Service Mesh en Kubernetes con Istio

Sergio García Ojeda
Administración de Sistemas Informáticos en Red
Trabajo de Fin de Grado

Índice

1. Introducción

Desde hace años, los servicios informáticos, son una de las bases de la informática, y desde no hace tantos años, estos servicios, se han ido descomponiendo en servicios muchos más pequeños destinados a desarrollar una única función. Estos pequeños servicios, llamados microservicios, están programados para trabajar de manera independiente, pero en conjunto, desarrollar una función final.

Hace relativamente pocos años, surgieron la idea de los contenedores de aplicaciones, como por ejemplo Docker, que ejecutan una aplicación o servicio, dentro del equipo principal, pero aislada. La idea era replicar esa misma estructura en cualquier equipo que tuviese este software instalado.

Poco tiempo después, salió a la luz la idea de Kubernetes, un orquestador de estos contenedores, que permitía un fácil despliegue de estos contenedores y el mantenimiento de estos.

Llegados a este punto, se podría decir que la idea de los microservicios ya ha llegado a un punto en el que no se puede avanzar más hasta un futuro indefinido. Pero, ¿es esto así? A raíz de esta estructura de microservicios, surgió la idea del Service Mesh, una forma de controlar la comunicación y el funcionamiento de los diferentes microservicios entre ellos.

Así es como, llegamos a Istio. Un sistema para implementar el service mesh en Kubernetes.

En este documento se va a explicar en profundidad que es Istio, como funciona y las ventajas y desventajas de usarlo, y llegaremos a la conclusión de si verdaderamente es útil; y cuando y por qué lo implementaremos.

2. ¿Qué es Kubernetes?

Kubernetes es un sistema open source creado por Google para la gestión de aplicaciones en contenedores. Es un sistema de orquestación para contenedores, que permite acciones como programar el despliegue, escalado y monitorización de nuestros contenedores, entre muchas otras más.

2.1. Escenario que vamos a montar en Kubernetes.

En nuestro caso, vamos a montar un cluster de kubernetes con kubeadm. Hemos elegido este método, por su fácil instalación, que se llevará acabo en las siguientes maquinas:

- Nieve. Maquina Debian de 4GB de RAM. Sera el Master.
- Sansa. Maquina Debian de 2GB de RAM.
- Arya. Maquina Debian de 2GB de RAM.

La instalación se llevará acabo de la siguiente manera:

2.2. Instalación de Docker

Lo primero que haremos será instalar Docker en nuestras tres máquinas, al ser las tres máquinas Debian, la instalación se llevará acabo de la siguiente manera:

Instalamos los paquetes necesarios para poder instalar paquetes con apt mediante https.

```
sudo apt install apt-transport-https ca-certificates curl gnupg2
software-properties-common
```

Añadimos las claves GPG oficiales de Docker:

```
curl -fsSL https://download.docker.com/linux/debian/gpg | sudo apt-key
add -
```

Añadimos el repositorio de Docker:

```
sudo add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/debian \
$(lsb_release -cs) \
stable"
```

Actualizamos los repositorios e instalamos Docker:

```
sudo apt-get update
sudo apt-get install docker-ce
```

2.3. Instalación de kubeadm

Ahora deberemos instalar kubeadm, kubelet y kubectl en las 3 máquinas. Para ello, seguimos los pasos que aparecen en la documentación de kubernetes, que se basa en añadir la clave GPG de kubernetes, su repositorio, actualizar los repositorios e instalar los paquetes necesarios:

```
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key
add -
cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF
apt-get update
apt-get install -y kubelet kubeadm kubectl
```

Ahora deberemos inicializar el nodo nieve como master, para ello ejecutamos:

```
kubeadm init –pod-network-cidr=10.0.0.0/16
```

Si todo sale bien, se nos proporcionara un comando con la siguiente estructura:

```
kubeadm join –token <token> <master_ip>:<master-port>
```

Esto nos servirá más adelante para añadir los otros nodos al master.

También nos proporcionara las siguientes líneas, que ejecutaremos para poder utilizar kubeadm con nuestro usuario:

```
mkdir -p $HOME/.kube
sudo cp-i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOE/.kube/config
```

2.4. Instalación de addon de red interna

A continuación, instalaremos un addon para gestionar la red interna por la que se comunicaran los nodos. Entre la gran lista de opciones, nosotros nos hemos decantado por weave. Su instalación se realizará ejecutando la siguiente sentencia:

```
kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-
version=$(kubectl version | base64 | tr -d '\n')”
```

Una vez instalado, podremos ver si su instalación es correcta ejecutando:

```
root@nieve:/home/debian# kubectl get pods -n kube-system
NAME                              READY   STATUS    RESTARTS   AGE
coredns-86c58d9df4-9bws4          1/1     Running   0          12m
coredns-86c58d9df4-ljwjr          1/1     Running   0          12m
etcd-nieve                        1/1     Running   0          11m
kube-apiserver-nieve              1/1     Running   0          11m
kube-controller-manager-nieve     1/1     Running   0          11m
kube-proxy-4cwm9                  1/1     Running   0          10m
kube-proxy-rg8xl                  1/1     Running   0          10m
kube-proxy-z5vsg                  1/1     Running   0          12m
kube-scheduler-nieve              1/1     Running   0          11m
weave-net-4rb8n                   2/2     Running   0          10m
weave-net-8wp6j                   2/2     Running   1          10m
weave-net-bd4hd                   2/2     Running   0          11m
```

2.5. Añadir nodos al master

Solo nos faltaría añadir los nodos al master, para ello, ejecutamos el comando que mencionamos antes que nos proporcionó kubadm al convertir a nieve en master, en todos los nodos.

```
kubeadm join <ip_master:port> --token <token> --discovery-token-ca-cert-
hash <hash>
```

Por último, ejecutamos el siguiente comando para ver si efectivamente se ha creado correctamente el cluster. Todo estará correcto si el “STATUS” está en Ready.

```
root@nieve:/home/debian# kubectl get nodes
NAME    STATUS   ROLES    AGE   VERSION
arya    Ready    <none>   13m   v1.14.2
nieve   Ready    master   15m   v1.14.2
sansa   Ready    <none>   13m   v1.14.2
```

3. ¿Qué es el Service Mesh?

Según “techtarget”, un service mesh es una capa de infraestructura que controla la comunicación entre los servicios de una red. Esta tiene el control del envio y entrega de solicitudes entre los servicios. Entre las características comunes que proporciona un service mesh, destacan: el descubrimiento de servicios, el balanceo de carga, el cifrado, la tolerancia a fallos y la resiliencia. En resumen, el service mesh hace que la comunicación entre servicios sea rápida, confiable y segura.

