

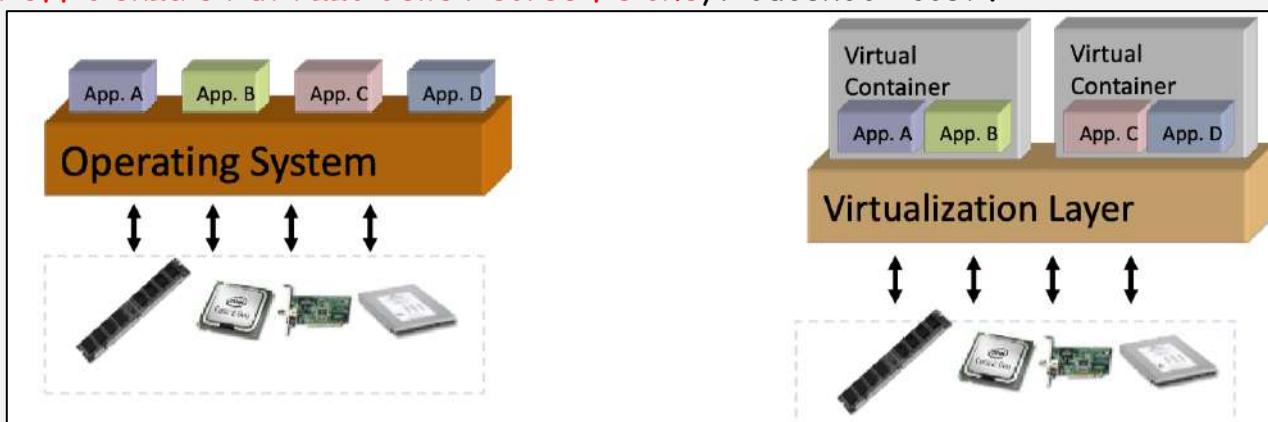
Virtual Networks and Cloud Computing

Written by Simone Maiorani

Capitolo 2: Virtualizzazione

La virtualizzazione si riferisce alla creazione di **risorse** IT virtuali (come server, storage, reti, etc.) che possono essere utilizzate come se fossero risorse fisiche, ma sono in realtà **fornite tramite software**.

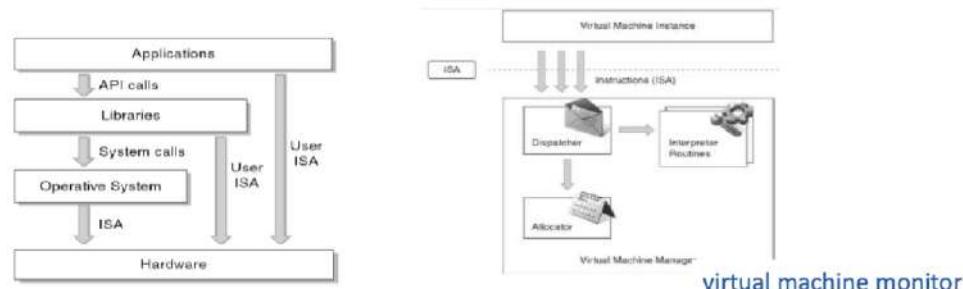
In altre parole, la virtualizzazione consente di creare una replica virtuale di una risorsa fisica (ad esempio, un server), **consentendo a più utenti di accedere e utilizzare tale risorsa contemporaneamente. Ciò consente di massimizzare l'efficienza e l'utilizzo delle risorse fisiche**, riducendo i costi.



L'hardware è accessibile attraverso un **instruction set** che consiste praticamente nel linguaggio macchina. Alcune istruzioni le utilizza il sistema operativo, altre le applicazioni stesse. Se in mezzo ci mettiamo un monitor di fatto dobbiamo metterci in mezzo e intercettare le chiamate all'hardware fatte attraverso l'istanza di una macchina virtuale e cercare di far eseguire in maniera più opportuna le istruzioni da un **dispatcher**. Abbiamo poi un **allocator** che fornisce alla macchina virtuale le risorse necessarie evitando conflitti. Abbiamo poi un **interpreter** che simula le istruzioni che fanno riferimento alle risorse in modo tale da riflettere la loro esecuzione nell'ambiente della macchina virtuale (indispensabile poiché le macchine virtuali non hanno accesso diretto alle risorse fisiche). Infine, **lo scopo della virtualizzazione dei sistemi è quello di condividere in modo trasparente le risorse hardware di una macchina fisica tra i sistemi operativi di più macchine virtuali per fargli eseguire quante più istruzioni possibili direttamente sul processore senza interazione con il VMM (Virtual Machine Monitor)**.

Virtualization Requirements

- Virtual Machine Monitor
- A virtual machine monitor is a control program comprising:
 - A dispatcher
 - An allocator
 - A set of interpreters, one per privileged instruction.



Un **VMM** fornisce un ambiente duplicato, o essenzialmente identico al computer originale, per i programmi. Qualsiasi programma eseguito con VMM dovrebbe mostrare un effetto identico a quello effettuato se il programma fosse stato eseguito direttamente sulla macchina originale, con la possibile eccezione delle differenze causate dalla disponibilità delle risorse del sistema.

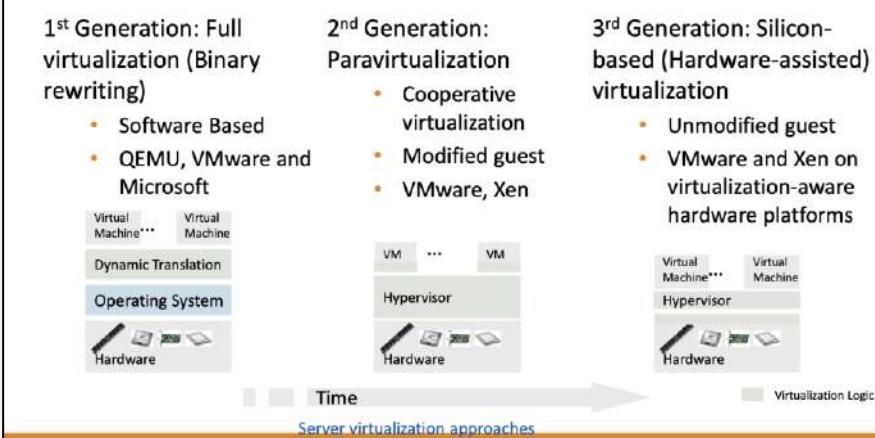
Come VMM abbiamo gli **hypervisor** ovvero dei **mediatori** tra VM, l'**hardware fisico** e le **applicazioni** (poste al di sopra). Ci sono gli **hypervisor di tipo 1** che sono posti immediatamente sopra l'hardware e **ha tutti i meccanismi di un normale kernel o sistema operativo** in termini di memoria, periferiche e gestione del processore; inoltre, implementa i meccanismi per la gestione delle macchine virtuali. Le macchine virtuali in esecuzione su VMM dispongono di un proprio sistema operativo e sono denominate macchine "guest".

Di contro sono **difficili da sviluppare** perché sono complessi.

Ci sono gli **hypervisor di tipo 2** che sono dei normali processi in esecuzione in un sistema operativo denominato "host".

Gestiscono direttamente le macchine virtuali, che sono i suoi sottoprocessi, mentre **la gestione dell'hardware è affidata al sistema host**.

Evolution of Software solutions



Prima generazione: si traduce tutto, l'hypervisor utilizza l'hardware per emulare tutte le operazioni, ogni singola istruzione viene tradotta. Approccio di natura un po' lento.

Seconda generazione: consiste nella para-virtualizzazione, alcune istruzioni sono mandate direttamente all'hardware mentre altre sono emulate garantendo buone prestazioni (ma non eccellenti).

Terza generazione: la virtualizzazione è gestita dall'hardware poiché le architetture moderne prevedono che l'instruction set includa funzioni di virtualizzazione. Le ultime generazioni di processori funzionano in questo modo, incremento della velocità e della sicurezza.

Docker

Non si virtualizzano le risorse hardware ma si virtualizza il sistema operativo. I container virtualizzano il sistema operativo così come gli hypervisor virtualizzano l'hardware. **Un container non implementa l'intero sistema operativo con le sue applicazioni sopra ma porzioni di librerie**, andando ad **utilizzare il sistema operativo nativo presente nelle macchine host**.

Vengono creati degli spazi isolati dove vivono le singole applicazioni che fanno uso dello stesso sistema operativo. I **container** sono molto diffusi.

Se implementassimo un'applicazione come server web, broker o servizio di storage al suo utilizzo essa farà uso delle proprie librerie che a loro volta faranno uso di altre librerie generando uno stack più o meno complesso.

Stack: tutto quello che serve per far girare l'applicazione a livello sottostante (librerie, etc.). **Risolve molti problemi** perché se queste applicazioni convivessero tutte all'interno della stessa macchina, le varie librerie potrebbero essere tra loro incompatibili o magari di versioni diverse.

Senza contare che questi stack dovrebbero essere portati su un hardware del tutto generico il che è quasi impossibile.

Incrociare tutta la vastità di applicazioni con i possibili stack che occorrono per implementarle va a generare la cosiddetta **matrice infernale**.

Le righe sono i vari stack mentre le colonne sono le applicazioni.

The Matrix From Hell							
Stacks	DBs	Web Frontend	Background Workers	User API	Analytics DB	Clients	Cloud
DBs	MySQL	Redis	MongoDB	Node.js	Apache Beam	React Native	AWS Lambda
Web Frontend	Angular	React	Node.js	Node.js	Apache Beam	React Native	AWS Lambda
Background Workers	Redis	Redis	Apache Beam	Node.js	Apache Beam	React Native	AWS Lambda
User API	Node.js	Node.js	Apache Beam	Node.js	Apache Beam	React Native	AWS Lambda
Analytics DB	Apache Beam	Apache Beam	Apache Beam	Apache Beam	Apache Beam	Apache Beam	AWS Lambda
Clients	React Native	React Native	Apache Beam	Apache Beam	Apache Beam	Apache Beam	AWS Lambda
Cloud	AWS Lambda	AWS Lambda	AWS Lambda	AWS Lambda	AWS Lambda	AWS Lambda	AWS Lambda



Possiamo fare un'analogia con degli oggetti reali e i vari mezzi di trasporto utilizzati. Nessuno dei due durante la propria realizzazione può ignorare l'altro. Il problema è trovare un protocollo, alias una regola, che vada bene per tutto e che non richieda una customizzazione per un mezzo di trasporto in base all'oggetto. Qualcosa di universale che non dia problemi con dogana, etc. Soluzione: grazie al container (quello vero) si inserisce la merce dentro fregandosene del mezzo di trasporto che verrà utilizzato. Viceversa, chi trasporta se ne frega di cosa c'è dentro il container, basta che lo trasporti. Questo è il concetto di container nell'informatica.

Dentro al container non occorre inserire tutto ma solo quello che serve per far girare l'applicazione sul sistema operativo. Ad esempio, se installiamo il container su Windows faremo uso di Windows ma dentro al container vengono inserite solo le librerie che servono per una determinata applicazione.

Se si vuol far girare un'altra applicazione, creeremo un altro container e le librerie del primo non andranno a interferire con quelle del secondo.

Il sistema operativo ha tutto l'essenziale per far girare i container.

Ognuno fa il suo mestiere e sono indipendenti.

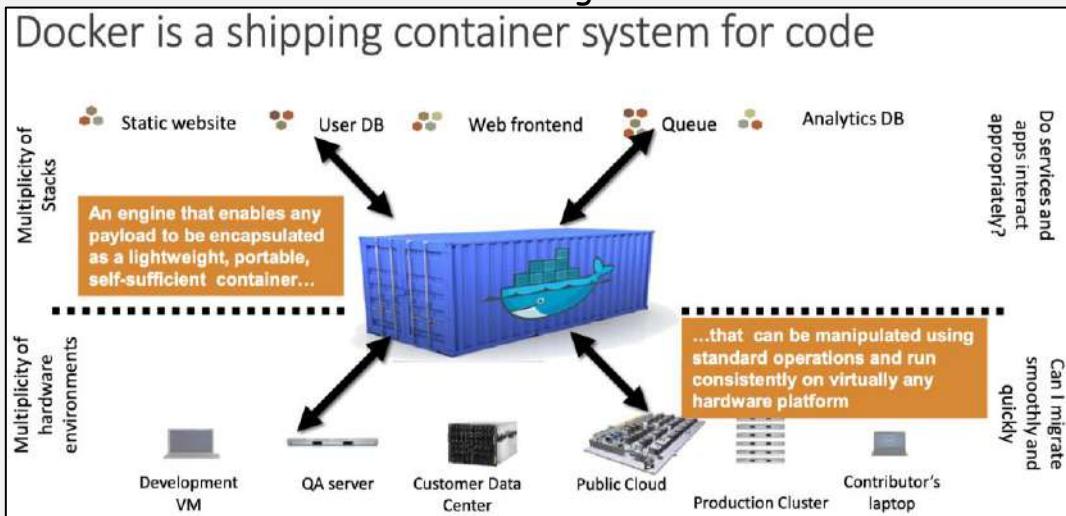
Utilizzando un hypervisor, è possibile utilizzare un altro ed intero sistema operativo nelle macchine virtuali, aumentando però la pesantezza.

Chi sviluppa lo fa senza tener conto di server, sistema operativo, etc.

Sopra ci sono gli sviluppatori, sotto coloro che devono offrire applicazioni agli utenti, ovvero i DevOps.

Grazie ai container si separano le funzioni e si semplifica la vita alle persone.

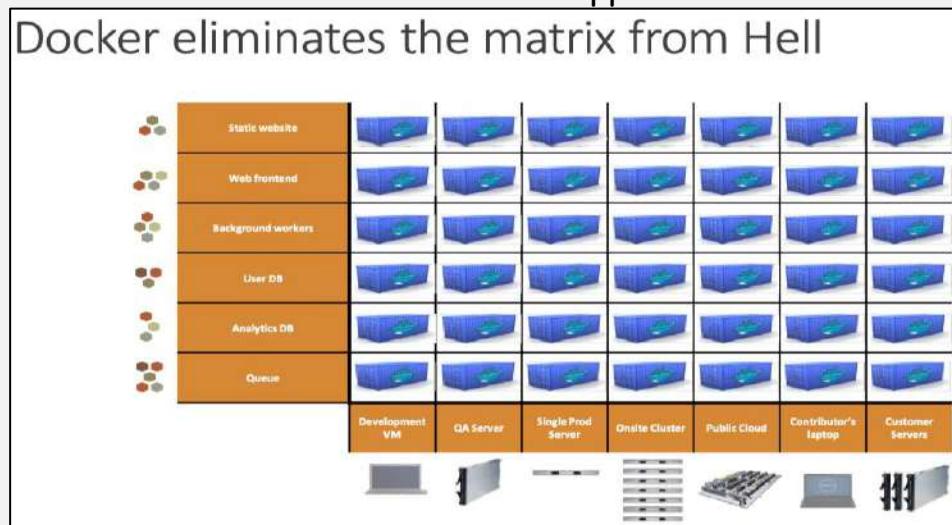
La matrice infernale diventa ora facile da gestire.



