

TESI DI LAUREA IN
INFORMATICA

Implementazione di politiche di accesso stateful con Rego: sfide e soluzioni

CANDIDATO

Fabio Ionut Ion

RELATORE

Prof. Marino Miculan

CORRELATORE

Dott. Matteo Paier

CONTATTI DELL'ISTITUTO

Dipartimento di Scienze Matematiche, Informatiche e Fisiche

Università degli Studi di Udine

Via delle Scienze, 206

33100 Udine — Italia

+39 0432 558400

<https://www.dmif.uniud.it/>

A Lara, la mia certezza nella vita,
alla mia famiglia, per il continuo supporto
e a mio zio, che vorrei fosse qui con me.

Ringraziamenti

Un sentito grazie a tutte le persone che mi hanno permesso di arrivare fin qui e di portare a termine questo lavoro di tesi.

Ci tengo innanzitutto a ringraziare il mio relatore Prof. Marino Miculan, che mi ha seguito, passo dopo passo, in questo percorso. In particolare, per avermi fatto avvicinare al progetto e per il tempo dedicatomi durante tutto il processo di tesi. In seconda battuta voglio ringraziare il mio correlatore Dott. Matteo Paier, per la sua grandissima lungimiranza e per i consigli ricevuti durante lo sviluppo della tesi, e Massimiliano Baldo, per l'aiuto e per le delucidazioni date, nonostante la nostra conoscenza sia avvenuta solo recentemente.

Ringrazio tutta la mia famiglia e, in particolare, i miei genitori e mia sorella per avermi dato l'opportunità di intraprendere questo percorso universitario, per il continuo supporto, per i saggi consigli dati e per la carica ricevuta prima di un esame.

Ringrazio la famiglia della mia fidanzata e tutto lo staff della Sound Promotion A.P.S. Ringrazio tutti gli amici che mi sono stati vicini in questi ultimi anni, soprattutto Sarta per i momenti spensierati passati insieme e le risate condivise.

Infine, ringrazio la mia amata Lara per avermi trasmesso la sua immensa forza e il suo coraggio. Per esserci sempre stata, nel momento del bisogno. Il suo supporto è stato fondamentale in questo viaggio, ispirandomi ogni giorno a dare il massimo e a raggiungere i miei obiettivi. Spero che questo lavoro possa riflettere non solo il mio impegno, ma anche il rapporto speciale che ci lega.

Indice

1	Introduzione	1
2	Le architetture a container	3
2.1	Architettura a microservizi vs. monolitica	3
2.2	Docker	4
2.3	Kubernetes	5
2.3.1	L'architettura dei cluster	6
2.3.2	Nodi	6
2.3.3	Pods	7
2.3.4	Namespaces	8
2.3.5	Network Policies	8
2.4	Presentazione caso di studio	9
3	Service Mesh	11
3.1	Architettura di una service mesh	12
3.2	mTLS	14
3.3	Policy di Sicurezza	15
3.4	Applicazione caso di studio	17
4	Policy-as-Code: OPA	19
4.1	Rego	19
4.2	Principali differenze tra Policy-as-Code e Policy-as-Yaml	22
4.3	Implementazione di due policy tramite Rego	23
4.3.1	Policy scenario Contatore	23
4.3.2	Policy scenario Tre Microservizi	23
4.4	Ulteriori tool per Policy-as-Code	24
5	OPA-Wrapper State Manager	25
5.1	Fondamenti e motivazioni del wrapper	25
5.2	Principio chiave per salvare lo stato	26
5.3	Architettura di OSWM	27
5.4	Implementazione di OWSM	28
5.4.1	Datastore	29
5.4.2	Opawrap	31
5.5	Esempi di utilizzo di OWSM	37
5.5.1	Policy scenario Contatore	37
5.5.2	Policy scenario Tre Microservizi	37
6	Valutazione OWSM	39
6.1	Esperimento 1	39
6.2	Esperimento 2	40
6.3	Esperimento 3	40
6.4	Considerazioni finali	44

7 Conclusioni	45
7.1 Possibili implementazioni e sviluppi futuri	46
A Esempio utilizzo OWSM	47
A.1 Scenario Contatore	47
A.2 Scenario Tre Microservizi	49

Elenco delle figure

2.1	Architettura di un container a confronto con l'architettura delle VM (figura da [27]).	4
2.2	Applicazione monolitica e l'equivalente versione a microservizi (figura da [36]).	4
2.3	Architettura di Docker Engine (figura da [26]).	5
2.4	Architettura di Kubernetes (figura da [8]).	6
2.5	Un worker node (figura da [8]).	7
2.6	Quattro pods in Kubernetes (figura da [8]).	8
2.7	I tre microservizi del caso di studio.	9
2.8	Una network policy per il microservizio authors.	10
3.1	Sidecar proxies e la rete di mesh che creano (figura da [24]).	12
3.2	Control plane e data plane di una generica service mesh (figura da [32]).	13
3.3	Control plane e data plane di Linkerd (figura da [15]).	14
3.4	Control plane e data plane in Istio (figura da [5]).	14
3.5	mTLS (figura da [31]).	15
3.6	Funzionamento AuthorizationPolicy in Istio (figura da [5]).	17
3.7	Authorization Policy per il microservizio authors usando Linkerd come service mesh (si veda fig. 2.8 per la versione equivalente in Kubernetes).	18
3.8	I tre microservizi del caso di studio dopo aver aggiunto i proxy di Linkerd.	18
4.1	Architettura dell'interazione tra OPA e i microservizi (Figura da [1]).	20
4.2	Un esempio di regola multi-linea in Rego (usando <i>The Rego Playground</i> [38]).	21
4.3	Un esempio di regola incrementale in Rego (usando <i>The Rego Playground</i> [38]).	22
4.4	Policy per lo scenario Contatore	23
4.5	Policy per lo scenario Tre Microservizi	23
5.1	Esempio di come restituire lo stato in Rego (usando <i>The Rego Playground</i> [38]).	26
5.2	Architettura OWSM.	27
5.3	Interazione tra i vari moduli di OWSM.	28
5.4	JSON della richiesta a /data/{key}.	31
5.5	Moduli e package del progetto OWSM in Go	33
5.6	Policy con lo stato per lo scenario Contatore	37
5.7	Policy con lo stato per lo scenario Tre Microservizi	38
6.1	Esperimento 1 Policy 1 - a sinistra andamento OWSM e OPA, a destra il peggioramento in percentuale di OWSM.	41
6.2	Esperimento 1 Policy 2 - a sinistra andamento OWSM e OPA, a destra il peggioramento in percentuale di OWSM.	41
6.3	Esperimento 1 Policy 3 - a sinistra andamento OWSM e OPA, a destra il peggioramento in percentuale di OWSM.	41
6.4	Esperimento 2 Policy 1 - a sinistra andamento OWSM e OPA, a destra il peggioramento in percentuale di OWSM.	42
6.5	Esperimento 2 Policy 2 - a sinistra andamento OWSM e OPA, a destra il peggioramento in percentuale di OWSM.	42

6.6	Esperimento 2 Policy 3 - a sinistra andamento OWSM e OPA, a destra il peggioramento in percentuale di OWSM.	42
6.7	Esperimento 3 Policy 1 - a sinistra andamento OWSM e OPA, a destra il peggioramento in percentuale di OWSM.	43
6.8	Esperimento 3 Policy 2 - a sinistra andamento OWSM e OPA, a destra il peggioramento in percentuale di OWSM.	43
6.9	Esperimento 3 Policy 3 - a sinistra andamento OWSM e OPA, a destra il peggioramento in percentuale di OWSM.	43
A.1	Contenuto datastore al momento dell'avvio nello scenario Contatore	47
A.2	Output query e contenuto datastore con input che viola la policy per lo scenario Contatore	47
A.3	Output query e contenuto datastore con un input accettato dalla policy per lo scenario Contatore	48
A.4	Output query e contenuto datastore superato il numero di accessi consentiti dalla policy per lo scenario Contatore	48
A.5	Contenuto datastore al momento dell'avvio nello scenario Tre Microservizi	49
A.6	Output query e contenuto datastore con input B e C per la policy dello scenario Tre Microservizi	49
A.7	Output query e contenuto datastore con input A e B per la policy dello scenario Tre Microservizi	49
A.8	Output query e contenuto datastore con input B e C dopo la comunicazione tra A e B per la policy dello scenario Tre Microservizi	50

Listings

5.1	Struttura del datastore	29
5.2	Il cuore del server datastore	30
5.3	Salvataggio di una nuova chiave-valore nello store	30
5.4	Firma della funzione da passare all'handler	30
5.5	Il cuore del server opawrap	32
5.6	Pseudo-codice della funzione <code>handleQuery</code>	32
5.7	Recupero dei dati presenti nel datastore	32
5.8	Funzione che delega la valutazione della query a OPA.	36
5.9	Aggiornamento dello stato in datastore.	36

1

Introduzione

I container al giorno d'oggi hanno un ruolo fondamentale nello sviluppo di applicazioni basate su un'architettura a microservizi, sostituendo le consuete Virtual Machine (VM). Difatti, secondo un sondaggio di CNCF, il 90% delle applicazioni a microservizi utilizza i container [20].

Sebbene il loro impiego porti a numerosi benefici quali per esempio la portabilità, la flessibilità e la scalabilità, questi presentano delle problematiche a livello di sicurezza. Difatti non è possibile considerarle sicure per natura. Si pensi che solamente Twitter (attuale X) usufruisce di $O(10^3)$ diversi microservizi e $O(10^5)$ istanze di essi [28], perciò valutare la loro sicurezza è abbastanza difficile.

Essendo la comunicazione tra i microservizi così vasta e accessibile in rete, ciò può creare una potenziale possibilità di attacco. Infatti potrebbe succedere che un singolo microservizio compromesso attacchi l'intera applicazione, inviando richieste maliziose a un altro microservizio.

Questa è una situazione piuttosto verosimile in quanto, come un recente rapporto ha sottolineato, il 51% delle immagini di container presenti nel Docker Hub, presenta delle vulnerabilità [9]. Per questo motivo è stato istituito il progetto *SecCo*, il cui obiettivo è quello di supportare lo sviluppo sicuro di applicazioni containerizzate in architetture distribuite ed eterogenee, come nel caso delle architetture a microservizi [41].

Questo lavoro di tesi affianca quindi il progetto maggiore *SecCo*, in quanto l'obiettivo è quello di rendere la comunicazione tra microservizi sicura, facendo sì che questa sia guidata dalla definizione di politiche di sicurezza.

Una politica è un insieme di regole che definiscono lo stato desiderato dell'applicazione basata su un'architettura a microservizi. Tramite una politica di sicurezza si stila quindi una lista di concessioni, col fine di prevenire gli accessi indesiderati alle risorse di un particolare microservizio. A fronte di una adeguata definizione di politiche di sicurezza, si può dunque governare la protezione di risorse delicate. In questo modo si evita che un microservizio, anche nel momento in cui potrebbe risultare compromesso, possa comunicare con i microservizi che contengono informazioni o risorse riservate.

Durante il processo di ricerca, sono stati individuati diversi modi per scrivere politiche di sicurezza che permettono di raggiungere l'obiettivo finora discusso. Ognuno di essi presenta, oltre a degli effettivi vantaggi, delle sostanziali mancanze per riuscire a risolvere il problema in maniera quanto più completa possibile. Tra tutte le modalità studiate, che verranno presentate nei primi capitoli, è emerso un nuovo approccio di implementazione di politiche di sicurezza: *Policy-as-Code*.

Questa è una filosofia che permette di scrivere politiche di sicurezza attraverso l’uso di linguaggi dichiarativi, che prendono il nome di *linguaggi decisionali*. La caratteristica principale dei linguaggi decisionali è di concentrarsi su chi può fare cosa in un’architettura a microservizi. Visto l’emergente problema e l’enorme interesse da parte della letteratura e di grandi aziende del mondo informatico, sono stati presentati diversi linguaggi decisionali, i quali sono in continua evoluzione.

Tra i linguaggi decisionali presenti, *Rego* si è dimostrato una valida opzione per riuscire ad esprimere policy in maniera estesa, nei contesti più generali possibili. Le politiche di sicurezza verranno quindi scritte nel linguaggio Rego, e tramite *Open Policy Agent (OPA)*, motore di policy, verrà assicurato il rispetto di queste nell’ambito di un’architettura a microservizi. OPA prenderà così il ruolo di *terza parte fidata*, il cui compito sarà quello di prendere tutte le decisioni che riguardano il rispetto delle politiche di sicurezza all’interno dell’architettura a microservizi. Avverrà quindi un disaccoppiamento tra la *business logic*, contenuta all’interno di ciascun microservizio, e le decisioni di sicurezza, che verranno prese direttamente da OPA.

Tuttavia, queste tecnologie presentano delle defezienze. Il problema sostanziale è quello di essere stateless. Ossia, una volta ricevuta una richiesta, questa viene valutata da OPA rispetto alla policy e ai dati presenti, restituendo un risultato senza mantenere alcuno stato interno. Quindi eventuali modifiche ai dati non vengono salvate. Per questo motivo non è possibile far rispettare politiche di sicurezza che si basano su variabili, il cui valore evolve nel tempo. Quindi, per quanto riguarda certe politiche di sicurezza simili a quella sopracitata, il disaccoppiamento tra le scelte di sicurezza e la *business logic* dei microservizi non è realmente mantenuta. In definitiva Rego e OPA sono stati progettati per valutare le richieste ricevute e prendere una decisione, piuttosto che sul mantenere uno stato mutevole. Per questo motivo è stato necessario lo sviluppo di un wrapper che permetta di risolvere questo problema, rendendo OPA *stateful*, cioè mantenendo uno stato tra le varie richieste ricevute.

Ricapitolando, il wrapper che verrà presentato introduce una componente essenziale, assente in OPA e Rego: la gestione dello stato. Verrà aggiunta la capacità di aggiornare dati e memorizzarli nel tempo, integrandosi in maniera naturale con le policy definite tramite il linguaggio Rego e permettendo quindi di definire politiche di sicurezza impossibili da applicare altrimenti. Vedremo come le policy di accesso possono essere suddivise in *statiche*, *stateless basate su dei dati da interrogare* oppure *stateful basate su dei dati da interrogare*.

Il resto della tesi sarà strutturata come segue. Nel capitolo 2 si discuterà più nel dettaglio il concetto di container e l’orchestratore Kubernetes, presentando gli strumenti offerti da esso per esprimere politiche di sicurezza. Il capitolo 3 introdurrà le *service mesh*, indicando i vantaggi che derivano dal loro utilizzo, focalizzando l’attenzione sulle politiche di sicurezza. Questi due capitoli presenteranno un paradigma per scrivere policy, a cui mi sono permesso di attribuire il nome *Policy-as-Yaml*. Tramite *Policy-as-Yaml* è possibile esprimere policy *statiche*. Nel capitolo 4 verranno analizzati OPA e Rego, che permettono di esprimere policy *stateless basate su dei dati da interrogare*. Il capitolo 5 presenterà più nel dettaglio le motivazioni del wrapper, così come la sua architettura, la sua implementazione e il suo comportamento tramite la messa in pratica degli scenari **Contatore** e **Tre Microservizi**, per i quali è necessario l’uso di policy *stateful basate su dei dati da interrogare*. Nel capitolo 6 verrà presentata la valutazione effettuata sul wrapper, confrontando le sue performance con quelle di OPA a sé stante. Infine, il capitolo 7 riassumerà quanto presentato e raggiunto, delineando possibili implementazioni e sviluppi futuri.

2

Le architetture a container

Docker definisce un container nel seguente modo:

“Un container è un’unità standard di software che impacchetta il codice e tutte le sue dipendenze in modo che l’applicazione funzioni in modo rapido e affidabile da un ambiente informatico all’altro.”
- Docker [27]

Fin dalle origini, i container hanno sempre occupato un importante ruolo nell’implementazione e nella progettazione di sistemi software [3, 16]. Grazie ai loro vantaggi quali portabilità, tolleranza ai guasti, scalabilità orizzontale e flessibilità, le architetture a container hanno un ruolo predominante nel deployment di applicazioni suddivise in microservizi. Gli autori di [42] affermano che l’isolamento tra container è meno sicuro rispetto alle macchine virtuali (VM), in quanto i primi condividono lo stesso kernel del sistema operativo. In ogni caso l’impiego dei container porta ad un’efficienza in termini di risorse consumate, perché ogni container contiene solo lo stretto necessario per poter operare e perciò non è presente alcun *overhead* dovuto al sistema operativo, in quanto si trova in uno strato inferiore. Si veda fig. 2.1

2.1 Architettura a microservizi vs. monolitica

Dal punto di vista del sistema operativo, l’intera applicazione monolitica viene eseguita come un singolo processo. Un software viene normalmente diviso in più moduli, per rispettare il concetto di separazione delle responsabilità. Nel caso di un’applicazione monolitica, questa viene distribuita come un unico servizio che contiene tutta la logica (per esempio il front-end in React, il back-end in NodeJS e il database MongoDB). Il problema sostanziale dell’architettura monolitica è la mancanza di flessibilità: una singola modifica ad un modulo comporta il *testing* e il *deployment* dell’intera applicazione, con la possibilità di riscontrare un comportamento inaspettato in un qualsiasi punto di un altro modulo dell’applicazione stessa. Le architetture monolitiche, in definitiva, hanno una semplicità intrinseca e potrebbero essere utili e quindi la scelta preferibile quando la logica dell’applicazione è relativamente semplice e non presenta modifiche sostanziali nella sua natura.

Lo sviluppo di software che ha come obiettivi principali la flessibilità e scalabilità viene reso possibile se si implementa un’architettura a container, oppure con un nome più generale: *architettura a microservizi*. L’idea è quella di decomporre l’applicazione principale in piccoli servizi, ognuno dei quali eseguito

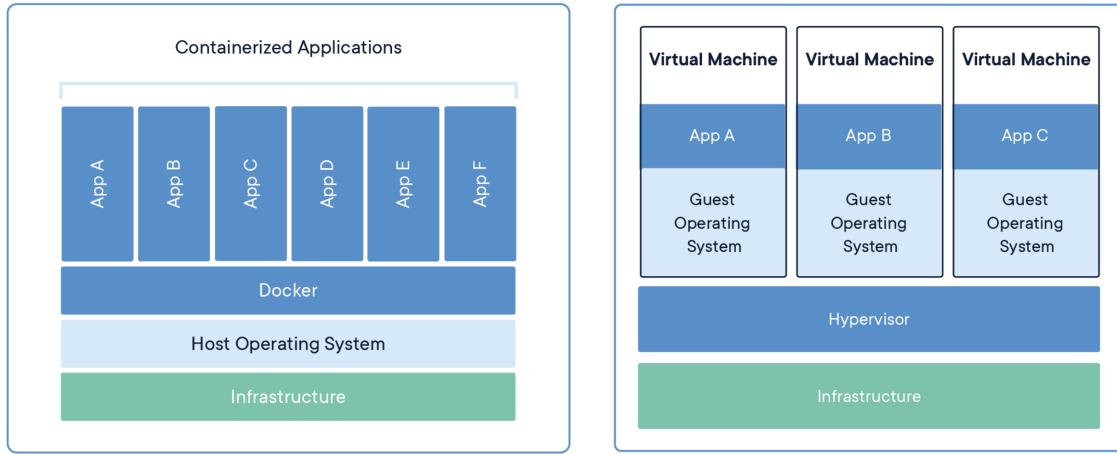


Figura 2.1: Architettura di un container a confronto con l’architettura delle VM (figura da [27]).

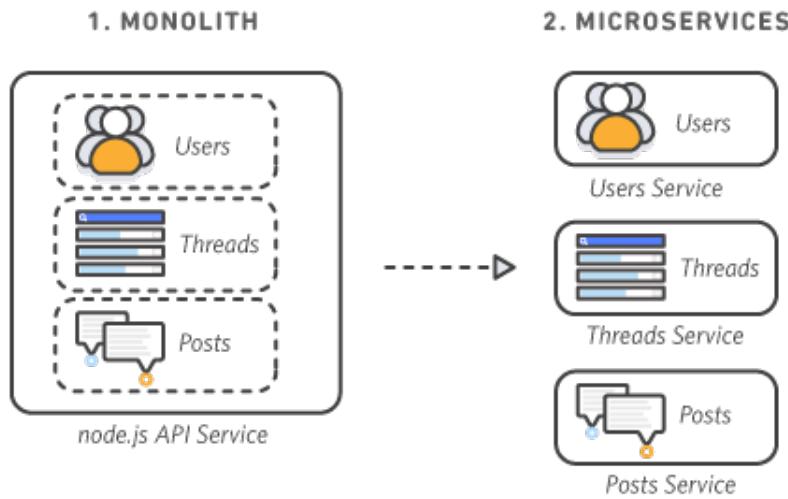


Figura 2.2: Applicazione monolitica e l’equivalente versione a microservizi (figura da [36]).

in un singolo processo, con la possibilità di collaborazione tra essi. Nel caso in cui i singoli microservizi vengano eseguiti su container diversi allora parliamo di architettura a container. I microservizi sono autonomi, ossia implementabili in maniera completamente indipendente e sono gli unici responsabili dell’esecuzione dell’attività che controllano [12]. La collaborazione tra microservizi avviene mediante le API che ognuno espone e di conseguenza tutta la *business logic* viene incapsulata in ognuno. Nel caso di un’applicazione a microservizi potremmo pensare di avere tre servizi separati per il front-end in React, il back-end in NodeJS e il database MongoDB, ognuno in esecuzione su un container apposito.

