Politecnico di Torino

FACOLTÀ DI INGEGNERIA
Corso di Laurea Magistrale in Ingegneria Informatica percorso Reti

Tesi di Laurea Magistrale in
Ingegneria Informatica

Connect, secure, manage and monitor a microservices application by using Istio

Relatori

Prof. Riccardo Sisto
Prof. Fulvio Risso

Candidato

Giuseppe Sardone

Tutor Aziendale, manager di Blue Reply srl

Dott. Guido Vicino

Anno Accademico 2018/2019

Abstract

I microservizi ormai stanno prendendo piede all'interno del mondo cloud e tutte le architetture vengono rinnovate cambiando il loro pattern applicativo. Si passa, quindi, da una grande applicazione monoblocco ad una più granulare composta da diversi microblocchi. Connettere, assicurare, gestire e monitorare un'applicazione divisa è, quindi, il nuovo compito dei team di DevOps per garantire il corretto funzionamento di ogni microblocco. Docker e Kubernetes permettono, grazie alle loro funzionalità, di creare un cloud di host che gestisce diversi microservizi. Una volta, però, che i microservizi vengono distribuiti all'interno del cloud, vi è il bisogno di gestire e monitorare le comunicazioni dei microservizi stessi, nonché proteggerle da attacchi esterni. Istio, una piattaforma open-source, offre meccanismi per instradamento, monitoraggio e assicurazione del traffico di rete, permettendo una configurazione personalizzata della rete di microservizi. Obiettivo di questo studio è quindi mostrare una serie di possibili configurazioni, utilizzabili su una semplice applicazione a microservizi, per garantire una comunicazione sicura per ogni microservizio, controllare e autorizzare gli accessi, monitorare lo stato e la configurazione della rete. Lo studio dunque è stato sviluppato agendo soprattutto su due livelli, rete e sicurezza, monitorando sempre lo stato dei servizi ad ogni cambiamento. In particolare, si sono configurate una serie di regole di routing per il traffico di rete http, crittografandolo all'interno di un canale TLS e verificando l'autenticazione e l'autorizzazione di tutte le chiamate. I risultati ottenuti mostrano che attraverso l'utilizzo della piattaforma Istio si può avere una rete configurata, a seconda delle volontà dell'utente, utilizzando semplici file di configurazione senza implementare pesanti modifiche a livello di codice applicativo a fronte di un utilizzo aggiuntivo sulle risorse fisiche del cluster.

Indice

Capitolo 1

Introduzione

I nuovi mondi *mobile* e *cloud* sono le leve che stanno spingendo verso una trasformazione dalle attuali applicazioni aziendali legacy rispetto alle applicazioni per componenti, ossia i microservizi. Lo scenario dei sistemi informativi in azienda negli ultimi anni è stato travolto da una serie di cambiamenti che hanno impattato sia utenti che fornitori di soluzioni e servizi. Aspetti come la migrazione verso servizi cloud e la sempre più alta diffusione di smartphone e dispositivi IoT hanno costretto a ridefinire rispettivamente i processi e le modalità operative, variando anche le dinamiche di offerta. La sfida adesso è quindi progettare architetture estremamente flessibili e implementabili anche su realtà piccole e medie, che trovano un limite invalicabile nelle attuali tecnologie e nei costi necessari a sostenerle.[1]

1.1 Cosa sono i microservizi

I microservizi rappresentano un'architettura ma soprattutto un approccio alla scrittura di software. Tramite i microservizi, le applicazioni vengono scomposte nei loro elementi più piccoli, indipendenti l'uno dall'altro. Rispetto allo stile monolitico tradizionale, per cui ogni componente viene generato all'interno di un unico elemento, i microservizi interagiscono per portare a termine le stesse attività.

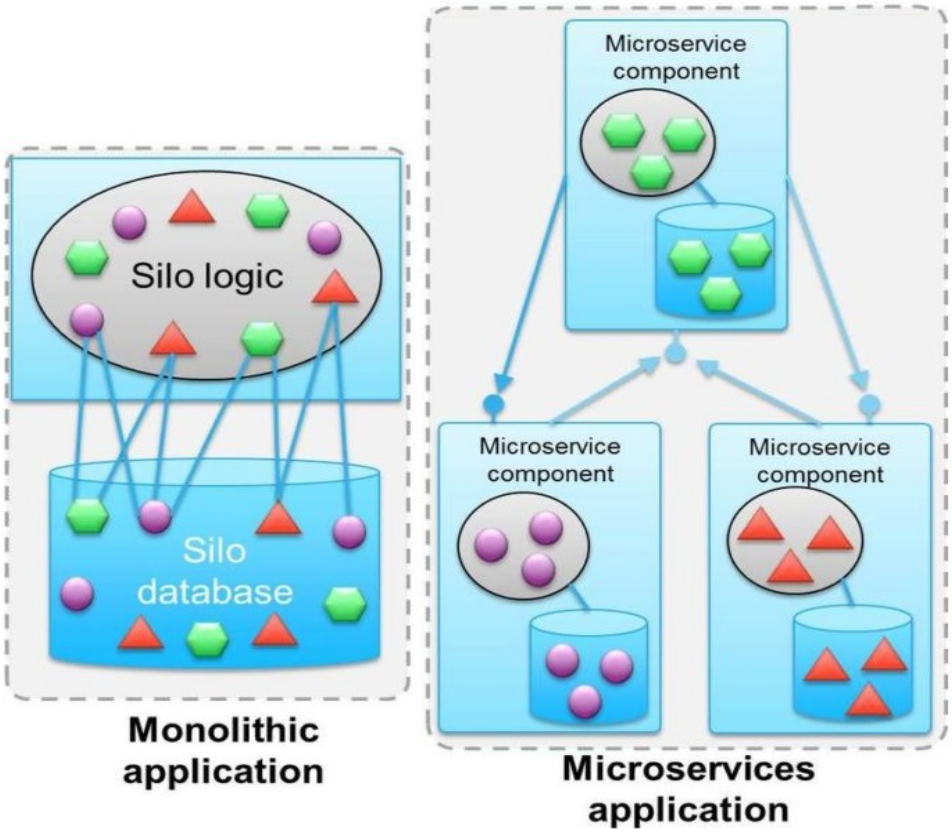


Figura 1.1: *Architettura applicazione monolitica e a microservizi.* [1]

Ciascun componente, o processo, rappresenta un microservizio. Questo tipo di approccio allo sviluppo del software, essendo molto leggero, promuove la granularità e consente di condividere processi simili tra più applicazioni. L'orientamento ai microservizi è uno dei principali fattori che, ottimizzando lo sviluppo applicativo, si avvicina ad un modello cloud-native.[2]

1.2 Architettura monolitica vs Architettura di microservizi

Nelle architetture monolitiche tutti i processi sono strettamente collegati tra loro e vengono eseguiti sequenzialmente come un unico servizio. Ciò significa che, qualora si volesse limitare un processo dell'applicazione che esegue un numero elevato di richieste, sarebbe necessario ridimensionarla interamente. Aggiungere, eliminare o migliorare una singola funzione dell'applicazione monolitica risulta più complesso in quanto richiede un riadattamento di tutto il progetto. Tale complessità limita la sperimentazione, rendendo più difficile l'implementazione di nuove idee. Questo tipo di sviluppo rappresenta un ulteriore rischio per la disponibilità dell'applicazione, in quanto la presenza di numerosi processi dipendenti

e strettamente connessi tra loro aumenta l'impatto di un errore in un singolo processo. Grazie ad un'architettura incentrata su microservizi, un'applicazione è formata da componenti indipendenti che eseguono ogni processo applicativo come un servizio. I servizi sono realizzati in modo da svolgere una sola funzione, questi ultimi non devono condividere alcun codice o risorse con gli altri. Qualsiasi comunicazione tra i componenti individuali avviene attraverso API leggere ben definite. Poiché eseguiti in modo indipendente, ciascun servizio può essere monitorato, aggiornato, distribuito e ridimensionato per rispondere alla richiesta di funzioni specifiche di un'applicazione.[3]

1.3 Vantaggi dei microservizi

La divisione in componenti isolati sicuramente rende un ambiente predisposto al build e alla gestione di applicazioni altamente scalabili. I servizi sviluppati e distribuiti in modo indipendente portano a funzionalità più agili rispondendo facilmente ai cambiamenti ambientali odierni. In linea generale i microservizi offrono:

- Eliminazione di "*single point of failure*": un'applicazione scomposta in microservizi riduce notevolmente il rischio che un errore di codice o configurazione si riversi sull'intero sistema. Eventuali servizi instabili possono essere gestiti singolarmente, corretti e distribuiti nuovamente senza necessariamente interrompere il flusso dell'intera applicazione.

- Amministrazione più elastica: lo sviluppo dei processi (build, test, deploy, update) può essere gestito in modo agile e semplice avendo microservizi. Gli ambienti di sviluppo, test e produzione rimangono quindi più coerenti e allineati tra loro.

- Scalabilità efficace: la scalabilità è definita a livello di servizio. In base alle esigenze del sistema, ogni componente può essere replicato e distribuito sull'hardware che è più adatto in termini di risorse.

- Riusabilità: Durante un aggiornamento o una migrazione di un'applicazione alcuni microservizi che non devono essere modificati possono rimanere intatti per essere utilizzati dalla nuova versione e da applicazioni diverse.

- Versionamento: le API possono seguire un flusso diverso rispetto a quello dei servizi. Major e minor release vengono proposte a livello di applicazione, mentre i servizi richiedono aggiornamenti solo su richiesta.

- Libertà del linguaggio di sviluppo: il pattern dei microservizi elimina ogni vincolo preso a lungo termine sulla scelta del linguaggio. Quando si implementa un nuovo servizio, gli sviluppatori sono liberi di scegliere il linguaggio di programmazione e framework corrispondente più idonei al nuovo servizio. Inoltre, poiché i servizi sono di ridotte dimensioni, intervenire usando linguaggi e tecnologie migliori risulta molto agevole.

Bisogna ricordare, però, che l'aspetto fondamentale di un'applicazione a microservizi deve risultare, all'utente finale, una singola applicazione. Tra i diversi servizi è necessaria una forte coesione, quindi un mantenimento ed una distribuzione paralleli preservando la user experience finale.[4]

1.4 Container

Grazie all'utilizzo dei container le applicazioni possono essere astratte dall'ambiente in cui sono eseguite e impacchettate, per poi essere gestite in modo indipendente. Il disaccoppiamento tra implementazione e ambiente consente di eseguire facilmente il deployment delle applicazioni containerizzate, indipendentemente dall'ambiente di destinazione, che sia un data center privato, il cloud pubblico o il laptop di uno sviluppatore. Tramite l'approccio ai container, gli sviluppatori si occupano della logica e delle dipendenze dell'applicazione, mentre i team operativi DevOps possono concentrarsi sul deployment e sulla gestione, senza preoccuparsi di dettagli applicativi come linguaggio, framework o configurazioni dell'applicazione.[5]

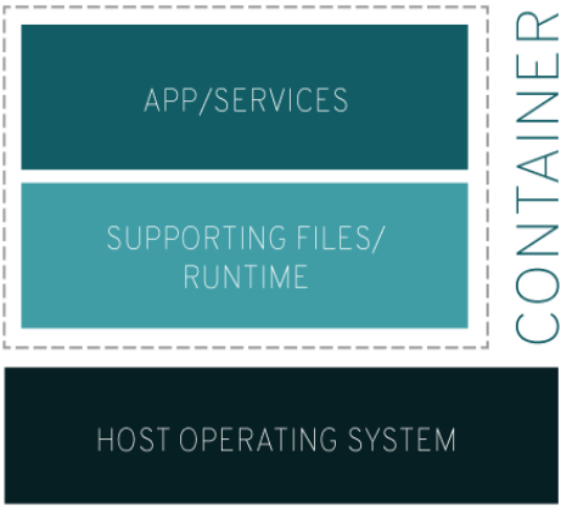


Figura 1.2: *Container generico.* [6]

1.5 Container e Macchine Virtuali

L'isolamento delle applicazioni in rete può essere riprodotto sia utilizzando i containers che le virtual machines, lavorando a differenti livelli di virtualizzazione. Le VMs possiedono ognuna un sistema operativo "guest" diverso mentre i container lo virtualizzano, ed è per questo che i containers sono maggiormente portabili ed efficienti.

1.5.1 Macchine Virtuali

Le Virtual Machines permettono di avere più server logici all'interno dello stesso server fisico. Grazie all'*hypervisor*, infatti, è possibile eseguire diverse macchine virtuali sullo stesso hardware.

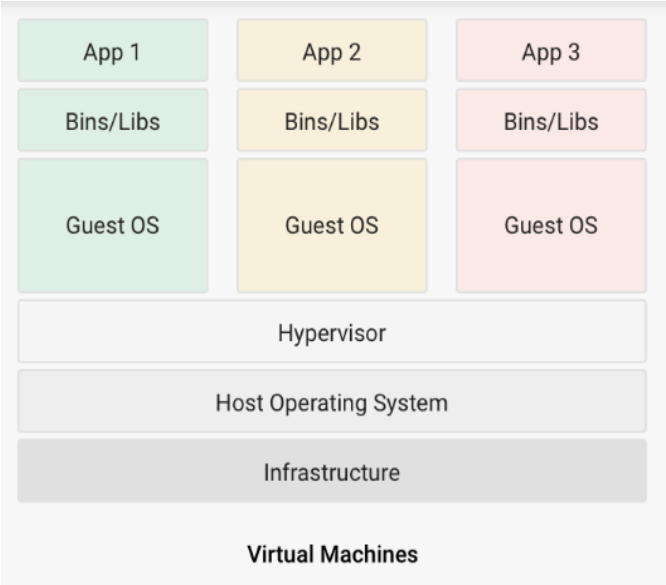


Figura 1.3: *Schema virtualizzazione tramite VMs.* [5]

L'aspetto sfavorevole di questo tipo di virtualizzazione è che, per ogni applicazione, è presente una copia esatta del sistema operativo offerto oltre alle librerie e i file binari per risolvere le dipendenze. Per questa ragione le macchine virtuali possono essere di grandi dimensioni, anche decine di GBs, ritardando il processo di avvio.[7]

1.5.2 Container

L'astrazione per mezzo dei containers viene effettuata allo strato applicativo impacchettando insieme sia il codice che le sue dipendenze. Inoltre svariati containers possono girare sullo stesso hardware condividendo il kernel del sistema operativo, per cui ogni container sarà un processo isolato a livello utente.

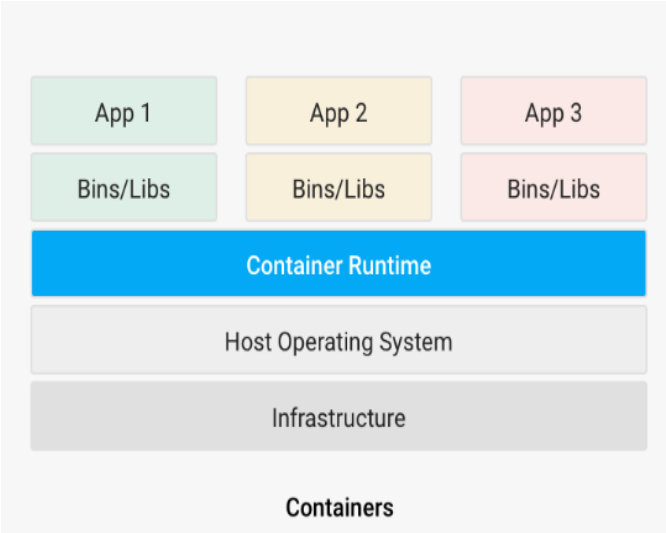


Figura 1.4: *Schema virtualizzazione tramite containers.* [5]

Diversamente dalle virtual machines, i container occupano molto meno spazio, comunemente pari a qualche centinaia di MBs, riducendo il tempo di avvio.[7]

1.6 La scelta dei Containers

Il concetto di microservizio si associa molto bene a quello di container: entrambi sono componenti indipendenti e stateless. Ogni microservizio, infatti, viene incapsulato in un container, creando così un ambiente virtualizzato con l'opportunità di scalare orizzontalmente il microservizio solo su richiesta, sfruttando al meglio le risorse. Naturalmente, con un'architettura a microservizi che cerca di risolvere i problemi di scalabilità, isolamento e portabilità, si hanno una serie di aspetti da valutare data la complessa configurazione degli ambienti distribuiti. Disponibilità, comunicazione, monitoraggio e orchestrazione dei servizi divengono fattori molto importanti da considerare, ma ci sono nuove tecnologie che aiutano alla conservazione di uno stato coerente dell'ambiente.[4]

1.7 Scopo del progetto

Il problema principale all'interno del mondo cloud è quello di monitorare e gestire tutte le comunicazioni interne dei servizi. Sicurezza, monitoraggio, affidabilità, gestione e bilanciamento del traffico sono gli ambiti principali che si vogliono configurare all'interno di una rete di servizi.

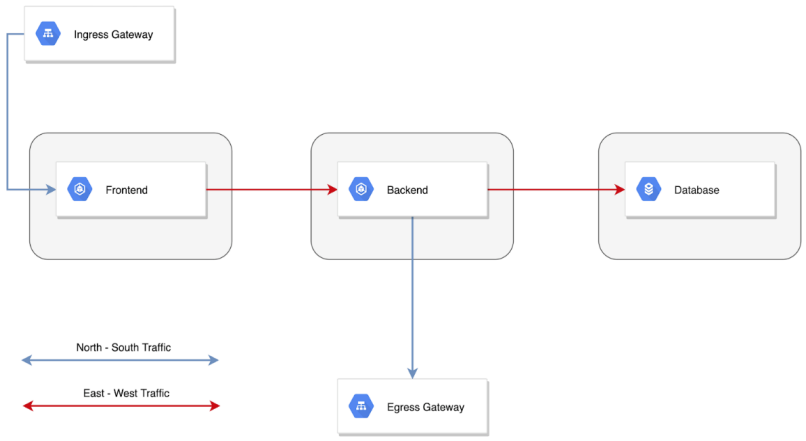


Figura 1.5: *Schema rete di servizi nel cloud.* [8]

Il progetto di tesi propone, quindi, una soluzione per questo tipo di problema, andando gradualmente a configurare i vari criteri all'interno della rete gestendo, assicurando e monitorando tutti gli stati e il traffico generato. In prima battuta, dunque, si andranno ad installare in un cluster dei microservizi ottenuti da una semplice applicazione monolitica, utilizzando tecniche per creare container per mezzo di Docker. Si è proceduto, successivamente, a distribuire i microservizi in una piattaforma di controllo ed orchestrazione di container, Kubernetes. Una volta inserita un'applicazione a microservizi nel mondo cloud computing, lo studio è proseguito con il vero e proprio scopo finale, ovvero quello di gestire e controllare la rete di servizi e il suo stato. Tramite Istio, che figura tra le tecnologie avanzate più appropriate per lo scopo, sono stati implementati all'interno del cloud vari criteri per la gestione ordinata e personalizzata del traffico di rete, molteplici policy di sicurezza per abilitare autenticazione, confidenzialità ed autorizzazione per le comunicazioni tra servizi. Infine sono stati analizzati lo stato della rete di servizi, il suo monitoraggio per mezzo di dati di telemetria generati ad ogni richiesta e le prestazioni che Istio introduce all'interno del cluster.

1.8 Tecnologie utilizzate

Questo studio è stato svolto in collaborazione con la società Blue Reply, una delle principali technology consulting companies del gruppo Reply, nonché partner di IBM, Dynatrace, PingIdentity, Splunk e Axway. Tale società focalizza la sua attenzione su diversi campi quali Finance (Banking, Credit e Insurance), Manufacturing, Telco & Media, Artificial Intelligence, Internet Of Things, Cybersecurity, Big Data, eCommerce e Cloud Computing, ambito in cui la tesi è stata sviluppata. Grazie ad un cluster offerto dall'azienda, è stato possibile utilizzare principalmente strumenti come Docker e Kubernetes per la gestione e l'orchestrazione dei container all'interno di un'architettura cloud.

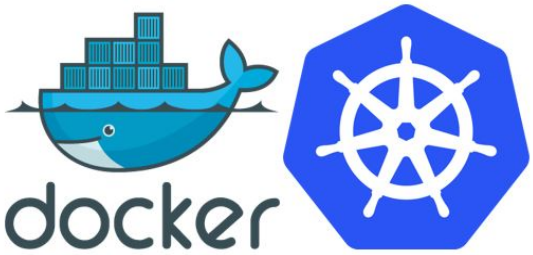


Figura 1.6: *Simboli piattaforme Docker e Kubernetes.* [9]

Per la storicizzazione di configurazioni e implementazione applicativa, invece, sono state adoperate tecniche e piattaforme per il versionamento del codice utilizzando Github e Gitlab, seguendo quindi il tipico path dei team di sviluppo e di DevOps. Infine, per lo sviluppo vero e proprio del codice applicativo si è usufruito di due software, SpringBoot e Maven, per la realizzazione di web server.

Figura 1.7: *Simboli software Springboot, Maven e Gitlab.*

1.9 Progetti esistenti

Esistono diverse tecnologie per risolvere il problema della rete dei servizi, tra queste figurano Consul e Linkerd, ma ognuna di esse ha delle carenze che, in base alle esigenze architetturali, possono essere più o meno rilevanti.

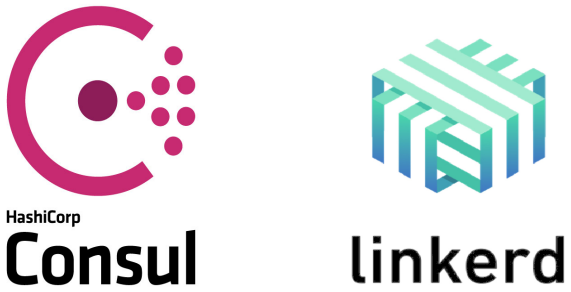


Figura 1.8: *Infrastrutture Consul e Linkerd.* [10]

Consul è un framework per la gestione della rete di servizi e fornisce la funzionalità per il rilevamento automatico dei servizi. Consul è un prodotto dell'infrastruttura HashiCorp partito per gestire servizi in esecuzione su Nomad. Al giorno d'oggi, tuttavia, esso supporta altre piattaforme di gestione dei container tra cui Kubernetes. Consul utilizza un daemon agent, installato su ogni nodo della rete, che comunica con i proxy sidecar allegati ai servizi, gli Envoy, che gestiscono il routing e l'inoltro del traffico. Linkerd, invece, si presta più per la semplicità che per la flessibilità. Tale software, infatti, supporta solo la piattaforma Kubernetes, diminuendo così la complessità ed evitando un ambiente cluster eterogeneo. [10]

	Linkerd	Consul
Workload supportati	Kubernetes	Kubernetes + VM
Single Point of Failure	No	No, ma con complessità nel HA aggiuntiva per le operazioni del server Consul
Sidecar	Si	Si
Node Agent	No	Si
Sicurezza	AuthN, AuthZ, mTLS	AuthN, AuthZ, mTLS
Protocolli di comunicazione	TCP, HTTP/1, HTTP/2, gRPC	TCP, HTTP/1, HTTP/2, gRPC
Distribuzioni A/B	Si	Si
Circuit Breaker	No	Si
Fault Injection	Si	Si
Limitare Rate	No	Si
Test per resilienza	Limitato	No
Monitoraggio	Si, con Prometheus	Si, con Prometheus
Multicluster	No	Si

Figura 1.9: *Confronto tecnologie Linkerd e Consul.* [10]

Istio si pone, come queste tecnologie, l'obiettivo di risolvere i problemi legati alla rete di servizi che si distribuisce in un cluster, coprendo tutti gli ambiti e colmando le lacune che altre tecnologie presentano.

Capitolo 2

Docker

Docker è la piattaforma che permette lo sviluppo di applicazioni basate su container Linux. Lo strato "Linux containers" crea un'ulteriore livello di astrazione tra l'applicativo e il sistema operativo. La virtualizzazione dei sistemi operativi è eseguita in modo da ottimizzare l'utilizzo delle risorse, così facendo l'hardware condiviso resta libero la maggior parte del tempo tranne per alcuni momenti dovuti all'utilizzo di applicazioni.

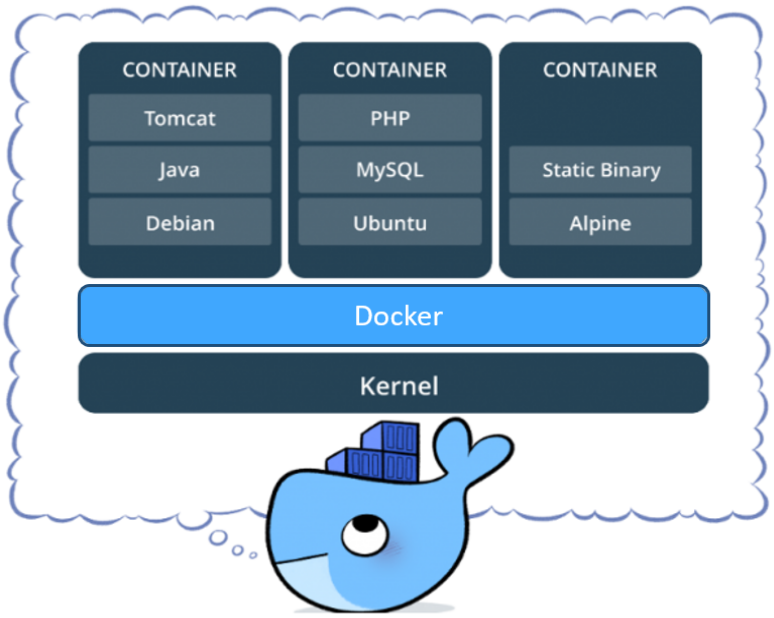


Figura 2.1: *Docker container.* [11]

Docker container risolve il problema delle copie complete dei sistemi operativi, che risultavano dispendiose nell'avvio di un'applicazione e per lo più inutilizzate visto che, per una corretta esecuzione, solo poche funzionalità del sistema operativo virtualizzato sono necessarie. Tramite appositi comandi, con i container è possibile specificare il sistema operativo e quali funzioni virtualizzare, riducendo

le dimensioni dell'immagine finale. L'immagine del container risulterà così leggera, indipendente dal kernel fisico in cui viene eseguita e sicura. Il container in esecuzione sarà infatti isolato, pertanto un errore presente in un'immagine non graverà su gli altri container presenti.