Diversi container possono essere orchestrati in modo che ciascuno di loro esegua un pezzo dell'applicazione complessiva. Inoltre, una volta creato un container per un determinato lavoro, è possibile utilizzarlo per più applicazioni in cui occorre tale funzionalità, senza dover programmare ogni volta.

Docker è una piattaforma di containerizzazione che consente di creare, distribuire e gestire applicazioni in ambienti virtualizzati chiamati **container**. In pratica, Docker permette di isolare l'esecuzione di un'applicazione e le sue dipendenze in un ambiente controllato e preconfigurato, rendendo l'applicazione più portabile e facile da distribuire su diverse infrastrutture. Con Docker è possibile **creare un'immagine del proprio software**, contenente tutte le dipendenze e le configurazioni necessarie per eseguire l'applicazione. Quest'immagine può essere distribuita attraverso il Docker Hub o attraverso una rete privata e utilizzata per creare uno o più container che eseguono l'applicazione su diversi host.

I container Docker sono leggeri, veloci e isolati tra loro. Consentono di eseguire più applicazioni su un singolo host senza interferenze tra di loro. Docker fornisce anche uno strumento di orchestrazione chiamato **Docker Swarm** per gestire e scalare i container su più host, offrendo una maggiore flessibilità e scalabilità nell'esecuzione delle applicazioni.



Abbiamo quindi due tipi di virtualizzazione: i **container** e le **macchine virtuali**.

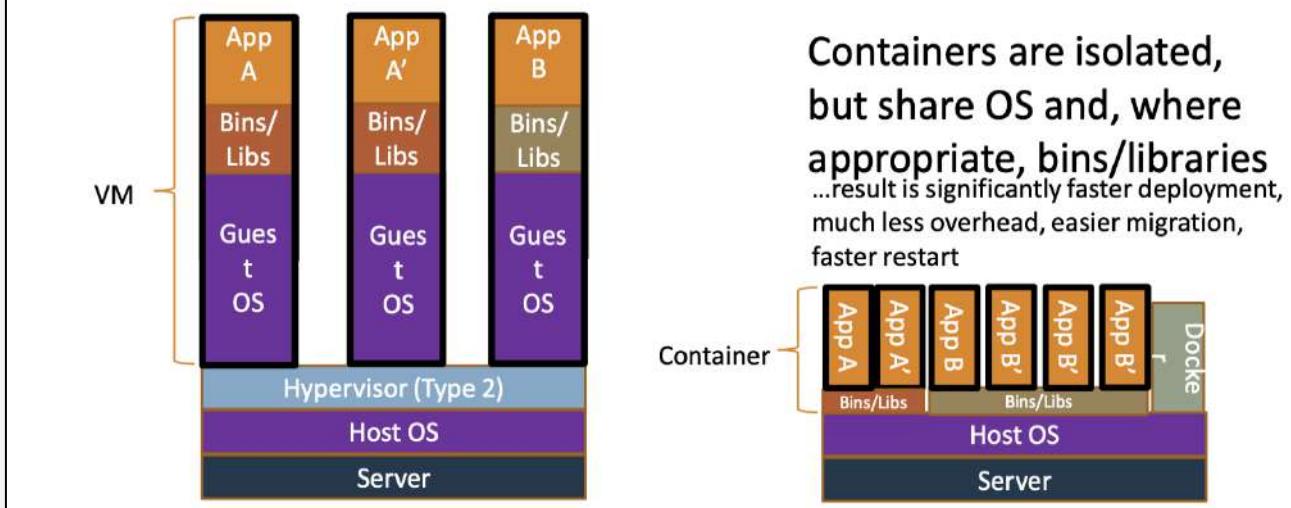
I container sono soluzioni più leggere e flessibili in quanto non contengono al loro interno un intero sistema operativo.

Le VM sono, invece, più sicure ma più pesanti.

Per sicurezza intendiamo il livello di isolamento e/o la gestione dei container. L'attaccante esperto, infatti, è in grado di spiare cosa succede negli altri container. Ci si può fidare dei container ma le VM restano più sicure.

L'utilizzo di uno o dell'altro dipende da cosa occorre realizzare.

Containers vs. VMs



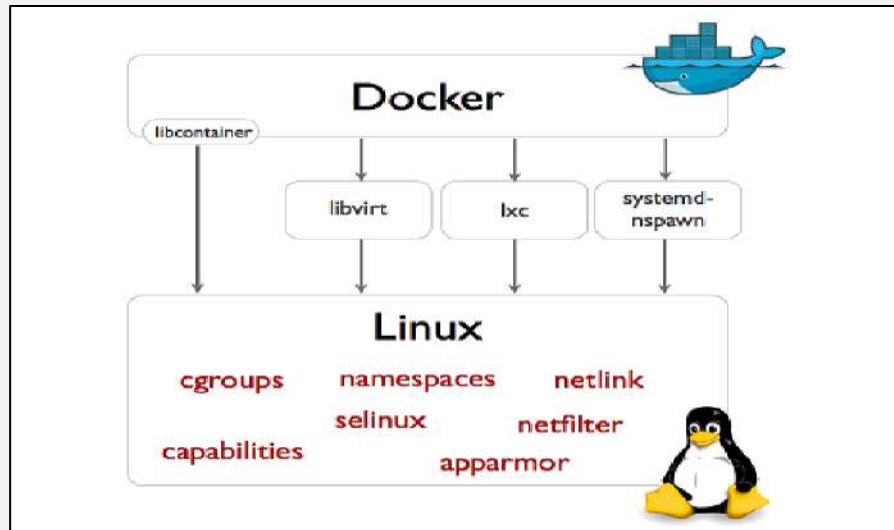
Nel caso delle VM abbiamo un'infrastruttura fisica (Server) ed abbiamo bisogno di un sistema operativo (Host OS, colui che ospita, il proprietario). Su questo sistema possiamo installare un hypervisor: se di tipo 1, comprende anche il sistema operativo, se di tipo 2, è come se fosse un'applicazione. Sopra l'hypervisor possiamo avere una o più VM ciascuna con sistema operativo ospite (Guest OS). Questo crea stabilità da un lato ma spreca risorse dall'altro poiché ci sono tanti sistemi operativi che girano sulla stessa macchina. Nel sistema operativo abbiamo tutte le librerie necessarie per il suo corretto funzionamento e per far girare tutte le applicazioni presenti al suo interno. Paghiamo la versatilità con il fatto che abbiamo nella stessa macchina più sistemi operativi che girano (usando le risorse in maniera poco efficiente). Quando si virtualizza, la RAM è la risorsa più preziosa.

Nel caso dei Container vi è un solo sistema operativo e delle librerie che consentono di creare una specie di isolamento tra i processi. Se si utilizza un container per eseguire delle operazioni, è difficile guardare gli altri container presenti nella stessa macchina. Le applicazioni possono girare in ambienti separati con librerie simili o molto diverse. Si rinuncia alla flessibilità sul sistema operativo.

Quando non occorre un Guest OS specifico conviene utilizzare i container in quanto sono più leggeri e con migliori performance.

Se, ad esempio, ad un utente si eroga l'infrastruttura allora serve una VM perché dobbiamo fargli scegliere il SO. Se, invece, ad un cliente si eroga un servizio va bene un container perché non ci si interessa del SO in uso.

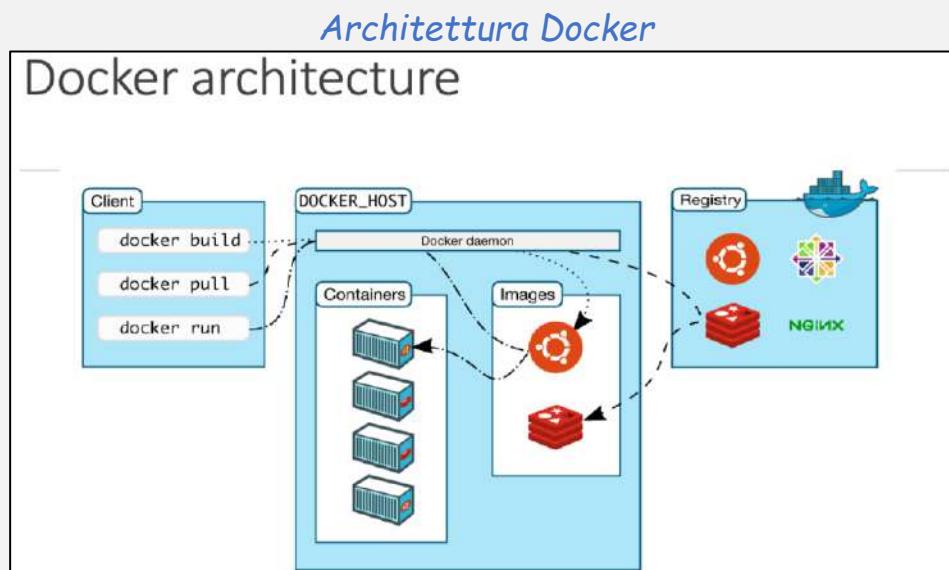
Le tecnologie sottostanti a Docker:



Impara l'utilizzo di Linux dal sito: <https://www.linuxjourney.com>

Il primo container sicuro di natura professionale nasce per Linux che consente l'isolamento dei container tramite i **namespace** ovvero una porzione di risorse all'interno della quale il nome della singola risorsa dev'essere univoco.

Se una risorsa si chiama Pippo allora dentro quel namespace non devono esserci altre risorse di nome Pippo. In diversi namespace si può, invece, ripetere.



Docker funziona attraverso un'architettura **client-server**.

Il suo cuore è dentro il **Docker Host**. Possiamo connetterci ad esso attraverso un **client** che può essere nella stessa macchina o in macchine diverse.

Nel Docker Host c'è un processo **Docker daemon**, il quale gestisce il ciclo di vita dei container, ovvero: attivazione, gestione e abbattimento quando terminano.

Per istanziare i container fa uso delle **immagini**. Grazie ad esse è possibile ricreare più volte lo stesso container, come se fossero degli stampini.

Le immagini possono essere create o prese direttamente dalla rete.

Docker Hub è il repository di immagini di Docker.

Docker build, pull e run consentono al client di interagire con il Docker Host.

Docker Registry esegue le operazioni di pull e di run delle immagini scaricandole in automatico dal repository di Docker.

Una volta che abbiamo la nostra immagine configurata possiamo creare un backup o inviarla al Docker Hub per scaricarla quando occorre.

Docker Desktop ha al suo interno il client Docker.

Docker Compose è uno strumento utilizzato per gestire simultaneamente più container, facendoli operare tra loro per la stessa applicazione.

Non si costruiscono più le applicazioni in maniera monolitica ma le si suddividono in **microservizi** che possono essere anche riutilizzati per altro.

Quando andiamo ad utilizzare Docker si vanno a creare e utilizzare gli **oggetti** di quest'ultimo: immagini, reti, volumi, container, etc.

Le immagini sono di sola lettura e fondamentalmente contengono le istruzioni necessarie per far girare un container.

Un container è un'istanza eseguibile di un'immagine. Possiamo crearlo, eliminarlo, fermarlo, spostarlo, etc. Generalmente, un container è isolato dagli altri. Una volta eliminato un container, le informazioni al suo interno, se non sono state salvate in una memoria permanente, vengono completamente rimosse. I **volumi** sono degli hard disk virtuali che possiamo decidere di attaccare ad un container. Generando dei dati e salvandoli nei volumi, anche se il container muore essi restano nei volumi.

I volumi possono essere attaccati a qualsiasi container.

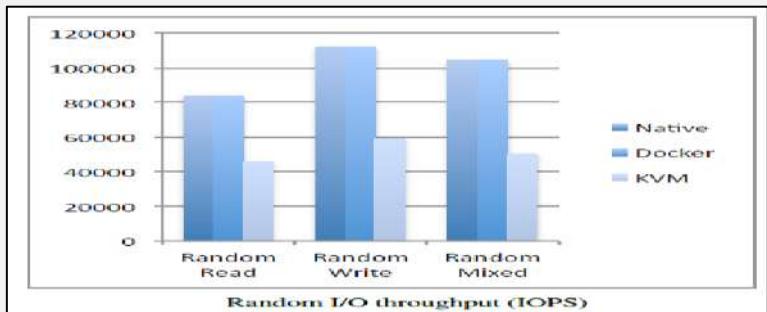
Example: docker run

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

Dobbiamo mandare in esecuzione l'immagine di Ubuntu (se non è in locale la va a cercare nel repository configurato). Lancia l'immagine, crea il container e dopo averlo aperto lancia il comando bin/bash. Il comando **-i** sta per interactive, il comando **-t** sta per terminal (l'output è mostrato nel terminale).

Digitando **exit** chiudiamo il comando bin/bash ma il container rimane attivo.

Confronto tra sistema operativo nativo, Docker e VM KVM.



Ovviamente le prestazioni dipendono anche dall'hardware.

La latenza (definita come l'intervallo di tempo tra il verificarsi di un evento e il tempo in cui quell'evento viene "gestito") è molto importante, soprattutto per applicazioni real-time come il controllo di una macchina a guida autonoma.

Se occorre tirare su un'applicazione con latenza molto bassa con il container si può fare. Con una macchina virtuale di KVM (che è un hypervisor di buon livello, superiore a VirtualBox) si ottiene una latenza di grandezza maggiore.

Dockerfile

```
FROM centos:7
RUN yum install -y python-devel python-virtualenv
RUN virtualenv /opt/indico/venv
RUN pip install indico
COPY entrypoint.sh /opt/indico/entrypoint.sh
EXPOSE 8000
ENTRYPOINT /opt/indico/entrypoint.sh
```

Di **Dockerfile** ne esiste solo uno, il nome Dockerfile è obbligatorio.

È uno script. Si usa per realizzare un'applicazione. O meglio, è possibile indicare l'immagine su cui si baserà l'istanza del container. Nell'esempio si utilizza il sistema operativo CentOS. Scarichiamo un'immagine e la customizziamo.