2.2 Docker

Un’immagine di container *Docker* è un pacchetto software leggero, autonomo ed eseguibile che include tutti i file necessari per eseguire un’applicazione, come per esempio il codice, le librerie di sistema e le impostazioni [27]. Quando un’immagine di container viene eseguita sul *Docker Engine* allora questa diventa un container.

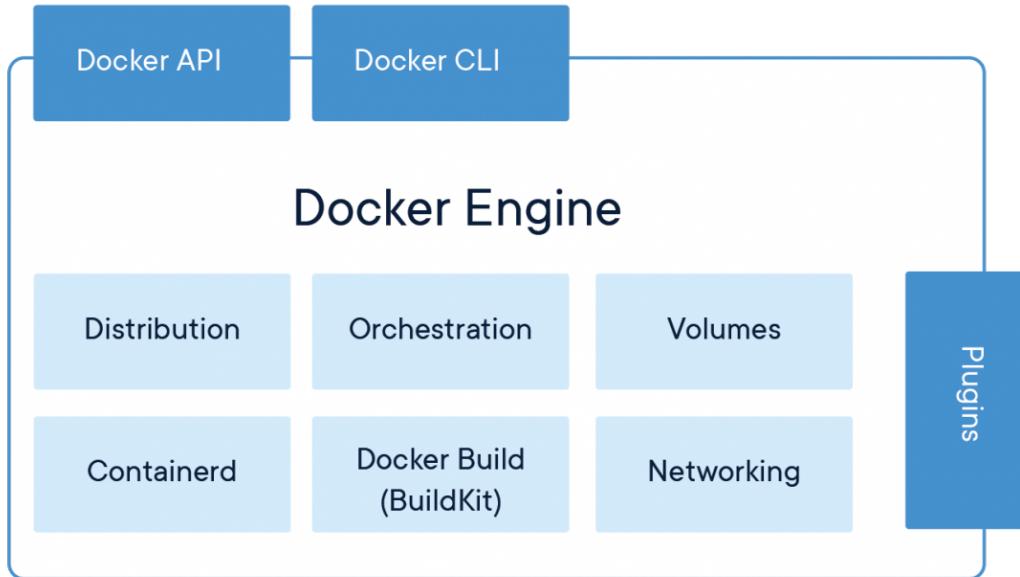


Figura 2.3: Architettura di Docker Engine (figura da [26]).

Docker Engine, la cui visualizzazione è presente in fig. 2.3, è diventato lo standard *de-facto* che esegue tutte le dipendenze di un'applicazione presenti all'interno di un container, permettendo di eseguire il container su qualsiasi infrastruttura. Docker Engine è dunque il responsabile che risolve i problemi di dipendenza ed evita il fastidioso problema “funzionava sul mio computer” [26].

La tecnologia dei container Docker, lanciata nel 2013, non è stata una novità in tutto e per tutto, in quanto si sono sfruttati concetti già esistenti nel mondo Linux, come le primitive *cgroups* e *namespaces*, rendendole multi-piattaforma e garantendo funzionalità di isolamento più sicure [27].

Uno dei principali vantaggi nel “*containerizzare*” le applicazioni consiste nel fatto che i vari container condividono il kernel del sistema operativo, al contrario dell’impiego delle macchine virtuali, permettendo quindi di eseguire più applicazioni ma su meno infrastruttura. Si veda fig. 2.1.

Nel 2015, Docker ha contribuito alla creazione dell’*Open Container Runtime (OCI)* [19] rendendo la specifica dell’immagine dei container e il codice di runtime, ora noto come *runc*, lo standard dei container. Nel 2017 il progetto *containerd*, demone che si occupa di tutto il ciclo di vita dei container in esecuzione sul sistema operativo ospitante, è stato donato alla *Cloud Native Computing Foundation (CNCF)* [21].

2.3 Kubernetes

Kubernetes è una tecnologia *open-source* per l’orchestrazione di container, rilasciata inizialmente da Google. Le origini possono essere ricondotte al sistema *Borg* [7], utilizzato internamente e segretamente da Google, reso successivamente pubblico per promuovere l’adozione di standard aperti attorno al mondo dei container. L’obiettivo di Kubernetes è gestire il ciclo di vita delle applicazioni containerizzate, facilitandone il deployment, la scalabilità e in generale la gestione [8].

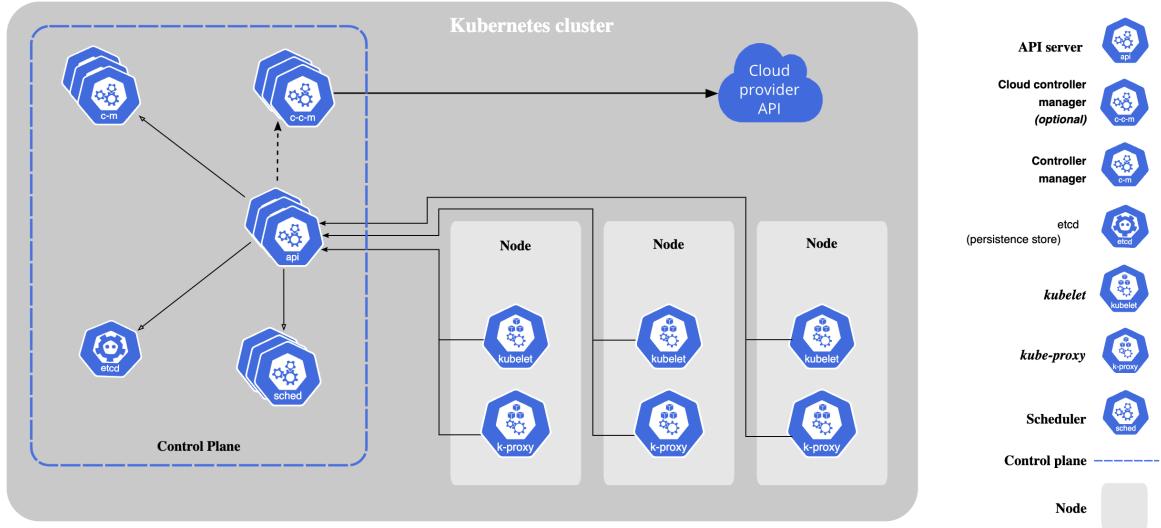


Figura 2.4: Architettura di Kubernetes (figura da [8]).

2.3.1 L'architettura dei cluster

L'architettura di Kubernetes, si veda fig. 2.4, si basa sul concetto di cluster che presenta due risorse principali:

- il **Control Plane**, il cui compito è coordinare il cluster;
- i **Worker Node**, che eseguono le applicazioni containerizzate.

Il Control Plane è costituito da componenti che gestiscono e coordinano il cluster, in modo da suddividere le varie responsabilità. Tra i componenti principali ci sono il **kube-apiserver**, l'interfaccia principale per le comunicazioni all'interno del cluster, il quale espone il *server API*; **etcd**, una base di dati distribuita che contiene lo stato di tutte le risorse del cluster; il **kube-scheduler**, il quale seleziona i pod appena creati non associati ad un nodo e li assegna al nodo più adatto per poter essere eseguito; il **kube-controller-manager**, nome ombrello che comprende vari tipi di processi di controllo, come *Node controller* o *Job controller*, per mantenere il numero desiderato di repliche di container di una determinata applicazione [8].

Il **kubelet** e il **kube-proxy** sono componenti che sono presenti in ogni nodo. Il primo gestisce i container locali al cluster e assicura che essi vengano eseguiti sui pod, mentre il secondo gestisce il traffico di rete, incluso il bilanciamento del carico da e verso i servizi in esecuzione sul cluster [8].

2.3.2 Nodi

Ci sono due tipi di nodi in Kubernetes, i *control plane nodes* e i *worker nodes*. I primi sono i nodi che contengono tutte le componenti del control plane descritte in precedenza, mentre i secondi eseguono le applicazioni containerizzate, comunicando con il control plane node attraverso il **kube-apiserver** [34]. In fig. 2.5 è presentata l'architettura di un worker node, dove il concetto di pod verrà discusso nella sezione 2.3.3.

La differenza sostanziale tra i due tipi di nodi è quindi che i worker node sono controllati da un singolo control plane node, il cui compito è quello di coordinare la comunicazione tra i vari worker

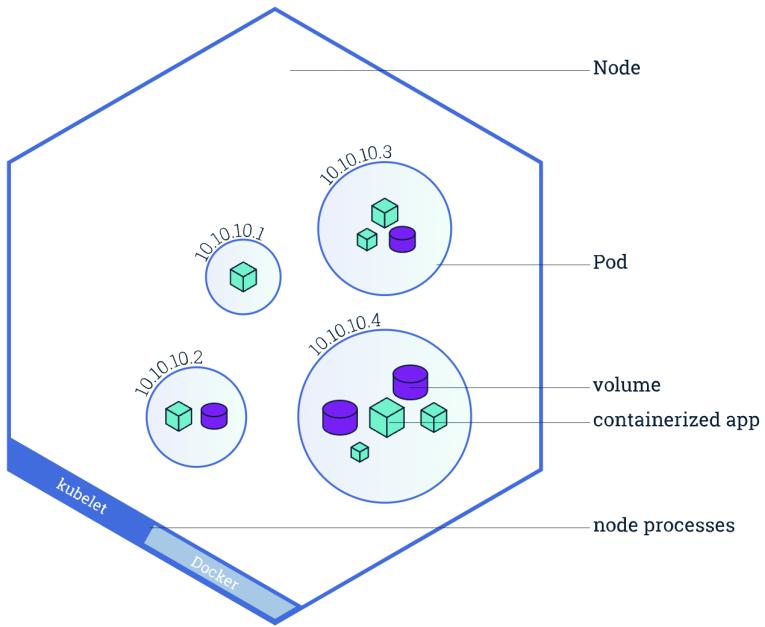


Figura 2.5: Un worker node (figura da [8]).

nodes. Il control plane node decide e programma i carichi di lavoro da eseguire su ciascun worker node, che saranno gestiti dal *kubelet* (presente in ciascun nodo).

2.3.3 Pods

Il **pod** è l’unità fondamentale e atomica in esecuzione su Kubernetes [8]. Contiene uno o più container che condividono le stesse risorse, come volumi e rete [34]. Si veda fig. 2.6. Possono essere visti come un insieme di container che condividono lo stesso *namespace* e lo stesso *filesystem*. Ogni singolo pod viene eseguito su un solo nodo.

Per permettere ai pod di poter essere eseguiti sui nodi è necessario installare un *Container Runtime* su ogni nodo del cluster. Esistono svariati tipi di Container Runtime e per poter essere supportati da Kubernetes devono sottostare alla *Container Runtime Interface (CRI)*. In tal modo Kubernetes si riserva il diritto di adoperare eventuali nuove tecnologie di runtime. Le Container Runtime più usate sono Docker Engine (87%), containerd (36%), o CRI-O(17%) [6].

Esistono due principali scenari d’uso dei pod in Kubernetes:

- **Pod che eseguono un solo container** - in questo caso il pod è una semplice astrazione del concetto di container, in quanto Kubernetes evita di gestire direttamente i container;
- **Pod che eseguono più container** - in questo caso invece il pod incapsula un’applicazione formata da più container, che lavorano a stretto contatto e condividono le risorse necessarie ad operare nel modo corretto. Per esempio un container con la business logic dell’applicazione e un container con il database e il volume per immagazzinare i dati.

Ciascun pod ottiene un indirizzo IP privato univoco all’interno del cluster. Nel caso in cui il pod contenga più container lo stesso indirizzo IP identifica tutti i container presenti e possono comunicare tra loro attraverso l’indirizzo di localhost.

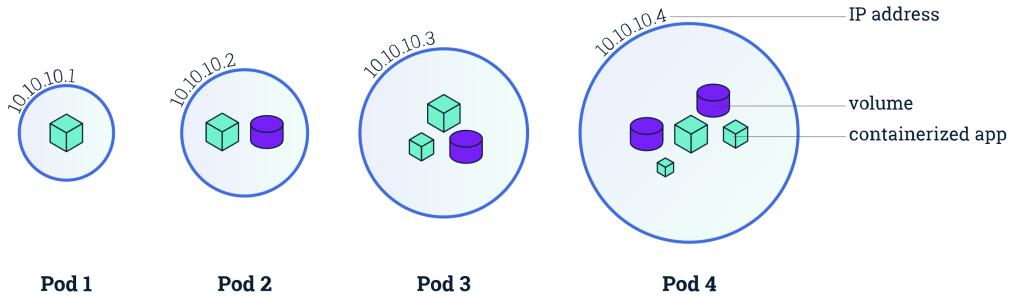


Figura 2.6: Quattro pods in Kubernetes (figura da [8]).

2.3.4 Namespaces

I *namespace* sono un meccanismo per implementare una suddivisione logica tra le risorse all'interno di un cluster. Forniscono un ambiente per i nomi delle risorse, i quali devono essere univoci all'interno del namespace. Kubernetes fornisce quattro namespaces nativi, *kube-system*, *kube-public*, *kube-node-lease* e *default*, con la possibilità di crearne altri. Brevemente, *kube-system* contiene le risorse create da Kubernetes, *kube-public* contiene le risorse leggibili da chiunque (anche da utenti non autenticati), *kube-node-lease* è usato per rendere più efficiente il monitoraggio dello stato dei nodi all'interno del cluster, mentre *default* contiene le risorse create alle quali non è stato associato un namespace. Secondo gli autori di [34] usare i namespaces insieme a *Resource Policies* (ad es. Network Policies, si veda sezione 2.3.5) e *Role Based Access Control* (RBAC), offerti da Kubernetes, è sufficiente per garantire l'isolamento tra i *tenant* in un cluster.

2.3.5 Network Policies

Controllare il traffico di rete all'interno di un cluster, tra vari pods, da e verso l'esterno è possibile grazie alla risorsa ***Network Policies*** che Kubernetes offre. Tramite le Network Policies è possibile controllare il traffico di rete a livello di rete e di trasporto dello stack ISO/OSI, specificando per un determinato pod con quali altri pod esso può comunicare, usando il concetto di risorsa presente in Kubernetes (indicando un particolare namespace oppure un determinato pod in base al nome attribuito) oppure tramite indirizzo IP e porta. Per impostazione predefinita non viene applicata alcuna restrizione ai pod, permettendo sia il traffico in uscita sia quello in entrata. L'isolamento per *egress*, il traffico in uscita, è ottenuto indicando nella lista di egress le sole connessioni consentite dal pod. L'isolamento per *ingress*, il traffico in entrata, è ottenuto indicando nella lista di ingress le sole connessioni consentite verso il pod (le connessioni provenienti dallo stesso nodo in cui è contenuto il pod sono ugualmente consentite). Le *policy* sono additive, cioè la valutazione delle stesse viene effettuata sull'unione di tutte le policies applicate a un pod in una determinata direzione. Le Network Policies non sono supportate direttamente da Kubernetes ma è necessario un *plugin* di rete apposito che le supporti e che aderisca alla specifica *Container Network Interface (CNI)* [8].

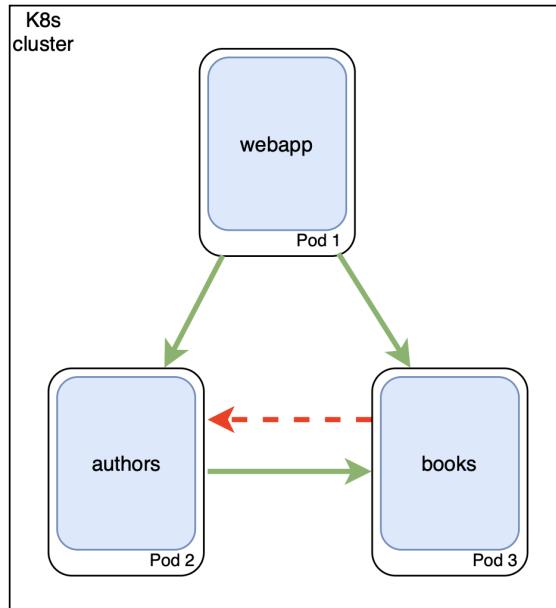


Figura 2.7: I tre microservizi del caso di studio.

2.4 Presentazione caso di studio

In questa sezione verrà presentato un caso di studio che verrà usato come esempio anche nei prossimi capitoli. Una volta presentato il caso di studio verrà mostrato un esempio di *Network Policies*.

Il caso di studio che utilizzeremo è stato sviluppato dagli autori di *Linkerd*, una service mesh che analizzeremo in dettaglio nel prossimo capitolo. È possibile reperire il codice dei microservizi, così come il file `.YAML` di deployment dell'intera architettura a microservizi, dalla pagina *GitHub*¹.

Come ambiente di esecuzione si è usato un multi-node cluster tramite **Kind**, un tool che permette di eseguire cluster di Kubernetes sulla propria macchina. Come detto in sezione 2.3.5, per poter eseguire le Network Policies in Kubernetes è necessario una CNI che le supporti, per questo motivo si è scelto di non usare *kindnet*, scelta di default per i cluster Kind, ma di usare *Calico* [40], che invece supporta le Network Policies.

L'applicazione consiste in tre microservizi scritti in *Ruby*: **webapp**, **authors** e **books**. **webapp** è il front-end che presenta la pagina web tramite la quale è possibile aggiungere/eliminare libri e/o autori. I libri sono associati ad un autore e un autore presenta un contatore di quanti libri ha scritto. Come si può vedere in fig. 2.7 **webapp** invia richieste HTTP sia a **books** che **authors**. **authors** comunica con **books**, tramite protocollo HTTP, ma non viceversa. Quindi, volendo negare la comunicazione dal microservizio **books** a **authors**, sarà necessario scrivere una network policies come quella in fig. 2.8.

La network policies è applicata solo ai microservizi che corrispondono alla label `app:authors`. Il tipo della policy è *Ingress*, perciò ci si preoccupa soltanto del traffico in ingresso al microservizio **authors**. Consentiamo quindi solo le richieste dal microservizio che corrisponde alla label `app:webapp` sulla porta 7001. L'obiettivo di scrivere la policy voluta è stato raggiunto. In questo modo si permette solo al microservizio **webapp** di effettuare richieste HTTP al microservizio **authors**.

