POLITECNICO DI TORINO

Corso di Laurea in Computer Engineering

Tesi di Laurea

Custom Control of Network Services in Kubernetes

Relatori
prof. Fulvio Giovanni Ottavio Risso
prof. Guido Marchetto

Candidato
Francesco Valente

Supervisore aziendale
TIM Innovation
Dott. Marco Signorelli

Anno accademico 2019-2020

Dedicato alla mia famiglia

Sommario

Questo lavoro di tesi si concentra sull'impatto della cloudification sugli operatori di telecomunicazione. L'evoluzione degli ultimi anni vede le compagnie di telecomunicazioni cambiare il loro approccio riprogettando l'architettura centrale e adottando orchestratori di container come Kubernetes. L'obiettivo di questa tesi è definire e sviluppare un operatore Kubernetes, model driven e non specifico per una determinata applicazione, in grado di acquisire un numero di input generico (fonte dati che può influenzare il comportamento di funzioni di rete virtuali), trattarli con una logica di business programmabile dall'esterno e non nota a priori, e generare un certo numero di output arbitrari (come sollecitazione agli input ricevuti). Questo paradigma applicativo ben si colloca nella gestione del ciclo di vita dei servizi basati su Virtual Network Function (VNF) in una rete di telecomunicazioni ed in particolare copre le fasi di maintenance di un servizio tramite il concetto di closed loop. Inizialmente si è cercato di risolvere il problema usando gli operatori standard di Kubernetes e quindi sfruttando il meccanismo dei controller e delle CRDs. Questa non si è rivelata la soluzione più adatta principalmente perché questo tipo di approccio è rigido, poco consono alla realizzazione di logiche flessibili e variabili nel tempo. Per questo motivo si è deciso di implementare una soluzione basata sull'utilizzo dei framework Knative e Krules, i quali sfruttando il paradigma event-driven permettono un'elevata modularità, riusabilità del codice e rendono il sistema resiliente e versatile. Tali caratteristiche risultano particolarmente apprezzate da parte degli operatori di rete in quanto abilitano la maintenance di sistemi complessi. Nello specifico Knative fornisce tutta l'infrastruttura per creare e gestire eventi mentre KRules permette di scrivere delle regole in Python basate sul paradigma evento-condizione-azione. Mediante la definizione di queste regole è possibile definire a run-time il comportamento che dovrà avere l'applicazione. In altre parole la soluzione implementata permette, in maniera semplice e dichiarativa, di gestire eventi provenienti da una sorgente generica e influenzare il ciclo di vita delle VNF.

Ringraziamenti

Un ringraziamento speciale al mio supervisore aziendale Marco Signorelli, project manager per la divisione Innovation di TIM, che si è mostrato sempre disponibile e mi ha aiutato durante il lavoro di tesi nonostante le circostanze generate dallo smart working. Vorrei ringraziare inoltre Airspot e in particolare Antonio Murciano, Alberto Degli Esposti e Lorenzo Campo per il supporto che mi hanno fornito nell'utilizzo del framework KRules da loro sviluppato. Ringrazio anche Raffaele Giuseppe Trani per la sua disponibilità e gentilezza nell'assistermi nella fase iniziale del lavoro di tesi. Infine ringrazio la mia famiglia, amici e colleghi universitari per il supporto datomi in questi anni e senza il quale non avrei raggiunto gli stessi risultati.

Indice

Capitolo 1

Introduzione

Negli ultimi anni il panorama tecnologico è stato caratterizzato sempre più da quella che oggi viene chiamata cloudification, ovvero la migrazione di applicazioni e servizi da computer e server locali sulla nuvola di internet. Questa migrazione ha visto sicuramente l'alternarsi di diverse fasi, ma ciò che caratterizza in particolar modo l'evoluzione degli ultimi anni è l'utilizzo dei container. Un software container è un tipo di virtualizzazione che opera a livello del sistema operativo incapsulando un intero ambiente runtime (l'applicazione, le binary libraries che traducono il sorgente in linguaggio macchina e i configuration files) in un pacchetto che il kernel Os considera in modo isolato da ogni altro processo. I container hanno iniziato a sostituire le macchine virtuali in quanto più semplici da installare e più leggeri perché gestiscono in modo granulare le risorse di calcolo usando solo le risorse della macchina host necessarie. Considerando che ogni macchina virtuale è essenzialmente una copia virtuale dell'hardware della macchina host e utilizza il proprio sistema operativo, può essere pesante e lenta in termini di risorse, consumando al contempo una grande quantità di memoria e potenza di elaborazione. Ogni macchina virtuale può anche avere dimensioni significative, il che limita la sua portabilità e complica la sua condivisione. La containerizzazione è stata sviluppata per risolvere molti dei problemi della virtualizzazione. Lo scopo dei contenitori è incapsulare un'applicazione e le sue dipendenze all'interno del proprio ambiente. Ciò consente loro di utilizzare le stesse risorse di sistema e lo stesso sistema operativo, lavorando però in maniera isolata. Poiché le risorse non vengono sprecate per l'esecuzione delle attività di sistemi operativi separati, la containerizzazione consente una distribuzione delle applicazioni molto più rapida e leggera. Ogni immagine del contenitore può avere una dimensione di pochi megabyte, semplificando

la condivisione, la migrazione e lo spostamento [1].

Una delle tecnologie più promettenti e apprezzate per la gestione dei container è Kubernetes. L'orchestrazione di Kubernetes consente di creare servizi applicativi che si estendono su più container, programmare tali container in un cluster, gestirne la scalabilità e l'integrità nel tempo.

Questo lavoro di tesi si concentra maggiormente sull'impatto della cloudification sugli operatori di telecomunicazione. L'evoluzione degli ultimi anni vede quindi le compagnie di telecomunicazioni cambiare il loro approccio riprogettando l'architettura centrale. Il processo evolutivo prevede il graduale passaggio da centrali fondate sul massiccio impiego di hardware specifico, fornito dai vendor di apparati di telecomunicazione, ad un approccio basato su Data Center con hardware e software general purpose che ospitano funzionalità di rete virtualizzate (VNF). Tale approccio permette agli operatori di rete di allentare il vendor lock-in nei confronti dei fornitori e aumentare il livello di flessibilità, efficacia ed efficienza nelle proprie centrali. Kubernetes può giocare un ruolo fondamentale nell'orchestrazione dei servizi di rete virtuali.

In questo contesto, una sfida chiave è rappresentata dal compromesso tra i compiti di orchestrazione responsabili del gestore di Kubernetes e quelle responsabili dell'orchestratore specifico per i servizi di rete (come ONAP). Un esempio significativo di tale tematica può essere individuata nella gestione della configurazione del data plane di una VNF di tipo Load Balancer. Nativamente Kubernetes non gestisce il livello di dataplane dei servizi e quindi bisogna decidere se utilizzare un orchestratore specifico per le funzionalità di rete (ONAP) oppure sviluppare degli add-on a Kubernetes (Specific Operator) associati a ciascuna VNF. Questa tesi valuta i vantaggi e limiti della soluzione basata su Specific Operator e propone una soluzione alternativa di carattere innovativo (Generic Operator).

1.1 Obiettivo

L'obiettivo di questa tesi è definire e sviluppare un operatore Kubernetes, model driven e non specifico per una determinata applicazione, in grado di acquisire un numero di input generico (fonte dati che può influenzare il comportamento delle VNF), trattarli con una logica di business programmabile dall'esterno e non nota a priori, e generare un certo numero di output arbitrari (come sollecitazione agli input ricevuti).

Cosa si intende con input generico? In primo luogo il numero di sorgenti che possono essere prese in considerazione e che possono generare informazioni rilevanti non deve essere fisso. Inoltre, la sorgente del dato deve essere generica, come ad esempio un file, una risorsa Kubernetes, un database esterno, e così via. Per concludere, il dato stesso potrebbe essere qualsiasi cosa, una particolare riga di un file, l'attributo di un oggetto, il corpo di una richiesta http, ecc.

Per quanto riguarda l'output valgono più o meno le stesse regole dell'input in quanto a numero e tipologie di possibili output. Alcuni esempi di output possono essere: riconfigurazione di una risorsa Kubernetes, modifica di un file o di un database interno o esterno al cluster e creazione di nuove risorse Kubernetes.

Queste specifiche consentirebbero di specializzare l'operatore a run-time, senza doverlo riprogrammare, permettendo ad esso di controllare servizi generici semplicemente cambiando la logica di business. In altre parole, il sistema deve essere in grado di modificare il suo comportamento, a seconda del caso d'uso, in maniera semplice e dichiarativa.

Questo paradigma applicativo ben si colloca nella gestione del ciclo di vita dei servizi basati su Virtual Network Function (VNF) in una rete di telecomunicazioni ed in particolare copre le fasi di maintenance di un servizio tramite il concetto di closed loop. Il closed loop può essere visto come un sistema che recepisce informazioni dall'ambiente in cui agisce (input), analizza le informazioni pervenute (logica di business) e effettua una retroazione (output) per configurare al meglio la rete.

Uno use case interessante potrebbe riguardare l'instanziazione di nuovi oggetti di rete (proxy, firewall, ecc) nel caso venga individuata una sofferenza prestazionale delle virtual network function in campo in un determinato istante. Questo esempio risulta particolarmente significativo perchè, per raggiungere lo scopo, non sarà sufficiente scalare l'oggetto tramite la creazione di una nuova istanza software dello stesso ma sarà necessario andare ad agire anche sul piano di indirizzamento a livello di data plane. Un altro use case particolarmente interessante è quello che ipotizza una modifica della configurazione della rete esistente non in funzione di parametri tecnici caratteristici dell'ambiente in cui operano le network function ma di informazioni provenienti da ambienti esterni alla rete di telecomunicazioni. Ad esempio si può ipotizzare che una porzione di rete venga riconfigurata a fronte di una situazione di emergenza pubblicata in un sistema di allerta della protezione civile.

Capitolo 2

Stato dell'arte

2.1 Background

2.1.1 Kubernetes

Kubernetes (K8s) è un software open-source per l'automazione del deployment, scalabilità, e gestione di applicativi in containers [2]. Inizialmente sviluppato da Google, adesso è mantenuto da Cloud Native Computing Foundation. Funziona con molti sistemi di containerizzazione, compreso Docker.

Attraverso il deployment di Kubernetes, si ottiene un cluster. Un cluster Kubernetes è un'insieme di macchine, chiamate nodi, che eseguono container gestiti da Kubernetes. Un cluster ha almeno un Worker Node. Il/I Worker Node ospitano i Pod che eseguono i workload dell'utente. Il/I Control Plane Node gestiscono i Worker Node e tutto quanto accade all'interno del cluster. Per garantire la high-availability e la possibilità di failover del cluster, vengono utilizzati più Control Plane Node. In figura 2.1 è rappresentato il diagramma di un cluster Kubernetes con tutti i suoi componenti e le loro relazioni.

I componenti del Control Plane sono responsabili di tutte le decisioni globali sul cluster (ad esempio, lo scheduling) oltre che a rilevare e rispondere agli eventi del cluster (ad esempio, l'avvio di un nuovo pod quando il valore replicas di un deployment non è soddisfatto). I componenti sono:

- API server: componente che espone le Kubernetes API. L'API server è il front end del control plane di Kubernetes. La principale implementazione di un server Kubernetes API è kube-apiserver.

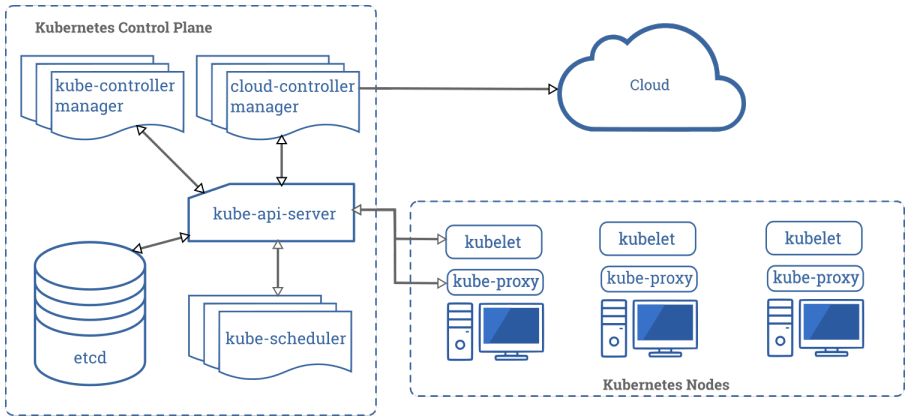

Figura 2.1. diagramma di un cluster Kubernetes

- etcd: database key-value ridondato, usato da Kubernetes per salvare tutte le informazioni del cluster.
- kube-scheduler: controlla i pod appena creati che non hanno un nodo assegnato, e dopo averlo identificato glielo assegna.
- kube-controller-manager: gestisce i controllers. Esempi di controllers sono: Node Controller, Replication Controller, Endpoints Controller, Service Account e Token Controllers.
- cloud-controller-manager: aggiunge logiche di controllo specifiche per il cloud. Questa componente permette di collegare il cluster con le API del cloud provider e separa le componenti che interagiscono con la piattaforma cloud dai componenti che interagiscono solamente col cluster.

I componenti presenti sui Worker Nodes vengono eseguiti su ogni nodo, mantenendo i pod in esecuzione e fornendo l'ambiente di runtime Kubernetes. Questi componenti si dividono in:

- Kubelet: agente che viene eseguito su ogni nodo del cluster. Si assicura che i container siano eseguiti in un pod. La kubelet riceve un set di PodSpecs che vengono forniti attraverso vari meccanismi, e si assicura che i container descritti in questi PodSpecs funzionino correttamente e siano sani.
- Kube-proxy: proxy eseguito su ogni nodo del cluster, responsabile della gestione dei Kubernetes Service. I kube-proxy mantengono le regole di

networking sui nodi. Queste regole permettono la comunicazione verso gli altri nodi del cluster o l'esterno.

- Container runtime: software responsabile per l'esecuzione dei container. Kubernetes supporta diversi container runtimes, tra i quali Docker.

Kubernetes è un sistema abbastanza semplice da usare inizialmente, ma capire come funziona al suo interno non è affatto facile. Una delle parti più complesse, e probabilmente più critiche è la gestione del Networking.

Alla base del networking in Kubernetes c'è un concetto fondamentale: ogni Pod ha un unico IP [3]. Questo IP è condiviso da tutti i container nel Pod, ed è indirizzabile da tutti gli altri Pods. Su ogni nodo Kubernetes ci sono dei container chiamati "sandbox containers", il cui unico scopo è quello di riservare e trattenere un network namespace (netns) condiviso da tutti i container in un Pod. In questo modo l'IP di un Pod non cambia anche se un container viene distrutto e ne viene creato un altro al suo posto. Il grande vantaggio di avere un IP per Pod è che non ci possono essere collisioni di IP o porte con le applicazioni sottostanti e non bisogna preoccuparsi di quale porta useranno.

Detto questo, l'unico requisito imposto da Kubernetes è che gli IP dei Pod siano instradabili/accessibili da tutti gli altri Pod, a prescindere dal nodo in cui sono questi Pod. A questo punto è necessario analizzare la comunicazione intra-node e inter-node.

Comunicazione intra-node

Su ogni nodo Kubernetes c'è un root network namespace (root netns). L'interfaccia di rete principale eth0 è nella root netns. Allo stesso modo, ogni pod ha la sua netns, con una connessione Ethernet virtuale alla root netns. Questo collegamento è una pipe che da una parte termina nel netns del pod e dall'altra nel root netns. L'interfaccia di rete del pod viene indicata con eth0, mentre l'altra con vethxxx. Questo viene fatto per ogni pod del nodo. I pod possono parlare internamente al nodo grazie a un bridge cbr0. Osservando la figura 2.2 si supponga che un pacchetto deve andare dal pod1 al pod2. Tale pacchetto lascerà il netns del pod1 dall'interfaccia eth0 ed entrerà nel root netns dall'interfaccia vethxxx. A questo punto passerà dal cbr0, il quale scoprirà il destinatario usando un ARP request. vethyyy dirà di essere lui ad avere quell'IP e il bridge saprà dove inoltrare il pacchetto. Infine il pacchetto raggiungerà vethyyy, attraverserà la pipe e raggiungerà il netns del pod2.

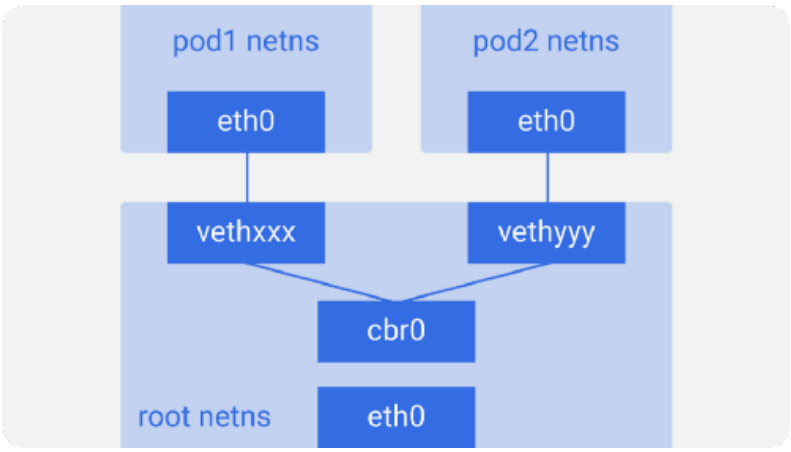

Figura 2.2. Comunicazione intra-node

In questo modo i container sullo stesso nodo possono parlare tra loro. Naturalmente ci sono anche altri metodi, ma questo è probabilmente il più facile, e anche quello che utilizza Docker.

Comunicazione inter-node

Come già anticipato i pod devono poter comunicare anche tra i nodi. Kubernetes non fornisce alcun componente per configurare la rete, questo è demandato a CNI plug-ins. K8s definisce una serie di comportamenti che devono essere supportati da ogni network provider, a prescindere dall'implementazione [4]. Kubernetes impone i seguenti requisiti fondamentali per ogni implementazione di networking:

- I pod su un nodo possono comunicare con tutti i pod su tutti i nodi senza bisogno del NAT.
- Gli agents su un nodo (es. system daemons, kubelet) possono comunicare con tutti i pod su quel nodo.