3.1. ¿Cómo funciona un Service Mesh?

Un Service Mesh, normalmente usa un tipo de proxy conocido como Sidecar. Un “Proxy Sidecar”, se vincula a un servicio para ampliar o añadir funcionalidades. Su nombre proviene de la semejanza de su implantación con el sidecar de una moto.

En el caso que estamos desarrollando, y al ser una estructura de microservicios en Kubernetes, cada proxy sidecar se implantará en cada pod.

Una vez implementado el proxy, se podrán distinguir dos “capas” en el Service Mesh: el data plane(o plano de datos) y el control plane(o plano de control).

El data plane está formado por las instancias, los sidecars y sus interacciones; mientras que, por otro lado, el control plane, es el encargado de la administración de tareas, la monitorización, y la implementación de políticas.

3.2. Estudió de un Service Mesh. Ventajas y Desventajas

La implementación de un Service Mesh resuelve algunos problemas clave a la hora de montar una estructura de microservicios, pero no todos. Entre las ventajas que destacan a la hora de montar una Service Mesh destacan:

- Simplifica la comunicación entre los microservicios.
- Facilidad a la hora de encontrar fallos en la comunicación, ya que esta se encuentra “empaquetada” en su propia capa de infraestructura.
- Permite el cifrado, la autenticación y la autorización.
- Permite un desarrollo, prueba y despliegue de las aplicaciones.
- Mejora en la administración de las redes de un cluster de contenedores.

Por otro lado, también existen desventajas relacionad.as con la implantación de un Service Mesh, entre ellas:

- El tiempo de ejecución de las instancias aumenta exponencialmente.
- Añade un paso más en la comunicación, ya que la comunicación debe pasar a través del proxy.

4. ¿Qué es Istio?

Istio, es una plataforma abierta para gestionar, conectar y securizar microservicios, o, dicho de otra manera, es una forma de implementar “service mesh” en nuestra estructura de microservicios.

Entre las ventajas que nos da Istio (y una vez estudiado que es el “Service Mesh”), podemos enumerar, las opciones que nos provee:

- Balanceo de carga y enrutado inteligente.
- Tolerancia a fallos.
- Aplicación de políticas en todos los servicios.
- Telemetría y generación de informes y reportes en profundidad.

El mayor punto a favor de Istio reside en la facilidad de crear una red de servicios desplegados con balanceo de carga, autenticación, monitorización, etc sin realizar ningún cambio en el código de cada servicio, ya que el soporte es agregado a los servicios mediante la implementación de un proxy de tipo sidecar, nombrado en Istio como “Envoy”.

Actualmente sólo permite despliegues sobre Kubernetes, aunque se tiene previsto que se añada su soporte para otros entornos en el futuro.

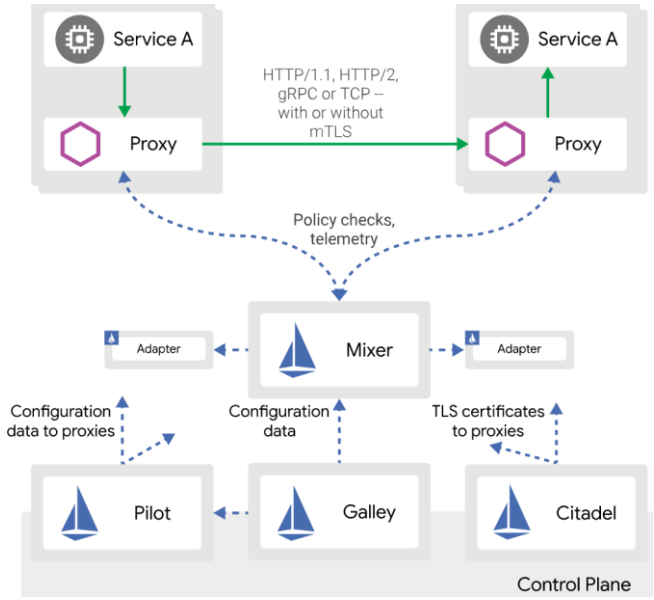
Como vimos a la hora de ver que es un Service Mesh, la arquitectura de Istio se divide en:

- Plano de datos, formado por los Envoy, que se encargan de controlar el tráfico de red de todas las comunicaciones entre microservicios.

- Plano de control, es responsable de gestionar y configurar los proxies y de gestionar las políticas.

4.1. Componentes de Istio

El siguiente diagrama, que nos proporciona Istio a través de su web, nos muestra los componentes que forman Istio, y el plano en el que trabajan:



https://istio.io/docs/concepts/what-is-istio/arch.svg

Podemos diferenciar diferentes partes dentro de este diagrama, que forman el "cuerpo" de Istio:

- **Envoy:** Se trata del proxy sidecar que usa Istio, el cual usa para usar las características que provee: balanceo de carga, protocolo TLS, HTTP/2, etc. Este se despliega dentro del mismo pod de kubernetes.

- **Gallery:** Es el encargado de la validación, ingestión, procesamiento y distribución de la configuración de Istio. También aísla los componentes propios de Istio, de la configuración propia de Kubernetes.

- **Mixer:** La principal función de este componente es aplicar políticas y Access control a partir de la información obtenida de Envoy.

- **Pilot:** Provee descubrimientos de servicios para Envoy, así como, enrutado inteligente (aplicación de tests) como también resiliencia (reintentos, tiempo de espera, etc).

- **Citadel:** Por último, Citadel, se encarga tanto de la comprobación de credenciales a nivel de servicio como a nivel de usuario.

4.2. Objetivos

Istio se ajusta a una serie de objetivos en su diseño, para maximizar su rendimiento y facilitar su uso de cara al usuario. Entre estos objetivos destacan:

- Máxima transparencia: La principal idea de Istio, es que se requiera el mínimo esfuerzo si se quiere adoptar este.
- Incrementabilidad: El sistema está planteado para que crecer según las necesidades del entorno, dado que se creara una dependencia a Istio.
- Portabilidad: Istio está pensado para ser usado en cualquier plataforma con el coste mínimo de esfuerzos y recursos.
- Uniformidad de política: Permite aplicar políticas a todo lo que controla, este expuesto al exterior o no.