¹<https://github.com/BuoyantIO/booksapp/tree/main>

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: booksapp-pol
spec:
  podSelector:
    matchLabels:
      app: authors
  policyTypes:
    - Ingress
  ingress:
    - from:
        - podSelector:
            matchLabels:
              app: webapp
  ports:
    - port: 7001
```

Figura 2.8: Una network policy per il microservizio authors.

In questo esempio si è usato il concetto di *label* che Kubernetes offre, permettendo quindi di far riferimento ai microservizi in base a specifici valori indicati nel file di deployment.

Limitare le connessioni soltanto a livello 3-4 dello stack ISO/OSI non è sufficiente. Sarebbe opportuno soddisfare requisiti di autenticazione e crittografia, ossia garantire che un microservizio possa comunicare con un altro solo dopo essersi scambiata opportuna informazione riguardo la loro l'identità oppure crittografare le richieste che i microservizi si scambiano. Una soluzione elegante è quella di usare una **service mesh**, che non solo permette di incontrare questi requisiti ma disaccoppia queste funzionalità dalla *business logic* del microservizio.

3

Service Mesh

Le **Service Mesh** si pongono come obiettivo quello di gestire efficacemente la inter-comunicazione tra i microservizi, aggiungendo, ad ognuno di essi, un *proxy*. Ogni microservizio avrà un proxy ad esso dedicato, in esecuzione “a fianco” a lui. Per questo motivo vengono chiamati ***sidecar proxy***, si veda fig. 3.1. Innanzitutto è importante specificare che le service mesh si appoggiano a Kubernetes, in quanto quest’ultimo, insieme a Docker, è una delle tecnologie che ha migliorato notevolmente il modo di operare con i microservizi [32].

Infatti, Docker (sezione 2.2) ha reso semplice il modo di impacchettare le applicazioni e le sue dipendenze all’interno di un container, mentre Kubernetes (sezione 2.3) ha reso semplice la gestione dinamica della distribuzione dei container, relativamente alle macchine sulle quali essi sono in esecuzione. Quindi, usare una service mesh insieme a Kubernetes è relativamente semplice:

1. Si impacchettano i proxy all’interno di container;
2. Si chiede a Kubernetes di gestire il *deployment* di questi.

Una service mesh offre una grande mole di funzionalità, che è possibile suddividere in tre classi principali: **Traffic Management**, **Observability** e **Security**. Le analizziamo una alla volta, discutendo brevemente come esistono già dei modi per aggiungere queste funzionalità, senza l’utilizzo di una service mesh. Questo approccio è stato ispirato dagli autori di [37].

Traffic Management. La gestione del traffico in Kubernetes è gestita dalla risorsa *Ingress*, la quale consente di esporre un cluster al mondo esterno. Per usare la risorsa Ingress è necessario un *controller di ingresso*, il cui compito è gestire gli oggetti Ingress e assicurarsi che i pod all’interno del cluster ricevano correttamente il traffico. I controller di ingresso sono pensati per gestire il traffico dal mondo esterno verso il cluster. Se, per esempio, volessimo limitare la quantità di traffico verso un particolare microservizio di back-end all’interno dello stesso cluster, la risorsa Ingress non sarebbe la più adatta.

Observability. Kubernetes offre una vasta varietà di tecniche e metriche per monitorare lo stato dei microservizi. Difatti è possibile ampliare l’insieme di funzionalità offerte in base al controller d’ingresso scelto e pure aggiungerne di personalizzate all’interno del microservizio stesso. Tramite tool come *Prometheus* e *Grafana* è possibile usare le metriche per effettuare particolari decisioni sull’architettura dell’applicazione. Sarebbe interessante visualizzare il traffico tra microservizi e i collegamenti logici tra essi, in quanto in un’architettura a microservizi questi ultimi comunicano spesso tra di loro. Per fare ciò si potrebbero usare un tool come *DataDog* oppure un’alternativa open-source come *Cilium*.

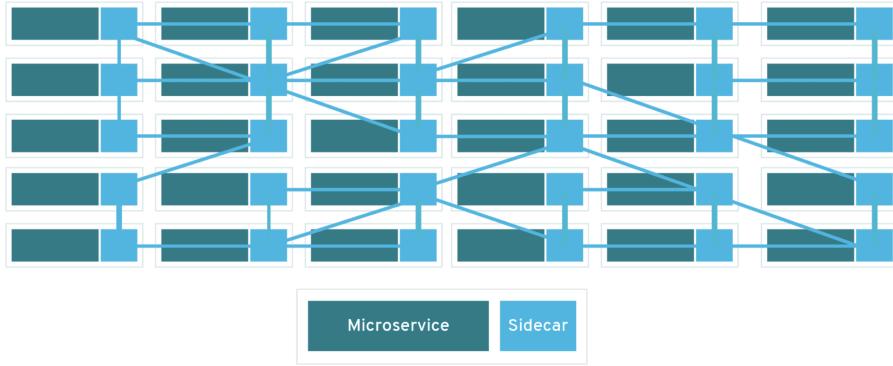


Figura 3.1: Sidecar proxies e la rete di mesh che creano (figura da [24]).

Security. Nel capitolo precedente si è vista la risorsa NetworkPolicies di Kubernetes, che limita la comunicazione tra microservizi a livello 3-4 dello stack ISO/OSI. Intuitivamente si potrebbe pensare che le network policies si comportino sostanzialmente come un *firewall*, nativo di Kubernetes. Usare solo le network policies non è sufficiente per gestire efficacemente la sicurezza tra pod. Per raggiungere un adeguato livello di sicurezza all'interno dell'applicazione a microservizi sarebbe consigliabile usare meccanismi di sicurezza che permettano di implementare l'autorizzazione tra i pod, in base alla loro identità, e di crittografare l'inter-comunicazione tra microservizi.

Come brevemente accennato, esistono metodi e strumenti che permettono di aggiungere le funzionalità di gestione del traffico, observability o sicurezza all'interno dell'applicazione a microservizi. Si potrebbe pensare di adoperare la miglior tecnologia per risolvere ciascuna di queste necessità, ma una sola tecnologia che offre la possibilità di gestire tutto questo contemporaneamente è senza dubbio più elegante. Questo è il grado di flessibilità che una service mesh offre.

Ricapitolando, una service mesh disaccoppia la *business logic* dalla logica di inter-comunicazione tra microservizi, delegando quest'ultima responsabilità al sidecar proxy, cercando di offrire contemporaneamente un'unica soluzione per la gestione del traffico, dell'observability e della sicurezza di un'infrastruttura a microservizi.

Nelle sezioni successive vedremo l'architettura generale delle service mesh e ci focalizzeremo sulle funzionalità di sicurezza che esse offrono. Le altre due classi di funzionalità, nonostante siano molto interessanti, non verranno considerate in quanto esulano dagli argomenti di questa tesi. Procederemo il cammino verso la scoperta del mondo delle service mesh considerando due particolari esempi: **Linkerd** e **Istio**. Nonostante esistano diverse implementazioni, come *Kuma*, *Cilium*, *Traefik Mesh* e *Consul Connect*, le quali sono valide alternative, queste due sono le più emblematiche.

3.1 Architettura di una service mesh

Come discusso in precedenza, i proxy implementano un insieme di funzionalità che permettono ai vari microservizi di comunicare. Viene gestito sia il traffico ingress sia il traffico egress, quindi tecnicamente siamo in presenza di un *proxy* che funge anche da *reverse proxy*.

L'architettura delle service mesh viene divisa tra **data plane** e **control plane**, fig. 3.2, dove nel primo troviamo i proxy discussi fin'ora e nel secondo un insieme di processi che gestiscono la coordi-

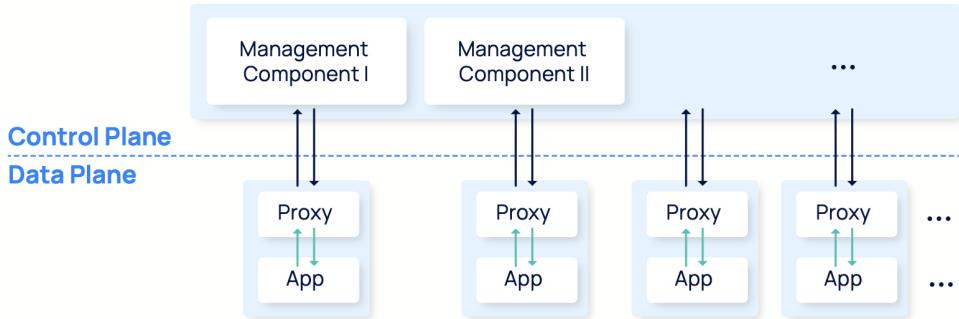


Figura 3.2: Control plane e data plane di una generica service mesh (figura da [32]).

nazione tra proxy. Il data plane si focalizza sull’inter-comunicazione tra microservizi, implementando le funzionalità necessarie per le varie chiamate tra microservizi. Il control plane fornisce funzionalità come:

- *Service discovery* - decide verso quale istanza di un microservizio inviare una determinata richiesta;
- *mTLS* - implementa la mutua autenticazione tra microservizi;
- *Policy information* - recupera quali richieste sono consentite; etc.

Analizzando meglio i proxy si nota che sono dei proxy TCP di livello 7, in quanto gestiscono la comunicazione tra microservizi basandosi su protocolli del livello applicativo, i quali si appoggiano al protocollo di trasporto TCP [32]. La scelta di quale proxy usare dipende da scelte implementative di ciascuna service mesh. **Envoy** [4] è una scelta molto comune, usata per esempio da *Istio*, *Cilium* e *Kuma*. A livello implementativo, in Kubernetes, un sidecar proxy è semplicemente un container in esecuzione nello stesso pod in cui è contenuto il microservizio associato ad esso. Si veda fig. 3.3.

Linkerd

La prima vera service mesh, la quale ha coniato il termine “service mesh”, è **Linkerd** [15]. In fig. 3.3 è possibile vedere come il control plane contenga diversi componenti che si occupano del service discovery (destination), di mTLS facendo da *TLS certificate authority* (identity) e di iniettare nei pod il linkerd-proxy (proxy-injector). A differenza di altre implementazioni, dove si predilige l’uso di Envoy come proxy, gli sviluppatori di Linkerd hanno preferito implementarne uno in *Rust* chiamato **Linkerd2-proxy**.

Istio

Introdotta da Google, IBM e Lyft nel 2016, Istio è una delle service mesh più usate [28]. In fig. 3.4 è possibile vedere come l’architettura differisce da quella di Linkerd. Innanzitutto il control plane è formato da un singolo componente, ***istiod***, che si occupa di *service discovery*, *certificate authority* e *proxy-injection*. Istio è un esempio di service mesh che utilizza **Envoy** come proxy, scritto in C++ invece che Rust.

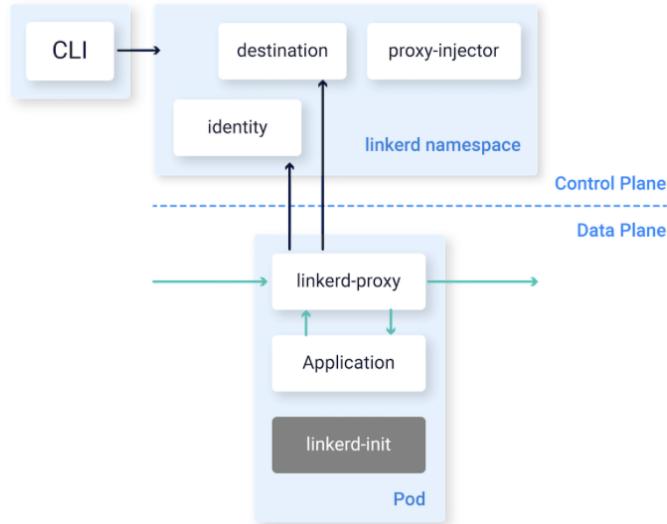


Figura 3.3: Control plane e data plane di Linkerd (figura da [15]).

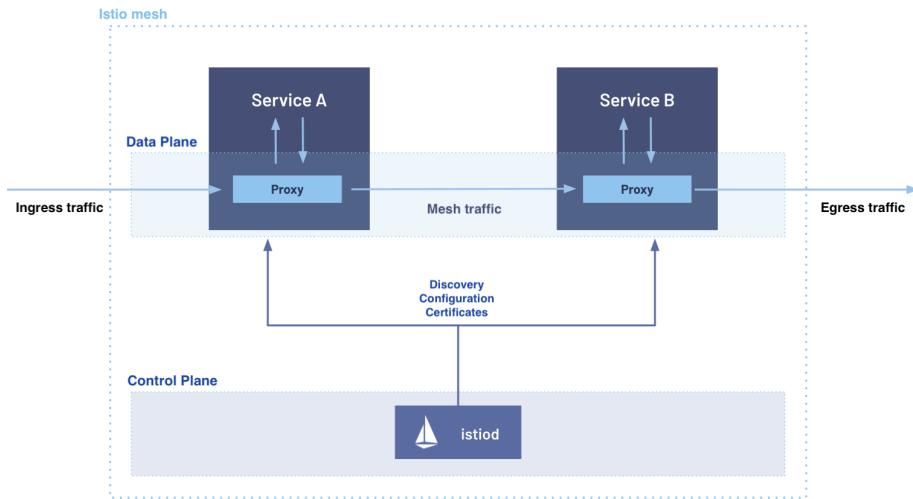


Figura 3.4: Control plane e data plane in Istio (figura da [5]).

3.2 mTLS

Definire *mTLS* è relativamente semplice: *mTLS* è un TLS dove anche il client è autenticato [31]. Questa definizione presuppone la conoscenza di cos'è TLS e quali garanzie fornisce ad una connessione.

TLS è un protocollo che opera a livello 5-6 dello stack ISO/OSI, rendendo sicuro il protocollo TCP. Per esempio HTTPS non è nient'altro che il protocollo HTTP con l'aggiunta di TLS, quindi ogni volta che si visita una pagina web con HTTPS stiamo usando TLS. TLS garantisce tre servizi:

- Autenticazione;
- Confidenzialità dei dati;
- Integrità dei dati.

L'autenticazione in TLS è unidirezionale, quindi solamente una parte della comunicazione è autenticata. Tipicamente il client autentica il server, quindi riceve un certificato che verifica l'identità del server, ma il server è ignaro dell'identità del client.

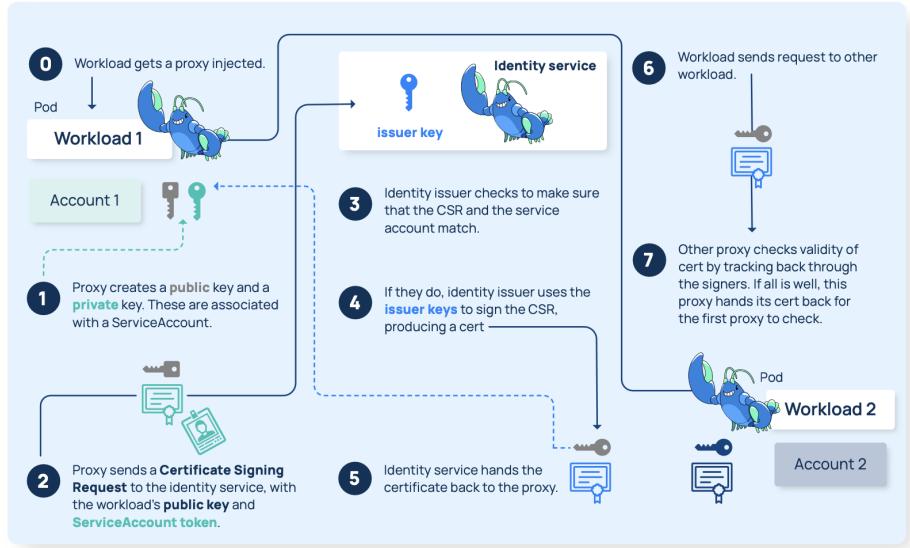


Figura 3.5: mTLS (figura da [31]).

mTLS permette l'autenticazione in entrambi i sensi, fig. 3.5, e in un'architettura a microservizi è essenziale non solo autenticare il microservizio verso il quale si effettua la richiesta (server), ma anche quello da cui si riceve la richiesta (il client).

Linkerd

Linkerd abilita automaticamente mTLS per tutto il traffico TCP tra i pod “meshati” (pod che contengono un proxy) e tra i componenti del control plane [15]. Linkerd intraprende un approccio più minimalista, automatizzando tutto il processo di configurazione di mTLS, diminuendo il grado di personalizzazione.

Istio

Istio permette una configurazione più dettagliata e flessibile tramite la risorsa **PeerAuthentication**, decidendo se applicare o meno mTLS a livello di namespace, mesh intera o singolo pod [5]. In particolare è possibile scegliere tra tre modalità:

- **STRICT** - è consentito solo il traffico protetto da mTLS;
- **PERMISSIVE** - è consentito sia traffico protetto da mTLS e non;
- **DISABLE** - mTLS è disabilitato.

3.3 Policy di Sicurezza

Nella sezione 2.3.5 abbiamo discusso il concetto di network policies presente in Kubernetes, concludendo che sono responsabili solamente delle comunicazioni a livello 3-4. Vista la natura dei proxy, tramite le service mesh è possibile scrivere policy che controllano il traffico a livello applicativo, per esempio relativamente a protocolli come *HTTP* o *gRPC*. Implementare policy a questo livello permette un grado di espressività maggiore, permettendo per esempio solo richieste protette da mTLS, controllando l'identità dei microservizi.

Linkerd

Linkerd può far rispettare le policy solo laddove è presente il proxy, quindi sarebbe opportuno garantire che ogni pod appartenga alla rete di mesh attraverso la modalità *High Availability (HA)* [14].

Linkerd offre due possibilità, applicare delle *default policies* ad ogni pod oppure applicare un insieme di *Custom Resource Definition (CRD)* che permette di definire policy personalizzate per ogni servizio. Questi meccanismi dovrebbero lavorare in contemporanea, quindi per esempio applicando una default policy che vietи qualsiasi comunicazione e poi definendo policy opportune per il livello di granularità desiderato [13].

Tra le default policies troviamo:

- ***all-unauthenticated*** - abilitata di default, permette tutte le comunicazioni;
- ***all-authenticated*** - permette le richieste solo dai microservizi appartenenti alla rete di mesh;
- ***cluster-authenticated*** - permette le richieste solo dai microservizi, appartenenti alla rete di mesh, all'interno dello stesso cluster;
- ***deny*** - vieta tutte le richieste;
- ***audit*** - simile a *all-unauthenticated*, ma si registrano log relativi al traffico e metriche.

Per riuscire a scrivere policies ad un livello di granularità più fine vi sono varie risorse, e come suggerito dagli autori di Linkerd [15], il pattern generale per applicarle è il seguente:

1. Indicare un insieme di pods e una porta a cui verrà applicata una policy tramite la risorsa ***Server***;
2. Opzionalmente, tramite la risorsa ***HTTPRoute*** indichiamo un sottoinsieme di traffico HTTP abilitato verso la risorsa ***Server*** (utile per esempio per permettere solo richieste GET e invece vietare le richieste DELETE e POST);
3. Tramite risorse quali ***MeshTLSAuthentication*** o ***NetworkAuthentication*** indichiamo i client che sono autorizzati ad accedere a una certa risorsa ***Server***;
4. ***AuthorizationPolicy*** è la risorsa che contiene la vera e propria policy, creando un legame tra una risorsa ***Server*** (o alternativamente ***HTTPRoute***) e un client (***MeshTLSAuthentication*** o ***NetworkAuthentication***).

Intuitivamente, a questo punto quando viene effettuata una richiesta il funzionamento dovrebbe essere il seguente [15]:

1. Se la porta della richiesta è associata ad una risorsa ***Server*** ed esiste un client autorizzato ad accedervi, **consenti** la comunicazione;
2. Se la porta della richiesta è associata ad una risorsa ***Server*** ma il client non è autorizzato ad accedervi, **vieta** la comunicazione;
3. Altrimenti, applica la *default policy* (da cambiare in ***deny*** per migliorare la sicurezza).

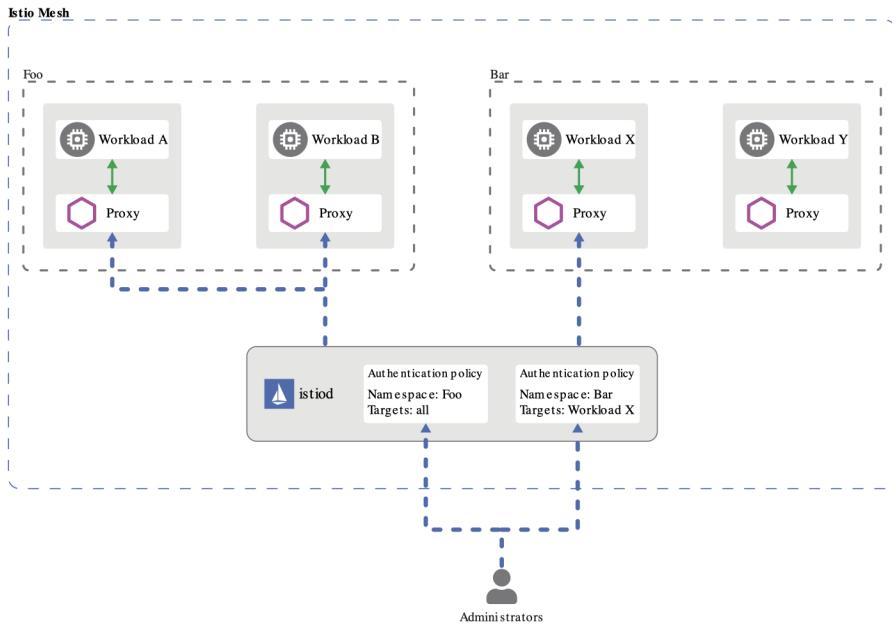


Figura 3.6: Funzionamento AuthorizationPolicy in Istio (figura da [5]).

Istio

A differenza di Linkerd, in questo caso si usa un approccio più sintetico, evitando di configurare numerose risorse per far rispettare una singola policy. Infatti, in Istio è presente soltanto una risorsa, **AuthorizationPolicy**, che permette di scrivere policy [5].

Quando nessuna **AuthorizationPolicy** è applicata allora tutte le comunicazioni sono consentite, altrimenti quando arriva una richiesta la si valuta rispetto l'**AuthorizationPolicy** specifica e si accetta o nega la richiesta [5].

Tra i campi più importanti troviamo:

- **namespace** e **spec/selector** - è possibile indicare a quale risorsa Kubernetes viene applicata la policy. Nel caso in cui l'ultimo campo sia assente la policy viene applicata all'intero namespace;
- **action** - specifichiamo la natura della policy, se di tipo **allow** (default se non specificata), **deny** o **audit**;
- **rules** - In base alla natura delle policy vengono valutate delle regole scritte nella forma (SOURCE, OPERATION, CONDITION), indicando chi effettua la richiesta, in cosa consiste la richiesta e in quali condizioni è stata effettuata. Tramite OPERATION è possibile indicare il particolare metodo usato nell'inviare la richiesta HTTP, esprimendo quindi la possibilità di limitare traffico HTTP solo per un sottoinsieme di metodi (comportamento equivalente a **HTTPRoute** di Linkerd).

3.4 Applicazione caso di studio

In fig. 3.7 è possibile vedere la stessa policy descritta nella sezione 2.4 ma implementata usando il concetto di policy a livello applicativo offerto da Linkerd. La stessa policy è possibile esprimerla con Istio apportando le opportune modifiche.