Per tutte quelle piattaforme che supportano l'esecuzione dei Pod sull'host network (es. Linux):

- I pod sull'host network di un nodo possono comunicare con tutti i pod su tutti i nodi senza bisogno del NAT.

Il modo in cui pods su nodi diversi comunicano non è importante dal punto di vista di Kubernetes, il quale si occupa solo di definire dei requisiti da mantenere, l'importante è che questa comunicazione sia possibile. Ad esempio

si può usare una soluzione L2 (switching), L3 (routing) oppure overlay networks [5]. La soluzione L3 è sicuramente più scalabile rispetto quella di livello due. Questa può essere realizzata popolando il default gateway con le rotte per le sottoreti come mostrato nella figura 2.3. Le rotte verso 10.1.1.0/24 e 10.1.2.0/24 sono configurate per essere inoltrare rispettivamente verso i nodi 1 e 2. Questa operazione può essere automatizzata mantenendo la tabella delle rotte aggiornata quando i nodi vengono aggiunti o tolti nel cluster. In

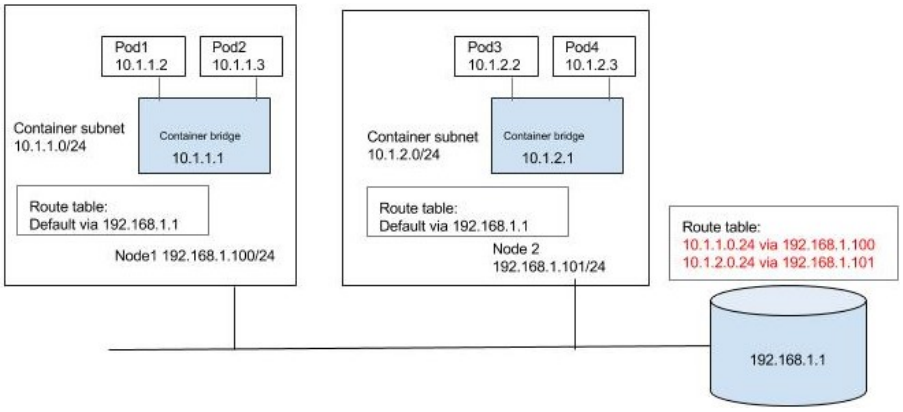

Figura 2.3. Soluzione L3

alternativa ogni nodo può essere popolato con le rotte verso le altre sottoreti come mostrato nella figura 2.4

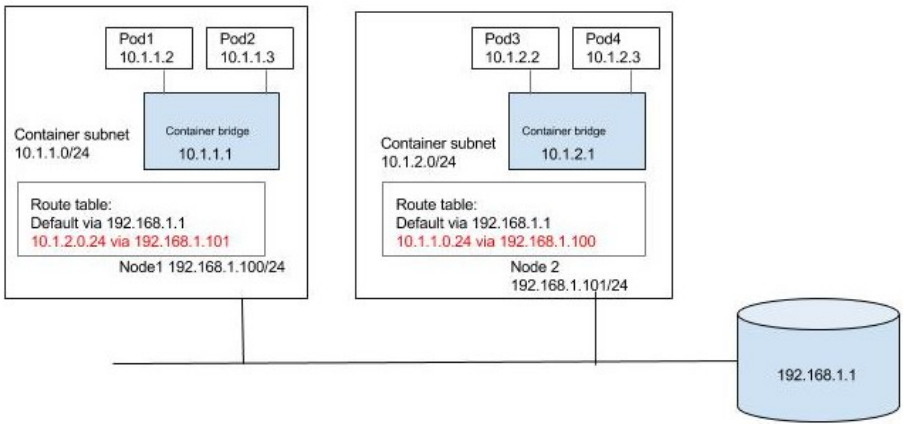

Figura 2.4. Seconda soluzione L3

Kubernetes richiede che i nodi siano in grado di raggiungere tutti i pod, anche se questi pod sono in una overlay network. Allo stesso modo i pod

dovrebbero raggiungere ogni nodo. Affinché questo sia possibile è necessario settare correttamente le host routes nel nodo. Il traffico che va da nodo a nodo deve essere incapsulato nel nodo sorgente. I pacchetti incapsulati sono inoltrati al nodo destinatario dove verranno decapsulati. Una soluzione può essere quella di usare uno dei meccanismi di incapsulamento esistenti di Linux. Quindi c'è bisogno dell'interfaccia di un tunnel (con VXLAN, GRE, ecc.) e di un host route per far si che il traffico da pod-to-pod sia inoltrato attraverso l'interfaccia del tunnel.

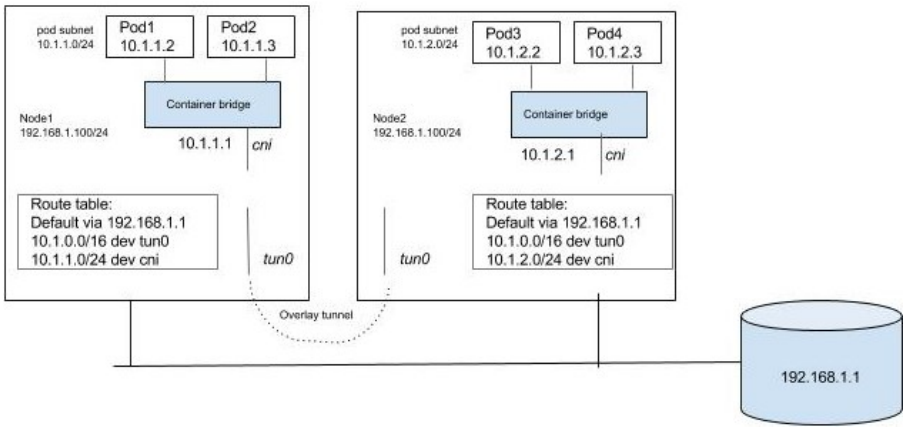

Figura 2.5. Overlay solution

2.1.2 Multus

Multus è un container network interface (CNI) plugin per Kubernetes che permette di installare più interfacce di rete ai pod [6]. In Kubernetes, di base, ogni pod ha solo un'interfaccia di rete (senza contare quella di loopback). Grazie a Multus è possibile creare un pod multi-homed che ha più interfacce. Multus in realtà agisce come un "meta-plugin", un CNI plugin che può richiamare altri CNI plugins. Il diagramma in figura 2.6 mostra un pod con tre interfacce: eth0, net0 e net1. eth0 connette il pod alla rete del cluster Kubernetes, per collegarlo ai server/servizi(Kubernetes api-server, kubelet, ecc.). net0 e net1 sono interfacce di rete aggiuntive e connettono il pod ad altre reti usando altri CNI plugins (es. vlan/vxlan/ptp).

Per creare interfacce aggiuntive usando multus bisogna creare una configurazione per ogni interfaccia che si vuole collegare al pod. Questo si può fare creando una Custom Resources (CR) che definisce la configurazione per

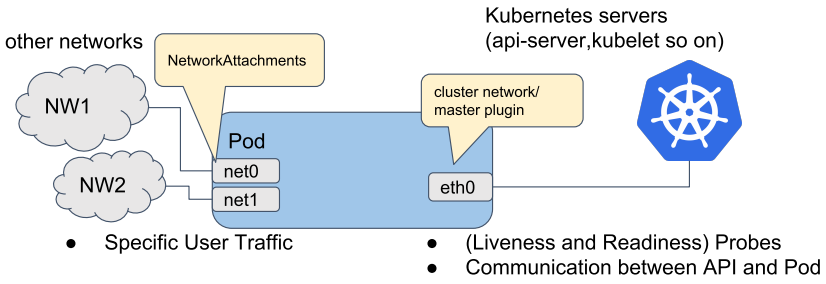

Figura 2.6. Interfacce di rete collegate da Multus

l'interfaccia. Questa CR sarà del tipo "NetworkAttachmentDefinition". Questo tipo di risorsa viene definito da Multus mediante una Custom Resource Definition (CRD). Di seguito un esempio di configurazione di un'interfaccia.