4.3. Funcionalidades

En este apartado, vamos a ver las funcionalidades que nos aporta Istio en nuestra red de microservicios. Estas funcionalidades se pueden agrupar en cinco importantes grupos:

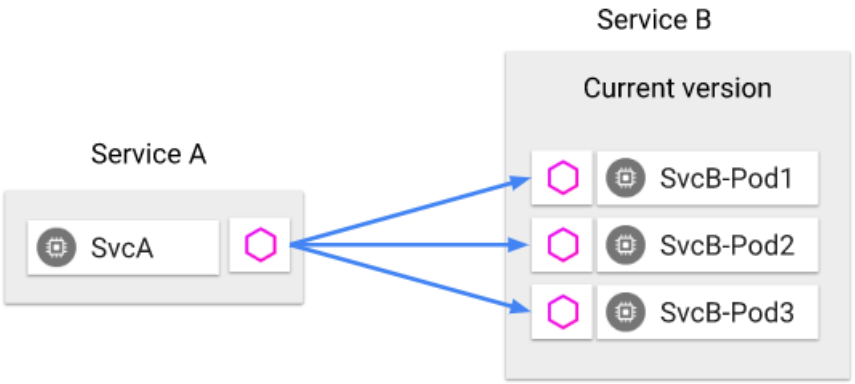
- Gestión del tráfico.
- Seguridad.
- Políticas y Telemetría.
- Rendimiento y Escalabilidad.

Aunque solo las tres primeras pertenecen a las funciones que podemos configurar, pienso que el cuarto se deberá explicar para aclarar conceptos.

Una vez enumerados estos grupos, procederemos a explicar un poco de que se encarga cada parte de estas funcionalidades.

4.3.1. Gestión del tráfico

Si recordamos, cuando explicamos los componentes de Istio, nombramos el componente Pilot que gestiona y configura el envoy, y vimos como provee de enrutado inteligente a la red que montamos. Es así como Istio desautoriza la conexión interna propia del cluster de kubernetes, y toma el control de esta, especificando que reglas deben seguir cada pod. Por ejemplo, en la página oficial de Istio, nos muestra este escenario:

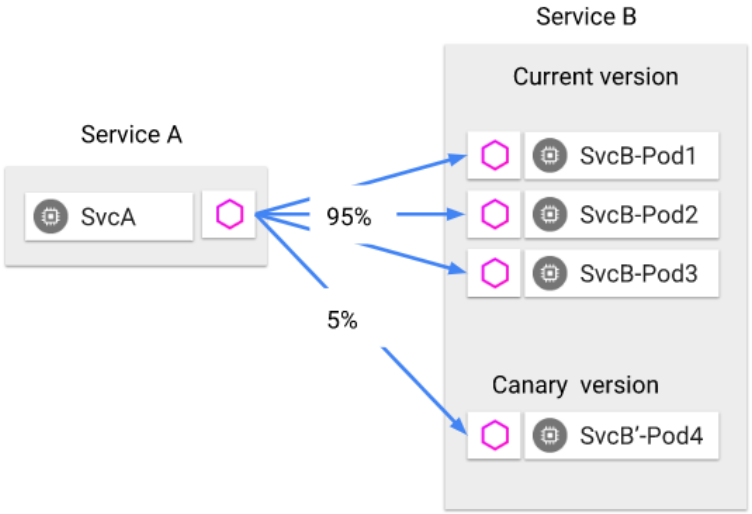


https://istio.io/docs/concepts/traffic-management/TrafficManagementOverview.svg

En él, vemos como tenemos dos servicios, el servicio A y el servicio B. El servicio A formado por un pod llamado SvcA y el Servicio B, formado por 3 pods: SvcB-Pod1, SvcB-Pod2 y SvcB-Pod3. Cada uno de estos pods, tiene asociado un proxy envoy que se encarga de la comunicación entre servicios.

Actualmente, las peticiones llegan del Service A al Service B de una forma equitativa a cada pod, asi pues, un 33,3% de peticiones llegaran a SvcB-Pod1, etc.

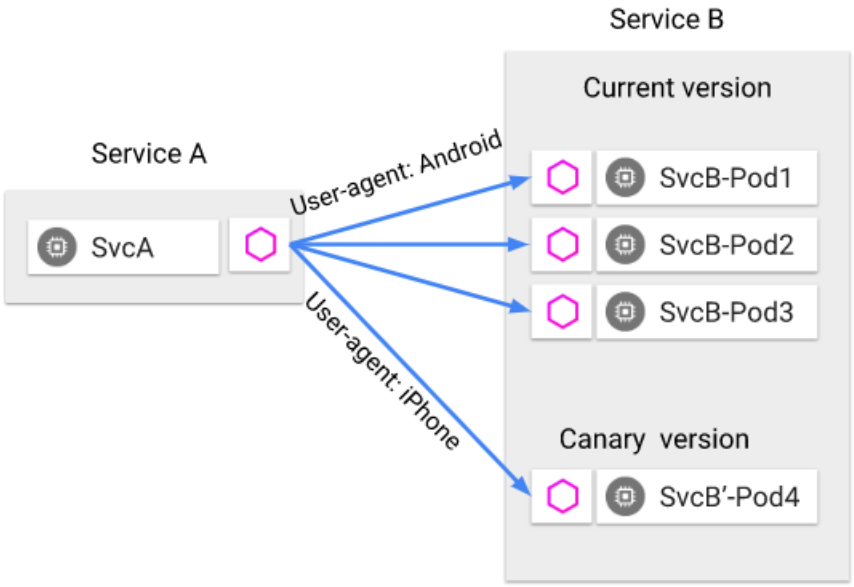
Ahora imaginemos que queremos desarrollar una versión de pruebas(canary version) en otro pod dentro del Service B, pero queremos que las peticiones que le lleguen a esta sean una pequeña parte de la total, con Istio podemos establecer qué porcentaje de peticiones llegan a cada pod, y como veremos en el siguiente ejemplo, podemos establecer que a los 3 pods que teníamos desde el principio le llegue un 95% de las peticiones, y a la canary version solo un 5%.



https://istio.io/docs/concepts/traffic-management/TrafficManagementOverview.svg

Es más, no solo podremos filtrar las peticiones por cantidad de peticiones, tambien podremos filtrar por el tipo de equipo que nos hace la petición. Por ejemplo, si tenemos una pagina que se muestra correctamente en equipos Android, y estamos

desarrollando una versión para dispositivos Iphone, podemos hacer una versión de pruebas en un nuevo pod, y que todas las peticiones que lleguen con cabecera de Iphone sea enviadas a la versión de pruebas:



https://istio.io/docs/concepts/traffic-management/TrafficManagementOverview.svg

En definitiva, una vez implementado Istio, sera este el que tome el control de las comunicaciones entre los servicios.