FROM: indica l'immagine da cui partire, se c'è localmente ok sennò si prende dal Docker HUB (il più grande repository di sempre per Docker).

RUN: si usa per installare applicazioni, ad es. Python, con le varie librerie etc.

Con il comando RUN si crea di fatto un layer dell'applicazione.

Essa viene realizzata infatti a strati, il primo è l'OS. (in questo caso CentOS).

COPY: si utilizza per copiare file dal FS locale al FS del container.

EXPOSE: si usa per indicare la porta, ad esempio 8000, dove il container si mette in ascolto e ci scrive, principalmente utilizzando il protocollo TCP.

ENTRYPOINT: mandiamo in esecuzione il processo che deve girare nel container al momento della sua esecuzione. In questo caso entrypoint.sh.

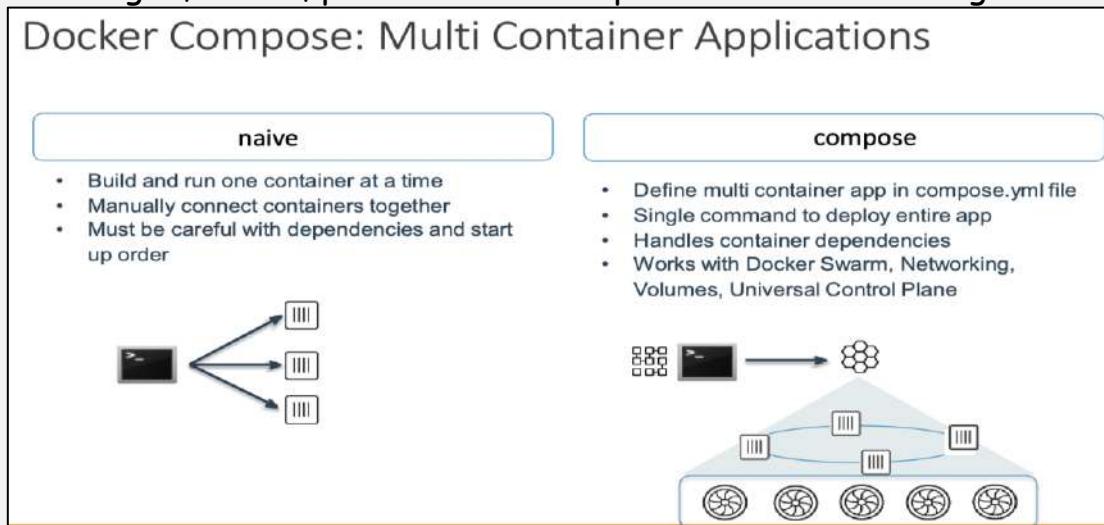
Entrypoint è più leggero di RUN e si limita solo a mandare in esecuzione un programma già installato. Non può installare un applicativo.

CMD: si utilizza per mandare in esecuzione un processo di default all'avvio del container. Può essere sovrascritto da terminale.

In sintesi: CMD viene utilizzato per specificare il comando di default da eseguire quando si crea un nuovo container partendo da un'immagine, mentre ENTRYPOINT viene utilizzato per specificare il comando principale da eseguire quando il container viene avviato e non può essere sovrascritto con un altro.

In generale, bisogna andare nella directory dove si trova lo script Dockerfile SENZA ESTENSIONE. Docker ne prende il contenuto, costruendo un'immagine sfruttando il Dockerfile che trova in quella directory.

Fino ad ora abbiamo visto il lancio di immagini per la creazione di container direttamente da client con l'utilizzo del Dockerfile che può basarsi su una sola immagine che può essere personalizzata. Se si necessita di un'applicazione che fa uso di più container possiamo utilizzare due tipologie di approcci: un approccio Naive dove si creano diversi container e si gestisce lo scambio tra di essi tutto "a mano", anche la rete che li interconnette. Utilizzando un Docker Compose il quale semplifica il tutto attraverso un unico client in cui è possibile mandare in parallelo più container specificando anche le dipendenze reciproche e la rete che li interconnette. Per utilizzarlo dobbiamo generare uno script con estensione ".yml", ovvero il docker-compose.yml all'interno del quale possiamo gestire immagini, volumi, porte... Ci sarà un problema di networking da risolvere.



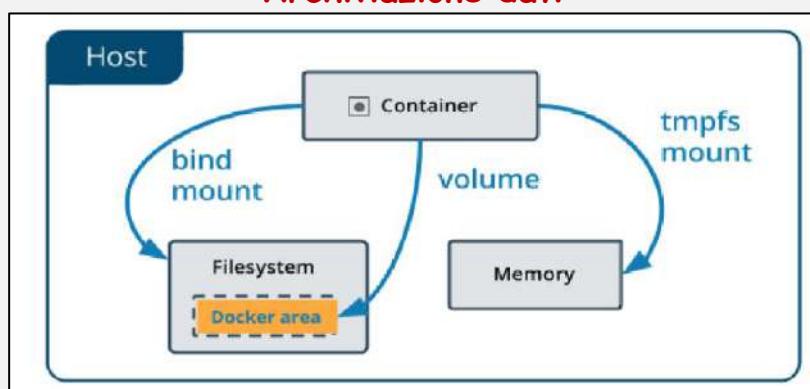
In questo caso si usano le porte del container e le porte delle applicazioni.



Per ciascun servizio si specifica dove prendere le immagini.

Dove c'è scritto build ci si aspetta un Dockerfile con cui creare il container.

Archiviazione dati



Ci sono **tre possibili approcci**:

- **tmpfs mount:** viene utilizzato quando si prevede di non conservare i dati in memoria e di eliminarli allo spegnimento o cancellazione del container. Utile quindi per evitare di lasciare dati permanenti nello strato di scrittura del container, rendendolo quindi più efficiente. Andiamo ad emulare un filesystem che quando spegniamo il container viene azzerato.
- **bind mount:** consente di associare una directory presente nell'host con una directory presente all'interno del container Docker, in modo che entrambe possano condividere lo stesso spazio di archiviazione. In pratica, si crea un collegamento tra una directory sul sistema host e una directory all'interno del container Docker, consentendo di condividere file e dati tra i due ambienti. Sono utili per accedere a file di configurazione o per salvare i dati generati all'interno del container su un disco persistente esterno.
- **volume:** montano una directory dell'host all'interno del container in una posizione specifica. Possono essere utilizzati per condividere (e rendere persistenti) i dati tra i container. La directory persiste dopo l'eliminazione del container a meno che non venga eliminata in modo esplicito. I volumi sono il meccanismo predefinito per la persistenza dei dati generati e utilizzati dai container Docker.

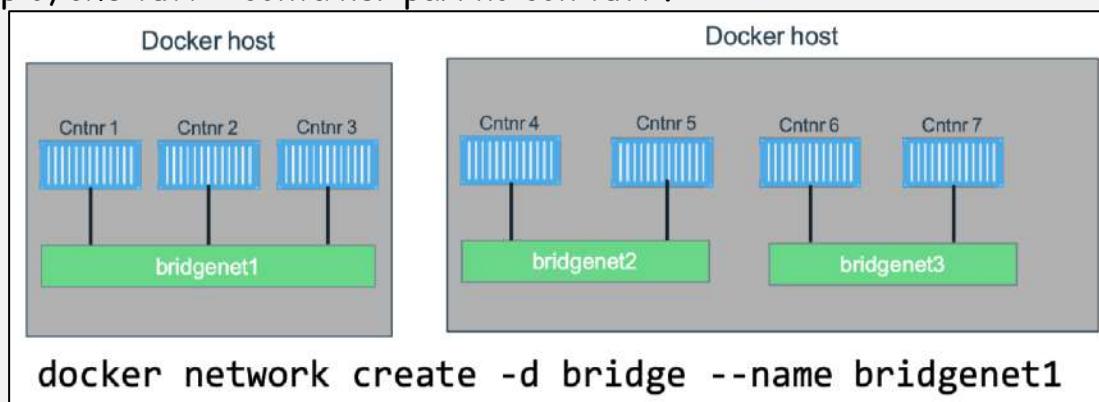
Sono directory gestite da Docker, separate dalla directory del filesystem dell'host e possono essere condivisi tra i container. Ognuno di questi metodi ha le proprie caratteristiche e funzionalità e la scelta dipende dalle esigenze dell'applicazione.

Docker Networking

I container Docker possono essere collegati insieme.

Docker Compose crea degli **switch o dei bridge virtuali** per interconnettere i vari container. Sono i migliori quando si ha bisogno di più container per comunicare sullo stesso Docker Host.

È possibile definire una rete e poi con l'opzione network dire il bridge utilizzato per connettersi con il resto del mondo. In questo modo si gestiscono le comunicazioni permettendo solo la comunicazione a gruppi e per evitare, ad esempio, che tutti i container parlino con tutti.



Ci sono diverse opzioni settabili, come la sottorete, il range degli IP, ecc. Quando si creano o si eliminano i bridge, o quando si connettono o disconnettono i container ai bridge, Docker utilizza una serie di tool specifici per operare con il sistema e gestire la sottostante architettura di rete, in maniera simile alla creazione della tabella IP in Linux.

Orchestratori di Container

Per poter scalare e gestire i vari Container abbiamo bisogno di un **orchestratore**, ovvero un framework di gestione.

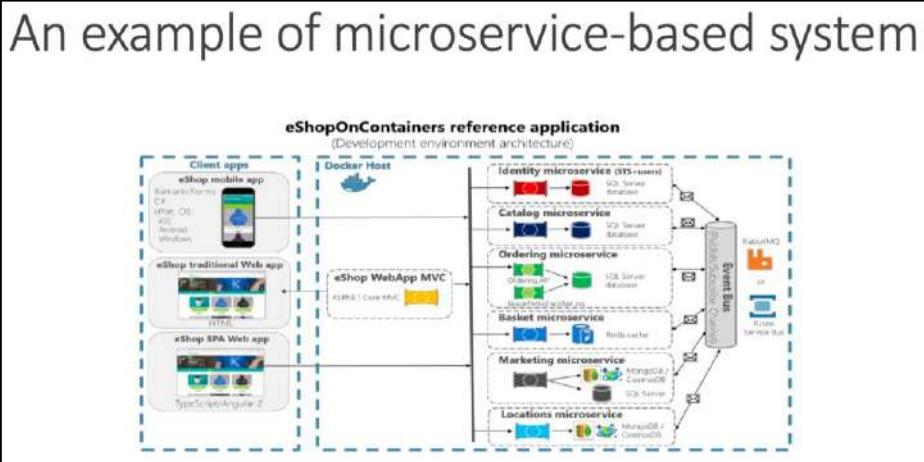
I principali gestori di Container Docker sono al giorno d'oggi **Docker Swarm** e **Kubernetes**. Sono entrambi framework utilizzati per gestire e orchestrare i container Docker su un **cluster di nodi**. Entrambi i framework consentono di gestire la scalabilità e l'affidabilità delle applicazioni containerizzate, ma hanno differenze nella loro architettura e nelle loro funzionalità.

Docker Swarm è una soluzione di orchestrazione dei Container fornita da Docker stessa. Swarm consente di gestire un cluster di nodi Docker in modo centralizzato, fornendo funzionalità di bilanciamento del carico, distribuzione degli aggiornamenti e ripristino automatico in caso di guasti di un nodo.

Kubernetes è un framework open source per l'orchestrazione dei Container sviluppato da Google. Kubernetes consente di gestire in modo efficiente cluster di nodi con centinaia o migliaia di container, gestendo la scalabilità, la disponibilità e la gestione degli aggiornamenti. Kubernetes utilizza un modello di dichiarazione dello stato desiderato, dove lo stato desiderato viene dichiarato attraverso un file di configurazione e Kubernetes lavora per raggiungere tale stato desiderato. Kubernetes ha un'architettura altamente scalabile e modulare. Controlla anche lo stato di salute dei container e se non sono in salute li istanzia di nuovo in automatico, in cloud questo è fondamentale.

In sintesi, Docker Swarm e Kubernetes sono entrambi framework utilizzati per gestire e orchestrare i container Docker su un cluster di nodi. Docker Swarm è più semplice da utilizzare, ma ha meno funzionalità rispetto a Kubernetes.

Kubernetes, d'altra parte, è più complesso ma offre un'architettura altamente scalabile e modulare.

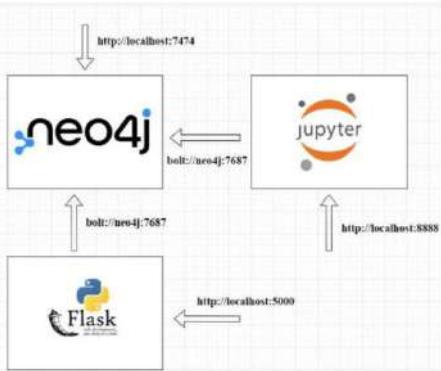


Un broker è uno strumento che implementa una coda di quello che dev'essere spedito. A manda un qualcosa al Broker che poi lo manda a B.

Inviandola al Broker, A è sicuro che prima o poi il pacchetto arriva a B.

Così facendo A può schiodarsi dal dover aspettare la certezza che il messaggio arrivi a B e può dedicarsi ad altro (ad esempio, andare in spiaggia a rilassarsi).

Architecture



Esempio di un'architettura multi-container che utilizza Flask, Neo4j e Jupyter Notebook.

The Docker Compose

```
File Edit Selection View Go Run Terminal Help
docker-compose.yml X
docker-compose.yml
1 version: '3.3'
2
3 services:
4
5   app:
6     build: .
7     container_name: graphapp
8     restart: always
9     depends_on:
10       - neo4j
11     ports:
12       - "5000:5000"
13     command: python main.py
14     networks:
15       - graph_network
16
```

```
17
18   neo4j:
19     image: neo4j:4.3.7-enterprise
20     container_name: neo4j
21     restart: always
22     ports:
23       - "7474:7474"
24       - "7687:7687"
25     volumes:
26       - ./data:/data
27     environment:
28       - NEO4J_ACCEPT_LICENSE AGREEMENT=yes
29       - NEO4J_AUTH=neo4j/pierluigi
30     networks:
31       - graph_network
32
33   jupyter:
34     image: jupyter/minimal-notebook:latest
35     container_name: jupyter
36     restart: always
37     depends_on:
38       - neo4j
39     ports:
40       - "8888:8888"
41     networks:
42       - graph_network
43
44   networks:
45     graph_network:
46       name: graph_network
```

build: avvia il dockerfile e tutto ciò che c'è dentro.
depends_on: stabilisce una dipendenza, il container viene avviato dopo il container da cui dipende.
restart: riavvio del container in caso di malfunzionamenti.

command: eseguiamo l'applicazione python main.py

networks: indichiamo il bridge a cui connettere i container.