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: macvlan-conf-1
spec:
  config: '{
            "cniVersion": "0.3.0",
            "type": "macvlan",
            "master": "eth1",
            "mode": "bridge",
            "ipam": {
                "type": "host-local",
                "ranges": [
                    [ {
                         "subnet": "10.10.0.0/16",
                         "rangeStart": "10.10.1.20",
                         "rangeEnd": "10.10.3.50",
                         "gateway": "10.10.0.254"
                    } ]
                ]
            }
        }'
```

In questo file bisogna inserire i dettagli sull'interfaccia di rete dell'host, la sottorete, il range di indirizzi da assegnare ai pod e il gateway. A questo punto basterà mettere all'interno della sezione annotation dei pod la stringa

"k8s.v1.cni.cncf.io/networks: macvlan-conf-1" e Multus si occuperà di configurare correttamente l'intefaccia di rete del pod con le specifiche indicate nella risorsa NetworkAttachmentDefinition chiamata "macvlan-conf-1".

Come verrà spiegato successivamente, Multus è stato utilizzato nella realizzazione del caso d'uso per collegare le VNF tra loro. Nello specifico ogni VNF ha tre interfacce: eth0 che la collega ai servizi Kubernetes e le altre due, create mediante multus, che la collegano rispettivamente alla network function precedente e successiva. In altre parole è stata creata una chain di VNF. Questo argomento verrà trattato con maggiore dettaglio nel capitolo riguardante l'implementazione del caso d'uso.

2.1.3 Controllers

Per lavorare con Kubernetes, si usano gli oggetti API Kubernetes per descrivere lo stato desiderato del cluster: quali applicazioni o altri carichi di lavoro si vogliono eseguire, quali immagini del container usano, numero di repliche, quali risorse di rete e disco si vogliono rendere disponibile e altro ancora [7]. Si può impostare lo stato desiderato creando oggetti mediante l'utilizzo dell'API di Kubernetes, in genere tramite l'interfaccia della riga di comando, kubectl. Si può anche utilizzare direttamente l'API di Kubernetes per interagire con il cluster e impostare o modificare lo stato desiderato.

Una volta impostato lo stato desiderato, il Kubernetes Control Plane si occupa di garantire una corrispondenza tra stato corrente del cluster e stato desiderato. Per fare ciò, Kubernetes esegue automaticamente una serie di attività, come l'avvio o il riavvio dei contenitori, il ridimensionamento del numero di repliche di una determinata applicazione e altro ancora.

Kubernetes contiene una serie di astrazioni che rappresentano lo stato del sistema: applicazioni e carichi di lavoro distribuiti in container, le loro risorse di rete e disco associate e altre informazioni su ciò che sta facendo il cluster. Queste astrazioni sono rappresentate da oggetti nell'API di Kubernetes. Gli oggetti di base di Kubernetes includono: Pod, Service, Volume, Namespace. Inoltre, Kubernetes contiene una serie di astrazioni di livello superiore denominate Controllori. I controller sono circuiti di controllo che osservano lo stato del cluster, e apportano o richiedono modifiche quando necessario. In particolare, un controller che interagisce con un oggetto di base trova il suo stato desiderato attraverso l'API server, quindi comunica direttamente con l'oggetto per portare il suo stato corrente più in linea possibile con lo stato desiderato. Questa serie di operazioni realizzano quello che viene chiamato

control loop. I controllori implementano questi control loop, operando non solo su oggetti di base ma anche su altre risorse.

In generale, il control loop è composto dalle seguenti fasi [8]:

1. legge lo stato delle risorse (registrando dei watches sulle risorse a cui è interessato);
2. cambia lo stato degli oggetti nel cluster o esterni al cluster. Per esempio, lancia un pod, crea un network endpoint o esegue una query alle cloud API;
3. aggiorna lo stato delle risorse del punto 1 nell'etcd tramite il server delle API;
4. ripete il ciclo tornando al punto 1.

Non importa quanto sia complesso o semplice il controller, questi tre step (leggere lo stato della risorsa, modificare il cluster o l'esterno del cluster, aggiornare lo stato della risorsa) rimangono gli stessi.

La figura 2.7 illustra le componenti che entrano in gioco in un control loop, con il loop principale del controller al centro. Il loop principale è costantemente in esecuzione all'interno del processo del controller. Questo processo di solito viene eseguito dentro un pod nel cluster.

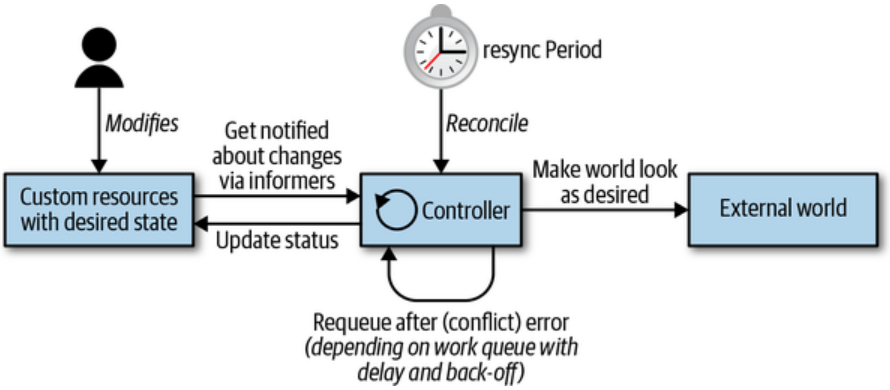

Figura 2.7. Kubernetes control loop

2.2 Architettura Event-Driven

Event-Driven Architecture (EDA) è un modello o un paradigma architetturale per software che supporta la produzione, il rilevamento, il consumo e

la reazione all'evento o a un cambiamento significativo nello stato del sistema [11]. Questa struttura è costituita da creatori (o sorgenti) di eventi e consumatori di eventi. I creatori sono quelli che producono eventi mentre i consumatori sono quelli che vengono influenzati dagli eventi che si verificano e sono anche coinvolti nell'elaborazione degli eventi.

I motivi per cui usare un'architettura basata sugli eventi sono molteplici, ma possiamo riassumerli in questi sei punti:

- Struttura scarsamente accoppiata: il creatore e il consumatore dell'evento si possono associare liberamente mediante l'aiuto di un bus di eventi che li aiuta ad accoppiarsi o disaccoppiarsi in risposta a vari eventi.
- Analisi in tempo reale: l'architettura basata su eventi è altamente ottimizzata per l'analisi in tempo reale. Ha una capacità molto più elevata di trovare, analizzare e quindi rispondere ai pattern in tempo prima che si verifichino eventi critici.
- Versatilità: un sistema che segue un'architettura basata sugli eventi permette una maggiore reattività perché i sistemi guidati dagli eventi sono stati progettati per lavorare in ambienti imprevedibili, non lineari e asincroni. Questi sistemi sono altamente adattabili in diverse circostanze.
- Comunicazione multicast: il creatore può mandare un evento contemporaneamente a più di un consumatore interessato.
- Comunicazione asincrona: il creatore non attende che il consumatore abbia ricevuto/elaborato un evento prima di mandarne un altro. L'architettura basata sugli eventi può eseguire operazioni asincrone. Ciò si traduce in prestazioni più affidabili dell'architettura.
- Comunicazione puntuale: i creatori possono pubblicare eventi piccoli e specifici invece di attendere per un singolo evento aggregato.

Le architetture basate su eventi e i modelli di programmazione reattiva non sono un'idea nuova. Ma, con l'avvento delle architetture cloud-native, dei microservizi, dei container e della computazione serverless, torna utile riconsiderare un approccio event-driven in un contesto cloud-native.

2.2.1 Publish/Subscribe (Pub/Sub) Messaging

É una forma di comunicazione asincrona service-to-service usata nelle architetture a microservizi e serverless. Nel modello pub/sub, ogni messaggio

pubblicato sotto uno specifico topic è immediatamente ricevuto da tutti i consumatori sottoscritti a quel topic. Una volta ricevuto un messaggio non si potrà più ripetere, e un nuovo sottoscrittore all'evento non vedrà i messaggi precedenti alla sua sottoscrizione, ma solo quelli successivi. Un subscriber non ha bisogno di sapere niente del publisher, e viceversa. In questo modo viene realizzato il disaccoppiamento tra creatore e consumatore di eventi, ed è questo che rende il modello pub/sub una possibile soluzione al problema di questa tesi. Perchè questo permette di disaccoppiare le sorgenti che generano gli eventi (che possono essere viste come gli input del nostro problema) dai consumatori che si occuperanno di elaborare questi input (questa elaborazione può essere vista come l'output del nostro problema).

2.2.2 Eventi in Kubernetes

Gli eventi Kubernetes sono oggetti che forniscono informazioni su ciò che sta accadendo all'interno di un cluster, ad esempio quali decisioni sono state prese dallo scheduler o perché alcuni pod sono stati rimossi dal nodo [12]. Gli eventi Kubernetes vengono automaticamente generati quando ad esempio cambia lo stato delle risorse, quando ci sono errori o altri messaggi che devono essere dichiarati al sistema. Questi sono più che altro messaggi di servizio, che vengono usati per introspezione e debug. Ma in realtà questi eventi portano con se molte informazioni importanti e utili.

Come si potrà vedere nell'implementazione del caso d'uso della tesi, questi eventi verranno usati per ottenere informazioni sulle risorse interne al cluster, in particolare per sapere quando viene creata una risorsa. Ma questo è solo un esempio, le informazioni che si possono ricavare da questi eventi sono tantissime.

Questi eventi però non seguono una specifica o un formato standard utilizzabile in maniera univoca da qualunque sistema. Sono comprensibili all'interno di Kubernetes, ma in molti casi è necessario avere uno standard univoco e riconosciuto per descrivere un evento. Queste specifiche vengono fornite da CloudEvents.

2.2.3 CloudEvents

La mancanza di un metodo comune per descrivere gli eventi porta gli sviluppatori a dover imparare come consumare un evento ogni volta che ne arriva uno descritto in un modo nuovo. Ciò limita anche la possibilità per le librerie, gli strumenti e l'infrastruttura di aiutare la consegna delle informazioni

degli eventi attraverso ambienti come SDK, router di eventi o sistemi di tracciamento. La portabilità e la produttività che si possono ottenere dai dati degli eventi sono complessivamente ostacolate.

CloudEvents è una specifica per descrivere i dati degli eventi in un formato comune per fornire l'interoperabilità tra servizi, piattaforme e sistemi [13]. CloudEvents fornisce SDK per Go, JavaScript, Java, C#, Ruby e Python che possono essere utilizzati per creare router di eventi, sistemi di tracciamento e altri strumenti.

2.3 Related work

A questo punto è facile intuire come sia necessario entrare più nel profondo di Kubernetes e "programmare Kubernetes" per poter raggiungere l'obiettivo della tesi.

Programmare Kubernetes può voler dire diverse cose, ma ciò che si vuole far intendere in questa tesi è questo: sviluppare un'applicazione nativa di Kubernetes che interagisce direttamente con il server API, interrogando lo stato delle risorse e/o aggiornando il loro stato [8]. In generale ci sono diversi modi per customizzare e/o estendere Kubernetes. Questo può essere fatto usando file e flag di configurazione per i componenti del control plane come kubelet o l'API server di Kubernetes, oppure attraverso le seguenti estensioni:

- Cloud providers
- Kubelet plug-ins.
- Kubectl plug-ins.
- Estensione per l'accesso all'API server, come l'admission control con i webhooks.
- Custom Resources e Custom Controllers
- Custom API servers.
- Scheduler exetension.

Nel contesto di questa tesi verranno usati solo Custom Resources e Custom Controllers (e di conseguenza gli operatori) in quanto sono l'unica soluzione insieme alle custom API servers che potrebbero aiutare nella ricerca della soluzione. Le custom API servers sono state scartate perchè anche se sono

più flessibili e superano alcuni limiti delle Custom Resource Definition, introducono anche maggiore complessità e il risultato finale non cambierebbe ai fini della tesi.

Nel paragrafo 2.1.3 è stato approfondito il concetto di controller. Adesso è fondamentale mettere a confronto controller e operator:

- I controller possono operare su risorse core di Kubernetes o su service, che sono tipicamente parte del Kubernetes controller manager nel control plane, o possono manipolare delle risorse custom definite dall'utente.
- Gli operatori sono controller che, insieme alle custom resources definite dagli utenti, includono una conoscenza operativa, come può essere la gestione del ciclo di vita di un'applicazione.

Una risorsa è un endpoint nell'API Kubernetes che archivia una raccolta di oggetti API di un certo tipo. Una Custom Resource (CR) è un'estensione dell'API di Kubernetes. Le CR vengono utilizzate per piccoli oggetti di configurazione interni senza alcuna logica di controllo corrispondente, definite in modo puramente dichiarativo. Una Custom Resource Definition (CRD) è anch'essa una risorsa Kubernetes e serve a descrivere una CR. Permette di aggiungere una propria CR al cluster Kubernetes e usarla proprio come viene usato qualunque altro oggetto nativo di Kubernetes.

2.3.1 Operatori

Un Operatore è un controller specifico per un'applicazione che estende le API di Kubernetes permettendo la creazione, configurazione e la gestione delle istanze di applicazioni stateful e complesse per conto di un utente Kubernetes. Un operatore si basa sui concetti di risorsa e di controller ma include conoscenze specifiche del dominio o dell'applicazione per automatizzare i task comuni [8].

Affinchè un operatore sia usato correttamente è necessario che le seguenti condizioni siano verificate:

- esistono processi operativi specifici del dominio che si vogliono implementare;
- le procedure per questa conoscenza operativa sono note e possono essere rese esplicite;
- tale operatore sia costituito da:

– un set di custom resource definitions che rappresentano lo schema specifico del dominio e custom resources relative alle CRDs le cui istanze rappresentano il dominio di interesse;

– un custom controller, che supervisiona le custom resources, potenzialmente insieme alle risorse principali.

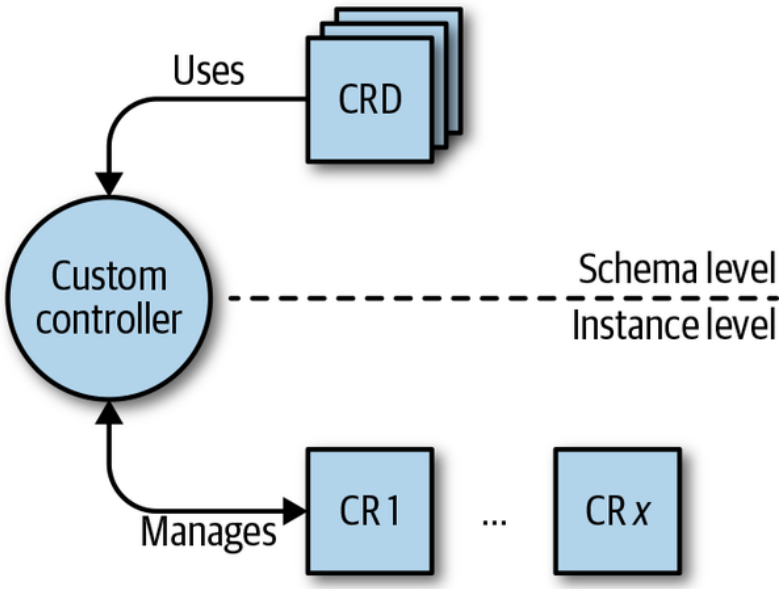

Figura 2.8. Il concetto di operatore

La figura 2.8 mostra tutti i componenti di un operatore.

Da queste definizioni si intuisce subito come un semplice operatore non è pensato per gestire casi generici così come ci si aspetta dall'obiettivo della tesi. Questo non significa che gli operatori sono inutili ma che da soli non sono una soluzione sufficiente. Nel capitolo 3 si affronterà meglio questo argomento.

Per semplificare la creazione di un operatore ci sono diverse soluzioni. Quella che è stata usata durante il lavoro di tesi è Kubebuilder. Kubebuilder è un framework per creare APIs Kubernetes usando custom resource definitions [9].

Un'altro tool interessante per creare operatori che è stato analizzato durante la fase di ricerca è KUDO.

2.3.2 KUDO

Kubernetes Universal Declarative Operator (KUDO) fornisce un approccio dichiarativo per la creazione e la gestione di operatori Kubernetes [10].
KUDO utilizza una logica interessante che permette all'utente di definire un *plan*, composto essenzialmente da *task* (che sono raggruppati in *phases*). Un task è un elemento di base che esegue piccole operazioni. KUDO offre quattro tipi di attività principali: Apply, Delete, Pipe and Toggle.

- Apply: permette di definire un elenco di risorse Kubernetes che verranno create (se non esistono) o aggiornate (se presenti);
- Delete: molto simile a un apply-task, elimina le risorse;
- Pipe: permette di generare file in un task e salvarli come Secret o ConfigMap per l'utilizzo nei passaggi successivi;
- Toggle: crea o elimina le risorse in base a un valore booleano del parametro.

Gli operatori possono essere personalizzati usando dei parametri. Più precisamente questi parametri vengono definiti in un file yaml. Qui è dove KUDO differisce da altri tool simili: i parametri degli operatori sono legati ai plans. In questo modo quando un parametro cambia, un plan viene automaticamente triggerato. Ogni parametro ha un campo trigger che specifica quale plan sarà eseguito quando il parametro cambia.

Questa feature è molto interessante ma il valore dei parametri può essere sovrascritto solo dall'utente mediante l'uso della CLI. Non è possibile modificare i parametri dinamicamente come risultato di alcune operazioni o input generico. Nonostante l'approccio dichiarativo nella creazione e gestione di operatori manca la possibilità di definire una logica centrale che cambi il suo comportamento sulla base di input generici. Per questo motivo KUDO è stato scartato.

Capitolo 3

Utilizzo di operatori standard in Kubernetes

Come primo passo, verso il raggiungimento dell'obiettivo, si è scelto di analizzare a fondo il funzionamento degli operatori Kubernetes e di sfruttare il meccanismo dei controllori. Come detto in precedenza i controllori da soli non forniscono una generalizzazione sufficiente, ma iniziando da questi lo scopo è di elaborare un'architettura in grado di rispettare i termini stabiliti da questo lavoro di tesi. Nei prossimi capitoli vedremo che la soluzione adottata alla fine non è quella basata sull'utilizzo degli operatori, ma lo scopo di questo capitolo è quello di capire le potenzialità di un'architettura di questo tipo e il motivo per cui non è stata scelta.

A tal fine è stato creato, mediante l'uso di Kubebuilder, un operatore Kubernetes di esempio che svolgesse alcuni compiti. Questo operatore è servito sia per iniziare a prendere confidenza con la programmazione di controller, sia per sperimentare le sue reali capacità e capire i suoi limiti.

3.1 Implementazione operatore di esempio

Nonostante l'esempio realizzato dovesse solo dare un'idea generale delle potenzialità degli operatori, si è cercato di rappresentare degli scenari utili e quanto più aderenti possibile a situazioni reali. Questo è stato fatto per verificare la fattibilità di alcune operazioni come ad esempio mettere il controller in ascolto su un Deployment e reagire quando quest'ultimo cambia il suo stato (ad esempio scalando il numero di istanze di pod attive in un determinato

istante), reperire informazioni da prometheus ed elaborarle, oppure ricavare informazioni dall'esterno del cluster come per esempio un server REST.

Il Controller è il componente fondamentale di un operatore. Ogni controller si concentra su un predefinito CRD (o Kind, come viene chiamato in Kubebuilder), ma può interagire con altri Kinds. È compito del controller garantire che, per ogni dato oggetto, lo stato effettivo del sistema (sia lo stato interno del cluster, sia lo stato potenzialmente esterno) corrisponda a quello desiderato nell'oggetto. Questo processo è denominato *reconciling*. La funzione che implementa il reconciling per uno specifico Kind è chiamata *Reconcile*. É all'interno di questa funzione che dovrà essere inserita tutta la logica di business che verrà eseguita dal controller. Ogni qual volta il controller viene triggerato, il metodo reconcile verrà eseguito. Alcuni dei casi per cui un controller può essere triggerato sono:

- creazione di una CR relativa al controller;
- ritorno di un errore da parte del metodo reconcile. In tal caso significa che c'è stato un errore e deve essere rieseguito;
- ritorno di un risultato da parte del metodo reconcile (ctrl.Result) con la proprietà RequeueAfter settata. Vuol dire che dopo un certo lasso di tempo definito nella proprietà il reconcile deve essere eseguito di nuovo;
- modifica di una CR relativa al controller da parte di un'altra risorsa;
- modifica di una CR relativa al controller da parte dell'utente;
- modifica di una risorsa su cui il controller ha impostato un watch.

Nello specifico, l'implementazione realizzata durante questa fase del lavoro di tesi è composta da due CR e dai relativi controller. Le due Custom Resources verranno chiamate CR1 e CR2. L'idea alla base è che nel cluster c'è un Deployment (la cui immagine è ad esempio un web server nginx) che modifica il suo numero di repliche come conseguenza dell'aumento delle richieste. Il controller di CR1 si occupa di osservare il Deployment (dietro registrazione di un watch) e modificare una proprietà di CR2. La proprietà aggiornata è quella che indica il numero di repliche del deployment. Il controller di CR2 come conseguenza della modifica dello stato reagisce andando a controllare se il valore di questa proprietà supera una certa soglia. Il risultato del controllo viene poi segnato su un'altra proprietà di CR2 che definisce lo stato corrente della risorsa.

Come è possibile notare ci sono due controller che rispondono ad eventi diversi e svolgono compiti diversi. Quello di CR1 viene triggerato quando il deployment che sta osservando subisce una modifica (oltre che nel momento in cui viene creata la CR relativa) e si fa carico di modificare CR2. Il controller di CR2, invece, viene triggerato quando la proprietà della custom resource viene modificata e il suo compito è quello di modificare un'altra proprietà della stessa risorsa. Quest'ultima operazione però è solo un esempio, il controller di CR2 potrebbe fare qualunque cosa dopo essersi accorto dell'aumento delle repliche del deployment, come ad esempio andare a modificare la configurazione di una funzione di rete (ipoteticamente un Load Balancer). Come già anticipato lo scopo di questo esempio era solo quello di testare la comunicazione tra gli operatori e altre risorse interne ed esterne al cluster programmando i controller e trovando in questo modo i loro limiti. Per questo motivo nel controller della CR1 oltre ad avere come input informazioni sul deployment interno a Kubernetes, sono state testate altre opzioni come interrogazioni al server Prometheus in esecuzione nel cluster. Per ottenere informazioni dal server è stata utilizzata la libreria client scritta in Go per prometheus. I dati ricavati nell'esempio non vengono utilizzati, ma in un'applicazione reale potrebbero essere usati per cambiare il comportamento delle funzioni di rete sulla base dei dati di monitoraggio raccolti.

Allo stesso modo nel controller della CR1 vengono fatte delle interrogazioni verso un server esterno al cluster, che espone delle API interrogabili mediante richieste HTTP, al solo scopo di mostrare che è possibile cambiare il comportamento delle risorse in un cluster Kubernetes anche usando informazioni esterne. Un esempio può essere il monitoraggio di un sito della protezione civile, piuttosto che di un ente che espone informazioni rilevanti mediante delle API. Per un operatore di telecomunicazioni è importante poter riconfigurare la propria rete sulla base dei cambiamenti che interessano l'ambiente circostante. Soprattutto in questo periodo più che mai ci si sta rendendo conto come i fattori esterni possano influenzare il traffico di rete. Ne è un esempio il periodo di lockdown degli scorsi mesi che ha visto milioni di persone connesse contemporaneamente ad internet. In casi come questi è fondamentale avere dei sistemi dinamici che sanno agire prontamente ai cambiamenti del traffico di rete dovuto a fattori esterni.

Per poter interrogare i server esterni è stata utilizzata la libreria HTTP client scritta in GO.

3.2 Limiti operatori

Dall'esempio realizzato si intuisce che mediante la programmazione degli operatori e in particolar modo dei controller è possibile usare qualunque tipo di input, da semplici proprietà delle CRD, o di altre risorse appartenenti al cluster, a metriche raccolte da Prometheus o perfino esterne al cluster. Come conseguenza dell'elaborazione di questi dati è possibile fare diverse operazioni, come ad esempio si può voler creare, distruggere o cambiare la configurazione di una risorsa, oppure interfacciarsi con un servizio esterno ad esempio tramite richieste HTTP. Queste operazioni sono tutte ugualmente implementabili all'interno di un controller. È questo il motivo che ha portato all'utilizzo dei controllers: essi possono agire sulle risorse core di Kubernetes, deployments o servizi, che sono tipicamente parte del Kubernetes controller manager o possono manipolare e registrare dei watch su risorse custom definite dall'utente [8]. In altre parole mediante i controller possiamo gestire e operare su tutto ciò che riguarda Kubernetes e mediante delle librerie in Go anche ciò che è al di là di Kubernetes.

Ciò che si può notare però è che un operatore ci permette di lavorare con uno specifico caso, come abbiamo visto anche dalla sua definizione, senza darci la possibilità di gestire in modo generale un'applicazione. In altre parole ciò che manca è la possibilità di generalizzare la logica di business, ovvero il nucleo di elaborazione che rende operativa un'applicazione. Per lo stesso motivo il numero e il tipo di input dell'applicazione, ovvero i dati su cui poter fare dell'elaborazione, è specifico da caso a caso e tipico della singola implementazione. Allo stesso modo, neanche le funzioni di rete da configurare possono variare in base al caso d'uso, perchè se cambiassero, in tipo o in numero, cambierebbe il modo di configurarle e quindi bisognerebbe riprogrammare l'operatore.

È chiaro quindi che per raggiungere l'obiettivo della tesi è stato necessario fare un passo in più per trovare il modo di ottenere il comportamento desiderato sfruttando gli operatori e superando queste limitazioni.

3.3 Operatore generico

In questa fase del lavoro di tesi ci si è concentrati sulla ricerca di una soluzione al problema della generalizzazione. In un primo momento si è cercato di raggiungere questa soluzione usando gli operatori e creando un framework che permettesse la composizione di questi operatori a seconda del caso d'uso. In

un secondo momento ci si è resi conto che il primo rimedio non era abbastanza flessibile e non si adattava poi così bene al flusso di lavoro di un'applicazione cloud, così è stata presa in considerazione una seconda soluzione.

In questo paragrafo verrà analizzata la prima soluzione, quella basata sugli operatori. In realtà verrà fatto vedere solo uno schema del framework che avrebbe dovuto risolvere il problema della generalizzazione. Verrà mostrata l'idea di base di questo framework e verranno mostrati i punti di forza e di debolezza rispetto a quella che è stata scelta alla fine.

Come già anticipato questo primo rimedio si basa sull'utilizzo degli operatori Kubernetes per cercare di ottenere un risultato che rispetti i vincoli imposti dall'obiettivo della tesi. I fattori in gioco sono principalmente due: gli input dell'applicazione e le funzioni di rete da gestire sulla base degli input. Gli input possono essere di tipo diverso e bisogna poterli definire a seconda del caso d'uso, segnalando a quali funzioni di rete serve quell'informazione ed indicando per ogni input e VNF l'azione da intraprendere.

Si può pensare quindi di avere per ogni funzione di rete un operatore dedicato che verrà quindi creato appositamente per essa. Questo sarà costituito da una CRD che definisce alcune proprietà che verranno usate dal corrispettivo controller. Quest'ultimo quindi avrà al suo interno la logica di gestione della singola funzione per la quale è stato creato. In questo modo l'operatore verrà utilizzato, come previsto dalla sua definizione, per automatizzare i compiti di una specifica applicazione, ovvero quella di gestire la funzione di rete. Ad esempio potrà aumentare o diminuire il numero di repliche della risorsa, modificare la config map associata ad essa, cambiare delle variabili d'ambiente e così via. La CRD invece avrà una serie di proprietà che verranno modificate in base ai dati in input e serviranno al controller per sapere come configurare il nodo di rete. Il problema di questa CRD è che dovrà avere un'interfaccia standard per tutte le funzioni di rete, altrimenti si perderebbe la possibilità di aggiungere tali funzioni senza modificare il codice del core del framework. Per questo motivo bisogna definire uno standard secondo il quale i dati vengono memorizzati nella custom resource.

Per quanto riguarda il componente centrale di questo framework, il suo compito è quello di creare l'associazione tra i dati in ingresso e le funzioni di rete. Queste associazioni dipendono dal caso d'uso e verranno passate al core mediante un metodo da definire ma ad esempio potrebbe essere una CRD. Quindi in base all'applicazione per cui verrà usato, al framework verranno passate queste associazioni input - network function e sarà questo componente che si occuperà di prendere le informazioni di input e modificare le CR delle corrispettive funzioni di rete.

Il componente centrale che realizza la logica di business per mezzo delle associazioni passate, potrebbe essere implementato mediante un controller che viene triggerato ogni volta che viene modificata la CR associata.

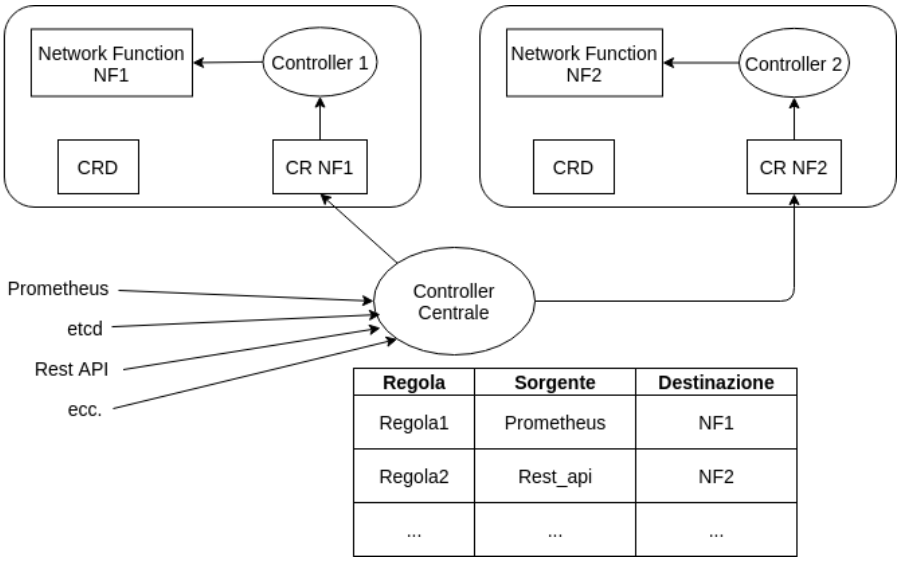

Figura 3.1. Diagramma operatore generico

Uno dei principali problemi di questo schema è la definizione dello standard da applicare alle CRD delle network function. Qualunque sia la funzione di rete in questione, la sua CR associata dovrà avere un'interfaccia comune che permetta al controller centrale di modificare tale custom resource senza preoccuparsi della funzione di rete e del controller che la andrà a gestire. Come si possono sapere in anticipo tutte le informazioni di cui potranno aver bisogno i controller delle network function? E se ci fosse la necessità di una nuova proprietà? Si potrebbe pensare di creare una nuova CRD che definisca un'altra interfaccia, ma così facendo il controller centrale non saprebbe come configurarla e questo richiederebbe la modifica del suo codice, cosa che si vuole evitare. Il codice del componente centrale non deve mai essere modificato perché tale componente deve essere abbastanza versatile da cambiare il suo comportamento solo in base alle impostazioni della custom resource associata, senza la necessità di ulteriori modifiche.

Un altro problema non risolto da questo approccio è quello di accettare input generici. Nella logica del componente centrale ci occupiamo di collegare i dati in ingresso con le funzioni di rete, ma se si decide di usare come input una sorgente di dati nuova che prima non veniva usata, come fa il controller a sapere come estrarre i dati necessari? Bisognerebbe creare un'ulteriore

interfaccia standard per prendere tutti i tipi di dato nello stesso modo, ma è difficile poter definire un'interfaccia che vada bene sempre, anche per sorgenti di dati che magari ancora non esistono e che verranno usate in futuro.

Inoltre, bisogna notare che in questo momento stiamo tenendo in considerazione solo le modifiche alle network function, ma non stiamo proprio considerando eventuali modifiche a risorse esterne al cluster. La generalizzazione si complicherebbe ancora di più. Per questo motivo e per quelli indicati prima si è scelto di abbandonare questa soluzione in favore di una più flessibile e dinamica basata su un'architettura event-driven.

Capitolo 4

Frameworks Knative e KRules

4.1 Knative

Knative estende Kubernetes fornendo un set di componenti middleware che sono essenziali per creare applicazioni moderne, source-centric e basate su container che possono essere eseguite ovunque: localmente, nel cloud o anche in data center di terze parti [14].

Ciascuno dei componenti nell'ambito del progetto Knative tenta di identificare modelli comuni e codificare le best practice condivise da framework e applicazioni di successo, reali e basati su Kubernetes. I componenti Knative si concentrano sulla risoluzione di compiti noiosi ma difficili come:

- deploying di un container;
- instradamento e gestione del traffico in un deployment blue/green;
- scalabilità automatica e dimensionamento dei carichi di lavoro in base alla richiesta;
- associazione dei servizi in esecuzione al sistema di eventing.

Knative è un progetto della community open source e facilita il deployment, l'esecuzione e la gestione di applicazioni serverless e cloud native. Il modello di cloud computing serverless può incrementare la produttività degli sviluppatori e ridurre i costi operativi. É un modello di sviluppo cloud native che consente agli sviluppatori di creare ed eseguire applicazioni senza gestire

i server. Dopo il deployment le app serverless rispondono alle richieste e si adattano automaticamente in base alle diverse esigenze di scalabilità.

Con un'architettura serverless le applicazioni vengono avviate solo quando necessario. Quando un evento attiva l'esecuzione del codice, vengono assegnate dinamicamente le risorse per tale codice e vengono usate solo fino alla fine dell'esecuzione, per poi essere liberate. Oltre ai vantaggi in termini di costo ed efficienza, il metodo serverless evita agli sviluppatori le attività di routine e manuali necessarie per garantire la scalabilità delle applicazioni e il provisioning del server.

Knative è composto principalmente da due componenti: Serving ed Eventing. Come si potrà notare per l'implementazione finale verrà usata solo la parte di Eventing, che è quella più interessante ai fini della tesi, ma per completezza analizziamo anche la componente di Serving.

4.1.1 Serving

Knative Serving consente il deployment rapido e la scalabilità automatica dei container tramite un modello basato sulla richiesta per mettere a disposizione i carichi di lavoro on demand. Knative Serving definisce una serie di oggetti come Kubernetes Custom Resource Definition (CRDs). Questi oggetti sono usati per definire e controllare come il carico di lavoro dei serverless si comporta nel cluster:

- Service: la risorsa service.serving.knative.dev gestisce automaticamente l'intero ciclo di vita del carico di lavoro. Controlla la creazione di altri oggetti per assicurarsi che l'app abbia una route, una configuration e una nuova revision per ogni aggiornamento del service. Il Service può essere settato per redirigere il traffico sempre all'ultima revision oppure ad una revision fissata.

- Route: la risorsa route.serving.knative.dev mappa un network endpoint a uno o più revision. Il traffico può essere gestito in diversi modi.

- Configuration: la risorsa configuration.serving.knative.dev mantiene lo stato desiderato per il deployment. Fornisce una chiara separazione tra il codice e la configurazione e segue la metodologia Twelve-Factor App. La modifica della configurazione porta alla creazione di una nuova revision.

- Revision: la risorsa revision.serving.knative.dev è uno snapshot del codice e delle configurazioni in un determinato momento. Le Revision sono

oggetti immutabili e possono essere consevati per tutto il tempo necessario. Le Knative Serving Revisions possono scalare automaticamente a seconda del traffico in arrivo.

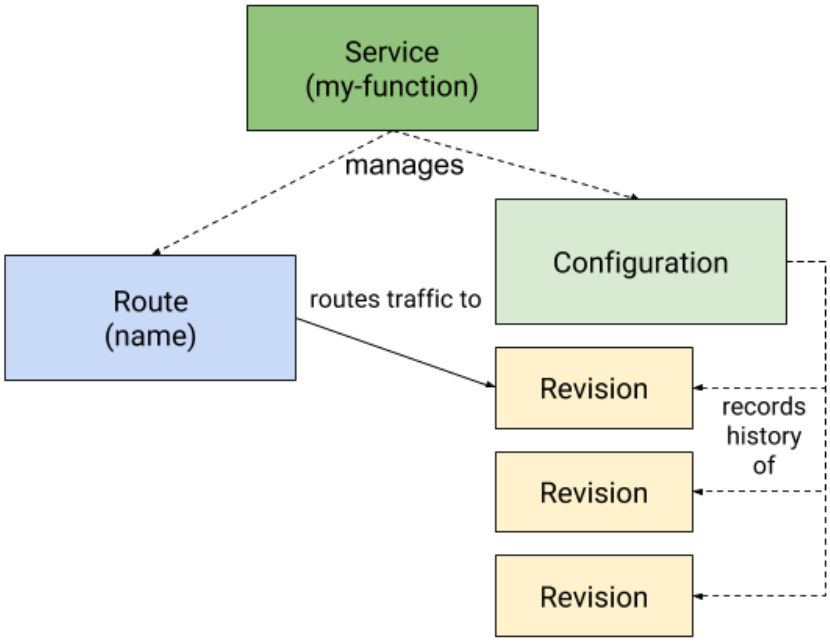

Figura 4.1. Knative Serving

4.1.2 Eventing

É un'infrastruttura che fornisce delle primitive per la creazione di applicazioni event-driven, garantendo il disaccoppiamento delle sorgenti e dei consumatori di eventi. L'eventing di Knative può essere utilizzato in diversi modi. I seguenti scenari sono tutti supportati dalle componenti esistenti. La modularità del sistema permette di combinare a piacimento tali componenti [15].

- si vuole solo pubblicare eventi senza sapere chi li consumerà: gli eventi vengono mandati al Broker come HTTP Post e dimenticati;
- si vuole solo consumare eventi senza sapere chi li ha prodotti: si usa un Trigger per sottoscriversi, sulla base di attributi CloudEvent, ad eventi da un Broker;

- si vogliono trasformare eventi attraverso una serie di steps: per creare topologie di trasmissione di messaggi complesse si possono usare le componenti Channels e Subscriptions.

L'eventing è stato progettato nel rispetto dei seguenti principi guida:

- I servizi di Knative Eventing sono scarsamente accoppiati. Questi servizi possono essere sviluppati e distribuiti in modo indipendente su una varietà di piattaforme (ad esempio Kubernetes, VM, SaaS o FaaS).
- I produttori di eventi e i consumatori di eventi sono indipendenti. Qualsiasi produttore (o sorgente) può generare eventi prima che vi siano consumatori di eventi attivi in ascolto. Qualsiasi consumatore di eventi può esprimere interesse per un evento o una classe di eventi, prima che ci siano produttori che creano quegli eventi.
- Altri servizi possono essere collegati al sistema di Eventing. Questi servizi possono svolgere le seguenti funzioni:
  - Creare nuove applicazioni senza modificare i produttori o i consumatori dell'evento.
  - Selezionare e scegliere sottoinsiemi specifici degli eventi dai loro produttori.
- Garantire l'interoperabilità tra i servizi. Knative Eventing è coerente con la specifica CloudEvents sviluppata dal CNCF Serverless WG.

Sources

Una sorgente di eventi è una Custom Resource (CR) di Kubernetes, creata da uno sviluppatore o da un amministratore di cluster, che funge da collegamento tra un produttore di eventi e un event sink. Un sink è il primo componente che riceve l'evento generato da una sorgente. Il sink può essere un servizio Kubernetes, incluso un Knative Services, un Channel o un Broker.

Le event sources vengono create istanziando un CR da un oggetto Source. L'oggetto Source definisce gli argomenti e i parametri necessari per istanziare un CR. Knative mette a disposizione alcune sorgenti di eventi integrate all'interno del sistema di eventing. Alcune di queste sono:

- APIServerSource: permette di intercettare gli eventi di Kubernetes e inoltrarli al broker per l'utilizzo come eventi Knative. L'APIServerSource lancia un nuovo evento ogni volta che una risorsa viene creata, modificata o cancellata.

- Apache Camel: abilita l'uso delle componenti di Apache Camel per inviare eventi a Knative. Un CamelSource è una sorgente che può rappresentare qualunque componente Apache Camel.

- PingSource: produce eventi con un payload fisso sulla base di uno schedulatore Cron.

Quelli riportati sopra sono solo alcuni esempi ma altre sorgenti sono disponibili. Nello specifico la sorgente dati che useremo nell'implementazione del caso d'uso è APIServerSource. Oltre alle sorgenti che vengono fornite da Knative ci sono anche quelle di terze parti supportate dalla community e mantenute nella repo GitHub Knative Eventing-Contrib. Inoltre è anche possibile scrivere una nuova sorgente di eventi usando le guide fornite da Knative.

Le sorgenti di eventi sono perfette per rappresentare gli input generici definiti dall'obiettivo della tesi. Le caratteristiche che deve avere un input generico sono:

- numero di input non fisso: grazie all'eventing di Knative sorgenti e consumatori sono indipendenti e si possono istanziare tutte le sorgenti di cui si necessita. É inoltre possibile aggiungere o togliere sorgenti in un qualunque momento senza alcuna ripercussione sul funzionamento dell'applicazione;

- sorgente del dato generica: la possibilità di usare diversi tipi di sorgenti, come quelle elencate prima, garantisce la generalizzazione necessaria. Ma non solo, perché se si vuole usare una sorgente che non è presente tra quelle fornite da Knative o di terze parti, è possibile crearla su misura;

- il tipo di dato generico: usando delle sorgenti generiche, anche le tipologie di dato che analizziamo sono generiche.

Consumers

I consumers sono quei componenti dell'architettura ad eventi che si occupano di elaborare gli eventi. Per permettere a diversi tipi di servizi di ricevere eventi, Knative Eventing ha definito due requisiti generici che devono rispettare i consumer e che possono essere implementate da più risorse Kubernetes:

- Addressable: devono essere in grado di ricevere e riconoscere un evento consegnato su HTTP a un indirizzo definito nel campo *status.address.url*.

- Callable: devono essere in grado di ricevere un evento consegnato su HTTP e trasformare l'evento, restituendo 0 o 1 nuovi eventi nella risposta HTTP.

I consumers generalmente ricevono solo alcuni eventi, quelli che sono rilevanti per la loro funzione. L'eventing di Knative mette a disposizione due oggetti che facilitano il filtraggio degli eventi in base agli attributi: il Broker e i Trigger.

Un Broker fornisce un bucket di eventi che possono essere selezionati per attributo. Riceve gli eventi e li inoltra ai sottoscrittori definiti da uno o più Trigger corrispondenti. Poiché un Broker è un Addressable, le sorgenti possono inviare eventi al Broker inviando l'evento all'indirizzo status.address.url del Broker.

Un Trigger descrive un filtro sugli attributi dell'evento che dovrebbe essere consegnato a un Addressable. In altre parole rappresenta il desiderio di sottoscriversi ad uno specifico tipo di evento per un ben definito Broker. Non c'è un limite al numero di Trigger che si possono creare.

Il codice di seguito rappresenta un Trigger di nome *my-service-trigger* che riceverà tutti gli eventi dal Broker *default* e li inoltrerà al service Knative *my-service*.

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: my-service-trigger
spec:
  broker: default
  subscriber:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: my-service
```