```

apiVersion: policy.linkerd.io/v1beta3  apiVersion: policy.linkerd.io/v1alpha1
kind: Server                           kind: AuthorizationPolicy
metadata:
  name: authors-server
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: authors
      project: booksapp
  port: service
  metadata:
    name: authors-auth
  spec:
    targetRef:
      group: policy.linkerd.io
      kind: Server
      name: authors-server
  requiredAuthenticationRefs:
  - name: webapp
    kind: ServiceAccount

```

Figura 3.7: Authorization Policy per il microservizio authors usando Linkerd come service mesh (si veda fig. 2.8 per la versione equivalente in Kubernetes).

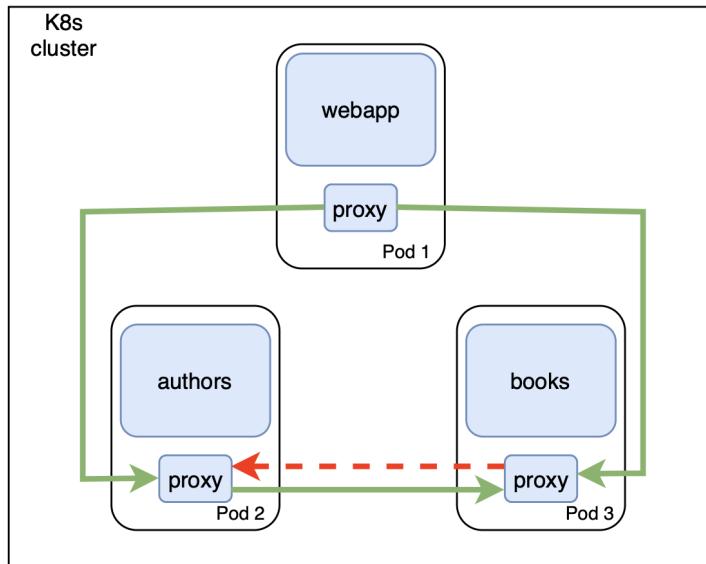


Figura 3.8: I tre microservizi del caso di studio dopo aver aggiunto i proxy di Linkerd.

Per riuscire a far rispettare la policy dobbiamo definire le due risorse ***Server*** e ***AuthorizationPolicy***. ***Server*** si applica al pod che contiene il microservizio `authors` tramite la porta del servizio stesso (7001, in fig. 2.8). ***AuthorizationPolicy*** determina che soltanto il microservizio `webapp`, previo opportuno controllo della sua identità tramite il concetto di ***ServiceAccount***, possa accedere al ***Server*** associato a `authors`.

ServiceAccounts è un modo *naïve* per implementare il concetto di identità, presente intrinsecamente in Kubernetes.

4

Policy-as-Code: OPA

L'***Open Policy Agent (OPA)*** è un motore di policy che permette di far rispettare policy in un’architettura a microservizi [1]. OPA è un progetto CNCF iniziato nel 2016.

Le politiche OPA sono espresse tramite un linguaggio di query dichiarativo, ***Rego***, progettato per definire le query su complesse strutture dati gerarchiche, es. JSON. Poiché le query Rego sono asserzioni sui dati memorizzati in OPA, queste possono essere utilizzate per individuare situazioni anomale, come per esempio violazioni delle regole definite, violando quindi lo stato previsto del sistema [17]. Quando un microservizio vuole inviare una richiesta, fornisce un input in JSON a OPA e lo interroga rispettivamente su dei dati memorizzati all’interno del motore e su delle regole scritte in Rego. Quindi, le politiche scritte in Rego e i dati memorizzati vengono valutati rispetto all’input in JSON fornito dal microservizio e, di conseguenza, OPA genera una decisione di politica, si veda fig. 4.1. È interessante notare che queste decisioni non si limitano a una semplice risposta “consenti/nega”, in quanto sono il risultato di una query [17]. Più nel dettaglio, il risultato è un dato semi-strutturato complesso che contiene qualsiasi tipo di informazione. L’output restituito sarà quindi in formato JSON.

OPA è uno strumento abbastanza flessibile poiché sia il motore che il linguaggio delle policy sono indipendenti dal dominio, in quanto si usano dati strutturati come input [17]. Quindi fornisce un meccanismo universale e generico per far rispettare policy in sistemi con tecnologie diverse in uso.

OPA può essere usato sia come un processo separato in esecuzione, che comunica con i microservizi tramite chiamate REST (*OPA-server-mode*), oppure integrato all’interno di un microservizio come libreria *Golang (OPA-SDK-mode)*.

4.1 Rego

Rego si è ispirato a *Datalog* [29], un linguaggio di query derivato a sua volta da *Prolog*, aggiungendo la possibilità di manipolare documenti strutturati come JSON [1]. Quindi Rego, come già detto, è un linguaggio dichiarativo che permette di scrivere regole, con lo scopo di indicare come arrivare a prendere una decisione deterministica. Una policy scritta in Rego è semplicemente un insieme di regole [39]. Le regole più basilari possono essere definite in termini di valori scalari (stringhe, numeri, booleani o nulli) o compositi (oggetti o insiemi).

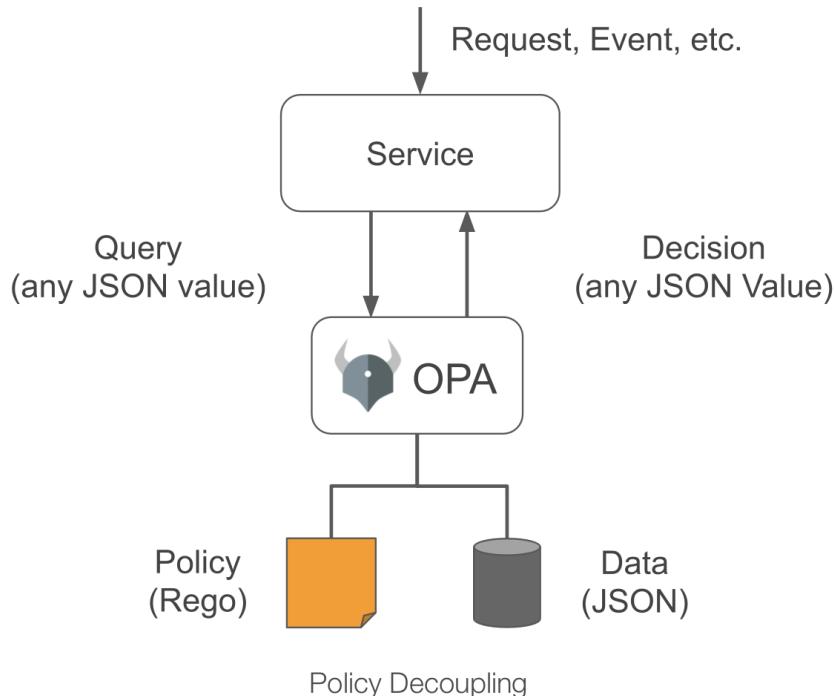


Figura 4.1: Architettura dell'interazione tra OPA e i microservizi (Figura da [1]).

Gli oggetti sono raccolte composte da coppie chiave-valore non ordinate, mentre gli insiemi sono raccolte che contengono valori univoci non ordinati. Rego non genera solo un valore booleano come risultato, infatti è possibile restituire valori scalari o compositi come risultato di una query.

È possibile scrivere regole multi-linea, permettendo quindi di esprimere regole più complesse. In questo caso ogni linea della regola deve portare ad una valutazione *true*, comportandosi come una congiunzione di *statement*. Si veda fig. 4.2 per un esempio di regola multi-linea dove il risultato non è un semplice valore booleano ma una stringa.

L'esempio in fig. 4.2 ci permette di discutere di un aspetto di Rego molto importante. A riga 7 e 8 troviamo la dichiarazione di due regole, dal nome rispettivamente *x* e *y*. Queste due regole sono annidate nella regola *MSG*, quindi nell'output non vedremo il valore restituito da queste due regole, in quanto è come se si comportassero come regole locali. Nulla avrebbe vietato di dichiarare le due regole fuori dal blocco della regola *MSG*, in questo caso però avremmo visto il valore restituito come output.

In Rego è possibile scrivere regole incrementali, ovvero ri-dichiarare una regola con lo stesso nome in modo da unire le varie dichiarazioni. Quindi possiamo affermare che la valutazione delle regole avviene sia tramite congiunzione che tramite disgiunzione. Si valutano le singole linee delle varie dichiarazioni di una regola, si effettua l'AND logico tra esse, portando ad una valutazione finale della dichiarazione in oggetto. Infine si valuta il valore finale effettivamente restituito in output, facendo l'OR logico tra le diverse valutazioni finali delle singole dichiarazioni.

L'esempio in fig. 4.3 permette di discutere meglio questo comportamento. Innanzitutto, vista la mancanza di una semantica per il linguaggio Rego, non è chiaro a priori il comportamento dei programmi scritti in questo linguaggio, per questo motivo, senza perdita di generalità, supponiamo che Rego valuti le regole in ordine. Si valuta la prima dichiarazione di *ALLOW*, restituendo valore *true* in quanto l'input

The screenshot shows a code editor on the left containing a Rego rule definition:

```

1 package play
2
3 import rego.v1
4
5 msg := "hello world" if {
6     input.source == "x"
7     x := 42
8     y := 12
9     x > y
10}
11
12

```

To the right, there are three panels: INPUT, DATA, and OUTPUT.

- INPUT:**

```

1 { "source": "x"
2 }
```
- DATA:** (empty)
- OUTPUT:**

```

Found 1 result in 88µs.
1 {
2   "msg": "hello world"
3 }
```

Figura 4.2: Un esempio di regola multi-linea in Rego (usando *The Rego Playground* [38]).

fornito soddisfa la riga 9 e la riga 10 (fig. 4.3), successivamente si valuta la seconda dichiarazione, portando a valore false. In conclusione si effettua l'OR tra le due valutazioni portando al valore finale *true*.

Definizione di una regola in Rego

Sia una regola definita come $\text{RULE} = \{s_{ij} \mid i = 1, \dots, k; j = 1, \dots, n_i\}$ dove:

- k è il numero di dichiarazioni della regola;
- n_i rappresenta il numero di *statement* nella i -esima dichiarazione.

La regola viene valutata come $\bigvee_{i=1}^k \left(\bigwedge_{j=1}^{n_i} s_{ij} \right)$.

Esempio

Consideriamo la regola ALLOW in fig. 4.3. Sia ALLOW definita come $\text{ALLOW} = \{s_{11}, s_{12}, s_{21}, s_{22}\}$, dove le s_{ij} sono

- $s_{11} : \text{input.source} == "a"$
- $s_{12} : \text{input.dest} == "b"$
- $s_{21} : \text{input.source} == "b"$
- $s_{22} : \text{input.dest} == "c"$

In questo esempio $k = 2$ e $n_1 = n_2 = 2$, ossia sono presenti due dichiarazioni della regola ALLOW, entrambe con 2 *statement* al loro interno.

La regola ALLOW viene valutata nel seguente modo: $(s_{11} \wedge s_{12}) \vee (s_{21} \wedge s_{22}) = \text{true}$.

The screenshot shows a code editor on the left containing a Rego script named `play.rego`. The script defines two `allow` rules. The first rule allows traffic from source "a" to destination "b". The second rule allows traffic from source "b" to destination "c". An `INPUT` panel shows a JSON object with fields `source` and `dest`. An `OUTPUT` panel shows the result of the query, indicating that the `allow` field is set to `true`.

```

1 package play
2
3 import rego.v1
4
5 default allow := false
6
7 # Dichiaraione 1 della regola allow
8 allow if {
9     input.source == "a"
10    input.dest == "b"
11 }
12
13 # Dichiaraione 2 della regola allow
14 allow if {
15     input.source == "b"
16     input.dest == "c"
17 }
  
```

INPUT	DATA	OUTPUT
<pre> 1 { "source": "a", "dest": "b" } </pre>		
		Found 1 result in 131µs.
		<pre> 1 { 2 "allow": true 3 } </pre>

Figura 4.3: Un esempio di regola incrementale in Rego (usando *The Rego Playground* [38]).

4.2 Principali differenze tra Policy-as-Code e Policy-as-Yaml

Usare una policy engine come OPA e un linguaggio come Rego permette una maggiore espressività nello scrivere policy. Le policy scritte in Rego rappresentano un vantaggio rispetto alle policy scritte con gli strumenti visti nei capitoli precedenti, *Policy-as-Yaml*, le quali sono delle semplici configurazioni statiche. Infatti, usando solamente le *Network Policies* native in Kubernetes oppure le *Authorization Policy* offerte da una service mesh, non è possibile esprimere policy di accesso che dipendono da dei dati da interrogare oppure da uno stato. In particolare non sarà possibile esprimere policy per scenari quali:

- Contatore** - Consentire a un determinato utente di poter accedere ad un servizio al più n volte, dopodiché sarà necessaria una licenza;
- Tre microservizi** - Consentire a due microservizi A e B di comunicare tra loro, fintantoché un altro microservizio C non inizi a comunicare con il microservizio B.

Difatti, per chiarire ulteriormente le idee, le policy di accesso possono essere suddivise in tre principali categorie:

- Statiche o stateless* - Come quelle esprimibili in Kubernetes o tramite una service mesh, mediante file YAML;
- Stateless basate su dei dati da interrogare* - Come quelle esprimibili in Rego, usando i dati forniti a OPA, senza la possibilità di modificarli;
- Stateful basate su dei dati da interrogare* - Con la possibilità di modifica dei dati.

Grazie alla sua espressività, Rego permette di definire regole basate sui dati (`data.json`), a esso forniti tramite OPA. Essendo un linguaggio dichiarativo con operatori logici, Rego facilita la scrittura di policy condizionali complesse. Tramite Rego è possibile esprimere policy *stateless basate su dei dati da interrogare*, ma vedremo come nel capitolo 5 sarà possibile estendere Rego e OPA al fine di riuscire ad esprimere policy *stateful basate su dei dati da interrogare*.

```

1 package exemplerego
2
3 import rego.v1
4
5 default allow := false
6
7 allow if {
8     input.user == "fabio"
9     data.counter > 0
10 }
11
12
13

```

Figura 4.4: Policy per lo scenario **Contatore**.

```

1 package exemplerego
2
3 import rego.v1
4
5 default allow := false
6
7 allow if {
8     input.source == "b"
9     input.dest == "c"
10    data.a_to_b == false
11 }
12
13 allow if { input.source == "a"; input.dest == "b" }

```

Figura 4.5: Policy per lo scenario **Tre Microservizi**.

4.3 Implementazione di due policy tramite Rego

In questa sezione si presentano i due scenari **Contatore** e **Tre Microservizi**, fornendo una possibile soluzione per le policy da far rispettare usando Rego come linguaggio. Come probabilmente già intuito, le policy per questi due scenari appartengono alla categoria *stateful basate su dei dati da interrogare* e nella sezione precedente è stato detto che in Rego non è possibile esprimere questo tipo di policy. Difatti, l'implementazione delle policy presentate in questa sezione è corretta ma incompleta, in quanto manca la parte di modifica dei dati, che è demandata ad altre componenti. Dunque, la reale soluzione ai due scenari elencati sopra verrà fornita nel capitolo 5, in quanto è necessaria l'implementazione di un *tool* aggiuntivo per permettere di esprimere le policy in Rego e di farle rispettare tramite OPA.

4.3.1 Policy scenario Contatore

Lo scenario da soddisfare è quello in cui un determinato utente, es. FABIO, possa accedere a un microservizio (es. consultazione libri) per un determinato numero di volte, in quanto superato questo limite è necessario l'acquisto di una licenza. Il funzionamento è relativamente semplice. Supponiamo che l'utente FABIO invii una richiesta GET al servizio di consultazione libri. Quando quest'ultimo riceve la richiesta, chiede a OPA di valutare la policy relativamente ai *data* a cui OPA fa riferimento e all'*input* che gli fornisce. La situazione potrebbe essere quella in fig. 4.4, dove con *data.counter > 0* si fa riferimento al dato *counter* salvato nel *data.json*.

4.3.2 Policy scenario Tre Microservizi

Lo scenario da soddisfare è quello in cui tre microservizi A, B e C possono comunicare tra di loro rispettando le seguenti regole:

1. B può comunicare con C;
2. A può comunicare con B;
3. Dal momento in cui si verifica 2. allora 1. non potrà più verificarsi.

Queste regole definiscono quindi che tutte le altre combinazioni di comunicazione non possono sussistere (es. C non può comunicare con B, C non può comunicare con A, etc.). La situazione potrebbe essere quella descritta in fig. 4.5, dove con *data.a_to_b == false* si fa riferimento al dato *a_to_b* salvato nel *data.json*.

4.4 Ulteriori tool per Policy-as-Code

OPA e Rego non sono l'unica soluzione per scrivere Policy-as-Code. Esistono infatti altre tecnologie emergenti che accolgono questa filosofia. Nell'ultimo periodo, nella letteratura sono emerse numerose proposte per accogliere la filosofia del Policy-as-Code e si è iniziato a parlare nello specifico di *linguaggi decisionali*. Cedar [25] e OpenFGA [22], come anche Rego, sono esempi di linguaggi decisionali. Rego è stato il primo linguaggio decisionale, Cedar e OpenFGA invece sono due linguaggi decisionali relativamente nuovi e in via di sviluppo. Vediamo brevemente le funzionalità che contraddistinguono i vari linguaggi. Rego, come già discusso ampiamente durante tutto il capitolo, è un linguaggio multi-dominio e abbastanza complesso da poter esprimere decisioni di qualsiasi natura. La forza di Rego è proprio quella di essere un linguaggio agnostico del contesto nel quale viene utilizzato.

Cedar, sviluppato da *Amazon Web Services (AWS)*, è un linguaggio dedicato all'autorizzazione a livello applicativo [10] e non supporta le decisioni *ReBAC*. Secondo gli autori di [30] è il linguaggio decisionale più semplice e il migliore per approcciarsi a scrivere Policy-as-Code.

OpenFGA, d'altro canto, non è un vero e proprio linguaggio decisionale, viene classificato dalla CNCF [30] come una piattaforma di autorizzazione. È la scelta ottimale per quanto riguarda le autorizzazioni *ReBAC*, dato che si utilizza un modello di autorizzazione basato sui grafi [11].

In definitiva, si può affermare che ognuno di questi linguaggi presenta delle valide funzionalità che motivano la scelta di usarne uno oppure un altro. Non è scopo di questa tesi determinare il miglior linguaggio decisionale, ma la scelta è ricaduta su Rego prevalentemente per la sua capacità di essere utilizzabile in diversi contesti e di restituire una struttura generica in output. Quindi si differenzia da Cedar, che restituisce come risultato un valore booleano, e da OpenFGA, che è più restrittivo in termini di policy di autorizzazione esprimibili. In ogni caso, nel prossimo capitolo verranno presentate più nel dettaglio le motivazioni che hanno portato alla scelta di OPA e Rego.

Rego, Cedar e OpenFGA sono linguaggi decisionali, la cui logica si concentra su chi può fare cosa in un sistema. La decisione viene presa rispetto ai dati e alle informazioni correnti, fornite al momento della query. Quindi, per scelta implementativa, non è loro responsabilità ricordarsi cosa è stato deciso nelle precedenti richieste. In questo caso, la filosofia di queste tecnologie è quella di essere *stateless*, cioè di non mantenere alcuno stato interno tra le varie richieste ricevute.

In definitiva, i linguaggi decisionali presentati, sono stati progettati per valutare le richieste ricevute e prendere una decisione, piuttosto che sul mantenere uno stato mutevole e persistente. Questa è una grossa deficienza per quanto riguarda l'esprimibilità di molte policy di sicurezza. Infatti, per poter esprimere le policy degli scenari **Contatore** e **Tre Microservizi** è inevitabile una modifica ai dati.

Per questo motivo, nel capitolo successivo, verrà presentato un programma *wrapper*, scritto in linguaggio *Golang* [23], che permette di gestire lo stato in OPA e di esprimere policy *stateful basate su dei dati da interrogare* in Rego.

5

OPA-Wrapper State Manager

Vista l'importanza di gestire policy di sicurezza nelle moderne architetture a microservizi, in questo capitolo verrà presentato un *wrapper* attorno a OPA e si discuterà come il suo impiego possa migliorare la gestione e l'implementazione di policy di decisione in molteplici contesti.

5.1 Fondamenti e motivazioni del wrapper

Per quanto concerne la gestione delle policy decisionali per architetture a microservizi, OPA è uno tra gli strumenti più versatili per gestire policy, grazie soprattutto alla sua intrinseca natura nell'integrarsi facilmente in diversi sistemi o architetture, indipendentemente dal linguaggio di programmazione utilizzato, e grazie al suo utilizzo di un linguaggio dichiarativo come Rego. OPA è quindi un policy engine agnostico dell'infrastruttura nella quale viene impiegato. Ciononostante, OPA non è sufficiente per gestire scenari in cui è necessario mantenere uno stato tra le diverse richieste. Infatti, per come è stato concepito, OPA è un sistema *stateless*, dove ogni decisione viene presa solamente in base ai dati attuali, senza conservare informazioni aggiuntive alle richieste precedenti. Questo può rappresentare un limite in contesti più complessi, come nei due scenari presentati nel capitolo precedente: **Contatore e Tre Microservizi**, dove è necessario gestire variabili incrementali oppure gestire l'interazione tra entità che evolve nel tempo. Per questo motivo, è di fondamentale importanza la necessità di sviluppare un *wrapper* che permetta di estendere OPA. Lo scopo principale è di introdurre il concetto di stato mutuabile nel tempo, tra le diverse query ricevute da OPA, aggiornando i *data* usati da essa. In particolare, l'obiettivo è quello di aggiungere uno strato a OPA che permetta di aggiornare dinamicamente i dati su cui le query operano, aggiungendo la capacità di salvare uno stato, che verrà usato dalle regole in Rego per prendere future decisioni.

La scelta di OPA come policy engine è stata motivata principalmente dai seguenti fattori:

- **Domain-agnostic** - OPA è ignaro dell'infrastruttura nella quale viene adoperato, quindi indipendentemente dal dominio, riceve query e restituisce un risultato [1];
- **Supporto di JSON** - OPA opera direttamente su dati strutturati come il formato JSON, quindi le richieste in input, così come i dati utilizzati per prendere decisioni, sono in formato JSON. Questo facilita eventuali estensioni, in quanto JSON è uno standard molto utilizzato e indipendente dal dominio in cui viene impiegato;