In questo esempio applicativo si avvia per prima cosa il database di neo4j perché il programma basa il suo funzionamento su di esso.



Il token di autorizzazione serve per accedere durante la fase di autenticazione su jupyter. Il comando `docker exec` con l'id del container permette di navigare dentro quest'ultimo e visualizzare le informazioni. Come i processi attivi, ecc.

In questo esempio applicativo non c'è un'orchestrazione dei container. Ci siamo limitati solo al loro istanziamento e alla loro reciproca connessione.

Capitolo 3: Kubernetes

Kubernetes è un sistema open-source per l'orchestrazione di container che semplifica la gestione di applicazioni containerizzate su larga scala. Kubernetes consente di gestire e scalare le applicazioni in modo affidabile e automatizzato, garantendo una maggiore disponibilità e affidabilità delle applicazioni.

Kubernetes fornisce una vasta gamma di funzionalità per la gestione dei container, come l'allocazione delle risorse, il bilanciamento del carico, il ripristino automatico, la scalabilità orizzontale e la gestione delle configurazioni. Inoltre, Kubernetes offre funzionalità avanzate per la gestione della sicurezza, del monitoraggio e del logging delle applicazioni.

Consente di orchestrare container attraverso un cluster (insieme di host).

Consente di aggiornare i container mantenendo l'accesso da parte degli utenti.

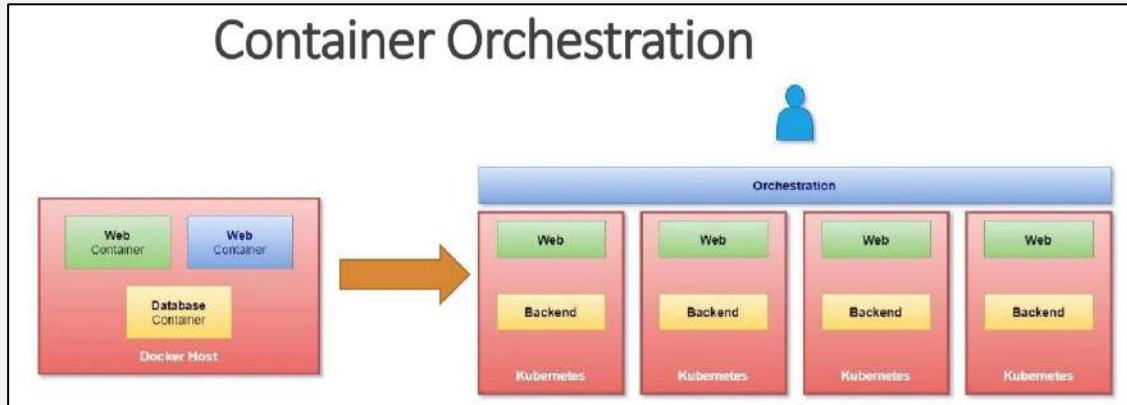
Per ogni container, è possibile impostare delle repliche.

Inoltre, Kubernetes consente di definire una configurazione dell'applicazione attraverso un unico file YAML o JSON, semplificando l'automatizzazione della configurazione dell'applicazione stessa.

Consente di far girare le applicazioni fintanto che noi vogliamo farle girare.

La peculiarità principale di Kubernetes, che lo ha reso di fatto un must nel mondo dell'informatica, è il monitoraggio della salute dei container che gestisce: esegue continui check relativi allo stato di salute dei container e in caso di malfunzionamenti di qualcuno di essi procede con il ripristino automatico. Questo garantisce la continua accessibilità di un qualunque servizio, cosa fondamentale nel mondo di oggi (immaginiamo se un sito di e-commerce avesse fuori uso per un tot intervallo di tempo il meccanismo di pagamento).

Quindi, potremmo immaginare di avere un cluster di macchine fisiche con una serie di repliche dove in ognuna è presente **Kubernetes** che **orchestra il tutto secondo la politica stabilità nei file di configurazione appositi** (**l'approccio dichiarativo** è il più utilizzato in quanto è più conforme alla logica di funzionamento di Kubernetes).



L'applicazione, in questo modo, è altamente disponibile in quanto abbiamo molteplici istanze di essa contenute in diverse macchine fisiche.

Il traffico inoltre viene bilanciato attraverso tutti i nodi disponibili (e non solo su uno) andando ad aumentare o diminuire il numero dei nodi necessari in base alle esigenze (esempio: Amazon durante il Black Friday incrementa il numero container per gestire le numerose richieste di persone attratte dagli sconti).

Kubernetes Pod

In Kubernetes, un **pod** rappresenta l'**unità minima di esecuzione** che contiene uno o più container, un'unità di archiviazione condivisa e un IP di rete condiviso. I container all'interno di un pod **condividono lo stesso spazio di rete** e possono comunicare tra di loro utilizzando come indirizzo il localhost.

Dentro un pod c'è un solo container nel 90% dei casi.

In caso di più container allora è presente anche il supporto al networking.

In generale conviene restare nell'ambito di lavoro specifico (micro-servizio).

Un pod dev'essere un qualcosa di "piccolo" e se, ad esempio, si occupa di logging allora non ha senso inserire altre funzioni.

Due container in un pod possono esserci nel caso in cui uno dei due container deve comunicare con l'esterno, fungendo da **proxy**.

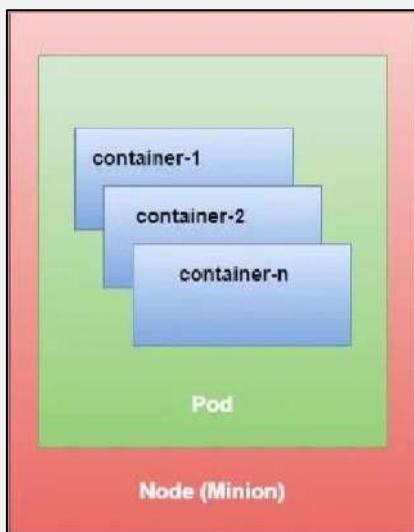
Per ottenere la persistenza dei dati si utilizzano, come in Docker, i **volumi**.

Il concetto di pod semplifica la gestione dei container all'interno di Kubernetes: non si opera direttamente sui container ma essi vengono inscatolati all'interno di un pod.

I pod in Kubernetes sono la più piccola unità di gestione, poiché Kubernetes gestisce i pod come un'entità coesa. Ad esempio, **Kubernetes gestisce il bilanciamento del carico e la scalabilità dei pod e non dei singoli container**. In Kubernetes, un pod viene creato a partire da un'immagine di un container e viene configurato tramite un file YAML o JSON.

Il file di configurazione del pod può specificare le risorse necessarie, le porte esposte e le variabili d'ambiente per i container all'interno del pod.

Ogni pod ha il proprio indirizzo IP univoco (che va a risolvere i problemi legati all'utilizzo delle porte). Quando un pod viene riavviato, in realtà viene creato un nuovo pod con un nuovo indirizzo IP, mentre il vecchio pod viene eliminato.

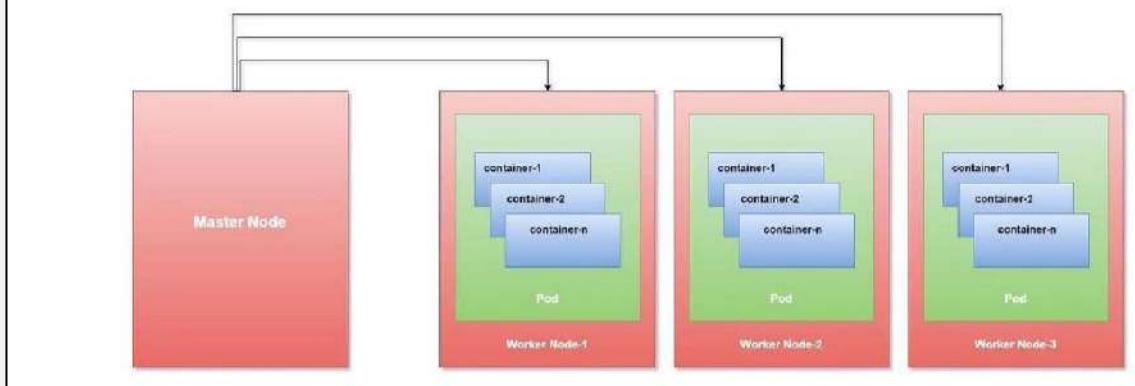


Nodo: un insieme di container, pod e network.

Si tratta di singoli host (fisici o virtuali), su cui Docker verrebbe installato per ospitare diversi container, all'interno del cluster.

Kubernetes Architecture: cluster

Kubernetes Architecture: cluster



L'architettura di Kubernetes possiamo definirla di tipo **master-worker**.

Un cluster Kubernetes è costituito da un insieme di macchine worker, denominate nodi, che eseguono applicazioni containerizzate.

Ogni cluster include almeno un nodo di lavoro.

I nodi di lavoro ospitano i pod, che implementano le applicazioni containerizzate.

Il master gestisce i nodi di lavoro e i pod del cluster.

In sintesi, il Master è il cervello mentre i Worker sono coloro che lavorano.

1 Master e 1 Worker devono esserci per costruzione.

Kubernetes contiene tutte le informazioni dei nodi del cluster, monitora ogni nodo e se un Worker Node fallisce allora sposterà il carico di lavoro dal nodo guasto a un altro nodo di lavoro funzionante.

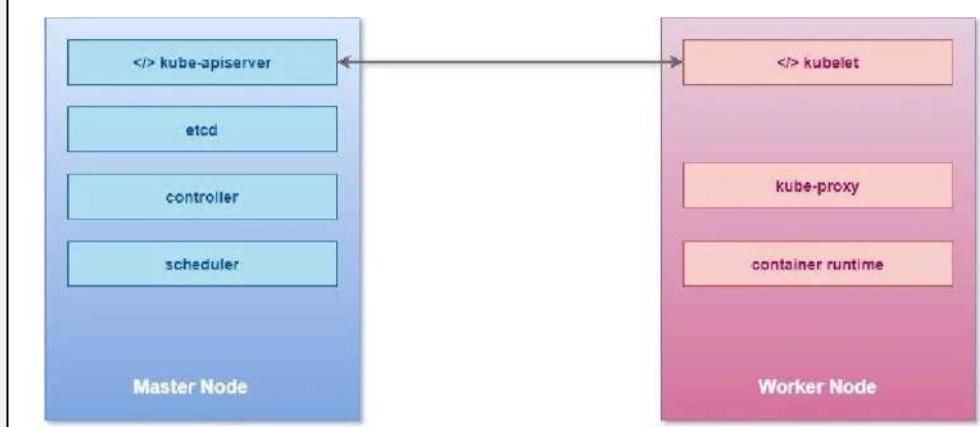
Un nodo è una macchina fisica o virtuale su cui è installato Kubernetes.

Kubernetes non distribuisce i container direttamente nei worker ma li incapsula in oggetti Kubernetes noti come pod.

Un pod è una singola istanza di un'applicazione ed è l'unità di calcolo distribuibile più piccola che si possa creare e gestire in Kubernetes.

Distribuiamo i container all'interno dei pod, dove si distribuisce l'applicazione.

Kubernetes Components



Nodo Master

Contiene il punto di accesso al cluster Kubernetes, ovvero **kube-apiserver**.

Contiene **etcd**, un database distribuito di tipo chiave-valore d'importanza fondamentale. Esso, infatti, contiene lo stato di tutti gli elementi presenti nel

cluster. Qualsiasi modifica fatta all'interno del cluster avrà un riflesso nel database etcd. Lo stato desiderato viene confrontato con quello presente all'interno del db e, in caso non siano uguali, verrà eseguita una certa azione. Il master dev'essere ridondante per motivi di sicurezza.

La replica dell'etcd, per via della sua importanza, è indipendente dal master.

Abbiamo poi i controller che costituiscono il cervello dietro l'orchestrazione. Sono i responsabili dell'avviso e della risposta quando i nodi, i pod o gli endpoint non funzionano e posso decidere, all'occorrenza, di avviare nuovi pod.

Abbiamo poi lo scheduler, il cui compito è decidere a quali worker andare ad assegnare i nuovi pod istanziati.

È un'operazione delicata che dipende da tanti parametri.

La decisione di stanziare un pod all'interno di un nodo, infatti, non ha una risposta binaria ma dipende da fattori come, ad esempio, le risorse disponibili.

Il nodo master è anche chiamato con il termine piano di controllo.

Il Master si occupa di:

- Esporre l'API REST di Kubernetes.
- Programmare le applicazioni.
- Gestire il cluster.
- Dirigere le comunicazioni dell'intero sistema.
- Monitoraggio dei container in esecuzione in ciascun pod, nonché dello stato di salute di tutti i nodi appartenenti al cluster.

Nodo Worker

All'interno dei worker abbiamo kubelet che di fatto rappresenta l'elemento più importante in questi ultimi. È un processo demone eseguito su ciascun nodo del cluster. Quest'agente si occupa di verificare che i container siano in esecuzione nei vari nodi correttamente. Comunica con kube-apiserver del Master per controllare quale sia lo stato desiderato e verificare quindi se è necessario mandare in funzione i pod. Potrebbe comunicare anche i cosiddetti Pod Secrets: informazioni segrete per stanziare determinati pod sensibili e che non possono essere lasciate in chiaro. Riportano il suo stato al master che lo salva nell'etcd.

C'è poi il kube-proxy, il quale si occupa della comunicazione con l'esterno.

Il container runtime è il software sottostante responsabile per l'esecuzione dei container. Kubernetes supporta diversi container runtimes.

Nel nostro caso utilizzeremo Docker.

Bisogna tener conto anche della cosiddetta Data Locality, ovvero dove poter salvare i dati. È la policy a cui bisogna fare riferimento. Il GDPR. Ad esempio, ci possono essere determinate normative, come, il divieto di conservare dati in IRAN o in Nazioni potenzialmente pericolose.

Creazione di un pod

Un pod può essere creato attraverso un file YAML o da terminale.

L'approccio da terminale è imperativo in quanto si specifica passo dopo passo come bisogna procedere per ottenere un risultato. L'approccio con file YAML è dichiarativo perché viene specificato solo il risultato che si vuole ottenere, lasciando libera interpretazione a Kubernetes su come raggiungerlo.

