoría de funcionalidades que proporciona Istio del control de las comunicaciones entre los microservicios
3. *Clusters*: Los backends a los que se enviara el trafico una vez procesado el tráfico.

En la imagen 2.2 podemos observar la estructura del `envoy proxy`.

Haciendo uso de `Envoy proxy` ganamos las siguientes funcionalidades:

1. *Localizador de servicios*:
   `Envoy` se puede configurar para que automáticamente busque `endpoints` de un `discovery API`, la aplicación que se ejecuta en el microservicio no sabrá ni le interesa saber como estos `backends` son localizados. El `discovery API` es simplemente un `REST API` que se usa para agrupar `backends` comunes.

2. *Balanceador de carga*:
   `Envoy` dispone de varios algoritmos de balanceo de carga como `round robin, weighted round robin, consistent hashing`, aleatorio etc.

3. *Enrutamiento de trafico*:
   `Envoy` entiende protocolos de aplicación como `HTTP1.1` y `HTTP2` puede redirigir el tráfico a los distintos `backends` según los valores de las cabeceras de estos protocolos. No solo redirige según las cabeceras sino que puede aplicar distintas políticas como reintentos de peticiones.

4. *Clonación de tráfico*: Con `envoy` se puede hacer una copia del tráfico entrante y redirigir esta copia a `backend` no críticos y así poder estudiar su comportamiento con tráfico real.

5. *Resiliencia de red*:
   Dado que la comunicación entre los microservicios irá por una red esta puede experimentar fallos en momentos de sobrecarga o simplemente en momentos aislados, su rendimiento se puede ver afectado por causas ajenas a la lógica de estos microservicios por lo que es importante tener un sistema preparado para estos incidentes. Usando `Envoy proxy` se puede configurar políticas de reintentos y/o políticas de tiempos de espera. El procedimiento de reintentos es muy útil cuando se experimenta inestabilidad aislada en la red pero puede ser peligrosa si no se usa adecuadamente, por ejemplo si un microservicio falla en un 10% de las peticiones por algún tipo de fallo lógico esto podría ser ocultado mediante un número de reintentos entre más alto sea el número de reintentos menos posibilidad de fallo, esto, a su vez incrementaría la carga sobre el microservicio, ya que por cada petición, puede que en realidad tenga que estar procesando un mayor número de peticiones lo cuál terminaría sobrecargando el microservicio. Aunque el usuario final no sea afectado por estos reintentos ya al realizar petición obtiene el resultado deseado esta característica no se tiene que usar para ocultar problemas en el funcionamiento del microservicio ya que lo único que conseguirá es ocultar un fallo en la lógica de la aplicación que un futuro desencadenara más fallos.

6. *HTTP/2 and gRPC*:
   El protocolo `HTTP/2` tiene una gran mejora de rendimiento con respecto a su sucesor `HTTP/1.1` ya que permite multiplexar peticiones sobre una única conexión `TCP` y así elimina las múltiples conexiones entre cliente/servidor. `Envoy` ha sido desarrollado para soportar los 2 protocolos por lo que puede recibir tráfico con `HTTP/1.1` y redirigirlo con `HTTP/2` o viceversa. También puede procesar trafico gRPC.

2.1.2 Plano de control, Istiod

`Envoy` es el motor que impulsa `Istio` otorgándole soluciones a todo el trabajo pesado y así poder implementar las funcionalidades que Istio proporciona. Sin embargo, para sacarle el máximo rendimiento es necesario una infraestructura que permita la gestión y configuración de la flota de `envoys` que se instalarán como `sidecars` en todos los microservicios de la arquitectura. El plano de control de Istio esta dividido en tres componentes: *Pilot, Mixer, Citadel,* pero en las versiones mas recientes estos 3 componentes se han unificado en un componente llamado *Istiod*. Para un mejor entendimiento de las funcionalidades de Istiod se analizará cada componente por separado aunque sepamos que en las nuevas versiones ya no existe y están unificados bajo un único componente.

Pilot

Este componente es responsable de gestionar la flota de `envoys proxys` que se ejecutan como `side containers` en todos los microservicios del cluster asegurándose de que tienen actualizada la topología de red, políticas de red etc.

Mixer

Como su nombre indica junta información. Cada `envoy` envía periódicamente al *Mixer* métricas sobre el comportamiento del tráfico que pasa a través del `envoy`. `Mixer` usa un *modelo canónico de datos*. [1] Gracias al `mixer` se pueden utilizar distintas aplicaciones de monitorización que trabajen en conjunto con Istio como pueden ser `Prometheus, Kiali, Jaeger, Grafana` etc.

Citadel

Este componente es el encargado de gestionar la emisión,firma, renovación de `certificados X.509` que son usados entra la flota de `envoys` para proporcionar una comunicación segura entre los microservicios cifrando todo su tráfico sin que estos sean consientes por lo que no afectaría a la funcionalidad del microservicio.

[1] Modelo de datos en el que hay un punto en común donde distintas aplicaciones envían/recogen información, en este punto se traduce esa información a un lenguaje/protocolo común que entienda el resto de aplicaciones.

Chapter 3

Análisis de Istio

Para poder llevar acabo un análisis de las funcionalidades que otorga Istio se he creado una pagina web de ejemplo siguiendo una arquitectura basada en microservicios en un clúster de `kubernetes`. Sobre esta arquitectura haremos todas las pruebas necesarias para determinar el valor que aporta Istio.

3.1 Infraestructura

Se ha utilizado `GCP(Google Cloud Platform)` como proveedor para desplegar nuestra arquitectura. `GCP` es uno de los proveedores de nube publica mas usados en el mundo el cual dispone de una gran variedad de herramientas. Haremos uso de un crédito que ofrece a estudiantes para así analizar Istio en un entorno real.

La infraestructura se va componer de un clúster de `kubernetes` compuesto de tres instancias virtuales conectados entre si por una `VPC(Virtual Private Cloud)`. Este cluster de `kubernetes` será accesible desde internet por un recurso de `Google` denominado Balanceador de carga al cuál se le asignará una IP pública y todo el tráfico que le llegue lo redireccionará al cluster de `kubernetes`. Este cluster de `kubernetes` es gestionado por `Google` directamente. En la siguiente imagen 3.1 se aprecia la estructura general de la arquitectura. Dado que tenemos un crédito limitado y `GCP` cobra por minuto de los recursos usados en su plataforma se ha decidido crear la infraestructura usando `Terraform` la cual nos permitirá crear y destruir de forma sistemática arquitecturas `cloud` completas, en lugar de configurarlas a mano usando el interfaz web que proporciona típicamente cualquier proveedor `cloud`. Más adelante, en el apartado 3.1.1 se hace una descripción más detallada de este servicio.

Los microservicios que componen esta infraestructura son una aplicación web creada

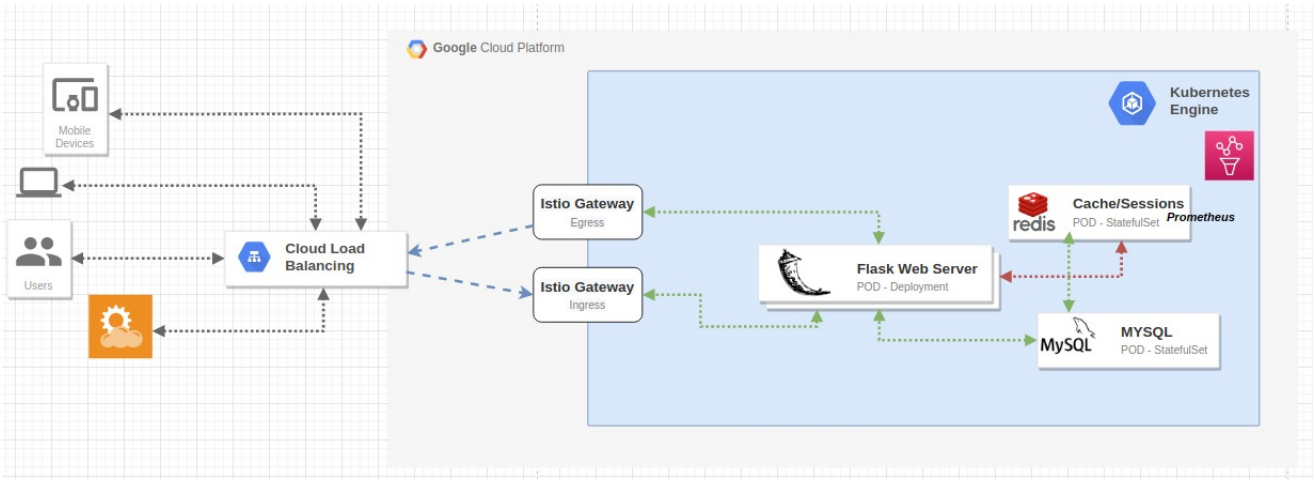


Figure 3.1: Infraestructura

usando el `framework` de `Flask` el cual se comunicara una base de datos y hará peticiones `API` a un servicio externo del cluster. También se ha usado un microservicio falso que puede interpretar tráfico HTTP para probar comunicaciones de servicios ascendentes y pruebas de redes de servicios y otros escenarios, este servicio falso es muy útil ya que mediante variables de entorno se puede configurar el comportamiento como el tiempo medio de respuesta, la variación de tiempo que tendrá la respuesta, que código http devolverá, un tanto por ciento de peticiones qué fallarán, que tipo de error devolverá etc. Para revisar en profundidad el funcionamiento de este servicio ver [14].

Se ha decidido usar el `framework` de `Flask` ya que de manera sencilla se puede definir `endpoints`, gestionar los datos de una petición y crear respuestas HTTP, por defecto con `Flask` tendremos un generador de plantillas llamado `jinja2` con el que podremos generar de forma dinámica archivos `html` y `css`. La simplicidad del `framework` junto con la del lenguaje `python3` permite generar entornos de prueba sencillos para así poder centrarnos en las funcionalidades que otorga Istio, sin entrar mucho en detalle la aplicación web estará formada principalmente de 3 `endpoints`[1]:

1. `/home,/`: Simplemente devolverá un html de la `home` de la web.
2. `/market`: Este `endpoint` se comunicara con 2 servicios, el primero sera la base de datos la cual estará en el mismo cluster de `kubernetes` por lo que la comunicación con esta sera interna, el segundo servicio hará una petición a api.openweathermap.org solicitando el tiempo de Madrid en este instante, esta comunicación no será privada ya que api.openweathermap.org es un servicio público.

[1]En una aplicación web se les define endpoints a la dirección URL la cual un servidor expone para que sus clientes hagan peticiones HTTP.

3. `/call-backend`: Este se comunicará con el servicio falso ya mencionado para poder testear comunicaciones ascendentes y observar el comportamiento del tráfico en servicios que estén fallando o experimentando un fallo en el rendimiento.

En el repositorio donde está todo el código de la aplicación web se encuentra en [4]

3.1.1 Terraform

`Terraform` es una herramienta que nos permite gestionar y automatizar la creación de una infraestructura en diferentes proveedores `cloud`. Actualmente tiene soporte sobre `AWS(Amazon Web Services)`, `GCP(Google Cloud Platform)`, `Azure` y muchos más. Esta herramienta es `open source` y usa un lenguaje declarativo, esto quiere decir que no hace falta declarar cada paso de la gestión/automatización de la infraestructura sino que se declara el estado final y `Terraform` se encargará de hacer todo lo necesario para obtener ese estado.