2.1 Architettura Docker

L'architettura Docker è basata sul pattern client-server. La comunicazione tra il client e il server, che svolge tutto il lavoro di build, run e deploy, avviene tramite delle api REST. Il client e il server possono trovarsi sia sullo stesso host che su diversi host, in quest'ultimo caso il client interagisce con un server remoto.

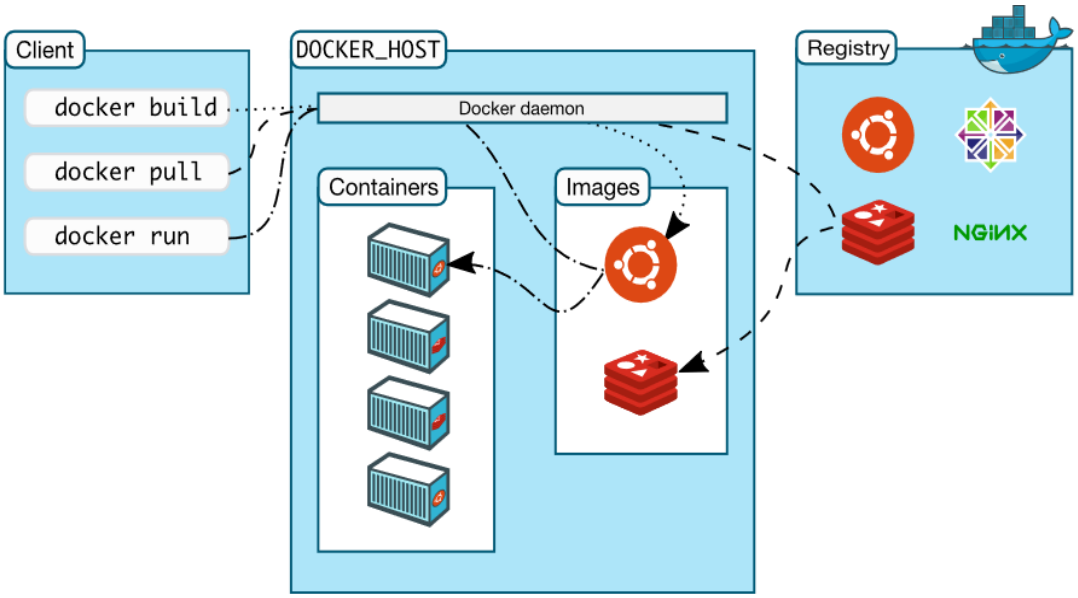


Figura 2.2: *Architettura Docker.* [12]

Fondamentalmente il framework Docker Engine è composto da un server che si occupa della gestione degli oggetti, un client e registri privati o remoti.

- **Docker Daemon**: il componente principale della piattaforma Docker. È il server interno predisposto a girare a lungo termine. Il docker daemon, tramite delle api REST, è in grado di comunicare con gli utenti docker e altri docker daemon remoti. Infine è proprio questo processo che coordina gli oggetti docker come immagini, container, volumi o servizi.
- **Docker Client**: si tratta di un'interfaccia che offre una serie di comandi per interagire con uno o più docker daemon.

- **Docker Registry**: uno o più repository da utilizzare nella gestione delle immagini. Il registro predefinito è DockerHub, pubblico, ma si possono creare anche registri privati su cui fare pull o push delle immagini.

2.2 Docker Images

Le immagini docker sono dei template funzionali per la creazione dei container. Ad esempio, un'immagine può essere creata partendo da un sistema operativo, aggiungendo solo alcune componenti come un server web o un applicazione java scritta dall'utente. Per costruire un'immagine personalizzata bisogna definire un Dockerfile, ovvero un file con una sequenza di comandi che definiscono l'immagine finale. In seguito, i comandi principali usati:

- *FROM*: la prima istruzione presente in un dockerfile perché definisce l'immagine di partenza da cui costruire l'immagine finale.
- *WORKDIR*: crea o imposta la directory specificata come direcotry di partenza per i successivi comandi nel dockerfile.
- *ARG*: utile quando si vuole dichiarare una variabile da utilizzare all'interno di altri comandi.
- *VOLUME*: associa una directory all'interno del container con una all'esterno, nell'host o in un altro container.
- *RUN*: esegue un comando all'interno del processo di creazione dell'immagine. Importante quando si vogliono installare determinati package da un sistema operativo in un passaggio intermedio.
- *COPY*: permette di copiare files da una directory locale ad una all'interno del container.
- *ENV*: simile al comando ARG, imposta variabili d'ambiente da utilizzare però all'interno del container.
- *CMD*: istruzione presente una sola volta nel dockerfile, descrive il comando che il container eseguirà subito dopo essere stato creato.
- *ENTRYPOINT*: come CMD definisce il comando da eseguire alla creazione del container ma, a differenza di quest'ultimo, non può essere sovrascritto in un comando di CLI (docker run).
- *EXPOSE*: rende raggiungibili dall'esterno le porte specificate.

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ARG DEPENDENCY=target/dependency
COPY ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY ${DEPENDENCY}/META-INF /app/META-INF
COPY ${DEPENDENCY}/BOOT-INF/classes /app
ENTRYPOINT ["java","-cp","app:app/lib/*","hello.Application"]
```

Figura 2.3: *Dockerfile per immagine docker.*

Il dockerfile in figura crea un'immagine a partire da jdk-alpine e, dopo aver copiato le dipendenze, setta il comando per eseguire l'applicazione java "hello.Application".

2.3 Docker Container

Un container è un'istanza che esegue un'immagine. Tramite api o, più comunemente, utilizzando i comandi della CLI è possibile creare, spostare, avviare, stoppare o cancellare un container. Da un container in esecuzione, installando o aggiornando alcuni package, è possibile salvare una nuova immagine. Ogni nuovo stato del container, se non salvato, all'atto della rimozione sarà perso. Nella rete docker ogni container è ben isolato dagli altri, ma è possibile modificare e controllare quanto e come tali container devono essere isolati tra loro.

Esempio comando *docker run*

```
docker run -i -t ubuntu /bin/bash
```

Comando che esegue il comando /bin/bash partendo da un'immagine base di Ubuntu. Nel dettaglio:

1. Nel caso in cui non si avesse l'immagine di Ubuntu localmente, docker esegue in automatico il pull della stessa dal registro configurato.
2. Creazione di un nuovo docker container.
3. Allocazione di un file system read-write come strato finale del container, consentendo la creazione e modifica di file e directory.
4. Inizializzazione di un'interfaccia di rete con indirizzo ip dedicato. Di default, i container possono comunicare con l'esterno utilizzando la rete dell'host in cui girano.

5. Avvio del container ed esecuzione del comando /bin/bash. Grazie all'opzione -i e -t (interattivo e terminale) è possibile utilizzare la tastiera per l'input ed il terminale per l'output.

6. All'uscita del comando /bin/bash, il container verrà stoppato in attesa di un riavvio o di una rimozione.

2.4 Docker Swarm

Una volta che i container vengono creati e avviati, il docker engine mette a disposizione un componente per l'orchestrazione e la gestione del cluster, docker Swarm. Uno "*swarm*" consiste di una serie di host che eseguono docker in modalità swarm. Ogni host può agire da swarm manager o worker, ma anche coprire entrambi i ruoli. Tutti gli host all'interno dello swarm concorrono a conservare uno stato coerente del cluster. Ad esempio, se un nodo worker non è più disponibile, lo swarm manager demanderà il task ad un altro nodo worker disponibile. Docker swarm permette anche la configurazione al volo dei servizi, così facendo un servizio verrà stoppato, aggiornato e ricreato automaticamente con la nuova configurazione.

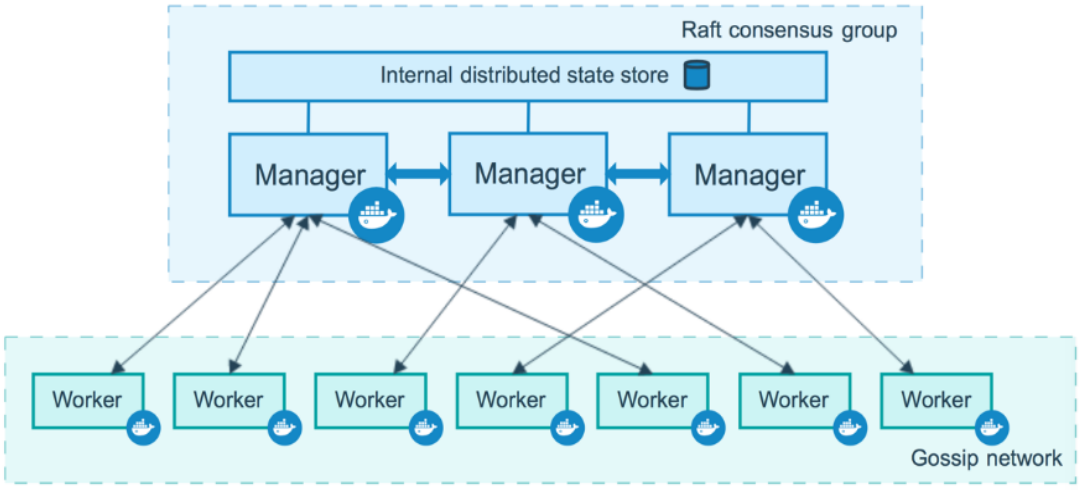


Figura 2.4: *Rete di un docker swarm.* [12]

Un'altra caratteristica del docker swarm è che non tutti i container devono appartenere allo swarm. Un container che non deve comparire all'interno dello swarm, infatti, può essere schedulato su qualsiasi docker daemon, anche interno allo swarm, ma rimarrà isolato. Infine, grazie al docker swarm, è possibile configurare un cluster:

- Decentralizzato: in fase di deploy non c'è bisogno di definire per ogni docker daemon, quali di questi saranno swarm manager e quali worker.
- Scalabile: ogni servizio può essere replicato o ridotto a runtime.
- Sicuro: le comunicazioni tra i nodi avvengono in mutual TLS.
- Aggiornato: ad ogni aggiornamento lo swarm manager è in grado di eseguirlo in modo incrementale sui vari nodi e, in caso di errori, effettuare un roll-back istantaneo.
- Bilanciato: viene esposto un servizio di load balancer che direziona il traffico all'interno del cluster sui vari nodi.[12]

Capitolo 3

Kubernetes

Kubernetes è una piattaforma open-source, portabile ed estensibile, per la gestione dei container. Tale infrastruttura permette quindi il deploy automatico di workload e servizi, all'interno di un cluster di hosts. Più in dettaglio:

- **Rilevazione dei servizi e bilanciamento**. È possibile esporre un'applicazione tramite nome DNS o usando il proprio indirizzo IP. Se il traffico verso l'applicazione esposta è elevato, kubernetes è capace di distribuire il traffico in modo bilanciato.
- **Gestione dello storage**. La piattaforma consente di allocare una memoria di massa a disposizione per un'applicazione montando un volume come local storage.
- **Rollout e rollback automatici**. Descrivendo lo stato desiderato di un'applicazione, kubernetes si occupa di modificare i container in esecuzione ad una velocità controllata.
- **Packing automatico**. Un container può aver bisogno di uno specifico livello di cpu e memoria ram. Kubernetes si occupa di assegnare il container a un nodo del suo cluster con le risorse adatte.
- **Auto risanamento**. Automaticamente vengono sostituiti e riavviati i container che falliscono, pubblicati all'utente solo quando sono in stato di ready.
- **Configurazione e gestione dei segreti**. Password, OAuth token e chiavi ssh possono essere memorizzate e gestite da kubernetes. È possibile infatti, il deploy e l'aggiornamento di segreti senza ridefinire l'immagine del container.

Sebbene sia Docker swarm che Kubernetes offrono una gestione dei container, quest'ultimo consente configurazioni più specifiche per ogni tipo di richiesta, si presta meglio per applicazioni complesse, tempi di risposta affidabili e veloci, e cluster di grandi dimensioni.

3.1 Componenti principali

Il cluster kubernetes è organizzato in un nodo master e più nodi worker. Il nodo master gestisce i nodi worker e i workload all'interno del cluster, mentre i nodi worker li ospitano. Più nodi master possono coesistere per fornire un cluster affidabile e reagente ai failover.

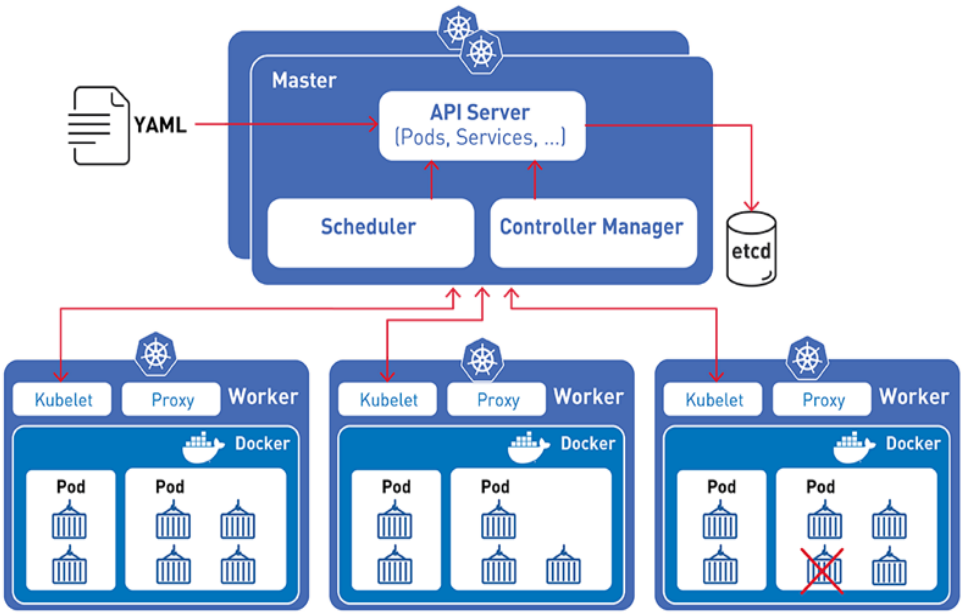


Figura 3.1: *Kubernetes framework.* [13]

3.1.1 Master

Le componenti kubernetes presenti nel master agiscono maggiormente sul control plane, rendendo le decisioni globali all'interno del cluster e reagendo agli eventi identificati.

Kube-Controller-Manager

Componente che esegue i controller. Ciascun controller è un processo separato, ma per semplicità, tutti i controller sono compilati in un unico file ed eseguiti in un unico processo. I controller includono:

- Node Controller: notifica e risponde quando un nodo è in fail.
- Replication Controller: mantiene il numero specificato di istanze per ogni applicazione nel sistema.
- Endpoints Controller: popola gli oggetti Endpoints, ovvero esegue il join tra servizio e workload.
- Service Account e Token Controller: creano gli account e le api di accesso ai token per nuovi namespace.

Kube-Scheduler

Monitora i nuovi workload che non hanno ancora un nodo worker assegnato e gliene assegna uno tenendo conto di alcuni fattori come risorse richieste, politiche di rete, requisiti hardware e software.

Etcd

Storage distribuito usato come punto di riferimento per tutti i dati del cluster all'interno di kubernetes. Grazie alla struttura di tipo key-value offre un'alta disponibilità di dati, generando sempre una nuova struttura e mantenendo tutte le versioni passate.

Kube-apiserver

L'API server è la componente del control plane che espone le API di kubernetes, il suo front end. Kube-apiserver è progettato per scalare orizzontalmente in modo da avere più istanze.

3.1.2 Worker

Le componenti del worker, invece, agiscono sul data plane mantenenedo le applicazioni in stato di running e l'ambiente kubernetes attivo.

Kubelet

È la componente che gestisce i container creati da kubernetes, assicurandosi che siano in stato di running. Essa comunica con il master e, una volta ricevute le specifiche di un workload, si assicura che esse vengano garantite.

Kube-proxy

Un proxy di rete che mantiene delle regole sui nodi. Tali regole permettono alle applicazioni di comunicare con l'esterno e viceversa. Tale proxy utilizza il filtro del sistema operativo, se presente, altrimenti inoltra tutto il traffico.

Container Runtime

Il software che esegue i container, ovvero docker, containerd o qualsiasi altro software, supportato da kubernetes, che implementa la CRI (Container Runtime Interface).

3.2 Unità fondamentali

Nella sezione precendente più volte si è fatto riferimento a workload o applicazioni che all'interno del framework kubernetes prendono il nome di Pod, a volte creati all'interno di Deploy. Nello specifico kubernetes utilizza pod, deploy e servizi per gestire le applicazioni containerizzate.

3.2.1 Pod

Un pod è l'unità base di kubernetes, l'oggetto più piccolo che si può creare e distribuire. Un pod rappresenta un processo all'interno del cluster e ingloba un container, un indirizzo ip e le opzioni di esecuzione del container. I pod possono contenere uno o più container:

- *Singolo container*. Il pod agisce da wrapper al container, permettendo a kubernetes di gestire indirettamente il container.
- *Molteplici container*. Il pod incapsula più container che vengono eseguiti in contemporanea, condividendo le stesse risorse tra loro. Kubernetes, grazie al pod, tratta i container inclusi come una singola entità.

Ciascun pod è pensato per eseguire una singola istanza dell'applicazione. Nel caso in cui si voglia scalare orizzontalmente quest'ultima, bisognerà utilizzare diversi pod monitorati da un controller.

3.2.2 Volume

Quando un container all'interno del pod salva dei file o log, questi sono coerenti solo fino a quando il pod è in esecuzione. Un volume è l'oggetto di kubernetes che permette il salvataggio e il recupero dei dati per un container anche se viene

riavviato. Il volume ha lo stesso ciclo di vita del pod a cui è associato. Per usare un volume, all'interno del template del pod bisogna indicare il tipo (campo .spec.volumes) e dove deve essere montato all'interno del container (campo .spec.containers[*].volumeMounts).

Figura 3.2: *Kubernetes, pod e volumi.* [14]

3.2.3 ReplicaSet

Un ReplicaSet è un controller definito da alcuni campi che consentono di selezionare i pod da acquisire, di indicare il numero di repliche che devono coesistere e, tramite un template, specificare i dati dei nuovi pod, come immagine e porta del container. Il controller, quindi, andrà a creare o cancellare pod a seconda delle istruzioni definite all'interno del suo template. All'interno di ogni pod acquisito dal replicaset, tra i metadati, vi è un link che punta al controller ed è proprio grazie a tale link che lo stato del pod viene controllato in tempo reale.

3.2.4 Deployment

Il Deployment è anch'esso un controller ma fornisce aggiornamenti continui a pod e replicaset. Tramite lo stato descritto all'interno del deployment il controller, ad una velocità controllata, può cambiare lo stato attuale dei pod. Il Deployment permette di creare nuovi replicaset, o rimuovere altri deployment acquisendo le loro risorse. Il deployment è utile quando si vuole creare, aggiornare o rimuovere un ReplicaSet, dichiarare un nuovo stato dei pod modificando il campo PodTemplateSpec, fare rollback ad un deployment precedente, scalare o mettere in pausa lo stesso, per poi farlo ripartire una volta corretto degli errori presenti.

3.2.5 Service

Un service è un'astrazione che raggruppa logicamente un insieme di Pod, definendo le politiche di accesso ad essi. Kubernetes assegna ad ogni servizio un indirizzo

ip e un nome dns univoco all'interno del cluster. Grazie a questa funzionalità i pod di un deploy possono comunicare con altri pod utilizzando i nomi dei servizi, a discapito degli indirizzi ip dei pod. L'ip di un pod, infatti, può essere modificato da un update di un deploy, o da un crash seguito da un riavvio del pod e il nome univoco di un servizio garantisce la continua comunicazione. All'interno del template di un servizio è possibile specificare, tramite un selector (campo spec.selector), la label che accosterà il pod al servizio. Il tipo di un servizio è di default ClusterIP ma si può esporre tramite:

- *NodePort*. Kubernetes esporrà il servizio su ogni nodo del cluster su una porta appartenente all'intervallo 30000-32767. Il servizio sarà raggiungibile quindi all'indirizzo `NodeIP:Port`
- *LoadBalancer*. Kubernetes userà il proprio Load Balancer per esporre il servizio su indirizzo ip esterno.

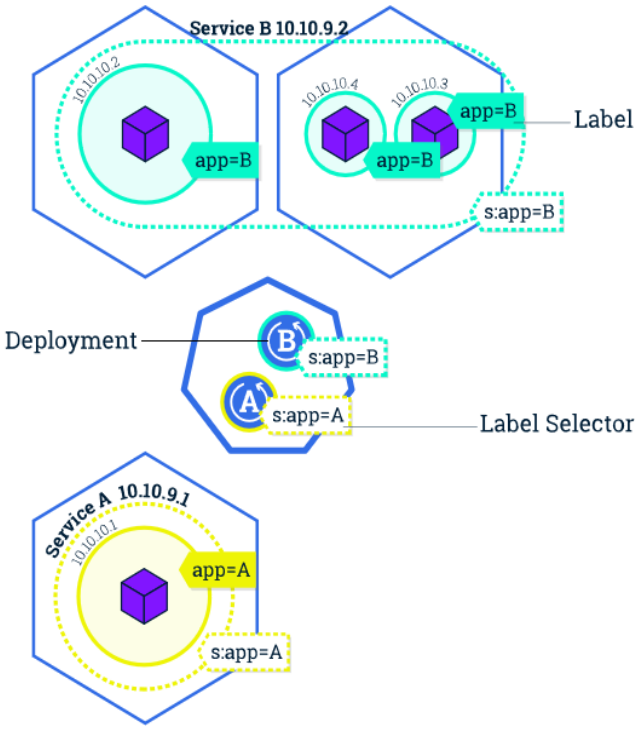


Figura 3.3: *Kubernetes, servizi e deployment.* [14]

3.3 CNI. Container Network Interface

La cni è la componente di kubernetes che si occupa di creare una rete di overlay per garantire la comunicazione tra i pod. In questo modo un pod assegnato ad un nodo i può comunicare con un pod di un nodo j senza che vengano configurate rotte o regole sui router e switch fisici. La rete di overlay in particolare crea un tunnel tra gli host che vengono aggiunti o rimossi dai nodi, rispettivamente all'invio o ricezione di un pacchetto. Kubernetes offre un plugin di default, *kubenet*, che però non è adatto per configurazioni complesse. Molti provider che implementano un servizio di cni come Weave, Flannel, Cilium, Calico sono supportati da kubernetes.[14]

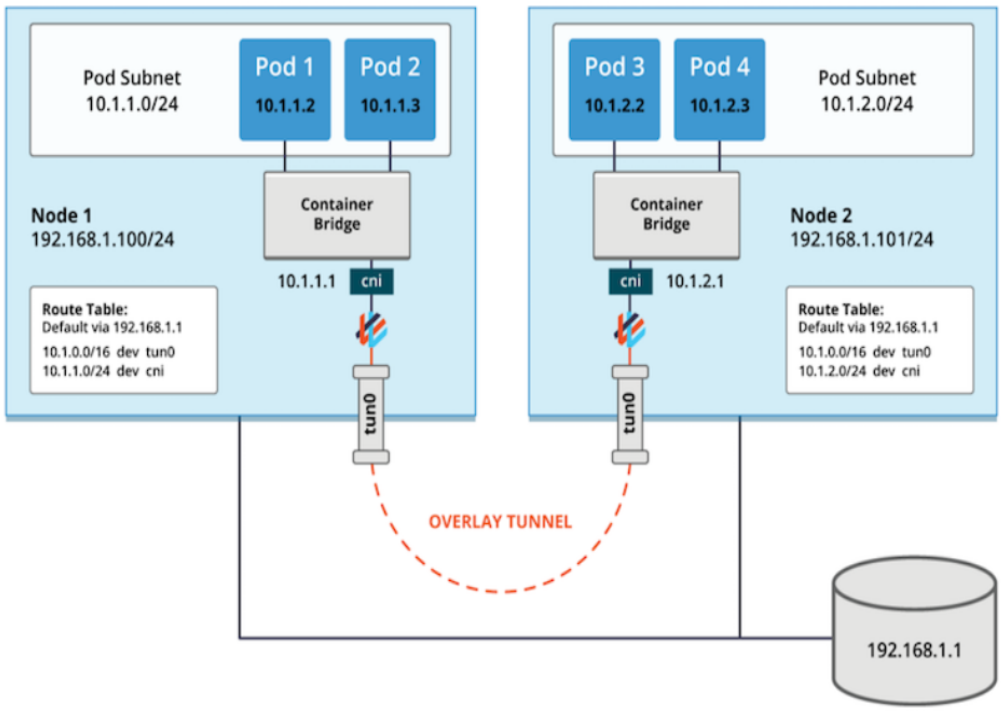


Figura 3.4: *Rete di overlay Kubernetes.* [15]

Capitolo 4

Istio

Istio è una piattaforma open source che si applica in modo trasparente alle applicazioni già esistenti all'interno del cluster. Istio aiuta i team di DevOps a ridurre la complessità delle operazioni di distribuzione controllando, proteggendo, connettendo e monitorando i servizi. Essendo una piattaforma libera, essa include le sue API consentendo l'installazione su qualsiasi infrastruttura cloud che controlla microservizi.

4.1 Service Mesh

Convertendo le applicazioni da monolitiche a microservizi, una volta distribuite, si viene a formare una rete di microservizi costituita dai vari workload e dalle interazioni tra loro. Man mano che la service mesh si estende, la complessità aumenta e può essere difficile gestirla. I requisiti da analizzare possono includere individuazione, bilanciamento del carico, ripristino degli errori, metriche e monitoraggio. Alcune volte bisogna anche limitare le connessioni, effettuare un controllo degli accessi, autenticare una comunicazione tra servizi e molto altro ancora. Istio si colloca proprio a questo livello e cerca di risolvere tali problematiche all'interno di una service mesh. Il framework Istio fornisce informazioni ed un controllo operativo sulla rete di servizi, offrendo una soluzione completa per soddisfare i diversi requisiti delle applicazioni a microservizi.

4.2 Aspetti Caratteristici

Gestione del traffico

Con Istio è possibile impostare delle semplici regole per il routing del traffico, controllando così il flusso e le chiamate API tra i servizi. Tale piattaforma semplifica la configurazione a livello di servizio inserendo regole, interruttori di cir-

cuito, timeout, tentativi di riconnessione e permette l'implementazione di attività importanti come test A/B, canary deploy e distribuzione del traffico basate su percentuali. Offrendo una migliore visibilità sul traffico, è possibile rilevare e correggere degli errori prima che causino fallimenti rendendo, così, la rete più affidabile.

Sicurezza

Le funzionalità di sicurezza di Istio mettono a disposizione degli sviluppatori una protezione a livello applicativo. Istio fornisce un canale di comunicazione sottostante protetto e gestisce crittografia, autorizzazione e autenticazione delle comunicazioni di un servizio, il tutto senza o con poche modifiche all'applicazione. Nonostante Istio sia indipendente dall'ambiente in cui viene installato, utilizzandolo con le policy di rete dell'infrastruttura Kubernetes, i vantaggi sono ancora maggiori, includendo la sicurezza di comunicazione pod-a-pod o servizio-a-servizio a livello di rete e applicativo.

Controllo

Mediante le funzionalità di Istio di tracciatura, monitoraggio e registrazione degli errori (logging) è possibile ottenere informazioni sulla rete di servizi e su quanto un servizio impatta sugli altri. L'architettura Istio divide il back-end dal front-end nascondendo i dettagli implementativi delle componenti di controllo al front-end.

4.3 Struttura Istio

La rete di servizi Istio divide l'architettura in due livelli, quello dei dati e quello di controllo. Il livello dati è composto da un insieme di proxy, denominati Envoy, che filtrano e monitorano il traffico tra gli Envoy o tra Envoy e il piano di controllo. Tali proxy vengono distribuiti all'interno della rete come sidecar allegati ai servizi. Il livello di controllo agisce, sostanzialmente, sul livello dati configurando gli envoy, applicando le policy, di rete e sicurezza e raccogliendo le metriche.

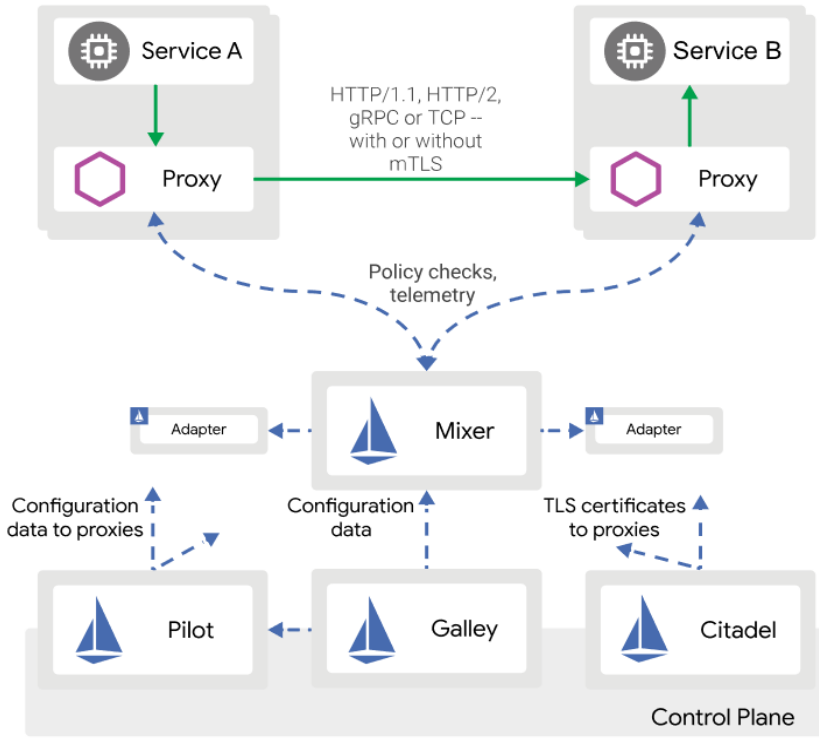


Figura 4.1: *Architettura Istio.* [16]

4.4 Data Plane

Envoy Proxy

Istio estende la versione base del proxy Envoy. L'Envoy è un proxy, sviluppato in C++, che agisce da mediatore per tutto il traffico dei servizi in ingresso e in uscita all'interno della rete mesh. Istio sfrutta le molteplici funzionalità integrate dell'Envoy come: individuazione dinamica dei servizi, bilanciamento del carico, proxy HTTP/2, HTTP e gRPC, divisione del traffico basata su percentuali, terminazione TLS e raccolta di metriche. L'Envoy è distribuito come sidecar del servizio all'interno del pod Kubernetes. Questa distribuzione consente a Istio di ricavare una grande quantità di informazioni sul comportamento del traffico come *"attributi"*. Istio utilizza queste informazioni per applicare le policy, di rete e sicurezza, e inviarle al sistema di monitoraggio per fornire indicazioni sul comportamento dell'intera mesh.

4.5 Control Plane

Pilot

Pilot consente l'individuazione del servizio per gli Envoy, fornisce le funzionalità di gestione del traffico per un routing intelligente come test A/B e canary deploy e provvede alla resilienza con timeout, tentativi di riconnessione e interruttori di circuito. Il componente Pilot converte le policy di routing e sicurezza in configurazioni personalizzate, inviate in fase di esecuzione agli Envoy. Pilot astrae i meccanismi di individuazione dei servizi, specifici della piattaforma, e li converte in uno standard per gli Envoy, permettendo ad Istio una portabilità su più ambienti come Kubernetes, Console o Nomad.

Galley

Galley è la componente che si occupa di inserire, elaborare, distribuire e validare la configurazione di Istio. Essa è responsabile dell'isolamento del resto dei componenti Istio dai dettagli della piattaforma sottostante.

Citadel

Citadel si occupa invece della sicurezza all'interno della rete di servizi. Essa consente un'autenticazione forte tra servizi e utente finale gestendo identità e credenziali. Tramite Citadel è possibile aggiornare il traffico nella rete mesh crittografandolo. Utilizzando Citadel, gli operatori possono applicare criteri anche ad alto livello, oltre che identificatori di rete di livello 3 o di livello 4.

4.6 Mixer

Sezione a parte merita il Mixer che si colloca come livello intermedio tra il data plane e il control plane. Grazie al Mixer, infatti, viene offerta ai front-end una singola interfaccia per interagire con i componenti del control plane. Il Mixer applica quindi ai servizi policy di rete, criteri di sicurezza e metriche per il monitoraggio.

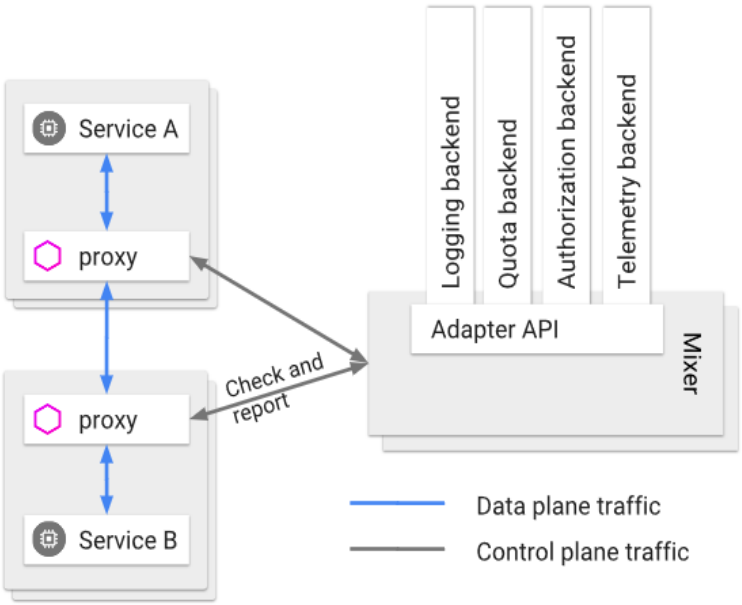


Figura 4.2: *Topologia del Mixer Istio.* [16]

Prima di ogni richiesta da parte di un servizio, il proxy associato chiama il Mixer per eseguire controlli sulle policy. Dopo la richiesta, invece, il proxy comunica con il Mixer per i report di telemetria. Il sidecar dispone di una cache locale in modo da effettuare una grande percentuale di controlli prima di chiamare il mixer. Inoltre, il sidecar memorizza nel buffer i dati di telemetria in uscita, limitando così le chiamate al Mixer. L'applicazione dei criteri e la raccolta di dati di telemetria sono interamente guidate dalla configurazione di Istio nel control plane. L'interazione tra envoy e mixer di default è disabilitata.

Adapter

Mixer è un componente altamente modulare ed estensibile. La sua funzione principale è quella di astrarre i dettagli dei diversi sistemi back-end e criteri di telemetria, consentendo al resto di Istio di essere indipendente dai back-end. La flessibilità del Mixer deriva proprio dal suo modello di plug-in generico. I singoli plug-in sono noti come adapter e consentono al Mixer di interfacciarsi con diversi back-end che forniscono funzionalità di base quali registrazione, monitoraggio e controllo degli accessi. Istio supporta adapter di back-end come Bluemix, Prometheus, AmazonWebService, Kubernetes e IBM cloud.

Attributi

Istio, come riportato nella sezione precedente, si avvale di informazioni per generare le metriche e filtrare il traffico attraverso le varie policy adottate, gli attributi.

Un attributo è un piccolo insieme di dati che descrive una singola proprietà di un servizio o di una specifica richiesta di quel servizio. Ogni attributo è costituito da un nome e un tipo (string, int64, boolean e altri). L'envoy, ad ogni richiesta del servizio associato, genera gli attributi relativi a quella richiesta e li fornisce al Mixer. Quest'ultimo elabora il set di attributi verificando policy o interagendo con diversi back-end del control plane.

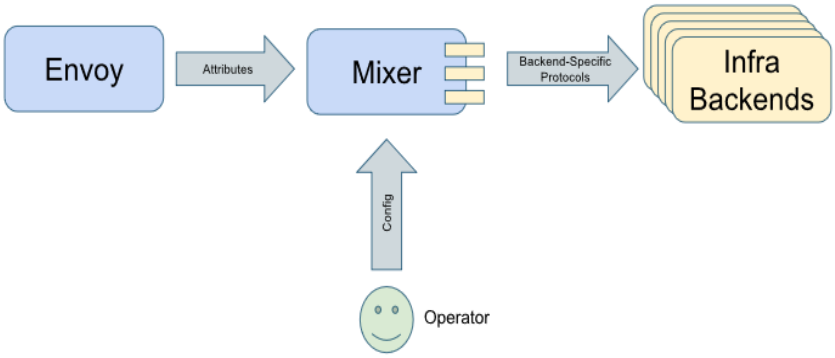


Figura 4.3: *Elaborazione attributi Istio.* [16]

Capitolo 5

Istio traffic management

Per indirizzare il traffico all'interno della rete mesh, Istio deve essere a conoscenza della posizione di tutti gli endpoint e a quali servizi appartengono. Istio si connette a un sistema di individuazione dei servizi popolando il suo registro. Tramite questo registro gli envoy possono inoltrare il traffico ai servizi richiesti. Con le applicazioni a microservizi, per ogni servizio ci possono essere molte istanze dello stesso workload, costruendo un pool di bilanciamento. Di default, il bilanciamento viene effettuato tramite round robin, ma con Istio si possono applicare policy specifiche indirizzando una percentuale di traffico a una nuova versione dell'applicazione in un test A/B o applicare criteri di bilanciamento del carico diversi solo per un particolare sottoinsieme di istanze del servizio. È anche possibile applicare regole al traffico in entrata o in uscita dalla rete aggiungendo, ad esempio, una dipendenza esterna. Per applicare queste configurazioni, Istio utilizza delle risorse personalizzate, le CRD (custom resource definitions), estensioni di alcune risorse kubernetes.

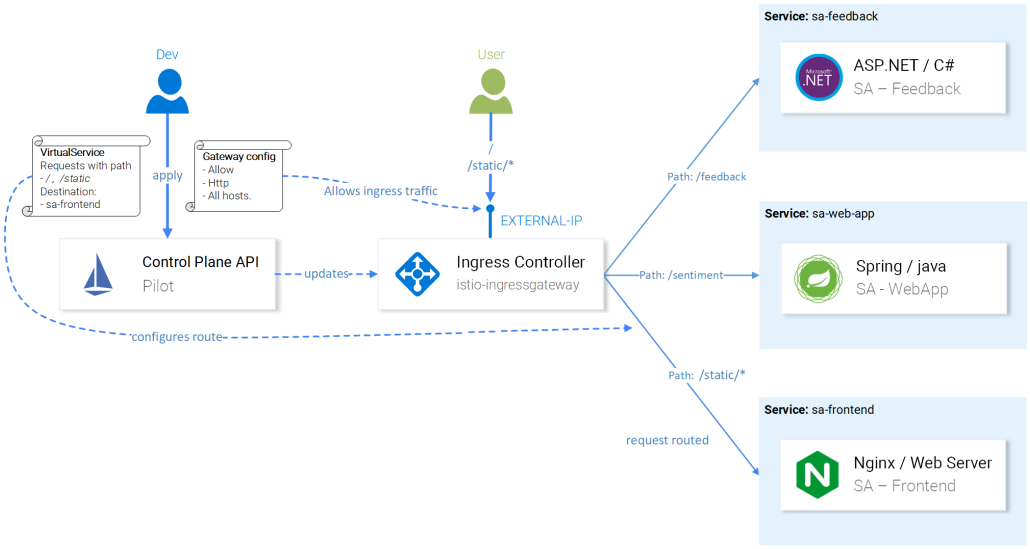


Figura 5.1: *Politiche di traffic routing.* [17]

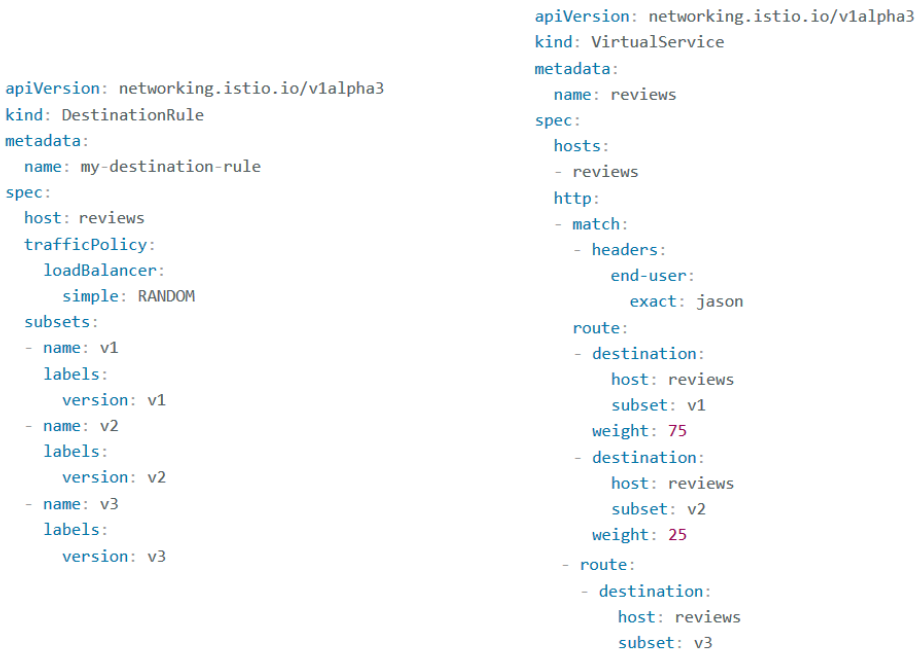
5.1 Virtual Service

I virtual service costituiscono un elemento fondamentale per il routing del traffico all'interno della rete Istio. Tramite i virtual service è possibile instradare il traffico verso istanze specifiche, applicando delle regole di routing. Gli Envoy, a questo punto, non inoltreranno più il traffico alle destinazioni appropriate secondo round robin, ma secondo le regole indicate all'interno del virtual service. Tra gli utilizzi tipici vi sono direzionare il traffico verso una specifica istanza di un servizio, controllare alcuni campi della richiesta prima di essere inoltrata, riscrivere parte dell'uri richiesto, inserire un ritardo o bloccare una richiesta inviando un codice di errore. Il routing del traffico è completamente separato dalla distribuzione dell'istanza, ciò significa che il numero di istanze che implementano una versione del servizio può aumentare o diminuire in base al carico di traffico senza fare riferimento alle regole di routing del traffico definite. Al contrario, le piattaforme di orchestrazione come Kubernetes supportano solo la distribuzione del traffico basata sul numero effettivo delle istanze, da scalare manualmente a priori.

5.2 Destination Rule

Un altro fattore che contribuisce, insieme ai virtual service, al traffic routing sono le destination rules, le quali definiscono i sottoinsiemi di una destinazione indicata all'interno dei virtual service, lavorando in associazione con essi. Le destination rules, inoltre, indicano il modo in cui il traffico deve essere distribuito all'interno

di una specifica destinazione. Esse vengono applicate dopo la valutazione delle regole di routing del virtual service, ovvero una volta arrivati alla destinazione "reale" e non al servizio.



```yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: my-destination-rule
spec:
  host: reviews
  trafficPolicy:
    loadBalancer:
      simple: RANDOM
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
  - name: v3
    labels:
      version: v3
```

(a)

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: reviews
spec:
  hosts:
  - reviews
  http:
  - match:
    - headers:
        end-user:
          exact: jason
    route:
    - destination:
        host: reviews
        subset: v1
      weight: 75
    - destination:
        host: reviews
        subset: v2
      weight: 25
  - route:
    - destination:
        host: reviews
        subset: v3
```

(b)

Figura 5.2: *Destination rules (a) con associato Virtual service (b).* [16]

Nell'esempio in figura 5.1a viene mostrato come per l'host reviews vengono definiti tre sottoinsiemi (v1, v2, v3) dati dalla label version. Di default viene applicata una politica random. Nel virtual service invece, come mostrato in figura 5.1b, viene controllato il traffico http e, se viene soddisfatta la regola data dal campo `spec.http.match`, il traffico viene inoltrato alle due versioni, v1 e v2, del service reviews in modo pesato. Infine il traffico che non soddisfa la regola, viene assegnato alla versione v3.

5.3 Gateway

Il gateway è uno strumento che permette la manipolazione del traffico in ingresso e in uscita dalla rete Istio. Le configurazioni del gateway vengono applicate a dei proxy Envoy autonomi in esecuzione sul "bordo" della rete. Diversamente dai meccanismi di altre piattaforme, i gateway Istio consentono di utilizzare le proprietà di livello 4-6, ad esempio le porte da esporre, le impostazioni TLS e così via, poiché ad ogni gateway può essere associato un virtual service. Principalmente vengono usati gateway di ingresso, ma anche quelli di uscita possono avere

un ruolo particolare. Fornendo un gateway di uscita alla rete Istio è possibile limitare le connessioni dei servizi verso reti esterne o generare un tunnel TLS. Istio fornisce delle distribuzioni gateway preconfigurate, istio-ingressgateway per l'ingresso e istio-egressgateway per l'uscita, ma è possibile creare i propri gateway che estendono quelli standard. I gateway vengono associati ai virtual service tramite la voce `spec.gateway`.

```yaml
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: my-gateway
  namespace: some-config-namespace
spec:
  selector:
    app: my-gateway-controller
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - uk.bookinfo.com
    - eu.bookinfo.com
```

Figura 5.3: *Istio ingress-gateway personalizzato.* [16]

5.4 Service Entry

Una service entry, come suggerisce il nome, aggiunge una voce al registro di sistema dei servizi che Istio gestisce internamente. Definendo una service entry, gli Envoy possono comunicarvi come se si trattasse di un servizio della rete mesh. La configurazione delle service entry consente, principalmente, di gestire il traffico dei servizi all'esterno della rete.

5.5 Sidecar

Nella rete Istio, di default ogni proxy è configurato per accettare il traffico su tutte le porte del workload associato e per raggiungere tutti gli altri workload presenti in rete. Tramite l'utilizzo dei sidecar è possibile limitare l'insieme di porte e protocolli accettati da ogni envoy e con quali servizi poter comunicare. Tramite il `workloadSelector` nel file di configurazione, viene indicato per quali workload si devono applicare le regole definite.

Capitolo 6

Istio security

All'interno di una service mesh, è possibile che i servizi abbiano bisogno di un certo livello di sicurezza per comunicare con altri servizi. Un servizio A che richiede risorse ad un servizio B, ad esempio, ha bisogno di essere autenticato e autorizzato. Istio cerca di offrire una soluzione completa ai problemi di sicurezza agendo su più fronti fornendo identità, criteri di accesso, crittografia TLS e strumenti di autenticazione, autorizzazione e controllo (AAA) per proteggere i servizi e i dati. La sicurezza di Istio non impone cambiamenti a livello di codice per applicazioni e infrastrutture, integra i criteri di sicurezza con quelli della piattaforma sottostante e costruisce una piattaforma sicura su reti non sicure.

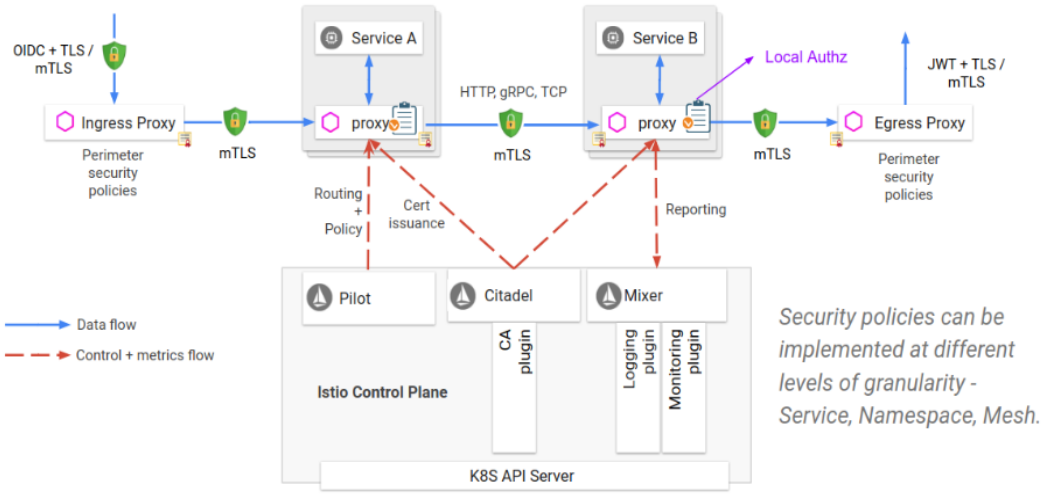


Figura 6.1: *Architettura sicurezza Istio.* [16]

L'architettura di sicurezza coinvolge diverse componenti di Istio, come il Citadel per la gestione dei certificati, il Pilot per le policy di autenticazione, autorizzazione e informazioni sui nomi dei servizi, il Mixer per il controllo e infine i proxy per una comunicazione sicura.

6.1 Identità

Le informazioni sui nomi dei servizi sicuri contengono mapping N-a-N delle identità del server codificate nei certificati. Un mapping dall'identità A al nome del servizio B significa che il client A è autorizzato a eseguire il servizio B. Il Pilot, tramite il Kubernetes apiserver, genera le relazioni dei nomi sicuri e le distribuisce in modo sicuro ai proxy Envoy. All'interno della service mesh i servizi, durante una comunicazione sicura, devono fornire la propria identità per confermare l'autenticazione. Il client confronta l'identità del server con le informazioni ricevute dal pilot. Il server può determinare, invece, a quali risorse il client può accedere in base ai criteri di autorizzazione e controllare chi ha avuto accesso a cosa e a che ora ed addebitare costi ai client in base ai servizi utilizzati. Nel modello di identità Istio, la piattaforma usa l'identità di prima classe per determinarne una di un servizio, offrendo maggiore flessibilità e granularità. Nelle piattaforme in cui non è disponibile tale identità, Istio può usarne altre come i nomi dei servizi. In Kubernetes viene usato il service-account, mentre in piattaforme GKE/GCE/GCP il service-account GCP. Istio aderisce allo standard di sicurezza SPIFFE, Secure Production Identity Framework for Everyone, per cui l'identità è descritta nel formato `spiffe://<domain>/ns/<namespace>/sa/<serviceaccount>`.

6.2 Public Key Infrastructure

L'infrastruttura PKI di Istio è costruita tramite la componente Istio Citadel e fornisce identità forti per ogni workload, trasmesse attraverso certificati x.509 in formato standard SPIFFE. L'infrastruttura PKI di Istio automatizza, inoltre, la gestione delle chiavi e dei certificati in base alla piattaforma sottostante.