Creation of a simple Pod

A Creating Pods using YAML file

Pods and other Kubernetes resources are usually created by posting a JSON or YAML manifest to the Kubernetes REST API endpoint.

```
[root@controller ~]# cat nginx.yml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: default
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 80
```

Creazione di un pod che mette in funzione un container che esegue al suo interno un'applicazione nginx.

Ogni volta che un pod viene creato, a prescindere da quanti container esso contiene, avrà sempre un container nascosto, il **container pause**. Quest'ultimo crea un'interfaccia di rete comune tra tutti i container all'interno di un pod.

Quindi, tutti i container dentro lo stesso pod hanno la stessa interfaccia di rete, ovvero lo stesso indirizzo MAC e lo stesso indirizzo IP.

I container in un pod perdono la loro individualità a livello di indirizzi e di rete.

Kubernetes Namespaces

In Kubernetes, un namespace è un meccanismo per isolare e dividere le risorse all'interno di un cluster. Un namespace consente di creare più ambienti logici all'interno di un cluster, in modo che le risorse di un ambiente non interferiscano con quelle di un altro ambiente.

Ad esempio, si potrebbe creare un namespace per l'ambiente di sviluppo, uno per l'ambiente di test e uno per l'ambiente di produzione. All'interno di ciascun namespace si possono creare e gestire delle risorse specifiche per quell'ambiente, come, ad esempio i pod, i servizi, i volumi e le configurazioni.

Ogni risorsa all'interno di Kubernetes appartiene a un namespace specifico e il namespace predefinito è "default". In questo modo, se non viene specificato un namespace, la risorsa viene assegnata al namespace predefinito.

Il namespace di Kubernetes funziona come un isolatore logico all'interno del cluster, dove le risorse vengono gestite in modo separato, mantenendo l'ambiente di sviluppo, di test e di produzione separati e isolati l'uno dall'altro. I nomi dei namespace devono essere univoci. Le risorse all'interno di diversi namespace possono invece avere gli stessi nomi.

Bisogna creare un namespace per ogni utente, in quanto ognuno vorrà il suo ambiente isolato. In questo modo è possibile offrire servizi a più utenti insieme. Possiamo pensare a un namespace come ad una cartella con dentro degli oggetti.

La connessione tra pod di diversi namespace è comunque possibile, nel caso in cui si conosca l'indirizzo IP di un pod, infatti, è possibile inviare traffico.

Ci sono 4 namespace che Kubernetes utilizza quando un cluster viene creato:

- Default: è quello dove di base sono assegnate le risorse.
- Kube-Node-Lease: riservato a scopi amministrativi.
- Kube-Public: contiene delle informazioni accessibili a tutti.
- Kube-System: contiene i pod per la gestione del sistema e contiene informazioni riguardanti il cluster.

Creating a namespace

Now, use kubectl to post the file to the Kubernetes API server:

```
[root@controller ~]# kubectl create -f create-namespace.yml  
namespace/app created
```

Using kubectl command: You can also create namespaces with the dedicated kubectl create namespace command:

```
[root@controller ~]# kubectl create ns dev  
namespace/dev created
```

Note the possibility of using both declarative and imperative styles.

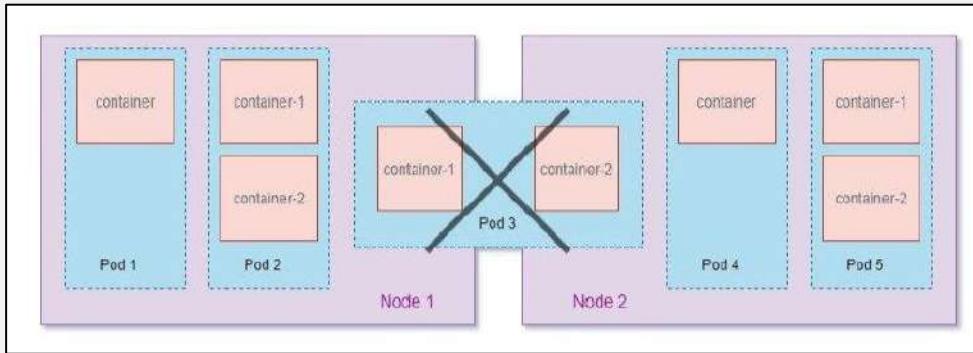
To get a much detailed output of individual namespace (e.g. default):

```
[root@controller ~]# kubectl describe ns default
```

To get the details of namespace in YAML format:

```
[root@controller ~]# kubectl get ns default -o yaml
```

L'aspetto fondamentale dei pod è che, anche quando contengono più container, vengono sempre eseguiti su un singolo nodo di lavoro e mai su più worker.



Tendenzialmente, come abbiamo ripetuto più volte, un pod non necessariamente ha al suo interno più container, bensì uno soltanto.

Il vantaggio è che pod piccoli posso essere "riciclati" ovunque.

Inoltre, meno applicativi ci sono nei pod e meno esposizioni a rischi ci sono in termini di sicurezza informatica.

Tuttavia, esistono casi in cui si potrebbe dover utilizzare più container che collaborano tra di loro all'interno di un pod, ovvero:

- **Sidecar**: container di supporto per l'applicazione principale, per esempio per svolgere operazioni di logging.
- **Ambassador**: container di supporto che funge da proxy verso l'esterno.
- **Adapter**: container di supporto che trasforma i dati di traffico entranti prima di cederli al database, rendendoli di fatto compatibili.
- **Pause**: container di supporto che si mette in pausa una volta che ha concluso il suo lavoro. Essenzialmente mantiene in memoria lo stato di un container che può essere rispristinato in caso di problemi.

Example of Sidecar Scenario

```
[root@controller ~]# cat create-sidecar.yml
kind: Pod
apiVersion: v1
metadata:
  name: sidecar-pod
spec:
  volumes:
    - name: logs
      emptyDir: {}
  containers:
    - name: app
      image: busybox
      command: ["/bin/sh"]
      args: ["-c", "while true; do date >> /var/log/date.txt; sleep 10; done"]
      volumeMounts:
        - name: logs
          mountPath: /var/log
    - name: sidecar
      image: centos/httpd
      ports:
        - containerPort: 80
      volumeMounts:
        - name: logs
          mountPath: /var/www/html
[root@controller ~]# kubectl create -f create-sidecar.yml
```

Occorre utilizzare dei volumi perché sidecar è un container utilizzato per operazioni di log. Perdere tali informazioni porta all'inutilità del container stesso.

Oggetti Kubernetes

Gli oggetti Kubernetes sono delle istanze di risorse fornite dal software stesso che permettono lo sviluppo, il mantenimento e la scalabilità delle applicazioni.

Kubernetes utilizza questi oggetti per rappresentare lo stato del cluster.

Gli oggetti Kubernetes vengono creati utilizzando dei file manifest in formato YAML, principalmente attraverso un approccio dichiarativo e non attraverso un approccio imperativo (mediante linea di comando).

Un problema potrebbe essere il dispendio di tempo nel dover creare per ogni oggetto un file manifest YAML annesso ma possiamo risolverlo semplicemente andando a prendere un file yaml da un oggetto già esistente mediante il comando `kubectl get <object> -o yaml` per poi utilizzarlo come **template** per creare un nuovo oggetto Kubernetes.

- Job
- ReplicaSet
- StatefulSets
- Deployment
- DaemonSet
- CronJob
- ReplicationController

Questi in elenco sono **oggetti Kubernetes** considerati di basso livello. Fatta eccezione per **DaemonSet** e **Deployment** che sono più **complessi**.

Deployment è un oggetto che può possedere altri oggetti di tipo ReplicaSet e aggiornarli insieme ai relativi pod tramite aggiornamenti progressivi dichiarativi lato server. Sebbene i **ReplicaSet** possano essere utilizzati in modo indipendente, oggi vengono utilizzati principalmente dai Deployment come meccanismo per orchestrare la creazione, l'eliminazione e gli aggiornamenti dei pod. Quando si utilizzano gli oggetti Deployment non occorre preoccuparsi di gestire i ReplicaSet che si creano.

L'oggetto **ReplicationController** assicura che un numero specificato di repliche di pod sia in esecuzione in qualsiasi momento.

In altre parole, un ReplicationController si assicura che un pod o un insieme omogeneo di pod sia sempre attivo e disponibile.

Se sono presenti troppi pod, ReplicationController termina i pod aggiuntivi.

Se sono troppo pochi, ReplicationController avvia più pod.

L'oggetto **Job** esegue il lavoro, ovvero un processo che termina nel tempo.

Ad esempio, un'operazione di backup.

Esegue qualcosa fino a terminarla. Una, due, tre volte, etc.

Può creare pod garantendo la loro terminazione un numero specifico di volte.

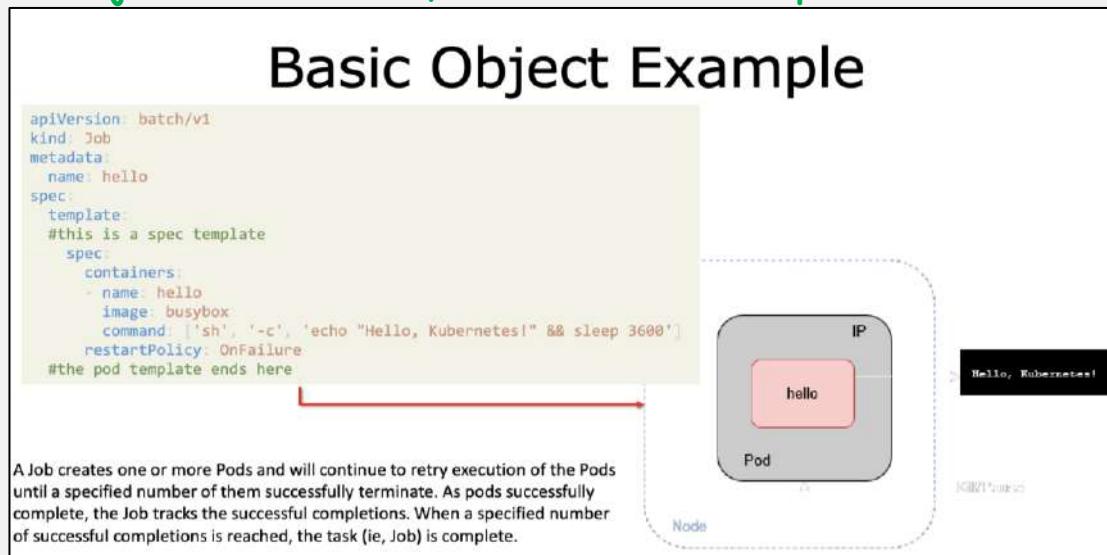
Ai job è associato un tempo di vita.

In caso di malfunzionamenti occorre specificare una **policy di restart**: con **OnFailure** si riavvia il container nello stesso pod in caso di fallimento, con **Never** si riavvia il container in un pod diverso.

Esistono tre differenti tipologie di job che possono essere istanziati regolando due parametri principali, ovvero **completamento** (quante volte un job deve portare a compimento con successo il lavoro) e **parallelo** (quanti job possono essere eseguiti contemporaneamente):

- **Non-parallel Jobs**: si avvia un pod, a meno che non presenti errori (completions=1, parallelism=1).
- **Parallel Jobs with a fixed completion count**: job eseguiti in parallelo che devono terminare con successo il loro compito n volte (completions=n, parallelism=m).
- **Parallel Jobs with a work queue**: vengono avviati più job in parallelo ma per terminare è sufficiente che uno solo di essi termini con successo. (completions=1, parallelism=m).

Quando un job viene eliminato, si eliminano anche i pod ad esso annessi.



Mandando in esecuzione questo esempio si ottiene un container che dice "Hello, Kubernetes!" per poi dormire per un'ora. Terminato lo sleep, termina il suo job. Il nome del pod, se non specificato, prende il nome del job che lo istanzia.

La sezione "**spec**" (abbreviazione di "specification") definisce le specifiche dell'oggetto Kubernetes, tra cui le sue proprietà e le configurazioni che determinano il comportamento dell'oggetto.

La sezione "**command**" specifica l'elenco di comandi da eseguire all'interno del container.

Example: Running job pods sequentially

If you need a Job to run more than once, you set completions to how many times you want the Job's pod to run.

```
[root@controller ~]# cat pod-simple-job.yml
apiVersion: batch/v1
kind: Job
metadata:
  name: pod-simple-job
spec:
  completions: 3
  template:
    spec:
      containers:
        - name: sleepy
          image: alpine
          command: ["/bin/sleep"]
          args: ["5"]
      restartPolicy: Never
```

```
[root@controller ~]# kubectl create -f pod-simple-job.yml
job.batch/pod-simple-job created
```

After some time, once all the jobs are completed:

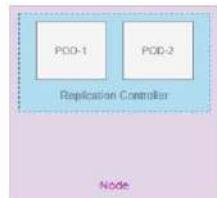
```
[root@controller ~]# kubectl get jobs
NAME      COMPLETIONS DURATION AGE
pod-simple-job 3/3           36s   3m36s
```

La sezione "args" (abbreviazione di "arguments") è utilizzata per specificare gli argomenti di linea di comando da passare a un container in un oggetto Kubernetes. In questo esempio specifichiamo l'intervallo di esecuzione (5).

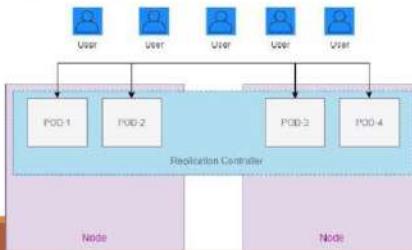
ReplicationController and ReplicaSet

A ReplicationController is a Kubernetes resource that ensures its pods are always kept running.

- If the pod disappears for any reason, such as in the event of a node disappearing from the cluster or because the pod was evicted from the node, the ReplicationController notices the missing pod and creates a replacement pod.
- The ReplicationController in general, are meant to create and manage multiple copies (replicas) of a pod



It is possible that your Node is out of resources while creating new pods with Replication controllers or replica sets, in such case it will automatically create new pods on another available cluster node



I pod sono inseriti nel cluster, che è gestito dal master.

Se è presente un solo nodo allora andranno tutti in quest'ultimo.

Se sono presenti più nodi allora andranno in uno qualsiasi di essi.