4.3.2. Seguridad

Al descomponer un sistema monolítico, a un sistema de microservicios, obtenemos unas grandes ventajas que ya hemos visto anteriormente, pero estos microservicios también tienen necesidades de seguridad particulares:

- Necesitan un trafico cifrado para evitar ataques “Man-in-the-middle”.
- Establecer un sistema de auditorías, para saber quién hizo qué a que hora.
- Políticas de acceso mutuo y TLS mutuo para proporcionar un control de acceso flexible.

La seguridad de Istio intenta proporcionar una solución de seguridad integra para resolver todos estos problemas, e intetna mitigar las amenzas internas y externas contra sus datos, puntos finales, comunicación y plataforma.

Estas características de seguridad propias de Istio proporcionan una identidad sólida, una política eficaz, un cifrado TLS transparente y herramientas de autenticación, autorización y auditoría para proteger los servicios y loas datos. Sus principales objetivos son:

- Seguridad por defecto. Para efectuar cambios en la seguridad de nuestro sistema no necesitaremos realizar cambios en el código, como ya vimos, esto es uno de los objetivos primordiales de Istio.

- Defensa en profundidad. Si ya tenemos sistemas de seguridad establecidos en nuestro sistema, la seguridad que implementa Istio se integrara con ellos, proporcionando múltiples capas de seguridad.

- Red de confianza nula. De manera predeterminada, Istio establece una red de confianza nula, es decir, Istio mantiene un control de acceso estricto y no confía en nadie de forma predeterminada, incluso, en aquellos dentro del perímetro de la red.

Por último, vemos que los componentes que se involucran en la seguridad de Istio son los siguientes:

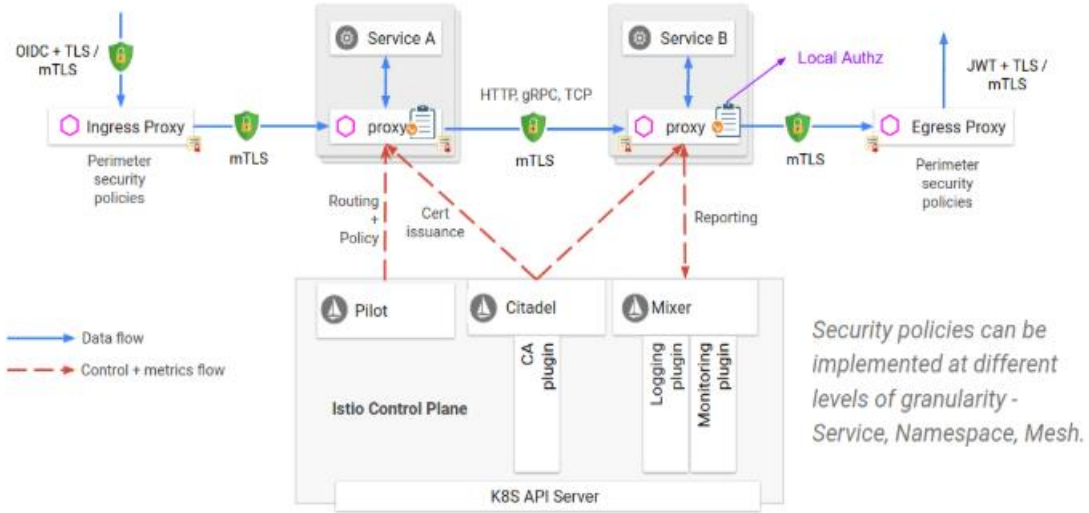
- Citadel. Se encarga de la gestión de claves y certificados.

- Proxy. Implementa la comunicación entre los clientes y los servidores.

- Pilot. Además de implementar enrutado, en la parte de seguridad también es el encargado de distribuir políticas de autenticación y de la resolución de nombres entre los proxies.

- Mixer. Gestiona las autorizaciones y las auditorias.

El funcionamiento de todos estos componentes lo podemos ver en conjunto en la siguiente imagen (propiedad de Istio):



https://istio.io/docs/concepts/security/architecture.svg

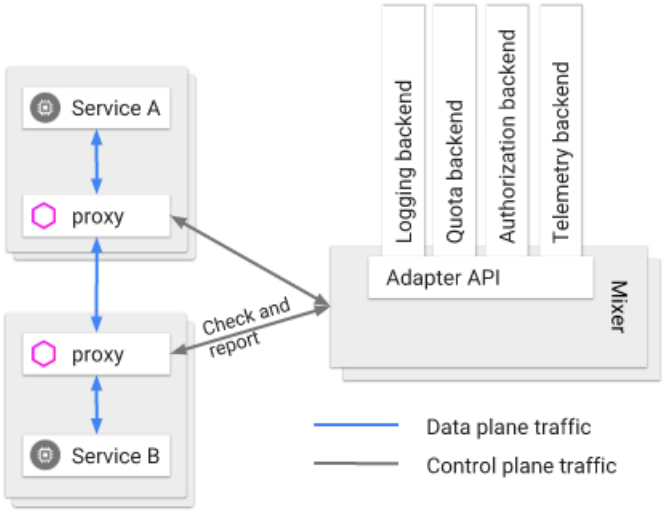
En este ejemplo vemos como todas las comunicaciones tienen implementadas TLS, el elemento Pilot, se comunica con el servicio A, para implementarle las políticas. Citadel, se comunica con ambos servicios, aportándole los certificados, y Mixer, se comunica con el Servicio B, reportándole las auditorias. También vemos como tanto el servicio A como el servicio B, tienen una autenticación local interna, para comunicarse entre ellos, implementada automáticamente con Istio.

4.3.3. Políticas y Telemetría

Istio por otro lado, proporciona un modelo flexible para aplicar políticas de autorización y recopilar telemetría para los servicios en una “mesh”.

La abstracción que proporciona Istio, hace posible que este se interconecte con un conjunto abierto de componentes de infraestructura. El resultado es proporciona de un control rico y profundo al operador, sin imponer ninguna carga a los desarrolladores. El diseño de Istio busca reducir la complejidad del sistema, eliminar la lógica de las políticas del código del servicio y dar control al operador.

El componente que se encarga de proveer de políticas de control y telemetría es Mixer, como vemos en este ejemplo de la web de Istio:



https://istio.io/docs/concepts/policies-and-telemetry/topology-without-cache.svg

El proxy realiza una llamada a Mixer antes de cada solicitud para realizar verificaciones previas, y después de cada solicitud para reportar telemetría. El proxy al tener almacenamiento en caché, puede realizar una gran cantidad de comprobaciones desde esta. Además, el proxy amortigua la telemetría saliente, de modo que llama al mezclador con poca frecuencia.