Per la maggior parte dei casi d'uso, è sufficiente un singolo bucket (Broker) per namespace, ma esistono casi d'uso in cui più bucket (Broker) possono semplificare l'architettura. Ad esempio, broker separati per eventi contenenti Personally Identifiable Information (PII) ed eventi non-PII possono semplificare l'audit e le regole di controllo degli accessi.

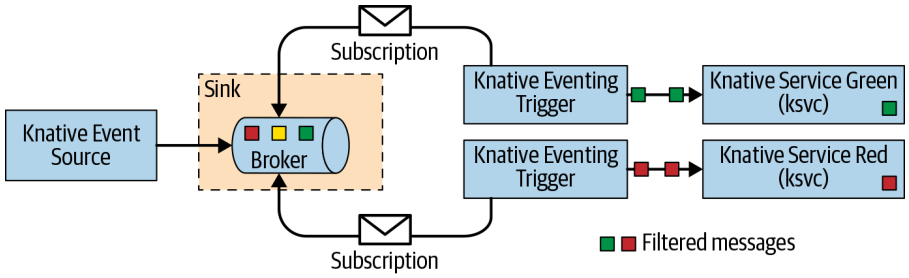

Figura 4.2. Broker e Trigger

4.2 KRules

Nel capitolo precedente si è visto come la componente Eventing di Knative permette di definire sorgenti e consumatori di eventi che possono essere considerati rispettivamente come gli input e gli output del sistema generico oggetto di questa tesi. Purtroppo Knative da solo non basta a rispettare i vincoli imposti dall'obiettivo. E' importante che il sistema generico possa trattare gli input, e definire gli output, con una logica di business non nota a priori e programmabile dall'esterno. Questo non viene trattato da Knative che si occupa solo di fornire un'infrastruttura per la gestione degli eventi. Creare, ogni volta che cambia il caso d'uso, delle componenti specifiche che gestiscono gli eventi renderebbe il sistema poco flessibile, senza contare lo sforzo che richiederebbe programmare tali componenti ogni volta in maniera diversa. Ciò che invece viene richiesto dall'obiettivo è che il sistema sia in grado di modificare il suo comportamento, a seconda del caso d'uso, in maniera semplice e dichiarativa.

In questo capitolo verrà analizzato nel dettaglio il framework KRules, il quale risponde perfettamente alle richieste dell'obiettivo, mettendo a disposizione un metodo dichiarativo per definire la logica di business.

