POLITECNICO DI TORINO

Corso di laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

# Orchestrazione distribuita di servizi in scenari *multi-provider*

**Relatore**
prof. Fulvio Risso

**Candidato**
Andrea Mora

Dicembre 2019

“Grazie soprattutto ai miei genitori,
questo traguardo è anche vostro.”

# Abstract

Negli ultimi anni, i metodi impiegati e le strategie di allocazione, distribuzione e fruizione di servizi sono state drasticamente rivoluzionate dall'introduzione di concetti e tecnologie innovative, come virtualizzazione e orchestrazione. Un ruolo preponderante è stato giocato da Kubernates, piattaforma open source sviluppata da Google, che ha migrato la gestione delle risorse presenti nei cluster di server verso una virtualizzazione basata su container e simili. L'avvento dell'edge computing ha portato alla nascita di progetti recenti che mirano all'utilizzo di Kubernetes all'edge della rete. Le soluzioni esistenti indirizzano principalmente lo scenario dell'IoT, e adottano un approccio di tipo gerarchico, in cui molteplici hosts alla periferia diventano parte di un cluster più ampio gestito in maniera centralizzata dal cloud. Recenti approcci indirizzano inoltre il problema della federazione di più clusters all'interno dello stesso dominio amministrativo, prevedendo che un cluster master mantenga il control plane della federazione. Invece, è attualmente ignorato il problema di federare clusters appartenenti a providers differenti, in cui l'esistenza di un master risulta impraticabile.

In questo lavoro viene proposta una nuova architettura basata su DRONE (Distributed Resources Offloading at the Network Edge), un algoritmo di orchestrazione distribuito sviluppato presso il Politecnico di Torino. In questo lavoro è stata dapprima svolta una validazione delle capacità offerte da DRONE in ambito distribuzione ed esecuzione di servizi tra providers differenti. Successivamente, viste le funzionalità del prototipo, sono state implementate soluzioni pratiche per l'utilizzo di questo algoritmo di orchestrazione tra cluster federati, con lo scopo finale di integrare DRONE in Kubernetes, permettendo così la federazione di molteplici clusters in maniera trasparente. I tests effettuati sul prototipo sviluppato consentono di attestare i vantaggi derivanti dall'utilizzo di questo approccio e di analizzare le performance della soluzione.

# Indice

**Elenco delle figure** 7

**1 Introduzione** 10

**2 Orchestrazione di servizi** 12
2.1 Orchestrazione Servizi . . . . . 12
2.2 Edge Computing . . . . . 13
2.3 Obiettivo della tesi . . . . . 14

**3 Stato dell'arte** 15
3.1 Kubernetes all'Edge . . . . . 15
3.1.1 K3S . . . . . 15
3.1.2 KubeEdge . . . . . 16
3.2 Orchestrazione in cluster . . . . . 18
3.2.1 Kubernetes Scheduler . . . . . 19
3.3 Orchestrazione multi cluster . . . . . 19
3.3.1 KubeFed . . . . . 19

**4 Tecnologie utilizzate** 22
4.1 Docker . . . . . 22
4.1.1 Componenti principali . . . . . 22
4.1.2 Container . . . . . 23
4.1.3 Networking . . . . . 24
4.2 Kubernetes . . . . . 25
4.2.1 Concetti base . . . . . 25
4.2.2 Networking . . . . . 28
4.2.3 CRD & Operator . . . . . 29
4.2.4 Admission Webhook . . . . . 30
4.3 RabbitMQ . . . . . 31
4.3.1 Tipologie di Exchange . . . . . 31
4.3.2 Management . . . . . 33
4.3.3 Federation . . . . . 33

**5 Il sistema di orchestrazione DRONE** 35
5.1 Panoramica di Drone . . . . . 35
5.2 Funzionamento dell'algoritmo . . . . . 36
5.3 Conclusione . . . . . 37

**6 Architetture dell'orchestratore in K8S** 38
6.1 Prima soluzione proposta . . . . . 38
6.1.1 Architettura dettagliata del sistema . . . . . 39
6.1.2 In-Cluster Orchestrator . . . . . 42
6.1.3 Multi-Cluster Orchestrator . . . . . 44
6.1.4 Proof-of-Concept: Avvio applicazioni in nodo remoto . . . . 45
6.2 Evoluzione della soluzione . . . . . 47
6.2.1 Architettura dettagliata del sistema . . . . . 47

**7 Implementazione orchestrazione in K8S** 50
7.1 Primo Prototipo . . . . . 50
7.1.1 Configurazione dei nodi . . . . . 51
7.1.2 Messaggi . . . . . 53
7.1.3 DRONE Controller . . . . . 58
7.1.4 DRONE Gui Interface . . . . . 63
7.1.5 Esempio VLC . . . . . 63
7.2 Evoluzione Prototipo . . . . . 69
7.2.1 Configurazione clusters . . . . . 69
7.2.2 Messaggi . . . . . 70
7.2.3 DRONE-Operator . . . . . 71
7.2.4 DRONE-Daemon-Resources . . . . . 78

**8 Risultati Sperimentali** 83
8.1 Validazione Algoritmo Drone in Cluster . . . . . 83
8.2 Validazione Approccio Multi Clusters . . . . . 85

**9 Conclusioni e sviluppi futuri** 87

# Elenco delle figure


3.1 K3S Architecture . . . . . 16
3.2 K3S Architecture for Edge use . . . . . 17
3.3 KubeEdge Architecture . . . . . 17
3.4 KubeEdge Architecture for Edge use . . . . . 18
3.5 Process a Pod . . . . . 19
3.6 KubeFed concepts . . . . . 20
3.7 KubeFed Cluster Configuration . . . . . 20
3.8 KubeFed Architecture . . . . . 21

4.1 Differenza tra Virtual-Machine e Container . . . . . 24
4.2 Kubernetes Architecture . . . . . 25
4.3 Kubernetes Worker nodes with pods . . . . . 26
4.4 Kubernetes Deployments . . . . . 27
4.5 Kubernetes services example: a frontend pod and three pod backend 27
4.6 Left: Pod in cluster; Right: cluster virtual namespaces . . . . . 28
4.7 Kubernetes Networking . . . . . 29
4.8 Kubernetes Operator . . . . . 30
4.9 Kubernetes Admission WebHook . . . . . 30
4.10 Flusso di messaggi in RabbitMQ (Fonte: CloudAMQP [9]). . . . . 32
4.11 Federation . . . . . 34
4.12 Pair Federated Exchange . . . . . 34

5.1 DRONE Orchestration . . . . . 37

6.1 Architettura utilizzo DRONE multi-cluster . . . . . 39
6.2 Esempio di comunicazione tra nodi connessi ad una rete . . . . . 40
6.3 Esempio di comunicazione tra cluster connessi . . . . . 40
6.4 Struttura del singolo nodo in questa architettura . . . . . 41
6.5 Architettura in-cluster con i messaggi . . . . . 43
6.6 Architettura multi-cluster con messaggi . . . . . 44
6.7 Prototipo realizzato . . . . . 45
6.8 Architettura Integrata in Kubernetes . . . . . 47
6.9 Messaggi tra Clusters . . . . . 48

7.1 Comunication RabbitMQ . . . . . 53
7.2 DRONE Controller FlowChart . . . . . 60
7.3 Queue Comunication RabbitMQ . . . . . 63
7.4 Gui Interface . . . . . 64
7.5 Evolution FlowChart . . . . . 71
7.6 Reconcile Evolution FlowChart . . . . . 76
7.7 Acknowledge consume FlowChart . . . . . 77
7.8 Advertisement consume Evolution FlowChart . . . . . 78
7.9 Result consume FlowChart . . . . . 79
7.10 Resources Daemon FlowChart . . . . . 81

8.1 Validazione DRONE Ambiente . . . . . 83
8.2 Validazione DRONE Algoritm . . . . . 84
8.3 Validazione Multi-Cluster Ambiente . . . . . 85
8.4 Validazione Multi-Cluster Scheduler . . . . . 85

# Listings


7.1 Advertisement Type . . . . . 54
7.2 Advertisement Message Structure . . . . . 54
7.3 Component Structure . . . . . 55
7.4 Resources Message Structure . . . . . 56
7.5 Solution Message Structure . . . . . 57
7.6 Local Offloading Structure . . . . . 57
7.7 Overall Offloading Structure . . . . . 57
7.8 UserResquest Message Structure . . . . . 58
7.9 DRONE-Controller Configuration . . . . . 59
7.10 Send Function . . . . . 62
7.11 Docker VLC . . . . . 64
7.12 Streamer YAML . . . . . 65
7.13 Gui YAML . . . . . 66
7.14 Acknowledge Message Structure . . . . . 70
7.15 Example CustomResources . . . . . 73
7.16 CustomResourcesDefinition Schema . . . . . 73
7.17 ConfigMap Drone Resources . . . . . 79
7.18 Role Drone Resources . . . . . 80
7.19 Deployment Drone Resources . . . . . 82

# Capitolo 1

# Introduzione

Negli ultimi anni, i metodi impiegati nei cluster e le strategie di allocazione, distribuzione e fruizione di servizi sono state drasticamente rivoluzionate dall'introduzione di concetti e tecnologie innovative, come virtualizzazione e orchestrazione. Con l'aumento del numero di container e servizi e diventato essenziale un sistema in grado di gestire il ciclo di vita dei container e verificarne lo stato

Per questo motivo un ruolo preponderante è stato giocato da Kubernates, piattaforma open source sviluppata da Google, che permette una distribuzione automatica, gestione, ridimensionamento dei container all'interno del cluster. L'avvento dell'edge computing ha portato alla nascita di progetti recenti che mirano all'utilizzo di Kubernetes all'edge della rete.

Sebbene Kubernetes sia in grado di fare tutto quello citato sopra non ha ancora la possibilità di gestire una distribuzione automatica su un ambiente multi cluster. Sono state proposte delle soluzioni, come KubeFed, per la federazione di cluster, ma tutte queste soluzioni sono improntate ad avere una centralizzazione.

DRONE è una soluzione completamente distribuita che permette di orchestrare container su cluster distinti.

Questo elaborato è strutturato come segue:

- **Capitolo 2:** presenta un rapido background della teoria che sta alla base dell'orchestrazione ed espone in maniera esauriente il problema che questa tesi si propone di risolvere.

- **Capitolo 3:** descrive lo stato dell'arte delle soluzioni e delle architetture utilizzate nel contesto dell'orchestrazione di servizi in rete in un cluster o tra cluster differenti.

- **Capitolo 4:** presenta una panoramica delle tecnologie sulle quali questa tesi si basa e dei componenti utilizzati.

- **Capitolo 5:** descrive l'architettura di DRONE, l'orchestratore distribuito sul quale si basa il lavoro di tesi.
- **Capitolo 6:** presenta le architetture proposte per affrontare il problema descritto nel Capitolo 2
- **Capitolo 7:** descrive delle implementazioni del problema descritto nei capitoli precedenti riguardo all'orchestrazione in un cluster e su più cluster.
- **Capitolo 8:** espone dei risultati ricavati misurando le performance ottenute utilizzando DRONE.
- **Capitolo 9:** definisce le conclusioni e presenta alcuni progetti futuri che avranno come base il lavoro svolto in questa tesi.

# Capitolo 2

# Orchestrazione di servizi

Questo capitolo si pone lo scopo di discutere e spiegare le basi su cui si è svolto il lavoro di tesi. Verrà affrontato il problema dell'edge computing e verrà esposto in modo esauriente l'obiettivo delle tesi.

## 2.1 Orchestrazione Servizi

In questa sezione verrà descritto il problema che si intende risolvere.

In questi anni con una continua evoluzione delle tecnologie di virtualizzazione si è arrivati ad avere cluster sempre più capienti e prestanti per sostenere la quantità di lavoro richiesto. In questo scenario, le infrastrutture posizionate all'edge della rete, risentono di un notevole carico che a volte viene migrato verso il cloud. Tutto questo con conseguenze di peggioramento delle prestazioni, come latenza ecc.

Nel corso degli anni sono nate diverse soluzioni in grado di creare degli insiemi di cluster così da poter sfruttare risorse e capabilities disponibili su cluster remoti. Tutte queste soluzioni però prevedono il controllo da parte di un'entità centralizzata. Questa entità deve conosce molte informazioni dei cluster membri della federazione. Inoltre questo orchestratore deve tener conto in fase di scheduling delle risorse/capabilities disponibili su ciascun nodo in modo da distribuire ciascuna richiesta su una infrastruttura in grado di soddisfarla.

Tutte queste considerazioni, in uno scenario in cui i cluster sono di domini differenti, potrebbero sollevare molti dubbi negli attori che vorrebbero "federare" i propri cluster. Semplicemente, ad esempio, un provider non vorrebbe comunicare molte informazioni interne dei suoi cluster con un provider avversario.

Il paradigma emergente di Edge Computing ha consentito ai service provider di fornire una grande varietà di nuove applicazioni, che beneficiano della presenza di storage e servizi informatici ai margini della rete e di una minore latenza verso gli utenti finali. Inoltre, le tecnologie di virtualizzazione consentono l'isolamento tra diverse applicazioni che possono essere eseguite simultaneamente su sezioni separate

di un'infrastruttura fisica condivisa. Negli ambienti basati su cloud, la distribuzione dei servizi viene spesso delegata a un componente centralizzato, denominato Orchestrator [1], che di solito sfrutta una politica adatta a tutti (one-size fits-all policy) (ad esempio, risparmio energetico, numero di nodi utilizzati, bilanciamento del carico) a decidere (i) dove posizionare i componenti del servizio, (ii) quante risorse devono essere assegnate a ciascuna di esse [2] e (iii) l'insieme di metriche / eventi che segnalano che il servizio deve essere riprogrammato (ad es. un aumento imprevisto del carico).

Tuttavia, tale approccio centralizzato può essere non ottimale o non applicabile ai margini della rete. In primo luogo, tale ambiente può essere caratterizzato da alti tassi di abbandono, cambiamenti imprevedibili nella topologia della rete e persino partizioni di rete temporanee; questo può favorire approcci di orchestrazione distribuiti contro un orchestratore centralizzato, che potrebbe non essere nemmeno raggiungibile. In secondo luogo, l'insieme di applicazioni ampiamente eterogenee in esecuzione sul bordo della rete può avere obiettivi diversi e imprevedibili, per non parlare della necessità di reagire in modo diverso allo stesso evento, come ad esempio un aumento del carico. Nella suddetta circostanza, alcune applicazioni possono aumentare o diminuire la scala, altre possono modificare il comportamento del servizio (ad esempio, scegliere un transcodificatore video più aggressivo), altre migrazioni e altro; i suddetti comportamenti sono difficili da ottenere con un orchestratore adatto a tutti.

## 2.2 Edge Computing

Evoluzione del cloud computing, consente di accorciare le distanze tra client ed esecutore di servizi e velocizzare tutte le operazioni interconnesse

Il cloud computing si tratta della modalità di erogazione di alcuni servizi – come l'archiviazione, l'elaborazione o la trasmissione di dati – che avviene tramite Internet e su richiesta esplicita dell'utente (on demand).

I maggiori fornitori di cloud pubblico (ossia, liberamente accessibile a qualunque utente) stanno iniziando a guardare con sempre maggior interesse a una nuova modalità di erogazione di servizi tramite la Rete: l'edge computing. Questa nuova tecnologia dovrebbe velocizzare l'erogazione e l'accesso alle varie applicazioni pensate per gli utenti e, allo stesso tempo, ridurre drasticamente la quantità di banda dati utilizzata dall'utente per accedere ai servizi e dall'erogatore per fornirli.

Non ci si deve stupire, quindi, se le varie Amazon, Microsoft, Google e IBM guardano con sempre maggior interesse all'edge computing e pensano di utilizzarlo per rimpiazzare il cloud computing. Come accennato, i vantaggi che questa modalità di erogazione e accesso a servizi Internet potrebbe garantire sono molteplici e potrebbero diventare molti di più, nel momento in cui diventerà di dominio comune.

La grande differenza tra le due tecnologie sta nella "distanza" che separa chi eroga il servizio e chi vuole accedervi. Mentre il cloud viene erogato in maniera centralizzata attraverso data center distanti centinaia e centinaia di chilometri dall'utente, l'edge computing è una tecnologia di "prossimità" (sul bordo, come suggerisce anche il nome), che consente di erogare servizi Internet analoghi a distanze di gran lunga inferiori.

Come già detto più volte, la possibilità di elaborare dati a breve distanza è il maggior vantaggio che l'edge computing garantisce rispetto al cloud computing. Da questo punto di partenza derivano, poi, tutti gli altri vantaggi che l'edge computing saprà garantire da qui a qualche anno. Prima di tutto, ci sarà una notevole riduzione della latenza nell'elaborazione e accesso ai dati; la maggior vicinanza tra client e dispositivi che eseguiranno materialmente le operazioni, poi, farà sì che la quantità di dati che deve essere "mossa" da un capo all'altro sarà inferiore portando a una minor richiesta di banda dati.

## 2.3 Obiettivo della tesi

In questa tesi viene delineato un nuovo sistema di orchestrazione che si pone di risolvere le problematiche dell'avere una orchestrazione centralizzata in un ambiente multi provider. Con la soluzione proposta si riuscirà ad avviare delle applicazioni standard all'interno di una federazione di cluster, più o meno grandi, senza dover esportare all'esterno dei clusters delle informazioni sensibili.

La nuova architettura è basata su l'algoritmo DRONE, sviluppato presso il politecnico, in cui si sfrutta la completa distribuzione. In questa implementazione ci si basa sul paradigma di *"publish/subscribe"* in cui ogni agente "agente" DRONE si occuperà di valutare un determinato job, pubblicando sul *message bus* in modo che l'interessato possa leggere le informazioni.

Ogni agente dell'orchestratore quindi resterà in ascolto sul bus condiviso, in attesa di nuovi job o nuove informazioni. Tutta questa parte di algoritmica verrà unita a una parte di implementazione della architettura generale in modo da poter usare la soluzione.

# Capitolo 3

# Stato dell'arte

Dopo aver esposto, nel capitolo precedente, il principale problema che questa tesi vuole risolvere in quest'altro capitolo viene descritto lo stato dell'arte dell'edge computing e dell'orchestrazione interna a un cluster e tra cluster differenti in questi anni e verranno analizzati i problemi che queste soluzioni lasciano irrisolti.

## 3.1 Kubernetes all'Edge

In questa sezione descriviamo i vari progetti esistenti indirizzati alla migrazione di un ambiente Kubernetes all'edge della rete.

### 3.1.1 K3S

K3s è una distribuzione certificata Kubernetes, prodotta da Rancher labs, progettata per carichi di lavoro di produzione in posizioni remote non presidiate, limitate dalle risorse o all'interno di dispositivi IOT [1].

K3S è leggera (impacchettato in un singolo binario di circa 40 MB), molto facile da installare e richiede solo 512 MB di RAM. In questo binario è racchiuso tutto ciò che è necessario per eseguire Kubernetes, incluso il runtime del container e tutte le utility host importanti come iptables, socat e du. Quindi risulta un candidato perfetto per dispositivi IoT ed edge computing o per processi di CI. [2]

I requisiti sono molto ridotti: Linux 3.10+, 512 MB di RAM per server, 75 MB di RAM per nodo, 200 MB di spazio su disco, x86_64, ARMv7, ARM64.

K3S raggruppa i componenti di Kubernetes (kube-apiserver, kube-controller-manager, kube-scheduler, kubelet, kube-proxy) in processi combinati che vengono presentati come un semplice modello server e agente. L'esecuzione k3s server avvierà il server Kubernetes e registrerà automaticamente l'host locale come agente. Per aggiungere più nodi al cluster, basta eseguire *"k3s agent –server ${URL} –token ${TOKEN}"* sull'host che si unirà al cluster.

Oltre ai componenti principali di Kubernetes, vengono eseguiti anche containerd, Flannel, CoreDNS, Traefik (controller di ingresso) e un semplice bilanciamento del carico del servizio. Come meccanismo di default per lo storage è stato scelto sqlite3 al posto di etcd, che rimane disonibile ma non default. Tutti questi componenti sono opzionali e possono essere sostituiti con l'implementazione scelta.

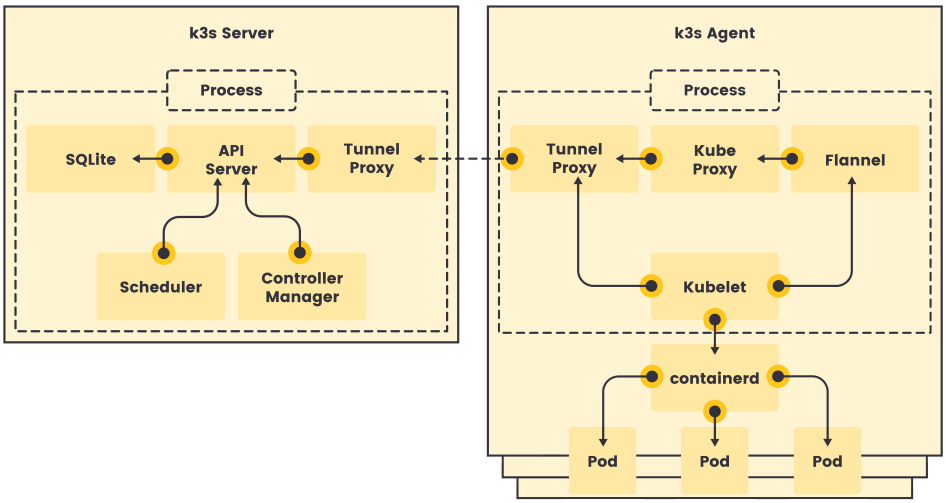


Figura 3.1: K3S Architecture

K3S esegue un cluster Kubernetes completo all'edge, il che significa che non è un modello di distribuzione decentralizzato e ogni cluster edge ha una propria logica di controllo. Questa soluzione può essere impiegata in un'ambiente multi edge come segue:

### 3.1.2 KubeEdge

KubeEdge è un sistema open source per estendere le funzionalità di orchestrazione delle applicazioni all'edge. É basato su Kubernetes e offre un supporto per gestire la rete, i deployments delle applicazioni e la sincronizzazione *cloud-edge* [3].

L'obiettivo è quello di creare una piattaforma aperta per sfruttare l'Edge computing, estendendo le capacità di orchestrazione nativa delle applicazioni agli host di Edge, che si basano su Kubernetes e forniscono un supporto di infrastruttura fondamentale per la rete, la distribuzione di app e la sincronizzazione dei metadati tra cloud e edge.

KubeEdge è nato da Huawei Cloud ed è il primo progetto open source nel campo dell'edge computing e Kubernetes.

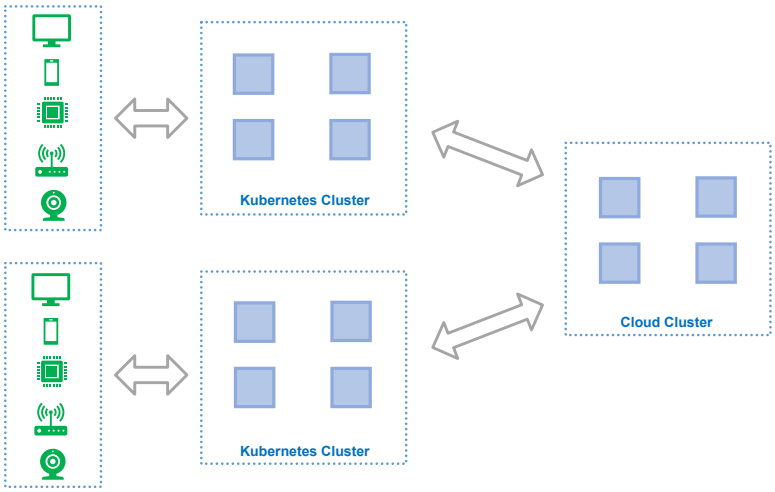


Figura 3.2: K3S Architecture for Edge use

KubeEdge utilizza il protocollo MQTT per la comunicazione tra DeviceTwin, dispositivi/app, machine-to-machine e per la comunicazione bidirezionale tra l'egde e il cloud. Ci sono vari borker MQTT disponibili, KubeEdge utilizza Mosquitto.