Pasándonos a un nivel más alto, Mixer proporciona:

- Abstracción del Backend. Mixer aísla al resto de Istio de los detalles de implementación en el Backend, sin tener que entrar en el código de este.

- Intermediación. Mixer permite tener un control entre todas las interacciones de los elementos de la "mesh".

La aplicación de políticas y la recopilación de telemetría se controlan completamente desde la configuración. Estas funciones están deshabilitadas de forma predeterminada en Istio, de forma que Mixer no se ejecuta a no se que se habiliten estas funciones.

4.4.4. Rendimiento y Escalabilidad

Istio facilita la creación de una red de servicios implementados con enrutamiento inteligente, balanceo de carga, autenticación servicio-servicio, monitoreo y más, todo sin realizar ningún cambio en el código de la aplicación. Para realizar todas estas funciones, Istio se esfuerza en tener una sobrecarga de recursos mínima y tiene como objetivo admitir "mesh" muy grandes con altas tasas de solicitud implementando una latencia mínima.

A diferencia de las otras funciones que implementa Istio, en esta entran en juego tanto las correspondientes al plano de datos (proxies); como los correspondientes al plano de control (Pilot, Gallery, Citadel y Mixer).

El control del rendimiento en el plano de control, puesto que este admite miles de servicios, distribuidos en miles pods con un número similar de servicios virtuales, y otros obejetos de configuración; se basan en escalar los requisitos de CPU y memoria de Pilot proporcionalmente al tamaño de estas configuraciones y teniendo en cuenta un posible estado del sistema.

Por otro lado, el rendimiento en el plano de control depende de muchos factores, como el numero de clientes conectados, el tamaño de las peticiones y las respuestas, los protocolos, el número de CPUs, etc. La latencia, rendimiento y consumo de CPU y memoria variaran en función de dichos factores.

5. Cómo Instalar Istio en Kubernetes

La instalación de Istio es sencilla, y aunque se resume en pocos pasos, el tiempo que consume la descarga de paquetes y su instalación es relativamente grande. Veamos los pasos a seguir para instalar Istio, partiendo del entorno que ya hemos explicado un poco antes.

Lo primero que haremos será descargarnos los ficheros de instalación de Istio. Para ello ejecutaremos:

```
curl -L https://git.io/getLatestIstio | sh -
```

Una vez ejecutado se nos descargará un directorio, con el nombre de Istio y la versión, en este caso, nuestro directorio se llamará Istio-1.1.7. Una vez descargado entraremos en la la siguiente ruta:

```
cd istio-1.1.7/install/kubernetes/Helm/istio-init/files
```

Una vez que nos localizamos en esta ruta, deberemos proceder a la instalación de Istio en sí, para ello ejecutamos un kubectl apply por cada uno de los cuatro ficheros yaml que tenemos:

```
kubectl apply -f crd-10.yaml
kubectl apply -f crd-11.yaml
kubectl apply -f crd-certmanager-10.yaml
kubectl apply -f crd-certmanager-11.yaml
```

Tras unos 5-10 minutos (en mi caso), la instalación habrá concluido, y veremos que se ha creado un namespace llamado “istio-system”, para verificarlo, ejecutaremos primero un kubectl get svc -n istio-system:

```
root@nieve:/home/debian# kubectl get pods,svc -n istio-system
NAME                                        READY   STATUS      RESTARTS
AGE
pod/grafana-67c69bb567-zfwsw                1/1     Running     1
8d
pod/istio-citadel-fc966574d-jqktv           1/1     Running     1
8d
pod/istio-cleanup-secrets-1.1.7-njzdj       0/1     Completed   0
8d
pod/istio-egressgateway-6b4cd4d9f-grzf8     1/1     Running     1
8d
pod/istio-galley-cf776876f-bvtt8            1/1     Running     0
19m
pod/istio-grafana-post-install-1.1.7-lbj4f  0/1     Completed   0
8d
pod/istio-ingressgateway-59cc6ccbcb-ht7gv   1/1     Running     1
8d
pod/istio-pilot-7b4dd9b748-rrm62            2/2     Running     2
8d
```

```
pod/istio-policy-5bcc859488-8qvcp               2/2     Running     8
8d
pod/istio-security-post-install-1.1.7-dlkzm    0/1     Completed    0
8d
pod/istio-sidecar-injector-c8ddbb99c-2c7tk     1/1     Running      1
8d
pod/istio-telemetry-7678c9bb4d-zq2x4            2/2     Running     8
8d
pod/istio-tracing-5d8f57c8ff-58p5j              1/1     Running     1
8d
pod/kiali-d4d886dd7-l42bs                       1/1     Running     1
8d
pod/prometheus-d8d46c5b5-d9hg8                  1/1     Running     1
8d

NAME                            TYPE          CLUSTER-IP      EXTERNAL-
IP                                                            PORT(ssS)
AGE
service/grafana                  ClusterIP     10.107.118.255   <none>
3000/TCP
8d
service/istio-citadel           ClusterIP     10.97.168.175    <none>
8060/TCP,15014/TCP
8d
service/istio-egressgateway     ClusterIP     10.99.247.42     <none>
80/TCP,443/TCP,15443/TCP
8d
service/istio-galley            ClusterIP     10.111.60.220    <none>
443/TCP,15014/TCP,9901/TCP
8d
service/istio-ingressgateway           LoadBalancer      10.111.98.26
<pending>
15020:30708/TCP,80:31380/TCP,443:31390/TCP,31400:31400/TCP,15029:326
00/TCP,15030:32762/TCP,15031:32016/TCP,15032:31161/TCP,15443:32587/T
CP   8d
service/istio-pilot             ClusterIP     10.101.247.155   <none>
15010/TCP,15011/TCP,8080/TCP,15014/TCP
8d
service/istio-policy            ClusterIP     10.101.14.107    <none>
9091/TCP,15004/TCP,15014/TCP
8d
service/istio-sidecar-injector   ClusterIP     10.96.82.91     <none>
443/TCP
8d
service/istio-telemetry         ClusterIP     10.108.35.229    <none>
9091/TCP,15004/TCP,15014/TCP,42422/TCP
8d
service/jaeger-agent            ClusterIP     None             <none>
5775/UDP,6831/UDP,6832/UDP
8d
service/jaeger-collector        ClusterIP     10.109.161.100   <none>
14267/TCP,14268/TCP
8d
service/jaeger-query            ClusterIP     10.97.158.93     <none>
16686/TCP
8d
service/kiali                   ClusterIP     10.110.233.124   <none>
20001/TCP
8d
```

```
service/prometheus              ClusterIP      10.110.147.208   <none>
9090/TCP
8d
service/tracing                 ClusterIP      10.111.75.239    <none>
80/TCP
8d
service/zipkin                  ClusterIP      10.96.36.188     <none>
9411/TCP
8d
```