```

1 package play
2
3 import rego.v1
4
5 state[>18] := "true" if allow
6
7 state["user"] := {"age": age, "name": name} if {
8     name := input.user
9     age := input.age
10 }
11
12 state["msg"] := "you're allowed" if state[>18]
13
14 state["msg"] := "you're not allowed" if not state[>18]
15
16 default allow := false
17
18 allow if {
19     input.age >= 18
20 }
21

```

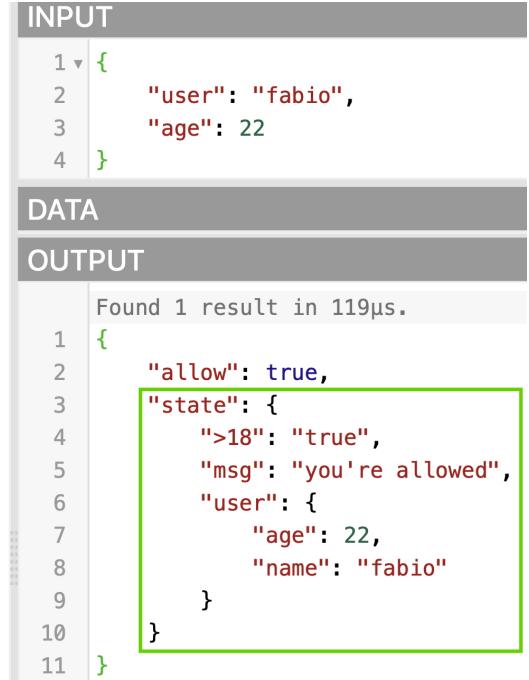


Figura 5.1: Esempio di come restituire lo stato in Rego (usando *The Rego Playground* [38]).

- **Output in formato JSON** - Questo è il principale motivo che ha guidato la scelta verso OPA. Infatti verrà sfruttata proprio questa funzionalità per restituire lo stato modificato dalle regole Rego. In effetti, come già discusso nella sezione 4.4, Rego è l'unico linguaggio decisionale per scopi generici, che non restituisce un semplice valore booleano come risposta ad una query.

Ricapitolando, il wrapper **OPA-Wrapper State Manager (OWSM)** introduce una componente essenziale, assente in OPA e Rego: la gestione dello stato. Questo wrapper aggiunge la capacità di aggiornare dati e memorizzarli nel tempo, integrandosi in maniera naturale con le policy già definite; ossia senza aggiungere particolari complicazioni, ma adoperando semplici principi nel modo di scrivere le policy, al fine di restituire lo stato modificato.

5.2 Principio chiave per salvare lo stato

Per riuscire a salvare lo stato è necessario adoperare un particolare stile nello scrivere policy in Rego. L'idea è quella di usare il concetto di valore composito presente in Rego, assimilabile ad un oggetto chiave-valore. Si scriverà quindi una regola di nome STATE che conterrà tutto ciò che vogliamo salvare come stato. Si veda fig. 5.1 per un semplice esempio. Il rettangolo verde racchiude ciò che vogliamo salvare come stato, nel *data.json* usato da OPA. Fatta questa considerazione, possiamo ora analizzare l'architettura del wrapper **OWSM**.

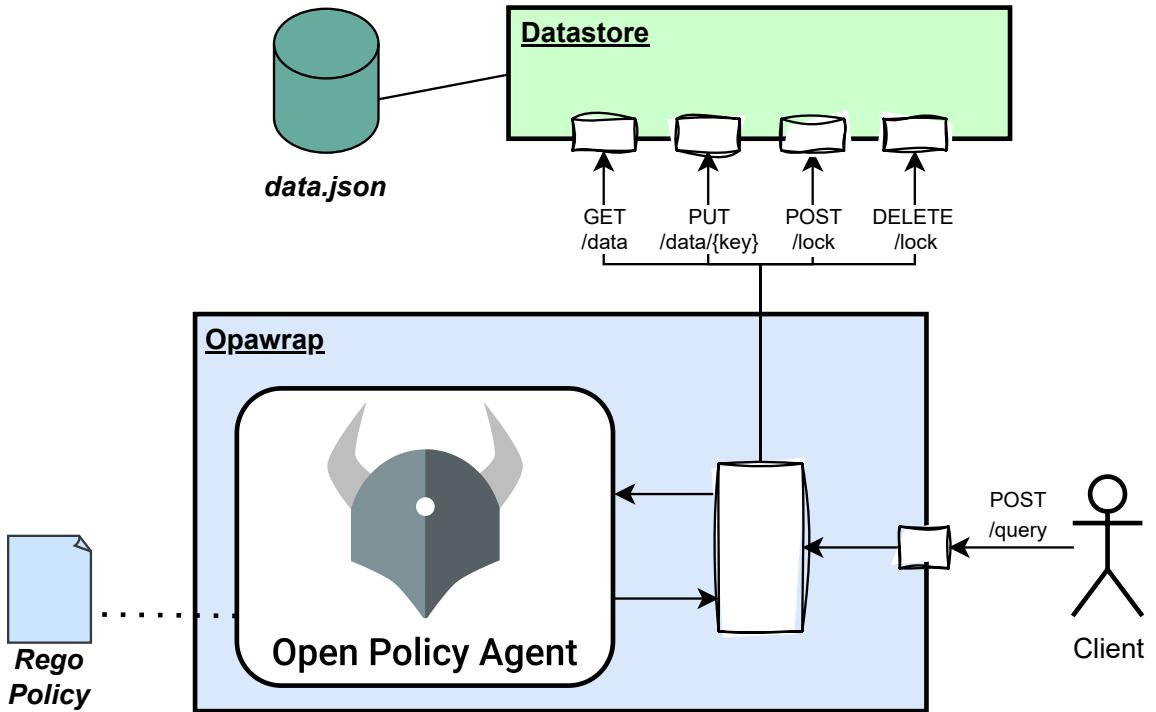


Figura 5.2: Architettura OSWM.

5.3 Architettura di OSWM

In fig. 5.2 è possibile vedere come, in realtà, sono presenti due moduli principali, ***datastore*** e ***opawrap***, che compongono l'**OWSM**.

Il modulo ***datastore*** si occupa di aggiornare i dati, usati da OPA per prendere decisioni. Mentre, il modulo ***opawrap***, è il vero e proprio wrapper che contiene tutta la *business logic* relativa alla ricezione della query, alla valutazione della stessa e alla restituzione del risultato.

Come è possibile vedere in fig. 5.2, sono presenti cinque *endpoint API*. Il client interagisce con **OWSM** tramite l'endpoint `/query`, inviando l'input della richiesta che si aspetta OPA, in formato JSON. Per poter modificare `data.json` e quindi salvare lo stato, ***datastore*** espone due endpoint principali: `/data` e `/data/{key}`. L'endpoint `/data` viene usato col metodo GET per poter prelevare i dati salvati nel `data.json`, mentre l'endpoint `/data/{key}`, viene usato col metodo PUT per aggiornare o aggiungere un dato nel `data.json`.

Facendo riferimento alla terminologia usata, chiariamo cosa si intende per un dato. Riferendosi all'esempio in fig. 5.1, un dato è la singola coppia chiave-valore presente nel blocco STATE `{...}` (i.e. "`>18`": "TRUE").

Quando un client invia una richiesta di valutazione di una query, questa viene reindirizzata a OPA. In base alla policy scritta in Rego e ai dati contenuti nel ***datastore***, OPA prende una decisione e la restituisce in output al client.

Come abbiamo già discusso nella sezione 5.2, l'output non presenterà solo la decisione presa, ma anche l'eventuale stato modificato. Per questo motivo, prima di restituire il risultato al client, bisogna asportare la parte relativa allo stato e salvarla nel ***datastore***.

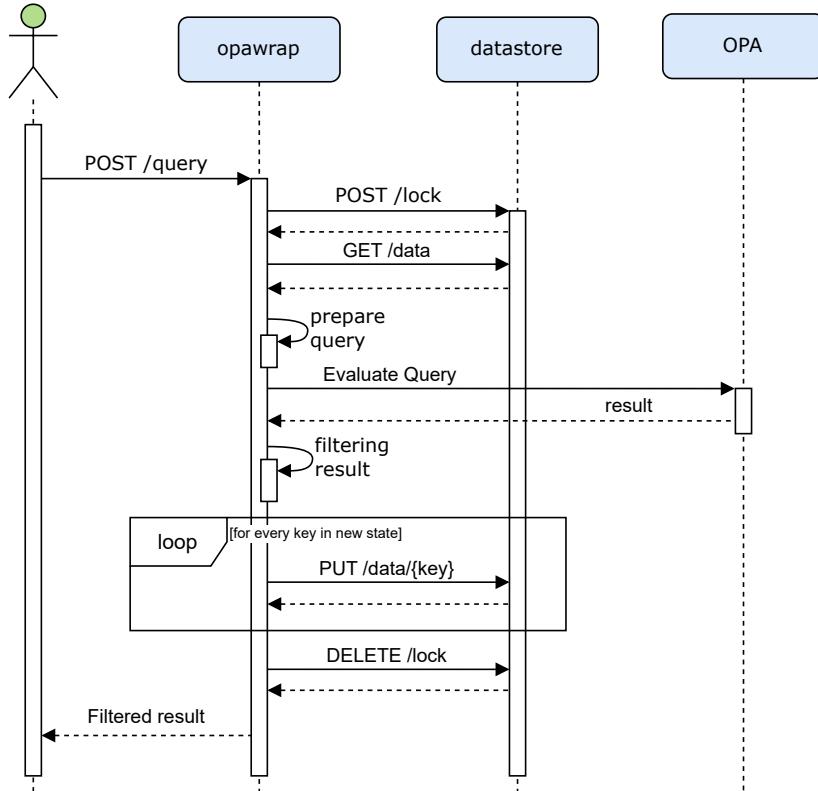


Figura 5.3: Interazione tra i vari moduli di OWSM.

Visto il contesto in cui OPA, e quindi pure **OWSM**, viene impiegato, è imprescindibile l'aggiunta di un meccanismo di sincronizzazione. In un'architettura a microservizi, il wrapper verrà usato come motore di decisione centralizzato dai vari microservizi, portando quindi a possibili *race conditions* nel momento in cui viene aggiornato lo stato.

Per questo motivo, è cruciale l'aggiunta di un meccanismo di sincronizzazione come il *mutex*. L'endpoint */lock*, usato con i metodi POST e DELETE, verrà quindi utilizzato per adempiere a questa necessità, ossia per effettuare il *lock* e l'*unlock* dei dati contenuti nel **datastore**.

5.4 Implementazione di OWSM

Discussa l'architettura generale di **OPA-Wrapper State Manager**, possiamo trattare gli aspetti più concreti e le scelte di implementazione fatte durante la scrittura del codice.

Per chiarire ulteriormente le idee, in fig. 5.3 è possibile vedere come i vari componenti di **OWSM** interagiscono tra loro, sfruttando l'astrazione offerta dal *sequence diagram* di *UML*.

OPA è stato sviluppato usando il linguaggio *Go*, quindi nel wrapper viene usato come libreria *SDK*. Le richieste a OPA vengono quindi effettuate tramite l'uso delle API offerte dal *package*, messo a disposizione dagli sviluppatori del policy engine. Il motivo principale nel scegliere *Go* come linguaggio di programmazione è stato quindi sfruttare proprio questa funzionalità.

Come già intuito, sia **datastore** che **opawrap** sono server che rispettano lo stile di programmazione *RESTful*. In Go è possibile implementare un server sfruttando la libreria standard *net/http*, che

presenta un insieme esaustivo di funzioni per riuscire nell'intento. Non è stato quindi necessario avvalersi di altri *framework*, come per esempio *gorilla* oppure *gin*.

5.4.1 Datastore

Iniziamo la discussione analizzando per prima cosa gli aspetti più importanti riguardo le scelte implementative effettuate su ***datastore***.

Come si può vedere nel listing 5.1, il modulo ***datastore*** contiene la dichiarazione di un nuovo tipo **ds**. Questo tipo è una combinazione tra un **sync.Mutex** e un **map[string]any**. In altre parole, abbiamo dichiarato che il modulo ***datastore*** sarà composto da due componenti essenziali: uno **store**, dove salvare lo stato, e un **lock**, usato per sincronizzare le varie richieste ricevute.

```

1 type ds struct {
2     lock sync.Mutex
3     store map[string]any
4 }
```

Listing 5.1: Struttura del datastore

Per implementare il meccanismo di sincronizzazione *mutex* ci si avvale semplicemente della struttura **Mutex** esposta dal pacchetto **sync** presente in *Go*. Una qualsiasi variabile dichiarata essere di tipo **sync.Mutex** può usufruire dei metodi **Lock()** e **Unlock()**. Infatti, nel modulo ***datastore***, il meccanismo di sincronizzazione *mutex* è stato implementato semplicemente effettuando una chiamata al metodo corrispondente alla rispettiva operazione, sulla componente **lock** della struttura **ds**.

La componente **store** verrà usata per contenere coppie chiave-valore nello stile JSON. Quindi è stata dichiarata una componente di tipo **map[string]any**, ossia una mappa tra valori di tipo stringa a valori di qualsiasi altro tipo. Infatti, in *Go* è possibile indicare tramite **interface{}** un'interfaccia vuota, che rappresenta un tipo senza metodi e quindi a cui può essere associato qualsiasi valore. **any** è un semplice *alias* per **interface{}**. Come paragone di familiarità, si può assimilare il concetto di interfaccia vuota di *Go* all'oggetto **Object** di *Java*.

La componente **store**, intuitivamente, verrà usata per contenere i dati usati da OPA per prendere le decisioni, quindi sarà il fulcro della gestione dello stato. Per questo motivo si è deciso di salvare il suo contenuto all'interno di un file, dall'ovvio nome *data.json*, per rendere quindi lo stato persistente.

Considerati questi aspetti, discutiamo il modo in cui è stato implementato il vero e proprio server. Sostanzialmente, avvalendosi della libreria standard **net/http**, è possibile sviluppare un server tramite passaggi semplici e intuitivi.

Tralasciando la dichiarazione della variabile di tipo **ds**, l'aspetto fondamentale è la dichiarazione a riga 3, nel listing 5.2. Tramite questa dichiarazione si istanzia un oggetto di tipo ***http.ServeMux**, che funge da *multiplexer* HTTP, smistando le varie richieste ricevute all'*handler* adatto.

Lo smistamento avviene in base al metodo usato nella richiesta (GET, PUT, POST o DELETE) e all'endpoint API a cui si fa la richiesta. Nel caso specifico, come si vede nel listing 5.2, sono presenti quattro handler, il cui compito è quello di gestire la richiesta ricevuta nel modo più adatto possibile. Infine, il server per poter eseguire deve essere avviato e deve rimanere in ascolto su uno specifico indirizzo IP e una specifica porta. In questo caso, il server rimarrà in ascolto sull'indirizzo :8081, che indica **localhost:8081**.

```

1 var datastore ds
2 datastore.store = make(map[string]any)
3 mux := http.NewServeMux()
4
5 mux.HandleFunc("GET /data", handleData(datastore.store))
6 mux.HandleFunc("PUT /data/{key}", handleUpdate(datastore.store))
7 mux.HandleFunc("POST /lock", handleLock(&datastore))
8 mux.HandleFunc("DELETE /lock", handleUnlock(&datastore))
9
10 log.Fatal(http.ListenAndServe(":8081", mux))

```

Listing 5.2: Il cuore del server datastore

```

1 func handleUpdate(store map[string]any) http.HandlerFunc {
2     return func(w http.ResponseWriter, r *http.Request) {
3         key := r.PathValue("key")
4         body, err := io.ReadAll(r.Body)
5         var data any
6         err = json.Unmarshal(body, &data)
7         store[key] = data
8         log.Printf("Set %q to %v", key, data)
9         w.WriteHeader(http.StatusOK)
10    }
11 }

```

Listing 5.3: Salvataggio di una nuova chiave-valore nello store

Per mantenere un certo ordine di presentazione dei concetti, non verrà discussa l'implementazione di ognuno degli handler. Verrà analizzato nel dettaglio soltanto la funzione `handleUpdate` che è l'handler più interessante, in quanto permette di salvare il nuovo stato. In ogni caso, è possibile reperire il codice completo dalla pagina *GitHub*¹. Il codice riportato in questo capitolo contiene solamente le parti di spicco, in modo da focalizzarsi principalmente sulla logica operativa, evitando quindi eventuali punti in cui vengono gestiti gli errori.

Analizzando ulteriormente il listing 5.2, notiamo che il metodo `HandleFunc()` a riga 6, viene utilizzato passando due argomenti. Il primo argomento è una stringa, la quale indica che l'handler si applica nel momento in cui avviene una richiesta col metodo PUT all'API `/data/key`. Il secondo argomento deve essere il nome di una funzione che rispetta la seguente firma:

```
1 func (http.ResponseWriter, *http.Request)
```

Listing 5.4: Firma della funzione da passare all'handler

Nel caso specifico dell'handler questa firma è assai limitante, in quanto non è possibile passare ulteriori argomenti alla funzione. Visto l'obiettivo dell'handler, ossia di aggiungere dati all'interno dello `store`, è necessario passare quest'ultimo come argomento alla funzione. Per questo motivo è stato necessario sfruttare le funzioni di ordine superiore, facendo sì che la funzione `handleUpdate()` del listing 5.3 prenda in input lo store e restituisca una funzione dal tipo equivalente a quanto presentato nel listing 5.4. Questo principio è stato utilizzato per ognuno degli handler presenti.

Nel listing 5.4 sono stati presentati due tipi: `ResponseWriter`, che rappresenta un insieme di funzionalità per gestire e inviare una risposta al client che ha effettuato la richiesta; le informazioni di

¹<https://github.com/ffabionski/opawrapper-statemanager>

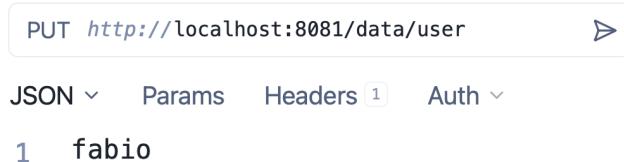


Figura 5.4: JSON della richiesta a /data/{key}.

quest'ultima sono contenute nel tipo `Request`. Entrambi i tipi sono esposti dal pacchetto `http` della libreria standard.

Finora è stato presentato il seguente endpoint `/data/{key}`, dove `{key}` è un semplice parametro che assume un valore arbitrario. Per esempio, rifacendosi alla fig. 5.1, considerando lo stato all'interno del rettangolo verde, si potrebbe effettuare una richiesta PUT al path `/data/user`. In questo caso `{key}=user`.

Le parti di codice importanti del listing 5.3 sono nelle righe 3, 6 e 7. Il tipo `Request` espone un metodo che offre la possibilità di recuperare il parametro associato a `{key}`. Successivamente si effettua un *unmarshal* del JSON ricevuto e si salva il valore nello `store` in base a `{key}`. Chiariamo le idee sfruttando l'esempio in fig. 5.4.

La chiamata tramite il metodo PUT a `localhost:8081/data/user` contiene come *body* della richiesta il JSON con il solo contenuto FABIO. Le variabili, nel listing 5.3, `key` e `data` avranno rispettivamente il valore "user" e `fabio`, di tipo `any`. Quindi l'assegnamento a riga 15 sarà `store["user"] = fabio`.

Come già accennato, lo `store` verrà salvato nel file `data.json`. Questo avverrà prima di liberare la variabile `datastore`, ossia prima di effettuare `Unlock()` del mutex.

Sono state descritte le principali funzioni del `datastore` e le scelte implementative effettuate durante il suo sviluppo. Ricapitolando brevemente, la responsabilità di questo modulo è quella di gestire i dati usati da OPA e di aggiornarli relativamente alle modifiche ricevute.

5.4.2 Opawrap

Questo è il modulo principale di **OWSM**, dove è contenuta tutta la *business logic*. In particolare le sue principali responsabilità sono:

1. Inviare le query a OPA, cosicché possa valutarle;
2. Decidere quando bloccare e sbloccare le risorse offerte da `datastore`;
3. Estrapolare lo stato dall'output fornito da OPA;
4. Restituire la valutazione della query al client.

Pure qui, come fatto per il modulo `datastore`, si è usata la libreria standard `net/http` per implementare le funzionalità che il server `opawrap` deve offrire. Facendo riferimento al listing 5.5, viene esposto solamente un endpoint API, quindi sarà presente un singolo *handler*, il quale intercetta le richieste effettuate col metodo POST a `localhost:8080/query`.

Si noti che in questo caso non è stato necessario sfruttare funzioni di ordine superiore per il secondo argomento alla chiamata `HandleFunc()`, in quanto non sono necessari parametri aggiuntivi alla firma presentata nel listing 5.4.