Come mostrato in Figura 3.3 KubeEdge è composto da due principali componenti: una parte cloud e una parte edge. Il piano di controllo solitamente risiede nella parte cloud ma allo stesso tempo con questa implementazione la parte edge può funzionare anche in assenza di connessione con la prima.

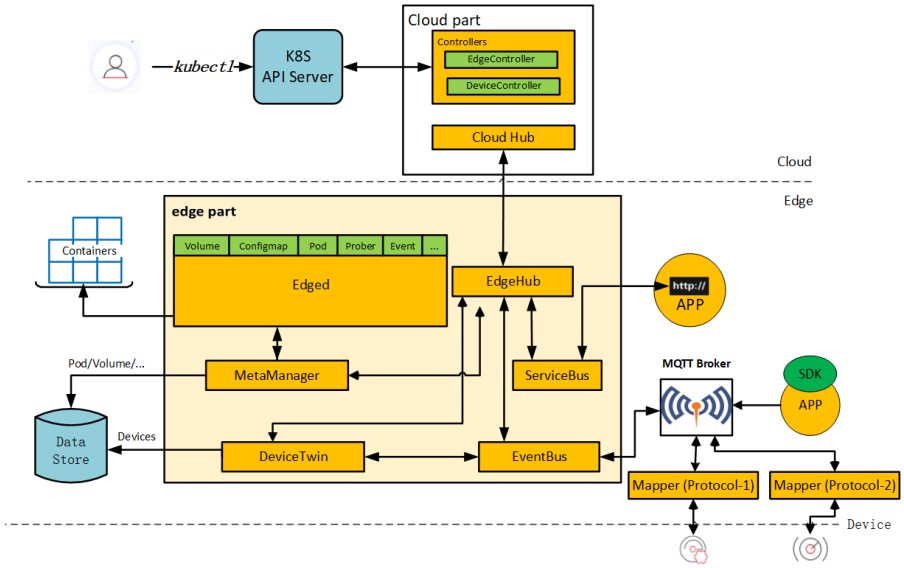


Figura 3.3: KubeEdge Architecture

Le parti più importanti di questa soluzione sono: [4]

- *Edged*: un agente che viene eseguito su nodi periferici e gestisce applicazioni e container;
- *EdgeHub*: un client socket web responsabile dell'interazione con il servizio cloud per il edge computing;
- *CloudHub*: un server socket Web responsabile della visualizzazione delle modifiche sul lato cloud, della memorizzazione nella cache e dell'invio di messaggi a EdgeHub;
- *EdgeController*: un controller kubernetes esteso che gestisce i nodi Edge e i metadati dei pod in modo che i dati possano essere indirizzati a un nodo Edge specifico.

KubeEdge è un modello di distribuzione completamente decentralizzato. Il piano di controllo è nel cloud. I nodi periferici possono eseguire degli agenti di Kubernetes senza molte risorse e il lato cloud può comunicare attraverso dei messaggi. Dal punto di vista di Kubernetes, Edge Node + Cloud è un cluster Kubernetes completo. Questo modello di distribuzione può soddisfare i requisiti degli scenari "device edge" e "infrastructure edge". La collaborazione sul cloud è un punto importante di KubeEdge

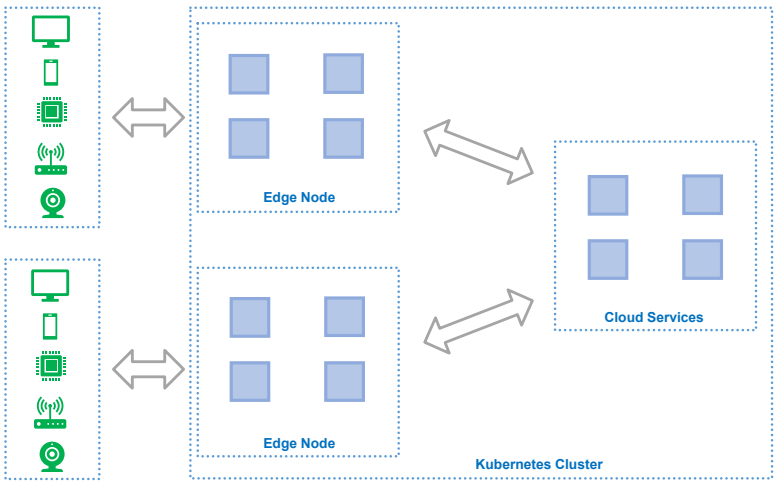


Figura 3.4: KubeEdge Architecture for Edge use

## 3.2 Orchestrazione in cluster

In questa sezione analizzeremo le principali soluzioni di scheduling esistenti in ambiente cluster Kubernetes. Tratteremo in breve lo scheduler standard di Kubernetes (Kube-scheduler) e alcuni scheduler personalizzati.

### 3.2.1 Kubernetes Scheduler

Lo Scheduler di Kubernetes, *kube-scheduler*, è responsabile dell'orchestrazione dei pod sui nodi del cluster.

- Ogni pod creato o che non è stato ancora assegnato viene aggiunto a una coda;
- Lo scheduler preleva dalla coda i pod e li assegna ad un nodo.

Lo scheduler non è responsabile dell'esecuzione effettiva del pod. Questo compito sarà svolto dalla *Kubelet* del nodo.

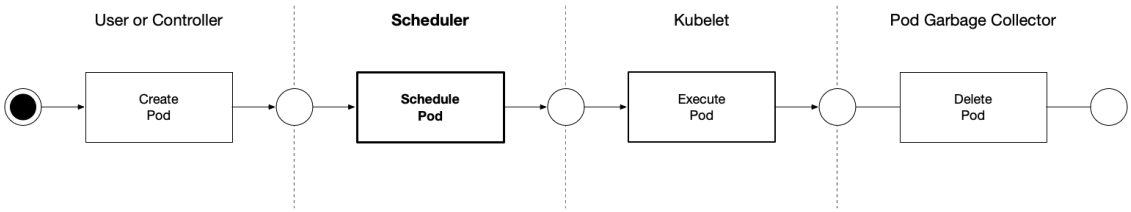


Figura 3.5: Process a Pod

In questa trattazione non andremo a descrivere come funziona lo scheduler standard di Kubernetes in quanto non è utile ai fini del lavoro di tesi. Si tratta di uno scheduler centralizzato che distribuisce i pod all'interno del cluster. In Kubernetes si possono definire degli scheduler custom da utilizzare per alcuni deployment.

## 3.3 Orchestrazione multi cluster

Allo stato attuale delle tecnologie non esistono soluzioni in grado di orchestrare dei pod su clusteer distinti. Ci sono state delle proposte di federazione tra cluster, come KubeFed, che però richiedono una centralizzazione del control plane in uno dei cluster.

### 3.3.1 KubeFed

Kubernetes Cluster Federation, *KubeFed*, è la proposta della comunità Kubernetes per la federazione tra cluster. Questo permette di coordinare più cluster da un cluster singolo, *host*, attraverso un set di API. KubeFed gestisce quali configurazioni e quali cluster utilizzare, e utilizza meccanismi di basso livello che permettono di gestire casi complessi come la distribuzione multi-geo o in casi di disaster recovery. KubeFed estende le API di Kubernetes per poter utilizzare le risorse federate attraverso il meccanismo delle CRD. Esso gestisce le CRD e implementa le funzioni come la sincronizzazione delle risorse e lo scheduling cross-cluster.[5]

KubeFed supporta due tipo di informazioni:

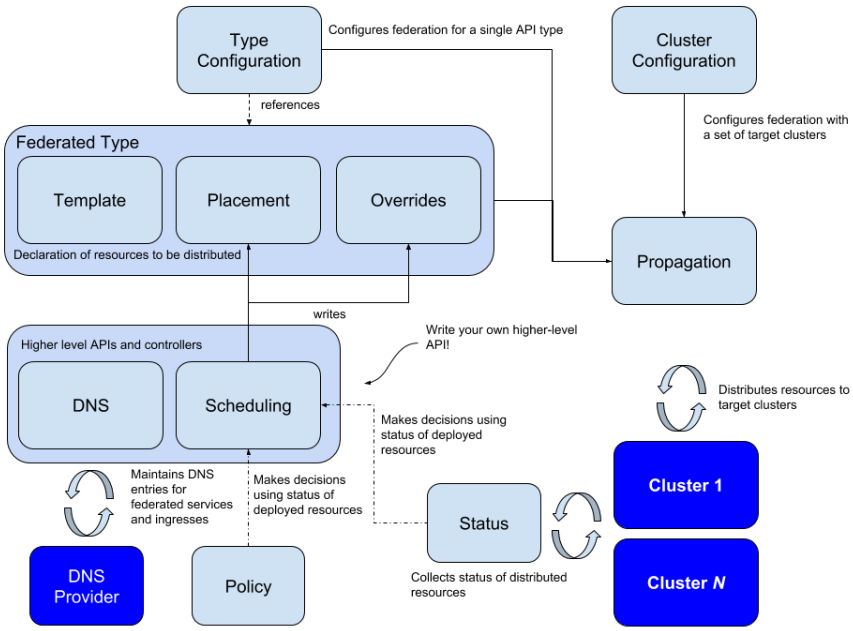


Figura 3.6: KubeFed concepts

- *Configurazione del tipo* dichiarando quali tipi di API devono essere gestiti da KubeFed;

- *Configurazione dei cluster* dichiarando quali cluster devono essere scelti dallo scheduler di KubeFed.

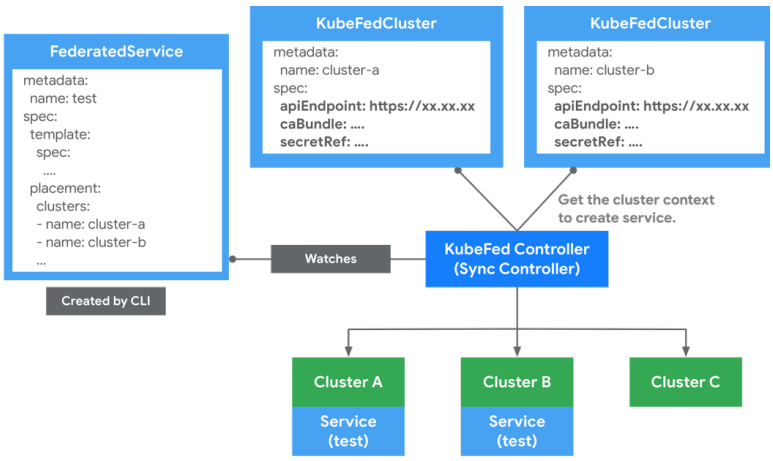


Figura 3.7: KubeFed Cluster Configuration

Come detto la configurazione dei cluster viene usata per definire quali cluster Kubernetes devono essere federati. Si può aggiungere/rimuovere un "membro" con il comando *"kubefedctl join/unjoin"*. Quando l'unione avviene con successo viene creato una CRD, *KubeFedCluster*, in cui vengono salvate le informazioni relative al cluster, come le API Endpoint, la CA e cosi via; tutte informazioni utilizzate dal controller KubeFed per accedere a differenti cluster.

Nella federazione di KubeFed esistono due tipi di clusters: *host cluster*: usato per gestire le API di KubeFed e il control plane; *member cluster*: un cluster registrato attraverso le API di KubeFed e che ha fornito le credenziali per abilitare il controller di KubeFed.

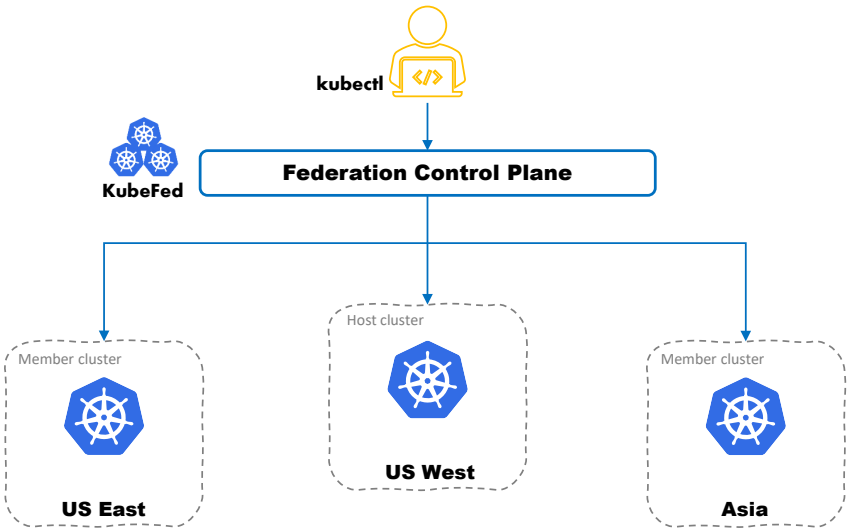


Figura 3.8: KubeFed Architecture

I tipi di risorse vengono strutturati con tre concetti fondamentali:

- *Templates*: definisce la rappresentazione della risorsa comune a più cluster;
- *Placement*: definisce in quali cluster la risorsa deve comparire;
- *Overrides*: definisce le variazioni da applicare al template per ogni cluster.

# Capitolo 4

# Tecnologie utilizzate

In questo capitolo vengono introdotte le principali tecnologie che sono state utilizzate per realizzare le architetture e le implementazioni proposte in questa tesi, spiegate in dettaglio nei capitoli 6 e 7.
Le due tecnologie usate maggiormente sono *Docker* e *Kubernetes*, tecnologie che negli ultimi anni sono state ampiamente utilizzate nel mondo cloud. In questo capitolo dedichiamo apposite sezioni per chiarire i concetti fondamentali di tali soluzioni. In aggiunta a quest'ultime si darà anche una breve spiegazione delle basi di *RabbitMQ*, message broker open-source che implementa il protocollo *AMQP*.
Molte sezioni di questo capitolo sui vari tools sono ispirate alle documentazioni e ai siti degli stessi.

## 4.1 Docker

Docker è un progetto open source nato con lo scopo di poter distribuire applicativi sotto forma di "contenitori" leggeri, autosufficienti e portabili da eseguire in locale o su dei cloud. L'idea della tecnologia Docker è stata sviluppata partendo dalla tecnologia LXC, una virtualizzazione leggera già esistente ma carente nell'utilizzo utente. Docker non consente solo di creare dei "container" ma semplifica anche tutto il processo di creazione, cancellazione, invio immagine e versioning. [**docker**] [6]. Docker è anche una società (Docker Inc.) che promuove e sviluppa questa tecnologia, collaborando con fornitori cloud, Linux e Windows.

### 4.1.1 Componenti principali

Docker è composto da quattro componenti principali:

- *Docker Engine*, é la base del funzionamento della piattaforma docker, si occupa di gestire i container e fornire tutte le funzionalità offerte da docker;

- *Docker Client*, mette a disposizione i comandi (CLI) con cui si può utilizzare le funzionalità offerte;
- *Docker Image*, è un file, composto da più *Layers*, utilizzato per eseguire in codice in un container docker, sono immutabili e non hanno uno stato. E' possibile costruire nuove immagini descrivendo il contenuto in file chiamati *Dockerfile* e possono essere archiviate in alcuni repository (pubblici o privati) così da poterle reperire sempre;
- *Docker Container*, è un istanza in esecuzione di un'immagine il cui file system è costituito da uno strato Read/Write sovrapposto agli strati immutabili dell'immagine.

### 4.1.2 Container

Docker raccoglie il software in unità standardizzate chiamate container che offrono tutto il necessario per la loro corretta esecuzione, incluse librerie, strumenti di sistema, codice, scripts, ecc. Questi container oltre ad essere autosufficienti sono anche portabili e vengono distribuiti in un formato standard, immagini, che possono essere lette da qualunque sistema in cui ci sia l'ambiente docker installato. Il software che viene racchiuso all'interno di un container può essere eseguito sia su Linux che su Windows e MacOS, sono quindi indipendenti dal sistema in cui vengono eseguiti. Inoltre sono leggeri perchè sfruttano i servizi offerti dal kernel del sistema ospitante invece di richiedere l'esecuzione di un kernel virtualizzato come succede per le VM. Come già accennato prima il concetto di Container non è stato introdotto da Docker ma esisteva già da tempo. Il loro utilizzo infatti si avvicina molto a tecnologie già esistenti come OpenVZ (ormai obsoleto) e il successore LXC. Tutte queste tecnologie si basano sul kernel linux in particolare su cgroups e namespaces. In seguito analizzeremo la differenza tra Virtual-Machine e Docker.

#### Container vs Virtual Machine

Container e VM, seppur sembrino molto simili tra loro e sfruttino gli stessi benefici legati all'isolamento delle risorse, nella sostanza risultano essere molto diversi. Detto in termini semplici: un container virtualizza un sistema operativo, mentre una virtual machine l'hardware. Per questo motivo i primi sono molto più efficienti dei secondi.

Un container è un'astrazione del layer applicativo che combina in un'unica soluzione codice e dipendenze. Più container possono essere istanziati sulla stessa macchina e condividere contemporaneamente lo stesso kernel, ognuno di essi viene infatti eseguito come un singolo processo in *user space*. I container hanno bisogno ovviamente di minor spazio delle VM ed il loro avvio avviene quasi immediatamente.

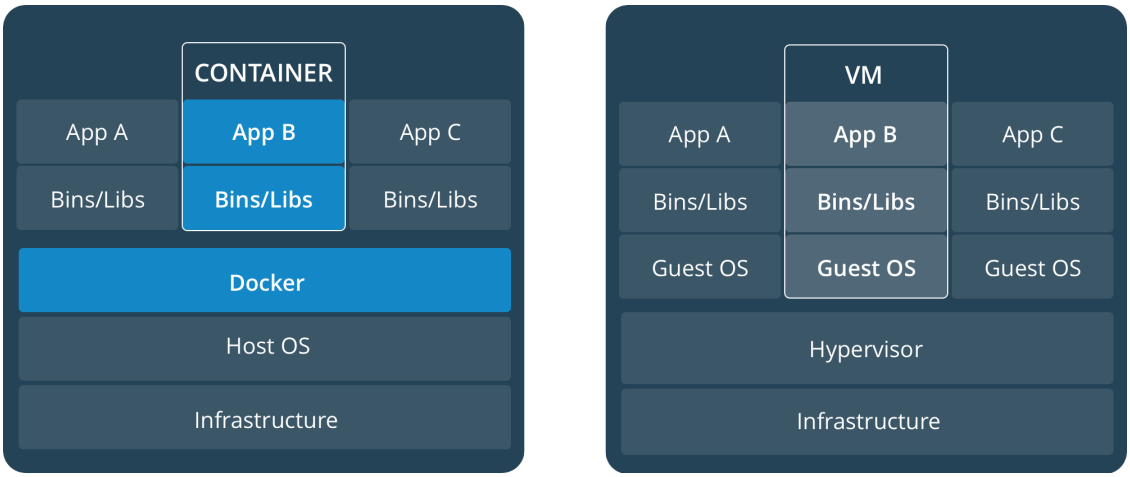


Figura 4.1: Differenza tra Virtual-Machine e Container

Con Virtual Machine si intende un'astrazione fisica dell'hardware. C'è la necessità di avere un hypervisor che si occupi di gestire più VM in una sola macchina. Ogni VM include una copia **intera** del sistema operativo, una o più applicazioni e tutte le librerie necessarie, per un totale di decine di GB. Il boot è molto spesso lento e non paragonabile ai tempi di avvio di un container.

### 4.1.3 Networking

Un altro aspetto importante di Docker è quello legato al networking. Durante l'installazione, vengono create tre reti: *bridge*, *none* e *host* [7]. Nel momento in cui si manda in esecuzione un container, è possibile selezionare tramite il comando `-network` quale rete utilizzare.

- **bridge**: è la rete di default. Tutti i nuovi container vengono connessi a questa rete se non gli viene specificato. Viene associata all'interfaccia virtuale `docker0` e vengono assegnati IP come 172.0.0.1/16. Quindi due container conoscendo l'ip reciprocamente possono comunicare.
- **none**: un container a cui viene specificato questa rete dispone solo dell'indirizzo di loopback e non può comunicare con altri container, diventa un container isolato.
- **host**: abilita un container a collegarsi all'intero stack di rete dell'host.
- **custom**: è possibile creare reti multiple personalizzate e aggiungere container a una o più reti. I container possono comunicare all'interno di una rete, ma non attraverso reti distinte, utilizzando il nome del container senza sapere l'indirizzo IP. Questa funzionalità è fornita dal DNS creato dal demone Docker all'atto della creazione di una nuova rete.

## 4.2 Kubernetes

[8] Kubernetes è una piattaforma portabile ed estensibile per la gestione di carichi di lavoro e servizi in container, open-source e creata da Google nel 2014. Orchestra l'infrastruttura informatica, di rete e di archiviazione per conto degli utenti dei carichi di lavoro ed è stato progettato anche come piattaforma per la creazione di un ecosistema di componenti e strumenti per semplificare l'implementazione, il ridimensionamento e la gestione delle applicazioni. Inoltre, il piano di controllo di Kubernetes è basato sulle stesse API disponibili per sviluppatori e utenti. Si possono ad esempio scrivere i propri controller, ed ampliare le API.

### 4.2.1 Concetti base

In questa sezione vengono analizzati i concetti base della piattaforma Kubernetes.

#### Nodes

Come la maggior parte delle architetture distribuite, il cluster Kubernetes è composto da almeno un nodo *master* e da più nodi computazionali, *worker nodes*. Un nodo rappresenta una singola macchina nel cluster, per la maggior parte delle volte è una macchina fisica in altre è una virtual machine.

Il master è responsabile di schedulare i deployments, gestire il cluster ed esporre le API. Ogni nodo esegue un demone runtime per gestire i container, come ad esempio Docker. I worker nodes sono i nodi di calcolo del cluster, il cluster utilizza le risorse computazionali, di rete e di memorizzazione per le applicazioni.

Su ogni nodo viene eseguito un agente, *Kubelet*, che permette al nodo di comunicare con il nodo master e con l'API server.

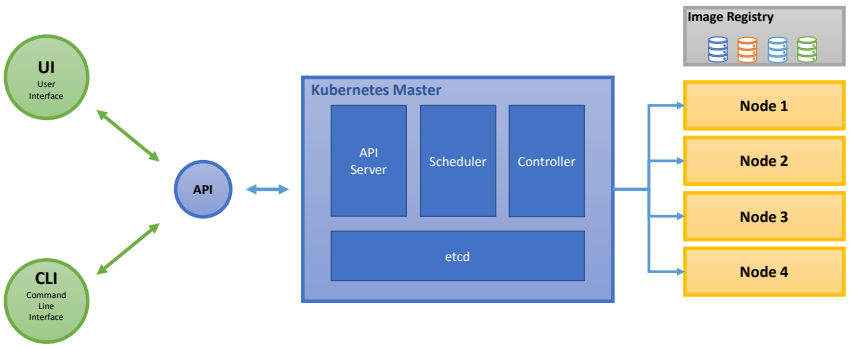


Figura 4.2: Kubernetes Architecture

### Pods

Kubernetes a differenza di altri sistemi non avvia container direttamente; esso infatti crea un *Wrapper* di alto livello intorno al container stesso chiamato *Pod*. I container nello stesso pod condividono le risorse e la rete locale. I pod sulla stessa macchina comunicavo facilmente attraverso il bridge locale, riuscendo a mantenere un livello di isolamento.

I pod sono usati come unità in K8S, essi sono quelli che consumano risorse e che possono essere replicati. Ad esempio se una applicazione diventa troppo carica, Kubernetes può essere configurato in modo da scalare e replicare il pod.

I pod possono contenere più Container ma si preferisce l'utilizzo con uno singolo, a causa del possibile scalamento del pod. Ci sono dei tipi di container invece che vengono eseguiti all'interno dello stesso pod. Questi possono essere dei container di supporto al processo principale, chiamati *side-cars*, oppure container di inizializzazione, chiamati *init container*.

I pod possono possedere i seguenti stati: Pending, Running, Succeded, Failed, Unknown. Inoltre possono avere: Error o CrashloopBackOff.