4.2.1 Principi Generali

KRules è un framework open source, sviluppato da Airspot Cloud Native Development, che fornisce, agli sviluppatori Python, un modo veloce e flessibile per sviluppare applicazioni cloud native mediante la creazione di microservizi reattivi e basati sugli eventi in un cluster Kubernetes [16]. KRules adotta un approccio basato sul paradigma evento-condizione-azione mediante l'utilizzo di regole. KRules trae appieno i vantaggi offerti da Kubernetes e Knative e si ispira ai pattern definiti da "The Reactive Manifesto":

- reattivo: il sistema risponde in maniera tempestiva, se possibile;
- resiliente: a fronte di errori il sistema rimane reattivo;
- elastico: il sistema rimane reattivo al variare del carico di lavoro;
- guidato dagli eventi: il sistema si basa sull'invio di eventi asincroni;

Queste caratteristiche lo rendono la soluzione ideale per applicazioni cloud native ed event driven come quella che questo lavoro di tesi si pone di trovare. KRules si integra alla perfezione con Knative, sfruttando a pieno le potenzialità sia della componente serving che eventing. É importante però capire la differenza tra i due: mentre Knative fornisce un'infrastruttura serverless per la gestione degli eventi, KRules lavora a livello più alto, potenziando la creazione di una logica applicativa definita attraverso un set di regole sotto forma di strutture dati Python.
In altre parole KRules è un framework che ha due compiti essenziali:

- memorizzare e gestire i subject;
- interpretare le rules.

4.2.2 Subjects

Uno dei concetti più importanti che sta alla base del paradigma di programmazione KRules è il subject. Ogni volta che viene prodotto un tipo di evento, è sempre possibile ricondurlo a un tipo di entità, che lo ha prodotto o che in qualche modo è correlato ad esso. In KRules il subject è la rappresentazione astratta attorno al quale viene modellata la logica.

KRules fornisce la possibilità di tener traccia dello stato relativo ai subject e reagire al suo cambiamento. Lo stato di un subject viene definito attraverso le reactive properties. Nel momento in cui viene assegnato un valore alle proprietà il subject incomincia ad esistere. Dietro le quinte c'è un componente chiamato Subject Property Store che oltre alla reattività fornisce anche tutte le strutture per lavorare in modo efficiente in un sistema altamente concorrente. L'assegnazione di un nuovo valore a una proprietà reattiva del subject produce un evento che porta con se sia il nuovo valore che quello vecchio. Ciò consente al sistema di reagire non solo a un nuovo stato, ma più ampiamente a ciascuna transizione di stato.

Il soggetto infatti potrebbe essere inteso come un digital twin, ad esempio di un dispositivo che invia aggiornamenti sulla sua temperatura interna,

un veicolo che periodicamente condivide la sua posizione, un utente che interagisce con un sito web, un documento o anche un servizio Kubernetes, essendo Kubernetes stesso è un produttore di eventi. In breve, tutto ciò che nel dominio dell'applicazione potrebbe produrre eventi e potrebbe avere uno stato è potenzialmente un soggetto.

Oltre alle reactive properties vengono definite anche le extended properties. Queste sono più simili ai metadata di un subject. Le proprietà estese sono comprese dall'infrastruttura di eventing Knative, quindi sono destinate a definire la logica a livello di trasporto, ad esempio per dirigere tutti gli eventi relativi a un sottoinsieme di subject della stessa classe a un broker specifico, attivando un gruppo isolato di microservizi nello stesso cluster o anche in uno diverso.

Per capire meglio la differenza tra le due proprietà facciamo un esempio pratico. Si supponga di avere un subject di nome "foo" e di assegnare a questo una proprietà reattiva di nome "moo" con il valore settato ad 1. Come è stato già anticipato, l'assegnazione della proprietà genera un evento che può essere intercettato e processato da una o più rules. Tale evento sarà così strutturato:

```
Validation: valid
Context Attributes,
  specversion: 1.0
  type: subject-property-changed
  source: my-ruleset
  subject: foo
  id: bd198e83-9d7e-4e93-a9ae-aa21a40383c6
  time: 2020-06-16T08:16:57.340692Z
  datacontenttype: application/json
Extensions,
  knativearrivaltime: 2020-06-16T08:16:57.346535873Z
  knativehistory: default-kne-trigger-kn-channel.iot-demo-gcp-01.svc.cluster.local
  originid: bd198e83-9d7e-4e93-a9ae-aa21a40383c6
  propertyname: moo
  traceparent: 00-d571530282362927f824bae826e1fa36-a52fceb915060653-00
Data,
  {
    "property_name": "moo",
    "old_value": null,
    "value": 1
  }
```

Quella sopra è una comoda visualizzazione dell'evento nel formato CloudEvent. L'evento è costituito da un insieme di attributi definiti come parte integrante dello standard e da eventuali attributi estesi definiti in base alle specifiche esigenze dell'applicazione.

Una delle prime cose da notare è il type dell'evento. Ogni volta che si assegna o si modifica una proprietà reattiva di un subject, viene generato un evento correlato al subject e con il type settato a "subject-property-changed". In questo modo è possibile all'interno di un trigger di Knative filtrare gli eventi in base al tipo "subject-property-changed".

Come è possibile osservare, l'evento contiene anche un'estensione chiamata propertyname che indica il nome della proprietà modificata. Attraverso questo attributo siamo così in grado di iscriverci a tutti gli eventi relativi ad una variazione di questa proprietà. propertyname altro non è che una extended property e in quanto tale è possibile usarla come filtro all'interno di un trigger di Knative per intercettare gli eventi con questa proprietà. Come vedremo in seguito, è possibile creare attributi aggiuntivi (come estensioni) per dare una classificazione più specifica dell'evento, o più propriamente, del relativo soggetto.

Da notare che il nome della proprietà modificata (moo) viene ripetuto sia nel payload che come proprietà estesa. Questo accade perché a livello di trasporto e indirizzamento dell'evento il contenuto del messaggio non viene preso in considerazione; il payload viene invece utilizzato dal servizio che consuma l'evento (nel nostro caso un ruleset).

Osservando il contenuto del payload si può vedere come l'evento contiene, oltre al nome della proprietà e al nuovo valore di essa, anche il vecchio valore della proprietà. Questo è molto utile quando si vuole implementare una logica stabilita non solo sul nuovo valore della proprietà ma anche sulle sue transizioni.

Memorizzazione dello stato

Come abbiamo appena visto il subject ha uno stato, e per questo ha bisogno di una soluzione di memorizzazione che gli dia persistenza e in alcuni casi gestisca la concorrenza in maniera appropriata. Tale soluzione dovrà essenzialmente memorizzare le proprietà, gestire gli accessi e verrà usata dall'interfaccia di alto livello del subject che è responsabile di gestire il caching e la reattività.

Proprio a causa delle differenti necessità che ci possono essere, in KRules ci sono diversi modi per implementare tale componente:

- implementazione basata su Redis: usato principalmente quando bisogna gestire tanti accessi concorrenti;

- implementazione basata su Mongodb: usata quando si necessita di un'alta scalabilità e il numero di subject è indefinito ma si sa che potenzialmente potrebbero essere tanti;

- implementazione Custom: questa implementazione consente di far riferimento in modo specifico alle risorse di Kubernetes rendendole automaticamente il contesto della regola che risponde all'evento relativo (aggiungi/aggiorna/elimina). Utilizza la stessa soluzione di archiviazione utilizzata da Kubernetes (di solito etcd), quindi non sono necessarie ulteriori dipendenze. Inoltre, ottimizza e riduce il numero di chiamate al server API Kubernetes. Gestisce in modo trasparente specifici attributi delle risorse (come namespace, kind e apigroup, ecc.) rendendoli disponibili come proprietà estese del subject e quindi come metadati dell'evento relativo. Ciò consente di utilizzarli a livello di trasporto e nello specifico nei trigger di knative per migliorare notevolmente la granularità della specializzazione dei micorservizi e quindi la resilienza generale del sistema.

La soluzione di memorizzazione Custom, a differenza delle altre, può essere attivata semplicemente impostando il subject di un evento in questo modo: "k8s:<resource_api_path>". Quindi quando il subject è nella forma appena vista, KRules automaticamente memorizzerà il suo stato nella corrispondente risorsa Kubernetes. Tale funzionamento è realizzato dal seguente pezzo di codice che fa parte del container di base.

```
subject_storage_factory.override(
    providers.Factory(lambda name, event_info, event_data:
                          name.startswith("k8s:")
                          and
                          k8s_storage_impl.SubjectsK8sStorage(
                              resource_path=name[4:],
                              resource_body=event_data
                          )
                          or
                          redis_storage_impl.SubjectsRedisStorage(
                              name,
                              subjects_redis_storage_settings.get("url")
                          )
```

```
                    )
)
```

Questa istruzione fornisce inoltre al subject il parametro *event_data* (se disponibile) che corrisponde alla definizione della risorsa se le regole rispondono a un evento k8s (altrimenti viene richiesto al server api). KRules fornisce anche un'implementazione alternativa (redis) da utilizzare con tipologie diverse di subject.

4.2.3 Rules

Le rules sono le regole concrete che permettono di definire la logica applicativa ad alto livello. Le rules sono raggruppate in rulesets. I rulesets sono i microservizi che vengono distribuiti sul cluster e rispondono in modo indipendente a specifici tipi di eventi e attributi grazie ai trigger di Knative. Non ci sono vincoli particolari sulla creazione di questi trigger o su quali eventi devono essere ricevuti dai rulesets. Più granulare sarà la definizione dei trigger e dei rulesets corrispondenti, più resiliente sarà il sistema risultante poiché ogni servizio, o meglio, ogni rulesets, è scalabile indipendentemente dagli altri. All'interno del ruleset possiamo avere più rules ognuna delle quali può iscriversi a tipi di eventi diversi (se catturati dai trigger) e con diversi criteri di attivazione in base al payload ricevuto. Ogni regola, è sempre contestualizzata a un subject.

Di seguito vediamo come è fatta una regola:

```
{
    rulesname: "name-of-the-rule",
    subscribe_to: [ "type1", "type2",... ],
    ruledata:{
        filters:[
            filter1(),
            filter2(),
            ...
        ],
        processing: [
            function1(),
            function2(),
            ...
        ],
    },
}
```

Una rules è una struttura dati in python con i seguenti attributi:

- rulesname: è costituito da una stringa che indica il nome della regola;
- subscribe_to: a questo attributo viene passato un vettore di stringe. Ogni stringa indica l'evento a cui reagisce la regola e corrisponde all'attributo type del cloudevent. Quindi la regola può reagire a più eventi con diversi type;
- ruledata: questo attributo è costituito da un dizionario con due attributi:
  - filters: lista di funzioni che hanno l'obiettivo di filtrare gli eventi;
  - processing: lista di funzioni eseguite come conseguenza dei filters.

4.2.4 Filters

Quando una regola riceve un evento e il suo type rientra tra quelli indicati nella sezione subscribe_to, come prima cosa vengono eseguite in maniera sequenziale le funzioni nella sezione filters.

Queste funzioni hanno lo scopo di filtrare ulteriormente gli eventi dando la possibilità di rendere le regole quanto più granulari possibile e facilitando la riusabilità del codice. Le funzioni devono ritornare true o false. Se una funzione ritorna true allora si passa alla prossima o si va nella sezione di processing, altrimenti la regola si ferma.

KRules mette a disposizione alcune funzioni generali di filtraggio che permettono di lavorare con gli eventi, ma se si ha bisogno di una funzione particolare è sempre possibile crearla su misura e richiamarla in questa sezione. Le funzioni che mette a disposizione KRules (nella versione v0.6) sono:

- Filter: valuta l'espressione booleana passata come argomento della funzione e ritorna il suo valore.
- SubjectNameMatch: ritorna true se il nome del subject coincide con l'espressione regolare passata come argomento della funzione.
- SubjectNameDoesNotMatch: ritorna true se il nome del subject non coincide con l'espressione regolare passata come argomento della funzione.
- CheckSubjectProperty: ritorna True se la proprietà del subject passata come primo argomento esiste e se coincide con il valore passato come secondo argomento. Il secondo valore è opzionale.

- PayloadMatch: confronta la jsonpath expression passata come primo argomento con il payload dell'evento.

- SubjectPropertyChanged: funzione specifica che filtra gli eventi di tipo subject-property-changed. Richiede tre parametri: nome della proprietà modificata, valore nuovo e valore vecchio.

4.2.5 Processing

Se tutte le funzioni della sezione di filters vengono superate allora verranno eseguite in maniera sequenziale le funzioni della sezione di processing. Queste funzioni sono quelle che definiscono il nucleo di elaborazione della Rule. Mediante l'utilizzo di queste funzioni possiamo fare pressoché qualunque cosa: interagire con l'api server di Kubernetes, creare/modificare/eliminare le risorse, propagare l'evento ricevuto rimandandolo indietro o su un altro Broker, interagire con componenti esterne al cluster. Quelli citati sono solo esempi, ma le possibilità sono molte.

Come per la sezione filters anche per la sezione processing KRules mette a disposizione una serie di funzioni di base che svolgono alcune operazioni generiche. Essendo KRules un framework non può fornire tutte le funzioni di processing possibili, ma mette a disposizione l'infrastruttura di base per la gestione di tali funzioni e ne fornisce alcune molto generiche. Per questo motivo è possibile estendere KRules creando delle funzioni custom a seconda del caso d'uso.

Il framework è ancora giovane (nel momento in cui viene scritta la tesi è alla versione v0.6) e quindi in continua evoluzione. Molte funzioni di base verranno aggiunte nelle versioni successive, ma alcune delle funzione di base fornite in questa versione sono:

- Route: produce un evento dentro o fuori il ruleset a seconda del valore di dispatch_policy. Mandare l'evento fuori dal ruleset significa mandarlo al dispatcher. I parametri che si aspetta sono:

  - type: il tipo dell'evento. Se non indicato viene usato il type dell'evento corrente che ha scatenato la regola.
  - subject: il nuovo subject o quello dell'evento corrente se non espressamente indicato.
  - payload: il nuovo payload o quello dell'evento corrente se non espressamente indicato.

  - dispatch_policy: può essere uno tra: DEFAULT, NEVER, DIRECT, ALWAYS. DEFAULT invia l'evento fuori solo quando non ci sono regole che lo gestiscono nella ruleset corrente. NEVER non invia mai l'evento fuori. DIRECT invia l'evento direttamente fuori senza controllare se nel ruleset corrente ci sono regole che possono gestire l'evento. ALWAYS anche se nel ruleset ci sono regole che processano l'evento, quest'ultimo viene comunque inviato anche fuori.

- SetSubjectProperty: setta una singola proprietà del subject. I parametri che si aspetta sono:
  - property_name: nome della proprietà da settare.
  - value: valore da assegnare alla proprietà.
  - extended: setta una proprietà estesa se True, altrimenti una proprietà standard (reattiva).
  - muted: se impostato a True non viene lanciato nessun evento subject-property-changed dopo che la proprietà viene settata.
  - use_cache: se impostato a False memorizza il valore immediatamente sullo storage, altrimenti attende la fine dell'esecuzione della regola.
- SetSubjectProperties: permette di settare più proprietà contemporaneamente nel subject da un dizionario passato come primo parametro. Questo è permesso solo usando la cache e non può essere fatto per le proprietà estese.
- StoreSubject: cancella la cache andando a scrivere il subject in memoria. Di solito questo viene fatto sempre alla fine dell'esecuzione di una regola.
- FlushSubject: rimuove il subject e tutte le sue proprietà dalla memoria.
- UpdatePayload: modifica il payload fondendolo con il dizionario passato come parametro.
- SetPayloadProperties: setta le proprietà passate come argomenti nel payload e se qualcuna esiste già viene sovrascritta.

Capitolo 5

Architettura sistema generico

5.1 Descrizione

Dopo aver trattato Knative e KRules, diventa facile comprendere come questi due framework comunicando tra loro soddisfano le richieste poste dall'obiettivo della tesi e quindi si è deciso di adottarli nell'architettura di riferimento implementata. É necessario precisare che l'operatore di cui si parla nell'obiettivo non fa più riferimento all'operatore vero e proprio di Kubernetes come lo abbiamo descritto nel paragrafo 2.3.1, ma va inteso come componente o sistema generico che estende il funzionamento di Kubernetes. Come abbiamo visto nel capitolo precedente, il primo vincolo, definito dall'obiettivo della tesi, viene ampiamente soddisfatto dal concetto di sorgente in un'architettura event-driven e in particolare dalla vastità di sources offerte da Knative. Queste sorgenti di eventi possono essere di qualsiasi tipo, e gli eventi possono portare con se informazioni prese da qualsiasi parte: proprietà di una risorsa, corpo di una richiesta http, metriche di prometheus, informazioni memorizzate su un documento o anche nell'etcd. Le sorgenti degli eventi sono completamente scollegate dai consumer o dai broker, offrendo quindi la possibilità di aggiungere o rimuovere sorgenti dati in qualunque momento. Quando aggiungiamo una sorgente di eventi se c'è un consumatore in ascolto di quegli eventi li catturerà ed inizierà a funzionare. Se allo stesso modo ad un certo momento togliamo la sorgente di eventi che vengono usati da un consumer, quest'ultimo semplicemente smetterà di eseguire l'elaborazione relativa a quell'evento, senza generare alcun tipo di errore o eccezione dovuto

alla mancanza di dati. Questo rende il sistema versatile e le sue componenti debolmente accoppiate.

In modo altrettanto flessibile, Knative, grazie all'infrastruttura che mette a disposizione, permette di definire dei consumer in grado di ricevere gli eventi prodotti dai sources ed elaborarli. Questa elaborazione rappresenta l'output dell'operatore generico, in quanto è possibile eseguire qualunque operazione con questi consumer. Si può pensare di creare consumer per apportare modifiche all'interno del cluster Kubernetes, come ad esempio creare/modificare/distruggere delle VNF o modificare delle configMap, oppure è anche possibile interagire con servizi all'esterno del cluster. Questi consumer come già anticipato devono rispettare dei requisiti (Addressable e Callable) per essere definiti tali. É importante però capire che la funzione di consumer può essere realizzata da qualunque risorsa/componente sia in grado di rispettare i vincoli, mentre Knative da la possibilità a tali consumatori di ricevere e di sottoscriversi agli eventi.