`Terraform` esta compuesto de dos componentes principales:

1. *Terraform core*: Este componente usa dos fuentes de entrada, archivos de configuración y el estado actual de la infraestructura. Los archivos de configuración son gestionados por el usuario. En estos archivos se define el estado final deseado de la infraestructura, una vez definido se compara con el estado actual de la infraestructura y `Terraform` se encargará de realizar las operaciones necesarias para conseguir llegar a ese estado definido en los archivos de configuración.Estas operaciones pueden ser la creación, destrucción y/o actualización de recursos del proveedor de nube.

2. *Proveedores*: Este componente definirá en que proveedor de nube donde son creados los recursos definidos en el *Terraform core*. Gracias a este componente podemos desplegar la arquitectura que estará dentro del cluster de `kubernetes`, es decir tendremos dos proveedores `GCP` y `Kubernetes`. Con el proveedor de `GCP` crearemos el cluster de `kubernetes` es decir la cantidad de nodos que tendrá, que tipo de maquinas usaran, cuanto discos de memoria tendrán a que red estarán conectados y con el proveedor de `Kubernetes` crearemos la arquitectura basada en microservicios.

Se ha decidido usar `Terraform` ya que para poder analizar Istio necesitamos de una infraestructura previa ya desplegada y con un simple comando podemos borrar y recrearla en cuestión de minutos así mismo reduciremos costes, ya que solo será cobrada durante su funcionamiento lo cual es importante ya que al desplegar la infraestructura sobre los servidores de Google estos seran cobrado por minuto de uso. Los archivos de configuración se encuentran en un repositorio de Github [19] por lo que habrá un controlador de versión

el cual es muy útil ya que tendremos un histórico de todas las modificaciones que se hagan en la infraestructura, a esto se le conoce como infraestructura como código ya que toda la infraestructura estará definida bajo código en lugar de hacerlo mediante proceso manuales y además tendremos un histórico de todos los cambios pudiendo revertilos siempre que queramos. En nuestro caso los distintos entornos de prueba se guardaran bajo etiquetas.

La infraestructura sobre la que vamos a trabajar va estar compuesta de los siguientes elementos de `Terraform`, se hará una descripción de los elementos y configuración mas significativos, la configuración completa se puede encontrar en el repositorio de `Github`:

1. `VPC`: Es una red virtual mundial que abarca todas las regiones disponibles por `GCP`, nuestro cluster de Kubernetes estará conectado a esta red virtual y así podrán los nodos comunicarse entre ellos. La definición se observa a continuación en la cual especificaremos que no cree subredes automáticamente ya que esto crearía un gran numero de subredes aumentado el coste de la infraestructura.

```
resource "google_compute_network" "vpc" {
  name                    = "${var.project_id}-vpc"
  auto_create_subnetworks = "false"
}
```

2. Subred: Dado que en la configuración de la VPC previa se ha desactivado la creación de subredes automáticas se ha definido una única subred la cual usaran los nodos del cluster de kubernetes, ya que se trata de un entorno de pruebas se la elegido el siguiente rango ”10.10.0.0/24” el cual no es mas que suficiente.

```
resource "google_compute_subnetwork" "subnet" {
  name          = "${var.project_id}-subnet"
  region        = "europe-west1-b"
  network       = google_compute_network.vpc.name
  ip_cidr_range = "10.10.0.0/24"
  private_ip_google_access   = true
}
```

3. Regla de cortafuegos: Se abrirán los puertos 80 y 8080 para que las peticiones que vengan del exterior pueden llegar al cluster de `Kubernetes`.

```
resource "google_compute_firewall" "allow_http" {
me    = "allow-http"
twork = google_compute_network.vpc.name

low {
protocol = "tcp"
ports    = ["80", "8080"]
```

4. Node pool: En esta configuración definiremos las propiedades de los nodos que compondrán el cluster de `Kubernetes`. En esta caso definimos que el cluster este compuesto de 2 a 3 nodos y las maquinas virtuales serán de tipo `n1-standard-1` por lo que dispondrán de 1 CPU virtual y 3.75 GB de memoria RAM.

   Para ahorrar costes estas maquinas serán de tipo `preemtible` por lo que los servidores de Google podrán eliminarla en cualquier momento para hacer frente a un incremento en la demanda de instancias virtuales, por este motivo estas instancias son un 60-80 por ciento mas baratas que una instancia estandar y dado que se trata de un entorno de pruebas y que es un cluster de `Kubernetes` el cual esta preparado para funcionar en caso de que un nodo sea eliminado no habrá ningún problema.

```
resource "google_container_node_pool" "np" {
me       = "${google_container_cluster.gke.name}-node-pool"
itial_node_count = 3
uster            = google_container_cluster.gke.name
cation           = var.zone

toscaling {
min_node_count = 2
max_node_count = 3
 node_config {
preemptible  = "true"
machine_type = "n1-standard-1"
disk_type    = "pd-ssd"
disk_size_gb = "50"

labels = {
  machine_type = "n1-standard-1"
  type         = "preemtible"
}
```

5. GKE: En esta configuración definimos el cluster `Kubernetes` haciendo referencia al `Node Pool` y `VPC` creados previamente.

```
resource "google_container_cluster" "gke" {
cation                  = var.zone
itial_node_count        = 1
move_default_node_pool  = true
fault_max_pods_per_node = 32
de_pool                 = google_container_node_pool.np
_allocation_policy {
cluster_ipv4_cidr_block  = "10.101.192.0/21"
services_ipv4_cidr_block = "10.101.200.0/25"

dons_config {
```

```
12 http_load_balancing {
13   disabled = false
14 }
15
16 twork    = google_compute_network.vpc.name
17 bnetwork = google_compute_subnetwork.subnet.name
```

3.2 Istio Ingress Gateway: Entrada de trafico

Los puntos de entrada de tráfico a una red son conocidos como *ingress points*. Estos puntos actúan como un controlador de acceso a la red interna aplicando políticas y normas que determinaran si el tráfico puede pasar y a que *backend*[2] es redirigido. En el caso en el que el punto de ingreso determine que el tráfico no es válido lo rechazará. Istio dispone de su propio punto de ingreso el cual está compuesto de un único `istio proxy`. Este *ingress point* puede ser creado manualmente o podemos usar el generado por defecto en la instalación de Istio[3].

Para configurar el punto de ingreso de Istio empezaremos definiendo dos recursos de Istio: el *gateway* y *VirtualService*. Los dos son fundamentales para la configuración del ingreso de tráfico al cluster de `kubernetes`.

3.2.1 Gateway

Para configurar un *ingress gateway* en Istio utilizamos el recurso *gateway* especificando que puerto van a escuchar para aceptar conexiones y que `hosts virtuales` están asociados a esos puertos:

```
1  apiVersion: networking.istio.io/v1alpha3
2  kind: Gateway
3  metadata:
4    name: flask-gateway
5  spec:
6    selector:
7      istio: ingressgateway
8    servers:
9    - port:
10       number: 80
11       name: http
12       protocol: HTTP
```

[2]Punto final de una red

[3]A la hora de instalar Istio se creará un namespace llamado istio-system donde habrían 3 pods, istiod, istion-ingressgateway y istio-egresswateway

```
13     hosts:
14       - "tfg.com"
```

Por defecto los *gateway* si no tienen configurado un *VirtualService* al cual enrutar el tráfico devuelve una respuesta HTTP con el código de respuesta 404. El pod que ejecuta el gateway que por defecto es *istio-ingressgateway*, necesitará poder escuchar a una IP:puerto que sea expuesto del cluster, dado que la infraestructura esta alojada en `GCP` automáticamente en la instalación de Istio al crear el *istio-ingressgateway y istio-egressgateway* se creará un balanceador de carga con una IP pública al cual serán asociados.

3.2.2 Virtual Services

Con este recurso podremos definir como enrutar el tráfico que llegue al `gateway`. Definirá cómo será la comunicación entre el cliente y los servicios de `kubernetes`.

```
1  apiVersion: networking.istio.io/v1alpha3
2  kind: VirtualService
3  metadata:
4    name: vs-flask-app
5  spec:
6    hosts:
7    - "tfg.com"
8    gateways:
9    - flask-gateway
10   http:
11   - route:
12     - destination:
13         host: webappv2
14         port:
15           number: 80
```

Aquí definimos un *virtualservice* el cuál se asocia al gateway `flask-gateway` por lo que todo el tráfico que le llegue al gateway con la cabecera del `Host` igual a tfg.com será redirigido al puerto 80 del servicio webappv2.

3.3 Métodos de despliegues

En `kubernetes` la procedimiento general de desplegar nuevo código en el entorno de producción es seguir una estrategia *blue-green deployment*. Los pasos de este procedimiento es levantar en el cluster de `kubernetes` una versión nueva de un microservicio, cuando esta ya esté lista para recibir tráfico se redirigirá todo el tráfico a la nueva versión.

La versión antigua se le denomina entorno azul y al nueva versión entorno verde. Una vez el tráfico de datos ha sido completamente redirigido al entorno verde se procederá a

eliminar el entorno azul el cuál se queda activo para poder hacer un *rollback* en caso de que la nueva versión no tenga el comportamiento esperado. Con este modelo se despliega nuevas funciones sin que el cliente se vea afectado, es decir, no tiene tiempo de caída, solo en el caso de que todo vaya bien ya que el problema principal de esta estrategia es que el nuevo entorno pasa a procesar todo el trafico de un instante a otro.

3.3.1 Canary Release

Para reducir el riesgo de introducir nuevo código en un entorno de producción es importante tener claro la diferencia entre despliegue y lanzamiento. Cuando se realiza un despliegue en producción se instalá el nuevo código en el servidor, contenedor, cluster, etc. Pero no redirigimos el tráfico hacia él, este proceso no afecta a los usuarios ya que no procesa ninguna petición del cliente. En este punto podemos poner a prueba el nuevo código desplegado, sometiéndolo a un número de pruebas que verifiquen su correcto funcionamiento y una vez comprobado se tomará la decisión de cómo **lanzarlo** a los usuarios. El lanzamiento de código significa redirigir tráfico real a la nueva versión, en vez de seguir una estrategia blue/green deployment con Istio podemos seguir la estrategia *canary release*. Con esta técnica iremos exponiendo a los usuarios gradualmente la nueva versión. Podemos redirigir solo el tráfico de peticiones que lleguen con una cabecera HTTP especial y así solo los trabajadores internos pueden probar el nuevo código. Con esta técnica podemos comprobar que la nueva versión funciona correctamente de una manera más segura y teniendo siempre la opción de redirigir el tráfico a la versión previa.

Para aplicar esta estrategia sobre nuestra infraestructura de prueba basta con configurar tres componentes de Istio: *gateway, virtualservice y destionationrule*. Se configurará que las peticiones que vayan a la aplicación se redirigen a la versión correspondiente según una cabecera HTTP. Como ejemplo se ha creado dos versiones y las peticiones que vayan con la cabecera type:light ira a la versión v2 y el resto de las peticiones irán a la versión v1. Para poder llevar acabo esta prueba se ha usado `Postman` y así poder ejecutar un número alto de peticiones rápidamente. Desde el panel 3.2 de `Kiali` podemos observar como se divide el tráfico. En la prueba hemos hecho 75 peticiones a la versión 1 y 25 a la versión 2. Una vez hecha las pruebas necesarias se determinará si redirigir todo el tráfico a la nueva versión, solo habría que modificar el recurso `Virtualservice` y posteriormente eliminar el despliegue de la versión antigua. La configuración de Istio se puede observar en el repositorio de github [19].