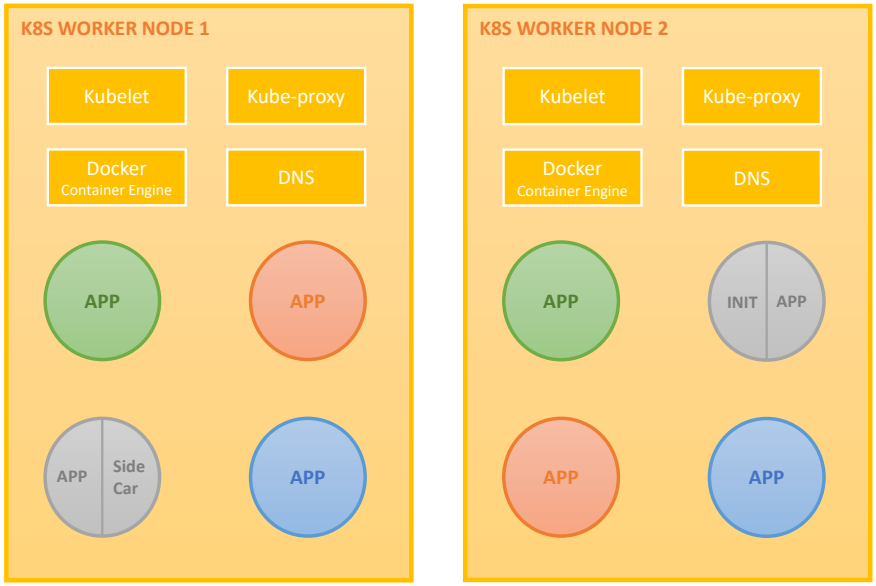


Figura 4.3: Kubernetes Worker nodes with pods

### Deployments

I pod non sono direttamente lanciati sul cluster anche se sono l'unità base. Questi sono di solito gestiti da un livello di astrazione superiore chiamato *Deployment*.

Lo scopo principale del Deployment è dichiarare il numero di repliche da eseguire per un pod. Quando un deployment viene creato sul cluster, vengono automaticamente create tutte le repliche definite e monitorate. Se per qualche motivo il pod muore, viene ricreato automaticamente.

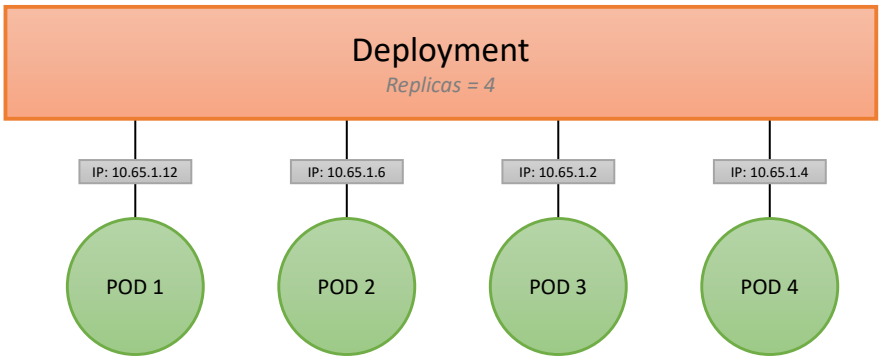


Figura 4.4: Kubernetes Deployments

### Services

I servizi in Kubernetes sono una astrazione che permette di raggruppare un set di Pod sotto una singola risorsa, risolvendo il problema del dover conoscere l'indirizzo IP di ognuno. Nell'immagine viene mostrato come ad esempio un pod di *frontend* non debba conoscere l'indirizzo di tutti i pod di *backend*, esso deve solo conoscere il nome/ip assegnato al *servizio* associato e verrà scelto un pod di backend tra quelli disponibili.

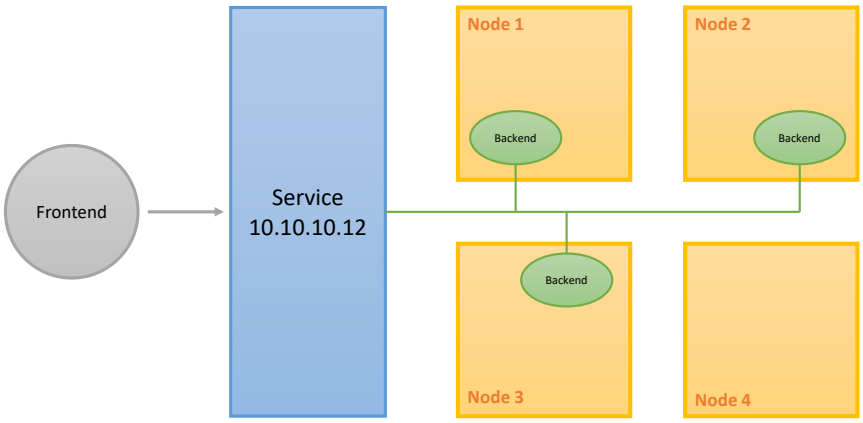


Figura 4.5: Kubernetes services example: a frontend pod and three pod backend

Esistono principalmente quattro tipo di servizi: ClusterIp, NodePort. LoadBalancer, Ingress.

#### Namespaces

Con il concetto di *namespace* è possibile creare dei cluster *virtuali* sullo stesso cluster fisico. Questo concetto è un astrazione del concetto più ampio "Ambiente". Ad esempio quando si voglio dividere in gruppi le applicazioni o controllare la visione del cluster da parte degli utenti vengono usati i namespaces. In questo modo si riesce ad avere una separazione logica delle applicazioni.

I Pod così risultano isolati solo logicamente, per l'isolamento fisico bisogna utilizzare le *Network policies*.

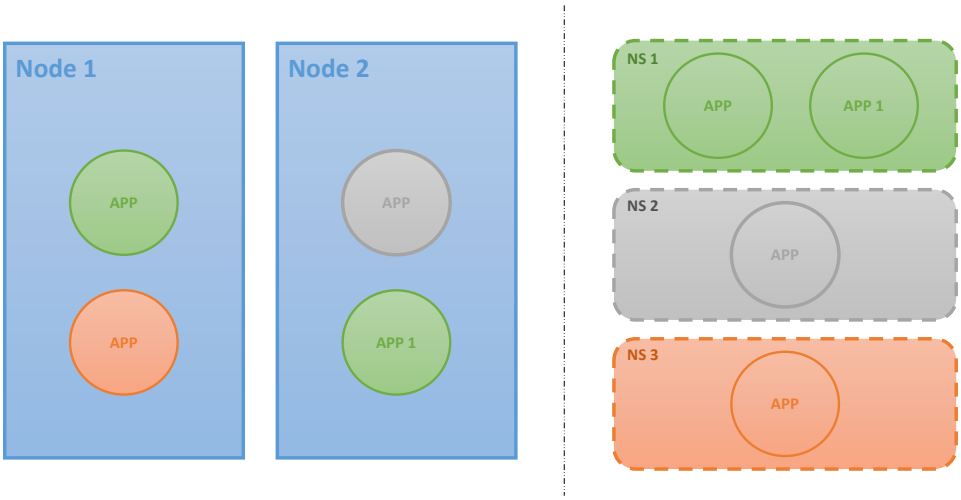


Figura 4.6: Left: Pod in cluster; Right: cluster virtual namespaces

### 4.2.2 Networking

Alla base del Networking di kubernetes c'è un concetto fondamentale: *Ogni Pod ha un IP unico nel cluster e ogni Pod deve essere accessibile/instradabile da tutti gli altri Pod.*

Tutti i contenitori all'interno del pod condividono lo stesso IP. Un enorme vantaggio quando si tratta di dover assegnare indirizzi, in quanto non si sono collisioni di IP o porte con l'host.

Le comunicazioni tra Pod sullo stesso nodo utilizzano il bridge interno all'host. Per quanto riguarda la comunicazione tra nodi differenti, ad ogni nodo viene assegnato un CIDR (intervallo di indirizzi IP) da utilizzare come PodIP, questo

garantisce l'unicità sull'intero cluster. Nel cluster verrà istanziata una "Route Table", che viene contattata nel caso in cui l'indirizzo destinazione non sia sullo stesso nodo. Questa tabella contiene tutte le impostazioni per ogni CIDR.

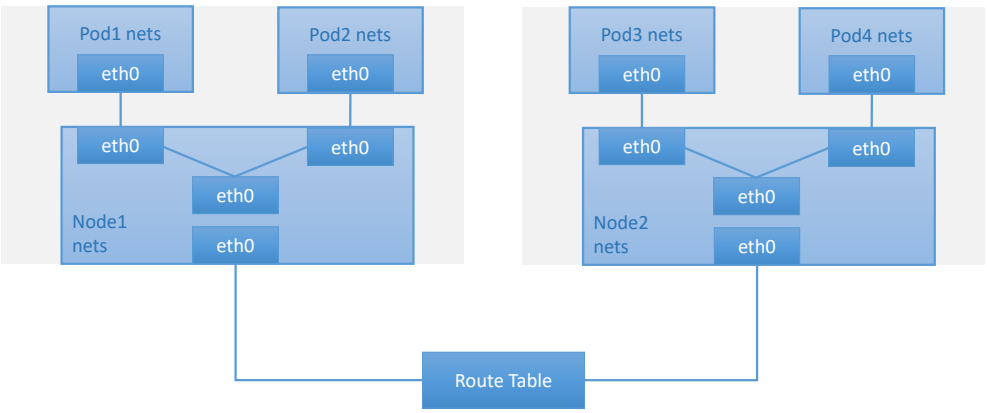


Figura 4.7: Kubernetes Networking

### 4.2.3 CRD & Operator

Kubernetes, con l'introduzione delle CRD, Custom Resources Definition, nella versione 1.7, è diventata una piattaforma estensibile in cui possono essere definite delle API. Con questo nuovo tipo di risorse gli utenti/admin possono definire delle proprie risorse.

Attraverso le CRD si possono definire delle API custom nella piattaforma Kubernetes. Queste CRD vengono utilizzate quando su vuole ad esempio ampliare i dati o creare dati custom. Non possono essere utilizzate se si vuole definire delle API con dei "verbi" differenti da quelli standard.

Tuttavia, queste CRD sono solo un mezzo per specificare una configurazione non per gestirla. Il cluster, come per gli altri oggetti, però ha bisogno di monitorare lo stato e riconciliare la risorsa. In questa situazione entra in gioco l'*Operator*.

Gli operatori sono dei controller che sono associati a delle risorse personalizzate e svolgono dei compiti sulla risorsa. Ogni volta che la configurazione della CRD varia il controller si preoccupa di riconciliare la risorsa. In realtà l'operator monitora due tipi di risorse: *primarie* e *secondarie*.

Ad esempio nel caso di un ReplicaSet, la risorsa principale è il ReplicaSet stesso, mentre la risorsa secondaria sono ad esempio i Pod che vengono creati. Quindi quando si verifica una variazione sia nella risorsa primaria che nei singoli Pod viene notificato il controller che di conseguenza si occupa di riconciliare lo stato. Risulta fondamentale identificare le risorse primarie e secondarie dell'operatore, perchè determinano il funzionamento dell'operator.

Scrivere un Operator e le relative CRD può risultare ostico ma ci sono dei tools che permettono di create lo *schafolding*.

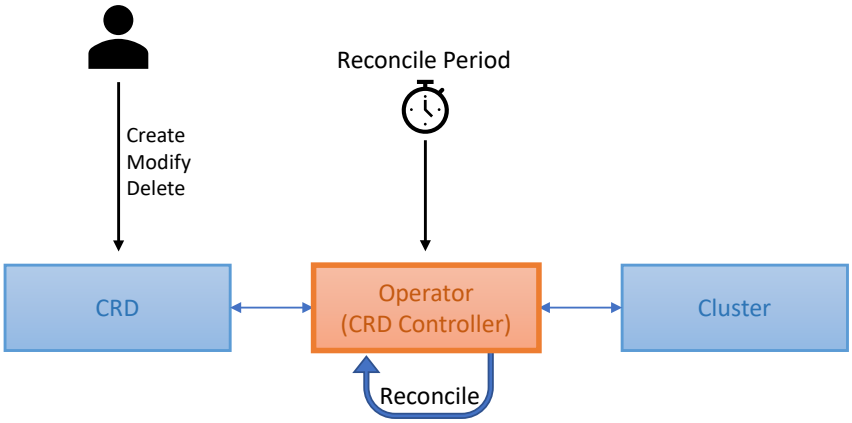


Figura 4.8: Kubernetes Operator

### 4.2.4 Admission Webhook

Un Admission WebHook è un pezzo di codice che intercetta le richieste al server API Kubernetes prima della persistenza dell'oggetto, ma dopo che la richiesta è stata autenticata e autorizzata. Ci sono due tipi di Admission WebHook: *validating* e *mutating*.

I controller di tipo "mutating" possono modificare gli oggetti che vengono ammessi, quello di tipo "validating" possono accettare o validare la richiesta. Se uno dei controller rifiuta la richiesta, la richiesta viene respinta e restituito un errore all'utente.

La Figura 4.9 mostra il percorso della richiesta.

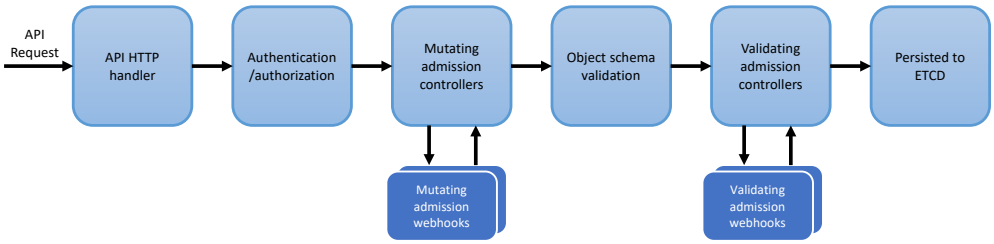


Figura 4.9: Kubernetes Admission WebHook

## 4.3 RabbitMQ

RabbitMQ è un message broker open-source che implementa il protocollo *Advanced Message Queue Protocol (AMQP)* per lo scambio di dati tra processi, applicazioni e server.

All'interno di RabbitMQ possono essere definite diverse **code**, alle quali le applicazioni hanno la capacità di connettersi per scambiare messaggi. Un **messaggio** può includere qualsiasi tipo di informazione per poter sincronizzare due o più applicazioni. Questi non vengono direttamente spediti nelle code, ma devono essere gestiti dagli **exchange**. Un exchange si occupa del routing dei messaggi: dopo averli prelevati dall'applicazione producer, li smista all'interno delle code grazie all'aiuto di vari attributi contenuti al loro interno, come le *routing key*. Ecco come funziona il flusso di messaggi con RabbitMQ:

1. Il produttore pubblica il messaggio verso un exchange (ne esistono diverse tipologie, si veda la Sezione 4.3.1).

2. L'exchange, non appena ricevuto il messaggio, ha la responsabilità di gestirne il routing e può farlo sfruttando vari attributi contenuti al suo interno, come la routing key, a seconda del tipo di exchange.

3. Exchange e code devono essere associati da un *binding*. In Figura 4.10 è possibile vedere come uno stesso exchange può essere associato a più code. Questo non è un problema: gli attributi interni al messaggio permettono all'exchange di sapere a quale coda inoltrare i messaggi senza commettere errori. Se un messaggio soddisfa più di un binding, questo viene inviato in copia a tutte quelle code che ne soddisfano i criteri.

4. I messaggi restano in una coda fin quando non vengono "prelevati" da un consumatore.

5. Il consumatore adesso può finalmente gestire il messaggio ricevuto.

### 4.3.1 Tipologie di Exchange

L'idea su cui si basa il funzionamento degli exchange è quella che un producer non invia mai direttamente un messaggio alla coda, infatti molto spesso il producer non sa neanche che il suo messaggio verrà inoltrato in una coda. Egli può solo inviare il messaggio ad un exchange. Esistono numerosi tipi di exchange, ognuno di essi viene utilizzato in contesti ben diversi. Tra questi troviamo: topic, direct, headers e fanout.

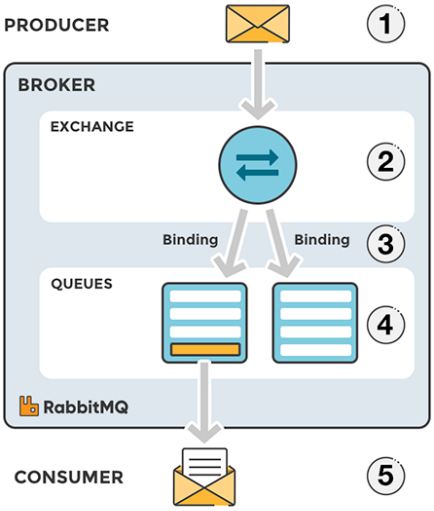


Figura 4.10: Flusso di messaggi in RabbitMQ (Fonte: CloudAMQP [9]).

- **direct**: i messaggi vengono consegnati alle code in base alla routing key specificata nel messaggio. Un exchange di tipo direct consegna i messaggi a tutte quelle code che hanno una *binding key*[1], verso quel exchange, esattamente uguale alla routing key contenuta nel messaggio. Su RabbitMQ esiste un exchange predefinito di tipo direct, che prende il nome di `amq.direct`.

- **default**: è un exchange predefinito di tipo direct senza alcun nome, cui generalmente si fa riferimento con una stringa vuota. Un messaggio che viene inviato a questo exchange verrà consegnato alla coda il cui nome è uguale a quello della routing key. Si tratta di un binding implicito, in quanto tutte le code vengono automaticamente collegate a questo exchange.

- **topic**: effettua il routing in base alle parole chiave inserite nella routing key e nelle binding key. La routing key deve essere una lista di parole separate da un punto; al suo interno possono essere specificate anche delle *wildcard*, che si esprimono con i simboli "*" e "#". L'asterisco specifica che in quella determinata posizione va bene qualsiasi parola presente nella binding key.

- **fanout**: questo exchange copia e inoltra il messaggio ricevuto a tutte le code ad esso collegato. La routing key viene semplicemente ignorata, tutte le code connesse vanno bene. Non ha quindi bisogno di alcun meccanismo di matching e quello che viene scritto nella routing key dal produttore non ha

[1] La *binding key* è una stringa di testo che lega una coda ad un determinato exchange e viene indicata in fase di definizione di un binding.

nessuna valenza. Questo exchange risulta essere molto utile e veloce nel caso in cui lo stesso messaggio deve essere trasmesso a una o più code nelle quali i consumatori processano i dati ricevuti in maniera indipendente. Ne è presente uno predefinito che prende il nome di `amq.fanout`.

- **headers**: questa tipologia è molto simile a quella del *topic*, ma sfrutta gli argomenti presenti sull'header del messaggio invece della routing key. Un messaggio viene inoltrato in una determinata coda se gli argomenti dell'header combaciano con quelli del binding. È quindi possibile avere più argomenti. Uno in particolare, che prende il nome di `x-match`, indica se a combaciare siano tutti gli argomenti o va bene anche soltanto uno di essi. Può avere due valori differenti: *all* (di default) oppure *any*. Se nulla viene specificato, allora tutti gli argomenti saranno obbligatori; con la presenza di *any* ne basta soltanto uno per dare l'ok al sistema di matching. Anche in questo caso è possibile trovarne uno predefinito: `amq.headers`.

### 4.3.2 Management

Il sistema di gestione, chiamato RabbitMQ management, fornisce un'interfaccia user-friendly che permette di monitorare e gestire il server RabbitMQ direttamente da un browser web. Code, connessioni, exchange, canali e molto altro possono essere gestiti direttamente da qui. È possibile anche monitorare il rapporto di messaggi inviati e ricevuti oppure scambiare messaggi manualmente attraverso l'interfaccia web. Il RabbitMQ management è di fatto un plugin che sfrutta le API HTTP di RabbitMQ fornendo informazioni fondamentali in caso di debug o controllo del sistema.

### 4.3.3 Federation

Il plugin di federazione permette di trasmettere messaggi tra broker senza richiedere il clustering. Questo plugin permette di federare exchange e code. Una coda o exchange federato può ricevere messaggi da uno o più *upstream* (exchanges remoti e code su altri brokers). La federazione connette gli upstreams usando un client RabbitMQ. [10]

Semplicemente i messaggi pubblicati su un upstream exchange sono copiati sull'exchange federato e pubblicato direttamente su esso.

In Figura 4.12 sono descritti due exchanges federati con un publisher e un consumer connessi allo stesso broker. Entrambi i consumers ricevono i messaggi pubblicati da tutti i publishers.

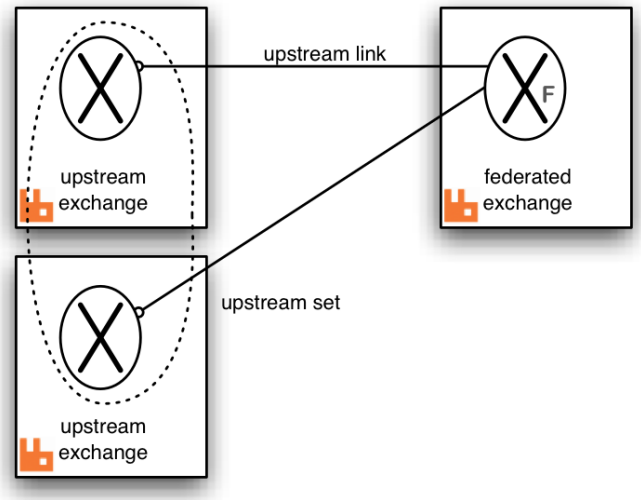


Figura 4.11: Federation

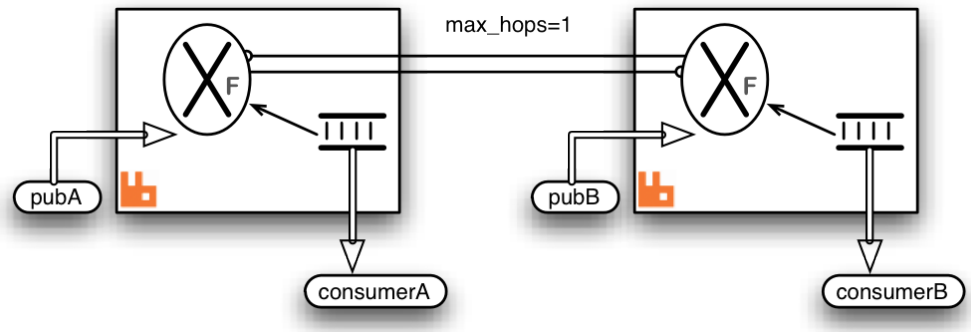


Figura 4.12: Pair Federated Exchange

# Capitolo 5

# Il sistema di orchestrazione DRONE

In questo capitolo viene presentata l'architettura del sistema Drone, *Distributed Resources Offload at Network Edge*, un sistema di orchestrazione distribuito sviluppato presso il Politecnico di Torino che costituisce la base su cui è stato sviluppato il lavoro di tesi. La soluzione presentata e sviluppata in questo elaborato si propone di utilizzare questa tecnologia nell'ottica di una orchestrazione multi tentant situata all'edge della rete.

## 5.1 Panoramica di Drone

Il paradigma emergente di Edge Computing ha consentito ai service provider di fornire una grande varietà di nuove applicazioni, che beneficiano della presenza di storage e servizi informatici ai margini della rete e di una minore latenza verso gli utenti finali. Inoltre, le tecnologie di virtualizzazione consentono l'isolamento tra diverse applicazioni che possono essere eseguite simultaneamente su sezioni separate di un'infrastruttura fisica condivisa. Negli ambienti basati su cloud, la distribuzione dei servizi viene spesso delegata a un componente centralizzato, denominato Orchestratore, che di solito sfrutta una politica adatta a tutti (one-size fits-all policy) (ad esempio, risparmio energetico, numero di nodi utilizzati, bilanciamento del carico) a decidere (i) dove posizionare i componenti del servizio, (ii) quante risorse devono essere assegnate a ciascuna di esse e (iii) l'insieme di metriche/eventi che segnalano che il servizio deve essere riprogrammato (ad es. un aumento imprevisto del carico).