Infine il vincolo sulla gestione dei dati mediante una logica di business non nota a priori viene soddisfatto da KRules. Mediante l'utilizzo dei rulesets, e quindi delle rules, possiamo definire in maniera semplice la logica che il sistema metterà in atto a fronte del verificarsi di determinati eventi. É possibile indicare gli eventi di interesse per una regola, definire degli ulteriori filtri per questi eventi e infine è possibile indicare una serie di funzioni che attueranno le operazioni necessarie per reagire al verificarsi degli eventi stessi. Queste funzioni altro non sono che codice python eseguito all'interno di un microservizio nel cluster Kubernetes, quindi in quanto tale è in grado di fare qualunque cosa con le giuste librerie e/o API. L'adozione del codice Python rende estremamente facile l'implementazione della logica di reazione abbassando notevolmente il livello di skill di cui un operatore di rete si deve dotare.

Questi microservizi possono assumere la forma di consumers finali, cioè che mettono in pratica la logica di reazione all'evento, oppure componenti intermedi che generano condizioni sotto forma di eventi complessi a cui si possono sottoscrivere altri consumers. Infatti un ruleset può semplicemente ricevere un evento, modificarlo e poi inoltrarlo nuovamente al Broker comportandosi come un componente intermedio, oppure essere lui stesso il nodo finale, comportarsi come un consumer e quindi essere l'output del sistema. Questo doppio funzionamento può essere molto utile in quanto consente la definizione di eventi astratti come composizione di eventi elementari. Si supponga di volere come output del sistema la modifica della configurazione di

una VNF come conseguenza di un'anomalia nelle metriche raccolte da Prometheus. Questa modifica dipende dalla funzione di rete, da come è fatta e da come legge la configurazione.
Se ad esempio si possiede già un'orchestratore specifico per quella risorsa di rete (ad esempio ONAP), che si occupa di effettuare tale modifica, non deve essere il ruleset a farlo. L'unico compito del ruleset sarà quello di definire la logica che si occuperà di intercettare gli eventi contenenti le metriche di Prometheus, analizzarli, capire se si è verificata o meno l'anomalia e generare un altro evento che servirà ad attivare la risorsa specifica per la modifica della configurazione. Nell'esempio appena visto il ruleset si comporta come un attore intermedio il cui compito è quello di realizzare solo la logica di business che permette di collegare input e generare eventi complessi a cui orchestratori specifici si agganceranno.
Adesso si pensi allo stesso esempio di prima, ma con la differenza di non aver più a disposizione un'orchestratore specifico per la risorsa di rete. In questa eventualità è possibile far diventare il ruleset il consumer finale, il quale, dopo aver analizzato i dati, si occupa di modificare la VNF. KRules è solo un framework e non può mettere a disposizione funzioni per modificare tutte le VNF possibili, però come abbiamo visto è possibile estendere il framework creando delle funzioni custom che risolvono la specifica implementazione. Quindi nell'esempio di prima avremmo dovuto creare una funzione custom che sarebbe stata chiamata nella sezione di processing della regola alla fine dell'analisi delle metriche per mettere in campo la reazione che si rende necessaria nella condizione evidenziata dagli eventi (la modifica di una VNF).

A runtime ogni ruleset è realizzato tramite microservizi e la possibilità delle regole di essere utilizzate in diversi modi permette a chi le usa di creare più microservizi collegati tra loro che realizzano logiche anche più complesse di quella appena vista. Creare più rulesets permette di aumentare la granularità delle singole regole, rendere il codice più resiliente e riutilizzabile. Se nell'esempio visto prima, invece di creare un solo ruleset che si occupa di recuperare i dati, valutare le metriche ed attuare le modifiche, si creano due rulesets (uno che si occupa di recuperare i dati, analizzare le metriche e lanciare un nuovo evento con i risultati dell'analisi e l'altro che invece si occupa di ricevere l'evento con i risultati ed effettuare la modifica alla VNF) si ottengono i due seguenti vantaggi:

- riusabilità del codice: la regola che modifica la configurazione del servizio di rete è una regola molto specifica, e anche se per realizzarla abbiamo

dovuto creare una funzione custom, questa potrà essere riusata in tutti i casi in cui sarà presente tale servizio;

- ulteriore disaccoppiamento: se ad esempio decidiamo di cambiare il calcolo dell'anomalia nelle metriche, oppure è cambiato il tipo di anomalia, ci basta cancellare il microservizio che implementa questa analisi e riscriverlo, ma il ruleset che implementa la riconfigurazione della VNF non viene toccato.

É chiaro quindi che le possibilità sono tante e grazie a questi due framework è possibile creare operatori più o meno complessi e specifici in base al caso d'uso.

Lo schema rappresentato nella figura 5.1 permette di comprendere meglio come Knative e KRules si integrano con Kubernetes e come interagiscono tra loro e con le risorse sia del cluster che esterne al cluster.

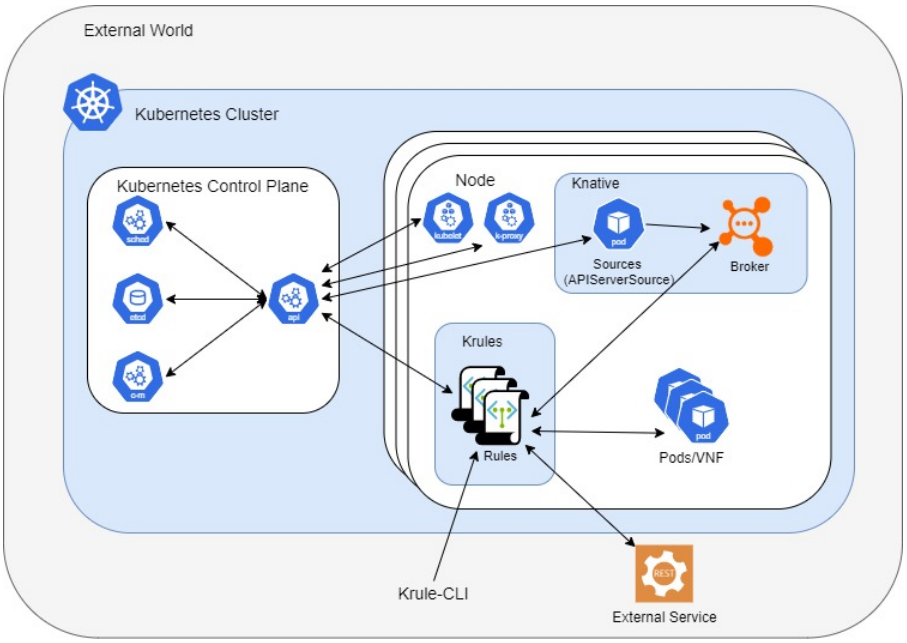

Figura 5.1. Schema architettura sistema generico

5.2 Utilizzo

Per creare un'implementazione del sistema generico e quindi iniziare a lavorare con eventi e regole è necessario installare Knative e KRules.

I due framework sono basati su Kubernetes, quindi per iniziare ad usarli per prima cosa bisogna avere un cluster Kubernetes. Successivamente è necessario installare prima Knative e solo dopo KRules, perchè Knative offre l'infrastruttura sulla quale verranno create le regole.

Come già anticipato Knative è costituito da due componenti: Serving e Eventing. Mentre la componente di Eventing è necessaria per il funzionamento del sistema, quella di Serving può essere installata o meno a seconda delle esigenze. In realtà il suo scopo è quello di semplificare e velocizzare il deployment dei container e la scalabilità automatica di essi dando la possibilità di scalare anche fino a 0 repliche, non consumando di fatto risorse quando non ci sono richieste per il container. Questi vantaggi però non compensano l'overhead che tale componente aggiunge. L'installazione del Serving porta con se diverse risorse che appesantiscono il cluster. Inoltre bisogna scegliere il networking layer che useranno i servizi e Knative mette a disposizione diverse scelte. Tra queste scelte la più usata è Istio, il quale però appesantisce di molto i singoli servizi in quanto aggiunge un sidecar per ogni Pod. Quindi è consigliato non usare questa componente e lasciare il sistema più leggero, ma se per qualche ragione dovesse servire, KRules permette la creazione dei rulesets sia mediante il Serving di Knative che senza.

Dopo aver installato l'Eventing è possibile installare KRules, il quale mette a disposizione una CLI che facilita e velocizza la creazione di un progetto basato su KRules e organizza i microservizi creati in una struttura gerarchica [17]. La CLI con un solo comando permette di creare un ruleset vuoto. A questo punto dopo aver scritto il codice del ruleset, grazie alla CLI è possibile creare in modo altrettanto semplice un trigger. In questo modo gli eventi che rispettano il filtro del trigger verranno inviati al ruleset appena creato. Infine l'interfaccia da linea di comando permette anche il deploy del ruleset e del trigger su Kubernetes.

Capitolo 6

Implementazione caso d'uso

Dopo aver analizzato i vari framework e visto come interagiscono tra loro, si è passati all'implementazione di un sistema concreto su un caso d'uso reale che rendesse l'idea delle potenzialità della soluzione trovata. La rete di telecomunicazioni in analisi è composta da una catena di Virtual Network Function collegate tra loro mediante l'utilizzo della CNI Multus. Nello specifico le funzioni di rete virtuali sono:

- un Web Server che espone un sito web raggiungibile dall'interno del cluster. Il web server è stato realizzato usando node.js e per semplicità accetta solo richieste GET che ritornando come risposta "Hello World";
- un Firewall che precede il web server nella catena di servizi e che può scalare a seconda del traffico o di altri parametri che non sono importanti per l'implementazione del caso d'uso. Il Firewall è stato realizzato usando un semplice codice C che intercetta e filtra il traffico in base agli indirizzi scritti in una variabile d'ambiente chiamata DENYLIST;
- un Load Balancer situato prima del Firewall e il cui compito è quello di bilanciare il traffico sulle istanze di Firewall il cui numero può anche cambiare a runtime. Per semplicità implementativa il LB è stato realizzato usando una semplice immagine linux e configurando un Multipath Routing (MR) per mezzo delle routing table. Con il multipath routing è possibile distribuire il traffico destinato a una sola rete attraverso diversi percorsi (rotte). Questa è un'estensione del concetto convenzionale di routing table, in cui ci può essere una sola associazione network-next

hop. Invece con il MR è possibile specificare più next hop per una sola destinazione. In questo modo il multipath routing può essere usato per bilanciare il flusso a livello 3. Il modo in cui il flusso viene direzionato quando c'è più di un percorso da scegliere dipende dalla configurazione di rete e dall'implementazione del kernel [18].

La sorgente del traffico che attraversa la catena di Virtual Network Function appena descritta è un semplice Client, realizzato mediante un'immagine linux, il cui unico compito è quello di far partire delle richieste HTTP verso il Web Server, che al contrario è la fine della catena.

É importante tenere a mente che l'implementazione del Multipath Routing in linux è pensata per distribuire il flusso di pacchetti su diversi path, non il singolo pacchetto. Per associare un pacchetto ad un flusso, lo stack di rete calcola un hash su un sottoinsieme dei campi dell'header del pacchetto (di solito porta e ip sorgente, porta e ip destinazione). Per questa ragione sono stati usati due istanze della risorsa Client, in modo da suddividere i flussi sulle diverse istanze di Firewall.

Ricapitolando, la struttura della catena è quella mostrata in figura 6.1, in cui Multus ha configurato due interfacce di rete per il LoadBalancer e il FireWall in modo tale da collegarli alle risorse precedenti e successive, e un'interfaccia di rete per Client e Web Server, creando in questo modo la catena di servizi mostrati in figura.

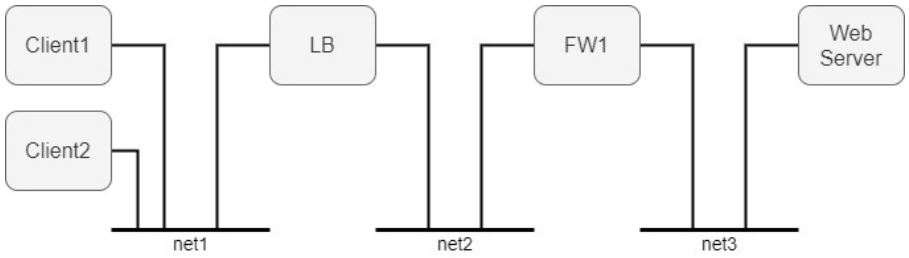

Figura 6.1. Configurazione iniziale della rete

A questo punto per mostrare le reali potenzialità della soluzione trovata si è scelto di prendere in considerazione due eventi caratterizzanti. Uno che dimostri la possibilità del sistema di reagire ad eventi provenienti dall'interno del cluster e uno invece che dimostri la capacità del sistema di interagire con il mondo esterno al cluster.

L'evento interno è l'aumento del numero di repliche del Firewall. Non è importante il motivo per cui il FW scala e aumenta il numero di repliche,

ma la cosa che ci interessa è il fatto che ci siano nuove istanze (nel caso di esempio lo scaling del firewall potrebbe essere una decisione autonoma attuata da kubernetes). Nel momento in cui verrà creata la nuova istanza Multus si preoccuperà di configurare le interfacce di rete sulle due reti net2 e net3. Supponendo che venga creata solo un'altra istanza di Firewall, in aggiunta a quella iniziale, a questo punto la struttura della catena in esame diventa quella rappresentata nella figura 6.2.

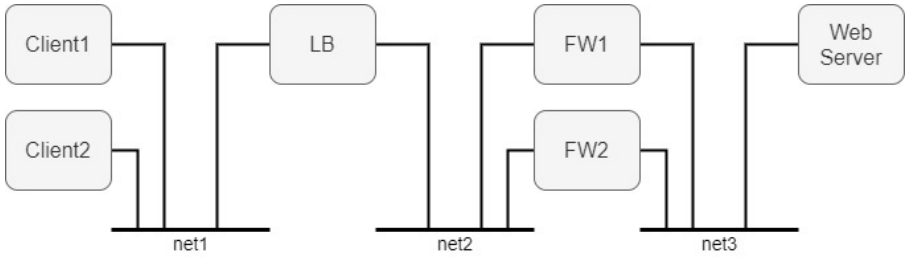

Figura 6.2. Configurazione finale della rete

L'aggiunta di una nuova istanza di firewall da parte di Kubernetes non basta ad abilitare il bilanciamento del traffico sul nuovo Firewall, infatti è necessario riconfigurare il LB in modo che venga a conoscenza della sua esistenza. Prima di configurare il LB, però, si vuole simulare la conferma da parte di un operatore di rete (o un tecnico di rete), che venuto a conoscenza dello scaling del Firewall decide se è opportuno bilanciare o meno il traffico sulle nuove repliche. Per simulare questa conferma si è scelto di usare un servizio esterno al cluster. La scelta è ricaduta su Slack perché grazie alle API che mette a disposizione è molto facile e veloce implementare richieste da e verso tale servizio. Dopo la creazione di FW2 il sistema manda una notifica su Slack all'operatore che si occuperà di confermare o rifiutare il bilanciamento del traffico sulla nuova istanza. Questa scelta fatta dall'operatore per noi è il secondo evento (quello esterno) che il sistema deve essere in grado di catturare ed elaborare. Questo evento non ha nulla a che vedere con il cluster e Kubernetes, ma è importante poterlo usare per poi riconfigurare il LB sulla base della risposta proveniente da Slack.

Le API di Slack mettono a disposizione un formato particolare di messaggi che permettono all'utente di interagire. Grazie a questi messaggi chiamati interattivi l'operatore potrà esprimere la sua preferenza mediante l'uso di due pulsanti. In figura 6.3 è possibile vedere un esempio di messaggio interattivo.

Dopo che l'operatore conferma o rifiuta mediante l'uso dei bottoni, le APIs di Slack si occupano di mandare una richiesta HTTP POST con la risposta.

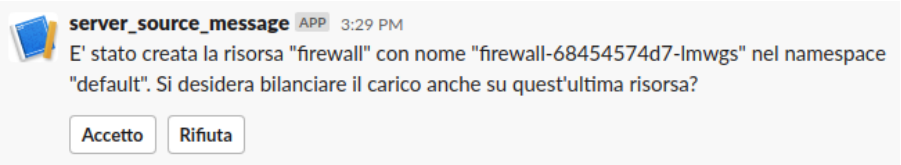

Figura 6.3. Esempio di messaggio Slack interattivo

Per poter intercettare tale richiesta è stato creato un web server in python usando Flask ed esposto verso l'esterno del cluster.
Quando la richiesta raggiunge il web server viene analizzata e generato un evento secondo le specifiche imposte dallo standard CloudEvent. Questo evento porta con se l'informazione sulla risposta dell'operatore. A questo punto una regola specifica riceverà l'evento e modificherà la configurazione del LB in base al contenuto dell'evento.

É importante capire quali sono gli attori in gioco e qual è il compito di ognuno di loro. Per questo motivo possiamo ricapitolare dividendo il flusso di esecuzione del sistema in due parti.

Evento interno

L'evento viene generato quando viene creata una nuova istanza di firewall (il numero di repliche passa da x a x+1). La logica di trattamento dell'evento interno è quella riportata di seguito:

1. Kubernetes lancerà un evento interno come conseguenza della creazione della nuova risorsa (come abbiamo visto nel paragrafo 2.2.2);
2. tale evento verrà intercettato dall'API Server Source di Knative (di cui abbiamo parlato nel paragrafo 4.1.2) e inoltrato al broker sotto forma di evento CloudEvent;
3. una specifica rule di KRules si occuperà di intercettare questo evento e dopo averlo elaborato manderà una notifica su Slack all'operatore di rete.

Evento esterno

L'evento viene generato quando l'operatore approva, o rifiuta, il bilanciamento sulla nuova replica di firewall. La logica di trattamento dell'evento esterno è quella riportata di seguito:

1. l'API di Slack invierà una richiesta HTTP POST al server interno al cluster ed esposto verso l'esterno;
2. il server analizzerà la richiesta e genererà un evento in formato CloudEvent diretto al broker;
3. una specifica rule di KRules intercetterà tale evento e modificherà la Routing Table del Load Balancer per consentire la ridirezione delle chiamate in arrivo anche verso il firewall aggiuntivo.