```

1 mux := http.NewServeMux()
2 mux.HandleFunc("POST /query", handleQuery)
3 log.Fatal(http.ListenAndServe(":8080", mux))

```

Listing 5.5: Il cuore del server opawrap

Se si analizza meglio il sequence diagram in fig. 5.3, si nota come il modulo *opawrap* effettui il maggior numero di interazioni con gli altri. In particolare si può dire che orchestra l'intera comunicazione.

Senza riportare l'intero codice della funzione, lo pseudo-codice nel listing 5.6 mostra le operazioni principali della funzione `handleQuery`.

```

1 func handleQuery(w http.ResponseWriter, r *http.Request) {
2     lock(datastoreURL)
3     retrieveInput(r.Body)
4     getState(datastoreURL, &data)
5     state, result = OPAevalQuery(data, input)
6     updateState(datastoreURL, state)
7     unlock(datastoreURL)
8     returnToClient(result, w)
9 }

```

Listing 5.6: Pseudo-codice della funzione `handleQuery`

Nello pseudo-codice nel listing 5.6 si è evitato di effettuare la dichiarazione esplicita delle variabili, per motivi di chiarezza nel leggere le varie operazioni. Indubbiamente, le operazioni più interessanti sono quelle che riguardano la gestione dello stato, quindi le righe 4-5-6 del listing 5.6.

Recupero dello stato corrente da datastore

La funzione `getState` si occupa di recuperare lo stato corrente presente nel datastore, ossia i dati che esso contiene.

```

1 func getState(baseURL *url.URL, state *map[string]any) {
2     baseURL.Path = "data"
3     client := http.Client{}
4     resp, err := client.Get(baseURL.String())
5     dataBytes, err := io.ReadAll(resp.Body)
6     err = json.Unmarshal(dataBytes, &state)
7 }

```

Listing 5.7: Recupero dei dati presenti nel datastore

Si effettua quindi una richiesta a `localhost:8081/data` tramite il `client`, si aspetta la risposta, la quale conterrà i dati in formato JSON. Si effettua l'`unmarshal` di questi e si salvano nell'indirizzo di memoria contenuto nella variabile puntatore `state`.

Ora, avendo i dati aggiornati e l'input ricevuto dal client che ha inviato la richiesta a `:8080/query`, possiamo delegare la valutazione della query a OPA.

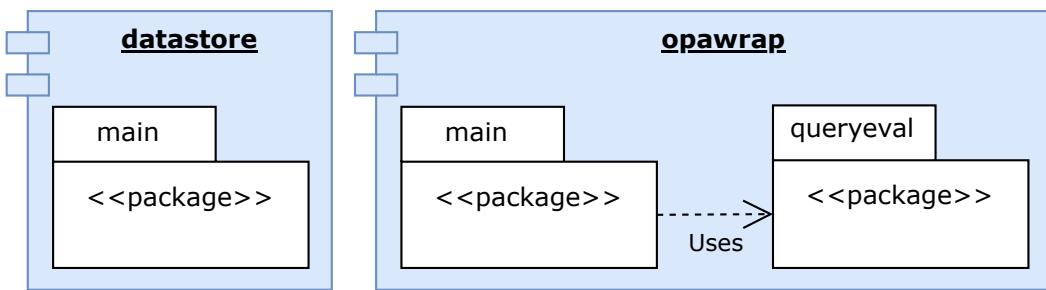


Figura 5.5: Moduli e package del progetto OWSM in Go

Valutazione delle query da parte di OPA

Prima di passare alla valutazione delle query da parte di OPA, deviamo il discorso visualizzando graficamente, in fig. 5.5, i *module* in Go e i package creati per **OWSM**.

Il modulo datastore presenta un unico package, `main`, che è obbligatorio in Go per tutti i file che presentano il metodo `main()` al loro interno. Stesso discorso per quanto riguarda opawrap, ma con l'aggiunta di un ulteriore package per la valutazione delle query: `queryeval`.

Questa è una scelta di design, per astrarre dal particolare policy engine usato per valutare le query. Quindi, presumibilmente, si potrebbero usare altri policy engine, per prendere decisioni riguardo le richieste ricevute. Come già discusso, il wrapper richiede che l'output restituito da un determinato policy engine sia in formato JSON.

Nello specifico caso di **OWSM** si usufruisce solamente di OPA come policy engine e quindi l'unico metodo esposto sarà per l'appunto `Opa()`.

Ritorniamo alla valutazione delle query da parte di OPA. Nella funzione `handleQuery`, indicata nello pseudo-codice del listing 5.6, la valutazione avverrà tramite la seguente riga di codice in Go

```
1 state, result := queryeval.Opa(data, input, w, r.Context())
```

Dove `data` e `input` contengono rispettivamente i dati e l'input da passare a OPA, d'altro canto, `w` e `r.Context()` verranno utilizzati rispettivamente per gestire la risposta verso il client (per es. restituendo codici HTTP) e per passare il contesto esatto associato alla richiesta ricevuta.

```
1 func Opa(data map[string]any, input any, w http.ResponseWriter,
2         ctx context.Context) (map[string]any, map[string]any)
```

Il package `queryeval` conterrà quindi l'unica funzione, il cui compito è quello di richiedere la valutazione a OPA della query ricevuta, usufruendo della libreria offerta da OPA. Per ulteriori informazioni si consulti la documentazione ufficiale di OPA [2]. Analizziamo nel dettaglio le parti rilevanti del corpo della funzione `Opa`, la cui firma è stata presentata sopra.

```
1 packageRego := "data.examplerego"
2 // data used by OPA
3 store := inmem.NewFromObject(data)
4 tx := storage.NewTransactionOrDie(ctx, store)
```

La variabile `packageRego` semplicemente indica di quali regole vogliamo ispezionare il risultato.

In questo caso indichiamo tutte le regole presenti in una policy Rego appartenente al package `examplerego`. Generalmente, nel caso in cui si voglia esaminare solamente l'output di una determinata regola, è sufficiente scrivere "`data.nomePackage.regola`".

Si ricorda che i dati da passare ad OPA, presenti nel `datastore`, sono stati precedentemente salvati nella variabile `data`, si veda lo pseudo-codice del listing 5.6 a riga 4. Per poter passare i dati ad OPA in formato di byte, è stato necessario memorizzare il contenuto di `data` in una variabile `store` dal tipo apposito `storage.Store`. OPA, per evitare inconsistenze indesiderate dei dati, richiede la creazione di una transazione al momento del caricamento dei dati.

Ora, possiamo creare un oggetto di tipo `rego` che verrà usato per valutare la query. Tramite il metodo `Query` indichiamo le regole di cui vogliamo sapere il risultato, tramite `Load` carichiamo il file `nome-policy.rego` contenente la policy scritta in Rego, tramite `Store` indichiamo i dati da utilizzare e tramite `Transaction` indichiamo la transazione.

```

1 // new rego object
2 re := rego.New(
3     rego.Query(packageRego),
4     rego.Load([]string{os.Args[1]}, nil),
5     rego.Store(store),
6     rego.Transaction(tx),
7 )

```

Una volta istanziato questo oggetto si può procedere con la preparazione della query e successivamente con la valutazione vera e propria di essa. C'è una distinzione tra le due fasi appena descritte, infatti nella prima fase si effettua il *parsing* controllando eventuali errori (i.e. errori nel formato dell'input/dati o errori dovuti alla compilazione della policy Rego); nella seconda fase si effettua la valutazione della query restituendo il risultato.

```

1 // prepare the rego object in order to evaluate the query
2 query, err := re.PrepareForEval(ctx)
3 // execute the prepared query and evaluate the result
4 rs, err := query.Eval(ctx, rego.EvalInput(input))

```

L'input inviato dal client verrà inviato a OPA solamente nel momento in cui si valuta la query, ossia al momento della chiamata `query.Eval`, tramite il metodo `EvalInput`. Questo perché si può riutilizzare lo stesso oggetto `rego`, una volta preparato per la valutazione, per valutare più query in base a input diversi. Ora, questo non è il nostro caso, in quanto l'obiettivo è proprio valutare una query con dati eventualmente modificati e quindi è necessario istanziare ogni volta un nuovo oggetto `rego`, con il nuovo stato.

Asportazione dello stato dal risultato ricevuto da OPA

A questo punto, dopo la valutazione della query, OPA restituisce come risultato la decisione presa. Il risultato restituito è in un formato sufficientemente complesso. È doveroso quindi deviare l'attenzione per capire effettivamente la sua struttura.

`Eval()` restituisce il tipo `ResultSet`, il quale è un array di `Result`, ossia rappresenta una collezione di output determinati dalla valutazione della query. Capiamo meglio il motivo per cui potrebbe verificarsi l'eventualità di avere più output.

Come sappiamo, vista la presentazione del linguaggio Rego nel capitolo 4, è possibile definire più regole all'interno della stessa policy, dove la valutazione di ognuna di queste regole produrrà un output. Quindi l'output di ogni regola sarà del tipo `Result`.

In realtà bisogna forzare OPA per restituire un `Result` per ogni regola valutata, passando al metodo `Query` la stringa "`data.example[_]`". Diciamo che questa opzione non è delle migliori, in quanto ogni `Result` conterrà l'output della regola rispettiva in base all'ordine in cui è stata definita nella policy Rego. Questo è il motivo per cui durante la scrittura di policy in Rego ci si può scontrare con il *warning* "`messy incremental rule`".

Per manipolare l'output in maniera più chiara non verrà usata questa opzione, infatti è possibile restituire la valutazione di ogni regola all'interno di un singolo `Result`. `Result` è una struttura che contiene due componenti: `Expressions` e `Bindings`. `Expressions` è la componente interessante, in quanto contiene il valore della valutazione di una espressione nella query, dove per "espressione" si intende "`data.example_rego`".

`Expressions` è un array di puntatori a `ExpressionValue`, che a sua volta è una struttura che finalmente contiene la parte rilevante nella componente `Value`. Quest'ultima è del tipo `interface{}`, dal valore reale `map[string]any`. Quindi `Value` conterrà le coppie (REGOLA,OUTPUT).

```

1 // manipulate the result to divide the state from the real output
2 resultRaw := rs[0].Expressions[0].Value
3 result, ok := resultRaw.(map[string]any)
4 stateRaw, ok := result["state"]
5 if ok {
6     delete(result, "state")
7     state := stateRaw.(map[string]any)
8     return state, result
9 }
10 return nil, result

```

Per estrarre lo stato è sufficiente accedere, tramite la variabile `result`, al valore dell'output della regola STATE. Si elimina dal risultato finale lo stato, in quanto il client non necessita di questa informazione, si effettuano i dovuti *casting* e si restituisce lo stato e il risultato finale. Assemblando i vari pezzi, otteniamo il codice del listing 5.8.

Avendo a disposizione il risultato, contenente la valutazione della query richiesta dal client e il nuovo stato, si può procedere con la restituzione del risultato al client e con l'aggiornamento dei dati presenti in `datastore`. Questa è l'ultima operazione principale effettuata dal modulo `opawrap`. Aggiornando i dati con il nuovo stato recuperato, si garantisce che le future valutazioni di query da parte di OPA

```

1 func Opa(data map[string]any, input any, w http.ResponseWriter,
2         ctx context.Context) (map[string]any, map[string]any) {
3     packageRego := "data.examplerego"
4     store := inmem.NewFromObject(data)
5     tx := storage.NewTransactionOrDie(ctx, store)
6     re := rego.New(
7         rego.Query(packageRego),
8         rego.Load([]string{os.Args[1]}, nil),
9         rego.Store(store),
10        rego.Transaction(tx),
11    )
12    query, err := re.PrepareForEval(ctx)
13    rs, err := query.Eval(ctx, rego.EvalInput(input))
14    resultRaw := rs[0].Expressions[0].Value
15    result, ok := resultRaw.(map[string]any)
16    stateRaw, ok := result["state"]
17    if ok {
18        delete(result, "state")
19        state := stateRaw.(map[string]any)
20        return state, result
21    }
22    return nil, result
23 }
```

Listing 5.8: Funzione che delega la valutazione della query a OPA.

avverranno rispetto ai nuovi dati presenti. Si noti che ci potrebbero essere casi in cui si decide di non restituire alcuno stato, ossia la valutazione della regola STATE della policy non conterrà alcun output.

Passaggio dello stato aggiornato a datastore

La funzione `updateState` si occupa di inviare lo stato corrente a `datastore`, cosicché possa essere utilizzato nelle future valutazioni di query da parte di OPA.

```

1 func updateState(baseURL *url.URL, state map[string]any) {
2     for key, value := range state {
3         baseURL.Path = path.Join("/data", key)
4         bodyBytes, err := json.Marshal(value)
5         body := io.NopCloser(bytes.NewBuffer(bodyBytes))
6         client := http.Client{}
7         req, err := http.NewRequest(http.MethodPut, baseURL.String(), body)
8         _, err = client.Do(req)
9     }
10 }
```

Listing 5.9: Aggiornamento dello stato in datastore.

Si effettua quindi una richiesta a `localhost:8081/data/{key}`, per ognuna `key` presente nel nuovo stato, effettuando dapprima il *marshal* in JSON del singolo dato da inviare e poi inviandolo tramite il `client`.

```

1 package exemplerego
2
3 import rego.v1
4
5 default allow := false
6
7 allow if {
8     input.user == "fabio"
9     data.counter > 0
10 }
11
12 state["counter"] := data.counter - 1 if allow

```

Figura 5.6: Policy con lo stato per lo scenario **Contatore**.

5.5 Esempi di utilizzo di OWSM

In questa sezione verrà presentato il funzionamento di **OWSM**, nel caso dei canonici esempi **Contatore** e **Tre Microservizi**.

Per presentare il funzionamento ci si avvale del tool *Yaak* [35], utile per effettuare richieste *REST*, evitando di utilizzare il verboso strumento a linea di comando *cURL* [33]. Le immagini che presentano il funzionamento di **OWSM** sono presenti nell'appendice A.

Innanzitutto, vediamo come riscrivere correttamente le policy, soddisfacendo il principio presentato nella sezione 5.2. Bisogna introdurre il concetto di stato, tramite la dichiarazione di una regola STATE. Nelle figure fig. 5.6 e fig. 5.7 vengono presentate le nuove policy.

5.5.1 Policy scenario Contatore

Una volta eseguiti i server ***datastore*** e ***opawrap***, quest'ultimo con la policy corretta per lo scenario del **Contatore** (fig. 5.6), possiamo effettuare una chiamata GET verso **localhost:8081/data** per verificare il contenuto dei dati. Siamo nella situazione in fig. A.1.

Successivamente si richiede a **localhost:8080/query** di valutare la query con il seguente input `{ "user": "mario" }`. L'output atteso è `"allow": false`. Ci si aspetta che il contatore non venga decrementato una volta effettuata la query, in quanto il permesso non è stato concesso. Infatti in fig. A.2, dopo aver effettuato un'ulteriore richiesta GET a **localhost:8081/data**, notiamo che il valore del contatore è invariato e che l'output è quanto ci si aspettava.

Ora inviamo come input `{ "user": "fabio" }`. In questo caso il permesso è concesso, con il conseguente decremento del contatore, come si può vedere in fig. A.3.

Se si esegue la query con lo stesso input per cinque volte il permesso verrà concesso, al sesto tentativo il permesso verrà negato in quanto il contatore è stato decrementato, contenendo il valore "0". La situazione è presentata in figura fig. A.4.

5.5.2 Policy scenario Tre Microservizi

Una volta eseguiti i server ***datastore*** e ***opawrap***, quest'ultimo con la policy corretta per lo scenario del **Tre Microservizi** (fig. 5.7), possiamo effettuare una chiamata GET verso **localhost:8081/data** per verificare il contenuto dei dati. Siamo nella situazione in fig. A.5.