Tuttavia, tale approccio centralizzato può essere non ottimale o non applicabile ai margini della rete. In primo luogo, tale ambiente può essere caratterizzato da alti tassi di abbandono, cambiamenti imprevedibili nella topologia della rete e persino partizioni di rete temporanee; questo può favorire approcci di orchestrazione

distribuiti contro un orchestratore centralizzato, che potrebbe non essere nemmeno raggiungibile. In secondo luogo, essendo un ambiente edge si può presupporre che non tutti i nodi siano sotto lo stesso dominio applicativo.

Il coordinamento di una pletora di applicazioni, ognuna delle quali presenta politiche e approcci di implementazione diversi, senza fare affidamento su un orchestratore centralizzato, mette in luce diverse sfide. In che modo diversi processi, ciascuno con obiettivi e politiche differenti, convergono verso una gestione ottimale delle risorse a livello globale su un'infrastruttura edge? Come possiamo evitare le violazioni delle politiche globali o i vincoli di flessibilità di diverse applicazioni coesistenti? Come possiamo garantire la convergenza verso un accordo di allocazione delle applicazioni distribuito e l'ottimizzazione delle prestazioni?

Per rispondere a queste domande è stato sviluppato DRONE (*Distributed Resources Offload at Network Edge*), un algoritmo distribuito di assegnazione e orchestrazione delle applicazioni. DRONE sfrutta un meccanismo ad aste per allocare un insieme di applicazioni, senza la necessità di un orchestratore centralizzato.

## 5.2 Funzionamento dell'algoritmo

Su ciascun nodo si esegue un agente DRONE che esegue, quando richiesto, una fase di orchestrazione in cui viene assegnato un "voto" che rappresenta un coefficiente di "desiderio" con cui il suddetto nodo vuole eseguire il job. Dopo che ogni nodo consegna il proprio vettore di voti, il voto finale per ogni applicazione viene aggiornato, secondo il processo elettorale distribuito, al più alto.

A Fase di Votazione:

Dopo l'inizializzazione del vettore delle applicazioni, ciascun agente di un nodo hosting potenziale controlla le risorse disponibili in quel momento, controllando anche cosa si sta già eseguendo, per verificare che può soddisfare l'applicazione. Se non è possibile la fase di votazione termina, al contrario se è possibile DRONE propone un "voto" per ogni applicazione. Dopo questa fase il vettore dei vincitori viene riempito e scambiato con il vicinato.

B Fase di Consenso Distribuito:

In questa fase, ricevuti i vettori dei vincitori dai vicini, i nodi usano una strategia di consenso per convergere a un vettore di vincitori finale. Il consenso per ogni nodo è attuato ricevendo i voti dei propri vicini e comparandoli con il proprio vettore. Il nodo host valuta tutti i voti per ogni applicazione e compone un vettore di vincitori a seconda di chi ha proposto il voto maggiore.

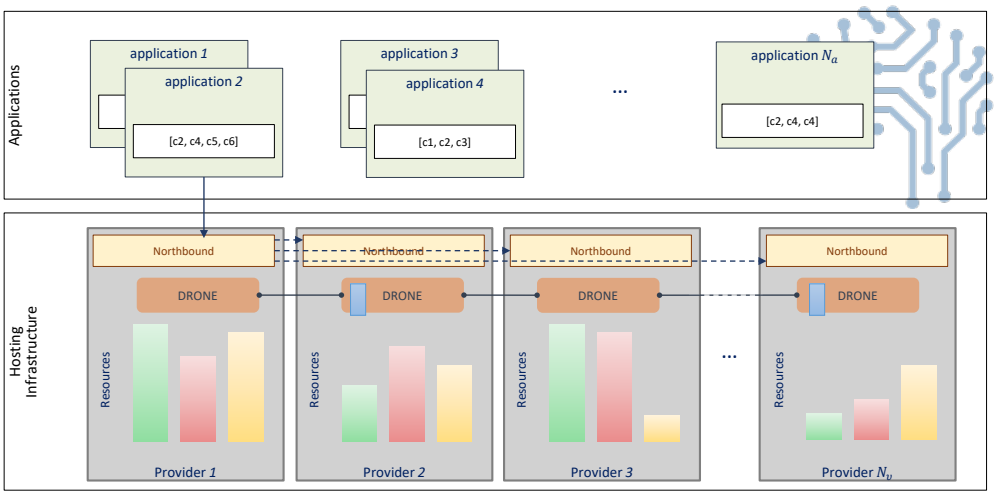


Figura 5.1: DRONE Orchestration

## 5.3 Conclusione

Questo algoritmo di orchestrazione distribuita risolve il problema di dover basarsi su una centralizzazione e del partizionamento in modo ottimale di un pool di applicazioni tra più nodi edge. DRONE consente a tali applicazioni di coesistere su un'infrastruttura condivisa attraverso un accordo dinamico.

# Capitolo 6

# Architetture dell'orchestratore in K8S

In questo capitolo viene presentata l'architettura del sistema di orchestrazione proposto per affrontare il problema descritto nel Capitolo 2. Utilizzando queste architetture sono state realizzate due implementazioni prototipali di un orchestrazione distribuita multi-cluster descritte nel prossimo capitolo [Capitolo 7]. Durante il

lavoro di tesi sono state ideate due soluzioni:

- *prima architettura*, una soluzione esterna a Kubernetes e funzionante sia in ambiente interno a un cluster sia in uno multi-cluster. Questa prima architettura è stata realizzata per mostrare il funzionamento e le potenzialità dell'orchestratore distribuito DRONE in un ambiente multi-provider;

- *seconda architettura*, basata sull'idea della prima ma innestata come applicazione interna in Kubernetes e applicabile in ambienti multi-cluster con cluster appartenenti a tenant differenti. Sviluppata successivamente visto il corretto funzionamento dell'algoritmo.

Rispondiamo subito a una domanda fondamentale, *Dove viene posto DRONE in ambiente multi-cluster?* DRONE, in una visione generale ad alto livello, si pone sopra al cluster Kubernetes e gestisce lo scheduling dei jobs.
In Figura 6.1 viene mostrato come DRONE dovrebbe comportarsi rispetto ai cluster.

## 6.1 Prima soluzione proposta

Nella prima soluzione proposta i vari nodi possono essere visti come nodi di un singolo cluster oppure interi cluster comunicanti fra loro. Grazie all'utilizzo di un orchestratore distribuito, i nodi (singole macchine o interi cluster) non devono

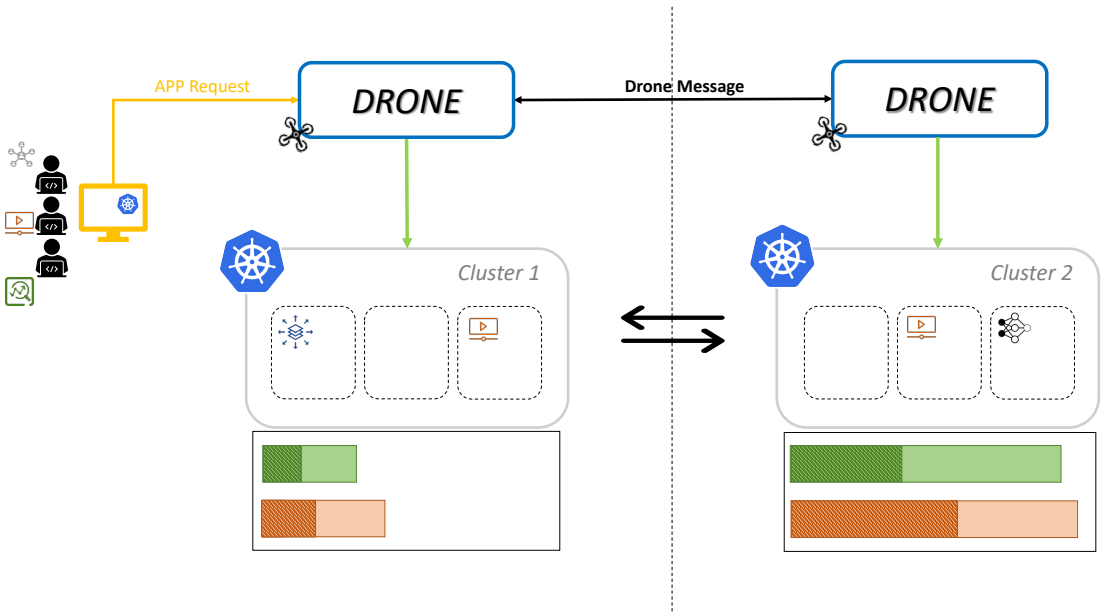


Figura 6.1: Architettura utilizzo DRONE multi-cluster

"conoscersi" ne devono contattare un controllore centralizzato che prende decisioni per conto loro. Questo concetto permette a questa architettura di porsi in un ambito *multi-tenant*.

L'obiettivo principale di questa architettura è dimostrare che è possibile, utilizzando l'algoritmo DRONE attivo su tutti i nodi coinvolti, prendere decisioni distribuite per eseguire dei jobs. Questo permette di condividere servizi su diversi nodi/cluster senza che vengano esportate informazioni sensibili riguardanti il nodo.

In Figura 6.2 è illustrata l'architettura basilare di una comunicazione tra nodi connessi sotto una rete mentre in Figura 6.3 è illustrato lo scenario su più cluster in cui sono attive politiche di offloading. Prendiamo come esempio il la Figura 6.2 in cui un utente, connesso al nodo 3, vuole eseguire una applicazione. L'utente, ignaro del sistema, da il comando di esecuzione. Un *controller*, istanziato sullo stesso nodo, intercetta la richiesta e la veicola al sistema DRONE che, contattando gli altri agenti DRONE presenti su altri nodi, decreta dei "vincitori" per i servizi richiesti. A questo punto l'applicativo o parte di esso viene istanziato sui nodi e viene aperto un canale di comunicazione tra i vari servizi e l'utente.

### 6.1.1 Architettura dettagliata del sistema

In questa sezione viene descritta nel dettaglio l'architettura del sistema di orchestrazione in tutte le sue funzionalità: struttura di un singolo nodo, intercettazione

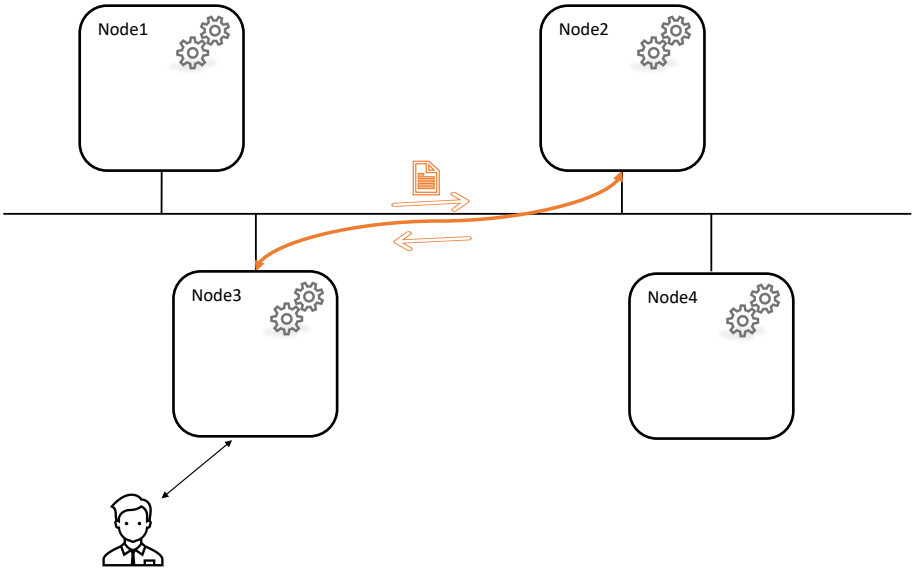


Figura 6.2: Esempio di comunicazione tra nodi connessi ad una rete

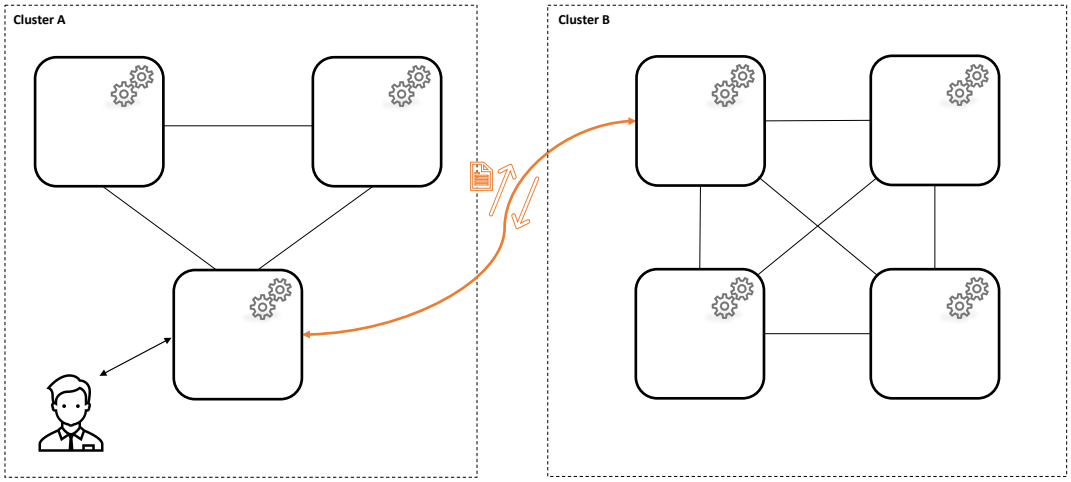


Figura 6.3: Esempio di comunicazione tra cluster connessi

della richiesta, orchestrazione dell'applicazione desiderata e deploy effettivo dei jobs sui nodi vincitori. Nonché la lettura delle risorse disponibili del nodo.

*Nodes.* In Figura 6.4 viene illustrata la struttura dei nodi. Essi sono configurati come descritto nella sezione YY, quindi dispongono di un broker RabbitMQ federato con altri broker su altri nodi, di un DRONE-Controller e di un istanza dell'orchestratore DRONE. In aggiunta a ciò ci sarà installato docker per permettere l'avvio di applicazioni tra macchine diverse. Si può inoltre pensare a un nodo come

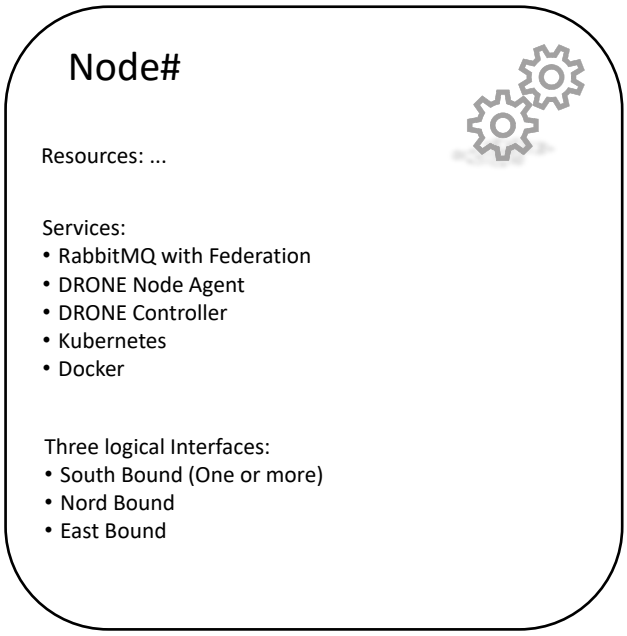


Figura 6.4: Struttura del singolo nodo in questa architettura

un "box" con tre interfacce verso l'esterno:

- *North-Bound*, interfaccia verso l'utente per richieste di nuovo applicazioni;
- *South-Bound*, per comunicare con il servizio di deploy, come ad esempio Kubernetes;
- *East-Bound*, per una comunicazione orizzontale tra nodi.

*Messages.* Sono stati definiti cinque principali tipi di messaggio per la comunicazioni tra le varie parti dell'architettura analizzata. Questi messaggi interessano le comunicazioni tra *DRONE - DRONE*, tra *DRONE-Controller - DRONE* e tra *User - DRONE-Controller*.

- *Advertisement message*, messaggio utilizzato per scambiare informazioni sulle nuove richieste di esecuzione/cancellazione di un Applicazione tra tutti i controller e i vari agenti DRONE;
- *UserRequest message*, messaggio inviato dall'utente che desidera avviare una nuova Applicazione;
- *Solution message*, messaggio di soluzione inviato da DRONE al DRONE-Controller;
- *Resources message*, messaggio utilizzato per scambiare informazioni sulle risorse consumate sul nodo;
- *DRONE message*, messaggio scambiato tra gli agenti DRONE per raggiungere l'agreement.

L'implementazione e la struttura di questi messaggi verrà discussa in seguito nel Capitolo di implementazione [Capitolo 7].

Ora, definita la struttura del singolo nodo e i vari messaggi che vengono scambiati dai vari componenti, possiamo dedicarci a una spiegazione ad alto livello della struttura dell'architettura sviluppata, focalizzandoci sul funzionamento dall'arrivo di una richiesta utente al momento dell'effettivo *deploy* sul nodo.

### 6.1.2 In-Cluster Orchestrator

In Figura 6.5 viene illustrata l'architettura della soluzione con i messaggi scambiati dalle varie parti. In questa figura viene mostrata la struttura per quanto riguarda l'orchestrazione distribuita interna ad un cluster Kubernetes. In seguito verrà esposta la medesima soluzione anche in ambiente *multi-cluster* (seppur in minor dettaglio).

Come si può notare è stata posta una numerazione sui messaggi per aiutare il lettore a seguire il flusso di esecuzione. Quando verrà richiesta l'esecuzione di una applicazione da parte di un utente, un messaggio di *UserRequest* [1] arriverà al DRONE-Controller che ha il compito di validare la richiesta, avvisare tutti gli altri DRONE-Controllers e cominicare agli agenti DRONE. Questo viene fatto con il messaggio di *Advertisement* [2]. Una volta che l'advertisement è stato distribuito a tutti, i DRONE-Controllers salveranno tra le applicazioni pendenti la richiesta mentre gli agenti DRONE faranno partire l'orchestrazione. Terminata la fase di orchestrazione ogni agente comunicherà al proprio controller la soluzione raggiunta, *Resources message* [3]. Ricevuta la soluzione della richiesta, il DRONE-Controller, dopo aver verificato i jobs della soluzione che dovrà effettuare, si occuperà del deploy in Kubernetes tramite le API esposte dal api-server di k8s [4]. Quando

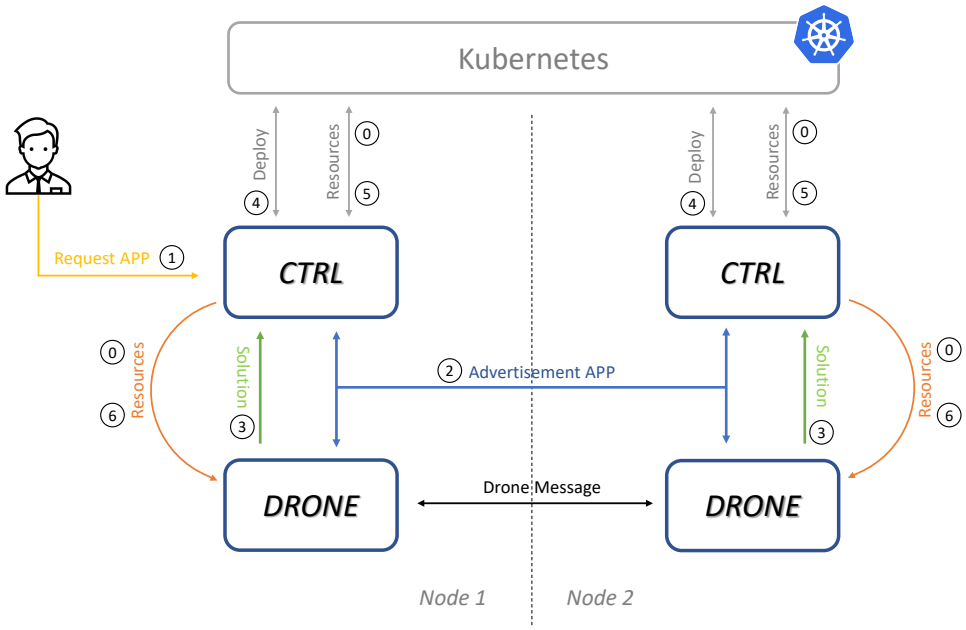


Figura 6.5: Architettura in-cluster con i messaggi

il deploy verrà completato il DRONE-Controller si impegnerà a scoprire le nuove risorse utilizzabili del nodo [5] ed informare DRONE per le future richieste [6].

Riassumendo i compiti del componente DRONE-Controller sono:

- *Attesa di nuove richieste*, ascolta le nuove richieste di deploy da parte degli utenti;
- *Annuncio dei nuovi jobs*, annuncia con un messaggio di *advertisement* alle altre istanze DRONE e Controller la nuova applicazione da eseguire;
- *Attesa delle soluzioni*, ascolta le soluzioni che il componente DRONE comunica dopo la fase di orchestrazione;
- *Deploy dei jobs*, comunica con Kubernetes per effettuare il deploy;
- *Gestione risorse*, esegue una lettura delle risorse da K8S e le comunica al componente DRONE;

### Criticità soluzione

Questa prima soluzione, chiamata *in-cluster*, presenta due problematiche principali:

- *Mancanza di integrazione con Kubernetes*, DRONE e CONTROLLER sono programmi "standalone" eseguiti su tutti i nodi del cluster che comunicano con le API di Kubernetes al fine di far comunicare i vari container sui nodi;

- *Utilizzo di DRONE in-cluster*, L'utilizzo di un algoritmo distribuito, come DRONE, all'interno di un cluster non è molto interessante in quanto non si sono trovati use cases degni di nota.

La seconda verrà risolta con lo sviluppo dell'architettura mostrata nel prossimo paragrafo, mentre la prima si discuterà nella sezione 6.1.4.

### 6.1.3 Multi-Cluster Orchestrator

In questa parte viene esposta brevemente la struttura vista nella precedente sezione ma utilizzata in ambiente multi-cluster.

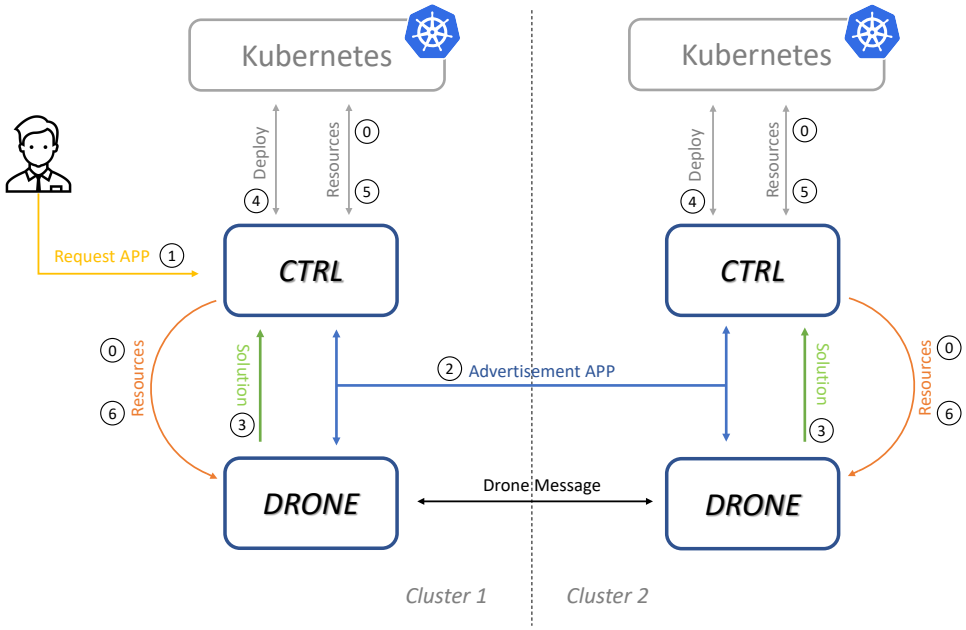


Figura 6.6: Architettura multi-cluster con messaggi

Come mostrato in Figura 6.6 la struttura della soluzione varia in misura minima rispetto all'architettura *in-cluster*. Per quanto riguarda il funzionamento interno del messaggi che vengono scambiati e i compiti delle varie componenti in gioco sono identici alla versione *in-cluster*.