6.1 Sources

Nell'implementazione del caso d'uso l'unica source di Knative usata è l'APIServerSource. Come già anticipato questa source consente di intercettare gli eventi che Kubernetes utilizza solo a scopo informativo e di debugging e di trasformarli in un formato definito dallo standard CloudEvent. Gli eventi raccolti e trasformati a questo punto vengono inviati al broker, pronti per essere inviati ai consumer che si sono sottoscritti.

Gli eventi che possono essere lanciati da Kubernetes possono riguardare decisioni prese dallo scheduler, errori o altri messaggi da dichiarare al sistema, oppure possono riguardare la creazione, modifica o eliminazione delle risorse di Kubernetes, come ad esempio nel caso d'uso di questa tesi può essere la creazione di una nuova replica di Firewall.

Per creare una sorgente mediante l'APIServerSource è necessario prima aver creato il broker (di solito con l'installazione della componente di Eventing viene fornito un broker di default) e impostato correttamente il serviceAccount (necessario affinché tale source possa contattare l'apiserver del cluster). Solo dopo è possibile definire una risorsa di tipo APIServerSource come quella di seguito:

```
apiVersion: sources.knative.dev/v1alpha1
kind: ApiServerSource
metadata:
  name: testevents
  namespace: default
spec:
  serviceAccountName: events-sa
  mode: Resource
  resources:
    - apiVersion: v1
      kind: Pod
```

```
sink:
  ref:
    apiVersion: eventing.knative.dev/v1alpha1
    kind: Broker
    name: default
```

Come è possibile osservare dall'attributo resources, questa sorgente è interessata solo e soltanto alle modifiche che interessano le risorse Pod. Ciò vuol dire che gli unici eventi che intercetterà saranno tutti quelli che riguardano la creazione, modifica ed eliminazione dei Pods. Gli eventi saranno poi inviati al Broker di default, come è possibile intuire dai valori passati al kind e al name situati nella parte relativa al ref del sink.

A seconda degli eventi di Kubernetes che vogliamo intercettare è possibile indicare diverse risorse. Ad esempio se si è interessati ai Deployments, alle ConfigMap, o a qualunque altra risorsa di Kubernetes, basta solo metterla nella sezione resources dell'APIServerSource. É anche possibile indicare più di una risorsa. Infine se si è interessati a tutti gli eventi lanciati da Kubernetes, basterà indicare il kind Event.

Oltre all'APIServerSource di Knative nell'implementazione del caso d'uso c'è un altro componente che di fatto si comporta come sorgente di eventi: il web server che riceve la richiesta HTTP da Slack. Infatti dopo aver elaborato il body della richiesta genera un evento che indica la scelta fatta dall'operatore. Questa è una sorgente che genera eventi a seconda di ciò che succede all'esterno del cluster, nello specifico su Slack. Questa sorgente non è stata creata usando una componente source di Knative, ma il suo funzionamento è stato implementato a basso livello usando le librerie messe a disposizione da CloudNative per generare eventi. Questo dimostra la flessibilità del sistema implementato in quanto anche se non c'è una source di Knative adatta ad uno specifico caso è sia possibile crearne una usando la guida di Knative, che far generare in maniera elementare eventi da una qualsiasi componente, rendendola di fatto una sorgente di eventi e quindi un input per il sistema. Naturalmente lo stesso risultato si sarebbe potuto ottenere con un source di Knative come ad esempio SinkBinding il quale si occupa di collegare le risorse di Kubernetes "addressable" che possono ricevere eventi con le risorse che vogliono produrre eventi. Questo faciliterebbe sicuramente l'implementazione, ma si è voluto solo dare un'idea delle varie possibilità che si hanno con le sorgenti.

6.2 Rules

Dopo aver analizzato il flusso di esecuzione che il sistema dovrà rispettare, gli eventi che dovrà intercettare e le sorgenti degli eventi non resta che analizzare nel dettaglio i rulesets che sono stati creati per il caso d'uso.

Nella figura 6.4 è possibile osservare come con soli quattro rulesets è stato possibile implementare la logica descritta precedentemente.

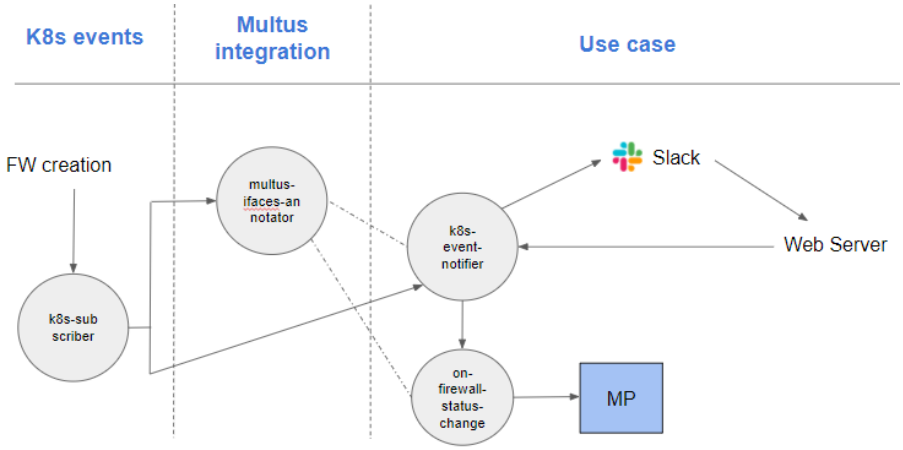

Figura 6.4. Schema riassuntivo dei rulesets creati per il caso d'uso

Nello schema però si evince anche un'altra cosa: tali rulesets non sono tutti allo stesso livello di astrazione. Infatti, in maniera più estesa, i rulesets si potrebbero dividere in tre macro livelli:

- estensioni del framework di basso livello: queste componenti inizialmente non sono parti del framework, ma vengono create per facilitare lo sviluppo del caso d'uso. In generale sono delle regole di basso livello che servono a supportare il programmatore nella creazione dei rulesets successivi. Quindi queste regole non fanno altro che creare uno strato di supporto che è al di sopra del framework ma che pone le basi per la creazione dei rulesets specifici per il caso d'uso. Nello schema di figura 6.4 questo tipo di componente è rappresentato dal ruleset *k8s-subscriber* il quale, come si vedrà successivamente, è il primo componente a ricevere gli eventi interni di Kubernetes e prima di inoltrarli ne modifica il nome e il tipo in modo da rendere il filtraggio e l'elaborazione seguente più semplice ed immediata. Nello specifico, lo sviluppo di questo ruleset è stato supportato da AirSpot, proprietario del framework KRules, che ha seguito l'attività di ricerca e sperimentazione;

- assets e ruleset di dominio riutilizzabili: questo livello comprende tutti quei rulesets e micro-assets (piccole funzioni riutilizzabili che compongono le regole) che vengono usati per un caso d'uso ma che in realtà risolvono problemi riguardanti domini più generali e possono essere usati per diverse applicazioni con poche o nessuna modifica. Nello schema precedente questo tipo di componente è rappresentato dal ruleset *multus-ifaces-annotator* il quale diventa utile ogni qual volta viene usato Multus. Nello specifico permette di operare in maniera più agevole con le risorse che hanno interfacce settate da Multus;

- caso d'uso specifico: questo livello racchiude tutti quei rulesets usati per una specifica applicazione. Nell'esempio realizzato per questa tesi, i rulesets che fanno parte di questo livello sono *k8s-event-notifier* e *on-firewall-status-change*. Difficilmente questi rulesets posso essere riutilizzati in altri domini, ma se la soluzione è stata ben strutturata in livelli, come è stato fatto per questo caso d'uso, le rules di questi componenti saranno molto semplici e piccole.

Questa suddivisione non è in alcun modo imposta dal framework ma rappresenta esclusivamente una suddivisione concettuale. Tale organizzazione concettuale indica come sia possibile, o meglio necessario, strutturare la propria soluzione a strati e renderla quanto più modulare possibile. Tale modularità favorisce il riutilizzo del codice e delle regole e rende il sistema ancora più versatile e resiliente.

Lo schema in figura 6.4 oltre a rappresentare i rulesets mostra anche come questi sono collegate tra loro e come interagiscono. Per comprendere meglio tale interazione è necessario analizzare più nel dettaglio ogni singola regola.

k8s-subscriber

Questo ruleset ha una sola rule che si sottoscrive direttamente all'ApiServerSource di Knative. Il suo compito è semplicemente quello di cambiare il *type* e il *subject* dell'evento che riceve nella forma "k8s:<resource_api_path>". Come abbiamo visto nel paragrafo 4.2.2 se il subject è strutturato in quel modo, il framework utilizzerà per la sua memorizzazione la soluzione Custom.

```
rulesdata = [
  {
    rulename: "on-resource-event-switch-subject",
    subscribe_to: [
      "dev.knative.apiserver.resource.add",
```

```
      "dev.knative.apiserver.resource.update",
      "dev.knative.apiserver.resource.delete",
    ],
    ruledata: {
      processing: [
        Route(lambda self:
               "k8s.resource.{}".format(self.type.split(".")[-1]),  #type
              lambda payload:
               "k8s:{}".format(payload["metadata"].get("selfLink")),
                        dispatch_policy=DispatchPolicyConst.DIRECT) #subject
      ]
    }
  },
]
```

Questa regola ci permette di definire dei trigger che per filtrare gli eventi generati dall'ApiServerSource possono basarsi sul type "k8s.resource.add", "k8s.resource.update" o "k8s.resource.delete" in base al tipo di evento che si vuole intercettare (creazione, modifica o eliminazione delle risorse Kubernetes). In questo modo i trigger possono mandare gli eventi ai rulesets che si sono sottoscritti e che useranno la soluzione custom per operare sul subject.

Un esempio di trigger è il seguente. Questo filtra solo sugli attributi *resourcetype* e *type*. In altre parole accetta solo gli eventi provenienti dalla creazione di un nuovo Pod. Gli eventi che corrispondono a questa descrizione vengono inviati al ruleset multus-ifaces-annotator.

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: multus-trigger-add
spec:
  broker: default
  filter:
    attributes:
        resourcetype: pods
        type: k8s.resource.add
  subscriber:
    ref:
      apiVersion: v1
      kind: Service
      name: multus-ifaces-annotator
```

Come è stato anticipato prima il ruleset k8s-subscriber è un'estensione del framework di basso livello perché di fatto non fa parte del caso d'uso ma è pensata per essere disponibile per tutti i progetti in cui vogliamo modellare la logica attorno agli eventi di Kubernetes.

multus-ifaces-annotator

Come abbiamo già avuto modo di vedere nel paragrafo relativo a Multus, tale CNI permette di aggiungere più interfacce di rete ai Pod. Se dal punto di vista di KRules una risorsa come un Pod può essere vista come un subject, allora le sue interfacce di rete possono essere delle proprietà reattive del subject.

Quando un Pod, che ha tra le sue annotazioni il riferimento ad un NetworkAttachmentDefinition, viene creato, multus si occupa di creare le interfacce di rete e di annotare le informazioni riguardo le interfacce sulla risorsa. Nella rule multus-resources-identifier la sezione filters si occupa proprio di intercettare gli eventi delle risorse che hanno tra le loro annotazioni i riferimenti alle interfacce aggiunte da multus: "k8s.v1.cni.cncf.io/networks" e "k8s.v1.cni.cncf.io/networks-status". Quindi nella sezione di processing arriveranno solo gli eventi corrispondenti all'aggiornamento di Pods che sono stati configurati con Multus. Il processing consta di sole due funzioni: *SetMultusInterfaces* e *SetSubjectExtendedProperty*.

```
rulesdata = [
  {
    rulename: "multus-resources-identifier",
    subscribe_to: ["k8s.resource.update"],
    ruledata: {
      filters: [
        IsTrue(lambda payload:
          "k8s.v1.cni.cncf.io/networks" in
            payload.get("metadata").get("annotations", {}) and
          "k8s.v1.cni.cncf.io/networks-status" in
            payload.get("metadata").get("annotations", {})),
      ],
      processing: [
        SetMultusInterfaces(),
        SetSubjectExtendedProperty("multusapp", lambda payload:
          payload.get("metadata").get("labels", {}).get("app", "unknown"))
      ],
    },
```

```
  },
]
```

La funzione *SetMultusInterfaces* è una funzione custom che è stata creata ad hoc ed è stata definita nello stesso file del ruleset appena visto. Questa funzione ci permette di aggiungere delle proprietà reattive al subject che contengono le informazioni sulle interfacce e sui loro indirizzi. In questo modo sarà più facile e veloce, per le regole che successivamente utilizzeranno il subject, estrapolare tali informazioni.

Memorizzare le interfacce di rete come proprietà reattive ha un duplice vantaggio:

1. possono essere facilmente lette e semplificare la lettura e la scrittura del codice;

2. in scenari più complessi possono essere usate per scrivere rules che vengono invocate solo quando, per qualche ragione, vengono create nuove interfacce o quelle esistenti sono alterate (per esempio viene assegnato un nuovo indirizzo IP ad un'interfaccia). Le regole riceveranno non solo il nuovo valore ma anche il vecchio, e risponderanno in maniera trasparente senza la necessità di sapere le motivazioni della modifica.

```
class SetMultusInterfaces(RuleFunctionBase):

  def execute(self):
    interfaces = self.payload["metadata"]["annotations"]
    interfaces = interfaces["k8s.v1.cni.cncf.io/networks"].split(', ')
    statuses = self.payload["metadata"]["annotations"]
    statuses = json.loads(statuses["k8s.v1.cni.cncf.io/networks-status"])
    for interface in interfaces:
      for status in statuses:
        if status.get("name") == interface:
          self.subject.set(status.pop("name"), status)
          break
```

La seconda funzione *SetSubjectExtendedProperty* è una funzione built-in, che setta una proprietà estesa, anche detta extended properties. Questa proprietà viene chiamata "multusapp" e utilizzata all'interno di un trigger ci permette di intercettare tutti (e solo) gli eventi riguardanti una precisa risorsa. In questo modo le regole successive potranno intercettare gli eventi relativi alle VNF semplicemente usando un trigger con un filtro sull'attributo multusapp. Questo è possibile solo perché ogni risorsa che rappresenta una VNF ha un

label con un valore che la identifica. Quindi ad esempio il Firewall ha un label del tipo "app: firewall". In questo modo se si vogliono intercettare solo gli eventi riguardanti il firewall ci basterà mettere tra gli attributi del trigger "multusapp: firewall".

Come è stato già anticipato multus-ifaces-annotator è un rulset riutilizzabile e torna utile tutte le volte che viene usato Multus. Infatti, facilità il filtering delle VNF e l'estrapolazione delle informazioni riguardanti le loro interfacce di rete. Questo diventa fondamentale per poter lavorare in maniera più agile con le regole successive. Per questo motivo nella figura 6.4 tale ruleset è collegato ai due successivi mediante una linea tratteggiata; perché anche se tra loro non c'è un collegamento diretto, senza multus-ifaces-annotator gli altri due rulesets non potrebbero funzionare correttamente.

k8s-event-notifier

A differenza dei due rulesets visti finora, k8s-event-notifier è specifico per il caso d'uso preso in esame da questo lavoro di tesi. Questo è costituito da due rules: intercept-fw-creation e new-firewall-approved-subscriber. Dopo che l'evento generato dalla creazione del Firewall viene modificato da k8s-subscriber e multus-ifaces-annotator ne aggiunge le proprietà riguardanti le interfacce di rete, inizia un processo di approvazione guidato dalle due rules di k8s-event-notifier. Per tenere traccia dei progressi di questo processo viene usata una proprietà reattiva chiamata "approval_status". Tale processo è costituito dalle seguenti fasi:

1. inizialmente viene intercettata la nuova risorsa Pod creata (o aggiornata) ed etichettata con "multusapp:firewall" ma con la proprietà approval_status non ancora settata;

2. successivamente viene mandato un messaggio Slack interattivo e in attesa di ricevere la risposta dall'operatore, la proprietà reattiva approval_status viene impostata a "pending" sul subject (ovvero la nuova istanza del Firewall);

3. quando la rule riceve l'evento di approvazione proveniente da Slack, approval_status viene settato ad "approved". Come sappiamo l'evento ricevuto viene prodotto dal server esposto verso l'esterno, ma lo avrebbe potuto generare qualunque servizio, l'importante è che sia formattato secondo lo standard definito da CloudEvent;

4. essendo approval_status una proprietà reattiva, nel momento in cui viene modificata viene lanciato un evento, il quale verrà intercettato dal ruleset on-firewall-status-change.

Nello specifico la rule intercept-fw-creation nella sezione di filters si assicura che la risorsa sia Running e che la proprietà approval_status non sia nel subject. In questo modo evita sia di lavorare su risorse che non sono ancora state del tutto create e che potrebbero avere problemi o essere distrutte e sia di elaborare eventi che sono già stati elaborati da questa regola e che hanno lo stato di approvazione già settato.

Gli eventi che passano i filtri finiscono nella sezione di processing. Questa sezione è costituita da tre funzioni: *ComposeText*, *SlackPublishInteractiveMessage*, *SetSubjectProperty*.

```
{
  rulename: "intercept-fw-creation",
  subscribe_to: [
    "k8s.resource.add",
    "k8s.resource.update"
  ],
  ruledata: {
    filters: [
      IsTrue(lambda payload:
               payload.get("status", {}).get("phase", "") == "Running"),
      IsTrue(lambda subject: "approval_status" not in subject),
    ],
    processing: [
      ComposeText("text"),
      SlackPublishInteractiveMessage(
                    channel="kr-k8s-lab-clusterevents",
                    text=lambda payload: payload["text"]
                    )
      SetSubjectProperty("approval_status", "pending"),
    ],
  },
},
```

ComposeText è una funzione custom il cui compito è quello di comporre il testo che verrà mostrato nel messaggio interattivo di Slack e mettere tale testo nel payload del subject in modo da poterlo ricavare facilmente nella funzione successiva.

SlackPublishInteractiveMessage è anch'essa una funzione custom che in ingresso richiede due parametri: il nome della proprietà che identifica l'incoming Webhook per Slack e il testo del messaggio che verrà visualizzato (preso dal payload del subject).
Infine SetSubjectProperty è la funzione integrata in Krules che si occupa di modificare la proprietà approval_status in pending.

Al contrario la rule new-firewall-approved-subscriber si occupa di intercettare l'evento che arriva da Slack che sarà del tipo "new-firewall-approved". Questa regola non ha filtri e la sezione di processing ha una sola funzione: SetSubjectProperty. Questa è la funzione integrata che modifica la proprietà approval_status in approved. Quest'ultima modifica scatena un evento che verrà intercettato dall'ultima regola, on-firewall-status-change, che viene eseguita solo dopo che l'operatore ha confermato da Slack il bilanciamento anche sulla nuova istanza di Firewall.