La técnica *canary release* se puede aplicar de otras formas aportando distintos puntos de vista sobre el nuevo código, además del funcionamiento se puede analizar el impacto que esta teniendo el código sobre los clientes. ¿Se vende más productos?¿Permanecen más tiempo conectados?¿En qué versión permanecen más tiempo? etc. Este análisis es posi-

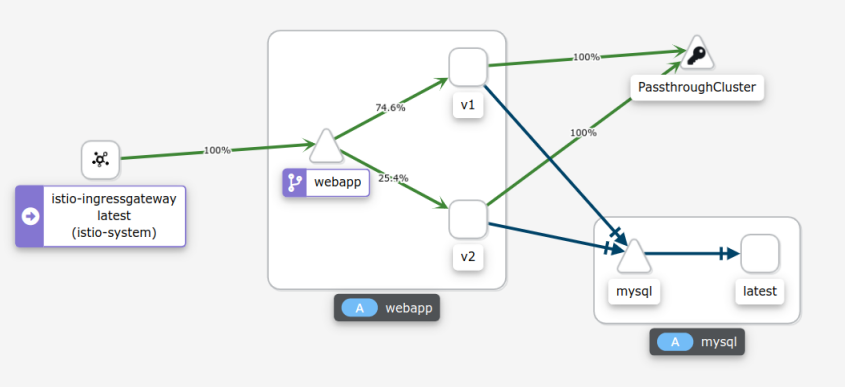


Figure 3.2: División de trafico según cabecera HTTP.

ble distribuyendo el tráfico real siguiendo una política basada en pesos, es decir, exponer un número pequeño de usuarios reales a la nueva versión y analizar su comportamiento. Modificando el `VirtualService` podemos llevar esto acabo fácilmente. Al igual que con el anterior método siempre tenemos la posibilidad de redirigir el tráfico a la versión antigua. Hay que tener en cuenta que el peso que se le asigna a cada versión puede variar entre 1-100 incluso se puede dividir en 3 o más versiones pero es necesario que la suma de todos los pesos sea 100 si no la distribución del tráfico se vuelve impredecible. Si aplicamos la configuración de Istio que se encuentra en el tag canary-release-v2 podremos recrear un escenario donde el 10% del tráfico entrante se redirige a la v2 el restante va a la v1. En la figura 3.3 observamos desde el panel de `kiali` ese comportamiento.

Lo ideal es ir incrementando la cantidad de tráfico a la nueva versión y estudiar su comportamiento. Este incremento de tráfico se puede ir haciendo manualmente e ir aplicando el cambio sobre el `Virtualservice` pero por lo general se automatizara usando alguna herramienta de `CI/CD`.

Automatización de canary-release

Hemos observado que Istio proporciona potentes funciones para controlar el encaminamiento de tráfico pero hay maneras más eficientes de activarlos que desplegando esas configuraciones a mano, es decir, en el caso de un lanzamiento de una nueva versión siguiendo una política `canary-release` por peso lo óptimo no es modificar cada X minutos el archivo de configuración del `VirtualService` para incrementar el peso del nuevo servicio ya que se tendría que crear varias versiones de la configuración, lo que crea más trabajo y oportunidades para una configuración incorrecta. Este proceso lo podemos automatizar

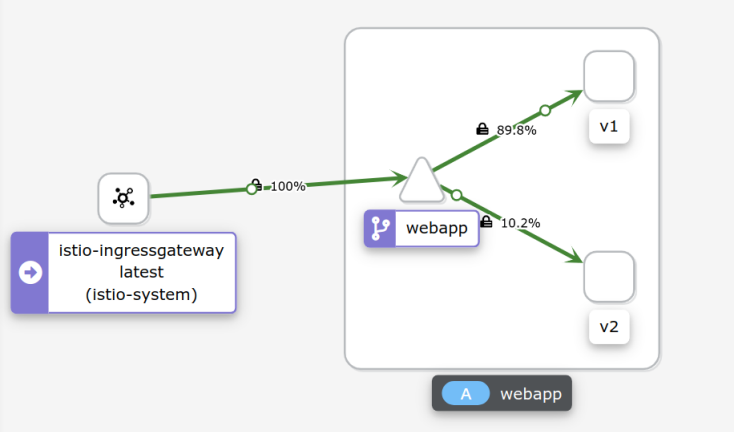


Figure 3.3: División de trafico siguiendo una política basada en pesos.

utilizando *Flagger*.[4]la cual permite configurar como realizar el lanzamiento, cuando exponer la nueva versión a más usuarios y cuando volver a la versión antigua si fuera falta de forma automática.

`Flagger` se basa en métricas proporcionadas por Istio y `Prometheus` para determinar el estado de un servicio. La configuración e instalación de `Prometheus` viene ya implementada con la instalación de Istio. `Flagger` usa un deployment de `kubernetes` y crea una serie de objetos(`servicio cluster IP, destination rules y virtualservices de Istio, deployments de kubernetes`) para exponer la aplicación dentro del cluster y dirigir el lanzamiento `canary-release`. La configuración de `Flagger` es similar a los objetos nativos de `kubernetes` y Istio. Para entenderlo mejor se analizara un ejemplo de configuración básica de `Flagger`.

Además de haber instalado `Prometheus` y Istio, `Flagger` necesitará que haya un deployment y un `Gateway` creado.

```
1  apiVersion: flagger.app/v1beta1
2  kind: Canary
3  metadata:
4    name: webapp-release
5    namespace: istioinaction
6  spec:
7    targetRef:
8      apiVersion: apps/v1
9      kind: Deployment
10     name: webapp
11   progressDeadlineSeconds: 60
```

[4]Flagger es una herramienta creado por Stefan Prodan para automatizar los despliegues canary-release.

```
12  # Service / VirtualService Config
13  service:
14    name: webapp
15    port: 80
16    targetPort: 8080
17    gateways:
18    - flask-gateway
19    hosts:
20    - "tfg.com"
21  analysis:
22    interval: 45s
23    threshold: 5
24    maxWeight: 50
25    stepWeight: 10
26    match:
27    metrics:
28    - name: request-success-rate
29      thresholdRange:
30        min: 99
31      interval: 1m
32    - name: request-duration
33      thresholdRange:
34        max: 500
35      interval: 30s
```

En las líneas 7-10 se ha definido sobre que `deployment` se realizará el lanzamiento siguiendo la política `canary-release`. Este proceso se activará cuando la configuración del `deployment`, en este caso el denominado `webapp` cambie, por ejemplo, al actualizar la imagen, variables de entorno, puertos expuestos, volúmenes montados, secretos mapeados etc. En las líneas 13-20 se define el servicio que `Flagger` creará para poder exponer el servicio a los demás objetos que haya en el cluster y que gateway será usado. En las líneas 21-35 se define como se llevara acabo el `canary-release`, en este caso en intervalos de 45 segundos se irá aumentando el tráfico a la nueva versión un 10% si las métricas definidas se cumplen, una vez llegue al 50% todo el tráfico irá a la nueva versión. La configuración de este ejemplo se encuentra en el tag canary-release-v3.

3.3.2 Clonación de tráfico

Con las técnicas previas reducimos el riesgo en el lanzamiento de nuevas versiones. Las dos técnicas utilizan tráfico real y pueden afectar a los usuarios aunque controlamos que el número de usuarios afectados sean mínimo dando un mejor servicio. Otro enfoque es clonar el tráfico entrante a un nuevo despliegue y estudiar como actúa, con esta técnica podemos analizar como reacciona una nueva versión con tráfico real sin que el usuario se vea afectado ya que Istio se encarga de que las respuestas no lleguen al usuario. Para poder

clonar el tráfico modificaremos el `Virtualservice` donde indicaremos a que servicio se enviará el tráfico clonado. Se puede configurar en porcentajes cuanto tráfico es clonado lo cual permite reducir gastos ya que si se clona el 100% del tráfico sería necesarios duplicar los recursos de la infraestructura. En el siguiente ejemplo se define que el 50% del tráfico que se dirigía a la versión uno del servicio `webapp` sea clonado y enviado a su versión dos. La configuración de Istio se encuentra en el tag shadowing-traffic.

```
kind: VirtualService
metadata:
  name: vs-flask-app
spec:
  hosts:
  - "tfg.com"
  gateways:
  - flask-gateway
  http:
  - route:
    - destination:
        host: webapp
        subset: version-v1
      weight: 100
    mirror:
      host: webapp
      subset: version-v2
    mirrorPercentage:
      value: 50.0
```

Haciendo uso de `Postman` se han hecho 100 peticiones las cuales han devuelto todas un status 200 pero si nos fijamos en el panel del `Kiali` 3.4 vemos como la versión 2 esta devolviendo un número elevado de 400, gracia a esta clonación de tráfico podemos identificar fallos en nuevas versiones sin que el cliente se vea afectado. Este comportamiento se ha logrado modificando la v2 de la aplicación web frontal para que devolviera aproximadamente el 30% de las peticiones un código de respuesta HTTP 400.

3.3.3 Enrutamiento de tráfico a servicios externos al cluster

Por defecto Istio permite la salida de todo el trafico del cluster, es decir si los servicios que componen la arquitectura necesitan comunicarse con servicios externos al cluster en donde estén Istio no los limitará. Teniendo en cuenta que todo el tráfico que entra/sale de un servicio dentro del cluster tiene que previamente pasará por el `sidecar proxy` se puede modificar este comportamiento y denegar el tráfico saliente del cluster.

Bloquear el tráfico saliente es una prádctica para evitar vulnerabilidades en los cuales

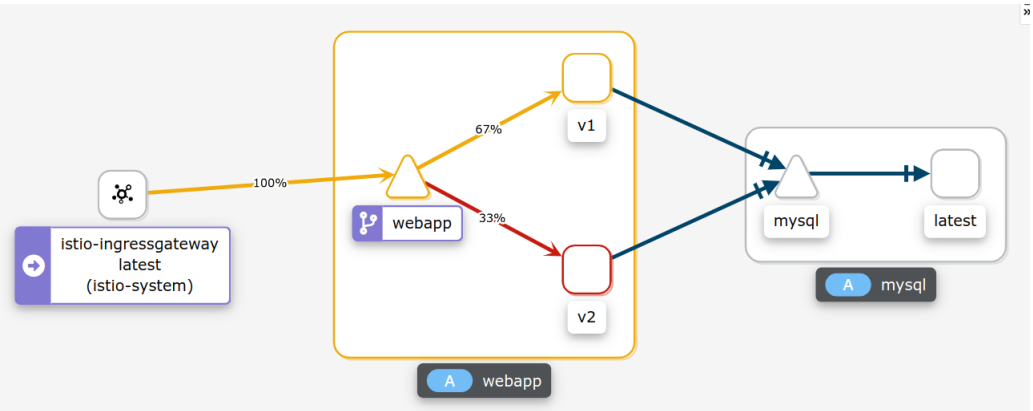


Figure 3.4: Clonación de trafico

el atacante una vez tomado el control de un servicio inyecta código para manipular el funcionamiento de este y filtrar datos privados como contraseñas, usuarios etc a servicios externos. No todos los servicios estarán dentro del cluster de `kubernetes`, podrían haber servicios HTTP externos, base de datos, etc. Istio puede gestionar con que servicios externos se podrá comunicar construyendo un registro de todos los servicios que viven dentro del cluster utilizando el API de `Kubernetes` basándose en los objetos "Service" de Kubernetes, para agregar a este registro los servicios que residen fuera del cluster se usara el objecto `"ServiceEntry"` el cuál encapsulará información del servicio externo. Un ejemplo básico de este objeto es:

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: jsonplaceholder
spec:
  hosts:
  - api.openweathermap.org
  ports:
  - number: 80
    name: http
    protocol: HTTP
  resolution: DNS
  location: MESH_EXTERNAL
```

En nuestra aplicación web utilizamos las `APIs` proporcionadas por openweathermap para recoger datos climatológicos de una región de España por lo que una vez activado el bloqueo de las comunicaciones con servicios externos al cluster haremos uso del `"ServiceEntry"` para agregar este servicio al registro de Istio y así se puedan hacer peticiones a api.openweathermap.org. La configuración de este ejemplo se encuentra en el tag serviceEntry-v1.

3.4 Resiliencia: Retos de la Red

Una vez el tráfico ha entrado en el cluster a través del `ingress gateway` de Istio se puede manipular a nivel de petición y controlar a que versiones redirigirlo dependiendo de distintos factores como la versión, cabeceras `HTTP`, métricas del rendimiento de los servicios, etc. El problema con las arquitectura basadas en microservicios es que los canales de comunicación entre los servicios no son siempre fiables y no se puede garantizar un `SLA`. Inicialmente para solventar este problema se puede introducir eventos dentro del código de los microservicios es decir delegar la solución al funcionamiento lógico de la aplicación, una mejor solución es dejar que Istio se encargue mediante recursos como `timeouts, retries and circuit breaking` y así no alterar la lógica de funcionamiento de los microservicios.

3.4.1 Resiliencia de red en la lógica de los microservicios

Antes de que herramientas como Istio que gestionan la comunicación entre servicios estuviera disponible los desarrolladores se veían en la obligación de implementar muchos de estos patrones básicos de resiliencia de red en el código de la aplicación. Algunos `frameworks` de código abierto fueron creados para solventar estos problemas como por ejemplo *Finagle* en 2011, el cuál es una librearía de `Java` que permite implementar varios patrones `RPC` como `timeouts, retries and circuit breaking`. Poco después Netflix hizo público sus `framework` *Netflix Hystrix & Netflix Ribon*. Estos dos `frameworks` son muy populares en la comunidad de `Java` pero introduce una gran carga de trabajo a la hora de desarrollar el código de la aplicación.

El problema principal de estas librerías es que a través de diferentes lenguajes, `frameworks` e infraestructura, tendrá diferentes implementaciones por lo que aunque Finalge y Netflix Hystrix funcionen correctamente en aplicaciones Java carece de compatibilidad con otras aplicaciones que estén desarrolladas con `Python, Go, NodeJs`, etc. En algunos casos estas librerías son bastante invasivas al código de la aplicación por lo que habrá código de red esparcido oscureciendo la lógica real del microservicio. Por último, el mantenimientos de estas librerías genera una gran carga de trabajo.

3.4.2 Tipos de balanceo de carga

Usando Istio se puede implementar balanceo a nivel del cliente esto quiere decir que si un microservicio A quiere comunicarse con un microservicio B tendrá la información de todos los `endpoints` disponibles del microservicio B y se encargara de elegir que algoritmo de balanceo a usar para una mejor distribución. Esto permite reducir la necesidad de usar

balanceadores de carga centralizados que puedan crear cuellos de botella, puntos de fallo y otorga al cliente la posibilidad de comunicarse con el servicio B directamente. Istio utiliza `"service and enpoint discovery"` para informar al `"client-side proxy"` con los datos de los otros microservicios en la infraestructura y es aquí donde se configurará que tipo de balanceo se realizará a los distintos microservicios disponibles.

Se podrá configurar el algoritmo de balanceo a usar utilizando los *destinationRule*, actualmente se disponen de 3 algoritmos:

1. *Round robin (Por defecto)*:
   Este algoritmo trata a todos los destinos como iguales por lo que distribuye la carga equitativamente entre ellos sin importar si estos responden más/menos rápido. Es un algoritmo bastante sencillo de utilizar y requiere de muy pocos recursos para ser implementado.

2. *Aleatorio*:
   Como su nombre indica este algoritmo distribuye la carga de forma aleatoria entre todos los posibles destinos pasado el suficiente tiempo alcanzara los mismos resultados que algoritmo `Round Robin` ya que por probabilidad se distribuirá la carga de forma equitativa entre los microservicios. Este algoritmo por lo general otorga mejor rendimiento que `Round robin` si no hay una política de `healthchecks` para verificar que microservicios están funcionando correctamente.

3. *Weighted least request*:
   Este algoritmo escogerá de manera aleatoria 2 microservicios a los que pueda distribuir el tráfico y elegirá el que tenga menos peticiones activas, este enfoque aunque parezca sencillo permite resultados casi tan buenos como un escaneo entero además su implementación y uso de recursos es mucho mas eficiente. Se le conoce como el poder de las 2 opciones ya que permite de forma eficiente drenar de carga a los microservicios, para información detallada de como funciona este balanceo revisar el poder de las 2 opciones [16].

Para probar el rendimiento de los distintos algoritmos de balanceo se ha preparado un entorno el cual esta compuesto de 3 microservicios: `webapp`, `simple-backend`, `simple-backend-slow`.

El microservicio `webapp` es un `deployment` compuesto de 2 `pods` el cual uno de ellos se comunican con el `simple-backend` y el otro con `simple-backend-slow`, como el nombre indica el `simple-backend-slow` es mas lento en contestar por lo que uno de los `pods` del `deployment` del `webapp` va a tener tiempos de respuestas mayores. En la imagen 3.5 se puede observar la estructura.

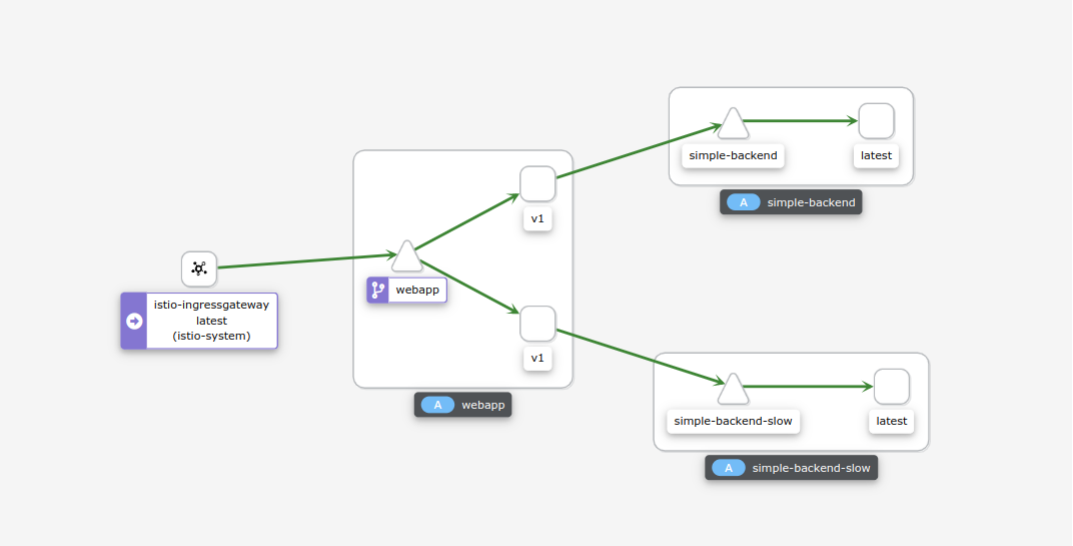


Figure 3.5: Arquitectura para estudiar las políticas de balanceo de carga

Tipo de Balanceo	Tiempos de respuesta(ms)			
Etiqueta	Media	Minimo	Maximo	95th Percentil
LEAST_CONN	354.65	211	1695	599
RANDOM	417.92	210	1793	697
ROUND ROBIN	405	210	1696	700

Table 3.1: Resumen de los resultados obtenidos sobre los distintos algoritmos de balanceo de carga

Para poner a prueba el escenario se ha utilizado una herramienta de código abierto llamada `Jmeter`. Con esta herramienta podemos generar carga y así comprobar el funcionamiento y rendimiento de los distintos tipos de balanceos proporcionados por Isito.

La simulación constaba de 5000 usuarios los cuales realizaban una única petición, la peticiones se distribuían en un tiempo de 5 minutos por lo que `Jmeter` crea un hilo de ejecución simulando la petición de un usuario cada 0,06 segundos por lo que simularía unas 17 peticiones por segundo . Tras la simulación se aprecia que el método de balanceo con mejores resultados es el del tipo `"LEAST_CONN"` como se observa en la tabla 3.1 este tipo de balanceo ha obtenidos la major media en tiempo de respuesta y el mejor 95th percentil.

Dado que la comunicación en una arquitectura basada en microservicios se realiza mediante la red siempre puede haber fallos. Estos errores pueden ser ocasionados por una gran variedad de factores como servidores, errores de red,fallos en el balanceo de la carga, fallos en la lógica de la aplicación, fallos en el sistema de operaciones incluso utilizando

proveedores como `GCP(Google Cloud Platform)` el cual proporciona unos servicios de gran calidad estos no se eximen de fallas y es por eso que en la mayoría de servicios aunque proporcionan un elevado `SLA`[5] la mayoría no son del 100%. Ya que es un hecho que habrán fallos es importante construir una arquitectura que este preparada para manejar estos errores. Para construir sistemas resilientes, se emplean herramientas esenciales como los *tiempos de espera* y *reintentos*.

3.4.3 Tiempos de espera

Hay una gran cantidad de errores que son evidentes cuando las peticiones se demoran mas de lo esperado y normalmente no terminan en completarse. Cuando un microservicio espera un largo periodo de tiempo para obtener una respuesta este también retiene recursos por lo que si hay un gran numero de peticiones que tardan más de lo esperado este microservicio puede quedarse sin recursos como la memoria, hilos disponibles en los procesos de ejecución, puertos efímeros etc. Esto puede desencadenar errores en cascada entre los microservicios ya que por el mal funcionamiento de uno puede dejar deshabilitado microservicios que están simplemente esperando una respuesta. Una buena configuración sobre los tiempos de espera a nivel de conexión y/o petición son muy útiles para evitar este problema.

No siempre se puede usar tiempos de espera, esto dependerá de la función lógica de cada microservicio por ejemplo en un `ecommerce` seria inviable poner tiempos de espera elevados al microservicio que procese un pago ya que aunque este demore es critico que se complete, pero si hay microservicios que agregan funcionalidades como un listado de las opiniones de un numero de clientes sobre un producto seria recomendable que si esta servicio tardara en responder el usuario pudiera seguir interactuando con la web para no verse bloqueado a no realizar compras.

Los tiempos de espera son un mecanismo sencillo de resiliencia de red y es muy importante saber utilizarlo de forma inteligente para dar un mejor servicio, obtener mejores resultados y disminuir la carga sobre los microservicios. El objetivo principal es aislar los casos en los que un microservicio tarda más en responder de lo esperado pero hay tener en cuenta que si este comportamiento persiste, es necesario investigar la causa del error, dado que aunque con tiempos de espera máximo prevenimos que no se saturen los otros microservicios de nada nos sirve que un numero elevado de peticiones estén respondiendo con un `timeout`.

[5]SLA(Service Level Agreement) es un compromiso entre un proveedor de servicios y un cliente. Los aspectos particulares del servicio (calidad, disponibilidad, responsabilidades) se acuerdan entre el proveedor del servicio y el usuario del servicio.

La configuración de tiempos de espera se realiza mediante *VirtualService* el mismo recurso con el que se configuraba el comportamiento de las peticiones según sus cabeceras `HTTP`. En el siguiente ejemplo se observa como se configura el *VirtualService* para que las peticiones dirigidas al `deployment simple-backend` tengan un tiempo máximo de espera de 0,4 segundos.