6. Habilitar el uso de Istio

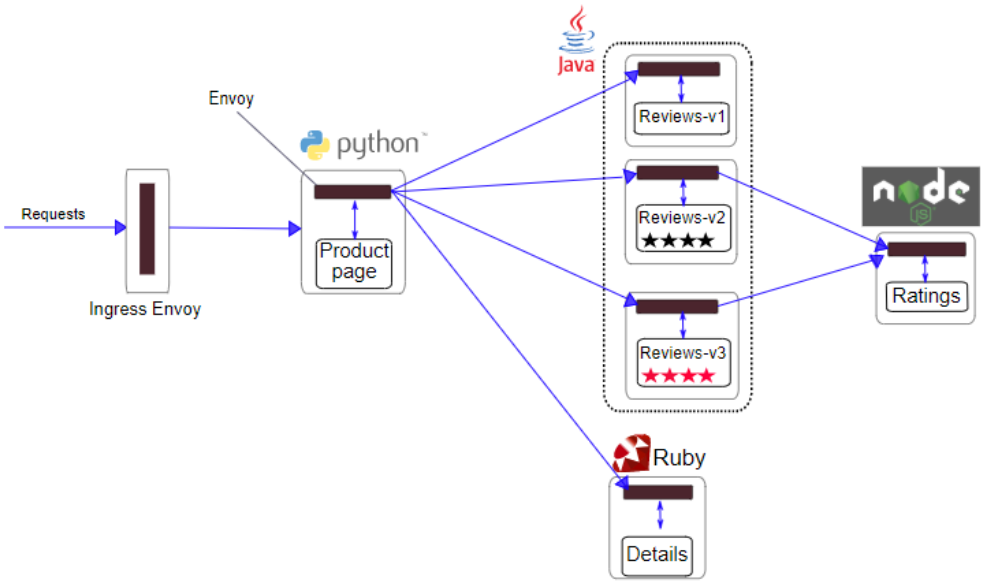
El uso de Istio, se implantará habilitando Istio en un namespace, en mi caso, en el namespace default, que es el que crea kubernetes por defecto, para ello ejecutamos:

```
kubectl label namespace default istio-injectio=enabled
```

Con esto, lo que hacemos es que cada vez que ejecutamos un kubectl apply, pasara primero por Istio, y este hará los cambios necesarios en el fichero yaml, para que Istio funcione en las aplicaciones que vamos a desplegar.

7. Escenario de Pruebas

Cuando nos descargamos todo lo necesario para la instalación de Istio, también se nos descargó una serie de escenarios para realizar pruebas. En concreto, en este caso vamos a usar el escenario llamado Bookinfo.



https://istio.io/docs/examples/bookinfo/withistio.svg

El escenario en cuestión se basa, en un ingress, conectado con una página en producción conectada a su vez con dos servicios, uno dedicado a Ruby, que contendrá los detalles de la “reviews”, y otro dedicado a Java, que mantiene desplegado 3 versiones distintas de un sistema de puntuación en 3 pods distintos, que a su vez tiene dos de sus pods conectados a un servicio de node.js, para ejecutar las reviews programadas en java.

La instalación de este escenario es muy simple. Una vez habilitado la istio-injection en el namespace que vamos a usar, ejecutaremos el siguiente comando:

```
kubectl apply -f istio-
1.1.7/simples/bookinfo/platform/kube/bookinfo.yaml
```

Una vez ejecutado, deberemos crear un nodeport con el en el deployment productpage:

```
kubectl expose deployment productpage-v1 -type=NodePort
```

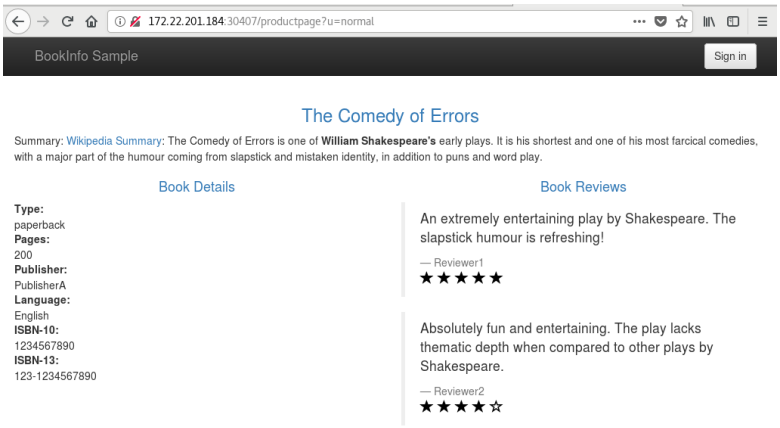
Una vez ejecutado, si hacemos un kubectl get services, vemos que tenemos la redirección activa, y que podremos acceder desde el puerto 30407:

```
root@nieve:/home/debian# kubectl get services productpage-v1
NAME                  TYPE                  CLUSTER-IP
EXTERNAL-IP          PORT(S)           AGE
Productpage-v1    NODEPORT     10.103.249.45    <none>
9080:30407/TCP  5d23h
```

Establecemos una variable, que será la URL de acceso, que será la combinación de cualquier ip del cluster, y el puerto que usaremos para futuras pruebas:

```
export GATEWAY_URL=172.22.201.184:30407
```

Por último, si accedemos en cualquier navegador a la dirección 172.22.201.184:30407, nos mostrara la página funcionando:



8. Pruebas de Funcionamiento

Entre las numerosas pruebas de funcionamiento que se podrían hacer, hemos elegido 3 que pienso que son las más interesantes, fáciles de asimilar y que demuestran el funcionamiento puro de Istio. Estas pruebas van a ser las siguientes:

- Blue Green deployment.
- Canary Release.
- Request Timeouts

8.1. Blue Green Deployment

Un blue green deployment consiste en tener varias versiones de una aplicación desplegada, pero mandar todo el tráfico a través de una sola de ellas. Con Istio podemos implementar esta eficaz técnica de una manera fácil y limpia, sin tener que cambiar el código de las aplicaciones.