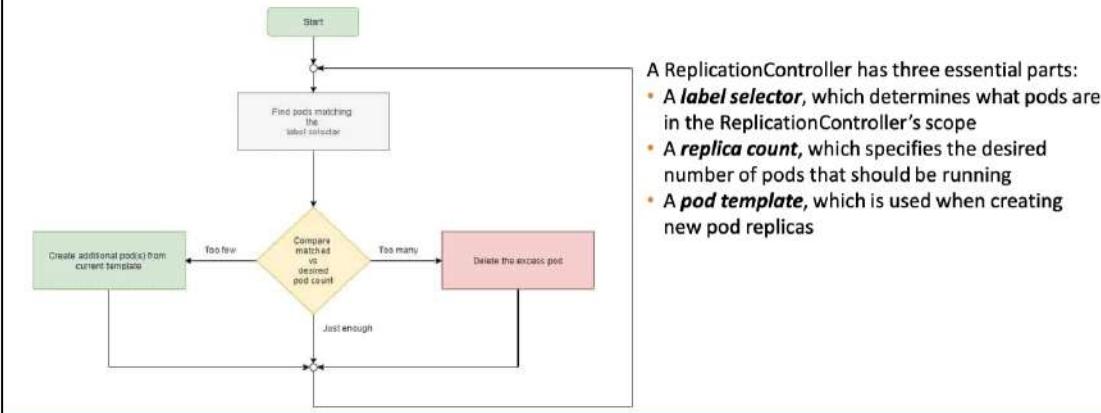
Non abbiamo il controllo su dove siano situati i pod.

Chi ha quest'informazione è il ReplicationController, che gira dentro al master.

Lo Scheduler, che gira dentro al master, decide, invece, dove allocarli.

ReplicationController and ReplicaSet

A ReplicationController's task is to make sure that an **exact number of pods always matches its label selector**. If it doesn't, the ReplicationController takes the appropriate action to reconcile the actual with the desired number. The ReplicationController in general, are meant to create and manage multiple copies (replicas) of a pod



Andando nello specifico, possiamo visualizzare il lavoro del ReplicationController con un diagramma ER. Essenzialmente ci sono tre parti che costituiscono quest'oggetto Kubernetes e si trovano tutte nel file yaml.

Label selector: seleziona la label indicata, scansione tutti gli oggetti e seleziona appunto quelli con la label desiderata. Ad esempio, si potrebbero selezionare determinati pod con una certa label e replicarli di un fattore x.

Replica count: controlla il numero di pod che matchano, se sono come il numero specificato dal count allora resta fermo, se sono pochi ne aggiunge, se sono troppi li elimina. Garantisce esattamente il numero di pod specificato nel count.

Pod template: indica un modello che viene utilizzato durante la creazione di nuove repliche di pod.

ReplicationController and ReplicaSet

```
[root@controller ~]# cat replication-controller.yml
apiVersion: v1
kind: ReplicationController
metadata:
  name: myapp-rc
  labels:
    app: myapp
    type: dev
spec:
  replicas: 3
  selector:
    app: myapp
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: dev
    spec:
      containers:
        - name: nginx-container
          image: nginx
```

Horizontally scaling pods

You've seen how ReplicationControllers make sure a specific number of pod instances is always running. Because it's incredibly simple to change the desired number of replicas, this also means scaling pods horizontally is trivial.

Assuming you suddenly expect that load on your application is going to increase so you must deploy more pods until the load is reduced, in such case you can easily scale up the number of pods runtime. For example, for having 6 replicas:

```
[root@controller ~]# kubectl scale rc myapp-rc --replicas=6
replicationcontroller/myapp-rc scaled
```

For downscaling to 3 replicas:

```
[root@controller ~]# kubectl scale rc myapp-rc --replicas=3
replicationcontroller/myapp-rc scaled
```

Le label servono per richiamarsi a livello gerarchico.

Vengono utilizzate per determinate operazioni. In questo caso, ad esempio, la label del ReplicationController è presente perché ci sarà un oggetto di più alto livello che la utilizzerà per richiamare il ReplicationController stesso!

Per quanto riguarda le repliche, se ci si accorge che sono poche allora è possibile utilizzare il comando indicato in blu nella figura sopra.

Stesso comando se ci si accorge che le repliche sono troppe.

Si utilizza dunque per scalare i pod.

Non è però la soluzione ottimale perché ci dovrebbe essere una persona in tempo reale che sta lì a controllare, non può essere professionale.

Per essere professionale bisogna automatizzare il processo.

Kubernetes permette sia l'autoscaling orizzontale (aumentare il numero di istanze, nuovi pod) sia l'autoscaling verticale (aumentare le risorse delle istanze già presenti). In generale, ReplicationController e ReplicaSet fanno la stessa cosa, cioè mettere in opera un numero predefinito di pod.

Con il ReplicaSet si è però più raffinati, è possibile, ad esempio, agire su pod che hanno una determinata chiave come label, senza badare al loro valore.

Per questo e altri motivi, il ReplicationController è ormai caduto in disuso ed è stato rimpiazzato dall'utilizzo esclusivo del ReplicaSet.

ReplicationController and ReplicaSet

```
[root@controller ~]# cat replica-set.yml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
    app: myapp
    type: dev
spec:
  replicas: 3
  selector:
    matchExpressions:
      - key: app
        operator: In
        values:
          - myapp
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: dev
    spec:
      ...
```

Now, we will rewrite the selector to use the more powerful matchExpressions property:
Here, this selector requires the pod to contain a label with the "app" key and the label's value must be "myapp".

You can add additional expressions to the selector. As in the example, each expression must contain a key, an operator, and possibly (depending on the operator) a list of values. You'll see four valid operators:

- **In**: Label's value must match one of the specified values.
- **NotIn**: Label's value must not match any of the specified values.
- **Exists**: Pod must include a label with the specified key (the value isn't important). When using this operator, you shouldn't specify the values field.
- **DoesNotExist**: Pod must not include a label with the specified key. The values property must not be specified.

Il kind specifica la tipologia. Ad esempio, se ReplicaSet o ReplicationController.

Il ReplicaSet ha una sintassi leggermente diversa nella sezione selector.

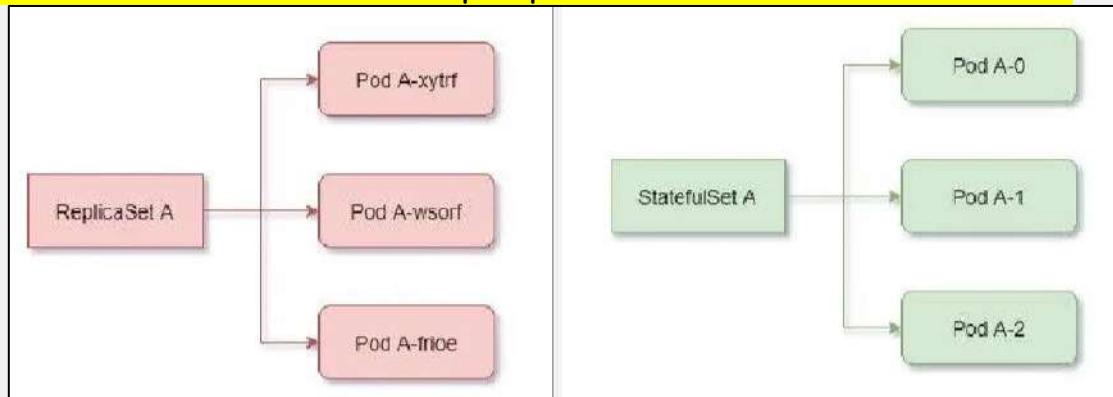
In questo esempio si vanno a selezionare tutti i pod che hanno come valore myapp quando la chiave è app.

È possibile aggiungere espressioni aggiuntive nel selector per renderlo ancora più flessibile. Ad esempio: In seleziona gli oggetti che hanno un valore specifico per l'etichetta indicata, NotIn seleziona gli oggetti che non hanno un valore specifico per l'etichetta indicata, Exists seleziona gli oggetti che hanno

un'etichetta specificata, indipendentemente dal valore associato, **DoesNotExist** seleziona gli oggetti che non hanno un'etichetta specificata. I **ReplicaSet** creano più repliche di pod da un singolo template di essi. Tali repliche non differiscono tra loro, se non per il nome e l'indirizzo IP.

Invece di utilizzare un ReplicaSet per creare questi tipi di pod, è possibile utilizzare l'oggetto **StatefulSet**, il quale ha come obiettivo quello di far mantenere l'individualità dei pod. È adatto per tutte quelle applicazioni in cui le istanze devono essere trattate come identità simili (stesse specifiche) ma non interscambiabili, ognuna con un nome e uno stato stabili che vengono preservati anche a fronte di re-scheduling.

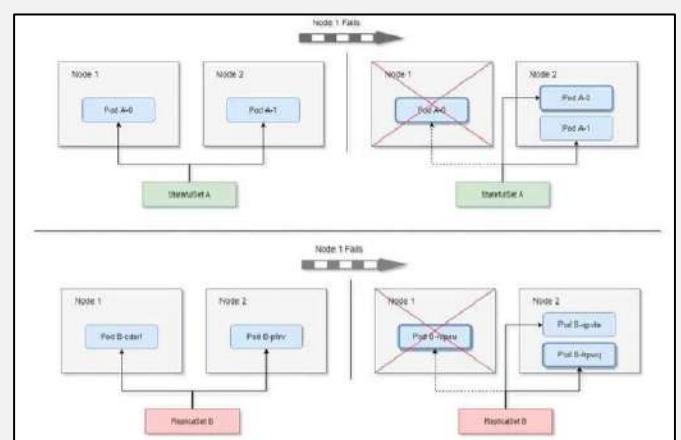
Non sappiamo dove sono ma sappiamo che esistono e che hanno un certo nome. A differenza del ReplicaSet, infatti, StatefulSet genera nomi chiari e sequenziali, formati dal nome del pod più un indice ordinale in base zero.



È più ordinato e preciso, si può utilizzare, ad esempio, per i volumi di archiviazione per fornire persistenza al carico di lavoro.

Se un pod gestito dall'oggetto StatefulSet cade a causa di problemi del nodo che lo ospita o altro, esso verrà istanziato nuovamente mantenendo lo stesso nome e hostname del pod caduto, a differenza del ReplicaSet che invece lo istanzia nuovamente senza mantenere nulla del pod precedente.

Per riassumere, Kubernetes StatefulSet gestisce l'implementazione e il ridimensionamento di un set di pod e fornisce garanzie sull'ordine e l'unicità di questi pod. Presenta comunque delle limitazioni da considerare, come il fatto di non garantire l'eliminazione dei pod in caso di un loro fallimento o il fatto di non eliminare i volumi ad essi associati in caso sempre di un loro fallimento.



DaemonSet

L'oggetto DaemonSet viene utilizzato per garantire che un'istanza di un pod sia eseguita su tutti i nodi del cluster o solo in alcuni nodi specifici.

Il DaemonSet è progettato per eseguire un'istanza di un'applicazione su ogni nodo del cluster, garantendo che l'applicazione sia in esecuzione in modo coerente su tutti i nodi. Se ci sono nodi che si aggiungono al cluster allora il DaemonSet istanzia una copia del pod su di essi.

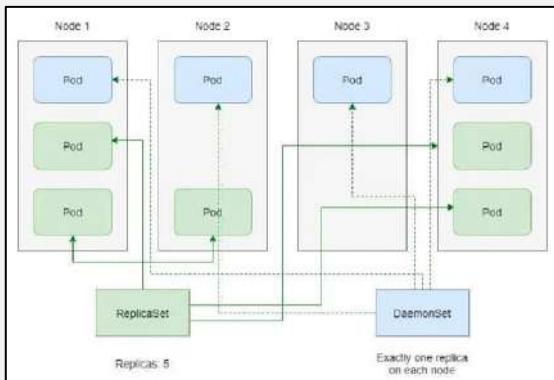
L'eliminazione di un DaemonSet comporta l'eliminazione dei pod che ha creato.

Alcuni usi tipici di un DaemonSet sono:

- Esecuzione di un demone di archiviazione del cluster su ogni nodo.
- Eseguire un demone per la raccolta dei log su ogni nodo.
- Eseguire un demone per il monitoraggio dei nodi su ogni nodo.

In un caso semplice, per ogni tipo di demone verrebbe utilizzato un DaemonSet, che copre tutti i nodi. Una configurazione più complessa potrebbe utilizzare più DaemonSet per un singolo tipo di demone.

Quindi, il DaemonSet gira nel nodo Master ed è utilizzato anch'esso per la gestione dei pod proprio come fanno Deployment, ReplicaSet e StatefulSet.



Con i ReplicaSet si replicano i pod e vengono messi nei vari nodi secondo la politica specificata dello scheduler.

Con il DaemonSet si garantisce che si sta istanziando un pod in ogni nodo presente.

Il ReplicaSet dovrebbe essere utilizzato quando si esegue un'applicazione completamente disaccoppiata dai nodi del cluster e che quindi può girare con diverse copie di sé stessa in vari nodi in modo indifferente.

Il DaemonSet dovrebbe essere utilizzato quando si esegue un'applicazione che deve necessariamente girare su tutti i nodi del cluster o su di un sottoinsieme.

Esempio di un DaemonSet per la gestione di operazioni di logging col supporto del volume per una memoria permanente.

```
DaemonSet

[root@controller ~]# cat fluentd-daemonset.yml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd
  labels:
    app: fluentd
spec:
  selector:
    matchLabels:
      app: fluentd
  template:
    metadata:
      labels:
        app: fluentd
    spec:
      containers:
        - name: fluentd
          image: fluent/fluentd:v0.14.10
          resources:
            limits:
              memory: 200Mi
            requests:
              cpu: 100m
              memory: 200Mi
          volumeMounts:
            - name: varlog
              mountPath: /var/log
            - name: varlibdockercontainers
              mountPath: /var/lib/docker/containers
              readOnly: true
          terminationGracePeriodSeconds: 30
      volumes:
        - name: varlog
          hostPath:
            path: /var/log
        - name: varlibdockercontainers
          hostPath:
            path: /var/lib/docker/containers
```

La sezione “resources” specifica le risorse disponibili in termini di memoria e CPU per i vari pod. Come unità di misura si utilizza il Mi (Mebi, si basa sulla potenza del 2 invece che del 10).

I core in Kubernetes vengono spezzettati in Mi.

La sezione “terminationGracePeriodSeconds: 30” è utile per liberare le risorse gradualmente e non lasciare niente appeso. Ad esempio, in questo caso si procede soft fino a 30 secondi, dopodiché si procede al kill brutale.

La sezione “limits” all'interno della sezione resources è molto importante per appunto limitare i processi daemon ed evitare che essi usino tutte le risorse.