```

1 package exemplerego
2
3 import rego.v1
4
5 default allow := false
6
7 allow if { input.source == "a"; input.dest == "b" }
8
9 allow if { input.source == "b"; input.dest == "c"; data.a_to_b == false }
10
11 state["a_to_b"] if {
12     input.source == "a"
13     input.dest == "b"
14 }
```

Figura 5.7: Policy con lo stato per lo scenario **Tre Microservizi**.

Successivamente si richiede a `localhost:8080/query` di valutare la query con il seguente input `{ "source": "b", "dest": "c" }`. In questo caso ci si aspetta che B e C possano comunicare per un indeterminato numero di volte. Si veda fig. A.6.

Eventualmente si ricade nella situazione in cui si chiede la valutazione di una query con il seguente input `{ "source": "a", "dest": "b" }`. In questo caso la comunicazione viene permessa, ma verrà modificato il valore del dato `"a_to_b"`, assumendo valore `true`. Si veda fig. A.7.

Da questo momento in poi la comunicazione tra B e C verrà negata, come si può evincere dalla fig. A.8.

6

Valutazione OWSM

In questo capitolo verrà presentata una valutazione quantitativa dell'impatto del wrapper **OWSM** nella valutazione delle query. La valutazione verrà effettuata su policy *stateless basate su dei dati da interrogare*, eseguendole sia con **OWSM** sia con **OPA-standalone**.

Per valutare le performance di OPA è stato creato un semplice server in Go, che espone il singolo API endpoint `localhost:8181/opa`, e che utilizza OPA come libreria SDK per valutare le query. In questo modo la valutazione delle query con OPA avviene utilizzando la libreria Go messa a disposizione dagli sviluppatori, sia per **OWSM** che per **OPA-standalone**. Per ridurre il *bias* della valutazione gli esempi di policy sono stati reperiti dalla piattaforma *The Rego Playground* [38]. Nella sezione *Examples/Access Control* sono presenti tre diversi scenari di controllo degli accessi: *RBAC*, *ABAC* e *RBAC con ruoli gerarchici*. Si consulti la pagina ufficiale [38] per maggiori informazioni riguardo i tre esempi. La piattaforma, oltre a dare una possibile implementazione delle policy scritte in Rego per questi scenari, fornisce i dati rispetto ai quali valutare la policy e l'input della query.

Verranno effettuate delle misurazioni per ognuno di questi scenari, valutando le policy sia con **OWSM** che con **OPA-standalone**. Verrà utilizzato lo strumento a linea di comando *ApacheBench* [18], il quale fornisce diverse misurazioni per valutare le prestazioni di un server web. Il comando di *ApacheBench* che verrà usato per effettuare le misurazioni sarà del tipo:

```
1 ab -n richieste -c livConcorrenza -p input.json -T application/json URL
```

Questo indica che verranno eseguite `-c livConcorrenza` richieste per volta, fino ad arrivare a `-n richieste` totali.

Nelle prossime sezioni verranno presentati i 3 esperimenti fatti al fine di effettuare il confronto tra il wrapper e OPA. Tutti gli esperimenti mostreranno il peggioramento in percentuale di **OWSM** rispetto a **OPA-standalone**, al crescere dei valori sull'ascissa. Inoltre, è stata eseguita una regressione lineare sui dati, in modo da visualizzare una stima dell'andamento del peggioramento. Tutti i grafici sono stati creati usando *Python*.

6.1 Esperimento 1

L'obiettivo di questo primo esperimento è quello di confrontare il tempo medio per eseguire ogni singola query usando **OPA-standalone** e usando **OWSM**. Le misurazioni verranno effettuate con *Apache-*

Bench, il quale restituirà il tempo medio per eseguire una singola richiesta. Per chiarire ulteriormente le idee, eseguendo `ab -n 100`, verrà restituito il tempo medio di una singola query, calcolato come

$$\text{tempoMedio} = \frac{\text{tempoTOT}}{100}$$

Visto l’obiettivo dell’esperimento, il livello di concorrenza dovrà essere fissato a 1, in questo modo verrà eseguita sempre una sola richiesta per volta. Quindi, verrà eseguito il comando `ab -n r -c 1`, dove $r \in [100, 200, \dots, 25600, 51200]$. Ossia $r = 2^i * 100$, dove $i \in [0, \dots, 9]$.

Come si può vedere dai grafici nelle fig. da 6.1 a 6.3 il tempo medio per valutare una policy usando **OWSM** è peggiorato rispetto a **OPA-standalone**. Rispetto ai dati ottenuti, il peggioramento è di circa 20-25% in media. Si nota però, nei grafici a destra (nelle fig. da 6.1 a 6.3), come la stima lineare del peggioramento decresce all’aumentare delle query totali effettuate.

6.2 Esperimento 2

L’obiettivo di questo secondo esperimento è quello di confrontare il tempo medio per eseguire un insieme di query concorrenti usando **OPA-standalone** e usando **OWSM**. Il livello di concorrenza quindi varierà ma il numero di richieste totali verrà fissato a 2000. Le misurazioni verranno effettuate con *ApacheBench*, il quale restituirà il tempo medio per eseguire un certo numero di richieste concorrenti. Per chiarire ulteriormente le idee, eseguendo `ab -n 2000 -c 10`, verrà restituito il tempo medio per eseguire 10 query simultanee, calcolato come

$$\text{tempoMedio} = \frac{\text{tempoTOT}}{2000} * 10$$

Ossia, è il tempo medio per ogni singola query moltiplicato per il livello di concorrenza. Quindi, verrà eseguito il comando `ab -n 2000 -c liv`, dove $liv \in [10, 20, \dots, 100]$.

Come si può vedere dai grafici in fig. da 6.4 a 6.6, notiamo che **OWSM** introduce un sovraccarico che aumenta sensibilmente all’aumentare del livello di concorrenza. Infatti, i grafici a destra (nelle fig. da 6.4 a 6.6) mostrano come la stima lineare del peggioramento aumenti notevolmente all’aumentare dei valori in ascissa.

6.3 Esperimento 3

In questo terzo esperimento sono stati rifatti i passi eseguiti nell’esperimento precedente, togliendo il meccanismo di sincronizzazione da **OWSM**. Ossia, sono state tolte le righe di codice 2 e 7 dallo pseudo-codice fornito nel listing 5.6, cosicché il modulo `opawrap` non effettui più il `lock()` e l’`unlock()` dei dati presenti nel **datastore**.

Dai grafici nelle fig. da 6.7 a 6.9 notiamo che togliendo il *mutex* a **OWSM** questo si comporta similmente a **OPA-standalone** al crescere del livello di overhead. È presente comunque un peggioramento di circa 20% in media, secondo i dati ottenuti. Dai grafici a destra (nelle fig. da 6.7 a 6.9), si vede come la stima lineare del peggioramento non abbia una pendenza così elevata come quella osservata nell’esperimento precedente.

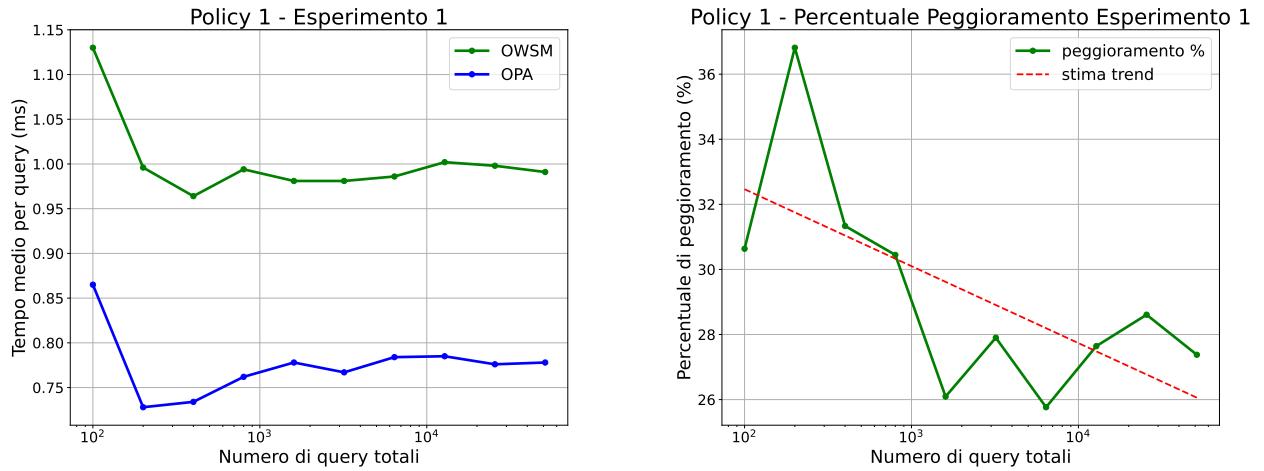


Figura 6.1: Esperimento 1 Policy 1 - a sinistra andamento OWSM e OPA, a destra il peggioramento in percentuale di OWSM.

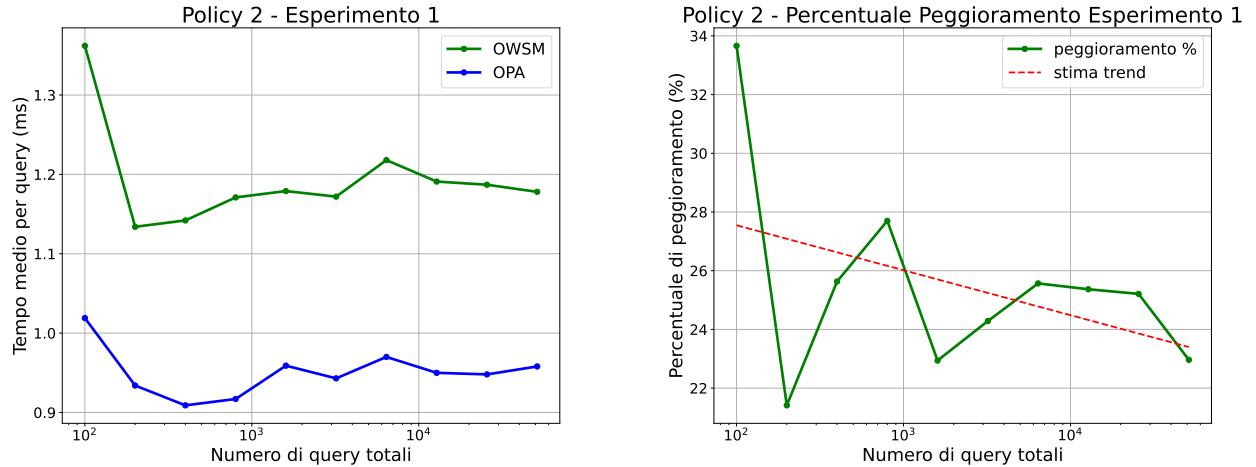


Figura 6.2: Esperimento 1 Policy 2 - a sinistra andamento OWSM e OPA, a destra il peggioramento in percentuale di OWSM.

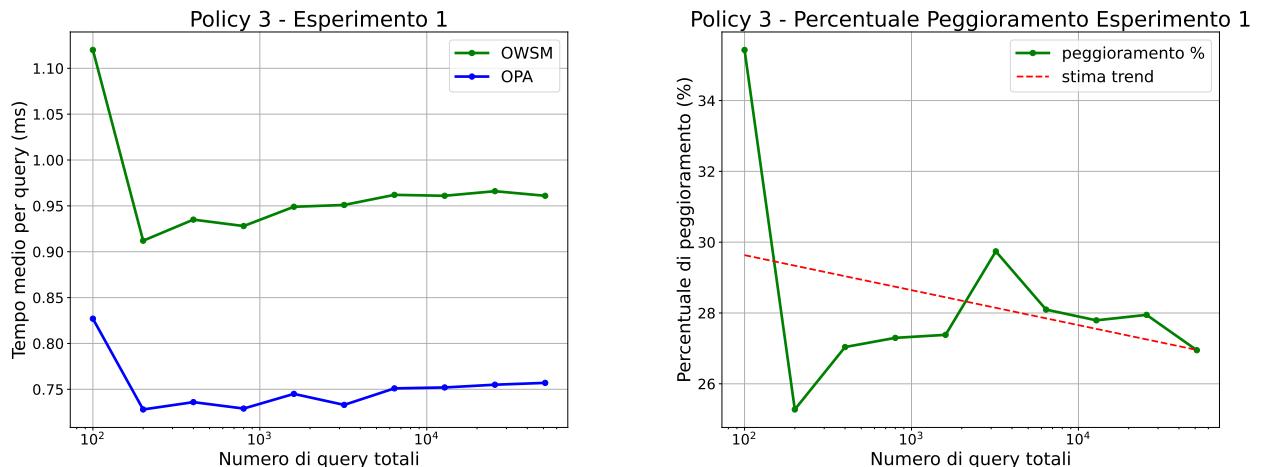


Figura 6.3: Esperimento 1 Policy 3 - a sinistra andamento OWSM e OPA, a destra il peggioramento in percentuale di OWSM.

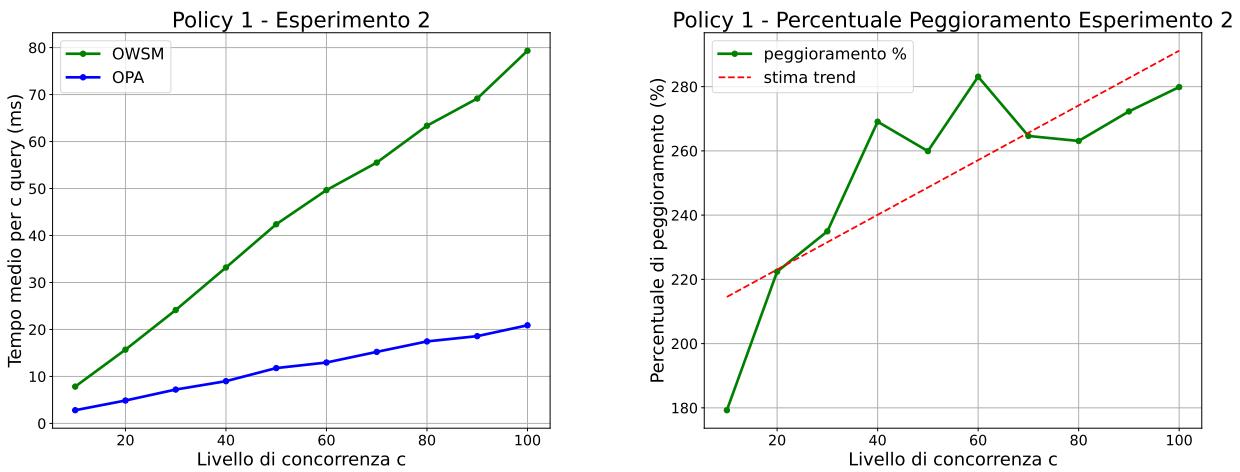


Figura 6.4: Esperimento 2 Policy 1 - a sinistra andamento OWSM e OPA, a destra il peggioramento in percentuale di OWSM.

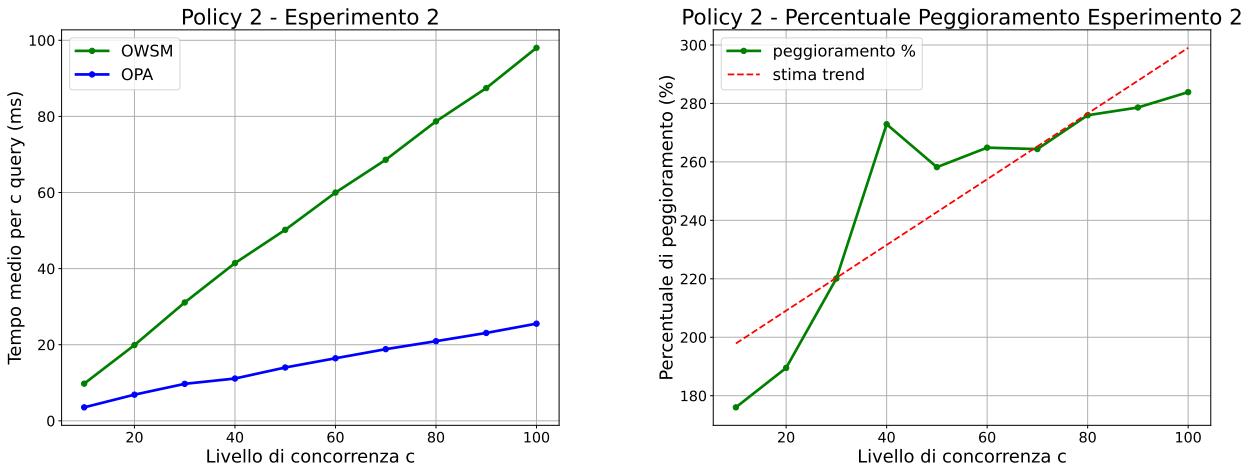


Figura 6.5: Esperimento 2 Policy 2 - a sinistra andamento OWSM e OPA, a destra il peggioramento in percentuale di OWSM.

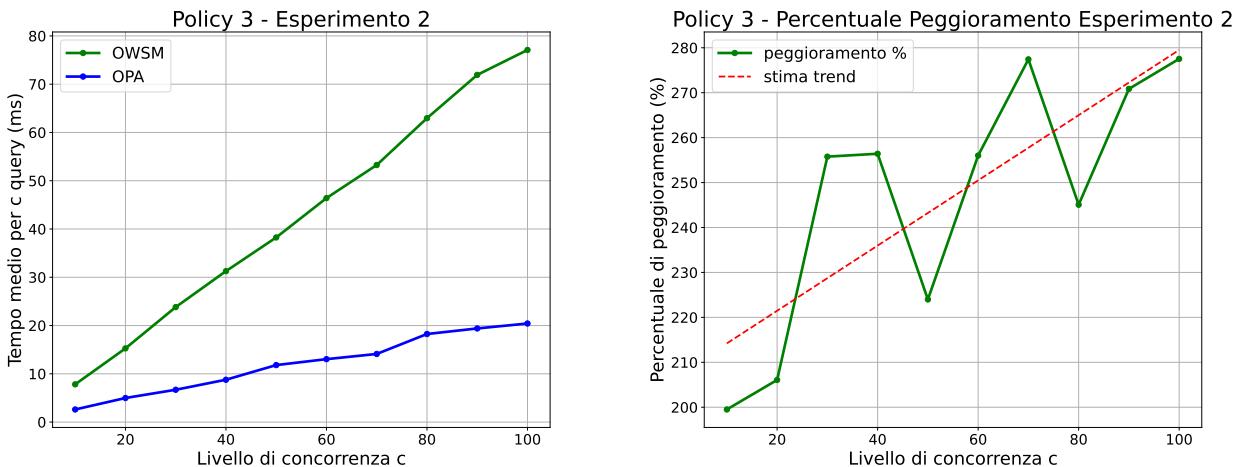


Figura 6.6: Esperimento 2 Policy 3 - a sinistra andamento OWSM e OPA, a destra il peggioramento in percentuale di OWSM.

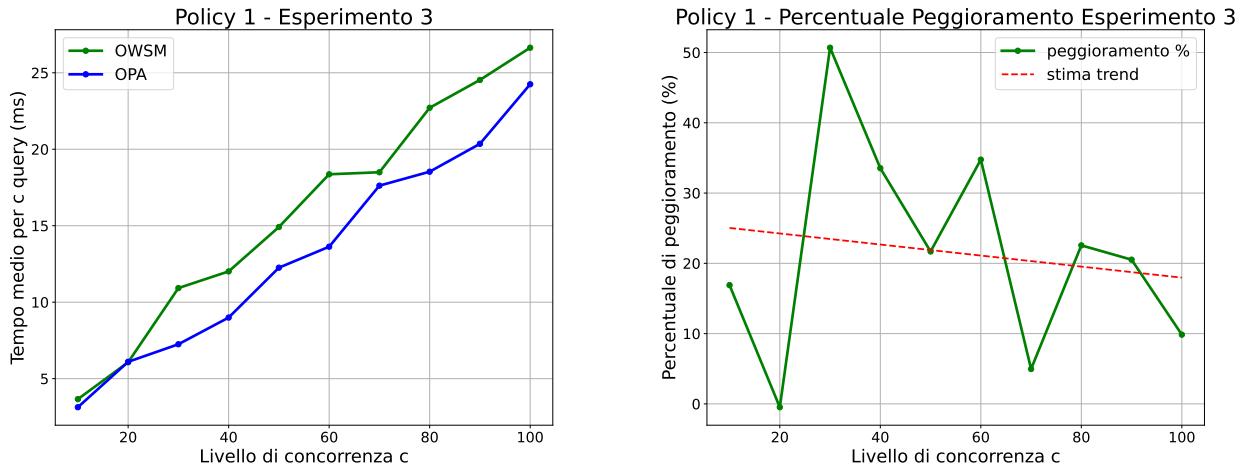


Figura 6.7: Esperimento 3 Policy 1 - a sinistra andamento OWSM e OPA, a destra il peggioramento in percentuale di OWSM.

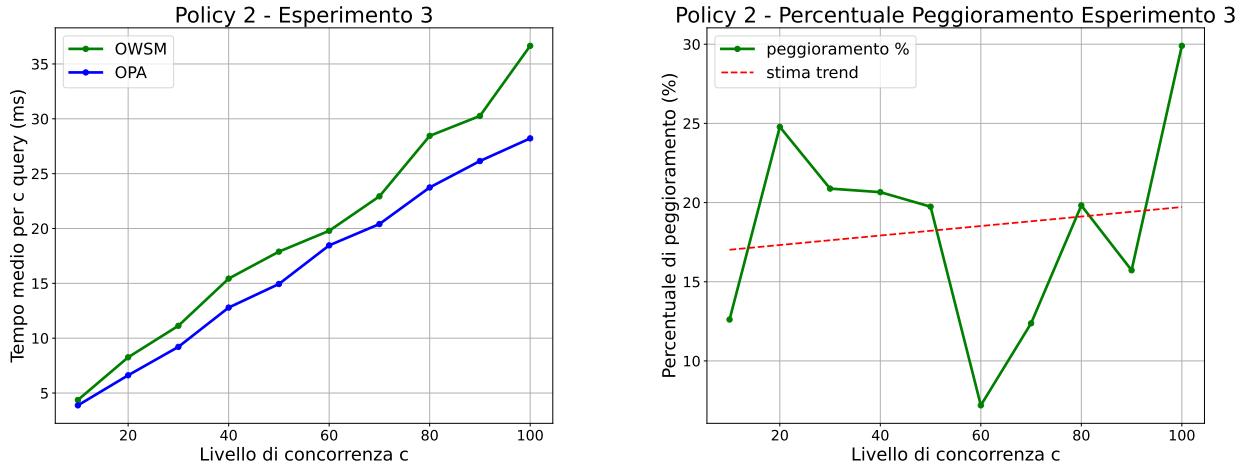


Figura 6.8: Esperimento 3 Policy 2 - a sinistra andamento OWSM e OPA, a destra il peggioramento in percentuale di OWSM.

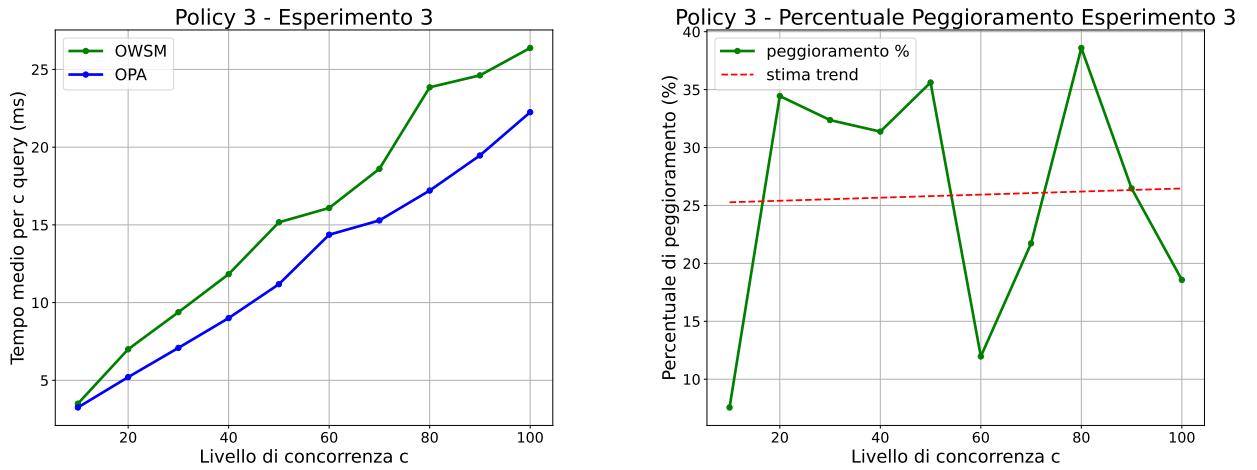


Figura 6.9: Esperimento 3 Policy 3 - a sinistra andamento OWSM e OPA, a destra il peggioramento in percentuale di OWSM.

6.4 Considerazioni finali

Tramite questi esperimenti abbiamo ottenuto una valutazione quantitativa di **OWSM** rispetto a **OPA-standalone**, rispetto a policy *stateless basate su dei dati da interrogare*.

Dai dati ottenuti nell'esperimento 1 (fig. da 6.1 a 6.3) e nell'esperimento 3 (fig. da 6.7 a 6.9), è possibile osservare come il wrapper presentato introduca un peggioramento delle performance complessive di circa 20% rispetto a OPA-standalone. Considerando sia i grafici dell'esperimento 2 (fig. da 6.4 a 6.6) sia quelli dell'esperimento 3 (fig. da 6.7 a 6.9), notiamo come sia presente un costo variabile dovuto all'attesa del *lock*, direttamente proporzionale al livello di concorrenza, in aggiunta ad un costo fisso attorno ai 5ms. L'esperimento 3 permette di ipotizzare che il sovraccarico osservato nell'esperimento 2 sia dovuto ai *lock* e *unlock* verso **datastore**, effettuati ad ogni singola richiesta, compromettendo quindi le performance di **OWSM** al crescere del numero di richieste parallele effettuate.

In definitiva, rispetto agli esperimenti fatti e ai dati ottenuti da essi, si può affermare che **OWSM** introduca un peggioramento delle performance di circa 20% rispetto a **OPA-standalone**, e il sovraccarico introdotto dal meccanismo di sincronizzazione *mutex*, inevitabile per mantenere i dati consistenti, aumenta sensibilmente al crescere del grado di concorrenza.

7

Conclusioni

In questo lavoro di tesi è stato presentato un wrapper, col fine di aggiungere la nozione di stato e di riuscire a modificare i dati usati per far rispettare le policy scritte in Rego. Quindi, si è aumentata la quantità di policy esprimibili e applicabili mediante Rego e OPA, aggiungendo la possibilità di esprimere policy *stateful basate su dei dati da interrogare*. Specificamente, il wrapper è ora in grado di intercettare le richieste ricevute da un client, inoltrarle a OPA per la valutazione, ricevere il risultato di questa e restituirla al client. Prima di restituire il risultato della valutazione al client, è stato estrapolato lo stato, contenente i dati da modificare, e salvato di conseguenza nei *data* usati da OPA. In questo modo le future richieste potranno essere valutate rispetto al nuovo stato.

È stato necessario analizzare diverse soluzioni per quanto riguarda la scrittura di policy di sicurezza, per riuscire a individuare in OPA e Rego l'alternativa ideale. Inizialmente, si è considerato il minimo indispensabile per quanto riguarda un'architettura a microservizi, distribuiti sui container tramite Docker e Kubernetes. Si è visto come la risorsa NetworkPolicies offerta da Kubernetes riesca ad esprimere solamente politiche di sicurezza a livello 3-4 dello stack ISO/OSI, comportandosi sostanzialmente come un firewall. Successivamente, introducendo un ulteriore livello all'infrastruttura, si è osservato che tramite l'impiego di una service mesh, in particolare Linkerd o Istio, è possibile scrivere politiche di sicurezza a livello 7, quindi esprimendo un insieme più ampio di policy rispetto al solo impiego di Kubernetes. Questo modo di scrivere policy si basa su una mera configurazione statica, ossia con policy espresse tramite file YAML, riuscendo ad esprimere solamente policy della categoria *statiche*. Queste presentano delle limitazioni che hanno portato ad analizzare ulteriormente lo stato dell'arte e a trovare la filosofia di Policy-as-Code, tramite la quale è possibile esprimere policy *stateless basate su dei dati da interrogare*.

Le attuali soluzioni di Policy-as-Code non permettono di far rispettare politiche di sicurezza che sfruttano la nozione di stato, quindi non è possibile modificare i dati rispetto ai quali si valuta la policy. È in questo caso che la soluzione presentata nella tesi diventa utile. Il wrapper è stato progettato attorno a OPA e Rego in quanto le decisioni avvenute, previa valutazione rispetto ad una policy, sono restituite in JSON. Tramite il wrapper siamo quindi ora in grado di salvare uno stato, tra le varie richieste ricevute da un motore di policy, cosicché la valutazione di una generica richiesta dipenda da quanto successo nel passato. Si è visto come, grazie all'impiego del wrapper, sia ora possibile esprimere le policy dei canonici scenari, **Contatore** e **Tre Microservizi**, che sono state scritte tramite l'inalterato Rego. Ora tramite Rego è possibile esprimere persino policy *stateful basate su dei dati da interrogare*.

La valutazione quantitativa effettuata su policy *stateless basate su dei dati da interrogare* ha mostrato come il wrapper introduca un peggioramento generale di circa il 20% rispetto a OPA a sé stante. Inoltre, al crescere del numero di query in parallelo è presente un sovraccarico che aumenta sensibilmente. Il sovraccarico è introdotto principalmente dal meccanismo di sincronizzazione *mutex*, inevitabile per mantenere lo stato consistente in contesti in cui è presente concorrenza. Infatti, si è stimato che senza il *mutex* l'andamento del sovraccarico sia una retta con una pendenza molto più bassa.

7.1 Possibili implementazioni e sviluppi futuri

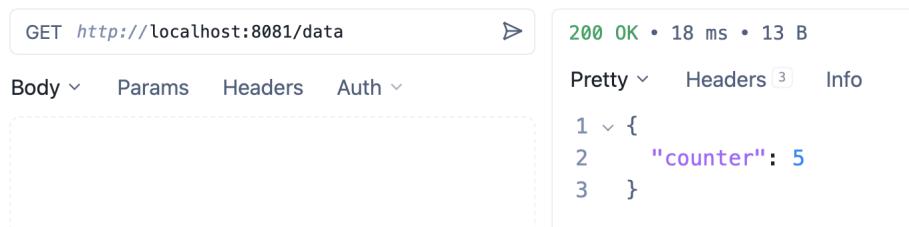
Tra le possibili implementazioni e sviluppi futuri si potrebbe pensare di estendere il wrapper a diversi linguaggi decisionali, che rispettino il prerequisito di restituire il risultato della decisione in formato JSON, per esempio OpenFGA. In questo modo si astrae dal possibile linguaggio decisionale e/o motore di policy utilizzato. Vista la mancanza di una semantica del linguaggio Rego, sarebbe opportuno e decisamente interessante definirla al fine di capire esattamente l'espressività di questo linguaggio. In questo modo, si può fare un confronto oggettivo tra l'espressività offerta da ciascun linguaggio decisionale. Inoltre, gli attuali motori di policy sono centralizzati, ossia è presente un singolo servizio che prende decisioni. Decentralizzare la decisione delle richieste, sarebbe molto efficiente e migliorerebbe la scalabilità. Si potrebbe pensare di decentralizzare e astrarre dallo specifico motore di policy e linguaggio decisionale impiegato, permettendo quindi una maggiore eterogeneità, tipica delle architetture a microservizi. Infine, Cedar potrebbe essere una soluzione interessante, in quanto è stata verificata formalmente la sua semantica. Si potrebbero indagare ulteriormente i linguaggi decisionali che restituiscono un risultato booleano per capire se è possibile progettare un'idea simile al wrapper per introdurre il concetto di stato.

A

Esempio utilizzo OWSM

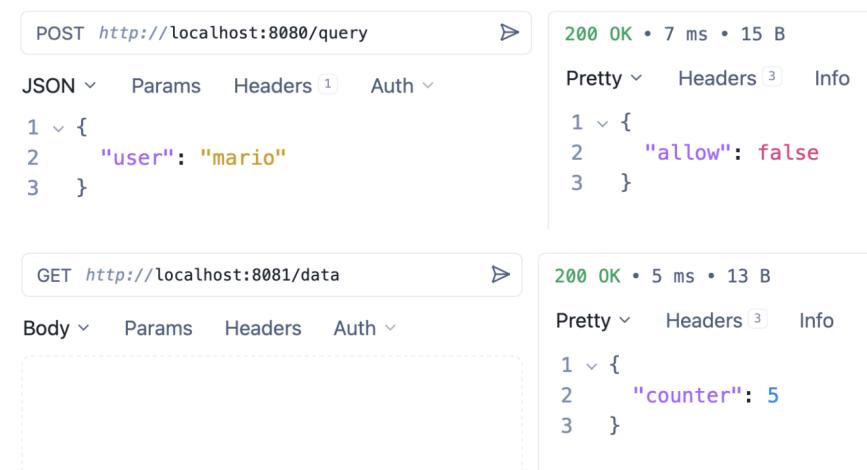
In questo appendice sono riportate le immagini che mostrano il funzionamento del wrapper **OWSM** nei due scenari **Contatore** e **Tre Microservizi**.

A.1 Scenario Contatore