```
1  apiVersion: networking.istio.io/v1alpha3
2  kind: VirtualService
3  metadata:
4    name: simple-backend-vs
5    label: virtual-service
6  spec:
7    hosts:
8    - simple-backend
9    http:
10   - route:
11     - destination:
12         host: simple-backend
13     timeout: 0.4s
```

3.4.4 Reintentos

Los reintentos son una herramienta de doble filo, cuando un cliente o microservicio reintenta una petición se invierte más tiempo de ejecución para obtener la respuesta deseada. Cuando los errores por los que se ha realizado un reintento son esporádicos no generan una sobrecarga sobre el microservicio y de forma sencilla se consigue dar mejor servicio. Por otro lado, cuando estos errores son producidos por una gran carga de tráfico, este mecanismo puede ser contraproducente, ya que los reintentos aumentarán aun más la carga del microservicio llegando incluso a retrasar su posible recuperación.

Dado que es muy difícil predecir picos de tráficos que puedan afectar a un microservicio se puede configurar el sistema de reintentos para que no se conviertan en un problema. Una de las soluciones proporcionadas por Istio es configurando un tiempo de espera por reintento con el parámetro *perTryTimeout* así evitando saturar el sistema en momentos de alta carga. A la hora de configurar este parámetro es importante tener en cuenta el tiempo de espera configurado en el `virtual service` ya que si este es menor que *perTryTimeout**numerodeintentos no se producirán todos los reintentos configurados, otra solución la cuál es la más potente es configurar bajo que tipo de errores hacer un reintento esto se consigue mediante el parámetro *retry-on*, con este parámetro podemos indicar bajo que respuesta `HTTP` reintentar la petición cual brinda muchas opciones, ya que por ejemplo si el código de respuesta en una petición `HTTP` es `400(Bad Request)` no tiene sentido reintentarlo ya que el fallo viene en la petición, por lo que el microservicio le devolverá un 400. También se pueden configurar los reintentos segundo el código de respuesta del protocolo `gRPC`. Por último, Istio proporciona la opción de realizar el intento a un microservicio de otra zona con el parámetro *retryRemoteLocalities*. A continuación, se muestra una configuración básica de reintentos con Istio:

```
1  apiVersion: networking.istio.io/v1alpha3
2  kind: VirtualService
3  metadata:
4    name: simple-backend-vs
5  spec:
6    hosts:
7    - simple-backend
8    http:
9    - route:
10     - destination:
11         host: simple-backend
12     retries:
13       attempts: 3
14       retryOn: "connect-failure,refused-stream,5xx"
15       perTryTimeout: 200ms
16       retryRemoteLocalities: true
```

Peticiones	Ejecuciones			Tiempos de respuesta (ms)			
Etiqueta	# Muestras	Fallidas	% de error	Media	Minimo	Maximo	Median
Sin reintentos	500	83	16.60%	320.50	63.00	626.00	229.50
3 reintentos	500	0	0.00%	214.55	161.00	405.00	179.00

Table 3.2: Resumen de los resultados obtenidos tras aplicar reintentos en la configuracion de Istio

Hay que tener en cuenta que el objetivo principal de Istio es gestionar las comunicaciones entre microservicios y que cada microservicio solo se centre en su lógica de funcionamiento. Si en la lógica del funcionamiento incluimos reintentos estos se multiplicaran con los reintentos que tengamos configurados en Istio, es decir si en la lógica de un microservicio esta configurado para que realice 3 intentos y en la configuración de Istio tenemos otros 3 intentos cuando el sistema falle se realizaran un total de 9 intentos pudiendo sobrecargar el sistema.

Para estudiar a fondo el funcionamiento de los reintentos se ha utilizado el mismo entorno de pruebas que para estudiar los tipos de balanceo 3.5 con la diferencia de que el microservicio `simple-backend-slow` en vez de tardar más tiempo en responder fallará el 25% de las veces devolviendo un código de error 502. Usando la herramienta `Jmeter` se ha simulado un total de 500 peticiones a lo largo de un minuto. Los resultados obtenidos se pueden visualizar en la tabla 3.2

Podemos observar que efectivamente sin la configuración de reintentos hay un total de 83 peticiones fallidas pero al activar el reintento de errores [500, 599] no ha habido ningún fallo. Con el panel de 3.6 `Kiali` se aprecia como se ha distribuido el tráfico y vemos que aunque el balanceo que hay configurado en el `virtual service simple-backend` es de tipo `roundrobin` el tráfico no es equitativo entre los `backends` ya que el `backend v1-fast` ha recibido el 66.6% del tráfico en cambio `v2-slow` solo el 33.4% esto se debe a que como efectivamente el método es `roundrobin` todas las peticiones reintentadas han ido al `backend` que funcionaba correctamente. Por lo que el `backend "v1-fast"` ha procesado un total de 250+83 peticiones de 500 el cual es el 66,6% de las peticiones. Aunque no haya habido errores percibidos por el usuario el panel de `Kiali` te indica que el `backend v2-slow` esta fallando.

Los tiempos de respuesta y reintentos son mecanismos simples y potentes para aumentar la disponibilidad de una arquitectura basada en microservicios pero un mal uso de estos mecanismos pueden generar comportamiento no deseados como sobrecargar los microservicios y generar errores en cascada.

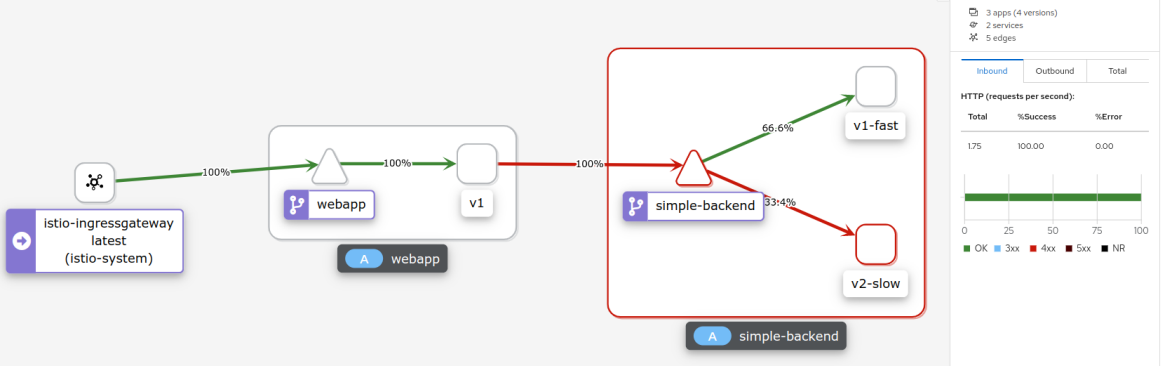


Figure 3.6: Balanceo de carga tras reintentos

3.4.5 Circuit breaker

`Circuit breaking` es una importante característica que aumenta la resiliencia de una infraestructura. El principal objetivo es prevenir fallos adicionales mediante el control y gestión de acceso a servicios que estén experimentando fallos, por ejemplo si un servicio empieza a fallar envés de seguir enviándole trafico incluso realizando reintentos podríamos detectar esta falla y reducir o incluso bloquear el trafico hacia este servicio.

Para habilitar un `circuit breaker` en Istio no utilizaremos un recurso especifico para ello si no que haremos uso de las configuraciones disponibles por el *destination rule*. Mediante estos parámetros de configuración se mantendrá un estado de los servicios y así se podrá detectar cuando falla, por ejemplo si tenemos un `deployment` de `kubernetes` con 5 réplicas si una de estas empieza a experimentar fallos el `circuit breaker` lo detectará y dejará o reducirá el tráfico que le llega según lo tengamos configurado y así se le dará tiempo al servicio para recuperarse.

Hay dos bloques de configuración que hacen posibles a un `circuit breaker`, el primero se llama *connectionPool* y se usa para gestionar cuantas conexiones y peticiones esta recibiendo un microservicio en tiempo real, con este bloque de configuración sabremos si el servicio se ha ralentizado ya que si el numero de conexiones que un microservicio tiene va en aumento significa que esta tardando mas tiempo de lo habitual en responder. Obviamente esto también puede ser causado por un pico de tráfico por lo que es importante tener un sistema de auto escalado eficiente. El segundo bloque se llama *outlierDetection* y se usa para saber el estado de los microservicios a diferencia del anterior bloque este no se centra en las conexiones que haya sino que resultados devuelve este servicio, es decir si durante un periodo de tiempo el microservicio empieza a incrementar el numero de errores el `circuit breaker` se activará.

Por cada bloque hay distintos parámetros de configuración para pode ajustar el `circuit breaker` a las necesidades del momento.

ConnectionPool

Los parámetros mas significativos son:

- **maxConnections**: Define el limite de conexiones que un microservicio deberia de tener, una vez este limite haya sido alcanzado el `circuit breaker` se activará.
- **http1MaxPendingRequests**: El máximo numero de peticiones `HTTP` en espera..
- **http2MaxRequests**: Limite de peticiones `HTTP` paralelas entre todos los microservicios de un grupo.

Hay más parámetros de configuración disponibles con los que se puede configurar, el máximo número de peticiones por conexión, máximo reintentos disponibles etc.

OutlierDetection

Los parámetros más significativos son:

- **consecutive5xxErrors**: Limita el numero de errores tipo 5xx que un microservicio puede devolver consecutivamente antes de ser excluido del grupo de servicios saludables, cuando el servicio experimenta errores de conexión también son catalogados como errores 5xx.
- **interval**: El intervalo de tiempo en el que se hará el analisis.
- **baseEjectionTime**: El tiempo en el que el servicio sera excluido del grupo de microservicios saludables una vez el `circuit breaker` haya sido activado.

Para estudiar a fondo el funcionamiento de los `circuit breakers` se ha utilizado el mismo entorno de pruebas que para estudiar los tipos de balanceo 3.5 con la diferencia de que el microservicio `simple-backend-slow` en vez de tardar maá tiempo en responder fallará el 100% de las veces devolviendo un código de error 500. Usando la herramienta `Jmeter` se ha simulado un total de 3000 peticiones a lo largo de 5 minutos y se ha ajustado el `circuit breaker` para que cada 12 segundos si encuentra más de 5 errores consecutivos se active y expulse del grupo de microservicios disponibles por 1 minuto. Los resultados obtenidos se pueden apreciar en la siguiente tabla, como se puede observar utilizando el `circuit breaker` se ha reducido significativamente el numero de errores ya que la mitad de las peticiones hubieran sido fallidas si este mecanismo no hubiera estado configurado por lo que se ha pasado de un 50 por ciento de tasa de error a un 2.5 por ciento.

Peticiones	Ejecuciones			Tiempos de respuesta(ms)		
Etiqueta	# de peticiones	Fallidas	% de error	Media	Minimo	Maximo
CIRCUIT BREAKER	3000	75	2.50	209.15	66	1310

Table 3.3: Resumen de los resultados obtenidos tras activar el `circuit breaker` en la configuración de Istio.

3.5 Observabilidad

No hay que confundir la monitorización con la observabilidad de un sistema, el objetivo de la monitorización es recolectar métricas como pueden ser, capacidad del disco, memoria utilizada, uso de CPU, latencia de la red etc y comparar estos datos con valores que consideremos correctos para el funcionamiento de la aplicación. En el caso de que este limite sea superado se tomarán las acciones oportunas para resolver el incidente y estabilizar el estado de la aplicación, por ejemplo, en una arquitectura basada en microservicios se puede monitorizar el uso de `CPU` de un `deployment` en específico, si este supera el límite previamente configurado se procedería a aumentar los recursos de `CPU` asignados. En cambio con la observabilidad suponemos que nuestros arquitectura es impredecible y no se puede saber todos los posibles puntos de fallos por lo cual no podríamos comparar una métrica continuamente para saber si el sistema esta fallando o fallará en un futuro cercano, por lo que se recopilará gran variedad de datos no solo métricas de los sistemas que componen la arquitectura, sino datos de por ejemplo las peticiones que reciben estos microservicios, cuantas peticiones por segundo llegan, la IP de quién ha realizado la petición, cuánto tiempo ha tardado en ejecutarse etc. Una vez recolectados se entenderá mejor el funcionamiento de la arquitectura al completo, esto es importante en las arquitecturas basadas en microservicios ya que la aplicación estará dividida en cientos de microservicios. Entendiendo en profundidad como los microservicios funcionan entre si sera mas fácil detectar y `debugear` fallos.

3.5.1 Observabilidad con Istio

Usando Istio podemos obtener observabilidad sobre un arquitectura basada en microservicios de manera muy eficiente ya que por cada servicios habrá un Istio proxy que procesara todas las peticiones que le lleguen generando datos, estos datos permitirán estudiar el comportamiento de los servicios liberando a los desarrolladores de esta tarea. A través de Istio, los operadores obtienen una comprensión profunda de cómo interactúan los servicios monitoreados, tanto con otros servicios como con los propios componentes de Istio.

Istio genera los siguientes tipos de datos:

- **Métricas**: Basándose en el tráfico, latencia, saturación y errores. Istio genera métricas para poder ser usados por distintos sistemas de visualización de datos como `Prometheus, Grafana, kiali` etc. Además de métricas relacionadas con los servicios que componen la arquitectura Istio también genera métricas de lo sistemas que los componen. Estas métricas se dividen en dos grupos:

  - **Métricas a nivel de Proxy**: Cada `proxy` recopila métricas de todo el tráfico que pase por el además de datos de las funciones administrativas que desempeña incluida su configuración y su estatus. Por defecto Istio solo recopila un subconjunto de las métricas para evitar sobrecargar los `backends` y reducir el uso de `CPU` sin embargo se puede escoger fácilmente que métricas recopilar. Se puede ver un listado de las posibles métricas a recopilar en el apéndice A.1

  - **Métricas a nivel de Servicio**: Istio proporciona métricas para la monitorizar la comunicación entre los microservicios. Se puede ver un listado de las posibles métricas a recopilar en el apéndice A.1

- **Trazas distribuidas**: Istio genera trazas de seguimiento distribuidos para cada servicio, lo que brinda a los operadores una comprensión detallada de los flujos de llamadas y las dependencias de los servicios dentro de una malla. Se explicara con mas detalle el funcionamiento de las trazas distribuidas en la aparatado 3.5.1.

Otro punto importante en las arquitecturas basadas en microservicios es poder tener un registro de las peticiones que pasan a través de varios servicios junto con métricas asociadas para poder entender que componentes están involucrados en el flujo de una petición en particular y así poder encontrar cuellos de botella de manera más sencilla.

En la instalación de Istio por defecto se obtienen las siguientes herramientas de monitoreo y observabilidad que, automáticamente se configuran para coger datos de los `istio proxys` y `data plane`.

Prometheus

`Prometheus` es un sistema de monitorización y alertas de código abierto, que recoge y almacena métricas en series temporales, por lo que, por cada métrica obtenida habrá un tiempo asociado en la que se registro. `Prometheus` fue creada para monitorizar entornos de contenedores altamente dinámicos como `Kubernetes, Docker Swarm` etc por lo que

es ideal para arquitecturas basadas en microservicios.

En la arquitectura de `Prometheus` esta compuesto de los siguientes componentes:

- **Servidor Prometheus**: Realiza la tarea de monitorización y esta compuesta de tres elementos:
  1. Una **base de datos** donde se almacenan todas las métricas.
  2. Un servicio que es responsable de **escuchar o solicitar** dichas métricas de los servicios a monitorizar y guardarlos en la base de datos.
  3. Un servidor web que mediante una interfaz gráfica o una `API` que acepta `queries` para mostrar información de la base de datos. En este componentes se pueden crear `dashboards` o utilizarlo como fuente para visualiza estos datos en otras herramientas como `Grafana`.
- **Targets**: Los elementos que `Prometheus` monitoriza se denominan `targets`, estos pueden ser servidores `linux`, servidores `Windows`, servidores `Apache`, una base de datos, un `Redis` etc.
- **Métricas**: Cada `target` tiene una sus propias unidades de monitorización, por ejemplo para servidores `Linux` podrían ser métricas del estado de la `CPU`, el uso de memoria `RAM`, capacidad disponible en los distintos discos montados en el servidor ...etc. En cambio, por ejemplo, una aplicación web usara métricas como número de peticiones por segundos, cantidad de excepciones o errores de la aplicación, duración de las peticiones etc. Estos valores se guardaran en la base de datos de `Prometheus` y serán categorizados en tres tipos: contadores, indicadores y histogramas.

`Prometheus` recolecta las métricas de los `targets` siguiendo un mecanismo atípico ya que en vez de esperar a recibir los datos los solicitara a un `endpoint` de los sistemas a monitorizar, cada una de estas aplicaciones tendrán que exponer un `endpoint /metrics` y la información que se recopile de estos `endpoints` tendrá que seguir un formato que `Prometheus` comprenda. Hay aplicaciones que ya disponen por defecto de una integración con `Prometheus` y tienen este `endpoint` pero cuando no es el caso se usará del **exporter**, es un servicio que obtendrá los datos de la aplicación o sistema y los procesara al formato que `Prometheus` usa.

Las arquitecturas basadas en microservicios que usan Istio disponen de una integración por defecto con `Prometheus`, esto es gracias a que cada `istio proxy` que se despliega por cada servicio y por el cual pasa todo el tráfico dispone de un `endpoint` al cual `Prometheus` puede solicitar datos que ya están en el formato que `Prometheus` interpreta y no hará falta de configurar un `exporter`, todas esta integración con `Prometheus` es completamente independiente de la funcionalidad lógica del servicio y del software que use.

Grafana

`Grafana` es una aplicación web para la visualización y monitorización de datos, permite construir gráficas, tablas y paneles de datos personalizados además de configurar un sistema de alertas. La arquitectura típica de `Grafana` está compuesta de 3 elementos.

El primer elemento se trata de los sistemas que producen los datos que se visualizaran en `Grafana`, podría ser un servidor `Linux`, una base de datos, una aplicación web o un conjunto de estos sistemas. Ya que con Istio por cada servicio de la arquitectura se recopilaran métricas `Grafana` se ajusta correctamente, Istio generara los datos que se visualizaran con `Grafana` para un mejor análisis.

El segundo elemento es la fuente de datos, aunque Istio genere las métricas a visualizar no significa que las envié directamente a `Grafana`, estas métricas tendrán que estar almacenadas en una de las siguientes fuentes de datos:

- Alertmanager
- CloudWatch
- Azure Monitor
- Elasticsearch
- Google Cloud Monitoring
- Graphite
- InfluxDB
- Loki
- MSSQL
- MySQL
- OpenTSDB
- PostgreSQL
- **Prometheus**
- Jaeger
- Zipkin
- Tempo
- Testdata

Dentro de todas las posibilidades destaca `Prometheus` ya que Istio dispone de una integración con dicha herramienta. Por lo que usando Istio junto con `Prometheus` podemos hacer uso de `Grafana` para la monitorización.

El último elemento es el servidor de `Grafana` el cual hará consultas a la fuente de datos y así visualizar en tablas y gráficas, hay 2 tipos de datos que se visualizan los cuales son métricas como puede ser %CPU usada, %disco ocupado, numero de peticiones etc y los `logs` de aplicaciones los cuales pueden mostrar por ejemplo las trazas de errores de ciertas aplicaciones.

Jaeger

Para entender como funciona `Jaeger` hay que tener claro que es el rastreo de entornos distribuidos. El objetivo de las arquitecturas basadas en microservicios es que componentes independientes trabajen entre si para cumplir un objetivo común por ejemplo una pagina web dedicada a la venta de zapatos, el objetivo es vender, aunque siga una arquitectura basada en microservicios donde un servicio se encargue de almacenar los productos que

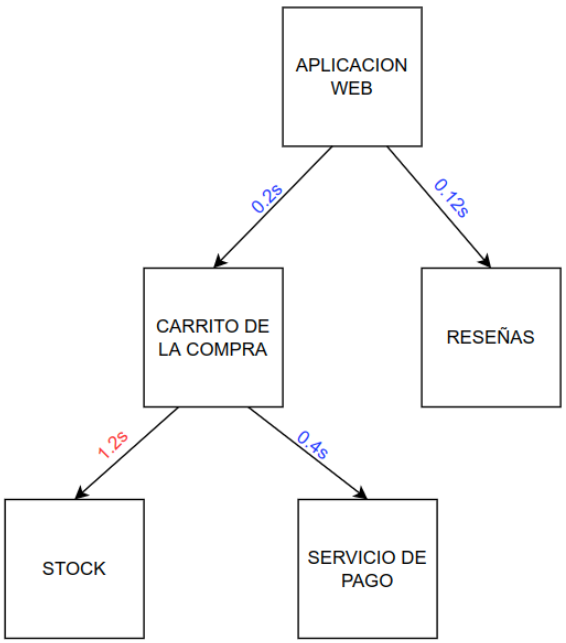


Figure 3.7: Rastreo de entornos distribuidos.

haya en el carrito, otro se encargue de mostrar las reseñas de los productos etc todos estos servicios se comunican entre si y para poder identificar rápida y eficazmente puntos de fallos y/o cuellos de botella es necesario saber que comunicaciones entre microservicios estuvieron involucrados para una solicitud determinado y cuanto tiempo duro cada comunicación 3.7.

`Jaeger` es una herramienta de rastreo distribuida de código abierto para monitorizar y encontrar puntos de fallos, la cuál sigue los estándares de `OpenTracing`. `OpenTracing` es una `API` independiente del proveedor para ayudar a los desarrolladores a instrumentar fácilmente el rastreo en su base de código. Ninguna empresa es propietaria, de hecho, muchas herramientas de rastreo como `Jaeger` respaldan a `OpenTracing` como una forma estandarizada de instrumentar el rastreo distribuido. El funcionamiento consiste en que las aplicaciones que se ejecutan en los microservicios creen `"spans"` por cada petición recibida los cuales son compartidos con el motor `OpenTracing` en este caso es `Jaeger` y que se propague un contexto de seguimiento a cualquiera de los servicios a los que llama posteriormente. Un `"span"` es un conjunto de datos que representa una unidad de trabajo de un componente, este conjunto de datos se componen por: tiempo de inicio y finalización de la unidad de trabajo, nombre de la operación, `tags y logs`.

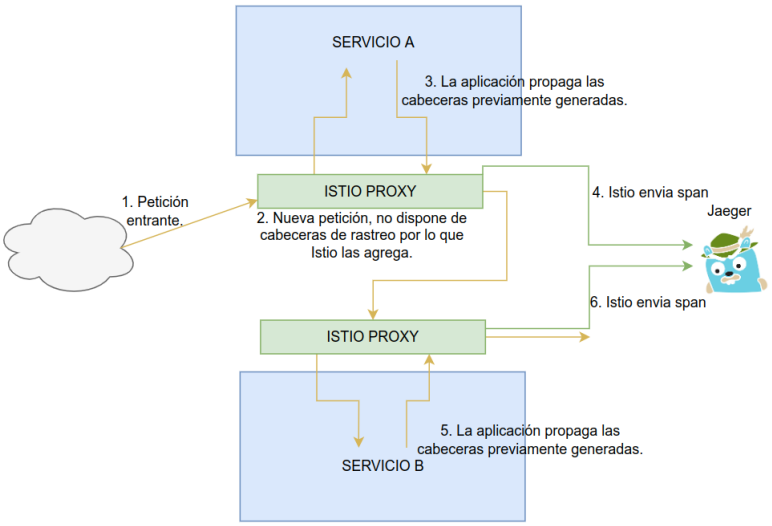


Figure 3.8: Workflow de peticiones de Istio con Jaeger.

Istio gestiona el envió y creación de los `"spans"` a `Jaeger` por lo que los desarrolladores no tendrán que implementar en su código ninguna librería de integración, cuando la petición pasa por el `istio proxy` de Istio este creara un nuevo `"trace"` si no hay ninguno en proceso, además agregara unas cabeceras `HTTP` para que `Jaeger` pueda relacionar los distintos `spans` que se vayan generando con su `trace` identificativo para su posterior análisis 3.8. Para que `Jaeger` funcione correctamente con Istio las aplicaciones tendrán que propagar las cabeceras que el `isito proxy` agrega a la peticiones. Istio generara las siguientes cabeceras:

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

3.5.2 Kiali

`Kiali` es una herramienta de visualización que junto con Istio permite mostrar la topología de una arquitectura basada en microservicios y así entender mejor su funcionamiento en ejecución. Esta herramienta recoge métricas de `Prometheus` y establece gráficas en tiempo real que permiten visualizar las comunicaciones entre los microservicios. A diferencia de `Grafana`, la cuál se especializa en paneles donde se visualizan métricas, tablas, contadores etc `Kiali` se centra en mostrar dinámicamente mapas de como están distribuidos los ser-

vicios y el estado de la comunicación entre ellos,las figuras 3.4, 3.2, 3.3 corresponden al panel de `Kiali`.

Para mostrar estos diagramas `Kiali` usa datos que `Prometheus`, recoge del plano de control y plano de datos de Istio, además de los datos que almacena `Jaeger`. En los diagramas se puede observar:

- Flujo de tráfico entre microservicios.
- Números de bytes, peticiones, errores etc.
- Múltiples flujos de tráfico para múltiples versiones (como versión Canary o enrutamiento ponderado)
- Estado de la aplicación basado en el tráfico de la red.
- Fallos de red, que se pueden identificar rápidamente.

Dado que `Kiali` es una herramienta especifica para Istio desde su panel se pueden ver como esta configurado el enrutamiento de tráfico entre los microservicios, si esta entrando en juego una de las políticas de reintentos o esta saltando el tiempo de espera, clonación de tráfico etc.

El funcionamiento de Istio permite integrarse correctamente con herramientas como `Grafana`, `Prometheus`, `jaegger`, `kiali` etc. Las cuales permiten ganar una gran observabilidad sobre la arquitectura y de esta manera evitando agregar complejidad a los microservicios.

3.6 Seguridad

La seguridad es un factor critico para proteger los datos que una aplicación gestione ya que un usuario no autorizado no debería poder acceder a los datos ni modificarlos. Para proteger los datos de una aplicación es necesario:

- **Autenticación** y **autorización** a un usuario antes de otórgale acceso a los datos.
- **Cifrado** de los datos a enviar en las comunicaciones entre servicios.

Tanto las arquitecturas compuestas de microservicios como las monolíticas necesitan implementar mecanismos de seguridad pero los microservicios tienen muchas mas interconexiones y todas las comunicaciones que realicen irán a través de la red al contrario de

los monolíticos ya que estas arquitecturas disponen de menos conexiones y se ejecutan en entornos mas estáticos como maquinas físicas o virtuales, al ejecutarse en estos entornos la dirección IP se convierte en una buena fuente de identidad y por eso generalmente usan certificados y/o reglas de cortafuegos para la autenticación. En cambio los microservicios fácilmente crecen en cientos o miles por los que se ejecutan en entornos mas dinámicos y esto convierte en inviable los métodos tradicionales de identidad como la dirección IP por lo que Istio usa `SPIFFE` la cuál es un grupo de estándares de código abierto para implementar identidad a cargas de trabajo altamente dinámicas.

La identidad `SPIFFE` es un `URI` compatible con `RFC 3986` compuesto en el formato `spiffe`://**dominio**/**ruta**

- **Dominio**, representa el emisor de identidades bien como individuo u organización.
- **ruta**, identifica una carga de trabajo del dominio de confianza .

Istio completa esta ruta con la cuenta de servicio bajo la cual se ejecuta una carga de trabajo particular. Esta identidad `SPIFFE` está codificada en un certificado X.509 gestionado por el plano de control. Estos certificados luego se utilizan para asegurar el transporte para la comunicación de servicio a servicio al cifrando los datos.

3.6.1 Cifrado Automático

El trafico entre servicios que dispongan del `istio proxy` será cifrado y mutuamente autenticado por defecto ya que el plano de control emite los certificados a los servicios y gestiona el rotado de estos evitando que expiren y la comunicación entre servicios falle, todo este proceso es ajeno a la lógica de aplicación del servicio gracias a que todo el trafico que va a un servicio pasa por su `istio proxy` correspondiente 3.9.

Aunque por defecto la comunicación entre servicios este cifrada Istio proporciona la opción de usar comunicaciones no seguras aunque estos tienen que estar previamente declarados en la configuración. En grandes arquitecturas basadas en microservicios es complejo introducir el `istio proxy` de Istio sobre todos los microservicios a la vez sobretodo en entornos productivos por lo que lo recomendable es ir introduciéndolos poco a poco para poder realizar todas las pruebas oportunas. Istio dispone de 2 recursos para la configuración de la seguridad sobre la arquitectura: **PeerAuthentication** y **AuthorizationPolicy**.

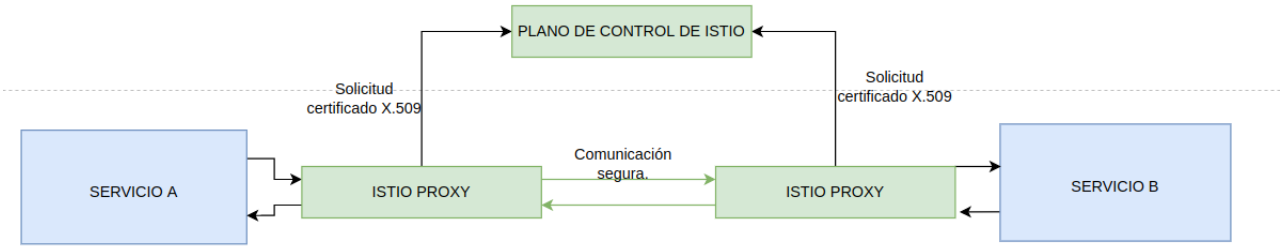


Figure 3.9: Comunicación segura usando certificados x.509 emitidos por el plano de control

PeerAuthentication

Con este recurso configuraremos que microservicios requerirán que sus comunicaciones vayan cifradas o en texto claro utilizando el modo de autenticación **permisiva** o **estricta**. Este recurso se puede configurar en tres niveles:

- A nivel global, la configuración definida se aplicara a todos los microservicios que compongan la arquitectura.
- A nivel de *namespace*, la configuración aplicara a todos los microservicios que pertenezcan al mismo *namespace*.
- A nivel de servicio, la configuración aplicara a uno o un grupo de microservicios específicos.

Se pueden configurar los 3 niveles simultáneamente para ganar mayor granularidad manteniendo estándares de seguridad sobre toda la arquitectura, por ejemplo: En un cluster de `kubernetes` donde tengamos Istio instalado y haya 2 `namespaces`. Si queremos que por defecto todas las comunicaciones entre los microservicios vayan encriptadas configuraremos el `PeerAuthentication` a nivel global.