CronJob: oggetto Kubernetes utilizzato per **eseguire un job in maniera programmata** in base ad un intervallo di tempo specifico o ad un'ora specifica. Quando viene creato un CronJob, viene specificato il lavoro che deve essere eseguito nell'intervallo di tempo indicato. Una volta lanciato il job, vengono istanziati uno o più pod ed è possibile specificare il numero di repliche del pod da eseguire. Il CronJob utilizza la stessa sintassi del CronJob di Unix per definire l'orario in cui deve essere eseguito il lavoro.

Quindi, è possibile specificare le ore, i giorni della settimana, i giorni del mese, i mesi e altre informazioni temporali per pianificare l'esecuzione del lavoro.

È utile per eseguire attività periodiche all'interno di un cluster Kubernetes, come l'esecuzione di backup, l'elaborazione di dati in batch, l'invio di report periodici e così via.

```

# For details see man 4 crontabs

# Example of job definition:
# ----- minute (0 - 59)
# | ----- hour (0 - 23)
# | | ----- day of month (1 - 31)
# | | | ----- month (1 - 12) OR jan,feb,mar,apr ...
# | | | | ----- day of week (0 - 6) (Sunday=0 or 7) OR sun,mon,tue,wed,thu,fri,sat
# | | | |
# * * * * * user-name command to be executed

```

CronJob

```

[root@controller ~]# cat pod-cronjob.yml
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: pod-cronjob
spec:
  schedule: */2 * * * *
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: pod-cronjob
              image: busybox
              args:
                - /bin/sh
                - -c
                - date; echo hello from k8s cluster
  restartPolicy: OnFailure

```

Esempio per creare una risorsa CronJob per eseguire un processo batch ogni due minuti. La sezione evidenziata è il template per le risorse Job che verranno create da questo CronJob in cui l'attività specificata verrà eseguita ogni due minuti.

Deployment

L'oggetto Deployment è considerato ad un alto livello d'astrazione in quanto integra altri oggetti che vanno ad articolare lo stato del pod.

Quando si eseguono i pod, potrebbero essere necessarie funzionalità aggiuntive come la scalabilità, gli aggiornamenti, i rollback, etc.

Tali funzioni sono offerte dal Deployment.

È pensato per distribuire applicazioni e aggiornarle in modo dichiarativo.

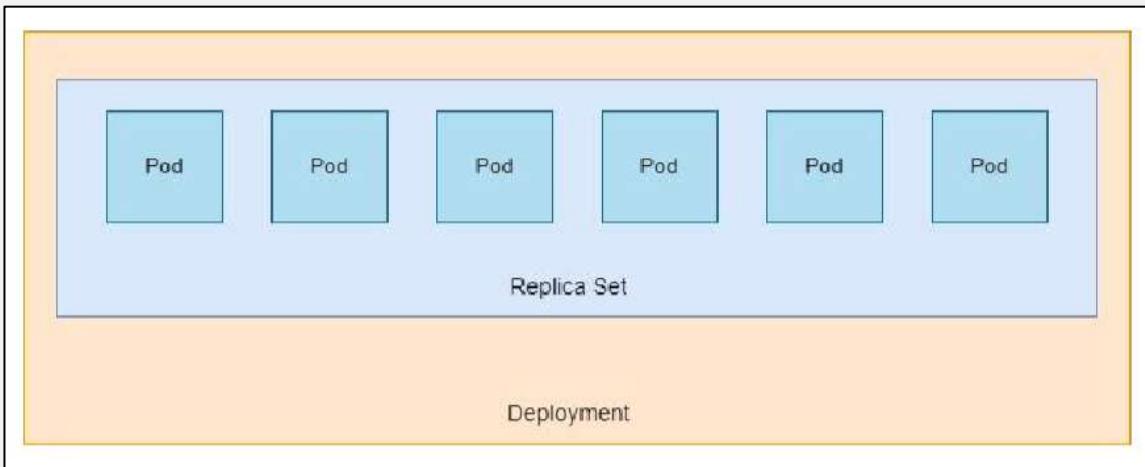
Invece di farlo tramite un ReplicationController o un ReplicaSet, che sono entrambi considerati di più basso livello, si utilizza il Deployment.

Quando si crea un oggetto Deployment, al di sotto viene creato un oggetto ReplicaSet che gestisce i pod. Pertanto, quando viene utilizzato un Deployment, i pod effettivi vengono creati e gestiti dai ReplicaSet del Deployment e non direttamente dal Deployment stesso.

Lo stato desiderato del ReplicaSet è descritto nel Deployment.

Il Deployment passa dallo stato effettivo allo stato desiderato ad una velocità controllata.

È possibile definire i Deployment per creare nuovi ReplicaSet o per rimuovere i Deployment esistenti e spostare tutte le relative risorse nei nuovi Deployment.



Il Deployment genera un oggetto Replica Set e poi fa qualcos'altro.

Nella sua sezione "spec" sono specificate e gestite le seguenti proprietà:

- **Replicas**: spiega quante copie di ciascun pod devono essere in esecuzione.
- **Strategy**: spiega come dovrebbero essere aggiornati i pod.
- **Selector**: utilizza matchLabels per identificare in che modo le etichette vengono abbinate ai pod.
- **Template**: contiene le specifiche dei pod e viene utilizzato dal Deployment per creare dei pod basandosi su di esso.

Example:

Creation of the Deployment:

kubectl apply -f <https://k8s.io/examples/controllers/nginx-deployment.yaml>

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 88

```

```
[root@controller ~]# cat rolling-ingress.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rolling-ingress
spec:
  replicas: 4
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
  selector:
    matchLabels:
      app: rolling-ingress
  template:
    metadata:
      labels:
        app: rolling-ingress
    spec:
      containers:
        - name: nginx
          image: nginx:1.9
```

Example:

Use of strategy: Deployment for **RollingUpdate**

The RollingUpdate options are used to guarantee a certain **minimal** and **maximal** number of Pods to be always available:

maxUnavailable: The maximum number of Pods that can be unavailable during updating. The value could be a percentage (the default is 25%) or an integer. If the value of maxSurge is 0, which means no tolerance of the number of Pods over the desired number, the value of maxUnavailable cannot be 0.

maxSurge: The maximum number of Pods that can be created over the desired number of ReplicaSet during updating. The value could be a percentage (the default is 25%) or an integer. If the value of maxUnavailable is 0, which means the number of serving Pods should always meet the desired number, the value of maxSurge cannot be 0

Here we want four replicas and we have set the update strategy to RollingUpdate. The MaxSurge value is 1, which is the maximum above the desired number of replicas, while maxUnavailable is 1. This means that throughout the process, we should have at least 3 and a maximum of 5 running pods.

La sezione **strategy** presenta delle opzioni aggiuntive:

- **maxSurge:** stabilisce il numero massimo di pod che **possono essere** creati rispetto al numero desiderato nel ReplicaSet durante l'aggiornamento. Il valore può essere una percentuale (il valore predefinito è 25%) o un numero intero. Nell'esempio sopra abbiamo che i pod devono essere 4 ma con maxSurge uguale a 1 **stiamo dicendo che durante un aggiornamento possiamo averne anche** 5.
- **maxUnavailable:** stabilisce il numero massimo di pod che **possono non essere** disponibili durante un aggiornamento. Il valore può essere una percentuale (il valore predefinito è 25%) o un numero intero.

Supponiamo di voler aggiornare un'applicazione dalla versione 1 alla versione 2.

Ciò significa dover aggiornare tutte le istanze dei pod dalla v1 alla v2.

Esistono due possibili modi per farlo:

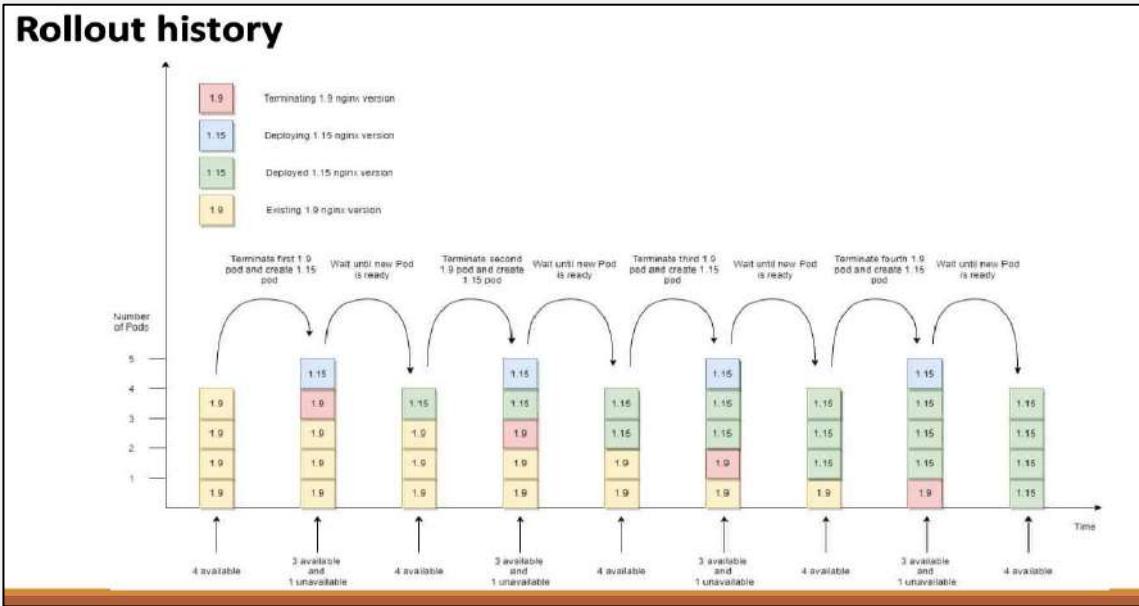
- 1) Eliminare prima tutti i pod esistenti e poi avviare quelli nuovi (**Recreate**) andando in contro a una temporanea indisponibilità.
- 2) Mediante un aggiornamento progressivo, aggiornando i pod uno alla volta per mantenere la disponibilità dell'applicazione (**Rolling Update**).

L'approccio 2 è chiaramente quello predefinito.

L'oggetto Deployment è molto utile perché consente di eseguire facilmente l'aggiornamento di un'applicazione containerizzata, sostituendo gradualmente le vecchie istanze con le nuove, senza interrompere il servizio.

Cosa che un semplice ReplicaSet non può fare.

La politica (strategia) da utilizzare è descritta nella sezione **strategy**.



Primo passo: viene terminato un pod con la versione v1 e se ne crea un altro avente la versione successiva v2.

Procedendo così iterativamente si aggiorna tutto il cluster mantenendo intatta quella che è l'operatività.

Quest'approccio implica il dover convivere con un intervallo di tempo in cui esistono sia pod nella versione v1 che pod nella versione v2.

Se la fortuna assiste, i pod più aggiornati gestiscono le varie richieste, altrimenti si ottengono risposte ancora sulla base della vecchia versione.

Il tutto è un qualcosa di secondi ma comunque rappresenta un piccolo prezzo da pagare per mantenere sempre attivo il servizio e non aggiornando tutto insieme. Seguendo l'esempio, avremo sempre al max 5 pod e al max 1 indisponibile.

Label and Selector

Ad un qualsiasi oggetto Kubernetes può essere associata una coppia chiave-valore.

Kubernetes fa riferimento a queste coppie chiave-valore come **labels**.

Le label non forniscono unicità.

In generale, molti oggetti potrebbero avere associate le stesse etichette.

Le label sono interrogabili, il che le rende particolarmente utili per organizzare le cose. Il meccanismo per questa query prende il nome di **label selector**.

Un label selector è una stringa che identifica le etichette che stiamo tentando di selezionare/utilizzare. Esistono attualmente due tipi di selettori:

selettori basati sull'uguaglianza e selettori basati sugli insiemi.

Tramite un label selector, il cliente può identificare un insieme di oggetti.

Il label selector rappresenta la principale modalità di raggruppamento in Kubernetes. Le label vengono **utilizzate per l'organizzazione e la selezione di**

sottoinsiemi di oggetti e possono essere aggiunte a quest'ultimi al momento della loro creazione e/o modificate in qualsiasi momento durante le varie operazioni che un cluster Kubernetes compie.

Example