```
{
  rulename: "new-firewall-approved-subscriber",
  subscribe_to: "new-firewall-approved",
  ruledata: {
    processing: [
      SetSubjectProperty("approval_status", "approved"),
    ],
  },
}
```

on-firewall-status-change

Nell'ordine di esecuzione dell'intero processo questo è l'ultimo ruleset che viene triggerato. Viene eseguito quando la proprietà approval_status del subject, corrispondente al Firewall appena creato, diventa "approved". Questo è possibile perchè approval_status è una proprietà reattiva e, in quanto tale, ad ogni sua modifica KRules genera un evento che porta con se il nuovo valore assegnato e il vecchio valore. Il type di questi eventi è settato come "subject-property-changed", quindi per intercettarli è sufficiente creare un trigger che filtra gli eventi in base a quel type. Questo è ciò che viene fatto con il ruleset on-firewall-status-change, infatti ad esso è associato un trigger che intercetta il cambiamento della proprietà approval_status e lo inoltra al ruleset. Di seguito il codice che realizza tale trigger.

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
```

```
metadata:
  name: on-approval-status-change
spec:
  broker: default
  filter:
    attributes:
      type: subject-property-changed
      propertyname: approval_status
  subscriber:
    ref:
      apiVersion: v1
      kind: Service
      name: on-firewall-status-change
```

Il compito di questo ruleset è quello di aggiornare la routing table del Load Balancer, in modo da bilanciare il traffico anche sulla nuova istanza di Firewall appena creata. Per fare questo è stata usata una sola rule chiamata on-firewall-approved e una funzione custom chiamata K8sObjectsQuery.

L'aggiornamento della routing table viene fatto mediante l'esecuzione del comando "ip route" sulla risorsa Load Balancer. Come abbiamo già visto, in realtà, il Load Balancer non è altro che un Hash-based Multipath Routing. Il Multipath Routing permette di spedire i pacchetti destinati ad una sola rete attraverso diverse rotte. Il modo in cui distribuisce i pacchetti dipende dall'implementazione del protocollo. Quello presente nel kernel linux dalla versione 4.4 per IPv4 è basato su hash, usando indirizzo di destinazione e sorgente [19]. Quindi non si ha un bilanciamento per pacchetto ma per flusso (a livello 3). Il comando ip route permette di definire un multipath routing usando la parola chiave nexthop. Un esempio di comando che imposta due rotte per una rete di destinazione è il seguente, in cui assumiamo che 192.168.3.0/24 sia l'indirizzo di net3 (la rete che collega i FW al WebServer), 192.168.2.0/24 sia l'indirizzo di net2 (la rete che collega il LB ai FW) e 192.168.2.1 e 192.168.2.2 gli indirizzi delle interfacce di rete dei due FW collegate alla net2.

```
ip route add 192.168.3.0/24 \
    nexthop via 192.168.2.1 \
    nexthop via 192.168.2.2
```

Prima di modificare la configurazione della routing table, il ruleset on-firewall-status-change ha bisogno di recuperare gli indirizzi della rete di destinazione (net3) e delle interfacce di rete dei FW collegati alla stessa rete

del Load Balancer. Questi valori vengono recuperati mediante l'utilizzo della funzione K8sObjectQuery. Questa è una funzione custom, ma data la sua utilità, nelle future versioni di KRules verrà integrata all'interno del framework.

K8sObjectQuery è una funzione che permette di lavorare con le risorse di Kubernetes. Nello specifico permette di recuperare informazioni riguardo le risorse, ma anche di crearle o modificarle. In ingresso a K8sObjectQuery è possibile passare fino a cinque parametri:

- apiversion: è l'apiVersion delle risorse che ci interessa recuperare. Se non viene indicato di default è "None".
- kind: è il kind delle risorse che ci interessa recuperare. Se non viene indicato di default è "None".
- filters: dopo che la funzione ha fatto una query, recuperando tutte le risorse del sistema con l'apiversion e il kind indicati prima, è possibile applicare dei filtri in modo da mantenere solo le risorse che ci interessano. Se non viene indicato di default è un oggetto vuoto.
- foreach: qui viene passata la funzione che vogliamo venga eseguita per ogni risorsa recuperata. Se non viene indicato di default è "None".
- returns: se diversa da "None" (valore di default) la funzione ritorna il vettore delle risorse, altrimenti solo il numero di risorse trovate.

La rule on-firewall-approved nella sezione di processing richiama la funzione K8sObjectQuery tre volte, una per recuperare l'indirizzo della rete di destinazione e salvarlo nel payload, una per recuperare le interfacce di rete dei FW e salvare anche queste nel payload e un'altra per modificare la routing table del Load Balancer.

```
{
  rulename: "on-firewall-approved",
  subscribe_to: "subject-property-changed",
  ruledata: {
    filters: [
      SubjectPropertyChanged("approval_status", "approved"),
    ],
    processing: [
      K8sObjectsQuery(
        apiversion="k8s.cni.cncf.io/v1",
```

```
        kind="NetworkAttachmentDefinition",
        foreach=lambda payload: lambda obj:
            set_subnet_in_payload(obj.obj,"fw-sv-macvlan-conf",payload)
      ),
      K8sObjectsQuery(
        apiversion="v1",
        kind="Pod",
        filters={
          "selector": {
            "app": "firewall",
          },
        },
        foreach=lambda payload: lambda obj:
            set_running_fw_in_payload(obj.obj, payload)
      ),
      K8sObjectsQuery(
        apiversion="v1",
        kind="Pod",
        filters={
          "selector": {
            "app": "multipath",
          },
        },
        foreach=lambda payload: lambda obj:
            add_fw_to_lb(obj,payload["subnet"],payload["fw_ips"])
      )
    ],
  },
},
```

Come possiamo vedere dal codice la prima K8sObjectQuery recupera tutte le risorse NetworkAttachmentDefinition e mediante la funzione chiamata set_subnet_in_payload (definita precedentemente nello stesso file) salva nel payload l'indirizzo di rete della risorsa NetworkAttachmentDefinition che si chiama fw-sv-macvlan-conf. Questo è possibile perché si è deciso di chiamare tutte le risorse NetworkAttachmentDefinition mediante una nomenclatura che faccia capire quale rete rappresenta. Ad esempio fw-sv-macvlan-conf rappresenta la rete che c'è tra i firewalls e il web server, lb-fw-macvlan-conf quella tra il load balancer e i firewall e infine cl-lb-macvlan-conf quella tra i client e il load balancer. In questo modo diventa semplice indicare una specifica rete.

La seconda K8sObjectQuery recupera tutti i Pods, ma filtra sul label "app: firewall". Su ogni risorsa che passa il filtro viene eseguita la funzione set_running_fw_in_payload (definita precedentemente nello stesso file), la quale si occupa di ricavare gli indirizzi IP delle interfacce dei Firewall running e memorizzarli nel payload per un successivo utilizzo. Queste informazioni sugli indirizzi delle interfacce sono quelle che ha salvato prima il ruleset multus-ifaces-annotator.

A questo punto tutte le informazioni necessarie per l'aggiornamento della routing table sono disponibili e memorizzate nel payload. L'ultima funzione K8sObjectQuery non fa altro che recuperare il Load Balancer, identificato dal label "app: multipath", ed eseguire la funzione add_fw_to_lb. É possibile eseguire il comando iproute, visto prima, sul container del Load Balancer grazie all'utilizzo della funzione exec_command che mediante l'uso della libreria pykube permette l'esecuzione di comandi direttamente sulla shell del container.

```
def add_fw_to_lb(ob, dest_net_subnet, fw_ips):
  command = [
    '/bin/sh',
    '-c',
    'ip route del '+dest_net_subnet]
  exec_command(ob, command=command, container="multipath",
               preload_content=False)
  if len(fw_ips):
    separator = ' via '
    if len(fw_ips) > 1:
      separator = ' nexthop via '
    command = [
      '/bin/sh',
      '-c',
      'ip route add '+dest_net_subnet+separator+separator.join(fw_ips)]
    exec_command(ob, command=command, container="multipath",
                 preload_content=False)
```

Terminata l'esecuzione di questo ruleset il sistema sarà stabilizzato e il traffico diretto dai client al web server verrà bilanciato anche sulla nuova istanza di Firewall.

Capitolo 7

Validazione e conclusione

Il sistema generico studiato durante questo lavoro di tesi ha prodotto ottimi risultati, in quanto non solo rispetta tutti i vincoli imposti dall'obiettivo, ma grazie ai framework Knative e KRules garantisce un'alta versatilità, riusabilità e capacità di estensione.
La conclusione sopra riportata è stata raggiunta dopo aver valutato le diverse possibilità tramite un approccio sperimentale realizzato mediante lo sviluppo di codice e customizzazioni prototipali. Di seguito sono riportate le fasi del cammino che ha portato alla nostra conclusione.

Inizialmente si è cercato di risolvere il problema usando gli operatori standard di Kubernetes e quindi sfruttando il meccanismo dei controller e delle CRDs. Questa non si è rivelata la soluzione più adatta per diversi motivi, ma essenzialmente il motivo principale è che questo tipo di approccio è rigido, poco consono alla realizzazione di logiche flessibili e variabili nel tempo. Infatti, come abbiamo visto, l'operatore non fa altro che automatizzare task noti e definiti per una specifica applicazione.
In un secondo momento, si è visto che neanche creare un sistema basato su operatori, combinandoli come abbiamo visto nel paragrafo 3.3, porterebbe un grande valore aggiunto al risultato, considerando i problemi e le limitazioni che sono state sottolineate.

Dopo aver analizzato pro e contro dei primi due approcci si è deciso di implementare una soluzione basata sull'utilizzo dei framework Knative e Krules per la gestione degli eventi che possono influenzare il ciclo di vita delle Virtual Network Function in un ambiente cloud native. Come evidenziato nel capitolo 5 e nel paragrafo 6.2, il framework permette un'elevata modularità che è necessaria per organizzare sistemi reattivi complessi che ben si adattano alla gestione di una infrastruttura da parte degli operatori di rete.

L'implementazione dei framewok utilizzati favorisce la riusabilità del codice e rende il sistema resiliente e versatile. Tali caratteristiche risultano particolarmente apprezzate da parte degli operatori di rete in quanto abilitano la maintenance di sistemi complessi. In questo modo è possibile aumentare il livello di complessità del sistema a seconda delle esigenze. La soluzione implementata in questa tesi rappresenta solo un Proof of Concept della soluzione adottabile ma evidenzia già in modo chiaro i vantaggi che questa può portare soprattutto con l'adozione del framework Krules che si è dimostrato potente ed estremamente estendibile.

Per quanto riguarda le prestazioni non è stato fatto uno studio prestazionale del sistema o dell'overhead introdotto da Knative e Krules. Senza ombra di dubbio l'utilizzo dei due framework va ad influire sulla quantità di risorse hardware che il cluster dovrà dedicare ad ogni componente, ma si possono fare alcune considerazioni. Knative è composto da due componenti, Serving ed Eventing. Nella soluzione proposta, la sola componente indispensabile è quella di Eventing, in quanto fornisce l'infrastruttura necessaria per la gestione degli eventi. Quindi le risorse accessorie richieste da questa componente, anche se minime, non possono essere evitate. A meno di particolari necessità è consigliato non usare il Serving se si vuole alleggerire il sistema e i singoli microservizi. Tale componente porta con se anche l'uso di Istio, che aggiungendo un sidecar per ogni Pod rallenta il loro deployment e l'esecuzione.
I microservizi sono semplicemente Pods (costituiti da un solo container) e Services. Inoltre se si utilizza come metodo di memorizzazione dei subject quello custom, che sfrutta l'etcd, si riducono anche le chiamate verso il server API e si risparmia lo spazio che occuperebbero altre soluzioni come Redis e Mongodb.
In termini di overhead la mia conclusione è che l'adozione della solo componente Eventing di Knative, insieme a Krules, rappresenti un compromesso accettabile considerati i vantaggi che questa soluzione offre.

L'ultimo aspetto da valutare è la quantità di codice scritto per l'implementazione del caso d'uso e la semplicità delle regole realizzate. Scrivere una regola richiede poche righe di codice in quanto consiste solo nel richiamare delle funzioni. Più complesso potrebbe risultare l'implementazione di nuove funzionalità di reazione specifiche del dominio di applicazione (in questo caso il dominio degli operatori di rete) nel caso in cui queste non siano già offerte in modo nativo dal framework Krules. Queste funzioni possono essere più o meno complesse a seconda dei task che devono compiere, ma il fatto che bisogna scriverle in Python rende sicuramente più facile e veloce la loro realizzazione piuttosto che usare un linguaggio come GO.

In questo momento il framework è ancora in fase di sviluppo ed evoluzione, quindi il numero e la tipologia di funzioni e assets disponibili non sono molte, ma ne verranno realizzate una quantità sempre maggiore nelle prossime versioni. Già durante lo svolgimento di questa tesi, il team di Airspot, che ha contribuito allo sviluppo dell'implementazione, ha realizzato e messo a disposizione funzionalità e assets specifiche per il caso d'uso che prima non esistevano. Ad esempio si pensi al ruleset multus-ifaces-annotator che semplifica la gestione delle risorse modificate da Multus; questo tipo di microservizio può risultare utile tutte le volte che si usa tale CNI. Ciò significa che una volta creato un componente questo può essere riutilizzato tutte le volte che vogliamo grazie al disaccoppiamento e alla riusabilità dei microservizi e alla natura versatile del framework. É chiaro che con un maggiore utilizzo e una maggiore apertura del framework alla community (cosa che avverrà nei prossimi mesi) non ci vorrà molto prima che vengano creati sempre più assets e funzioni da riutilizzare e richiamare semplicemente nelle regole.

Mentre Knative è un framework molto diffuso ed usato, con un'ampia documentazione e supporto dalla community, KRules è un framework ancora agli stati iniziali del suo processo evolutivo e non è ancora adatto ad essere utilizzato in ambiente di produzione. Inizialmente, infatti, sono stati riscontrati numerosi problemi con l'installazione e l'integrazione di KRules con gli altri componenti del cluster locale messo in piedi mediante l'uso di Kind. A causa di una mancanza di documentazione non è semplice iniziare lo sviluppo con questo framework e capire a fondo come funziona e come programmarlo.

Molti di questi problemi sono dovuti al fatto che KRules è ancora in fase di sviluppo e in continua evoluzione, infatti è cambiato molto nel corso della tesi e continuerà ad evolversi. L'utilizzo del framework Krules nel progetto di tesi ha permesso inoltre ad Airspot di redarre una guida all'installazione ed utilizzo del framework che viene messa a disposizione della community Open source.
Nonostante queste difficoltà iniziali, quando si comprende appieno il suo funzionamento diventa davvero semplice da usare e le sue potenzialità sono tante.

Dallo studio svolto da questa tesi è emerso che KRules risolve un problema reale, e mediante il suo utilizzo è possibile ottenere un risultato più efficace ed efficiente rispetto a quello che si otterrebbe con altri framework. In sua assenza l'unica possibilità è di programmare in modo imperativo degli operator per svolgere dei task per una specifica applicazione.

Un aspetto che potrebbe essere preso in considerazione in un lavoro futuro e che non è stato analizzato in questa tesi è sicuramente l'efficienza del

sistema, in termini di consumo di risorse e di tempi di risposta, confrontandola con una soluzione composta da sole risorse native di Kubernetes (come controller e CRD). In conclusione, nonostante le problematiche evidenziate su KRules, ci si può ritenere soddisfatti dei risultati raggiunti dal lavoro di tesi in quanto i benefici apportati dai due framework li rendono la soluzione migliore per un ambiente dinamico come il cloud.

Bibliografia

[1] Virtualization vs. Containerization. https://www.liquidweb.com/kb/virtualization-vs-containerization/. 12/2019

[2] Cos'è Kubernetes?. https://kubernetes.io/it/docs/concepts/overview/. 2020.

[3] An illustrated guide to Kubernetes Networking. https://itnext.io/an-illustrated-guide-to-kubernetes-networking-part-1-d1ede3322727. 2017

[4] The Kubernetes network model. https://kubernetes.io/docs/concepts/cluster-administration/networking/. 2020

[5] Kubernetes Networking. https://cloudnativelabs.github.io/post/2017-04-18-kubernetes-networking/. 2017

[6] Multus CNI. https://github.com/intel/multus-cni. 2020

[7] Kubernetes Objects. https://kubernetes.io/it/docs/concepts/ 2020

[8] Stefan Schimanski, Michael Hausenblas, *Programming Kubernetes: Developing Cloud-native Applications*, O'Reilly.

[9] Kubebuilder. https://github.com/kubernetes-sigs/kubebuilder.

[10] What is kudo. https://kudo.dev/docs/what-is-kudo.html.

[11] Event-Driven Architecture for Cloud-Native in Kubernetes. https://www.xenonstack.com/insights/eda-for-cloud-native-kubernetes/. 2020

[12] Events in stackdriver. https://kubernetes.io/docs/tasks/debug-application-cluster/events-stackdriver/. 5/2020

[13] CloudEvent. https://cloudevents.io/.

[14] Knative. https://knative.dev/docs/.

[15] Knative Eventing. https://knative.dev/docs/eventing/.

[16] KRules Documentation. https://intro.krules.io/. 11/2020

[17] KRules CLI. https://github.com/airspot-dev/krules-py-cli.

[18] Multipath Routing in linux. https://codecave.cc/multipath-routing-in-linux-part-1.html. 2017

[19] Linux kernel Multipath routing.
https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=0e884c78ee19e902f300ed147083c28a0c6302f0