```
GET http://localhost:8081/data
200 OK • 18 ms • 13 B
Pretty ▾ Headers [3] Info
1 ▾ {
2   "counter": 5
3 }
```

Figura A.1: Contenuto datastore al momento dell'avvio nello scenario **Contatore**.



```
POST http://localhost:8080/query
200 OK • 7 ms • 15 B
Pretty ▾ Headers [3] Info
1 ▾ {
2   "allow": false
3 }

GET http://localhost:8081/data
200 OK • 5 ms • 13 B
Pretty ▾ Headers [3] Info
1 ▾ {
2   "counter": 5
3 }
```

Figura A.2: Output query e contenuto datastore con input che viola la policy per lo scenario **Contatore**.

```

POST http://localhost:8080/query
JSON Params Headers [1] Auth
1 < {
2   "user": "fabio"
3 }

200 OK • 15 ms • 14 B
Pretty Headers [3] Info
1 < {
2   "allow": true
3 }

GET http://localhost:8081/data
Body Params Headers Auth
Body

200 OK • 2 ms • 13 B
Pretty Headers [3] Info
1 < {
2   "counter": 4
3 }

```

Figura A.3: Output query e contenuto datastore con un input accettato dalla policy per lo scenario **Contatore**.

```

POST http://localhost:8080/query
JSON Params Headers [1] Auth
1 < {
2   "user": "fabio"
3 }

200 OK • 8 ms • 15 B
Pretty Headers [3] Info
1 < {
2   "allow": false
3 }

GET http://localhost:8081/data
Body Params Headers Auth
Body

200 OK • 4 ms • 13 B
Pretty Headers [3] Info
1 < {
2   "counter": 0
3 }

```

Figura A.4: Output query e contenuto datastore superato il numero di accessi consentiti dalla policy per lo scenario **Contatore**.

A.2 Scenario Tre Microservizi

```
GET http://localhost:8081/data
200 OK • 4 ms • 16 B
Pretty ▾ Headers [3] Info
1 ▾ {
2   "a_to_b": false
3 }
```

Figura A.5: Contenuto datastore al momento dell'avvio nello scenario **Tre Microservizi**.

```
POST http://localhost:8080/query
200 OK • 6 ms • 14 B
Pretty ▾ Headers [3] Info
1 ▾ {
2   "allow": true
3 }
```



```
GET http://localhost:8081/data
200 OK • 2 ms • 16 B
Pretty ▾ Headers [3] Info
1 ▾ {
2   "a_to_b": false
3 }
```

Figura A.6: Output query e contenuto datastore con input B e C per la policy dello scenario **Tre Microservizi**.

```
POST http://localhost:8080/query
200 OK • 54 ms • 14 B
Pretty ▾ Headers [3] Info
1 ▾ {
2   "allow": true
3 }
```



```
GET http://localhost:8081/data
200 OK • 2 ms • 15 B
Pretty ▾ Headers [3] Info
1 ▾ {
2   "a_to_b": true
3 }
```

Figura A.7: Output query e contenuto datastore con input A e B per la policy dello scenario **Tre Microservizi**.

The screenshot shows a REST API tool interface with two requests:

- POST <http://localhost:8080/query>**:
 - Body (JSON):

```
1 ~ {  
2   "source": "b",  
3   "dest": "c"  
4 }
```
- 200 OK • 13 ms • 15 B**
Pretty:

```
1 ~ {  
2   "allow": false  
3 }
```
- GET <http://localhost:8081/data>**:
 - Body (JSON): [Empty dashed box]
- 200 OK • 3 ms • 15 B**
Pretty:

```
1 ~ {  
2   "a_to_b": true  
3 }
```

Figura A.8: Output query e contenuto datastore con input B e C dopo la comunicazione tra A e B per la policy dello scenario **Tre Microservizi**.

Bibliografia

- [1] Open Policy Agent. Introduction. <https://www.openpolicyagent.org/docs/latest/>, 2024. Accessed: 2024-11-13.
- [2] Open Policy Agent. Try opa as a go library. <https://www.openpolicyagent.org/docs/latest/#5-try-opa-as-a-go-library>, 2024. Accessed: 2024-11-13.
- [3] Andrea Altarui, Marino Miculan, Matteo Paier, e altri. Dbcchecker: A bigraph-based tool for checking security properties of container compositions. In *CEUR WORKSHOP PROCEEDINGS*, volume 3488. CEUR-WS, 2023.
- [4] Envoy Project Authors. Envoy. <https://www.envoyproxy.io>, 2024. Accessed: 2024-11-13.
- [5] Istio Authors. Istio documentation. <https://istio.io/latest/docs/>, 2024. Accessed: 2024-11-13.
- [6] Red Hat Authors. The state of kubernetes security report: 2024 edition. <https://www.redhat.com/en/engage/state-kubernetes-security-report-2024>, 2024. Accessed: 2024-11-13.
- [7] The Kubernetes Authors. Borg: The predecessor to kubernetes. <https://kubernetes.io/blog/2015/04/borg-predecessor-to-kubernetes/>, 2015. Accessed: 2024-11-13.
- [8] The Kubernetes Authors. Kubernetes documentation. <https://kubernetes.io/docs/home/>, 2024. Accessed: 2024-11-13.
- [9] Hrishikesh Barua. Half of 4 million public docker hub images found to have critical vulnerabilities. <https://www.infoq.com/news/2020/12/dockerhub-image-vulnerabilities/>, 2020. Accessed: 2024-11-13.
- [10] Daniel Bass. Opa, cedar, openfga: Why are policy languages trending right now? <https://www.permit.io/blog/opa-cedar-openfga-why-are-policy-languages-trending>, 2024. Accessed: 2024-11-13.
- [11] Daniel Bass. Policy engines: Open policy agent vs aws cedar vs google zanzibar. <https://www.permit.io/blog/policy-engines>, 2024. Accessed: 2024-11-13.
- [12] Grzegorz Blinowski, Anna Ojdowska, e Adam Przybyłek. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, 10:20357–20374, 2022.
- [13] Linkerd Authors Buoyant. Authorization policy. <https://linkerd.io/2.16/features/server-policy/#policy-overview>, 2024. Accessed: 2024-11-13.

- [14] Linkerd Authors Buoyant. High availability. <https://linkerd.io/2.16/features/ha/>, 2024. Accessed: 2024-11-13.
- [15] Linkerd Authors Buoyant. Linkerd documentation. <https://linkerd.io/2.16/overview/>, 2024. Accessed: 2024-11-13.
- [16] Fabio Burco, Marino Miculan, e Marco Peressotti. Towards a formal model for composable container systems. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pp. 173–175, 2020.
- [17] Mattia Caracciolo. *Policy as Code, how to automate cloud compliance verification with open-source tools*. Tesi di Dottorato di Ricerca, Politecnico di Torino, 2023.
- [18] The Apache Software Foundation. ab - apache http server benchmarking tool. <https://httpd.apache.org/docs/2.4/programs/ab.html>, 2024. Accessed: 2024-11-18.
- [19] The Linux Foundation. Open container initiative. <https://opencontainers.org>, 2024. Accessed: 2024-10-06.
- [20] The Linux Fundation. Cncf 2023 annual survey. <https://www.cncf.io/reports/cncf-annual-survey-2023/>, 2023. Accessed: 2024-11-13.
- [21] The Linux Fundation. containerd. <https://containerd.io>, 2024. Accessed: 2024-10-06.
- [22] The Linux Fundation. Openfga. <https://openfga.dev>, 2024. Accessed: 2024-11-13.
- [23] Google e The Go Authors. The go programming language. <https://go.dev>, 2024. Accessed: 2024-11-13.
- [24] Red Hat. What's a service mesh? <https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh>, 2018. Accessed: 2024-11-13.
- [25] Amazon Web Services Inc. Cedar language. <https://www.cedarpolicy.com/en>, 2024. Accessed: 2024-11-13.
- [26] Docker Inc. The industry-leading container runtime. <https://www.docker.com/products/container-runtime/>, 2024. Accessed: 2024-10-06.
- [27] Docker Inc. Use containers to build, share and run your applications. <https://www.docker.com/resources/what-container/>, 2024. Accessed: 2024-10-06.
- [28] Xing Li, Yan Chen, Zhiqiang Lin, Xiao Wang, e Jim Hao Chen. Automatic policy generation for inter-service access control of microservices. In *30th USENIX Security Symposium (USENIX Security 21)*, pp. 3971–3988, 2021.
- [29] David Maier, K. Tuncay Tekle, Michael Kifer, e David S. Warren. *Datalog: concepts, history, and outlook*, p. 3–100. Association for Computing Machinery and Morgan & Claypool, 2018.

- [30] Gabriel L. Manor. Love, hate, and policy languages: an introduction to decision-making engines. <https://www.cncf.io/blog/2024/05/21/love-hate-and-policy-languages-an-introduction-to-decision-making-engines/>, 2024. Accessed: 2024-11-13.
- [31] William Morgan. The enterprise architect's guide to the service mesh. <https://buoyant.io/download/the-enterprise-architects-guide-to-the-service-mesh>, 2023. Accessed: 2024-11-13.
- [32] William Morgan. What is a service mesh? <https://buoyant.io/what-is-a-service-mesh>, 2024. Accessed: 2024-11-13.
- [33] Contributors of cURL. curl - open source command line tool and library. <https://curl.se>, 2024. Accessed: 2024-11-13.
- [34] Marcus Rönnbäck e Fredrik Åberg. Automatic enforcement of container security guidelines through policy as code. *Chalmers ODR*, 2022.
- [35] Gregory Schier. Yaak - the api client for modern developers. <https://yaak.app>, 2024. Accessed: 2024-11-13.
- [36] Amazon Web Services. Qual è la differenza tra architettura monolitica e architettura di microservizi? <https://go.aws/47CijTn>, 2024. Accessed: 2024-11-13.
- [37] Kirill Shirinkin. What is a service mesh? <https://mkdev.me/posts/what-is-a-service-mesh>, 2024. Accessed: 2024-11-13.
- [38] Styra. The rego playground. <https://play.openpolicyagent.org/>, 2024. Accessed: 2024-11-13.
- [39] Junsheng Tan e altri. Ensuring component dependencies and facilitating documentation by applying open policy agent in a devsecops cloud environment. Tesi per Master, Aalto University, 2022.
- [40] Tigera. Calico docs. <https://docs.tigera.io/calico/latest/about/>, 2024. Accessed: 2024-11-13.
- [41] Luca Verderame, Luca Caviglione, Roberto Carbone, e Alessio Merlo. Secco: Automated services to secure containers in the devops paradigm. In *Proceedings of the 2023 International Conference on Research in Adaptive and Convergent Systems*, pp. 1–6, 2023.
- [42] Junzo Watada, Arunava Roy, Ruturaj Kadikar, Hoang Pham, e Bing Xu. Emerging trends, techniques and open issues of containerization: A review. *IEEE Access*, 7:152443–152472, 2019.