```
apiVersion: "security.istio.io/v1beta1"
kind: "PeerAuthentication"
metadata:
  name: "default"
  namespace: "istio-system"
spec:
  mtls:
    mode: STRICT
```

El modo de autenticación será `STRICT` y se configurara sobre el `namespace` "istio-sytem" el cuál es donde están todos los servicios que componen el plano de control. Al configurarlo sobre este `namespace` Istio entenderá que todas las comunicaciones deben de

ir cifradas. Si por ejemplo en un `namespace` en concreto no quisiéramos que las comunicaciones fueran cifradas configuraríamos otro `PeerAuthentication` a nivel de `namespace`.

```
apiVersion: "security.istio.io/v1beta1"
kind: "PeerAuthentication"
metadata:
  name: "default"
  namespace: "X_NAMESPACE"
spec:
  mtls:
    mode: PERMISSIVE
```

El modo de autenticación sera `PERMISSIVE` y dado que esta configurado sobre el `namespace` X_NAMESPACE solo en este `namespace` se permitirán las comunicaciones no cifradas por lo que esta configuración a nivel de `namespace` tiene prioridad con la configuración a nivel de global, esto es útil ya que si por defecto queremos que sobre todos los `namespace` la comunicación vaya cifrado salvo por uno no tendremos que ir uno a uno configurando el `PeerAuthentication`. La siguiente capa de granularidad es la configuración a nivel de servicio la cuál nos permite indicar un único servicio sobre todas la arquitectura en el cual permitamos que las comunicaciones no vayan cifradas.