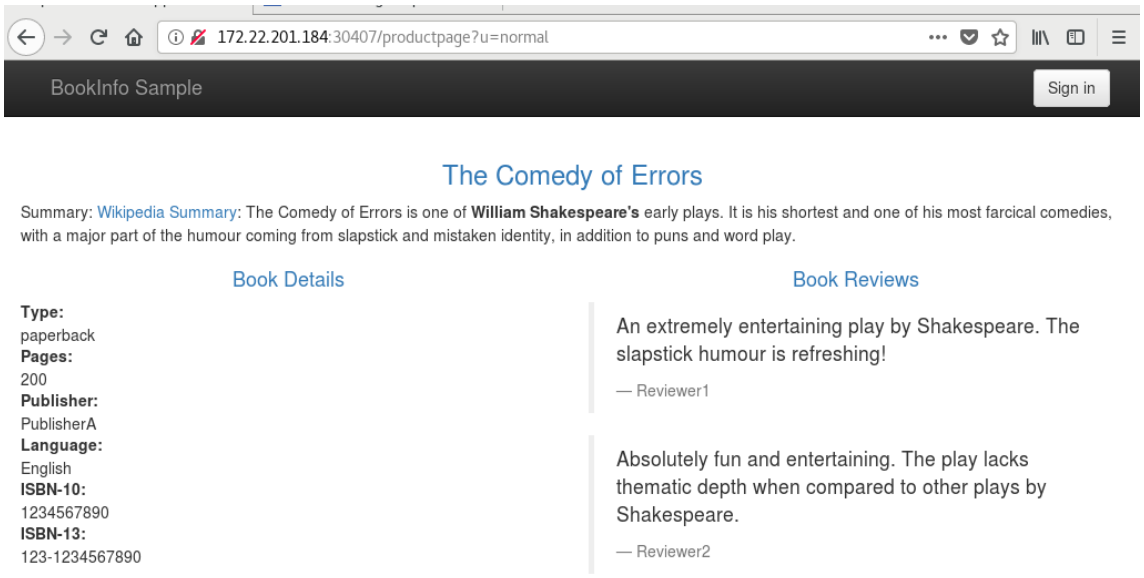
Una vez creado nuestro escenario de pruebas, y como vimos con la explicación de este, tenemos 3 versiones de una aplicación java. En este ejemplo de blue green deployment, haremos que todo el trafico vaya a la versión 1 de esta, para ello crearemos un fichero llamado blue-green.yaml que tendrá el siguiente contenido:

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

Con este sencillo fichero, le estamos diciendo a Istio, que todas las peticiones que lleguen sean mandadas a V1. A continuación, ejecutamos el siguiente comando, y esperamos unos segundos para que la configuración tenga efecto:

```
kubectl apply -f blue-green.yaml
```

Ahora si entramos en la página, veremos que no nos aparece ninguna estrella, ya que la v1, no tiene implementada el dibujo de las estrellas:



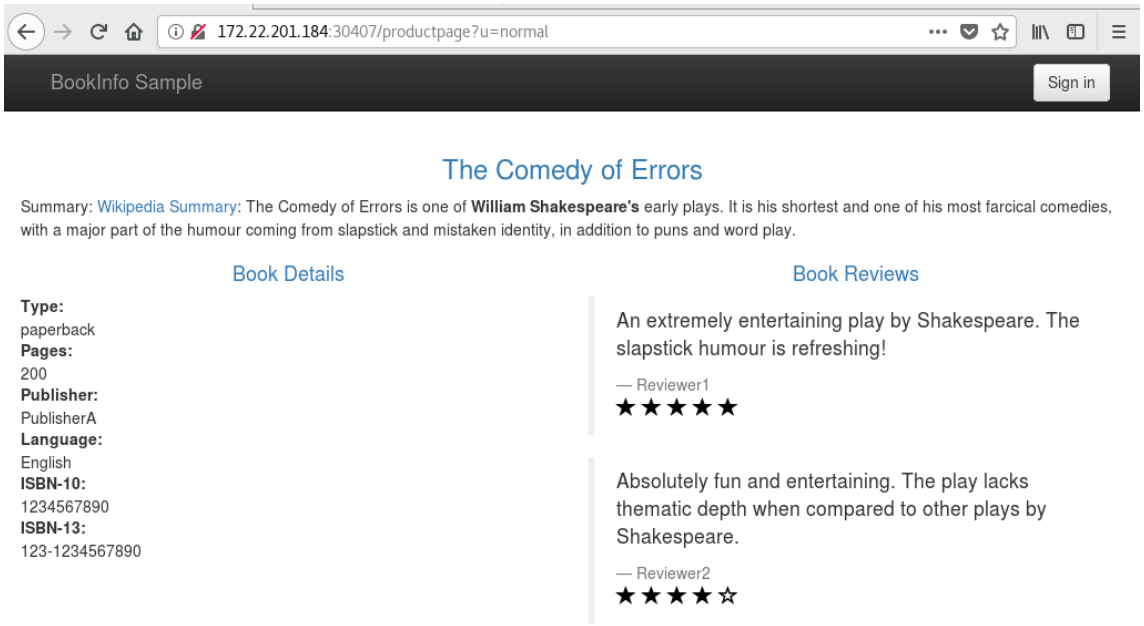
Con este sencillo paso, hemos redireccionado todo el tráfico a una única versión. Ahora imaginemos que hemos detectado un fallo en la versión V1, y debemos redireccionar el trafico a la versión V2. Para ello, solo deberemos hacer un pequeño cambio en el fichero blue-green.yaml, dejándolo así:

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

Aplicamos los cambios:

```
kubectl apply -f blue-green.yaml
```

Y si comprobamos, veremos que ahora nos aparecen estrellas negras cada vez que recargamos la página, habiéndose completado la configuración:



Si deseamos deshacer los cambios, y que siga con el Round-robin propio de kubernetes bastara con ejecutar un:

```
kubectl delete -f blue-gren.yaml
```

8.2. Canary release

Un canary release se basa tener una versión desplegada funcional, e introducir otra a la que le iremos incrementando el tráfico poco a poco, para ver como funciona esta a medida que se le va aumentando el número de peticiones.

Por ejemplo, tenemos una versión desplegada, a la que le llegan el 100% de las peticiones, pero hemos desarrollado otra, que queremos ir implementando poco a poco, asi que tendremos dos versiones, la primera V1 que ya no le llegara el 100% de las peticiones sino el 90% y una V2, que hemos implementado nueva, y le llegaran el 10% de las peticiones.

Con Istio, esta configuración, se implementaría creando un fichero llamado en esta caso, canary-release.yaml, con el siguiente contenido:

```
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
    - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v1
      weight: 90
    - destination:
        host: reviews
        subset: v2
      weight: 10
```

Con esta configuración, hemos establecido que el 90% vaya a v1 y el 10% a v2. Una vez creado el fichero, lo aplicaremos con un kubectl:

```
kubect apply -f canary-release.yaml
```

Una vez ejecutado, y al haber esperado un tiempo prudencial para que estos cambios hayan tomado efecto veremos que los porcentajes de las peticiones son los que hemos establecido.