- *Kubernetes*:
    1. Citadel, interagendo con il kubernetes apiserver, crea un certificato SPIFFE e una coppia di chiavi per ogni ServiceAccount, memorizzandoli tramite segreti kubernetes.
    2. Alla creazione di un pod, Kubernetes monta i certificati e le chiavi, secondo il service account specificato, all'interno di un volume segreto.
    3. Citadel controlla il ciclo di vita dei certificati e automaticamente ne crea nuovi riscrivendo i segreti kubernetes.
- *Macchine non Kubernetes*:
    1. Citadel crea un servizio gRPC per gestire le Certificate Signing Requests (CSRs).

2. La piattaforma una volta generate la chiave e la CSR, tramite node agent, invia le credenziali con la CSR alla componente Citadel per la firma.
3. Citadel conferma le credenziali create, firma la CSR e genera il certificato.
4. Il node agent quindi, invia il certificato e la chiave privata ricevuti da Citadel al proxy Envoy.
5. Il processo di generazione CSR e della firma viene poi ripetuto alla scadenza del certificato.

6.3 Autenticazione

Tramite file di configurazione si possono specificare requisiti di autenticazione che verranno salvati nello storage di configurazione Istio. Il controller Pilot monitora tali configurazioni e, in caso di aggiornamenti, ha il compito di modificare le configurazioni dei proxy Envoy in modo asincrono. Una volta che il proxy riceve la configurazione, il nuovo requisito di autenticazione ha effetto immediato su tale pod.

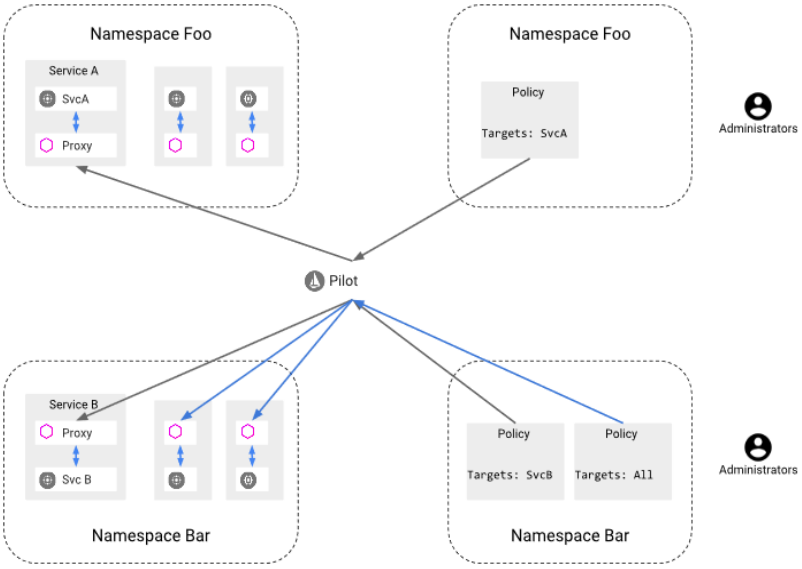


Figura 6.2: *Architettura autenticazione Istio.* [16]

I criteri di autenticazione possono essere abilitati per tutta la rete mesh attraverso una MeshPolicy, per un certo namespace attraverso una Policy indicando il namespace, o abilitare criteri più granulari a livello di servizi tramite una Policy

indicando sia il namespace che i servizi. A livello di rete mesh può esserci una sola MeshPolicy, mentre per namespace si può avere una Policy per ogni namespace. A livello di servizi si possono avere, infine, più policy per ogni servizio. Istio fornisce due tipi di autenticazione:

- *Autenticazione di trasporto, authN service-to-service*: verifica il client diretto che effettua la connessione. Istio offre mutual TLS per l'autenticazione di trasporto. Ad ogni servizio quindi viene fornita un'identità forte, proteggendo comunicazioni servizio-servizio e utente finale-servizio. La generazione e distribuzione di chiavi e certificati vengono eseguite in modo automatico.
- *Autenticazione dell'origine, authN end-user*: verifica il client originale che effettua la richiesta. Istio abilita l'autenticazione con la convalida JSON Web Token (JWT) e un livello semplificato per provider open source come ORY Hydra, Keycloak, Auth0, Firebase Auth, Google.

I servizi che agiscono da client sono responsabili di eseguire il meccanismo di autenticazione necessario.

6.3.1 Mutual TLS

Quando la comunicazione avviene tramite mutual TLS, il traffico da servizio a servizio viene fatto passare attraverso un tunnel sicuro tra i proxy Envoy. In questo modo quando un servizio client ne chiama uno server:

1. Istio reindirizza il traffico in uscita da un client al proxy Envoy del client.
2. L'Envoy client esegue la fase di TLS handshake con l'Envoy server. Durante l'handshake, l'Envoy client esegue anche un controllo sui nomi dei servizi sicuri per verificare che il servizio sia autorizzato a eseguire il servizio di destinazione.
3. Dopo una connessione TLS tra i proxy, Istio inoltra il traffico dall'Envoy client a quello server.
4. Il proxy Envoy server inoltra il traffico al servizio server tramite connessioni TCP locali.

Lato server il mutual TLS è configurabile tramite Policy o MeshPolicy con la voce `spec.peers:- mtls{}`. Di default Istio applica mutual TLS in modo STRICT abilitando solo connessioni TLS, ma è possibile impostare il modo PERMISSIVE per abilitare sia connessioni TLS che connessioni non crittografate e autenticate. Lato client, invece, sono le destination rules ad attivare il mutual TLS settando il campo `spec.trafficPolicy.tls.mode` che può avere i seguenti valori:

- *DISABLE*: disabilita connessioni TLS a monte.
- *SIMPLE*: abilita connessioni TLS a monte.
- *MUTUAL*: abilita connessioni in mutual TLS presentando il certificato client.
- *ISTIO_MUTUAL*: abilita connessioni in mutual TLS ma i certificati vengono generati e gestiti da Istio.

Il controller Pilot avrà cura di installare e fornire ai pod dell'applicazione i certificati e le chiavi private per mutual TLS.

6.3.2 Json Web Token

Per l'autenticazione dell'origine (JWT), l'applicazione deve richiedere e ottenere le credenziali JWT per la richiesta. Tale autenticazione viene abilitata nella sezione `spec.origins` che definisce i metodi e i parametri associati per l'autenticazione come le autorità, l'issuer o regole per l'esclusione di alcuni path relativi. L'autenticazione è superata se almeno uno dei metodi viene soddisfatto. Il Pilot, occupandosi dei criteri di autenticazione, recupera la chiave pubblica collegandola alla configurazione per la convalida JWT.

6.4 Autorizzazione

L'autorizzazione all'interno del framework Istio avviene tramite RBAC, controllo degli accessi in base al ruolo, e fornisce un controllo degli accessi a livello di namespace, servizi e metodi per i servizi. Le RBACs sfruttano i nomi dei servizi e i service-account, ma anche gli attributi generati dai proxy. I criteri di autorizzazione sono compatibili per tipi di traffico HTTP, HTTPS e TCP. Tali criteri vengono salvati nello storage di configurazione Istio e, ad ogni aggiornamento, il Pilot darà origine ai nuovi file di configurazione per i proxy Envoy. L'oggetto ClusterRbacConfig è una CRD singleton a livello di cluster con nome fisso *"default"* e ne può esistere una sola istanza. All'interno del suo file di configurazione, nella sezione mode si possono impostare quattro valori:

- *OFF*: disabilita l'autorizzazione Istio per tutti i servizi della rete.
- *ON*: abilita l'autorizzazione Istio per tutti i servizi della rete.
- *ON_WITH_INCLUSION*: abilita l'autorizzazione Istio per i namespace indicati nella voce `inclusion.namespaces` e per tutti i servizi della rete indicati nella voce `inclusion.services`.

- *ON_WITH_EXCLUSION*: contrariamente al modo precedente, si abilita l'autorizzazione per tutti i servizi della rete disabilitandola per quelli specificati nella voce `exclusion.namespaces` e nella voce `exclusion.services`.

I criteri di autorizzazione vengono definiti tramite l'uso di una coppia di CRDs, ServiceRole e ServiceRoleBinding. La crd ServiceRole, brevemente, definisce un insieme di autorizzazioni a cui è soggetto un utente che richiede una risorsa protetta. Nelle service role vengono indicati i servizi, i metodi e i path che un utente può richiedere. I servizi devono appartenere allo stesso namespace in cui è definita la service role. La crd ServiceRoleBinding, invece, associa una serie di soggetti a una o più ServiceRole. All'interno di una service role binding si può far riferimento a un utente generico o ad un utente con delle proprietà specifiche definite all'interno di una richiesta JWT.

```yaml
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRoleBinding
metadata:
  name: test-binding-products
  namespace: default
spec:
  subjects:
  - user: "service-account-a"
  - user: "istio-ingress-service-account"
    properties:
      request.auth.claims[email]: "a@foo.com"
  roleRef:
    kind: ServiceRole
    name: "products-viewer"
```

(a)

```yaml
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRole
metadata:
  name: products-viewer
  namespace: default
spec:
  rules:
  - services: ["products.default.svc.cluster.local"]
    methods: ["GET", "HEAD"]
```

(b)

Figura 6.3: *ServiceRoleBinding (a) con associata ServiceRole (b).* [16]

Nell'esempio mostrato in figura 6.3, i metodi GET e HEAD sul servizio products sono permessi all'utente *service-account-a* e all'utente *istio-ingress-service-account* se quest'ultimo, in fase di richiesta, specifica nell'autenticazione JWT il campo mail con il valore *"a@foo.com"*.

Capitolo 7

Istio monitoring

Le componenti di Istio generano dati di telemetria che aiutano i team di DevOps a risolvere i problemi e gestire le applicazioni monitorarando il comportamento dei servizi. Tramite la telemetria, gli operatori acquisiscono una conoscenza approfondita delle interazioni dei servizi con altri servizi o con le componenti del data e control plane Istio.

7.1 Metriche

Istio, tramite i dati di telemetria, fornisce delle metriche per controllare il corretto funzionamento di un servizio. Le metriche offerte interessano tutto il traffico di un servizio in ingresso, in uscita o all'interno della service mesh, informazioni sul volume complessivo del traffico, gli errori e i tempi di risposta per le richieste. Oltre al comportamento dei servizi, è utile monitorare anche il funzionamento delle componenti Istio. Pilot, Mixer, Galley e Citadel, infatti, generano delle metriche di auto monitoraggio permettendo agli operatori un controllo completo su tutta la rete mesh. Per garantire una certa flessione alle esigenze di monitoraggio, le metriche possono essere configurate selezionando come, quando e il livello di dettagli all'interno delle stesse. Le metriche Istio vengono raccolte a partire dai proxy Envoy. Ogni sidecar genera un insieme di dati su tutto il traffico, in ingresso e in uscita, che attraversa il proxy, nonché informazioni sulla configurazione e integrità del sidecar. Istio consente agli operatori di selezionare i dati che devono essere generati dal proxy poichè, come impostazione predefinita, in Istio è abilitato un piccolo insieme per evitare di sovraccaricare le risorse associate alla raccolta delle metriche. Tuttavia, espandere l'insieme di metriche generate dal proxy consente un debug mirato all'interno della rete.

```
envoy_cluster_internal_upstream_rq{response_code_class="2xx",cluster_name="xds-grpc"} 7163

envoy_cluster_upstream_rq_completed{cluster_name="xds-grpc"} 7164

envoy_cluster_ssl_connection_error{cluster_name="xds-grpc"} 0

envoy_cluster_lb_subsets_removed{cluster_name="xds-grpc"} 0

envoy_cluster_internal_upstream_rq{response_code="503",cluster_name="xds-grpc"} 1
```

Figura 7.1: *Esempio di metriche a livello proxy.* [16]

Oltre alle metriche a livello di proxy, Istio genera delle metriche per il monitoraggio dei servizi e delle sue comunicazioni. Le metriche Istio predefinite sono definite da un insieme di attributi di configurazione che vengono inviati nella fase iniziale di configurazione, modificabili poi in seguito.

```
istio_requests_total{
  connection_security_policy="mutual_tls",
  destination_app="details",
  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",
  response_flags="-",
  source_app="productpage",
  source_principal="cluster.local/ns/default/sa/default",
  source_version="v1",
  source_workload="productpage-v1",
  source_workload_namespace="default"
} 214
```

Figura 7.2: *Esempio di metriche a livello servizio.* [16]

7.2 Strumenti

Istio, oltre a generare i dati per le metriche, interagisce con strumenti che aiutano nella configurazione e nel monitoraggio del comportamento della service mesh tramite delle interfacce web. Tali strumenti permettono di visualizzare il grafo della rete di servizi, grafici delle metriche esposte o interrogare con delle query il database in cui istio memorizza tutti i dati collezionati.

7.2.1 Kiali

Kiali è lo strumento che permette di modificare i file yaml di configurazione delle varie componenti di networking come gateway, virtual service e destination rules, ma anche criteri di autorizzazzione e policy come le service role, service role

binding e cluster policy. Attraverso le finestre presenti all'interno dell'interfaccia si possono monitorare lo stato dei servizi, workload o applicazioni ed essere sempre a conoscenza dello stato di un sistema.

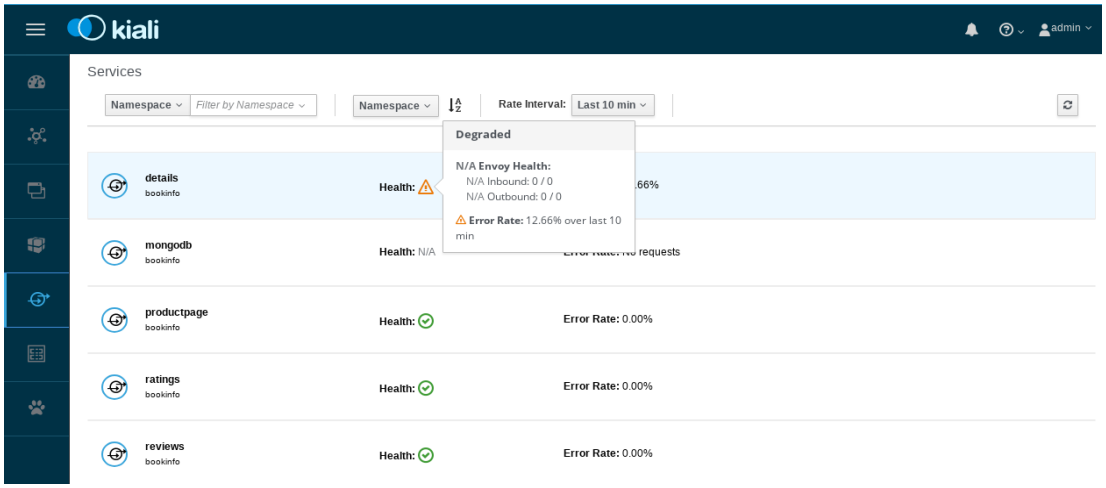


Figura 7.3: *Stato dei servizi monitorati tramite Kiali.* [16]

Un'altra funzionalità utile, messa a disposizione da Kiali, è la visualizzazione granulare della rete mesh. Si può generare, infatti, un grafo a diversi livelli quali applicazione, servizi, workload e app versionate. In questo modo viene mostrata la service mesh attiva e la rete di comunicazione tra le applicazioni o i servizi.

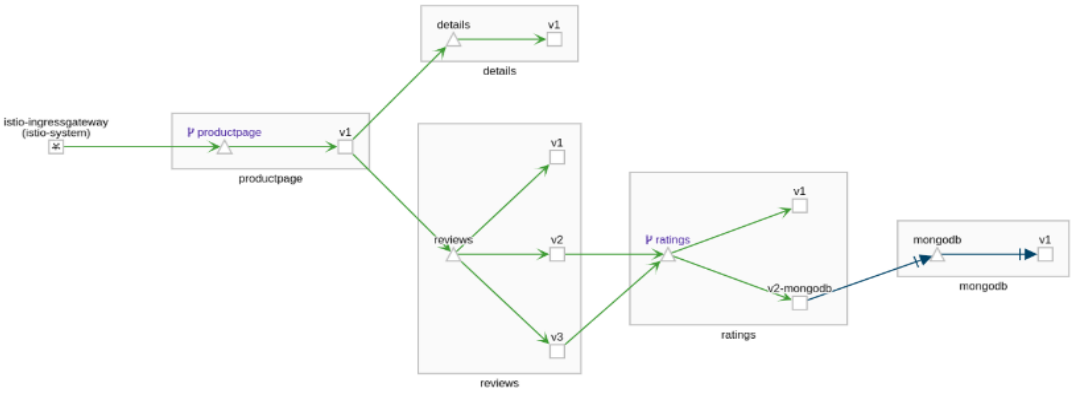


Figura 7.4: *Rete dei servizi monitorati tramite Kiali.* [16]

7.2.2 Grafana

Con il tool Grafana è invece possibile raggruppare i dati raccolti dalle metriche osservando il comportamento dei servizi tramite grafici e tabelle. Grafana, comunicando con il database di Istio, recupera i dati con delle query compilando così le dashbord offerte all'operatore. Istio presuppone questo tool con delle dashboard già predefinite per il controllo delle componenti presenti all'interno della rete Istio

come servizi, workload ma anche componenti del control plane Istio quali Galley, Mixer e Pilot.

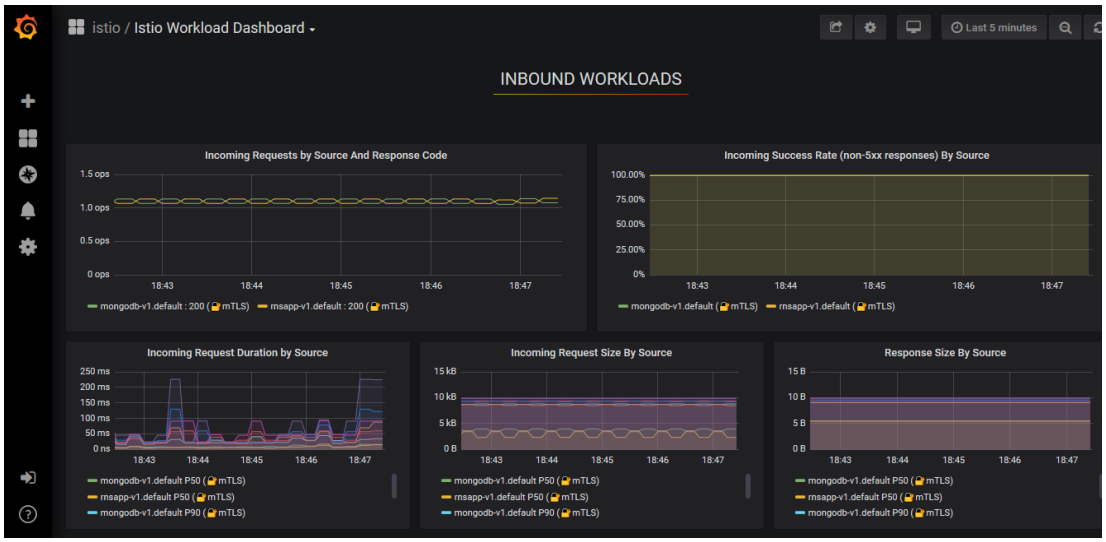


Figura 7.5: *Esempio di una Dashboard Grafana.* [16]

7.2.3 Prometheus

Prometheus è il database interno utilizzato per memorizzare tutti i dati raccolti dalle metriche Istio. Tramite delle query è possibile interrogare il db per un'analisi completa del comportamento dei servizi. Istanziando con dei file di configurazione specifici handler, è inoltre possibile indurre i proxy a generare delle metriche personalizzate migliorando il processo di debug. Prometheus, dunque, raccoglie tutte le metriche prodotte dai proxy o dalle componenti di Istio e le memorizza attraverso serie temporali.

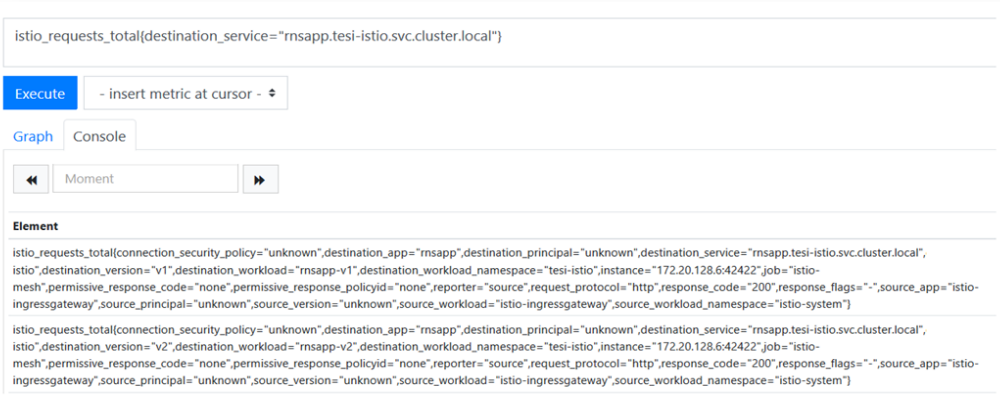


Figura 7.6: *Esempio di una query Prometheus.* [16]

Capitolo 8

Installazione Cluster

Il cluster utilizzato per questo studio è formato da due nodi, uno master e uno worker, rispettivamente denominati kube-master e kube-worker. Le risorse messe a disposizione per il cluster sono 4 virtual CPU, 2 per il master e 2 per il worker, e 10 GB di ram, 2 per il master e 8 per il worker. L'implementazione dell'architettura avviene su macchine con sistema operativo RedHat (CentOS 7) e ha come scopo l'installazione di Docker e Kubernetes, requisiti per l'installazione futura di Istio.

8.1 Configurazione Nodo

8.1.1 Check Preliminari

Prima di installare Docker, Kubernetes o qualsiasi altro software di gestione dei container, bisogna registrare il nodo con un nome che sarà univoco all'interno del cluster e modificare alcune componenti base del kernel linux. Tramite il programma `hostnamectl` è possibile modificare il nome dell'host, quindi si assegna il nome al nodo tramite il comando:

```
hostnamectl set-hostname 'kube-master'
```

Una volta eseguito il comando, modificare il file */etc/hosts* aggiungendo il nome e l'indirizzo ip del nodo. Questo passaggio verrà eseguito per ogni nodo all'interno del cluster, in questo modo i vari nodi saranno in grado di identificarsi e di comunicare sulla rete locale. Successivamente bisogna disabilitare lo swap e il modulo SELinux tramite i comandi:

```
setenforce 0
sed -i --follow-symlinks 's/SELINUX=enforcing/SELINUX=disabled/g' \
     /etc/sysconfig/selinux
swapoff -a
```

Lo swap nel sistema Linux viene fatto partire ad ogni avvio, ma è incompatibile con l'installazione di kubernetes. È necessario, quindi, commentare la linea relativa nel file */etc/fstab* se si desidera evitare di ripetere il comando a ogni riavvio della macchina. Vanno invece abilitati, tramite i seguenti comandi, il modulo kernel *br_ netfilter* e alcune porte del firewall. Per eseguire un'installazione di test può essere utile disattivarlo completamente.

```
modprobe br_netfilter
echo '1' > /proc/sys/net/bridge/bridge-nf-call-iptables
firewall-cmd --permanent --add-port=6443/tcp
firewall-cmd --permanent --add-port=2379-2380/tcp
firewall-cmd --permanent --add-port=10250/tcp
firewall-cmd --permanent --add-port=10251/tcp
firewall-cmd --permanent --add-port=10252/tcp
firewall-cmd --permanent --add-port=10255/tcp
```

8.1.2 Installazione Docker

Per quanto riguarda la versione specifica di Docker, si assume che non sia presente alcuna versione preinstallata sulla macchina. Inoltre si fa riferimento, nello specifico, alla versione 18.03.1 community edition per un'architettura a 64 bit. Si installano quindi i requisiti necessari e la versione specifica.

```
yum install -y yum-utils device-mapper-persistent-data lvm2
yum install --setopt=obsoletes=0 \
      docker-ce-18.03.1.ce-1.el7.centos.x86_64 \
      docker-ce-selinux-18.03.1.ce-1.el7.centos.noarch
```

Prima di eseguire il servizio Docker sul nodo, può essere utile creare un file di configurazione per settare alcuni parametri come un proxy aziendale o un indirizzo locale di un registro. Si crea, dunque, una cartella per il servizio con percorso */etc/systemd/system/docker.service.d* e un file di configurazione `.conf` al suo interno. Tramite la voce "Service" si definiscono le variabili d'ambiente utilizzate dal servizio Docker:

```
[Service]
Environment= "HTTP_PROXY=http://<INDIRIZZO_PROXY>:8080/"
"HTTPS_PROXY=https://<INDIRIZZO_PROXY>:443/"
"NO_PROXY=localhost,127.0.0.1,<INDIRIZZO_LOCALE>"
```

Una volta che tutte le impostazioni sono state definite, si può avviare Docker e verificarne la versione e lo stato con i seguenti comandi:

```
systemctl daemon-reload
systemctl restart docker && systemctl enable docker
docker version
systemctl status docker
```

L'output sul terminale sarà il seguente.