```
apiVersion: "security.istio.io/v1beta1"
kind: "PeerAuthentication"
metadata:
  name: "app"
  namespace: "Y_NAMESPACE"
spec:
  selector:
    matchLabels:
      app: "app"
  mtls:
    mode: PERMISSIVE
```

Gracias a las distintas capas en las que se pueden configurar el cifrado de las comunicaciones permiten una configuración flexible y adaptable a las distintos requisitos y problemas que puedan surgir por ejemplo con los 3 bloques previos se ha configurado que por defecto sobre todos los `namespaces` las comunicaciones vayan cifradas excepto en el `namespace` X_NAMESPACE y un servicio en particular del `namespace` Y_NAMESPACE.

AuthorizationPolicy

La autorización es el proceso que define si una persona o aplicación ya autenticado tiene permisos para realizar un conjunto de operaciones. Cuando se configuran estas políticas el plano de control actualiza los `istio proxy` acoplado a cada servicio indicándole que políticas debe seguir por lo cual es el `istio proxy` el que aplica la política por lo que el proceso es muy rápido y eficiente. Un ejemplo de la configuración del recurso es:

```yaml
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: httpbin
  namespace: foo
spec:
  selector:
    matchLabels:
      app: webapp
  action: ALLOW
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/default/sa/sleep"]
    - source:
        namespaces: ["test"]
    to:
    - operation:
        methods: ["GET"]
        paths: ["/info*"]
    - operation:
        methods: ["POST"]
        paths: ["/data"]
```

El *AuthorizationPolicy* esta compuesto de 3 elementos:

- **selector**: Define sobre que servicios se aplicaran la política.
- **rules**: Define un lista de reglas que identifican una petición para la cual se activará la política.
- **action**: Una vez una de las reglas identifica una petición la rechazara o aceptara dependiendo el valor de este campo el cual puede ser *ALLOW* o *DENY*. La petición la identificara basándose en 2 factores. quien hace la petición y que tipo de petición es, en el ejemplo anterior se permite las peticiones tipo `GET` al path /info* y `POST` al path /data que vengan del servicio sleep del namespace test. Esta politica solo se aplicara al servicio de webapp.

Es importante tener en cuenta que una vez se configure sobre un servicio una política de autorización con el `action ALLOW` por defecto rechazará todo el tráfico que no cumpla la política por lo que dependiendo de las necesidad del servicio puede ser más interesante configurar políticas con el `action deny` por que en este caso salvo las petición que estamos rechazando el resto serán admitidas.

Al igual que *PeerAuthentication* el *AuthorizationPolicy* se puede configurar a nivel global, `namespace` o servicio aportando una gran versatilidad dependiendo de los requisitos que tengan los microservicios que compongan la arquitectura.

Chapter 4

Conclusión y futuras lineas de estudio

Este proyecto ha sido un gran desafió ya que para entender como funciona Istio y los beneficios que aporta hay que tener claro muchos conceptos y así poder entrelazar estos conocimientos para darle un sentido. Además se ha sido ambicioso queriendo crear entornos de pruebas lo mas personalizados posibles por lo que simplemente para poder desplegar Istio en un entorno ha sido necesario aprender a usar herramientas como `Kubernetes, Docker, Terraform, GCP, Python` etc... Una vez desplegado Istio ha sido necesario crear las unidades de trabajo sobre las cuales se iban a ser las pruebas. La curiosidad ha sido el causante del uso de tantas herramientas ya que por ejemplo se podría haber utilizado imágenes de `Docker` proporcionadas en las misma documentación de Istio para realizar las pruebas pero se ha decido hacer pequeños desarrollos en `Flask`[1] de los cuales apenas se ha hecho referencia ya que están fuera del alcance y objetivos de este proyecto aunque han causados varios dolores de cabeza pero bueno el conocimiento no ocupa lugar.

A lo largo del proyecto se ha comprobado que Istio aporta muchas funcionalidades a la hora de la gestión de arquitecturas basadas en microservicios que por defecto no vienen en orquestadores de contenedores como Kubernetes estas funcionalidades se dividen en tres grupos:

1. Control del trafico: Con Istio se puede controlar de forma granular el comportamiento del trafico entrante y saliente de la arquitectura y así poder implementar diferentes tipos de despliegues, implementar mecanismos de resiliencia de red como los reintentos, tiempos de espera, circuit breakers, configurar distintos tipos de balanceos de carga, clonar el trafico para hacer pruebas con trafico real etc...

[1] Framework de Python para la creación de desarrollos de aplicaciones webs

2. Observabilidad: Las capacidades de recolección de métricas que aporta Istio sobre los microservicios que componen la arquitectura hace posible una buena integración con herramientas como `Prometheus, Kiali, Grafana, Jaeger` etc... Las cuales otorgan visibilidad y entendimiento de lo que esta pasando en la arquitectura.

3. Seguridad: Istio permite que las comunicaciones entre los microservicios sean seguras sin que la lógica de aplicación se vea afectada, además permite configurar políticas de autorización para limitar que servicios pueden comunicarse con cada microservicio.

Tras estudiar y analizar las funcionalidades que aporta Istio no dudaría en ponerlo en practica siempre que fuera posible ya que todo indica que las arquitecturas basadas en microservicios han venido para quedarse al igual que Istio la cual evoluciona bajo un continuo desarrollo.

Chapter 5

Presupuesto

5.1 Presupuesto de recursos materiales y software

Los gatos relacionados con los materiales para la realización de este proyecto se describen a continuación:

1. PC portatil HP Spectre X360 1500€.
2. Uso de infraestructura GCP 120€.
3. Cuostas de conexion a Internet 55€/mensuales

La suma total de los gastos de recursos materiales y software acumulan a 2940€.

5.2 Presupuesto de recursos humanos

Basándonos que en España el coste de la mano de obra de un empleado con estudios relacionados a una ingeniería de telecomunicaciones esta en los 15,45€se ha tomado este valor de referencia para los siguientes cálculos mostrados en la tabla 5.1.

5.3 Presupuesto completo

Uniendo los gastos asociados a los materiales usados y a la mano de obra empleada suma un total de 7977€5.3.

Tarea	Horas empleadas	Coste
Investigación previa a las distintas tecnologías a usar para poder realizar las pruebas sobre Istio.	50	772€
Despliegue de la infraestructura en GCP.	15	231€
Redacción y experimentos relacionados con la gestión tráfico usando Istio.	90	1390€
Redacción y experimentos relacionado con la observabilidad usando Istio.	70	1081€
Redacción y experimentos relacionados con la seguridad usando Istio.	50	722€
Total	350	5057€

Table 5.1: Presupuesto de mano de obra.

Concepto	Precio total
Recursos materiales y software.	2940
Recursos humanos	5057
Total	7977

Table 5.2: Resumen de los resultados obtenidos tras activar el `circuit breaker` en la configuración de Istio.

Table 5.3: Presupuesto total.

Appendix A

A.1 Observabilidad

Métricas del envoy proxy

Ejemplo de algunas métricas que se pueden obtener del envoy proxy son:

- envoy_cluster_internal_upstream_rq
- envoy_cluster_upstream_rq_completed
- envoy_cluster_ssl_connection_error
- envoy_cluster_lb_subsets_removed

Para obtener mas información de todas las métricas disponibles ver [11].

Métricas a nivel de servicio

Ejemplo de algunas métricas que se pueden obtener servicio de Istio son:

```
istio_requests_total{
  connection_security_policy="mutual_tls",
  destination_app="details",
  destination_canonical_service="details",
  destination_canonical_revision="v1",
  destination_principal="cluster.local/ns/default/sa/default",
  destination_service="details.default.svc.cluster.local",
  destination_service_name="details",
  destination_service_namespace="default",
  destination_version="v1",
  destination_workload="details-v1",
  destination_workload_namespace="default",
  reporter="destination",
  request_protocol="http",
  response_code="200",
```

```
16   response_flags="-",
17   source_app="productpage",
18   source_canonical_service="productpage",
19   source_canonical_revision="v1",
20   source_principal="cluster.local/ns/default/sa/default",
21   source_version="v1",
22   source_workload="productpage-v1",
23   source_workload_namespace="default"
24 } 214
```

Para obtener mas información de todas las métricas disponibles ver [11].

Bibliography

[1] *Architecture overview*. Accessed: 13/10/2021. URL: https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/arch_overview.

[2] *Arquitectura de maquinas virtuales*. https://istio.io/latest/docs/ops/deployment/vm-architecture/. Accessed: 22/11/2021.

[3] *Canary Release*. https://martinfowler.com/bliki/CanaryRelease.html. Accessed: 22/11/2021.

[4] *Código de la aplicación web desarrollada en python*. https://github.com/CristianCamilo98/TFG/tree/main/web-app-demo. Accessed: 21/04/2021.

[5] Ed Comer. *Canonical Data Model Design Guidelines*. Accessed: 14/10/2021. 2010.

[6] *Estructura de Istio, control plane y dataplane*. https://istio.io/latest/docs/ops/deployment/architecture/. Accessed: 13/10/2021.

[7] *Imagen de la estructura del Envoy proxy*. https://www.envoyproxy.io/docs/envoy/latest/_images/lor-architecture.svg. Accessed: 13/10/2021.

[8] *Istio Canary Deployments*. https://docs.flagger.app/tutorials/istio-progressive-delivery. Accessed: 6/12/2021.

[9] *Istio en ejemplos*. https://istiobyexample.dev. Accessed: 2/05/2022.

[10] *Kubernetes*. https://kubernetes.io/docs/concepts/overview/. Accessed: 21/08/2021.

[11] *Métricas de Prometheus*. https://prometheus.io/docs/concepts/metric_types/. Accessed: 2/08/2022.

[12] *Métricas del envoy proxy*. https://www.envoyproxy.io/docs/envoy/latest/configuration/upstream/cluster_manager/cluster_stats. Accessed: 21/08/2021.

[13] *Métricas proporcionadas por Istio*. https://www.envoyproxy.io/docs/envoy/latest/configuration/observability/statistics. Accessed: 2/05/2022.

[14] *Microservicio usado para realizar pruebas*. https://github.com/nicholasjackson/fake-service. Accessed: 21/07/2022.

[15] *Monitorizacion con prometheus*. https://sysdig.com/blog/kubernetes-monitoring-prometheus/. Accessed: 2/08/2022.

[16] *Opciones de enrutado de Envoy*. https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/router_filter#x-envoy-retry-grpc-on. Accessed: 2/05/2022.

[17] *Opentracing*. https://www.sentinelone.com/blog/what-is-opentracing/. Accessed: 21/08/2022.

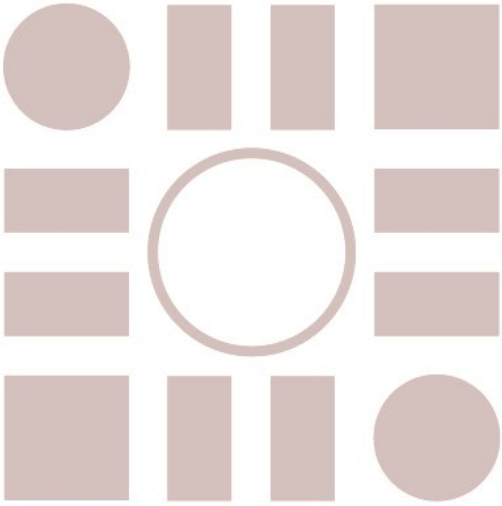
[18] Christian Posta. *Introducing Istio Service Mesh for Microservices: Build and Deploy Resilient, Fault-tolerant Cloud-native Applications*. Accessed 14/10/2021. O'Reilly Media, 2018.

[19] *Repositorio de la infraestructura*. https://github.com/CristianCamilo98/TFG. Accessed: 21/04/2021.

[20] *The power of 2 choices*. https://www.eecs.harvard.edu/~michaelm/postscripts/handbook2001.pdf. Accessed: 19/04/2022.

[21] *What is blue green deployment?* https://www.redhat.com/en/topics/devops/what-is-blue-green-deployment. Accessed: 17/11/2021.

Universidad de Alcalá
Escuela Politécnica Superior



ESCUELA POLITECNICA
SUPERIOR