Le modifiche apportate alla soluzione *in-cluster* sono tre:

- La principale sarà sulla gestione delle risorse che in questo caso non saranno più analizzate le risorse del singolo nodo ma dell'intero cluster, questa particolarità verrà illustrata nel Capitolo 7 in cui vengono trattate le implementazioni.;

- DRONE in questo ambiente deciderà solo il cluster destinazione del job e non il nodo;

- DRONE e CONTROLLER non saranno più eseguiti su tutti i nodi del cluster ma avranno una sola istanza per cluster;

#### Criticità soluzione

Questa evoluzione, chiamata *multi-cluster*, presenta anch'essa la problematica della mancanza di integrazione con K8S e quindi dell'avere software "standalone" eseguiti sulla macchina fisica (difficilmente gestibili in un cluster K8S.

### 6.1.4 Proof-of-Concept: Avvio applicazioni in nodo remoto

L'architettura presentata precedentemente nella Sezione 6.1 è stata utilizzata per realizzare un prototipo al fine di esempio in cui vengono eseguite due differenti applicazioni su nodi connessi in rete. Più nel dettaglio sono state avviate:

- *VLC*, applicazione per la visualizzazione di video, suddivisibile in due parti (streamer e visualizzazione);
- *Applicazione di Mining*, utilizzata per mostrare l'utilizzo elevato di risorse su un nodo secondario.

Come nodi sono state create delle Virtual-Machine ad hoc con differenti risorse. Su ogni VM sono stato installate delle istanze di RabbitMQ, DRONE-Controller e DRONE. L'implementazione del DRONE-Controller e di DRONE così come la configurazione dettagliata di RabbitMQ verrà discussa in seguito nel Capitolo 7.

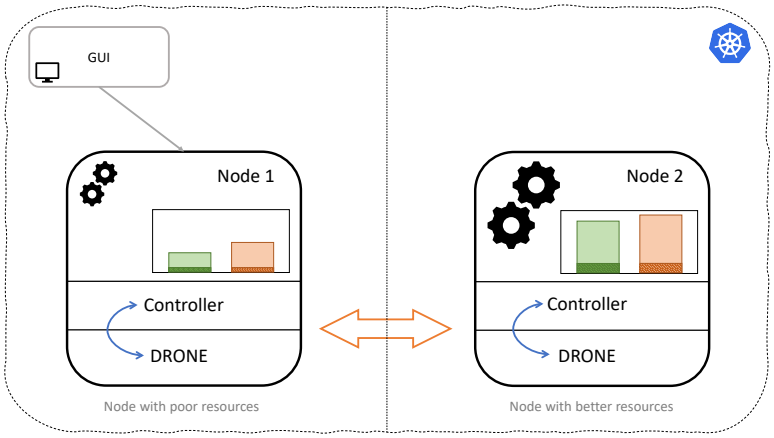


Figura 6.7: Prototipo realizzato

In Figura 6.7 è descritta l'architettura generale del prototipo realizzato. Ogni nodo (Virtual-Machine) è parte di un cluster Kubernetes. Questa scelta è stata

fatta per semplicità, come detto in precedenza lo scope di questo prototipo è dimostrare l'effettivo funzionamento dell'algoritmo, in quanto Kubernetes offre un overlay network su cui la comunicazione *pod-to-pod* e *service-to-pod* risultano già gestite.

Per realizzare questo prototipo sono state sviluppate tre componenti: DRONE-Controller, DRONE e Gui. Le prime due sono già state analizzate precedentemente. La *GUI* è stata sviluppata solo per poter dare una interfaccia grafica all'utente che desidera avviare una applicazione all'interno del cluster usando l'orchestrazione di DRONE.

Questo Proof-Of-Concept vuole essere soltanto un esempio di come sia possibile eseguire Job, o parte di essi, su nodi "federati" comunicanti. Si può pensare ai nodi come dei dispositivi situati all'edge della rete che interagiscono tra loro per creare una rete più o meno fitta di collaborazioni. Quindi lo scopo principale del prototipo è proporre una soluzione all'architettura proposta in questa prima parte del capitolo per orchestrare, senza bisogno di una centralizzazione, ed eseguire Job provenienti da un nodo di un cluster.

Con questo prototipo si è potuto dimostrare che applicazioni, avviate sul nodo "locale" (nodo con meno risorse), saturavano completamente il nodo, il quale non riusciva a soddisfare la richiesta dell'utente. Invece le stesse applicazioni migrate attraverso il sistema DRONE sul secondo nodo, molto più prestante, soddisfacevano la richiesta dell'utente lasciando il nodo "locale" libero per eseguire altri job più in linea con le proprie risorse.

## 6.2 Evoluzione della soluzione

Come già detto entrambe le soluzione proposte nelle sezioni precedenti soffrono del problema di essere composte da programmi "standalone" eseguiti su un host fisico del cluster (nella prima soluzione vengono avviati controller e drone su tutti i nodi mentre nella seconda viene avviata una singola istanza per cluster, ad esempio sull'host che copre il ruolo di master).

Questa condizione è problematica perché in un cluster gestito interamente con Kubernetes avere software standalone non è facilmente controllabile dall'esterno ed inoltre non si è compliant con K8S.
Per questi motivi si è incominciato a sviluppare una architettura completamente integrata in Kubernetes e gestibile con le API offerte; con l'obiettivo finale di avere una architettura trasparente nei confronti dell'utente che desidera eseguire uno o più jobs. Con questa nuova architettura quindi si è cercato di porsi come concorrente di *KubeFed* per un ambiente di cluster federati, evitando di avere una centralizzazione.

### 6.2.1 Architettura dettagliata del sistema

In questa sezione viene descritta nel dettaglio l'evoluzione dell'architettura del sistema di orchestrazione in un ambiente *multi-cluster* con cluster appartenenti a diversi operatori.

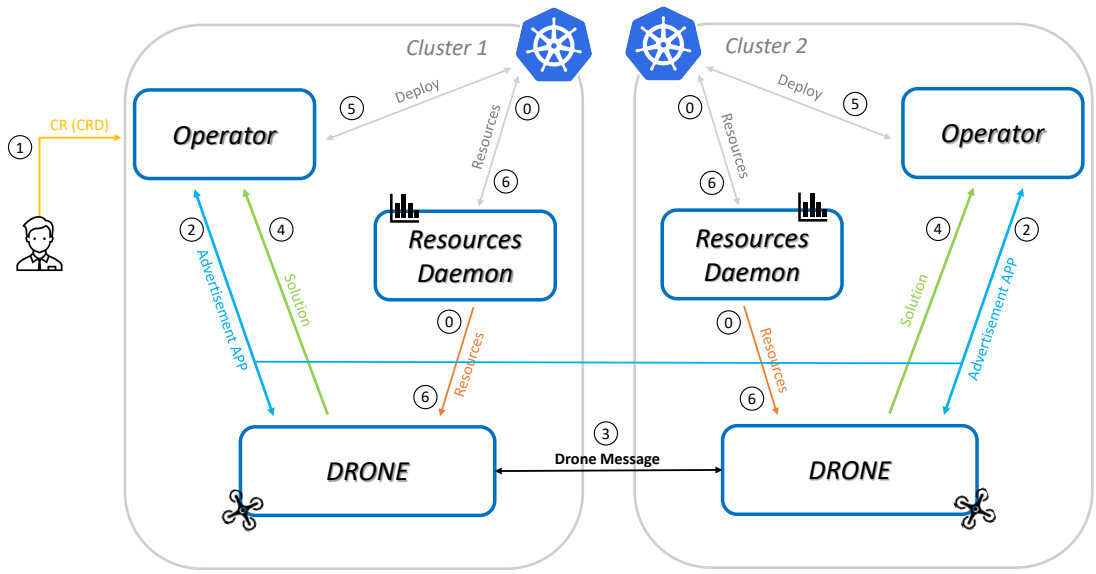


Figura 6.8: Architettura Integrata in Kubernetes

Come mostrato in Figura 6.8 il *DRONE-Controller* viene diviso in due parti: Operator e ResourcesDaemon. Mentre i messaggi scambiati tra i vari componenti rimangono i medesimi.

- *Operator.* Usando *l'Operator* con le *CRD* si riesce a creare risorse personalizzate in Kubernetes estendendo le funzionalità di base e le API. In seguito nel capitolo 7 tratteremo nel dettaglio le CRD e l'implementazione dell'Operator;
- *ResourcesDaemon.* Ha lo scopo di leggere le risorse del cluster K8S e notificarle a DRONE. Questo componente è pensato come componente custom, personalizzabile da ogni tenant in modo da fornire a DRONE le risorse che si vogliono utilizzare per la federazione. In questo modo ogni provider può essere libero di gestire il loro cluster indipendentemente.

In questa versione della architettura si è deciso di utilizzare l'operator in quanto permette di estendere le API standard di Kubernetes e permettendo di gestire un oggetto custom, CRD.

I messaggi scambiati tra i vari cluster sono il seguenti:

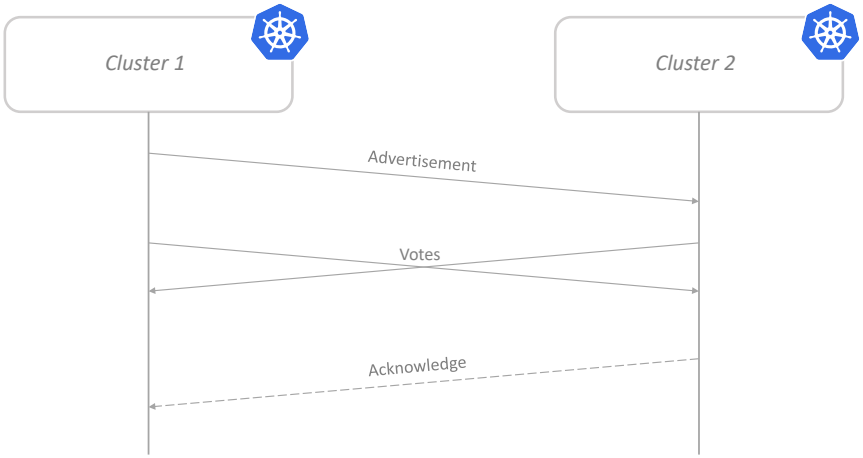


Figura 6.9: Messaggi tra Clusters

Come si può notare è stata posta una numerazione sui messaggi per aiutare il lettore a seguire il flusso di esecuzione. Quando verrà richiesta l'esecuzione di una applicazione da parte di un utente, una richiesta contenente una *custom resource* [1] arriverà all'operator che avrà il compito di validare la richiesta, avvisare tutte le altre istanze e comunicare agli agenti DRONE l'arrivo di una nuova richiesta. Questo viene fatto con il messaggio di *Advertisement* [2]. Terminata la fase di orchestrazione ogni agente comunicherà al proprio *Operator* la soluzione raggiunta,

*Solution message* [4]. Ricevuta la soluzione della richiesta, l'Operator, dopo aver verificato i jobs della soluzione che dovrà effettuare, si occuperà del deploy in Kubernetes tramite le API esposte dal api-server di K8S [4]. Quando il deploy verrà completato il *Resources Daemon* si impegnerà a scoprire le nuove risorse utilizzabili del nodo [5] ed informare DRONE per le future richieste [6].
Con questa soluzione si risolvono entrambi i problemi riscontrati nella prima versione ed inoltre vengono aggiunte funzionalità, come ad esempio la validazione della richiesta sia lato *Operator* sia lato client. Infatti con la nuova vesione di Kubernetes è stata aggiunta anche la validazione delle CRD anche lato client. Un'altra funzionalità aggiuntiva è il mantenimento dello stato della risorsa. Infatti essendo l'operator una applicazione nativa di Kubernetes dispone di una logica di controllo adatta a controllare le risorse così come fa K8S per le risorse base (Pod, Deployment, ...).

É possibile inoltre utilizzare un *Mutation WebHook* per rendere la soluzione trasparente all'utente. In questo modo l'utente effettua un deploy nel classico modo di Kubernetes. Il WabHook si pone come "filtro" prima che la richiesta di un oggetto venga creata nella base dati, in questo modo è possibile modificare la richiesta in base ad alcune caratteristiche.

# Capitolo 7

# Implementazione orchestrazione in K8S

In questo capitolo viene discussa l'implementazione dei due prototipi realizzati per l'orchestrazione interna a Kubernetes ed multi-cluster presentate nel Capitolo 6. Come già descritto nel capitolo precedente le implementazione sono state due. La prima al fine di validare il funzionamento di DRONE, la seconda per integrare DRONE in Kubernetes.

Il presente capitolo si suddivide in due macro-sezioni che analizzano le implementazioni realizzate. In queste sezioni verranno discussi sia vari elementi che compongono la soluzione sia le configurazioni dei nodi/cluster. Infine verrà discusso lo sviluppo di un *admission-webhook* per rendere trasparente la soluzione. Tutti i sorgenti e gli script per entrambe le soluzioni sono caricati sui repository *GitHub*: netgroup-polito/edge-offloading-gui[1], netgroup-polito/edge-offloading-controller[2] e netgroup-polito/dronev2[3]. Mentre le immagini Docker sviluppate sono raccolte su dei repository di DockerHub: andreamora/vlc, andreamora/mining per quanto riguarda la prima soluzione e andreamora/drone-agent, andreamora/drone-operator, andreamora/drone-daemon-resources per la seconda soluzione.

## 7.1 Primo Prototipo

Questa sezione illustra le configurazioni dei nodi e le implementazioni della prima soluzione utilizzata per validare l'utilizzo dell'algoritmo DRONE. Lo use case selezionato per provare il funzionamento dell'algoritmo è quello di applicazioni divisibili

[1]Contiene i sorgenti del componente GUI

[2]Contiene i sorgenti del componente CONTROLLER

[3]Contiene i sorgenti della seconda soluzione

in più servizi da poter istanziare in nodi differenti. Utilizzando il classico orchestratore centralizzato le due parti della stessa applicazione verrebbero distribuite tra nodi in modo da ridurre al minino la media dei carichi dei diversi nodi. Con una orchestrazione distribuita, come quella offerta da DRONE, ogni nodo può decidere se occuparsi o meno di eseguire un task e con quale peso. Di seguito quindi verrà descritto il lavoro che ha permesso di utilizzare l'algoritmo.

In questa sezione analizzeremo, inizialmente, come è stato configurato ogni singolo nodo e successivamente illustreremo nel dettaglio le principali componenti della soluzione.

### 7.1.1 Configurazione dei nodi

Questa sezione illustra la configurazione dei vari nodi utilizzati come nodi del cluster Kubernetes.

Ogni nodo è realizzato come una Virtual-Machine ed è stato configurato con una istanza del message broker RabbitMq installate e facente parte di un cluster Kubernetes. Inoltre, in ogni nodo saranno eseguite le istanze del CONTROLLER e di DRONE.

#### Configurazione RabbitMQ

Tutte le comunicazioni tra i vari componenti sono state implementate utilizzando il broker RabbitMQ. Per installarlo e configurarlo all'interno delle VM si sono seguiti vari passi. Nelle versioni successive dell'architettura RabbitMQ verrà istanziato come container Docker all'interno del cluster Kubernetes [7.2.1].
Per prima cosa occorre installare rabbitMq sull'host desiderato con il comando

```
$ apt install rabbitmq-server
```

Inoltre questo soluzione utilizza la federazione per la comunicazione tra broker, in questo modo si è riuscito a rendere distribuito l'utilizzo del broker. Per questo motivo (utilizzo della federazione tra broker) il *RabbitMQ server* deve essere configurato con un *user* differente da quello di "default", delle *policies* e *federation upstream*.

Quindi si è proceduto a installare i plug-ins desiderati

```
$ /usr/sbin/rabbitmq-plugins enable rabbitmq_federation
rabbitmq_federation_management
```

Per la configurazione della federazione tra differenti server RabbitMq bisogna quindi eseguire alcuni comandi su ogni nodo che ospita un'istanza.

```
# Add new User [username password]
sudo rabbitmqctl add_user drone drone
# Set User roles
sudo rabbitmqctl set_user_tags drone administrator
# Set permissione for user created
sudo rabbitmqctl set_permissions -p / drone ".*" ".*" ".*"
```

Dopo aver eseguito questi comandi su ogni nodo desiderato occorre federare i vari broker (esempio broker0 vuole federarsi con broker1 e 2)

```
 # Set federation upstream for broker1
 sudo rabbitmqctl set_parameter federation-upstream
rabbit1 '{"uri":"amqp://drone:drone@10.0.0.1"}'
 # Set federation upstream for broker2
 sudo rabbitmqctl set_parameter federation-upstream
rabbit2 '{"uri":"amqp://drone:drone@10.0.0.2"}'
 # Create a set of federated broker with name
 sudo rabbitmqctl set_parameter federation-upstream-set
drone '[{"upstream":"rabbit1"},{"upstream":"rabbit2"}]'
 # Set policy for upstream set created
 sudo rabbitmqctl set_policy --apply-to exchanges
federate-drone drone-exchange
'{"federation-upstream-set":"drone"}'
```

Questi comandi configurano due cose in particolare:

- *D*efinisce due nodi di upstream (rabbit1 e rabbit2) e assegna a loro il corretto indirizzo. Poi viene creato un set chiamato "drone" che contiene questi due nodi,

- *C*rea una policy sugli exchanges, chiamata "federate-drone", che selezione gli exchanges chiamati "drone-exchange" e li federa per l'*upstream-set* "drone".

Inoltre è stata attivata l'interfaccia grafica con il plugin *rabbitmq_management* disponibile su http://localhost:15672. É stato redatto uno script contenente tutti i seguenti comandi per una configurazione più veloce dell'ambiente. Le informazioni necessarie per lo script sono contenute in un file di configurazione modificabile per ogni host.

### Struttura comunicazione RabbitMQ

Il broker in questione è stato usato per tutte le comunicazione del progetto. In questo paragrafo mostreremo come i vari componenti comunicano tra di loro utilizzando le *code* e gli *exchanges*.

In Figura 7.1 viene descritta la struttura di come sono state definite le componenti di RabbitMQ. Si possono notare "n" code e un exchange federato.

Il messaggio di Advertisement viene consumato da due componenti. Sono state definite due code legate all'exchange federato. Queste due code oltre ad essere legate all'exchange sono vincolate a ricevere i messaggi con routing key *app-advertisement*. In questo modo l'exchange, per ogni messaggio che contiene nell'header quella key, direziona i messaggi su tutte le code legate con quella key.

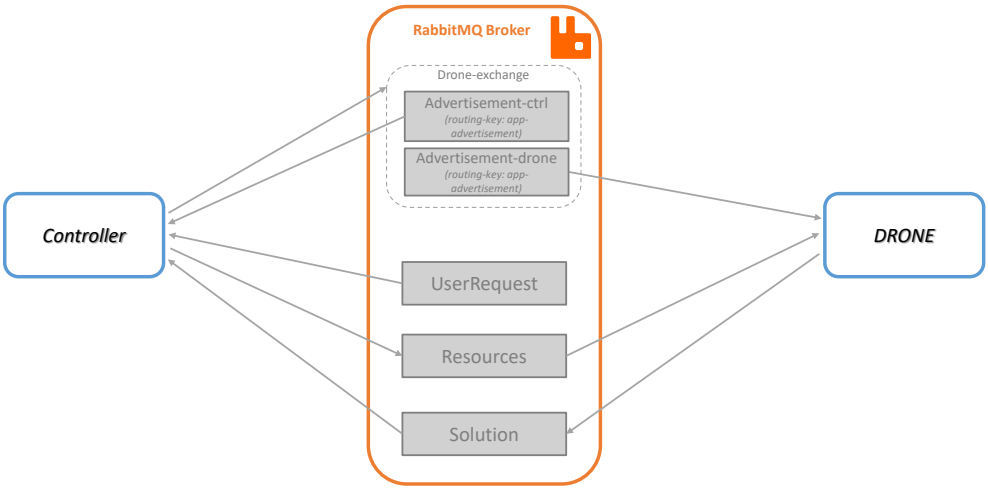


Figura 7.1: Comunication RabbitMQ

La necessità di avere i messaggi duplicati su due code è dovuta al fatto che sul nodo sono presenti due applicazioni che fungono da consumatori del messaggio di advertisement: il DRONE-Controller e DRONE. In questo modo si evita che le due applicazioni si sottraggano messaggi a vicenda. Per indirizzare i messaggi su tutti i broker RabbiMQ è stata utilizzata la *federation* una funzionalità stabile e affidabile che offre RabbitMQ per fare comunicare diversi broker remoti.

#### Kubenetes

Kubernetes come detto in precedenza nei capitoli precedenti in questa prima architettura era stato utilizzato solo al fine di semplicità, in modo da avere una comunicazione tra container già implementata. Tutti i nodi di questa soluzione fanno parte di un cluster K8S, quindi sono dotati di indirizzo ip e nome univoco nel cluster. Il cluster è stato creato utilizzando vagrant, gestore di macchine virtuali open source. I codici utilizzati in questa parte sono stati ignorati ma possono essere visionati sul repository *GitHub*: netgroup-polito/edge-offloading-controller[4].

### 7.1.2 Messaggi

In questa sezione vengono definite le strutture dei messaggi scambiati dai vari componenti delle soluzioni proposte.

Sono stati definiti cinque tipi principali di messaggi per questa prima architettura di validazione dell'algoritmo:

[4]Contiene i sorgenti del componente CONTROLLER

- *Advertisement message*, messaggio utilizzato per notificare a tutti gli agenti DRONE e ai DRONE-Controller l'arrivo di una nuovo richiesta di deploy. Questo messaggio verrà scambiato tra le istanze di DRONE e DRONE-Controller. Successivamente verrà spiegato come viene creato e utilizzato questo tipo di messaggio;
- *Resources message*, messaggio utilizzato per sincronizzare l'agente DRONE alle nuove risorse disponibili nel nodo;
- *Solution message*, messaggio utilizzato dall'agente DRONE per comunicare la soluzione aggiornata al DRONE-Controller;
- *UserRequest message*, messaggio utilizzato daal'utente per richiedere l'esecuzione di una nuova applicazione;
- *Drone message*, messaggio scambiato tra gli agenti DRONE per schedulare dei jobs.

### Advertisement Message

Questo messaggio viene utilizzato per comunicare l'arrivo della richiesta di una nuova applicazione da eseguire o la cancellazione di una già esistente. Nel messaggio sono presenti le informazioni necessarie per eseguire il job e soprattutto per poterlo distribuire su nodi differenti da quello d'origine.

Questo tipo di messaggio può essere di tre tipi; un primo tipo **ADD**, per notificare a tutti i componenti una nuova applicazione, e un secondo tipo **DEL**, per notificare la rimozione di una applicazioni, e un terzo tipo **MOD**, per la modifica del job eseguito.

```
class Adv_type:
    ADD = 'ADD'          # Add new application
    DELETE = 'DEL'       # Remove aplication
    MODIFIED = 'MOD'     # Modify application
```

Listing 7.1: Advertisement Type

La struttura del messaggio è composta da quattro campi principali come mostrato sotto:

```
{
    "app_name": String,
    "base_node": String,
    "type": Adv_type,
    "components": [ ]
}
```

Listing 7.2: Advertisement Message Structure

Ogni campo è descritto come segue:

- *app_name*, identifica il nome dell'applicazione da eseguire;
- *base_node*, identifica il nome del nodo originale da cui è stato richiesto l'avvio di una nuova applicazione;
- *type*, identifica il tipo del messaggio come spiegato sopra;
- *components*, è un vettore che contiene le componenti che compongono le applicazioni e i loro parametri.