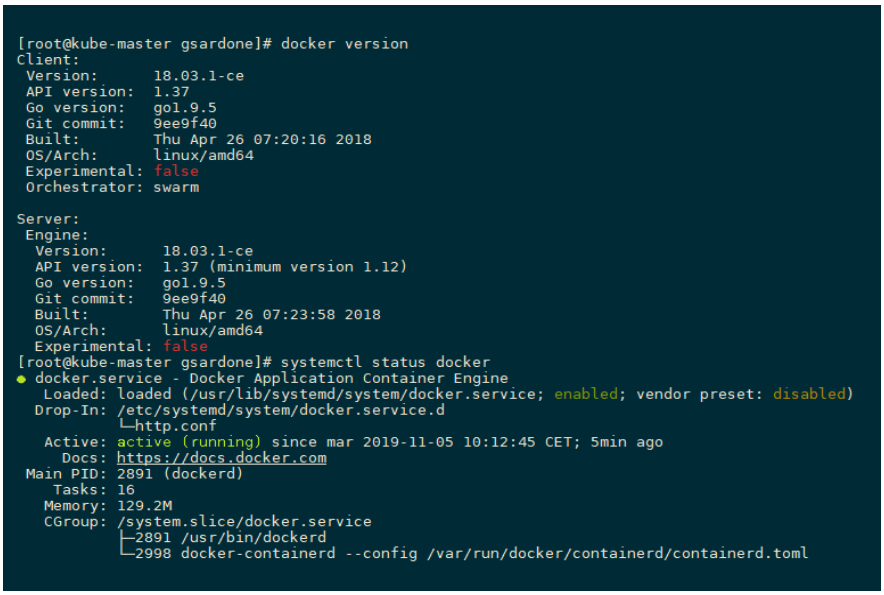


Figura 8.1: *Stato del servizio Docker.*

8.1.3 Installazione Kubernetes

Per prima cosa, bisogna aggiungere i riferimenti alla repository di kubernetes creando o modificando il file in */etc/yum.repos.d/kubernetes.repo* aggiungendo le seguenti impostazioni.

```
[kubernetes]
name= Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/
    kubernetes-el7-x86_64"
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
```

Come per Docker, si assume che non vi sia installata sulla macchina alcuna versione kubernetes. Devono essere installati, nell'ordine, Kubelet, ovvero il servizio di kubernetes che andrà in esecuzione sulla macchina, Kubectl, la console a linea di comando e Kubeadm, la console amministrativa. Tramite i seguenti comandi

si procede, dunque, all'installazione della versione 1.14.

```
yum install kubelet-1.14.8-0
yum install kubectl-1.14.8-0
yum install kubeadm-1.14.8-0
```

A questo punto bisogna assicurarsi che Docker e Kubernetes condividano lo stesso Control Group, altrimenti l'interazione tra loro non andrà a buon fine. Per far ciò basta assegnare lo stesso control group di Docker impostandolo nel file di configurazione di Kubernetes tramite il comando:

```
sed -i 's/cgroup-driver=systemd/cgroup-driver=cgroupfs/g' \
    /usr/lib/systemd/system/kubelet.service.d/10-kubeadm.conf
```

Un ulteriore accorgimento prevede di forzare kubelet a lavorare sull'IP primario dei nodi, prevenendo alcuni errori occasionali che si verificano quando si cerca di consultare i log dell'applicazione. Per evitarlo, occorre modificare il file */usr/lib/systemd/system/kubelet.service.d/10-kubeadm.conf*, inseredendo le righe:

```
Environment="KUBELET_CERTIFICATE_ARGS=--rotate-certificates=true
  --cert-dir=/var/lib/kubelet/pki
Environment="KUBELET_EXTRA_ARGS=--node-ip=<ip_del_nodo>"
```

Terminata l'installazione, effettuata su ogni nodo del cluster, rimane da inizializzare il nodo master e unire il nodo worker al cluster. Il comando da lanciare sul master è:

```
kubeadm init --pod-network-cidr=<INDIRIZZO_SOTTORETE_POD>
  --apiserver-advertise-address=<INDIRIZZO_SERVER_API>
```

Le varie opzioni consentono di bypassare alcune configurazioni di default. Nell'installazione sono stati impostati un range di indirizzi IP per la rete interna dei pod (pod-network-cidr) e l'indirizzo su cui è in ascolto il server api (apiserver-advertise-address). Al termine, verrà stampato a console un token con cui sarà possibile aggiungere i vari nodi worker al cluster. Per concludere, inizializzare il client sulla macchina tramite i comandi:

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

Una volta eseguito tale comando sul master, con il token stampato potrà essere eseguito il join del nodo worker digitando il comando:

```
kubeadm join --token <TOKEN_DEL_CLUSTER>
  --discovery-token-ca-cert-hash <DIGEST_CA_CERT> <MASTER_IP>:6443
```

8.2 Installazione CNI: Weave

Weave net è un servizio di rete per container Docker, sfruttabile quindi anche per pod Kubernetes, che permette la comunicazione tra pod. Per la sua installazione è necessario scaricare il file yaml di configurazione dalla repository ufficiale, all'indirizzo *"https://cloud.weave.works/k8s/v1.10/net.yaml"* per versioni kubernetes più recenti della versione 1.9. Il file contiene i dettagli di tutti gli elementi necessari al funzionamento di Weave, sottoforma di oggetti Kubernetes: ClusterRole, ClusterRolebinding, ServiceAccount e Daemonset. Nella specifica del Daemonset è possibile trovare le caratteristiche dei pod di Weave che verranno creati. In questa sezione è necessario aggiungere una variabile d'ambiente utilizzata dal container weave per definire il range degli indirizzi ip da assegnare ai pod, precisamente sotto il campo `spec.template.spec.containers.env`. Tale range non deve sovrapporsi con gli indirizzi di rete Docker o quelli configurati in fase di inizializzazione del cluster Kubernetes.

```
- name:  IPALLOC_RANGE
  value:  <IP_SUBNET>
```

Una volta confermata l'assenza di sovrapposizioni sulla sottorete scelta, si può procedere con il comando:

```
kubectl create -f net.yaml
```

Verranno così instanziate sul cluster tutte le risorse necessarie a Weave net e, dopo alcuni minuti, è possibile vedere come i nodi sono ora in stato di ready con il servizio kubelet in stato di running.

```
[root@kube-master gsardone]# kubectl get nodes
NAME          STATUS   ROLES    AGE     VERSION
kube-master   Ready    master   3h25m   v1.14.8
kube-worker   Ready    <none>   3h21m   v1.14.8

[root@kube-master gsardone]# systemctl status kubelet
● kubelet.service - kubelet: The Kubernetes Node Agent
   Loaded: loaded (/usr/lib/systemd/system/kubelet.service; enabled; vendor preset: disabled)
  Drop-In: /usr/lib/systemd/system/kubelet.service.d
           └─10-kubeadm.conf
   Active: active (running) since mar 2019-11-05 13:15:24 CET; 3h 26min ago
     Docs: https://kubernetes.io/docs/
 Main PID: 13271 (kubelet)
    Tasks: 18
   Memory: 83.4M
   CGroup: /system.slice/kubelet.service
           └─13271 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf ...
```

Figura 8.2: *Stato del servizio Kubelet.*

8.3 Configurazione Docker Registry

8.3.1 Creazione Registro

Per l'installazione di un registro locale, deve essere disponibile un file system dedicato alla storicizzazione delle immagini docker (eg. /DATA). Come primo step è necessario installare i tool necessari a creare un'autenticazione di tipo Basic, username-password, con apache e creare le cartelle per salvare i dati di autenticazione. Eseguire quindi i comandi seguenti per l'installazione e la creazione dei dati.

```
yum install httpd-tools
sudo mkdir -p /srv/registry/
sudo mkdir -p /srv/registry/security
sudo mkdir -p /srv/registry/data
:  | sudo tee /srv/registry/security/htpasswd
echo "password" | \
  sudo htpasswd -iB /srv/registry/security/htpasswd admin
```

Prima di utilizzare un registro bisogna generare un certificato contenente l'indirizzo IP e/o DNS, del registro che si utilizzerà, nel campo SubjectAlternativeName. Tale certificato può essere generato con programmi come OpenSSL. Una volta ottenuto il certificato, è possibile creare il servizio di Docker Registry in modalità container eseguendo il comando:

```
docker run -d -p 5000:5000 --name registry
-v /DATA:/var/lib/registry -v /srv/registry/security:/etc/security
-e REGISTRY_HTTP_TLS_CERTIFICATE=/etc/security/domain.crt
-e REGISTRY_HTTP_TLS_KEY=/etc/security/domain.key
-e REGISTRY_AUTH=htpasswd
-e REGISTRY_AUTH_HTPASSWD_PATH=/etc/security/htpasswd
-e REGISTRY_AUTH_HTPASSWD_REALM="Registry Realm"
-e REGISTRY_HTTP_ADDR=0.0.0.0:5000 --restart always registry:latest
```

Le opzioni indicate definiscono rispettivamente il certificato e la chiave da utilizzare, il metodo di autenticazione, l'indirizzo e la porta su cui ascoltare e la politica di riavvio del container.

8.3.2 Integrazione Kubernetes-Docker

Per poter utilizzare il Docker Registry dal cluster Kubernetes, è necessario inserire i certificati del Docker Registry fra i trusted dei nodi del cluster Kubernetes e creare un Docker Registry Secret come risorsa Kubernetes. Per ogni nodo del cluster bisognerà quindi validare il certificato del registro attraverso i prossimi comandi:

```
mkdir -p /etc/docker/certs.d/<URL_REGISTRY>
cp domain.crt /etc/docker/certs.d/<URL_REGISTRY>/ca.crt
cp domain.crt /etc/pki/ca-trust/source/anchors/<URL_REGISTRY>
update-ca-trust
```

Il segreto, invece, da utilizzare per effettuare il login al registro e che permetterà di effettuare le pull per gli altri nodi può essere creato digitando, sul master, il successivo comando.

```
kubectl create secret docker-registry <SECRET-NAME>
--docker-server=<URL_REGISTRY>
--docker-username=<USERNAME>
--docker-password=<PASSWORD> --docker-email=<EMAIL>
```

Il campo mail è puramente a scopo informativo e non verrà mai utilizzato. A questo punto il cluster è configurato e vi è possibile distribuire le applicazioni a monoservizi.

Capitolo 9

Caso d'uso

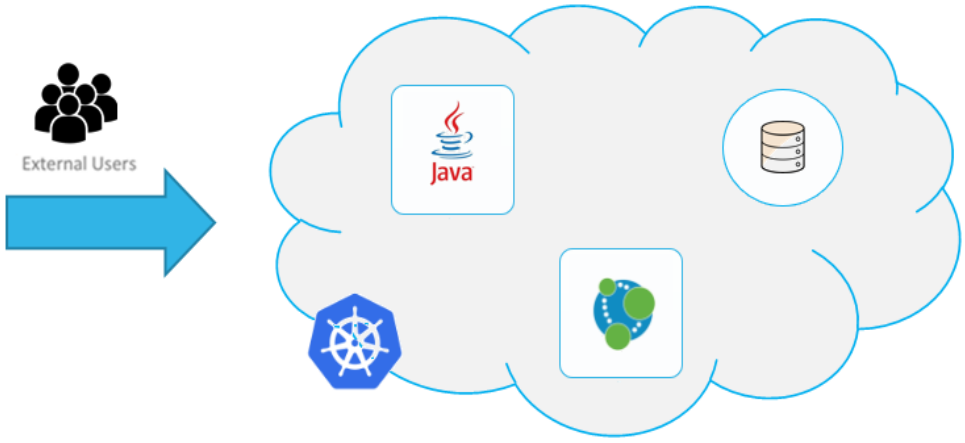


Figura 9.1: *Applicazione nel cloud kubernetes.*

L'applicazione utilizzata in questo studio è composta da tre microservizi, due dei quali esposti all'esterno come front-end e l'altro che agisce da database come back-end. L'applicazione, molto semplice, costituisce un sistema per la gestione di veicoli all'interno di un sistema di strade formate da segmenti, aree di servizio e gate connessi tra loro. Gli utenti di questo sistema possono ottenere indicazioni per una specifica destinazione all'interno del sistema, tenendo sempre traccia della posizione corrente. Per calcolare il percorso migliore il sistema sfrutta un servizio esterno, *neo4j*, adatto per la gestione di grafi orientati. Il veicolo richiede quindi l'ingresso al sistema indicando la destinazione da raggiungere, quest'ultimo effettua il controllo e il calcolo del percorso e inserisce il veicolo all'interno del sistema, tracciandolo.

9.1 Deploy nel cluster Kubernetes

Per distribuire l'applicazione all'interno del cluster bisogna prima creare un'immagine che deve eseguire il container, poi sarà possibile creare le risorse kubernetes corrispondenti. A partire quindi dal jar che contiene l'applicazione, si definisce il dockerfile:

```
FROM openjdk:8-jdk-alpine
VOLUME /tmp
ARG JAR_FILE=SpringJerseyRNS-0.1.0.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

Figura 9.2: *Dockerfile applicazione RoadNavigationSystem.*

Una volta che il dockerfile è definito, bisogna creare l'immagine ed inserirla nel registro condiviso, rendendola accessibile a tutti i nodi del cluster. Assumendo di aver fatto già il login al registro, è necessario digitare i seguenti comandi:

```
docker build -t rnsapp:v1 -f <PATH_DOCKERFILE>
docker tag rnsapp:v1 <URL_REGISTRY>/rnsapp:v1
docker push <URL_REGISTRY>/rnsapp:v1
```

Ora che l'immagine è pronta, si può passare alla creazione delle risorse kubernetes. Per gli altri due servizi, neo4j e il db, si usano delle immagini già fornite da Docker. Attraverso dei file di configurazione `.yaml` si crea un deploy e un servizio per ogni front-end, indicando le rispettive porte di ascolto. Per questioni di sicurezza che verranno analizzate in seguito, si creano anche due service account, uno per il front-end e uno per il back-end.

```
kubectl create serviceaccount "rns-system"
kubectl create serviceaccount "mongo-sa"
kubectl create -f <PATH_DEPLOY_YAML>
kubectl create -f <PATH_SERVICE_YAML>
```

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: "rnsapp-v1"
  labels:
    app: rnsapp
    version: v1
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: rnsapp
        version: v1
    spec:
      serviceAccountName: rns-system
      containers:
      - name: rnsapp
        image: 192.168.56.101:5000/rnsapp:v1
        imagePullPolicy: "IfNotPresent"
        ports:
        - containerPort: 8080
          name: dash
      imagePullSecrets:
        - name: reg-secret
```

(a)

```yaml
apiVersion: v1
kind: Service
metadata:
  name: rnsapp
  labels:
    app: rnsapp
    service: rnsapp
spec:
  ports:
  - port: 8080
    name: "tcp-dash"
    protocol: TCP
    targetPort: 8080
  selector:
    app: rnsapp
```

(b)

Figura 9.3: *Deploy (a) con Service allegato (b) per l'applicazione RoadNavigationSystem.*

Allo stesso modo, ci saranno un deploy e un servizio per il servizio del calcolo dei percorsi. Come database ne è stato scelto uno non relazionale, mongodb, per semplicità in fase di configurazione applicativa. Si procede, dunque, con la configurazione di un volume persistente che memorizzerà i veicoli e le entità presenti nel sistema, tramite `storageclass`, `persistentvolumeclaim` e `persistentvolume`. Nel deploy, invece, per sfruttare il volume persistente bisognerà indicare il nome e il percorso per effettuare il bound tra volume all'interno del container e volume esterno sulla macchina.

```yaml
spec:
  serviceAccountName:  mongo-sa
  containers:
    volumeMounts:
    - name: hostvol
      mountPath: /home/gsardone/mongo/mongo-data/mongo1
  volumes:
  - name: hostvol
    persistentVolumeClaim:
      claimName: mongo-pv-claim
```

Figura 9.4: *Riferimenti al volume persistente da associare al deploy.*

```yaml
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: local-storage
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer
---

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongo-pv-claim
  labels:
    app: mongodb
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: local-storage
  resources:
    requests:
      storage: 1Gi
```

(a)

```yaml
kind: PersistentVolume
apiVersion: v1
metadata:
  name: task-pv-volume
  labels:
    type: local
spec:
  storageClassName: local-storage
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 1Gi
  hostPath:
    path: /home/gsardone/mongo/mongo-data/mongo1
    type: DirectoryOrCreate
```

(b)

Figura 9.5: *Richiesta di volume (a) e volume persistente (b) per il database MongoDB.*

Distribuita l'applicazione nel cluster, bisogna gestire tutte le comunicazioni tra i servizi e applicare delle regole per garantire un controllo degli accessi solo a determinati utenti. L'applicazione delle policy di routing e del controllo degli accessi verrà configurata tramite la piattaforma Istio.

Capitolo 10

Configurazione Istio

Una volta installati sul cluster sia Kubernetes che Docker, è possibile installare Istio per effettuare tutti i controlli sulla rete dei servizi e applicare le diverse policy di rete e sicurezza. L'installazione può essere personalizzata a seconda delle risorse e delle componenti che si vogliono installare nel cluster. Istio offre una serie di possibili installazioni, in base alle esigenze degli operatori, riassunte nella seguente tabella.

	default	demo	minimal	demo-auth
Profile filename	values.yaml	values-istio-demo.yaml	values-istio-minimal.yaml	values-istio-demo-auth.yaml
Core components				
istio-citadel	X	X		X
istio-egressgateway		X		X
istio-galley	X	X		X
istio-ingressgateway	X	X		X
istio-nodeagent				
istio-pilot	X	X	X	X
istio-policy	X	X		X
istio-sidecar-injector	X	X		X
istio-telemetry	X	X		X
Addons				
grafana		X		X
istio-tracing		X		X
kiali		X		X
prometheus	X	X		X
Control Plane Security				X
Strict Mutual TLS				X

Figura 10.1: *Profili Istio.* [16]

Per una configurazione completa, l'installazione che è stata scelta è quella che fornisce tutte le componenti, compresi i criteri di sicurezza.

10.1 Installazione Istio

La versione che è stata installata è la 1.2.2 e si procede creando prima le risorse personalizzate di Istio e poi installando la piattaforma vera e propria. Una volta scaricata la versione dal link ufficiale con il comando

```
curl -L https://git.io/getLatestIstio | ISTIO_VERSION=1.2.2 sh -
```

si può procedere all'installazione tramite gli appositi comandi:

```
for i in install/kubernetes/helm/istio-init/files/crd*yaml; \
   do kubectl apply -f $i; done
kubectl apply -f install/kubernetes/istio-demo-auth.yaml
```

Dopo alcuni minuti sarà possibile verificare lo stato dei pod appena distribuiti all'interno di un namespace, istio-system, appena creato.

```
kube-master ~ # kubectl get pod -n istio-system
NAME                                    READY   STATUS      RESTARTS   AGE
grafana-6fb9f8c5c7-brwnc                1/1     Running     0          13d
istio-citadel-68c85b6684-5mtq2          1/1     Running     0          13d
istio-cleanup-secrets-1.2.2-4t7wd       0/1     Completed   0          13d
istio-egressgateway-54869665bc-dznw9    1/1     Running     0          13d
istio-galley-84d8cfd75d-zq59r           1/1     Running     0          13d
istio-grafana-post-install-1.2.2-dhbkr  0/1     Completed   0          13d
istio-ingressgateway-545dd78c-rgl58     1/1     Running     0          13d
istio-pilot-7d585b69cf-jjdbm            2/2     Running     0          13d
istio-policy-87ffb5b96-96hgj            2/2     Running     2          13d
istio-security-post-install-1.2.2-fphrh 0/1     Completed   0          13d
istio-sidecar-injector-66549495d8-r6rtq 1/1     Running     0          13d
istio-telemetry-f9b4b5cc-5hst2          2/2     Running     0          2d16h
istio-tracing-5d8f57c8ff-q2768          1/1     Running     0          13d
kiali-7d749f9dcb-vwjhv                  1/1     Running     0          13d
prometheus-776fdf7479-cdkbz             1/1     Running     0          13d
```

Figura 10.2: *Pod Istio.*

Con questo tipo di installazione è abilitata la funzione di auto-injection, ovvero applicando la label `"istio-injection=enabled"` su un certo namespace, ad ogni deploy effettuato all'interno del namespace etichettato verranno inseriti due container, *istio-init* e *istio-proxy*, all'interno di ogni pod appartenente al deploy. Come passo successivo, infatti, si etichetta il namespace e poi si effettua il deploy dell'applicazione.

```
kubectl label namespace <namespace> istio-injection=enabled
kubectl create -n <namespace> -f <app-spec>.yaml
```

Una volta che Istio e l'applicazione sono stati installati, si può notare che i pod all'interno del namespace etichettato contengono due container attivi, il container dell'applicazione stessa e il container istio-proxy, l'envoy allegato come sidecar.

10.2 Traffic Routing

Prima di iniziare a configurare alcune delle possibili regole di routing all'interno di Istio, bisogna creare le risorse capaci di gestire tali regole. In questo studio sono stati utilizzati un Gateway, un VirtualService applicato su due host e una DestinationRule per ogni servizio presente all'interno della mesh. Dunque, si creano le risorse necessarie alla gestione del traffico:

(a) (b) (c)

Figura 10.3: *Gateway (a) con VirtualService collegato (b) e DestinationRules (c) definite per il servizio.* [16]

Facendo il deploy di tali risorse con i semplici comandi kubernetes, si rendono accessibili dall'esterno il servizio *rnsapp* e i workload corrispondenti. Allo stesso modo si possono aggiungere le righe nel virtual service e le corrispettive destination rules per gli altri servizi come *mongodb* e *neo4j*.

10.2.1 Gateway

Il gateway è il punto di ingresso nella service mesh che riceve e gestisce le connessioni HTTP/TCP in entrata e in uscita. Il gateway è configurabile tramite le sezioni:

- `selector:` permette il bind della risorsa che agisce da gateway, quella di default di Istio o una personalizzata dall'utente, con la configurazione specificata.
- `servers:` parte in cui per ogni host vengono indicati i punti di accesso.

Di seguito viene mostrato un semplice esempio di gateway che sfrutta il gateway di default di Istio:

```
servers:
  - hosts:
      - '*'
    port:
      name: http-rns
      number: 80
      protocol: HTTP
    tls:
      httpsRedirect: true
  - hosts:
      - '*'
    port:
      name: https-rns
      number: 443
      protocol: HTTPS
    tls:
      mode: MUTUAL
      serverCertificate: /etc/certs/cert-chain.pem
      privateKey: /etc/certs/key.pem
      caCertificates: /etc/certs/root-cert.pem
selector:
  istio: ingressgateway
```

Figura 10.4: *Configurazione Gateway.*

Come si nota dalla figura, nella parte dei server viene definito un elenco di servizi per cui Istio deve gestire il traffico. I servizi vengono indicati tramite la voce `hosts` che include una lista di nomi DNS con presenza o meno di wildcard. Per ogni insieme di host viene poi definita la porta di ascolto caratterizzata da nome, numero e protocollo rispettivamente attraverso le voci `name`, `number` e `protocol`. Infine è possibile specificare un criterio di sicurezza tramite la vove `tls` con le opzioni corrispondenti:

- `httpsRedirect`, ridireziona il traffico http su una connessione sicura mandando un messaggio http redirect 301.
- `mode`, specifica il tipo di connessioni sicure, *SIMPLE* per standard TLS, *MUTUAL* per mutual TLS, *ISTIO_MUTUAL* per mutual TLS con certificati generati e gestiti da Istio, *PASSTHROUGH* per connessioni che non saranno terminate sul gateway.
- `serverCertificate`, obbligatorio se il modo è *SIMPLE* o *MUTUAL*, indica il percorso del certificato.
- `privateKey`, obbligatorio se il modo è *SIMPLE* o *MUTUAL*, indica il percorso della chiave privata.
- `caCertificates`, obbligatorio se il modo è *MUTUAL*, indica il percorso del certificato autoritario che viene usato per confermare il certificato del client.

Altre voci opzionali consentono di indicare la versione del protocollo tls, min e max, la cipherSuites da utilizzare o una lista di subject alternative names per confermare l'identità del client.

10.2.2 Virtual Service

Il virtual service rappresenta una serie di regole di routing da applicare al traffico di rete. Questa risorsa è caratterizzata da cinque voci fondamentali:

- `hosts:` lista di nomi DNS, completi o con wildcard, e/o indirizzi IP che indicano tutti i servizi verso cui si può inoltrare il traffico.
- `gateways:` nomi dei gateway ai quali viene associato l'insieme di regole di routing indicate nel virtual service.
- `http:` lista ordinata di regole per traffico di tipo http, http2, grpc, tls e https terminati.
- `tls:` lista ordinata di regole per traffico di tipo tls e https non terminati.
- `tcp:` lista ordinata di regole per traffico di tipo tcp.

Come si può notare dal file di configurazione del virtual service, figura 10.3b, è configurata una regola per il traffico http. Nella sezione `http` è infatti possibile configurare una serie di regole di routing che agiranno tutte sul traffico di tipo http. In questo caso, con l'opzione `match` viene prelevata la stringa dell'uri richiesto come parametro e viene controllato che il prefisso dell'uri contenga un determinato valore. Se il valore è quello specificato, la regola verrà confermata e si raggiungerà la destinazione indicata nel campo host. Essa è una destinazione virtuale ovvero quella indicata nella destination rule. Tramite la destination rule, invece, viene fatto il bound tra destinazione virtuale e quella reale, il workload corrispondente. In questo caso, sia per quella virtuale che per quella reale non vengono eseguite politiche di load balancing, quindi Istio applica la politica di default ROUND_ROBIN. La regola match, presente all'interno del virtual service, oltre al campo `uri` può verificare anche altri dettagli di una richiesta come un header, utilizzando il campo `headers`.

```
http:
- match:
  - headers:
      key :
        exact: v1
  route:
    - destination:
        host: rnsapp
        port:
          number: 8080
        subset: v1
```

Figura 10.5: *Controllo header nella regola match.*

Nella figura viene mostrato il controllo del pacchetto http. Precisamente si verifica che ci sia l'header *key* con valore esattamente pari a *v1*. Ogni campo compreso nella regola match può essere verificato attraverso i predicati:

- `exact:"value"`, per un esatto match sulla stringa.
- `prefix:"value"`, per un match controllato sul prefisso della stringa.
- `regex:"value"`, per controllare un'espressione regolare.