La idea de este tipo de despliegues seria llegara hasta un punto medio de peso de solicitudes, es decir 50% para V1 y 50% para V2, si vemos que con el 10% no ha habido problemas de uso, podremos subir poco a poco hasta llegar a esta situación.

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
    - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v1
      weight: 50
    - destination:
        host: reviews
        subset: v3
      weight: 50
```

Con un 50-50, las peticiones se repartirán de manera equitativa, y si V2 sigue funcionando sin problemas, se podrá llegar al punto final de esta casuística, que el 100% de las peticiones sean para V2, dejando algo así:

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

Como vemos, al no tener que mandarle tráfico a V1, podemos descartar este del fichero de configuración, llegando el 100% de peticiones a la nueva versión, y quedando algo parecido en funcionalidad al blue green deployment que vimos antes.

Por último, si queremos eliminar esta configuración, y volver la configuración por defecto, bastara con hacer un:

```
kubectl delete -f canary-release.yaml
```

8.3. Requests Timeouts

Por último, mostraremos como implementar requests timeouts, para demostrar esto haremos que entre la petición y la respuesta haya un tiempo de espera, para ello, primero haremos un blue green deployment para que todas las peticiones vayan a v1 exclusivamente, para ello, volveremos a ejecutar el fichero blue-green.yaml.

Ahora lo que haremos será establecer que use V2 de reviews:

```
kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
    - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v2
EOF
```

Y estableceremos un tiempo de espera de 2 segundos a ratings:

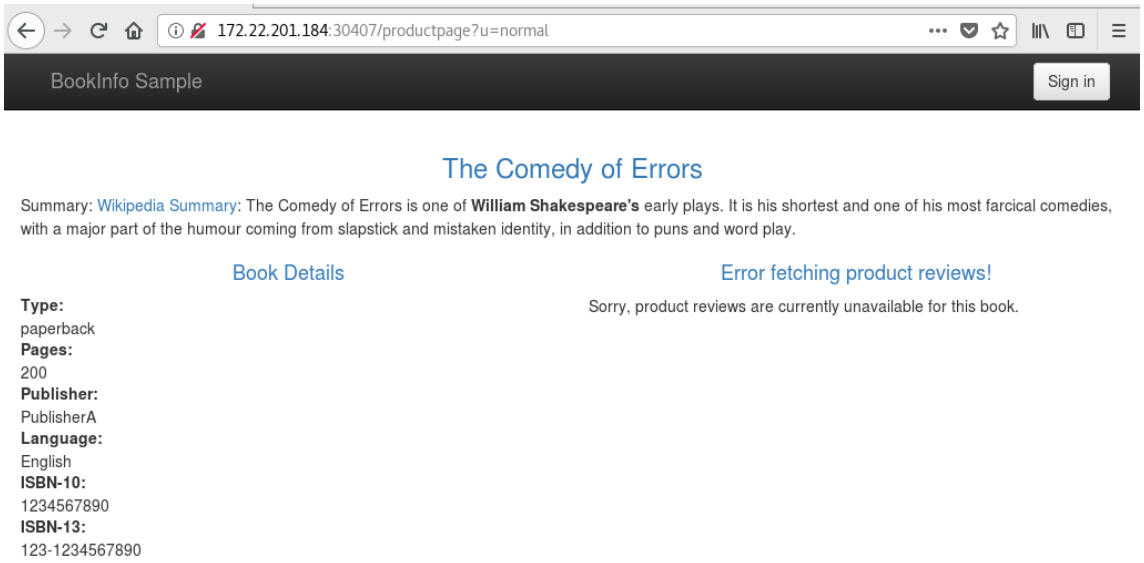
```
kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: ratings
spec:
  hosts:
  - ratings
  http:
  - fault:
      delay:
        percent: 100
        fixedDelay: 2s
    route:
    - destination:
        host: ratings
        subset: v1
EOF
```

Ahora si abrimos el navegador, y entramos en nuestra página, veremos que tardara 2 segundos en cargarse.

Ahora agregaremos un timeout a reviews:

```
kubectl apply -f - <<EOF
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - route:
    - destination:
        host: reviews
        subset: v2
    timeout: 0.5s
EOF
```

Ahora cuando actualicemos la página, veremos que además de cargarse en medio segundo, este nos muestra un mensaje de error, diciéndonos que ha habido un error:



Esto se debe a que le hemos puesto un timeout de 0’5 segundos a “reviews”, pero cuando “reviews” le hace una llamada a “ratings”, esta tiene un tiempo de espera de 2 segundos, por lo que al no recibir respuesta en 0’5 segundos, “reviews” corta la conexión, y nos aparece este error, con el que demostramos el funcionamiento de un time out en Istio.

9. Conclusiones

Tras haber visto el comportamiento de Istio, hemos demostrado que es una herramienta muy potente a la hora de gestionar las comunicaciones entre servicios dentro de una red, y a la que el propio Kubernetes a denominado “el futuro de este”.

Hemos conseguido ver como Istio es capaz de gestionar la comunicación entre los servicios de una manera “fácil” y limpia, sin tener que modificar el código de estás, pero como desventajas hemos encontrado que, al interponer un punto medio entre servicio y servicio, en este caso, los Envoy, existe una pequeña relentización, prácticamente nula, en la comunicación entre los servicios.

Como conclusión final, pienso que Istio es una herramienta muy potente, que, si se utiliza en las situaciones en las que se le puede sacar partido, es el claro candidato a su uso.

9.1 Trabajos Futuros

Al no haber tenido un tiempo ilimitado para la realización de este trabajo, no he podido probar el 100% de las funcionalidades de Istio, pero seria interesante en trabajos futuros, tanto la familiarización que hace Istio a la hora de establecer TLS para cifrar las comunicaciones internas; como algo que veo mucho más interesante, que seria montar un sistema de monitorización de los servicios, puesto que Istio, por defecto, viene implementado con “Prometheus”.

10. Bibliografía

Explicación de una Service Mesh

https://searchitoperations.techtarget.com/definition/service-mesh

Explicación de Istio

https://medium.com/@serrodcal/istio-conecta-gestiona-y-securiza-microservicios-89d332e5a681

Curso de Istio de OpenWebinars

https://openwebinars.net/academia/aprende/istio/

Creación de Cluster de Kubernetes

https://www.josedomingo.org/pledin/2018/05/instalacion-de-kubernetes-con-kubeadm/