Una applicazione può essere suddivisa in più servizi, per questo si è pensato di aggiungere un vettore di componenti della applicazione. Ogni componente deve avere delle informazioni di base come un **name**, nome del componente, e una funzione che ne esegue le funzionalità. Inoltre, dato l'ambiente distribuito che stiamo considerando, possono essere indicati altri parametri che permettono la completa configurazione del task su differenti nodi.

```
{
    "name": String,
    "function": {
        "image": String,
        "resources": {
            "memory": Number,
            "cpu": Number
        }
    },
    "parameters": { },
    "boot_dependencies": [ ],
    "nodes-blacklist": [ ],
    "nodes-whitelist": [ ]
}
```

Listing 7.3: Component Structure

La funzione che identifica il componente comprende due campi: il primo, **image** descrive l'immagine docker da istanziare, il secondo, **resources**, descrive quante risorse quel determinato componente andrà a consumare una volta avviato sul nodo. I campi **parameters** e **boot_dependencies** sono campi utilizzati rispettivamente per poter definire alcuni parametri del componente e per definire l'ordine si esecuzione qualora ci fossero più componenti da avviare che hanno dipendenze l'uno

dall'altro. Infine i due campi **nodes-blacklist** e **nodes-whitelist** vengono utilizzati, rispettivamente, per definire una restrizione su nodi (ad esempio in cui non si voglia eseguire il componente) e per definire un set di nodi dove è preferibile avviare il job.

### Resources Message

Questo messaggio viene utilizzato per sincronizzare l'utilizzo di risorse del nodo con le informazioni/metriche che DRONE utilizza per schedulare i jobs. Nel messaggio sono presenti le informazioni necessarie a DRONE per prendere decisioni sulla destinazione dei vari tasks.

La struttura del messaggio è composta da tre campi principali come mostrato sotto:

```
{
    "sender": String,
    "node_resources": {
        "memory": Number,
        "cpu": Number
    },
    "timestamp": String
}
```

Listing 7.4: Resources Message Structure

Il campo **sender** identifica il nodo che ha inviato il messaggio di sincronizzazione delle risorse. Il **timestamp** descrive il tempo in cui è stato creato e mandato il messaggio (formato iso). Infine il campo più importante è **node_resources** che definisce le risorse aggiornate del nodo, in questo caso vengono considerate solo *cpu* e *memoria* ma possono essere aggiunte altre metriche.

Questo messaggio viene inviato dal DRONE-Controller a DRONE all'inizio per definire le risorse iniziali in modo che se il nodo considerato non possa eseguire jobs non partecipi all'orchestrazione. Inoltre viene inviato ogni volta che le risorse del nodo si aggiornano.

### Solution Message

Questo messaggio viene utilizzato da DRONE per comunicare la soluzione aggiornata al DRONE-Controller. Nel messaggio sono presenti le informazioni necessarie al DRONE-Controller per eseguire i jobs a lui destinati e per conoscere dove sono stati eseguiti gli altri.

La struttura del messaggio è composta da quattro campi principali come mostrato sotto:

```
{
    "sender": String,
    "local_offloading": [ ],
    "overall_offloading": { },
    "timestamp": String
}
```

Listing 7.5: Solution Message Structure

Come illustrato sopra sono presenti quattro campi nel messaggio delle soluzioni. Il primo campo è **sender** utilizzato per identificare il nodo da cui stiamo inviando le soluzioni. Il **timestamp** descrive il tempo in cui è stato creato e mandato il messaggio (formato iso).

I due campi più importati sono **local_offloading** e **overall_offloading**.

```
"local_offloading": [
    "name": String,
    "app_name": String,
    "function": {
        "image": String,
        "resources": {
            "memory": Number,
            "cpu": Number
        }
    }
]
```

Listing 7.6: Local Offloading Structure

É un vettore che descrive quali componenti di quali applicazioni devono essere eseguite sul nodo in questione; di questi componenti vengono specificati *name*, **app_name** e **function**. Nella funzione viene descritto quale immagine utilizzare per eseguire il job e le risorse che dovrebbe consumare.

Il secondo è un oggetto e stato implementato come set di applicazioni, in cui gli elementi sono implementati come un "dizionario" chiave-valore, in cui la chiave è il nome del componente avviato e il valore è il nome del nodo su cui è stato destinato.

```
"overall_offloading": {
    app_name: {
        component1: node-name,
        component2: node-name
    },
}
```

Listing 7.7: Overall Offloading Structure

Ogni soluzione riporta tutti i componenti delle varie applicazioni avviate, così che si possa sempre sapere dove sono avviati le parti. Questo permette anche di rischedulare dei jobs.

#### UserRequest Message

Questo messaggio viene utilizzato per richiedere l'esecuzione di una nuova applicazione da parte di un utente. Nel messaggio sono presenti le informazioni necessarie al DRONE-Controller per contattare DRONE e poi eseguire il task.

La struttura del messaggio è composta da quattro campi principali come mostrato sotto ed è identica al messaggio di *Advertisement* spiegato sopra alla sezione dedicata:

```
{
    "app_name": String,
    "base_node": String,
    "type": Adv_type,
    "components": [ ]
}
```

Listing 7.8: UserResquest Message Structure

#### Drone Message

Questo tipo di messaggio è già stato descritto nel Capitolo 5 dedicato al funzionamento di DRONE.

### 7.1.3 DRONE Controller

Il DRONE-Controller, scritto in Python, si occupa di ricevere ed elaborare le richieste provenienti da un utente ed attendere messaggi provenienti da DRONE. In questa versione della soluzione si tratta di un programma "standalone", che viene eseguito su ogni nodo del cluster[5].

Il codice sviluppato sfrutta un file di configurazione che permette di specificare alcuni parametri necessari per l'avvio dell'applicazione (i.e. informazioni server RabbitMQ, informazioni per Kubernetes etc.). Questo permette di non dover modificare il codice dell'applicazione al cambiamenti di alcuni parametri.

Di seguito viene riportato il file di configurazione dell'applicazione in modo da spiegare le principali comunicazioni che utilizza.

[5]Codice presente su *GitHub* nel repository `edge-offloading-controller`

```
1 [file]
2 result = tests/result/result_drone.json
3 request = tests/request/advertisement.json
4
5 [folder]
6 yaml-folder = yaml_files/
7
8 # parameter for Kubernetes
9 [kubernetes]
10 namespace = demo
11
12 # parameter for RabbitMQ user
13 [userRabbit]
14 broker_address = localhost
15 username = drone
16 password = drone
17
18 # parameters for RabbitMQ federation
19 [federation]
20 exchange_name = drone-exchange
21 set_name = drone
22 policy_name = federate-drone
23 pattern = .*drone.*
24
25 # parameters for RabbitMQ local
26 [rabbit]
27 queue_adv_route = app-advertisement
28 queue_user_req = user_request
29 queue_result = offloading-solution
30 queue_resources = resources-advertisement
31 queue_adv_ctrl = app-advertisement-ctrl
32 queue_adv_drone = app-advertisement-drone
```

Listing 7.9: DRONE-Controller Configuration

Nella configurazione mostrata sopra ci sono delle parti che sono puramente di gestione interna, come ad esempio le sezioni *file*, *folder*, altre invece utilizzate per configurare l'ambiente e i messaggi scambiati. Alla riga 9 vengono settati i parametri per K8S, alla riga 13 si configura l'accesso al server broker RabbitMQ, alla riga 19 sono presenti le varie impostazioni per la federazione dei cluster e il nome dell'exchange configurato mentre alla riga 26 vengono impostati i nomi delle code utilizzate per la comunicazione.

In Figura 7.2 viene mostrato il funzionamento generale dell'applicazione. L'applicazione all'avvio, dopo aver caricato la configurazione, crea tre code locali condivise (che spiegheremo in seguito) e si connette con le istanze di messaggistica

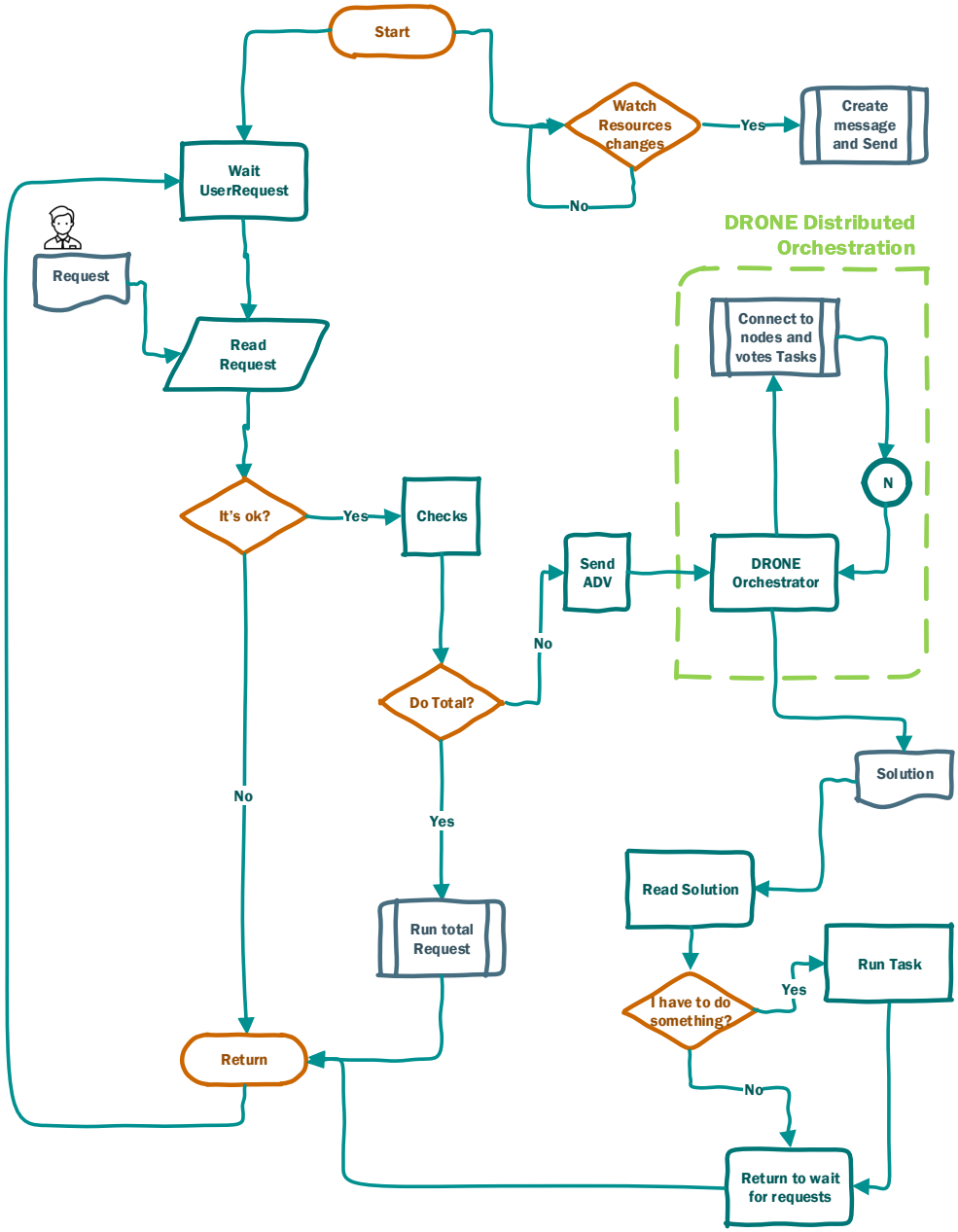


Figura 7.2: DRONE Controller FlowChart

necessarie (*advertisement*, *result*, *userRequest*, *resources*). Successivamente viene testato se il nodo del cluster su cui è stato eseguito non riporti errori, l'esito negativo del check porterà il DRONE-Controller ad inviare un messaggio di risorse nulle al componente DRONE in modo da non prendere parte all'orchestrazione. In

caso di esito positivo del check il programma potrà proseguire istanziando un *Watcher* sulle risorse del nodo e inviando un primo messaggio per notificare le risorse iniziali. In questo modo verrà inviato un messaggio di sincronizzazione delle risorse ogni volta che le risorse del nodo cambieranno. Per fare ciò si è usato un *Event Loop* che esegue la funzione ci calcolo risorse e invio del messaggio quando una condizione cambia. Questa funzione è stata implementata in modo tale da leggere le risorse totali del nodo, sottrarre le risorse consumate, infine creare il messaggio delle risorse disponibili ed inviare sulla code locale *resources-advertisement*.

Il resto del programma si svolgerà suddividendo il compito in sei *worker threads* differenti che si occuperanno di svolgere le attività e comunicare con gli altri componenti. Sono stati istanziati tre thread per attendere la ricezione dei messaggi, uno per ogni coda da cui bisogna leggere, e tre per "consumare" i messaggi ricevuti in modo da avere compiti separati per ogni worker.

I primi tre Thread, che attendono nuovi messaggi, svolgono due funzioni principali: attendono l'arrivo di un messaggio sulla coda RabbitMQ, una volta che il messaggio è arrivato si occupano di eseguire il *parsing* ed effettuare l'*enqueue* su una coda condivisa locale (coda di cui si è accennato sopra). Per fare ciò si sono definiti dei *message hendler* che verranno chiamati quando arriverà un nuovo messaggio sulla coda RabbitMQ. Gli altri Thread di cui abbiamo parlato si occupano di smaltire i messaggi caricati sulle code condivise locali. Estraggono i messaggi dalla coda uno alla volta, quando questi sono disponibili, e a seconda della loro funzione procedono processando i dati.

Questi tre Thread sono:

- *ThreadConsumeUserReq*, si occupa di smaltire i messaggi utenti per la richiesta di esecuzione/cancellamento o modifica di una applicazione. Esso si occuperà si capire il tipo di richiesta, creare il messaggio *Advertisement* ed inviarlo agli altri componenti, vedremo nella sezione 7.1.3 come viene effettuato l'invio. (In questa implementazione i due messaggi *Advertisement* e *UserRequest* utilizzano la stessa struttura. In caso contrario questo thread si occuperebbe della traduzione del messaggio);

- *ThreadConsumeAdv*, si occupa di smaltire i messaggi di *Advertisement*. Esso, dopo aver estratto il messaggio dalla coda, ne controlla il tipo, se è un messaggio di tipo *ADD* salva le informazioni mentre se è di tipo *DEL* controlla se su quel nodo si sta eseguendo qualche componente. Se sta eseguendo l'applicazione in parte procede alla rimozione dei componenti comunicando con le API di Kubernetes;

- *ThreadConsumeResult*, si occupa di smaltire i messaggi provenienti da DRONE di tipo *Solution*. Esso, dopo aver estratto il messaggio dalla coda, controlla se ci sono dei nuovi jobs da eseguire sul nodo. Se sono presenti nuovi jobs li esegue comunicando con le API di Kubernetes, se alcuni jobs sono stati modificati rispetto a ciò che è in esecuzioni sul nodo in quel momento li modifica.

L'implementazione di tutti i messaggi utilizzati è stata definita alla sezione 7.1.2.

### Invio messaggi

Per poter utilizzare il broker RabbitMQ e connettersi ad esso l'applicazione fa uso del protocollo nativo AMPQ attraverso la libreia "`pika`". Una volta stabilita la connessione con il broker e aver aperto un canale, l'applicazione dichiara l'exchange utilizzato per la federazione dei server RabbitMQ. Successivamente ad ogni invio di un messaggio crea una coda e ne effettua il binding sull'exchange se il messaggio deve essere direzionato alla federazione. In caso contrario crea solo la coda. La dichiarazione della coda ed il binding della stessa sono operazioni che hanno effetto solo alla prima esecuzione. Se la coda è già configurata nel message broker le due operazioni non apportano nessuna variazione.

A titolo di esempio viene ora spiegato come il messaggio di Advertisement viene inviato a DRONE-Controller e DRONE:

```
def send_adv(self, message, local=False):
    connection_id = connect()

    create_bind_queue(connection_id, QUEUE_ADV_CTRL,
   QUEUE_ADV_ROUTE, EXCHANGE)
    create_bind_queue(connection_id, QUEUE_ADV_DRONE,
   QUEUE_ADV_ROUTE, EXCHANGE)

    send_message(connection_id, QUEUE_ADV_ROUTE, message,
   local=False)
    disconnect(connection_id)
```

Listing 7.10: Send Function

Come si può vedere dalle rige 4 e 5 quando viene richiesto l'invio di un messaggio di advertisement, vengono create due code, QUEUE-ADV-CTRL e QUEUE-ADV-DRONE, entrambe con *Routing Key* QUEUE-ADV-ROUTE e legate all'exchange per la federazione. Questo accorgimento permette di pubblicare un messaggio sull'exchange assegnando una *Routing Key* comune ed affidare a RabbitMQ l'onere di distribuire il messaggio su tutte le code legate a quella *Routing Key*.

### Consumazione messaggi

Al fine di consumare i messaggi di una coda l'applicazione estrae da essa il contenuto e in base al tipo di messaggio che la coda dovrebbe contenere viene chiamata una `callback` apposita che si occupa del *parsing* del messaggio prima dell'elaborazione.

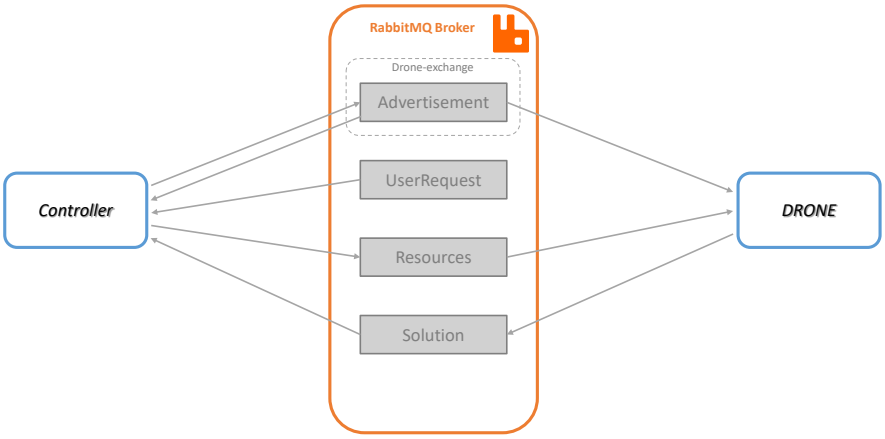


Figura 7.3: Queue Comunication RabbitMQ

## 7.1.4 DRONE Gui Interface

Il DRONE-Gui-Interface, scritto in Python, si occupa di fornire una interfaccia grafica per poter utilizzare il sistema drone in questa prima soluzione. [6].

Come si vede nell'immagine XX questo programma fornisce all'utente una GUI per poter interagire con il sistema. L'utente, Ignaro di come funziona il sistema sottostante, può selezionare delle applicazioni da un elenco, scegliere vari parametri per la configurazione ed effettuare la richiesta di orchestrazione. A questo punto il DRONE-Gui-Interface crea il messaggio di *UserRequest* e lo invia al DRONE-Controller del nodo che farà partire l'orchestrazione. Raggiunta una soluzione il DRONE-Gui-Interface si aggiornerà comunicando all'utente che il suo applicativo è stato avviato e su quali nodi sono stati distribuiti i componenti. Successivamente l'utente può scegliere di chiudere la sua applicazione e cancellare i jobs dal sistema.

In Figura 7.4 viene mostrata la Gui implementata.

## 7.1.5 Esempio VLC

Al fine di dimostrare il funzionamento della soluzione e dell'algoritmo sviluppato con DRONE, sono stati sviluppati degli esempi. In questa sezione viene riportata

[6]Codice presente su *GitHub* nel repository `edge-offloading-gui`

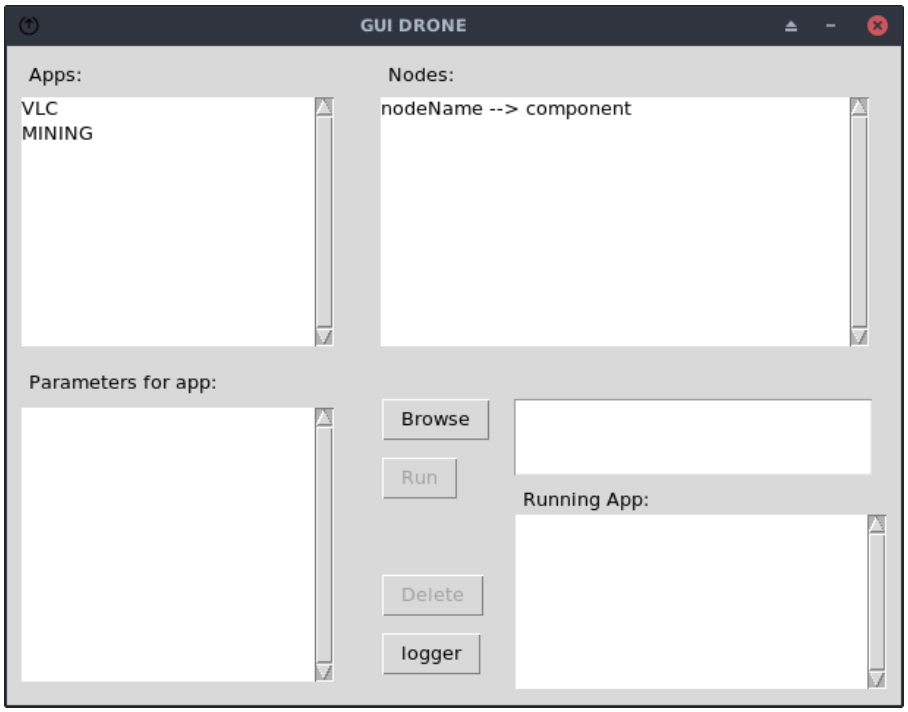


Figura 7.4: Gui Interface

una applicazione di video streaming perchè risulta uno *use case* molto interessante e sono state utilizzate varie tecniche.

### Dockerfile

La prima cosa che si è fatta è stato sviluppare il Dockerfile con i parametri necessari per l'avvio della applicazione VLC.

```
FROM debian:stretch-slim
MAINTAINER Andrea

# VARIABLES
ENV USER vlc
# Basics & VLC
RUN apt-get update && apt-get install --no-install-recommends
   -y \
    sudo libgl1-mesa-dri \
         libgl1-mesa-glx \
         vlc
# VLC: Dedicated User
ENV HOME /home/${USER}
RUN useradd -d ${HOME} -m ${USER} && \
```

```
    passwd -d ${USER} && \
    adduser ${USER} sudo
# SELECTION USER
USER ${USER}
# SELECTION STORAGE
WORKDIR ${HOME}

RUN sudo apt-get --purge autoremove -y && \
    sudo apt-get autoclean -y && \
    sudo rm /etc/apt/sources.list && \
    sudo rm -rf /var/cache/apt/archives/* && \
    sudo rm -rf /var/lib/apt/lists/*
CMD vlc
```

Listing 7.11: Docker VLC

### YAML per deployment

Per poter effettuare il deploy in Kubernetes sono stati scritti dei *manifest* in yaml che permettono di avviare le due componenti in modo indipendente, ma comunicanti. Utilizzando il gestore grafico X Window System (Noto anche come X11) è stato possibile avviare applicazioni in K8S e visualizzarne l'interfaccia grafica sul nodo host.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: video-streamer
spec:
  selector:
    matchLabels:
      run: video-streamer
  template:
    metadata:
      labels:
        run: video-streamer
    spec:
      nodeName: node_name # schedule pod to specific node
      containers:
        - name: video-streamer
          image: andreamora/vlc:latest
          imagePullPolicy: IfNotPresent
          volumeMounts:
            - mountPath: /home/vlc
              name: storagevideo
```

```
          command: ["/bin/sh","-c"]
          args: ["cvlc -vvv {path} --sout
   '#transcode{vcodec=h264, vb=1500, fps=35, width=640,
   height=360, acodec=mp3, ab=192, channels=2,
   samplerate=44100, scodec=none}:http{mux=ffmpeg{mux=flv},
   dst=:8080/}' --no-sout-all --sout-keep"]
          resources:
            requests:
              memory: "512Mi"
              cpu: "500m"
            limits:
              memory: "1024Mi"
              cpu: "1"
          ports:
            - containerPort: 8080
      restartPolicy: Always
      volumes:
        - name: storagevideo
          hostPath:
            path: /home/user/Videos   # directory location on
   host
---
apiVersion: v1
kind: Service
metadata:
  name: video-streamer
  labels:
    run: video-streamer
spec:
  ports:
    - port: 8080
      protocol: TCP
  selector:
    run: video-streamer
```

Listing 7.12: Streamer YAML

Il componente Streamer è composto da un Deployment e un Service che espone il flusso di dati verso l'esterno del nodo.

Il componente GUI invece è stato eseguito come semplice Pod configurato per utilizzare X11. Questo Pod contiene due container: il primo è il container che ospita l'applicazione, il secondo è un *InitContainer* che un particolare tipo di container che viene eseguito prima del container principale. In questo caso questo Container serve per attendere che il servizio di Streamer sia attivo.