Sempre all'interno della voce match è possibile riscrivere interamente o parte dell'uri ricercato o l'authority host utilizzando l'opzione `rewrite`. In questo modo, ad esempio, si possono nascondere al front-end parte dei percorsi utilizzati nel back-end della rete mesh.

```
http:
- match:
    - uri:
        prefix: /browser/
  rewrite:
    uri: /systemrns/rnsapp
  route:
    - destination:
        host: rnsapp
        port:
          number: 8080
        subset: v1
```

Figura 10.6: *Sovrascrittura dell'uri di un pacchetto HTTP.*

Il campo headers della regola match può essere facilmente confuso con la regola omonima. Con una corretta configurazione di tale regola si possono anche rimuovere, aggiungere o sovrasscrivere header già presenti. Gli header possono essere modificati sia in fase di richiesta che di risposta, con i corrispondenti campi `request` e `response`. I predicati seguenti permettono di manipolare i campi dell'intestazione del pacchetto http:

- `set:` accetta un tipo `‘map<string, string>’` come parametro e imposta o sostituisce le header tramite la coppia chiave-valore.
- `add:` accetta un tipo `‘map<string, string>’` come parametro e aggiunge le header tramite la coppia chiave-valore.
- `remove:` accetta un tipo `‘string[]’` come parametro e rimuove le header specificate.

La regola header può essere usata anche più volte, come si vede nella figura seguente, in cui si inserisce l'header *key* in fase di richiesta e si rimuove l'header *rnsapp* in fase di risposta.

```
http:
- headers:
    request:
      set:
        key : v1
  route:
    - destination:
        host: rnsapp
        port:
          number: 8080
        subset: v1
      headers:
        response:
          remove:
          - rnsapp
```

Figura 10.7: *Gestione di header HTTP in fase di richiesta e risposta.*

Un'altra opzione utile da aggiungere può essere quella di ritornare un codice di errore o di introdurre volontariamente un malfunzionamento nella rete mesh. Istio permette di inserire un errore nella richiesta causando un aborto o un ritardo. In questo modo è possibile testare la robustezza di un cluster proprio in caso di malfunzionamenti e ritardi.

```
- fault:
    abort:
      httpStatus: 555
      percent: 100
  route:
    - destination:
        host: rnsapp
        port:
          number: 8080
        subset: v1
```

```
- fault:
    delay:
      fixedDelay: 5s
      percent: 1
  route:
    - destination:
        host: rnsapp
        port:
          number: 8080
        subset: v1
```

Figura 10.8: *Errori di tipo fault-abort e fault-delay Istio.*

In questo modo, aggiungendo in fondo al file di configurazione del Virtual Service eventuali errori da iniettare, tutto il traffico che non soddisferà alcuna regola sarà affetto da un ritardo o da un aborto. Le regole possono anche essere interpolate tra loro, ad esempio inserendo nel campo fault la regola match per iniettare degli errori solo per un sottoinsieme di traffico. Nei file di configurazione tutte le operazioni di controllo vengono introdotte dal carattere `"-"`. In un elenco di operazioni da effettuare, le condizioni vengono valutate con un OR logico se introdotte con il carattere `"-"`, mentre con un AND logico se non è presente alcun carattere. Più volte si è parlato di test A/B e di deploy a step bilanciando il traffico in modo controllato da una versione all'altra. Ciò viene permesso da Istio applicando la voce `weight` all'interno della sezione route, quella in cui viene anche definita la destinazione del traffico. In questo modo si può controllare una piccola porzione di traffico su una nuova versione dell'applicazione tenendo, però, buona parte del traffico sulla precedente versione stabile.

```
route:
  - destination:
      host: rnsapp
      port:
        number: 8080
      subset: v1
    weight: 80
  - destination:
      host: rnsapp
      port:
        number: 8080
      subset: v2
    weight: 20
```

Figura 10.9: *Definizione del bilanciamento pesato del traffico Istio.*

Quanto appena visto è valido solo per il traffico http, mentre per il traffico di tipo tcp o tls è presente un set di regole molto ridotto. Per quanto riguarda il traffico tcp, è possibile configurare una regola `match` che ha meno opzioni rispetto a quella http. Sostanzialmente attraverso tcp-match è possibile controllare la porta destinazione del traffico che poi, come per il traffico http, verrà direzionato verso una destinazione avente come porta quella del container esposto. Infine, per traffico di tipo tls, oltre a controllare la porta destinazione del traffico si può controllare il *ServerNameIndicator* tramite la voce `sniHosts`. Attraverso quest'ultima voce è possibile utilizzare wildcard per raggruppare una serie di hosts che però deve essere nello stesso dominio degli host del virtual service.

10.2.3 DestinationRule

Le destination rules vengono applicate sulle destinazioni reali quando il routing è stato già effettuato. Con tali risorse si può effettuare una piccola gestione del traffico locale arrivato a destinazione. Una destination rule è costituita da tre voci principali:

- `host:` dichiara il servizio che gestirà il traffico.
- `trafficPolicy:` configura le operazioni per la gestione locale del traffico.
- `subset:` indica dei sottolivelli dell'applicazione, come una versione, in cui è possibile sovrascrivere le policy di traffico.

Le policy di traffico permettono di verificare e impostare alcuni criteri per le connessioni con la voce `connectionPool`, ma anche di eseguire bilanciamento con la voce `loadBalancer`. Con la prima è possibile specificare il numero di connessioni massime, il timeout o il keepalive per connessioni tcp, come pure il numero di retry, di richieste totali o pendenti al backend per il traffico http. Con la seconda, invece, è possibile effettuare bilanciamento in maniera semplice con i modi *ROUND_ROBIN*, *LEAST_CONN*, *RANDOM*, o complessa utilizzando l'hash di alcuni campi come il nome di un header, il nome, il percorso o il time to live di un cookie. Sempre nella sezione delle policy di traffico è possibile specificare delle impostazioni da usare per connessioni di tipo tls quando un client vuole comunicare con l'host indicato. Come per il gateway, vengono definiti il modo e il percorso dei certificati, in questo caso del client, per effettuare un corretto handshake.

```
host: rnsapp.default.svc.cluster.local
trafficPolicy:
  connectionPool:
    http:
      http1MaxPendingRequests: 100
      http2MaxRequests: 1000
      maxRetries: 10
    tcp:
      connectTimeout: 30ms
      maxConnections: 100
      tcpKeepalive:
        time: 7200s
    tls:
      mode: MUTUAL
      clientCertificate: /etc/certs/myclientcert.pem
      privateKey: /etc/certs/client_private_key.pem
  loadBalancer:
    consistentHash:
      httpCookie:
        name: user
        ttl: 0s
```

Figura 10.10: *Definizione di una destination rule per servizio http, tcp e tls.*

Infine, con la voce `subsets` è possibile definire dei sottolivelli dell'applicazione tramite l'uso del comando `labels`. In questo modo si creano dei sottoinsiemi

grazie a delle etichette aggiunte in fase di deploy. I virtual service riescono a direzionare il traffico tra differenti versioni proprio grazie ai sottinsiemi definiti nelle destination rules.

```
host: rnsapp.default.svc.cluster.local
trafficPolicy:
  loadBalancer:
    simple: LEAST_CONN
subsets:
  - labels:
      version: v2
    name: v2
    trafficPolicy:
      loadBalancer:
        simple: ROUND_ROBIN
  - labels:
      version: v1
    name: v1
```

Figura 10.11: *Definizione di subset all'interno della destination rule.*

In figura viene mostrato come si definiscono per l'host *rnsapp* una versione *v1* e una *v2*, in cui di default viene applicato per tutti i wokload della versione v1 un blianciamento di tipo `LEAST_CONN`, mentre per quelli della versione v2 viene sovrascritto il bilanciamento di default utilizzandone uno di tipo `ROUND_ROBIN`.

10.3 Politiche di Sicurezza

La sicurezza in Istio permette l'autenticazione e la crittografia, ma anche l'applicazione di politiche per l'autorizzazione degli accessi. In questo caso si è analizzata la rete dei servizi interna, tralasciando connessioni già sicure provenienti dall'esterno. Si assume quindi l'uso di connessioni al gateway di ingresso con protocolli http o tcp non protetti in alcun modo. Si inizia, quindi, abilitando nella rete il tls, in modalità mutua autenticazione, con una politica Mesh.

```
apiVersion: authentication.istio.io/v1alpha1
kind: MeshPolicy
metadata:
  labels:
    app: security
    chart: security
    heritage: Tiller
    release: istio
  name: default
spec:
  peers:
  - mtls: {}
```

Figura 10.12: *Configurazione TLS nella service mesh.*

Contemporaneamente alla mutua autenticazione nella rete lato server, bisogna configurare i client affinché comunichino con mutual TLS. Affidandosi ad Istio per la gestione dei certificati, si può abilitare il tls in modo `ISTIO_MUTUAL` all'interno delle destination rules dei servizi presenti. Così facendo, nella rete sono configurate l'autenticazione e la confidenzialità e si procede con la configurazione dell'autorizzazione degli accessi tramite RBAC. Analogamente a quanto eseguito precedentemente, si imposta l'autorizzazione in tutta la rete mesh che, non avendo ancora configurato alcune politiche, bloccherà tutte le comunicazioni nella rete.

```yaml
apiVersion: "rbac.istio.io/vlalphal"
kind: ClusterRbacConfig
metadata:
  name: default
spec:
  mode: 'ON_WITH_INCLUSION'
  inclusion:
    namespaces: ["tesi-istio"]
```

Figura 10.13: *Configurazione Cluster RBAC policy nella service mesh.*

Con l'utilizzo di service role e service role binding si garantiscono accessi controllati ai servizi. Ad esempio si può utilizzare una regola per garantire l'accesso ai front-end solo da parte del gateway Istio.

```yaml
apiVersion: rbac.istio.io/vlalphal
kind: ServiceRoleBinding
metadata:
  name: bind-service-viewer
  namespace: tesi-istio
spec:
  roleRef:
    kind: ServiceRole
    name: service-viewer
  subjects:
  - properties:
      source.namespace: istio-system
```

(a)

```yaml
apiVersion: rbac.istio.io/vlalphal
kind: ServiceRole
metadata:
  name: service-viewer
  namespace: tesi-istio
spec:
  rules:
  - constraints:
    - key: destination.port
      values:
      - "8080"
    services:
    - rnsapp.tesi-istio.svc.cluster.local
  - constraints:
    - key: destination.port
      values:
      - "7474"
      - "7687"
    services:
    - neo4j.tesi-istio.svc.cluster.local
```

(b)

Figura 10.14: *ServiceRoleBinding (a) e ServiceRole (b) per l'autorizzazione dall'esterno.*

In questo modo tutto ciò che proviene dal namespace `istio-system`, figura 10.14a, potrà accedere ai servizi *rnsapp* e *neo4j* sulle porte specificate, figura 10.14b. Se l'accesso venisse eseguito da altri punti di ingresso, il sistema ritornerebbe l'errore "RBAC: access denied". All'interno della rete sono stati creati due utenti per separare i lati dell'applicazione. Uno, infatti, verrà utilizzato nelle comunicazioni *rnsapp-neo4j*, mentre l'altro verrà utilizzato nel back-end nelle comunicazioni verso il database *mongodb*.

```yaml
apiVersion: rbac.istio.io/vlalphal
kind: ServiceRoleBinding
metadata:
  name: bind-rns-viewer
  namespace: tesi-istio
spec:
  roleRef:
    kind: ServiceRole
    name: rns-viewer
  subjects:
  - properties:
      source.namespace: tesi-istio
    user: cluster.local/ns/tesi-istio/sa/rns-system
```

(a)

```yaml
apiVersion: rbac.istio.io/vlalphal
kind: ServiceRole
metadata:
  name: rns-viewer
  namespace: tesi-istio
spec:
  rules:
  - constraints:
    - key: destination.port
      values:
      - "7474"
      - "7687"
    services:
    - neo4j.tesi-istio.svc.cluster.local
  - constraints:
    - key: destination.port
      values:
      - "8080"
    services:
    - rnsapp.tesi-istio.svc.cluster.local
  - constraints:
    - key: destination.port
      values:
      - "27017"
    services:
    - mongodb.tesi-istio.svc.cluster.local
```

(b)

Figura 10.15: *ServiceRoleBinding (a) e ServiceRole (b) per l'autorizzazione dei servizi.*

In figura viene mostrato come il traffico di rete proveniente dal namespace `tesi-istio` del solo utente `rns-system` è abilitato verso i servizi e le porte indicate. Identica sarà l'operazione eseguita per l'utente `mongo-sa`, che potrà chiamare solo il servizio *rnsapp*. In questo modo lo scheletro dell'architettura di sicurezza è configurato ma, tramite altre risorse Istio quali `Rule`, `Instance` e `Handler`, è possibile personalizzare ed estendere alcune politiche. Risulta infatti possibile, ad esempio, autorizzare la comunicazione di una sola versione di un servizio verso un altro servizio, oppure creare whitelist o blacklist di versioni o indirizzi ip che possono o non possono chiamare alcuni servizi. La crd `Instance` è la risorsa che indica al Mixer quali attributi fornire all'handler per analizzare la richiesta. Tramite la voce `compiledTemplate` si indica il tipo di template Istio da utilizzare, *ListEntry*, *LogEntry*, *Kubernetes*, *CheckNothing* mentre, utilizzando `params`, si forniscono una serie di valori con coppie '`value:<ATTRIBUTE_NAME>`'.

La risorsa `Handler` è quella che esegue le operazioni vere e proprie. Associata al template dell'instance, la voce `compiledAdapter` indica il tipo di adapter da utilizzare per eseguire i controlli definiti nella sezione `params`. Le `Rule`, infine, sono composte principalmente da due voci:

- `match:` condizione per cui viene valutata la regola.
- `actions:` operazioni da eseguire quando la condizione viene soddisfatta. In questa sezione vengono definiti l'handler e l'instance che eseguiranno le operazioni di controllo.

```
apiVersion: config.istio.io/v1alpha2
kind: handler
metadata:
  name: whitelistversion
spec:
  compiledAdapter: listchecker
  params:
    overrides: ["v1","v2"]  # overrides provide a static list
    blacklist: false
---
apiVersion: config.istio.io/v1alpha2
kind: instance
metadata:
  name: appversion
spec:
  compiledTemplate: listentry
  params:
    value: source.labels["version"] | "default"
---
apiVersion: config.istio.io/v1alpha2
kind: rule
metadata:
  name: checkversion
spec:
  match: destination.labels["app"] == "mongodb" && source.labels["app"] == "rnsapp"
  actions:
    - handler: whitelistversion
      instances: [ appversion ]
```

Figura 10.16: *Definizione whitelist basata su label.*

Nell'esempio mostrato in figura si abilita la comunicazione del servizio *rnsapp* verso il servizio *mongodb* solo se la versione della sorgente è di tipo *v1* o *v2*. In questo caso, l'instance fornisce il valore della label "version" o "default" se la versione non è presente e l'handler verifica che la versione sia presente all'interno di una whitelist definita. Un altro caso può essere quello di bloccare il traffico di una sola versione senza configurare una blacklist. Usando un handler di tipo `denier`, infatti, è possibile rifiutare una comunicazione. Nella rule, quindi, verrà eseguito il controllo tramite specifici predicati e l'instance e l'handler si impegneranno a bloccare la chiamata.

```yaml
apiVersion: config.istio.io/v1alpha2
kind: handler
metadata:
  name: denyrnsappv2handler
spec:
  compiledAdapter: denier
  params:
    status:
      code: 7
      message: Not allowed
---
apiVersion: config.istio.io/v1alpha2
kind: instance
metadata:
  name: denyrnsappv2request
spec:
  compiledTemplate: checknothing
```

Figura 10.17: *Denier per un tipo di richiesta.*

L'instance in figura, avendo effettuato tutti i controlli nella rule, non fornisce alcun tipo di attributo all'handler che bloccherà automaticamente la chiamata.

10.4 Monitoring

Nella service mesh Istio, una volta che i servizi sono stati distribuiti e che le politiche di routing e di sicurezza sono state configurate, è utile conoscere lo stato dei servizi e il percorso del traffico che una richiesta genera. Il servizio *Kiali* offre dei meccanismi che aiutano a monitorare lo stato della rete, ad esempio visualizzando il grafo dei servizi e lo stato di una richiesta o di un servizio. Nella figura successiva viene illustrato il grafo dell'applicazione *RoadNavigationSystem*:

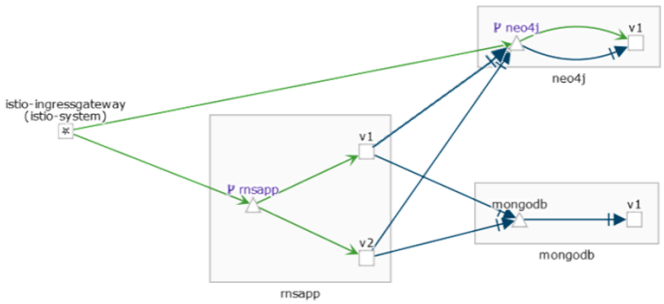


Figura 10.18: *Service Mesh rete Istio.*

Nella figura 10.18 viene mostrato il grafo dell'applicazione col dettaglio delle versioni di ogni servizio. Kiali permette la visualizzazione di un grafo meno dettagliato, mostrando solamente i deploy o i servizi. In caso di errori, Kiali fa notare che un servizio o un'applicazione non è in uno stato "healty" poiché ha riscontrato degli errori che possono essere stati generati dall'applicazione stessa o da una mal configurazione delle policy.

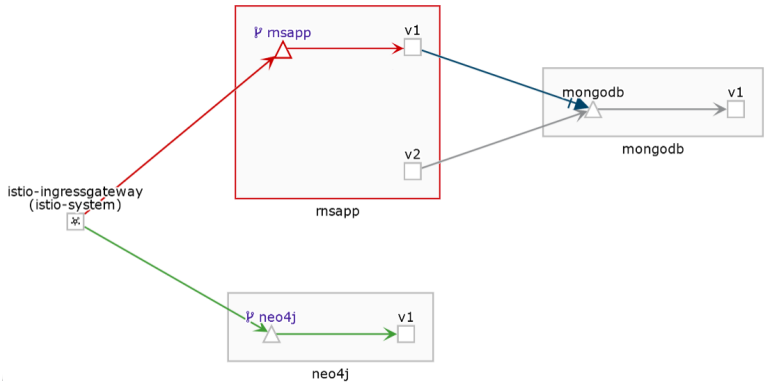


Figura 10.19: *Errore mostrato nel grafo Kiali.*

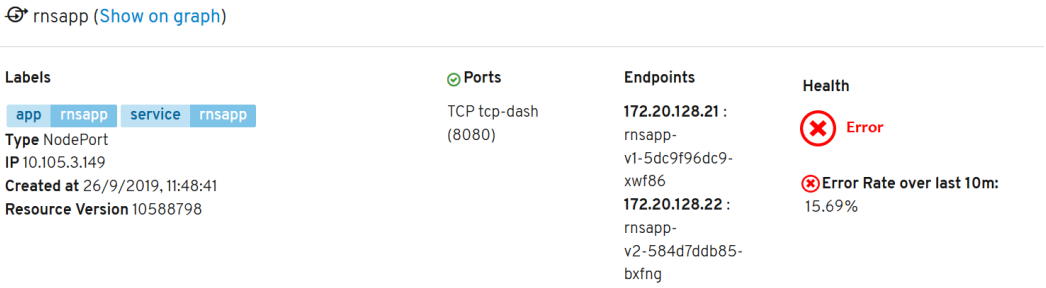


Figura 10.20: *Stato instabile del servizio.*

Con il tool Grafana è invece possibile monitorare e controllare i dati sul consumo delle risorse e sul traffico generato da ogni servizio. Grazie a questo tool si possono creare delle dashboard che utilizzano le metriche generate da Istio, in modo da raccogliere in un'unica finestra tutte le informazioni necessarie. Analizziamo ad esempio l'utilizzo della memoria dell'applicazione *RoadNavigationSystem*:

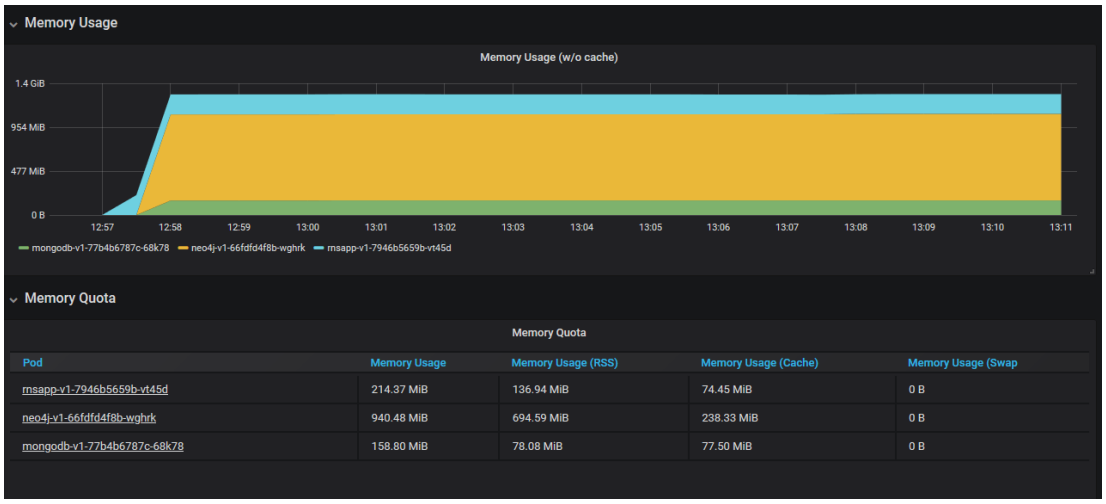


Figura 10.21: *Uso della memoria sistema RoadNavigationSystem.*

Come spiegato nel capitolo 7.2, Grafana recupera le metriche grazie al database Prometheus. Con Istio è possibile non solo eseguire delle query direttamente sul db, ma anche generare delle metriche personalizzate in modo da velocizzare il recupero dei dati. Le metriche vengono generate e gestite tramite `instance` e `handler`.

```
kind: instance
metadata:
  name: tcp-sentbytes
spec:
  compiledTemplate: metric
  params:
    value: connection.sent.bytes | 0 # uses a TCP-specific attribute
    dimensions:
      source_service: source.workload.name | "unknown"
      source_version: source.labels["version"] | "unknown"
      destination_service: destination.workload.name | "unknown"
      destination_version: destination.labels["version"] | "unknown"
      message: "my metrics sent"
    monitoredResourceType: '"UNSPECIFIED"'
---

apiVersion: config.istio.io/v1alpha2
kind: handler
metadata:
  name: tcphandler
spec:
  compiledAdapter: prometheus
  params:
    metrics:
    - name: tcp_sent_bytes # Prometheus metric name
      instance_name: tcp-sentbytes.instance.tesi-istio
      kind: COUNTER
      label_names:
      - source_service
      - source_version
      - destination_service
      - destination_version
      - message
```

Figura 10.22: *Metriche Prometheuts personalizzate auto-generate.*

Viene mostrato in figura come si crea la risorsa instance di tipo `metric` che genera i valori specificati a partire da un attributo della connessione TCP. L'handler di tipo `prometheus`, infine, esegue il conteggio delle richieste avendo i valori forniti dall'instance. Le metriche vengono generate in base alle condizioni definite nella rule associata all'instance e all'handler. Eseguendo quindi la query direttamente su prometheus, si può controllare il valore della metrica creata.

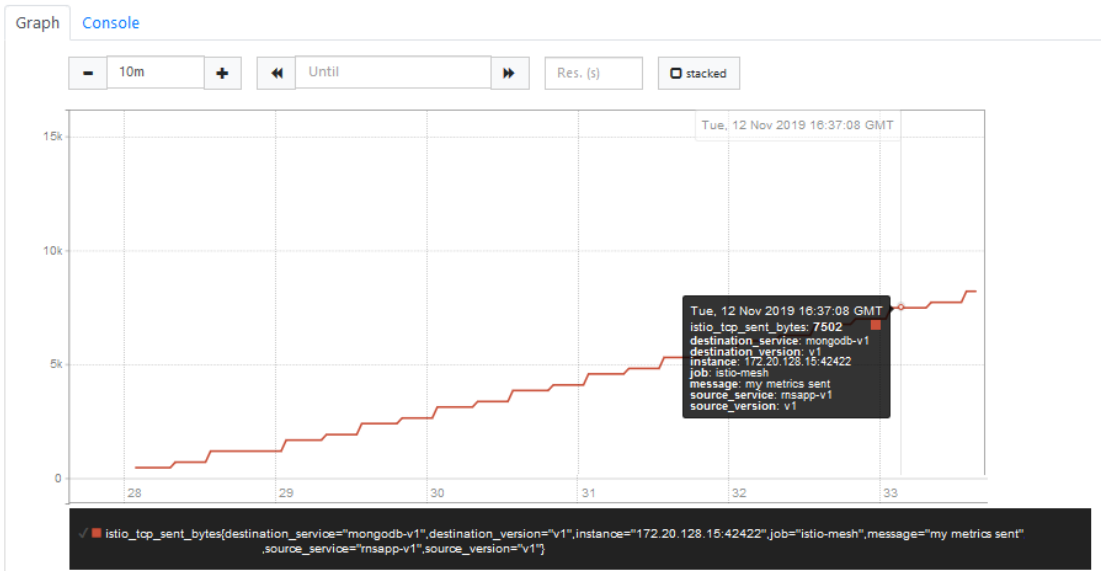


Figura 10.23: *Statistiche metriche analizzate su Prometheuts.*

10.5 Canary Deploy

Su Istio, grazie alle policy di rete nel virtual service basate sul bilanciamento pesato del traffico, può essere generato un test A/B con scalamento automatico delle applicazioni. Tramite virtual service e hpa, horizontal pod autoscaler, generando una grande quantità di traffico, verranno create automaticamente repliche dell'applicazione in base alla quantità di traffico assegnata ad ogni istanza. Ad esempio, si assume che il traffico diretto al servizio *rnsapp* sia inoltrato al 60% sulla versione *v1* e al 40% sulla versione *v2*. Infine si applica un autoscaler su entrambe le versioni del servizio con i seguenti comandi:

```
kubectl autoscale deployment rnsapp-v1 \
                    --cpu-percent=50 --min=1 --max=7
kubectl autoscale deployment rnsapp-v2 \
                    --cpu-percent=50 --min=1 --max=7
```

Iniziando a generare traffico elevato, si osserverà che verranno automaticamente create diverse repliche in base alle percentuali di traffico configurate nel virtual service.

```
kube-master ~ # kubectl get pod
NAME                          READY   STATUS    RESTARTS   AGE
mongodb-v1-f89698658-qldhx    2/2     Running   0          13d
neo4j-v1-68788c55d4-mwjcq     2/2     Running   0          13d
rnsapp-v1-5dc9f96dc9-xwf86    2/2     Running   0          13d
rnsapp-v1-5dc9f96dc9-66tkd    0/2     Pending   0          11m
rnsapp-v1-5dc9f96dc9-6jqcs    2/2     Running   0          12m
rnsapp-v1-5dc9f96dc9-8bdb9    2/2     Running   0          12m
rnsapp-v2-584d7ddb85-bxfng    2/2     Running   0          13d
rnsapp-v2-584d7ddb85-njrvw    0/2     Pending   0          9m32s
rnsapp-v2-584d7ddb85-r7jzs    0/2     Pending   0          9m32s
```

Figura 10.24: *Canary Deploy in Istio.*

Al termine del test, le repliche verranno rimosse grazie alla funzionalità dell'autoscaler, ritornando ad avere il numero minimo di repliche.

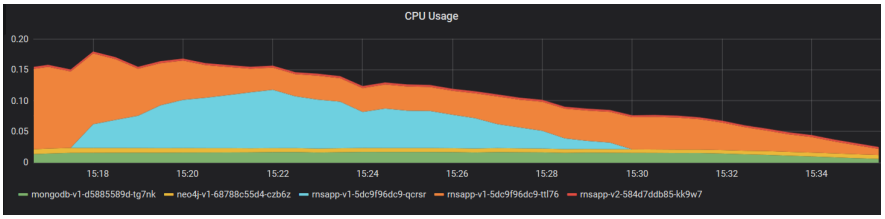


Figura 10.25: *Analisi Istio Canary Deploy su Grafana.*

Il grosso vantaggio di Istio in questo tipo di test è che non bisogna scalare a priori alcuna replica dell'applicazione. Configurando in modo appropriato nel virtual service le regole di traffic routing, le versioni saranno scalate valutando solo il volume del traffico indipendentemente dalle specifiche del deploy.

Capitolo 11

Analisi delle prestazioni

Sebbene con la piattaforma Istio vengono garantiti vantaggi facili da implementare, routing e bilanciamento personalizzato, autenticazione e controllo degli accessi, con zero modifiche al codice delle applicazioni, essa costituisce un layer aggiuntivo nel cluster. Istio aggiunge quindi dei requisiti al cluster sulle risorse come CPU, memoria RAM e spazio su disco, generando a volte anche quantità elevate di traffico nella rete. Si pensi alle metriche che vengono generate ad ogni richiesta per ogni servizio presente nella rete mesh. Per effettuare dei controlli sul carico di lavoro aggiuntivo da parte di Istio, si è utilizzato il tool Grafana per il controllo nel tempo dei consumi e il tool JMeter per generare una quantità elevata di traffico. Avendo poche risorse a disposizione per il cluster e non avendo un'applicazione con molti servizi, non è stato possibile effettuare uno stress test a grandi livelli, ma dai test effettuati si possono comunque notare delle particolarità nei diversi comportamenti. I test messi in pratica sul cluster comprendono una generazione di richieste con un diverso grado di parallelismo, uno, venti, cinquanta e ottanta thread paralleli. Per quanto riguarda l'uso della memoria, Istio aggiunge un piccolo overhead per gestire l'applicazione, ovvero 40-50 megabytes idonei all'allocazione del proxy Envoy associato ad ogni pod, aggiungendo altri 30-40 megabytes se viene abilitata la sicurezza con la gestione dei certificati e delle policy di accesso.

11.1 Memoria RAM

Per il consumo della memoria RAM non ci sono cambiamenti in base al numero di utenti che eseguono le richieste, dal momento che nei test l'applicazione non alloca oggetti in quanto l'allocazione sarebbe indipendente dal tipo di installazione di Istio. Le uniche modifiche si notano, quindi, solo in base alla presenza o meno di Istio nella service mesh.

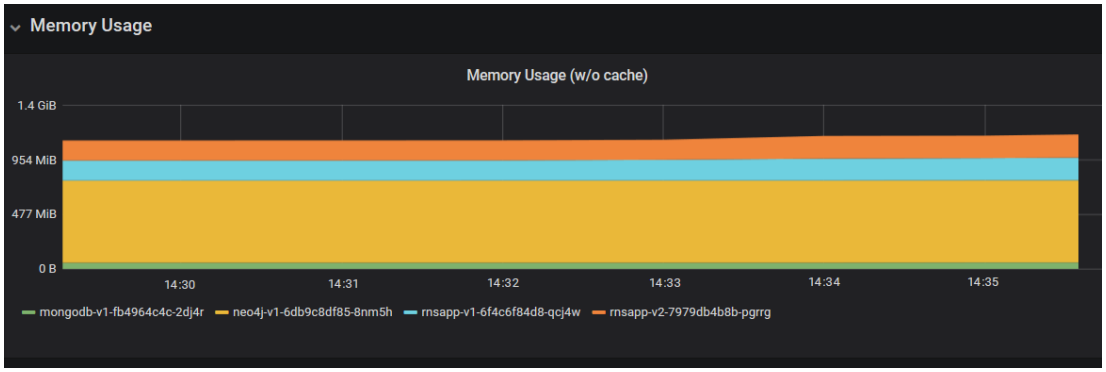


Figura 11.1: *Consumo Memoria sistema RoadNavigationSystem.*

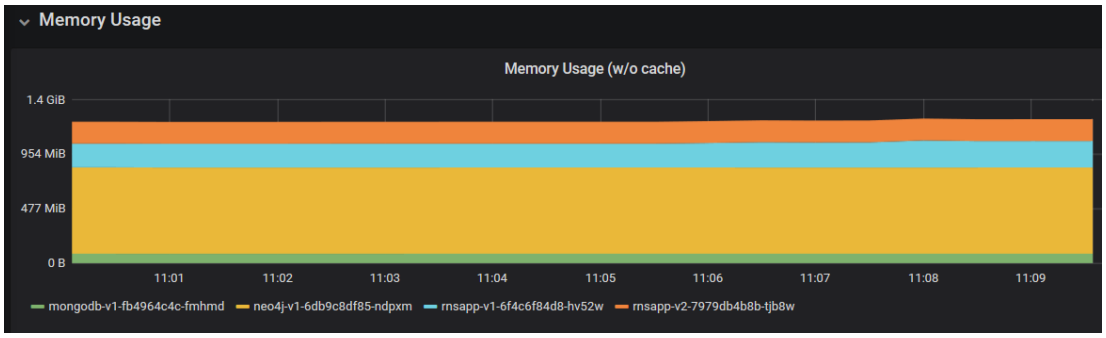


Figura 11.2: *Consumo Memoria sistema RoadNavigationSystem con Istio.*

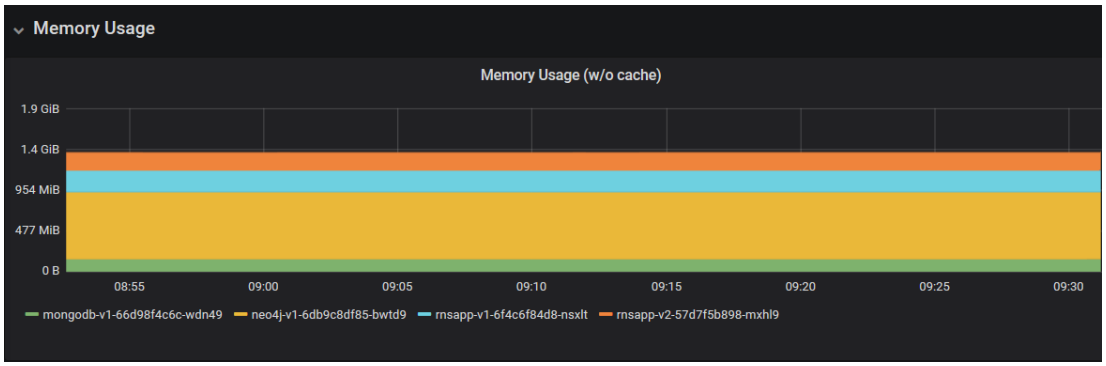


Figura 11.3: *Consumo Memoria sistema RoadNavigationSystem con Istio e sicurezza abilitata.*

11.2 Cosumo CPU

Per quanto riguarda il consumo della CPU si vede un netto aumento dal caso ad un utente a quelli multi utenti, tuttavia l'andamento del consumo della CPU rimane costante nel tempo. Con Istio si aggiunge, infatti, un aspetto computazionale legato ai compiti che esegue il proxy Envoy come l'handshake in una comunicazione tls tra servizi. La metrica seguente è stata utilizzata per l'analisi e lo sviluppo dei grafici:

```
sum(namespace_pod_container:container_cpu_usage_seconds_total:
    sum_rate{cluster="$cluster", namespace="$namespace"}) by (pod)
```

1 User

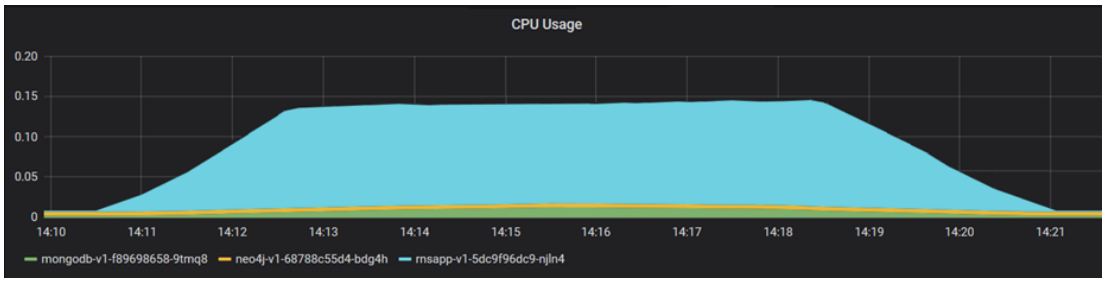


Figura 11.4: *Consumo CPU sistema RoadNavigationSystem.*

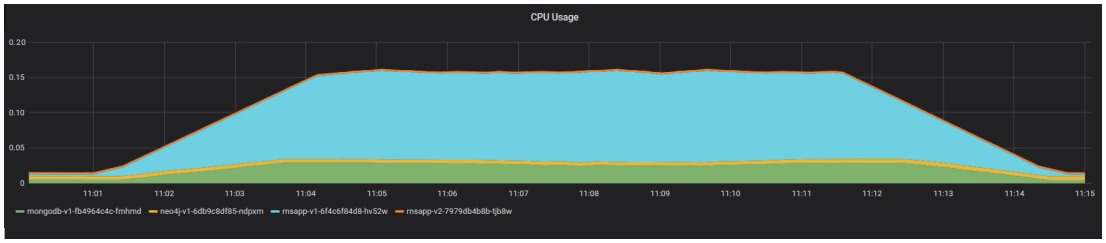


Figura 11.5: *Consumo CPU sistema RoadNavigationSystem con Istio e sicurezza abilitata.*

20 User

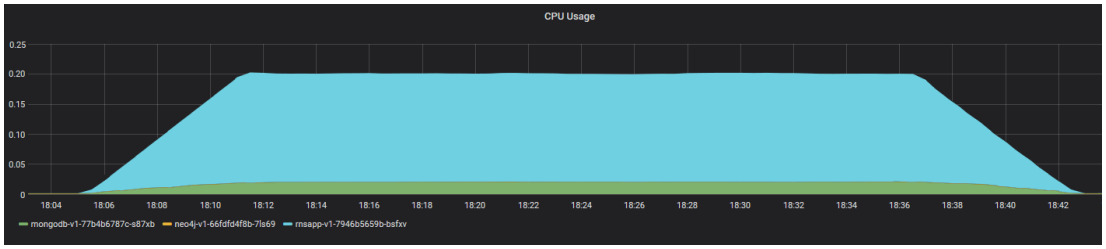


Figura 11.6: *Consumo CPU sistema RoadNavigationSystem.*

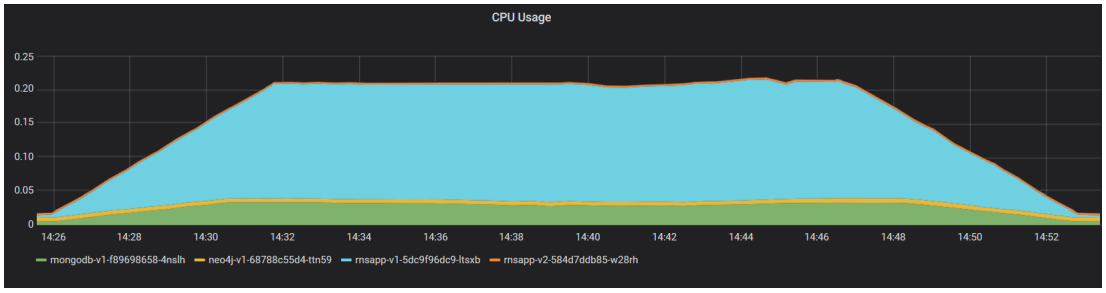


Figura 11.7: *Consumo CPU sistema RoadNavigationSystem con Istio e sicurezza abilitata.*

50 User

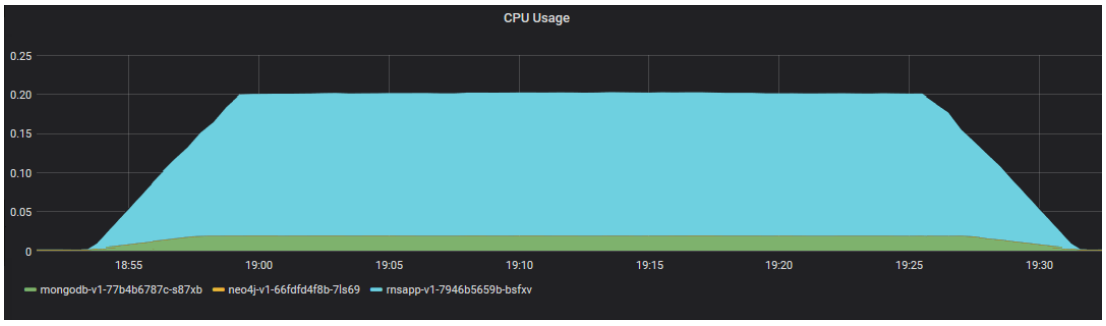


Figura 11.8: *Consumo CPU sistema RoadNavigationSystem.*

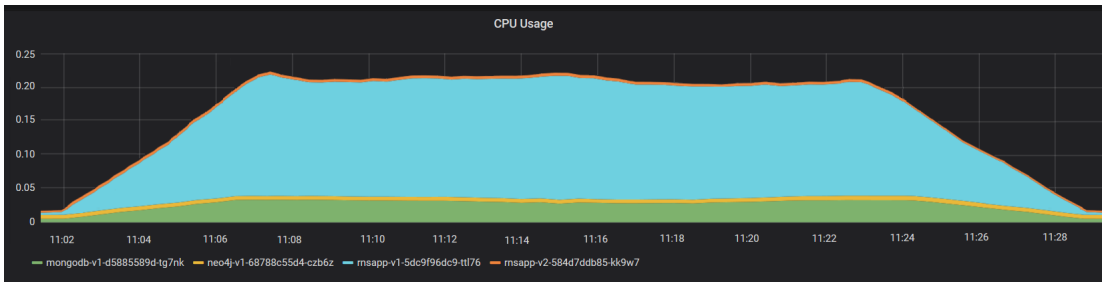


Figura 11.9: *Consumo CPU sistema RoadNavigationSystem con Istio e sicurezza abilitata.*

80 User

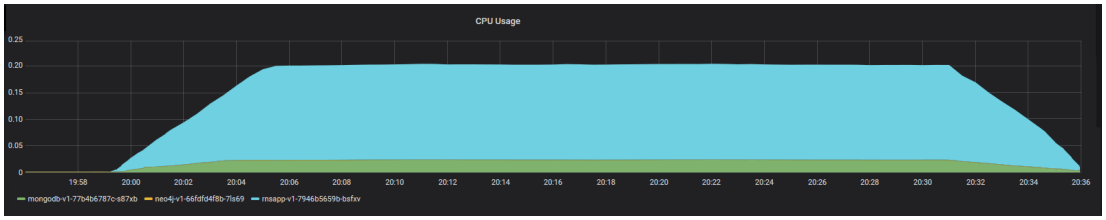


Figura 11.10: *Consumo CPU sistema RoadNavigationSystem.*

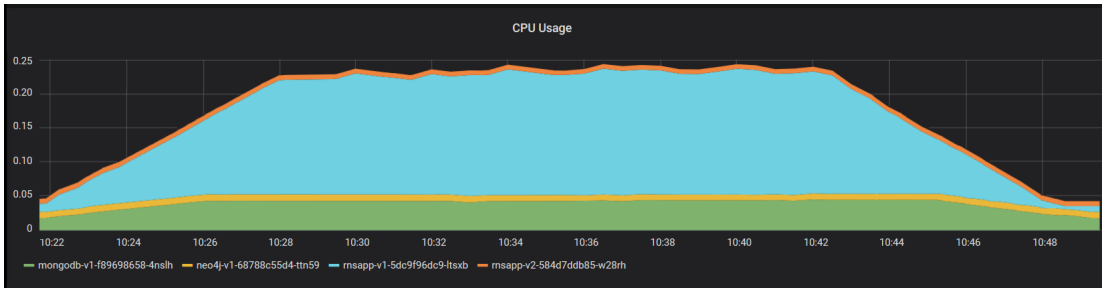


Figura 11.11: *Consumo CPU sistema RoadNavigationSystem con Istio e sicurezza abilitata.*

Come si può notare dai grafici nei diversi casi, Istio risulta avere un impatto ridotto sui pod dell'applicazione. Raggruppando, quindi, in un grafico tutti i pod del namespace applicativo, si osserva il carico di lavoro addizionale.

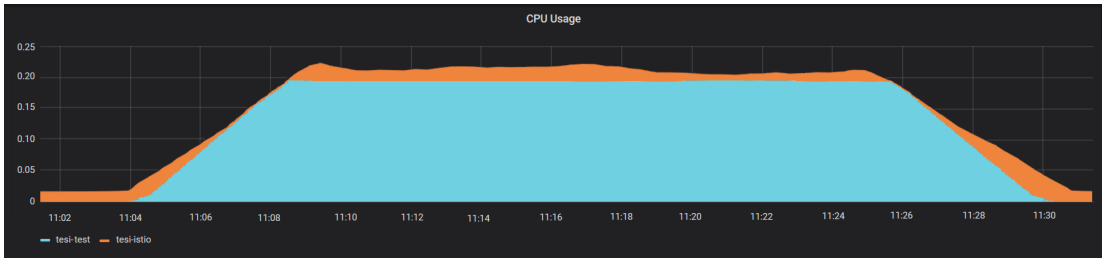


Figura 11.12: *Consumo totale della CPU sistema RoadNavigationSystem con e senza Istio.*

L'overhead sulla CPU è principalmente dato dal proxy che effettua dei controlli prima e dopo ogni richiesta, generando anche i dati per le metriche. Nella successiva figura si può osservare il consumo di CPU per container.

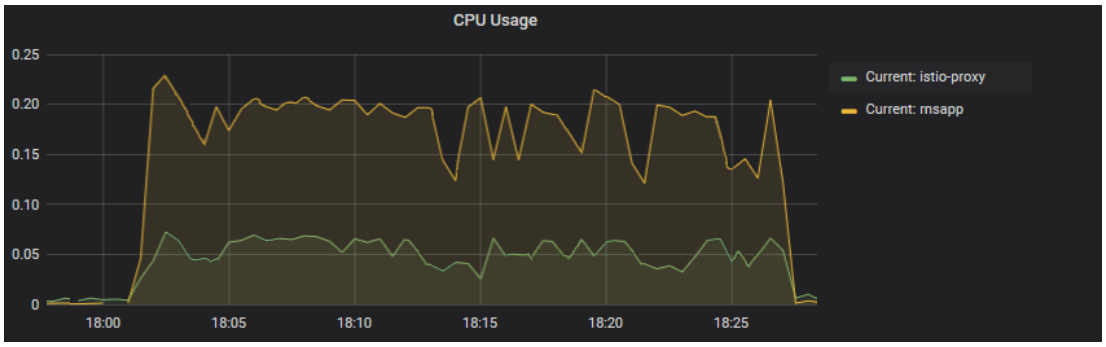


Figura 11.13: *Consumo CPU di un singolo pod.*

In questo caso, come mostrato in figura 11.13, si sta analizzando istantaneamente il pod che gestisce maggiormente la richiesta, quello del servizio *rnsapp*. Nel pod il container istio-proxy, aggiunto in fase di deploy da Istio, costituisce più o meno il 20% del calcolo computazionale, mantenendo così un basso consumo aggiuntivo sulle risorse. In alcuni casi, in cui Istio è abilitato, il cluster, essendo condiviso con altri processi e applicazioni, può presentare dei transitori nelle fasi iniziali dei test.

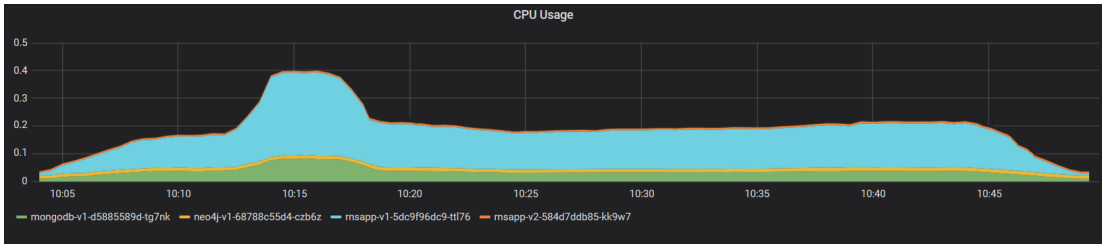


Figura 11.14: *Stati transitori nel cluster Istio.*

Questo è dovuto all'autoscaler interno di Istio. La piattaforma, infatti, abilita l'horizontal pod autoscaler su tutte le componenti Istio, mixer, pilot, galley, ingress ed egress gateway. A volte capita che alcune repliche rimangano per un po' di tempo in stato *pending* o *containerCreating* causando un consumo più elevato delle risorse.

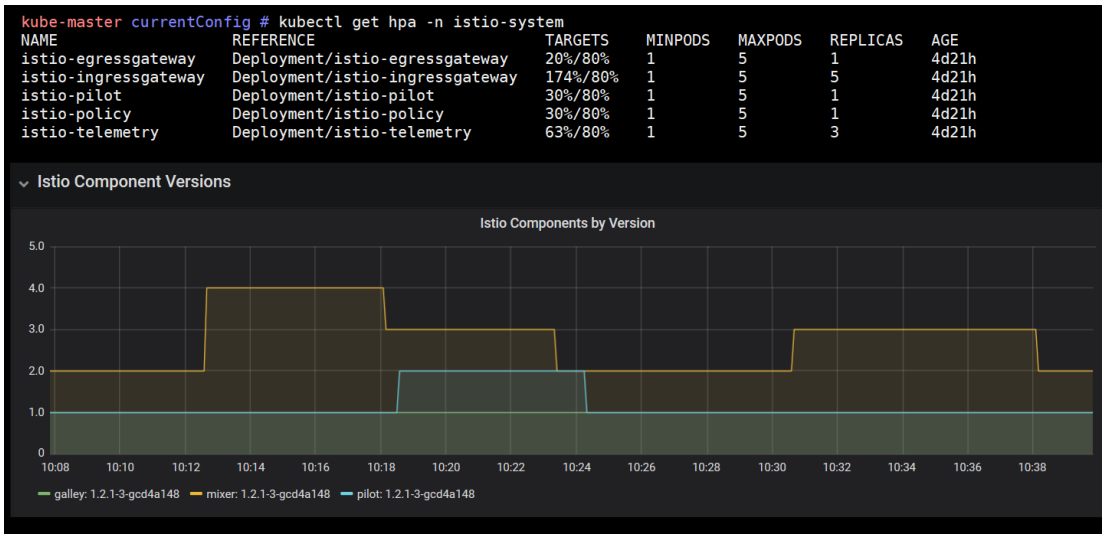


Figura 11.15: *Numero di repliche componenti Istio nel tempo.*

11.3 Traffic Networking

Quello che si osserva dai test analizzando il traffico di rete è che quello totale aumenta, poiché vi sono i dati di telemetria e lo scambio di identità tra proxy, mentre il traffico che arriva direttamente all'applicazione si riduce quando Istio è attivo.