```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: video-gui
spec:
  nodeName: node_name # schedule pod to specific node
  containers:
    - name: gui-pod
      image: andreamora/vlc:latest
      env:
        - name: DISPLAY
          value: :0
        - name: PULSE_SERVER
          value: unix:/run/user/1000/pulse/native
      stdin: true
      tty: true
      imagePullPolicy: IfNotPresent
      volumeMounts:
        - mountPath: /tmp/.X11-unix
          name: video
        - mountPath: /dev/snd
          name: sound
        - mountPath: /dev/shm
          name: tempsystem
        - mountPath: /var/run/dbus
          name: systembus
        - mountPath: /run/user/1000/pulse/native
          name: soundenv
      command: ["/bin/sh","-c"]
      args: ["vlc -V x11 http://{service-name}.demo:8080
   --no-qt-privacy-ask --no-metadata-network-access
   --snapshot-path=/home/vlc/snapshots
   --network-caching=1000"]
      resources:
        requests:
          memory: "256Mi"
          cpu: "256m"
        limits:
          memory: "256Mi"
          cpu: "500m" #=0.5cpu
  initContainers:
    - name: init-myservice
      image: busybox:1.28
      command: ['sh', '-c', 'until nslookup {service-name};
   do echo waiting for {service-name} service; sleep 2;
   done;']
  restartPolicy: OnFailure
```

```
volumes:
  - name: video
    hostPath:
      path: /tmp/.X11-unix    # directory location on host
  - name: sound
    hostPath:
      path: /dev/snd
  - name: tempsystem
    hostPath:
      path: /dev/shm
  - name: systembus
    hostPath:
      path: /var/run/dbus
  - name: soundenv
    hostPath:
      path: /run/user/1000/pulse/native
```

Listing 7.13: Gui YAML

## 7.2 Evoluzione Prototipo

Questa seconda sezione del capitolo illustra le configurazioni dei cluster e le implementazioni dei vari componenti della soluzione integrata in Kubernetes. Utilizzando questa soluzione si è incominciato a sviluppare una architettura completamente integrata in Kubernetes e gestibile con le API offerte; con l'obiettivo finale di avere una architettura trasparente nei confronti dell'utente che desidera eseguire uno o più jobs. Con questa soluzione i due "programmi" non saranno più esterni a Kubernetes ma saranno completamente integrati. Già partendo dalla configurazione del cluster vedremo che non ci saranno più applicativi non gestibili da K8S. Quindi in questa sezione analizzeremo, in primo luogo, come deve essere configurato ogni singolo cluster mentre successivamente illustreremo nel dettaglio le principali componenti della soluzione.

### 7.2.1 Configurazione clusters

In questa sezione viene illustrata la configurazione dei cluster Kubernetes per l'utilizzo della soluzione. Le varie componenti implementate verranno esposte nelle sezioni successive.

Anche questa soluzione come già detto utilizza il message broker RabbitMQ per la comunicazione tra i vari componenti. Il message broker è stato istanziato come container Docker all'interno del cluster. L'immagine utilizzata è quella disponibile sul repository ufficiale del Docker Hub, versione "3-management". Questa versione permette si istanziare RabbitMQ con il plug-in di management già abilitato.

Per utilizzare il broker su Kubernetes sono stati creati un deployment, un servizio clusterIp e un servizio nodePort. Per questione di spazio non si riportano tali codici, sono visibili sul repository *GitHub* `netgroup-polito/dronev2/rabbitMQ`.

Il servizio di tipo clusterip viene utilizzato per le comunicazioni interne al cluster, ad esempio per pubblicare il messaggio di advertisement. Il servizio di tipo nodeport invece viene utilizzato per effettuare la federazione tra servers rabbitMQ su cluster differenti.

Anche in questo caso, come per l'architettura precedente, è stato implementato uno script che, dopo aver creato i cluster utilizzando "kind", ci istanzia rabbitMQ e li federa. Per ragioni di spazio lo script non viene riportato ma è possibile consultarlo sul repository *GitHub* `netgroup-polito/dronev2`.

Per poter effettuare la federazione tra clusters è necessario disporre di un *ID* univoco per ogni cluster. La comunità di kubernetes si sta muovendo per definire uno standard da utilizzare come clusterID ma non è stata ancora raggiunta una soluzione accettata dai mantainer. Per ovviare a questo problema si è scelto di creare una configmap in cui configurare tutti i vari parametri che servono al cluster per funzionare, tra questi anche il *clusterID*.

### 7.2.2 Messaggi

La maggior parte dei messaggi scambiati dalle varie componenti di questa soluzione sono i medesimi della soluzione precedente presentati nella Sezione 7.1.2, quindi non verranno presentati nuovamente. In questa parte illustreremo solo la struttura di un nuovo tipo di messaggio che nella prima soluzione non era stato utilizzato.

Si tratta di un messaggio di *Acknowledge* utilizzato per notificare il cluster d'*origine* del avvenuto deploy del job, messaggio non necessario nella versione precedente in quanto era utilizzata internamente ad un cluster K8S. Spiegheremo in seguito, nella sezione dedicata all'*DRONE-Operator*, il motivo per cui si è risultato necessario l'introduzione di tale messaggio.

Questo messaggio può essere di tre tipi: **ADD_ACK** per identificare l'avvenuto deploy di un componente, **DEL_ACK** identifica la rimozione di un deploy ed infine **MOD_ACK** motifica l'avvenuta modifica di un componente.

Il messaggio di *Acknowledge*, sopra introdotto, è strutturato come segue:

```
{
    "sender": String,
    "base_node": String,
    "type_ack": AckType,
    "component": {
        "name": String,
        "app_name": String,
        "function": {
            "image": String,
            "resources": {
                "memory": Number,
                "cpu": Number
            }
        }
    },
    "timestamp": String
}
```

Listing 7.14: Acknowledge Message Structure

Il campo **sender** identifica il nodo che ha inviato il messaggio di *acknowledge* mentre il campo **base_node** designa il nodo destinatario del messaggio, il quale deve ricevere le notifiche dei componenti avviati. Il **timestamp** descrive il tempo in cui è stato creato e mandato il messaggio (formato iso). Il campo **type_ack** come già detto sopra identifica il tipo del messaggio. Infine il campo **component** definisce il nome del componente e il nome dell'applicazione. Inoltre è stato aggiunto anche il campo function per notificare con quale funzione è stato avviato il job e le risorse che utilizzerà; questo perchè con il messaggio di *solution* DRONE potrebbe decidere di avviare un componente con una funzione differente da quella

definita nel messaggio di *advertisement*.
Questo messaggio viene inviato dal DRONE-Controller al nodo d'origine dell'applicazioni ogni qual volta venga instanziato, cancellato, modificato un suo job.

Nella Figura 7.5 viene mostrato il funzionamento della soluzione. All'avvio vengono eseguiti quattro componenti e il server RabbitMQ.

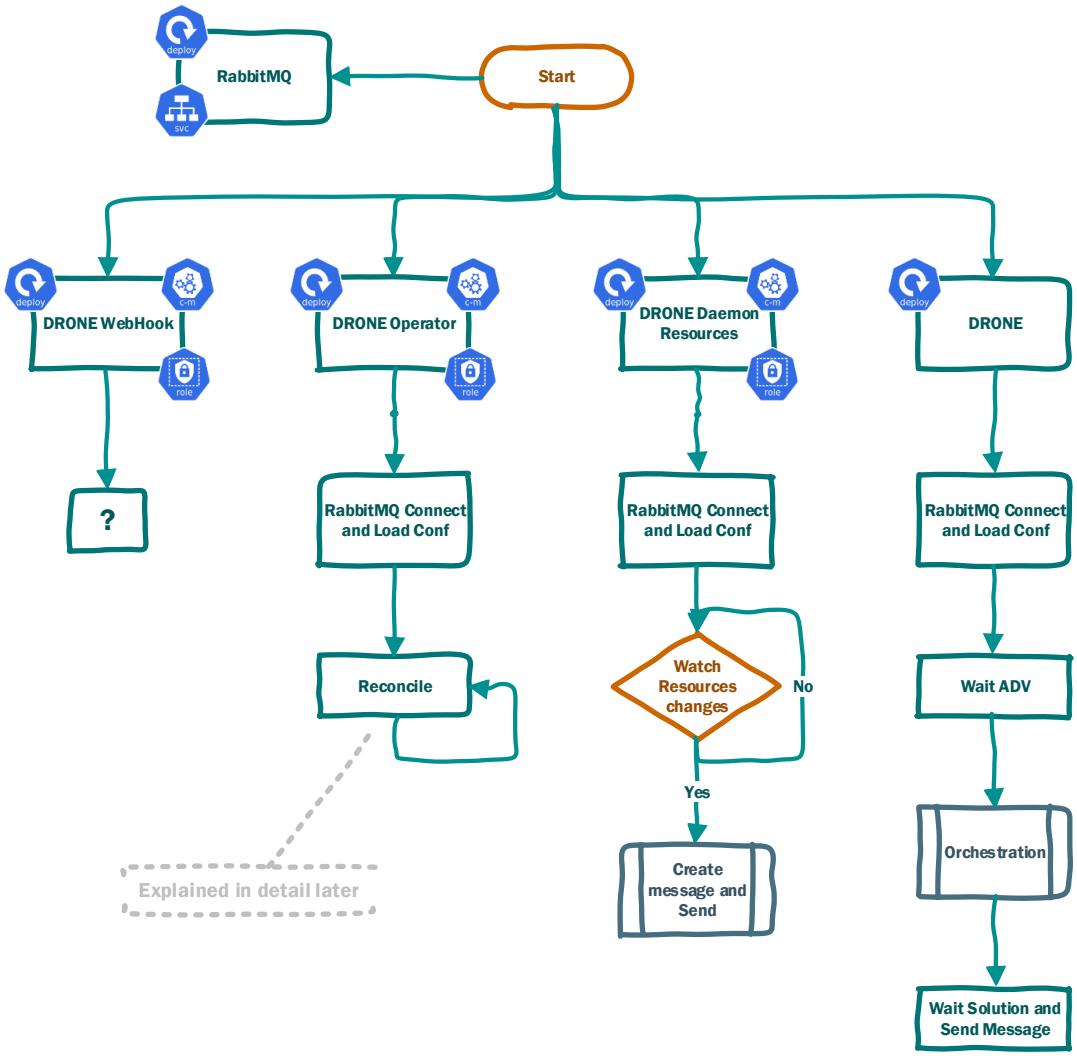


Figura 7.5: Evolution FlowChart

### 7.2.3 DRONE-Operator

In questa sezione verrà descritta l'implementazione del componente nominato *DRONE-Operator*. Questo componente svolge quasi tutte le principali funzionalità che

nell'architettura iniziale, descritta nella sezione 7.1, venivano implementate dal *DRONE-Controller*.

L'**Operator**, come descritto nella sezione 4.2.3, è un'applicazione nativa di Kubernetes capace di gestire il ciclo di vita di risorse che possono essere definite dall'utente. Utilizzando questo paradigma è possibile quindi estendere le API standard.

In questa implementazione si è deciso di utilizzare l'operator in quanto permette di estendere le API standard di Kubernetes e permettendo di gestire un oggetto custom, CRD. Successivamente vengono riportate le parti più significative dell'implementazione delle CRD implementate.

Il DRONE-Operator è un'applicazione scritta in GO che offre tre funzionalità principali:

- Resta in ascolto delle richieste, *CustomResource*, provenienti dall'utente che desidera avviare un job, le analizza, avvia lo scheduling e aggiorna lo stato della risorsa;
- Interagisce con le API esposte da Kubernetes per avviare jobs destinati a quel cluster;
- Resta in ascolto delle soluzioni provenienti dal componente DRONE.

Il sistema DRONE con questo componente può gestire due tipi di informazione. In primo luogo definisce quale tipi di API verranno gestiti da DRONE; in secondo luogo dichiara quali clusters DRONE verranno coordinati per la singola richiesta.

L'applicazione come già anticipato viene avviato all'interno di un pod quindi ne viene effettuato il deploy attraverso un *Manifest* che ne definisce la struttura.

### CRD

Sono state definite delle CustomResourcesDefinition che saranno trattate dal DRONE-Operator. Queste CRD sono molto simili a quelle utilizzate dal progetto KubeFed. La configurazione del tipo deve essere composta da tre concetti fondamentali:

- *Templates*, definisce la rappresentazione della risorsa da eseguire tra i clusters;
- *Placement*, definisce quali clusters non devono essere considerati nella schedulazione;
- *Overrides*, definisce le variazioni dei campi definiti in templates per ogni singolo cluster.

Queste tre astrazioni permettono di rappresentare una risorsa distribuita su più clusters. Queste sono le informazioni minime richieste per la propagazione di un job, inoltre possono essere aggiunte informazioni avanzate come le policy o condizioni di scheduling differenti.

Oltre a ciò si è pensato di aggiungere un campo che identifichi lo stato della risorsa sul cluster per ogni job eseguito. In questo modo si può leggere tutte le informazioni con un semplice comando come *"kubectl get DroneFederationDeployment -A"*, in linea con le funzionalità di K8S.

Di seguito viene riportato lo schema della CRD con un esempio, CR, per favorire la comprensione.

```
apiVersion: drone.polito.com/v1alpha1
kind: DroneFederatedDeployment
metadata:
  name: nginx-dronefederateddeployment
spec:
  template:
    metadata:
      labels:
        app: nginx
    spec:
      replicas: 1
      template:
        metadata:
          labels:
            app: nginx
        spec:
          containers:
            - name: nginx
              image: nginx:latest
              resources:
                limits:
                  cpu: "1"
                  memory: "512"
  placement:
    clusters:
      - name: cluster2
```

Listing 7.15: Example CustomResources

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: dronefederateddeployments.drone.polito.com
```

```
spec:
  group: drone.polito.com
  names:
    kind: DroneFederatedDeployment
    listKind: DroneFederatedDeploymentList
    plural: dronefederateddeployments
    singular: dronefederateddeployment
    shortNames:
    - dfd
  scope: Namespaced
  subresources:
    status: {}
  validation:
    openAPIV3Schema:
      properties:
        apiVersion:
          description: 'APIVersion defines...'
          type: string
        kind:
          description: 'Kind is a string value
   representing...'
          type: string
        metadata:
          type: ...  #(Validate field)
        spec:
          type: ...  #(Validate field)
        status:
          type: ...  #(Validate field)
  version: v1alpha1
  versions:
  - name: v1alpha1
    served: true
    storage: true
  additionalPrinterColumns: #(optional)
    - name: creation
      type: string
      JSONPath: .metadata.creationTimestamp
    - name: phase
      type: string
      JSONPath: .status.phase
```

Listing 7.16: CustomResourcesDefinition Schema

Come mostrato nella configurazione della CRD vengono definiti i nomi, la validazione e quali colonne stampare alla richiesta da parte dell'utente.

### Funzionamento algoritmo

In questa sezioni descriviamo il funzionamento del *operator* sviluppato. Come già detto precedentemente questa applicazione gestisce una *CRD*, sfruttando il gestore di Kubernetes. Ogni qual volta un oggetto del tipo gestito viene modificato il framework esegue una chiamata per effettuare la riconciliazione della risorsa.

Alla CRD sviluppata sono stati associati quattro stati che può avere:

- *SCHEDULING*, una CustomResource avrà questo stato quando sarà in fase di orchestrazione, quindi non è ancora stata notificata una soluzione per essa;
- *PENDING*, una CustomResource avrà questo stato quando la soluzione è stata notificata ma le componenti della CR non sono ancora state eseguite;
- *RUNNING*, una CustomResource avrà questo stato quando tutte le componenti che la compongono saranno eseguite;
- *DONE*, una CustomResource avrà questo stato quando le componenti hanno terminato i loro job (se sono componenti con un termine).

Attraverso queste quattro fasi è possibile delineare un algoritmo che detiene lo stato delle componenti avviato. La struttura dell'applicazione DRONE-Operator è basata su queste fasi.

Come mostrato nell'immagine 7.6, quando una nuova CR, *Custom Resource*, viene creata e percorre per la prima volta la "riconciliazione" gli viene assegnata la fase iniziale SCHEDULING. Avendo lo stato impostato ora la risorsa passerà nel controllo per quello stato che si occuperà di effettuare l'invio del messaggio di Advertisement. Infine, prima di aggiornare lo stato della risorsa salvata viene controllato se l'istanza contiene il *Finalizer*, in caso contrario viene aggiunto (verrà spiegato più avanti il motivo). Ora la risorsa risulta salvata sulla base dati di K8S con lo stato SCHEDULING e un Finalizer configurato. D'ora in poi ogni volta che la risorsa viene modificata verrà chiamata la riconciliazione passando attraverso il controllo per la fase sopra citata. La CR che è stata creata però non cambierà stato finchè l'orchestratore DRONE non notificherà la soluzione, in questo caso lo stato diverrà PENDING. In questa fase ogni operator presente su ogni nodo della federazione dovrà avviare le componenti a lui dedicate e notificare tramite un ACK al nodo originale l'avvenuta esecuzione (Figura 7.9). Quando tutti i nodi interessati hanno notificato la risorsa può passare alla fase RUNNING (Figura 7.7).

Nella Figura 7.8 viene descritto il comportamento dell'applicazione all'arrivo di un messaggio di Advertisement. Nella Figura 7.7 viene descritto il comportamento dell'applicazione all'arrivo di un messaggio di Acknowledge. Nella Figura 7.9 viene descritto il comportamento dell'applicazione all'arrivo di un messaggio di Solution.

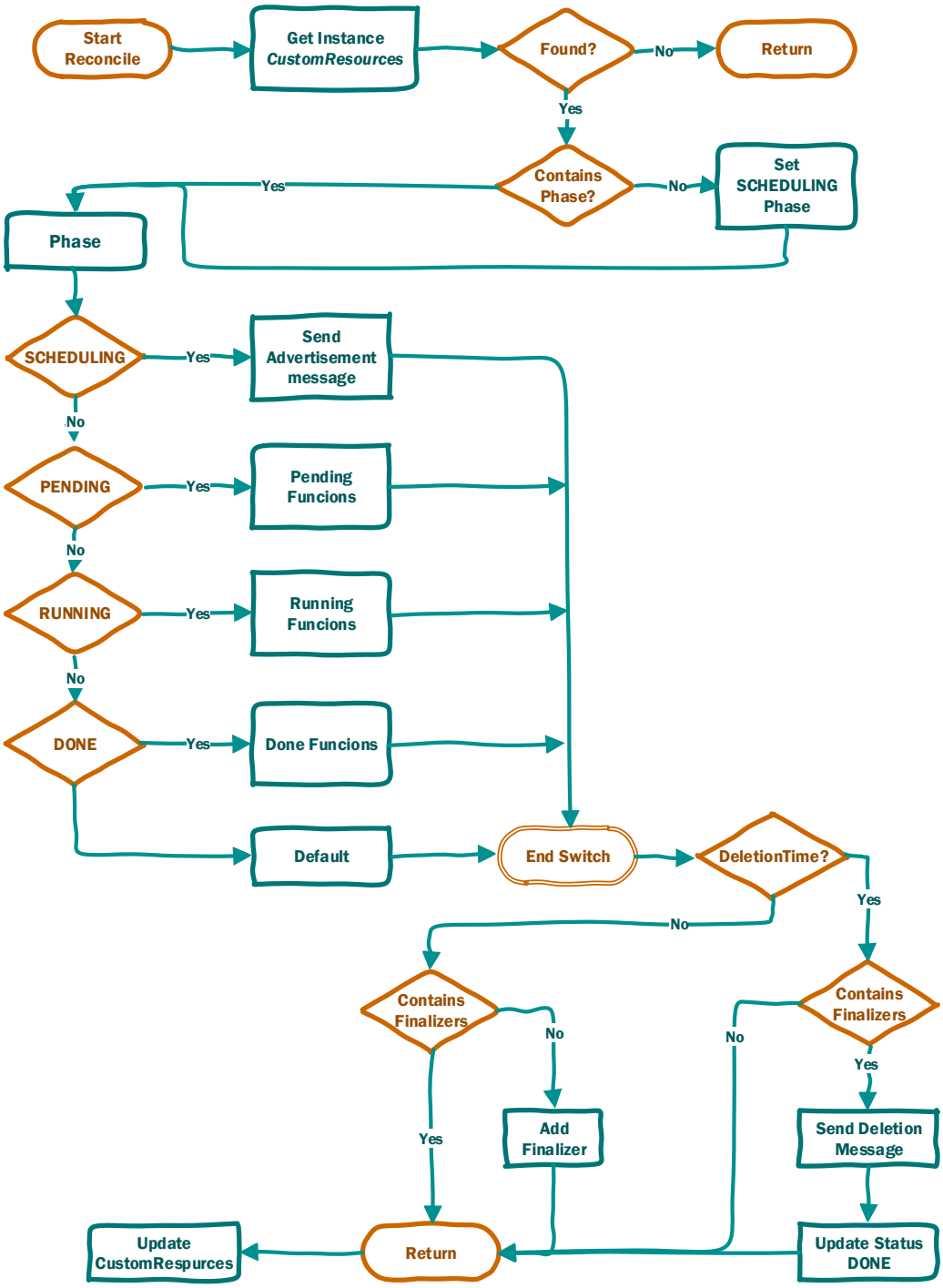


Figura 7.6: Reconcile Evolution FlowChart

Trattiamo ora la cancellazione della risorsa. Quando una CR viene cancellata verrà chiamata la "riconciliazione" anche in questo caso. La differenza che si

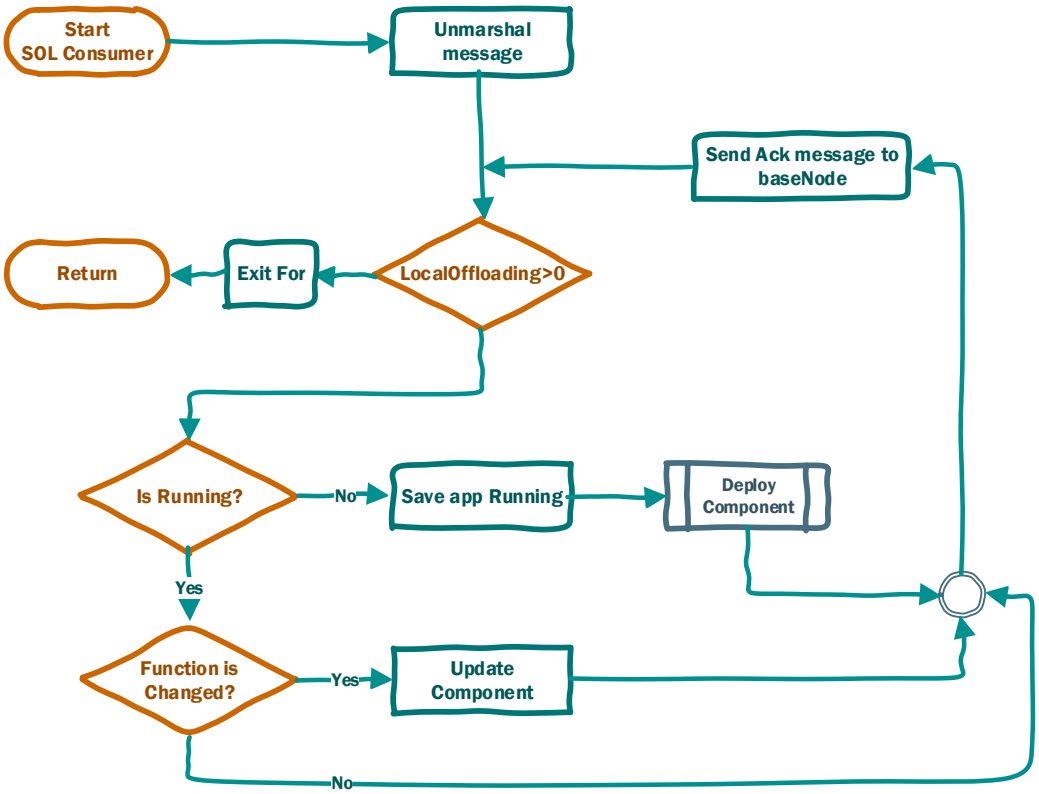


Figura 7.7: Acknowledge consume FlowChart

è sfruttata per capire che si è in fase di cancellazione è il campo nominato *DeletionTimestamp*. Quando questo campo è configurato significa che è stata richiesta la cancellazione della risorsa. In questo caso l'applicazione controlla se il campo *finalizers* contiene il finalizer desiderato (Figura 7.7). In caso affermativo procede con l'invio del messaggio di cancellazione dell'applicazione. Quando i nodi remoti notificano l'avvenuta cancellazione del job a loro assegnato si procede ad eliminare il finalizer dalla risorsa ed eliminare essa stessa.

Il controllo del *DeletionTimestamp* viene effettuato sempre ad ogni riconciliazione in quanto la richiesta di cancellazione può essere fatta durante ogni fase del ciclo di vita della risorsa.

In questa sezione non viene trattato il funzionamento del invio di messaggi tra i vari nodo perchè è già stato esposto nella Sezione 7.1.3. A parte qualche piccola differenza il funzionamento è il medesimo.

L'unica differenza significativa da citare è l'invio dei messaggi di *Acknowledge*. Questi messaggi vengono inviati alla coda dedicata del nodo d'origine sfruttando l'exchange federato e *Routing Key* uguale al nome della coda univoca per ogni nodo

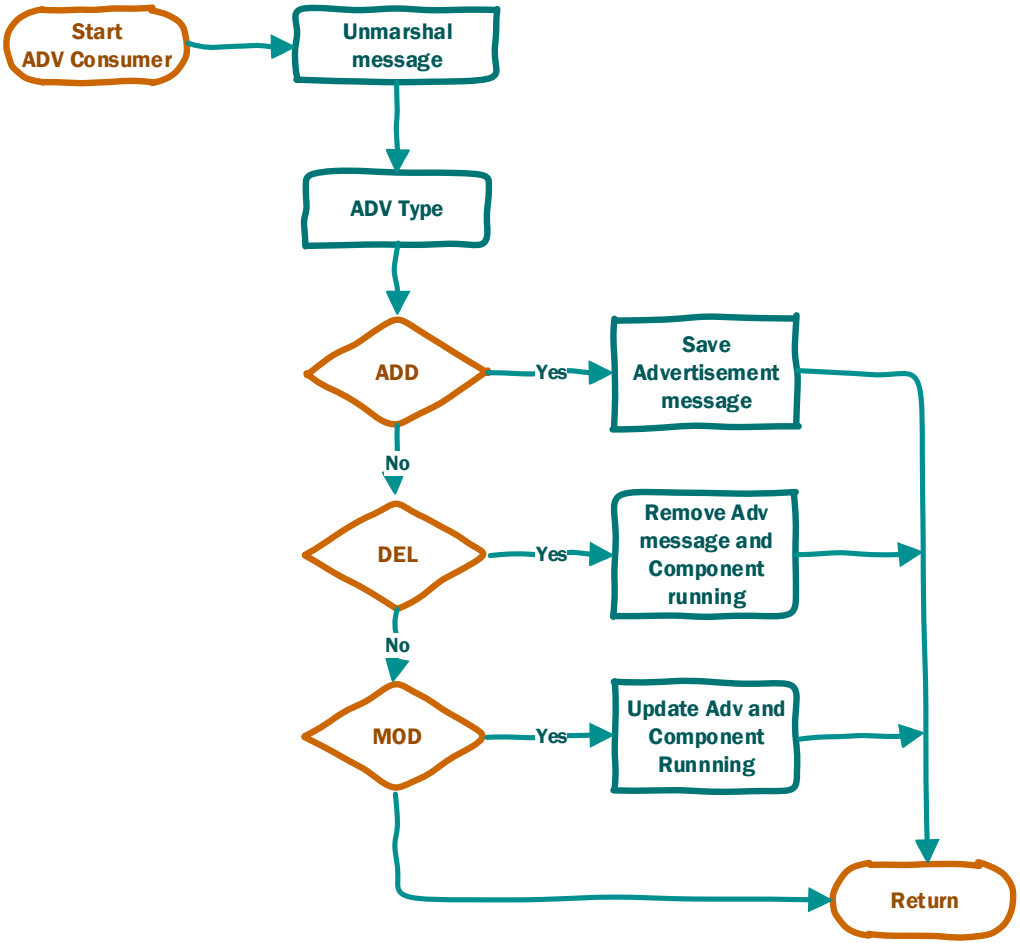


Figura 7.8: Advertisement consume Evolution FlowChart

(ack-queue-nodeName). Quindi penserà rabbitMQ all'instradamento del messaggio.

Al fine di avere un codice dinamico si è inserita una *ConfigMap* per permettere di applicare modifiche alla configurazione senza dover modificare il container avviato. I manifesti della config-map e altro non vengono riportati ma sono visibili sul repository *GitHub* e simili a quelli mostrati nella sezione 7.2.4.

### 7.2.4 DRONE-Daemon-Resources

Il DRONE-Daemon-Resources è un programma scritto in GO che si occupa di richiedere le risorse del cluster, modificarle se serve, e notificarle al componente DRONE attraverso il message broker RabbitMQ del cluster. Questo programma è

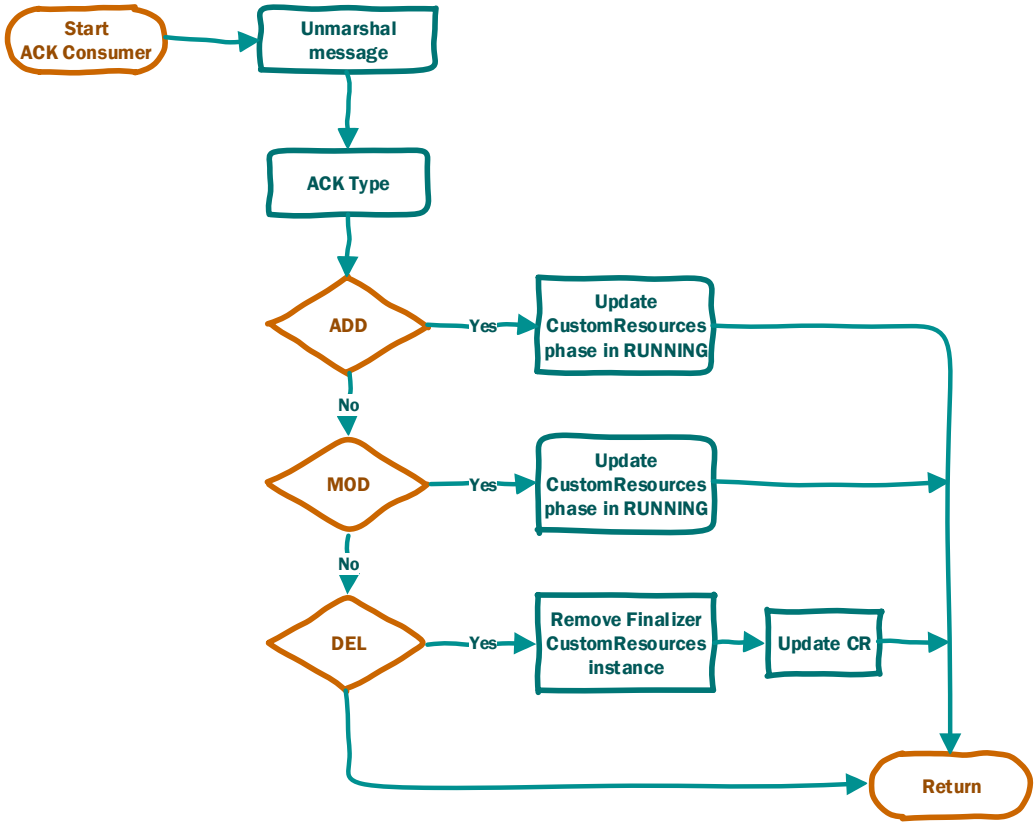


Figura 7.9: Result consume FlowChart

stato implementato come client Kubernetes con delle regole per poter dare accesso alle API. Questo componente nella architettura finale può essere personalizzato per ogni cluster, questa personalizzazione permette di poter far lavorare DRONE sulle metriche che ogni cluster desidera. Ad esempio se un provider vuole esporre per la federazione solo il 50% delle proprie risorse, con questa configurazione basta che configura il DRONE-Daemon-Resources in modo da notificare allo Scheduler DRONE solo il 50%.

Questa capacità di adattarsi alle modifiche a runtime è stata ottenuta utilizzando delle config-map per la configurazione dell'applicativo. Di seguito mostreremo come è stata scritta e utilizzata.

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: drone-resources-configmap
  namespace: drone
data:
```

```
  # Configuration values can be set as key-value properties
   or set
  conf.yaml: |
    # RabbitMq Configuration
    rabbit:
      broker_address: "rabbitmq-service"
      broker_port: "5672"
      v_host: ""
      username: "drone"
      password: "drone"
      queue_resources: "resources-advertisement"
    # Parameters for RabbitMQ federation
    federation:
      exchange_name: "drone-exchange"
    # Parameter for Kubernetes
    kubernetes:
      namespace: "demo"
      cluster_name: "cluster1"
    # Parameter for Resources (scale is percentage for reduce
   resources)
    resources:
      scale: 100
```

Listing 7.17: ConfigMap Drone Resources

In questa configurazione vengono descritti tutti i campi necessari per l'esecuzione del programma. Questa configurazione viene collegata come file nel deployment oppure come variabili d'ambiente. In questa soluzione è stata collegata come file di configurazione chiamato conf.yaml. Come detto in precedenza la config-map può essere usata per modificare la lettura delle risorse del cluster prima di notificarle a DRONE, ad esempio modificando il parametro *scale*.

Per poter leggere le risorse del cluster è stato effettuato il deploy di una *Role* e un *ClusterRoleBinding*. Di seguito viene riportato lo yaml.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: drone-reader
  namespace: drone
rules:
  - apiGroups: [""]
    resources: ["pods","nodes","nodes/status"]
    verbs: ["get", "watch", "list"]
---
apiVersion: rbac.authorization.k8s.io/v1
```

```
kind: ClusterRoleBinding
metadata:
  name: drone-reader-binding
  namespace: drone
subjects:
  - kind: ServiceAccount
    name: default
    namespace: drone
roleRef:
  kind: ClusterRole
  name: drone-reader
  apiGroup: rbac.authorization.k8s.io
```

Listing 7.18: Role Drone Resources

L'applicazione in questione come già detto può essere personalizzata per ogni nodo. L'implementazione che ora si propone è un implementazione base che si occupa di leggere le risorse massime del nodo sottrarre le risorse utilizzate e notificarle attraverso un messaggio su una coda di RabbitMQ. É stato utilizzato un watcher sui Pod che ascolta la creazione/rimozione/modifica dei pod e agisce solo quando le risorse variano, questo perchè i pod sono i soli oggetti che possono consumare risorse.

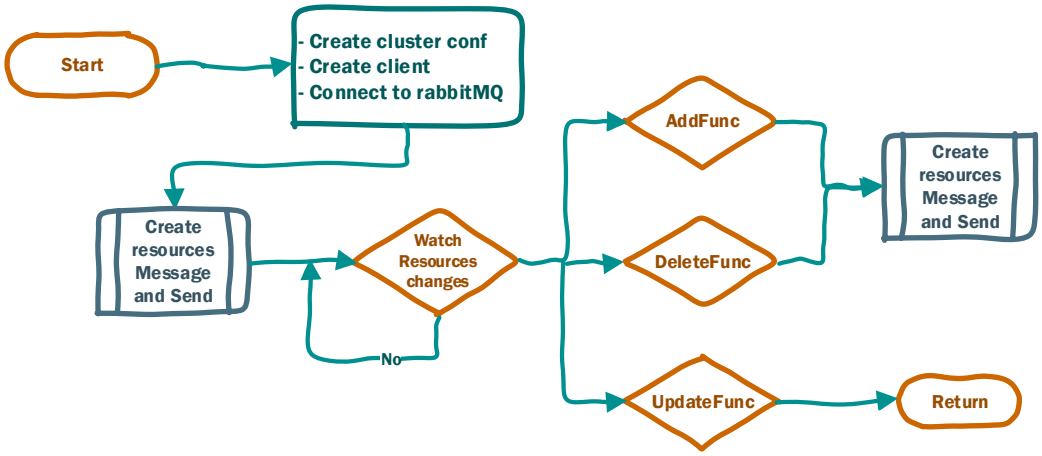


Figura 7.10: Resources Daemon FlowChart

Il codice del DRONE-Daemon-Resources, dopo essere stato implementato, è stato "buildato" per creare un eseguibile, successivamente "dockerizato" e caricato sul repository Docker Hub per essere sempre scaricabile da qualunque cluster. Il programma potrà essere avviato in un cluster Kubernetes eseguendo il deploy configurando uno specifico volume con la configurazione definita sopra.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: drone-daemon-resources
  namespace: drone
spec:
  selector:
    matchLabels:
      run: drone-daemon-resources
  template:
    metadata:
      labels:
        run: drone-daemon-resources
    spec:
      volumes:
        - name: drone-resources-configmap-volume
          configMap:
            name: drone-resources-configmap
      containers:
        - name: drone-daemon-resources
          image: drone-daemon-resources:first
          imagePullPolicy: IfNotPresent
          tty: true
          volumeMounts: pod
            - name: drone-resources-configmap-volume
              mountPath: /etc/config
      restartPolicy: Always
```

Listing 7.19: Deployment Drone Resources

Alla riga 22 viene definito quale configurazione utilizzare (nome della config-map descritta sopra).

# Capitolo 8

# Risultati Sperimentali

Questo ultimo capitolo analizza le prestazioni dell'algoritmo DRONE e la bonta delle soluzioni implementate. Sono state eseguite due tipi di validazione: la prima riguardante l'algoritmo DRONE utilizzando la prima soluzione proposta, la seconda basata sull'evoluzione descritta nei capitoli precedenti. I test basati sulla prima soluzione vogliono mostrare il funzionamento dell'algoritmo testando la velocità nel restituire una soluzione buona. Invece i test basati sulla seconda evoluzione sono improntati sulla velocità di schedulare dei servizi tra cluster appartenenti a domini differenti e sul numero di messaggi scambiati per raggiungere tale soluzione.

## 8.1 Validazione Algoritmo Drone in Cluster

I test sono stati eseguiti in un ambiente di cluster virtualizzato con tre nodi su cui partecipanti alla orchestrazione come descritto in figura

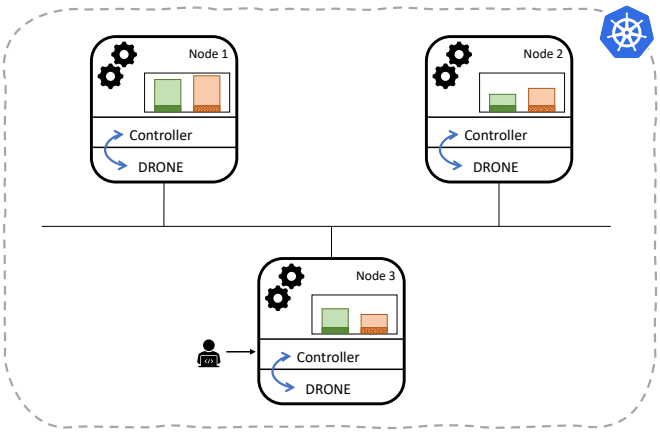


Figura 8.1: Validazione DRONE Ambiente

Con tre nodi in un ambiente virtualizzato le tempistiche medie per schedulare un job sono state dell'ordine di 4/5ms, con oscillazioni a seconda di come i messaggi venivano scambiati.

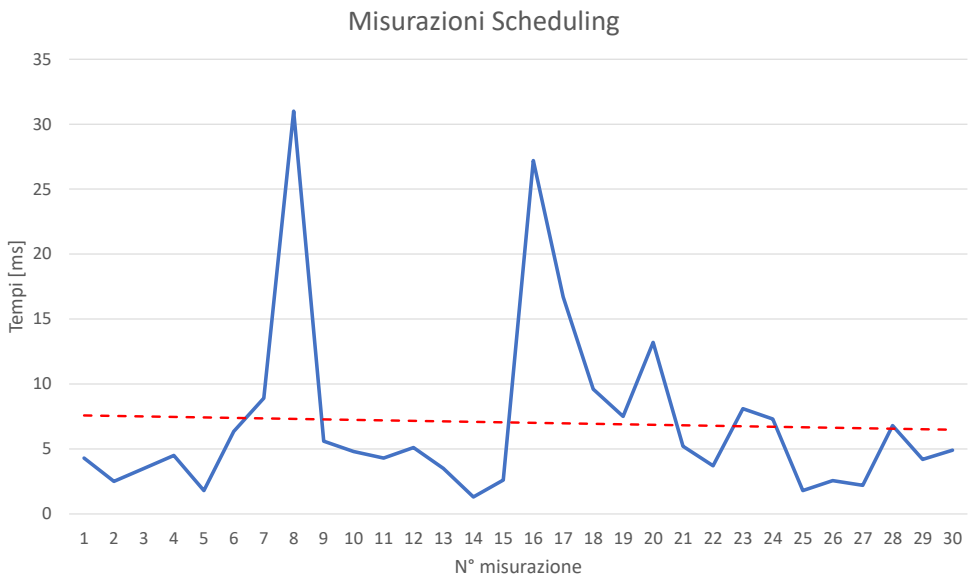


Figura 8.2: Validazione DRONE Algoritm

## 8.2 Validazione Approccio Multi Clusters

I test sono stati eseguiti in un ambiente di cluster virtualizzati come descritto in Figura 8.3, aumentando i cluster della federazione

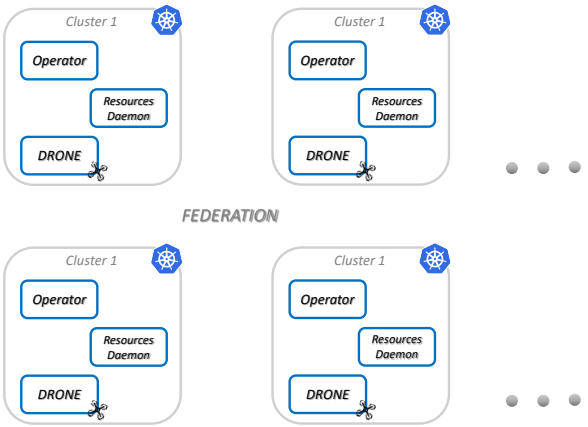


Figura 8.3: Validazione Multi-Cluster Ambiente

In Figura 8.4 si mostrano le medie dei tempi con cui si riesce a schedulare delle applicazioni all'aumentare del numero di cluster coinvolti nella federazione. Come si può notare i tempi per il raggiungimento di una soluzione buona aumentano con l'aumentare del numero di cluster e sono un poco più alti di quelli ottenuti prima nella validazione.

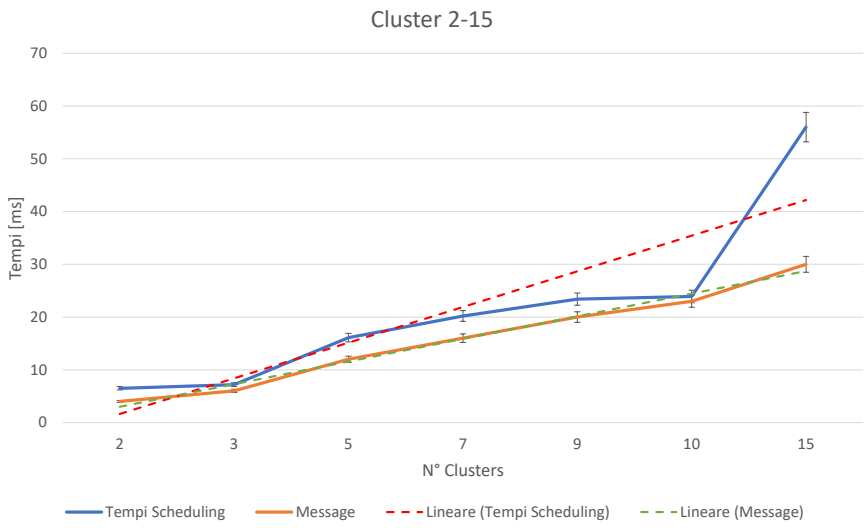


Figura 8.4: Validazione Multi-Cluster Scheduler

In questi test non sono stati presi in considerazioni i tempi impiegati da Kubernetes per eseguire il deploy. Quelle tempistiche esulano dalla implementazione sviluppata.

# Capitolo 9

# Conclusioni e sviluppi futuri

In questa tesi sono state definite delle architetture che consentono di utilizzare un algoritmo di orchestrazione distribuito, *DRONE*, all'interno di una federazione di cluster superando il diffuso utilizzo di una centralizzazione. Rimangono tutt'oggi sfide ancora aperte come la comunicazione delle componenti avviate su due nodi remoti distinti e un evoluzione dell'orchestratore per un uso in ambiente di produzione.

Analizzate le condizioni di business delle soluzioni proposte precedentemente sono state riscontrate alcune problematiche. Un provider difficilmente vuole che il proprio "voto" per un job di a visibile a tutti gli altri provider. Sotto queste condizioni si è pensato di migrare l'architettura ad una architettura con orchestrazione decentralizzata. La logica di orchestrazione, invece che completamente distribuita, sarà localizzata sul cluster che ha richiesto l'esecuzione del job così che i "voti" non siano visibili da altri.

# Bibliografia


[1] *K3S*. URL: https://medium.com/better-programming/using-a-k3s-kubernetes-cluster-for-your-gitlab-project-b0b035c291a9.

[2] *K3S website*. URL: https://k3s.io/.

[3] *KubeEdge website*. URL: https://kubeedge.io/en/.

[4] *KubeEdge github*. URL: https://github.com/kubeedge/kubeedge.

[5] *KubeFed*. URL: https://github.com/kubernetes-sigs/kubefed.

[6] *Docker*. URL: https://www.docker.com/what-docker.

[7] *Docker container networking*. URL: https://docs.docker.com/engine/userguide/networking/.

[8] *Kubernetes 101: Pods, Nodes, Containers, and Clusters*. URL: https://medium.com/google-cloud/kubernetes-101-pods-nodes-containersand-clusters-c1509e409e16.

[9] *What is RabbitMQ?* URL: https://www.cloudamqp.com/blog/2015-05-18-part1-rabbitmq-for-beginners-what-is-rabbitmq.html.

[10] *Rabbit Federation*. URL: https://www.rabbitmq.com/federation.html.

[11] *Kubernetes*. URL: https://kubernetes.io/docs/home/.

[12] *GoLang*. URL: https://golang.org/doc.

# Ringraziamenti

Dopo tutti questi anni, eccoci qua a concludere questo percorso. É stato un fantastico viaggio, un'incredibile esperienza. Ho imparato molto di più di quanto potessi mai pensare ed ora si apre un nuovo capitolo.

Innanzitutto, vorrei fare un grandissimo ringraziamento alla mia famiglia, mio padre Gianpiero e mia madre Antonella per il loro supporto sia emotivo che finanziario, senza la quale non avrei potuto finire questo lungo percorso. E, ovviamente, un grazie anche a mia sorella Enrica per essere sempre disponibile e ad Alice per avermi sostenuto in questi anni.

Un altro grande ringraziamento va al professore Fulvio Risso per avermi dato la possibilità di lavorare su questo progetto ed approfondire gli argomenti ampliando le mie conoscenze. Inoltre ringrazio Gabriele ed Alex per i consigli che mi hanno dato.

Infine, senza fare elenchi infiniti di nomi, ringrazio tutti coloro che mi sono stati vicini in questi anni.

Grazie a tutti.

Andrea