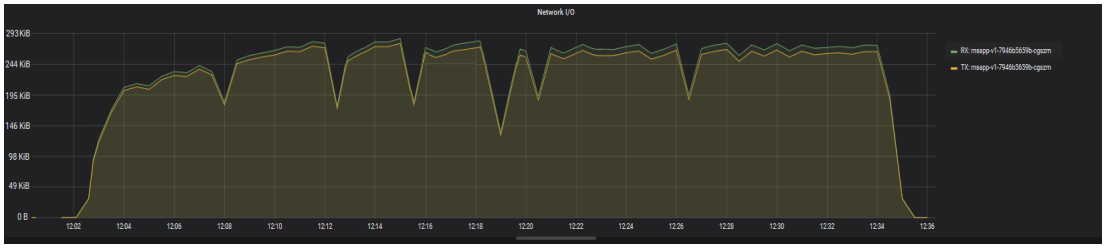


Figura 11.16: *Traffico di rete sull'applicazione senza Istio.*

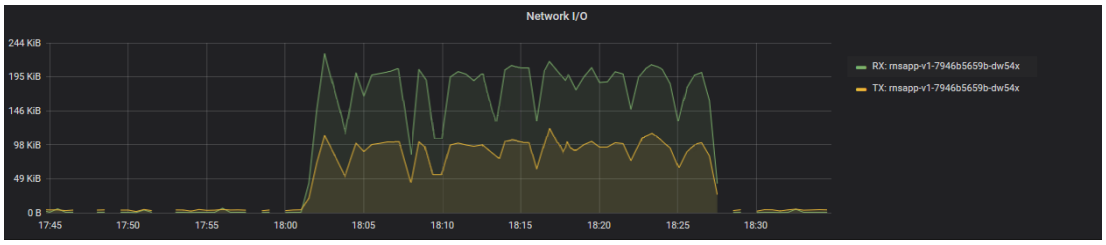


Figura 11.17: *Traffico di rete sull'applicazione con Istio abilitato.*

Quando Istio viene abilitato, il traffico di rete viene ridotto dal gateway che fa da front-end per tutte le connessioni e analizzato da tutte le altre componenti del back-end come galley, citadel e pilot. In questo modo all'applicazione arriverà

solamente il traffico dati, mentre tutto il traffico che si occupa di controllo e handshake si fermerà nel piano di controllo Istio. Stesso ragionamento si può fare per il traffico in trasmissione.

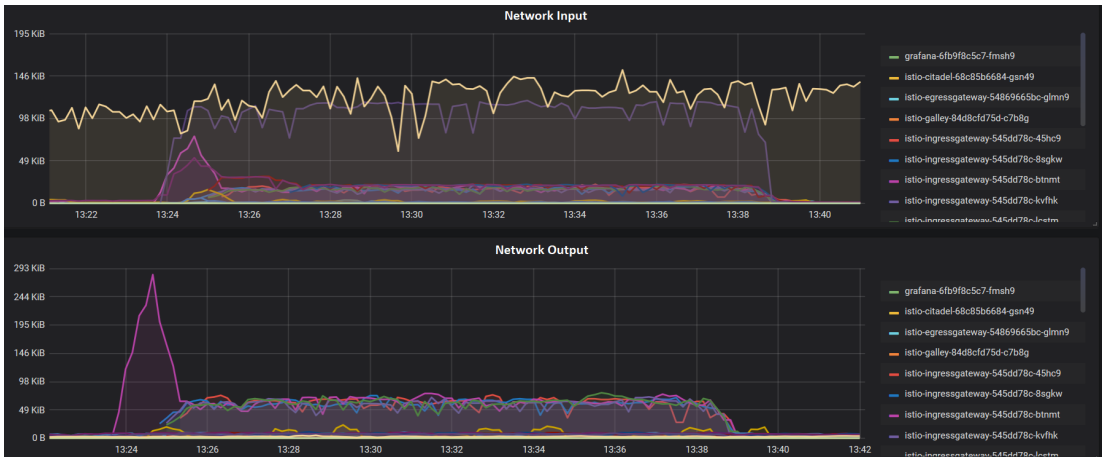


Figura 11.18: *Traffico di rete componenti Istio.*

11.4 Impatto Istio sul cluster

Dall'analisi precedente si evince che Istio impatta poco sul carico dell'applicazione sviluppata. Si analizza, ora, il peso di Istio all'interno del cluster. Sulla memoria RAM in totale Istio alloca circa 800 megabytes, come si vede dalla figura 11.19 della memoria utilizzata dai diversi namespace.

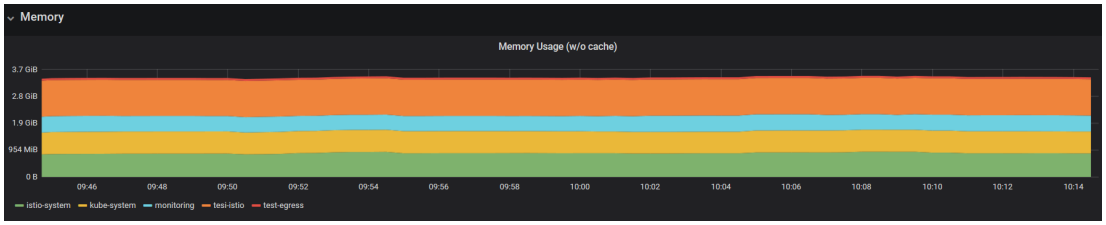


Figura 11.19: *Consumo della memoria componenti Istio totale.*

La memoria è utilizzata per lo più da Prometheus e dai componenti di telemetria per la raccolta delle metriche. In un'analisi del consumo della CPU e del traffico di rete si nota, infatti, che la componente istio-telemetry è la più attiva durante i vari test.

Figura 11.20: *Consumo della CPU e traffico di rete generato dalle componenti Istio.*

Le metriche utilizzate nelle dashboard precedenti sono le seguenti:

```
sum(irate(istio_response_bytes_sum))
sum(irate(istio_request_bytes_sum))
sum(rate(container_cpu_usage_seconds_total))
```

parametri analizzati: `source_workload_namespace, source_workload, destination_workload_namespace, destination_workload, job, pod_name, container_name.`

Un'altra componente attiva è quella composta dall'insieme dei proxy Envoy che interagiscono nelle comunicazioni e nella generazione dei dati per le metriche. Tale insieme non è formato solo dai proxy allegati ai servizi, ma anche dai proxy di "periferia" alla service mesh allegati a gateway, mixer e pilot. Come si osserva dalla figura sottostante, però, i consumi rimangono relativamente bassi e contenuti.

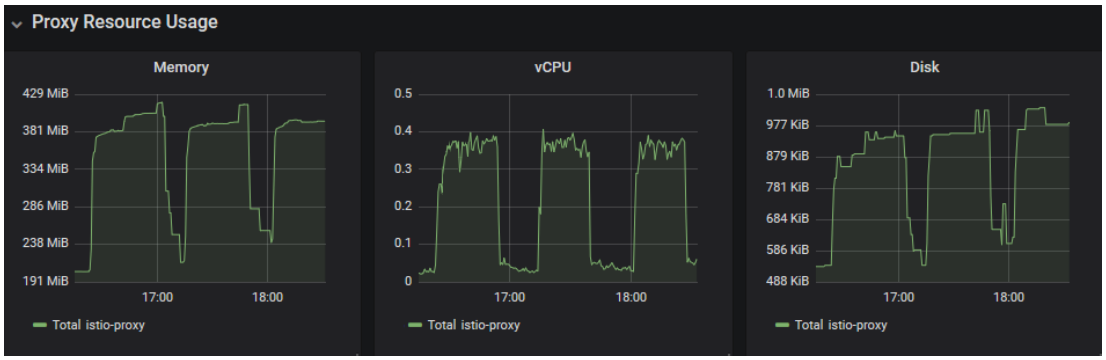


Figura 11.21: *Consumo delle risorse dei proxy Istio.*

Come per i proxy, anche i consumi delle componenti pilot e mixer rimangono contenuti e in linea con le analisi viste precedentemente.

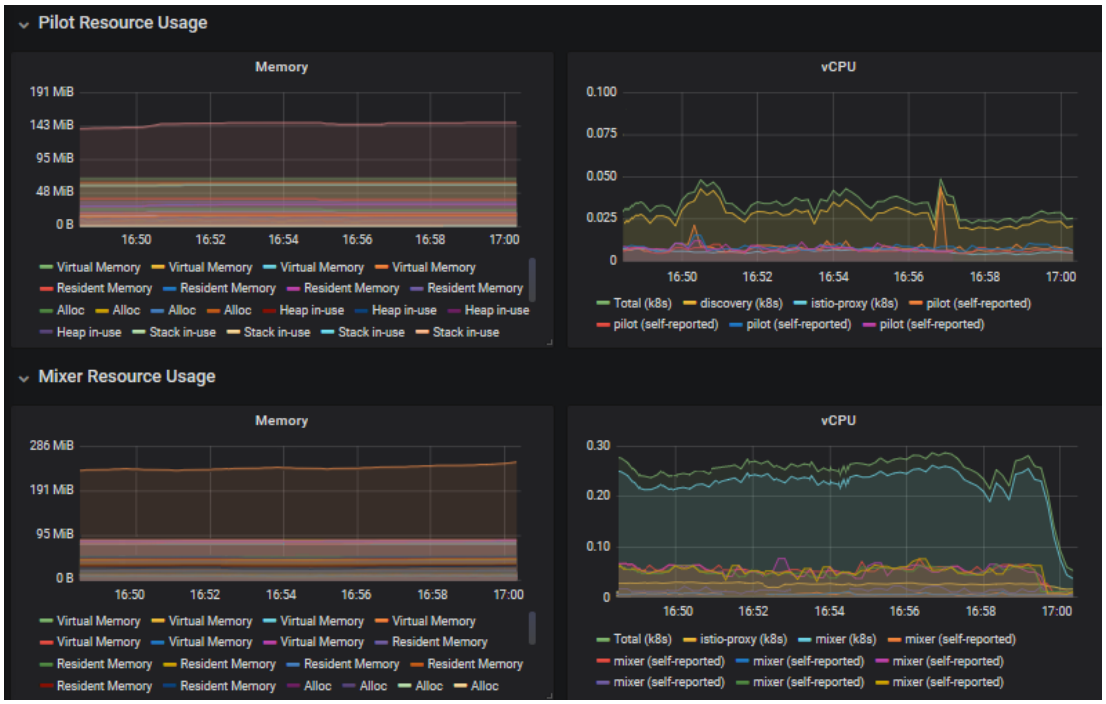


Figura 11.22: *Consumo delle risorse delle componenti Istio pilot e mixer.*

Capitolo 12

Sviluppi futuri

Questo studio è stato effettuato su un cluster che utilizza Kubernetes come strumento di orchestrazione con Docker, che gestisce l'esecuzione dei container. Uno dei possibili sviluppi futuri è quello di effettuare l'installazione di Istio su più cluster ed applicare le configurazioni di rete e sicurezza Istio non solo sulle comunicazioni intra-cluster, ma anche su quelle inter-cluster. Ad esempio, utilizzare un'unica service mesh ma organizzata su due cluster. I due cluster avranno due Istio control plane identici e un gateway sul bordo di un cluster, che permette la comunicazione tra i due cluster.

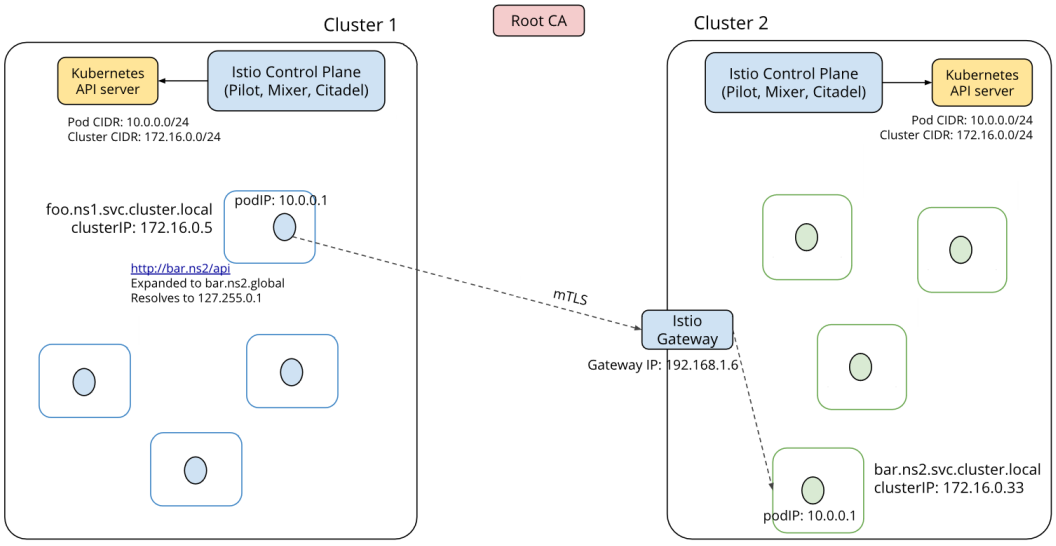


Figura 12.1: *Configurazione di service mesh in Istio Multicluster con replica.* [16]

In questo caso, l'utente si deve impegnare a mantenere una copia del control plane Istio tra i due cluster e delle componenti di configurazione dei servizi, e di configurare il DNS correttamente in modo che i servizi esterni ai cluster vengano espansi in `<name>.<namespace>.global`. Per garantire una comunicazione

sicura tra i due cluster, la root CA deve essere condivisa così che la componente citadel interna ai due cluster possa usare certificati intermedi generati e firmati dalla certificate authority condivisa. Il gateway dovrà inoltre essere raggiungibile dall'altro cluster, load balancer, per non creare un'ulteriore rete di overlay tra i due cluster. Per non avere repliche all'interno dei cluster, è possibile utilizzare un control plane Istio condiviso fra i due cluster. In base agli strumenti utilizzabili, si potranno creare degli ambienti multicluster con rete indipendente o condivisa.

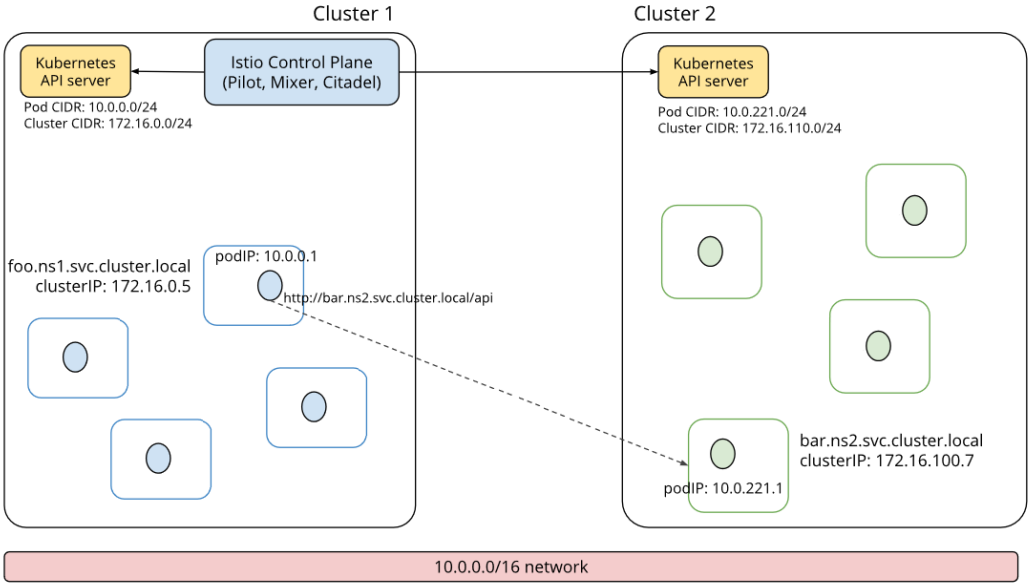


Figura 12.2: *Configurazione di service mesh in Istio Multicluster con rete condivisa.* [16]

Utilizzando una rete condivisa, i due cluster non devono avere reti sovrapposte sia per i pod che per i servizi. Ci deve essere un servizio, come una VPN che permette l'univocità dei pod e dei servizi all'interno del multicluster e la raggiungibilità di tutti i pod e di tutte le componenti del control plane kubernetes. Le risoluzioni DNS anche in questo caso devono essere configurate dall'utente. È infine possibile creare un multicluster con reti indipendenti tra un cluster ed un altro. Nel cluster primario ci saranno tutte le componenti Istio, mentre in quello secondario ci saranno solo le componenti Citadel, SidecarInjector e Gateway.

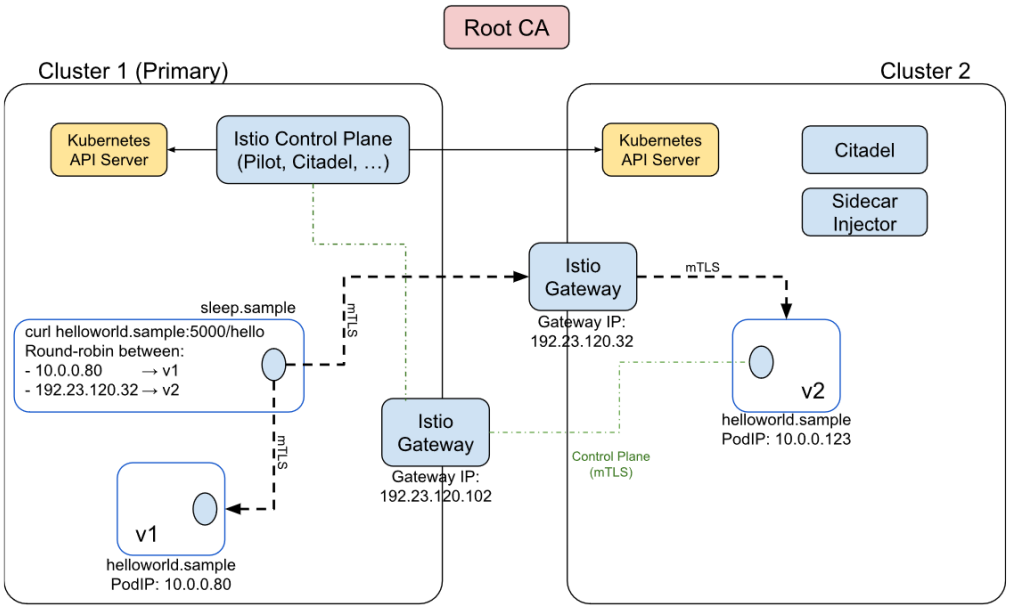


Figura 12.3: *Configurazione di service mesh in Istio Multicluster con reti indipendenti.* [16]

I due cluster non avranno bisogno di un servizio di vpn o di comunicazione diretta tra i pod dei cluster. A garantire la comunicazione tra pod di cluster diversi saranno i gateway, con specifiche configurazioni ricevute dal control plane. Il server kubernetes del cluster secondario deve però essere accessibile da quello primario affinchè si possano configurare correttamente tutte le policy. Altri progetti futuri potrebbero comprendere cluster non basati su piattaforme kubernetes, essendo Istio indipendente dal layer sottostante. Si potranno verificare, quindi, le funzionalità di Istio nelle interazioni tra più cluster con piattaforme diverse.

Capitolo 13

Conclusioni

Lo studio svolto illustra come alcune tematiche moderne, presenti nel mondo dell'informatica, possano essere combinate insieme per garantire raggiungibilità, sicurezza e monitoraggio per un'infrastruttura cloud. Si è visto come, creando ed eseguendo dei container con Docker, sia necessiaria la presenza di un orchestratore che li gestisca. Kubernetes si candida per questo scopo, inglobando nelle sue unità fondamentali e gestendo in maniera organizzata e controllata i container Docker. Tramite kubernetes, quindi, si è formata una rete di servizi dove ciascuno di essi comunica con tutti gli altri. Per controllare tale rete e coordinare il traffico tra i servizi si colloca Istio. Con tale piattaforma, infatti, si sono applicate gradualmente politiche di routing al bordo e all'interno della rete di servizi tramite virtual service, gateway e destination rule, nonché criteri di sicurezza per garantire confidenzialità, autenticazione e autorizzazione per ogni chiamata. Tramite poi degli strumenti addizionali che Istio offre, si è potuto analizzare il comportamento della rete e lo stato dei servizi ad ogni cambiamento, generare dati di telemetria per statistiche personalizzate, visualizzare su un'unica finestra vari grafici di come queste metriche variano nel tempo. Tali operazioni sono state svolte applicando semplici file di configurazione senza mai apportare pesanti modifiche al codice applicativo. Nell'ultima sezione del lavoro è stato analizzato l'impatto che Istio ha su un'applicazione di test all'interno della rete di servizi. Come si è visto dai risultati ottenuti, Istio aggiunge un overhead ammissibile sull'utilizzo delle risorse dell'applicazione, RAM e CPU, generando un traffico di rete aggiuntivo, per le metriche di controllo, non troppo elevato. Infine si è analizzato il peso che la piattaforma open-source Istio ha sull'intero cluster. Anche qui, viste le analisi eseguite, si nota come le componenti che durante i test sono maggiormente attive consumano una quantità non così cospicua, paragonabile a quella che consuma una semplice applicazione all'interno del cluster. CPU, RAM e traffico inoltrato dai proxy e dal mixer rimangono limitati durante il periodo di test, presentando picchi e utilizzi rilevanti sulle risorse solo in alcuni casi parti-

colari, controllati e ridimensionati da Istio stesso. Istio, quindi, nell'ottica cloud offre una maggiore visibilità sulle applicazioni e riduce il tempo necessario per reperire e correggere eventuali errori presenti nella rete. La piattaforma fornisce, infatti, subito una visibilità dettagliata di tutte le applicazioni presenti nella sua mesh in tempo reale e in maniera uniforme indipendentemente dal linguaggio di programmazione su cui si basano i servizi. Ciò non fa altro che migliorare le prestazioni globali di un cluster, aumentando la sua robustezza e affidabilità.

Elenco delle figure

Bibliografia

[1] 01NET, "Perché microservizi e container saranno la regola." https://www.01net.it/microservizi-container, 09 2017.

[2] Redhat, "Cosa sono i microservizi." https://www.redhat.com/it/topics/microservices, 03 2018.

[3] AmazonWebServices, "Microservizi." https://aws.amazon.com/it/microservices.

[4] 1ClickApp, "Microservizi e container: il nostro pattern architetturale cloud." http://www.1clickapp.it/2017/11/09/microservizi-e-container-il-nostro-pattern-architetturale, 11 2017.

[5] GoogleCloudPrivate, "Google container." https://cloud.google.com/containers.

[6] Redhat, "Cos'è un container linux." https://www.redhat.com/it/topics/containers/whats-a-linux-container, 01 2018.

[7] Docker, "What is a container." https://www.docker.com/resources/what-container.

[8] A. Buehrle, "Introduction to service meshes on kubernetes and progressive delivery." https://dzone.com/articles/introduction-to-service-meshes-on-kubernetes-and-p, 08 2019.

[9] T. Rahman, "Handling signals for applications running in kubernetes." https://tasdikrahman.me/2019/04/24/handling-singals-for-applications-in-kubernetes-docker, 04 2019.

[10] S. Manpathak, "Kubernetes service mesh: A comparison of istio, linkerd and consul." https://platform9.com/blog/kubernetes-service-mesh-comparison-of-istio-linkerd-and-consul, 10 2019.

[11] Partech, "Docker è la piattaforma software di containerizzazione leader nel mondo." https://www.partech.it/docker-container, 12 2017.

[12] Docker, "Docker documentation." https://docs.docker.com.

[13] Mapr, "Kubernetes. a portable, open-source platform for managing containerized applications and services." https://mapr.com/products/kubernetes/.

[14] Kubernetes, "Kubernetes documentation." https://kubernetes.io/docs.

[15] Weave, "Kubernetes on aws | weaveworks." https://www.weave.works/technologies/kubernetes-on-aws/.

[16] Istio, "Istio documentation." https://istio.io/docs.

[17] Rinormaloku, "Istio in practice – routing with virtualservice." https://rinormaloku.com/istio-practice-routing-virtualservices/, 01 2019.

Ringraziamenti

Sono giunto al termine del mio percorso formativo e adesso vorrei ringraziare tutte le persone che mi hanno aiutato a raggiungere questo traguardo. Un "Grazie" enorme va dedicato alla mia famiglia che mi ha sempre sostenuto e incitato a fare meglio. In particolare ringrazio mio padre, prima guida principale per ogni mio dubbio o problema informatico, nonché complice di alcuni esami e progetti. Ringrazio mia madre che non mi ha mai fatto mancare l'aria di casa con i suoi pacchi da giù. Ringrazio mio fratello e mia sorella colleghi di qualsiasi malefatta possibile e immaginabile. Ringrazio anche i miei nonni, i miei cugini e i miei zii per il supporto fin dal primo giorno di università. Ringrazio i miei amici Stefano, Michele e Angelo da sempre presenti al mio fianco e complici di ogni battaglia sul campo da calcio e soprattutto nei pub. Ringrazio poi tutto il team dell'azienda che mi ha aiutato in questo studio, Gioacchino, Federico, Guido, Michele, Daniele, Martina, Thomas, Carlo, Mario, Antonio, Valeria, Dario, Marco e tutti gli altri sempre disponibili e pazienti ad ogni mia domanda. Ringrazio i miei nuovi amici trovati qui a Torino, Francesco, Massimiliano, Simone, Claudio sempre presenti per una birra, una pizza o una partita. Ringrazio gli amici trovati nei miei cinque anni in Collegio, Gabriele Scaffidi, Alessio, Luca, Vincenzo, Simone, Michele, Alessandro e tutti gli altri "einaudini" che mi hanno aiutato e migliorato nello studio ma soprattutto hanno contribuito nel divertimento con feste e serate. Ringrazio la squadra delle Fatine Volanti, Gabriele, Stefano, Simone, Angelo, Francesco e gli altri membri, squadra di calcio in cui ho giocato, che mi ha sempre spinto a migliorare e distratto dallo studio.