```

apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80

```

version of the Kubernetes API used to create this object
type of object you want to create
data that helps uniquely identifying the object, including a name string, UID, etc
specific format of the object spec

The Deployment selects Pods with the label «app:nginx»
Pod label «app: nginx»

L'oggetto deployment seleziona i pod con l'etichetta app:nginx.

Se si utilizza un selettore basato sull'uguaglianza si effettua un test basato sulla domanda: "è? / non è?".

Ad esempio, *tier = frontend* restituirà tutti i pod che hanno un'etichetta con la chiave "tier" e il valore "frontend".

D'altra parte, se volessimo ottenere tutti i pod che non sono di tipo frontend, scriverebbero: *tier != frontend*.

È possibile combinare le due selezioni è / non è.

Ad esempio, *tier != frontend, gioco = super - shooter - 2* restituirebbe tutti i pod che fanno parte del gioco denominato super-shooter-2 ma che non sono di livello frontend.

Se si utilizza un selettore basato sugli insiemi si effettua un test basato sulla domanda: "sono in? / non sono in?".

Per esempio, utilizzando tre selezioni: *environment in (production, qa)*, *tier notin (frontend, backend)* e *Partition*. Il primo test restituirebbe i pod con l'etichetta environment che hanno come valore production o qa.

Il secondo test restituirebbe tutti i pod che non sono nei livelli frontend o backend. Infine, il terzo test restituirebbe tutti i pod che hanno l'etichetta Partition, indipendentemente dal valore che contiene.

Come limitare le risorse Kubernetes

Quando si allocano risorse in un container o in un pod si fa come al banco per prendere la mortadella, **ognuno chiede cosa e quanto vuole**.

Ci sono **tre diversi tipi di risorse** per le quali è possibile imporre richieste e limiti: **CPU, Memoria, Hugepages**. CPU e memoria sono indicate insieme come risorse di calcolo o semplicemente risorse. Le risorse di calcolo sono quantità misurabili che possono essere richieste, allocate e consumate.

In Kubernetes, si utilizzano il limite **soft** e il limite **hard** per le risorse da assegnare ad un container o ad un pod.

Il limite **soft** (o limite di soglia) rappresenta il valore massimo per una determinata risorsa. Il limite soft viene utilizzato principalmente come avviso per monitorare l'utilizzo delle risorse e intervenire in caso di necessità.

Il limite **hard** (o limite di esaurimento) rappresenta invece il valore massimo assoluto per una determinata risorsa, al di sopra del quale un container o un pod vengono terminati o la richiesta di risorse viene rifiutata.

Il limite hard viene utilizzato per evitare che una singola applicazione containerizzata utilizzi troppe risorse e interferisca con altre applicazioni.

Ad esempio, se si specifica un limite soft di 500 MB di memoria e un limite hard di 600 MB di memoria per un container, quando l'utilizzo di memoria del container supera i 500 MB, l'amministratore di sistema riceve un avviso.

Se l'utilizzo di memoria supera i 600 MB, il container viene terminato.

Qualora non siano specificate le richieste, il controller si basa sul limite hard.

Sia il limite soft che hard sono specificati in termini di CPU cores.

Il core è quell'oggetto che il sistema operativo può utilizzare per mandare in esecuzione un thread. L'unità minima assegnabile è 0.001 millicore ovvero 1 m.

Un container Docker una volta che ha utilizzato la sua quota di risorse dovrà attendere fino al successivo periodo di 100 ms prima di poter continuare a utilizzare la CPU. Il metodo utilizzato per condividere le risorse della CPU tra diversi processi in esecuzione in un cgroup (control groups) è chiamato

Completely Fair Scheduler o CFS. Funziona dividendo il tempo della CPU tra i diversi cgroup. Questo in genere significa assegnare un certo numero di slice a un cgroup. Se i processi in un cgroup sono inattivi e non utilizzano il tempo di CPU allocato, queste condivisioni diventeranno disponibili per essere utilizzate da processi di altri cgroup. Se vengono raggiunti i limiti di memoria, il runtime del container interromperà il container stesso (e potrebbe essere riavviato) con errore OOM (Out Of Memory). Se un container utilizza più memoria rispetto alla quantità richiesta, diventa un candidato per la terminazione forzata. Le quote delle risorse consentono di porre limiti al numero di risorse che un particolare namespace può utilizzare.

Kubernetes Networking Model

Si è visto come gli oggetti di Kubernetes siano delle strutture dati situate dentro la memoria principale del Master, ovvero l'etcd.

È possibile pensare a tali oggetti come a delle strutture dati chiave-valore.

Tutti gli oggetti Kubernetes hanno lo scopo di orchestrare container.

In Kubernetes, ci sono **quattro principali problemi di networking** da gestire:

- **Comunicazioni "highly-coupled" tra container**: risolvibili mediante l'utilizzo dei pod e delle comunicazioni tramite localhost.
- **Comunicazioni tra pod**.
- **Comunicazioni tra pod e servizi**: gestite dai servizi.
- **Comunicazioni tra i servizi e l'esterno**: gestite dai servizi.

Kubernetes utilizza tre ipotesi fondamentali per l'implementazione del networking, a prescindere dalla tipologia in questione:

1. I pod possono comunicare con tutti gli altri pod su qualsiasi altro nodo del cluster senza l'utilizzo del NAT, utilizzando il loro IP.
2. Gli agenti che si trovano su un nodo (come system daemons, kubelet, etc.) possono comunicare con tutti i pod presenti sul medesimo nodo.
3. I pod che si trovano nella rete host di un nodo possono comunicare con tutti i pod di tutti i nodi del cluster senza NAT.

NAT (Network Address Translation) è una tecnologia utilizzata nei router per consentire a più dispositivi di condividere una singola connessione Internet utilizzando un unico indirizzo IP pubblico.

In una rete locale, come ad esempio una casa o un'azienda, ogni dispositivo ha un indirizzo IP privato univoco, ma quando questi dispositivi si connettono a Internet, hanno tutti lo stesso indirizzo IP pubblico.

Ciò significa che i dati inviati da Internet a uno dei dispositivi devono prima passare attraverso il router e quindi essere indirizzati al dispositivo corretto all'interno della rete locale.

NAT consente di associare i dispositivi con gli indirizzi IP privati della rete locale all'indirizzo IP pubblico del router, consentendo al router di instradare i dati correttamente ai dispositivi appropriati.

Inoltre, NAT aumenta la sicurezza della rete locale nascondendo gli indirizzi IP privati dei dispositivi dalla rete pubblica.

Se un pod gira nella rete host di un nodo allora può utilizzare il Network namespace e il Resource namespace di quel nodo.

Ciò significa che **può avere accesso all'interfaccia di loopback** e monitorare il traffico degli altri pod sullo stesso nodo.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  hostNetwork: true
  containers:
    - name: nginx
      image: nginx
```

Se l'indirizzo IP del pod è uguale a quello del nodo allora questo indica che il pod gira nella rete host di quel nodo.

Networking interno ad un nodo

All'interno di ogni pod è presente il **container pause** che, tra le altre cose, ha il compito di **fornire un'interfaccia di rete a tutti i container presenti all'interno del pod**, consentendo una comunicazione fra essi e l'esterno.

Kubernetes istanzia il **container pause** prima di tutti gli altri.

Viene creata un'interfaccia virtuale di nome **veth0** (virtual ethernet).

Quest'interfaccia viene utilizzata da tutti i container creati all'interno del pod.

Quindi, **tutti i container all'interno di un pod hanno lo stesso indirizzo IP**.

Viene a mancare il problema di dover aprire le porte su ogni singolo container perché dall'esterno vengono visti come un unicum.

Si utilizza un'interfaccia a coppie.

L'**interfaccia virtuale veth0** si trova nel pod.

L'**interfaccia fisica eth0** si trova nel nodo host.

Le due interfacce sono collegate tra loro da un bridge di rete virtuale, nel nostro caso il **Docker0**, in modo che il traffico di rete possa fluire da un'interfaccia all'altra.

I container di un pod, avendo tutti lo stesso indirizzo IP, per comunicare utilizzano l'interfaccia di loopback.

Proprio come due programmi comunicano nello stesso pc.

Docker0 viene descritto come un bridge ma in realtà è

più complesso in quanto dispone di protocolli di strato 3 ed ha un indirizzo IP compatibile con quello dei container. Non è altro che il next op del container.

Docker0 è dotato di una tabella di forwarding dove ci sono le istruzioni su come inoltrare i pacchetti verso la destinazione. I Network namespace dei pod, utilizzati dai container, sono diversi dal Network namespace dell'**eth0**.

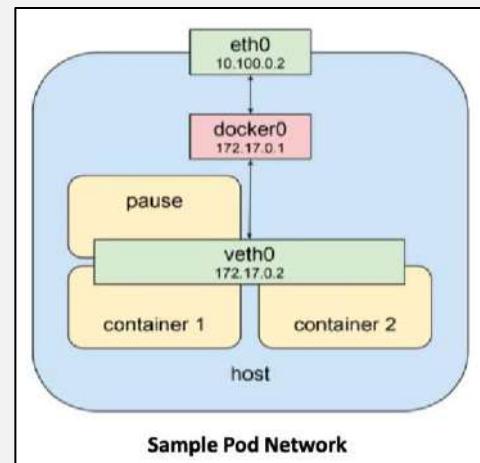
Fa uso dell'**iptables** (parlando in termini Linux) in modo da poter inoltrare i pacchetti nella subnet di **eth0**.

Docker0 è connesso a tutti i pod del nodo così che quando un pod deve comunicare con un altro pod invia la richiesta dal suo **vethX** a Docker0, il quale inoltrerà la richiesta a tutti i pod a lui connessi specificando l'indirizzo di destinazione. Il pod che ha tale indirizzo (ogni pod conosce il suo IP) risponde e il Docker0 lo inoltra a lui.

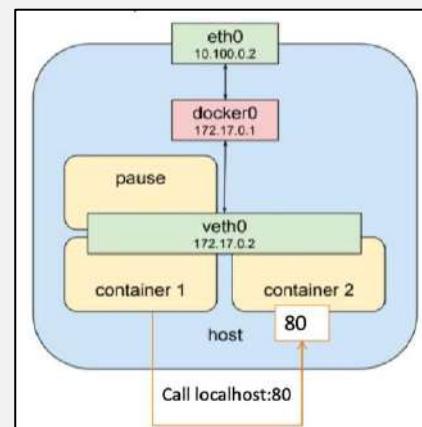
Il bridge Docker0 connette tutti i pod dello stesso nodo tra di loro.

Non è necessario creare esplicitamente collegamenti tra i pod.

Non è necessario mappare le porte dei container alle porte dell'host.



Sample Pod Network



Inoltre, i container all'interno dello stesso pod hanno anche accesso ai volumi condivisi, poiché fanno parte del pod.

Networking tra due nodi (pod to pod on different nodes)

Siamo dentro lo stesso cluster, abbiamo un unico master, abbiamo diversi pod su diversi nodi. **Come comunicano due container appartenenti a due pod diversi su nodi diversi?** Ci vuole un coordinamento di strato 3, ci vuole una tabella di forwarding/routing. Può essere implementata con un oggetto che distribuisce i vari indirizzi IP in modo da poter configurare diversi indirizzi IP.

Ogni singolo pod di un nodo ha il proprio Network namespace.

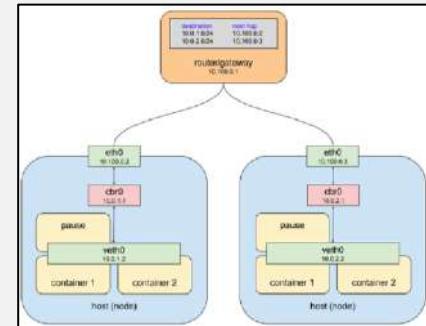
Quindi, ogni singolo pod ha il proprio indirizzo IP e pensa di avere un dispositivo ethernet del tutto normale chiamato **eth0** per effettuare richieste di rete.

Entrambi i pod possono legittimamente avere la stessa sottorete.

Il problema fondamentale è che un nodo in generale non ha idea di quale spazio di indirizzi privato sia stato assegnato a un bridge su un altro nodo, ma noi dobbiamo avere la certezza che se invieremo pacchetti essi arriveranno nel posto giusto.

Chiaramente è necessaria una struttura di gestione, come una rete sovrapposta (**overlay network**).

Kubernetes assegna uno spazio degli indirizzi complessivo per i bridge su ciascun nodo, assegnando gli indirizzi dei bridge all'interno di tale spazio in base al nodo in cui sono.



In seguito, Kubernetes aggiunge delle regole di instradamento al gateway.

La modifica del nome dei bridge da Docker0 a cbr0 (custom bridge) sta ad indicare che **Kubernetes può utilizzare diversi tipi di bridge**.

Dato che ogni pod ottiene un indirizzo IP reale (non privato della macchina), essi possono comunicare tra di loro senza proxy o traduzioni.

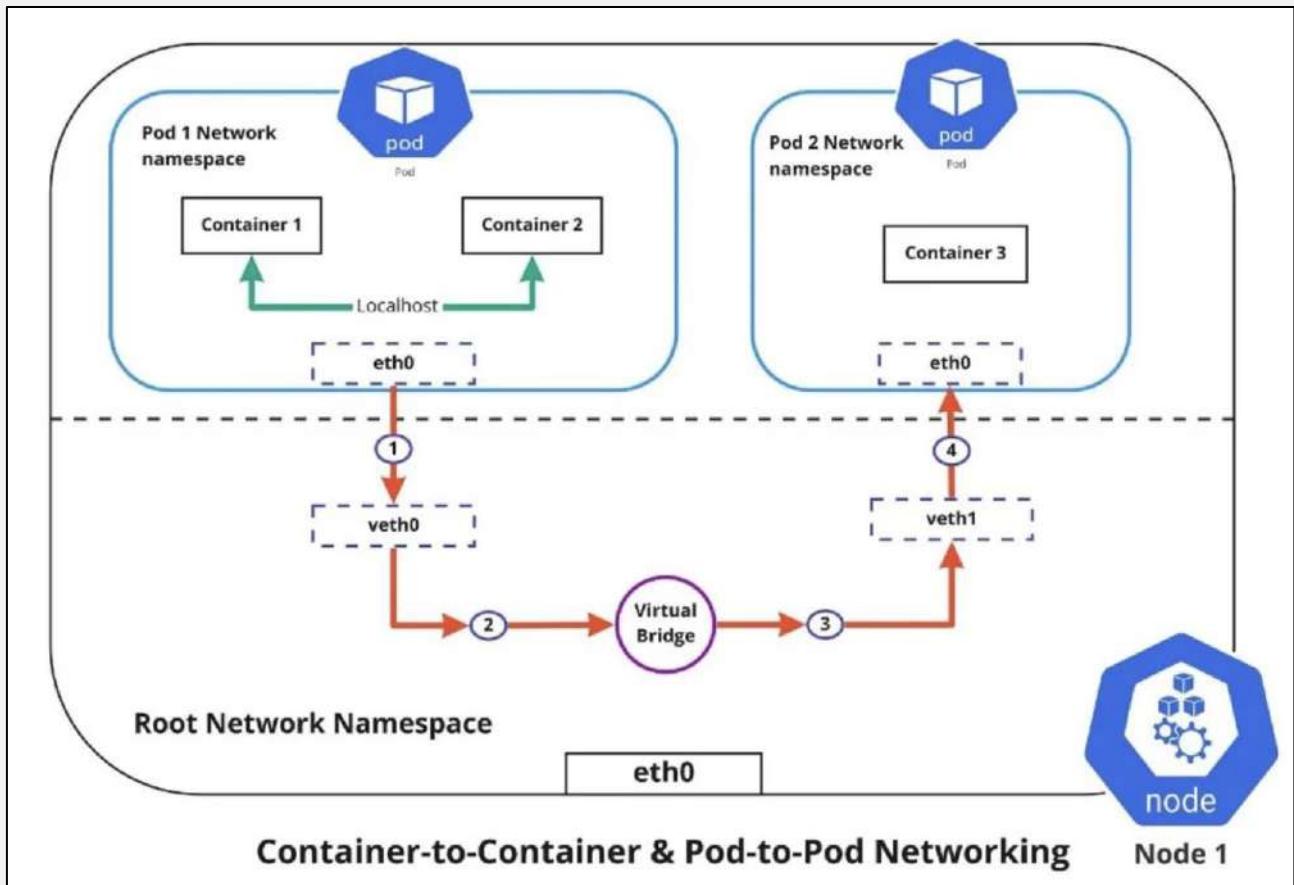
Lo scenario di rete standard in Kubernetes prevede che ogni pod abbia una singola interfaccia di rete (**eth0**). Il pod si comporta come un singolo host che può comunicare con ogni altro host sullo stesso nodo e con tutti gli host degli altri nodi della rete, a seconda della **Container Network Interface (CNI)** in uso.

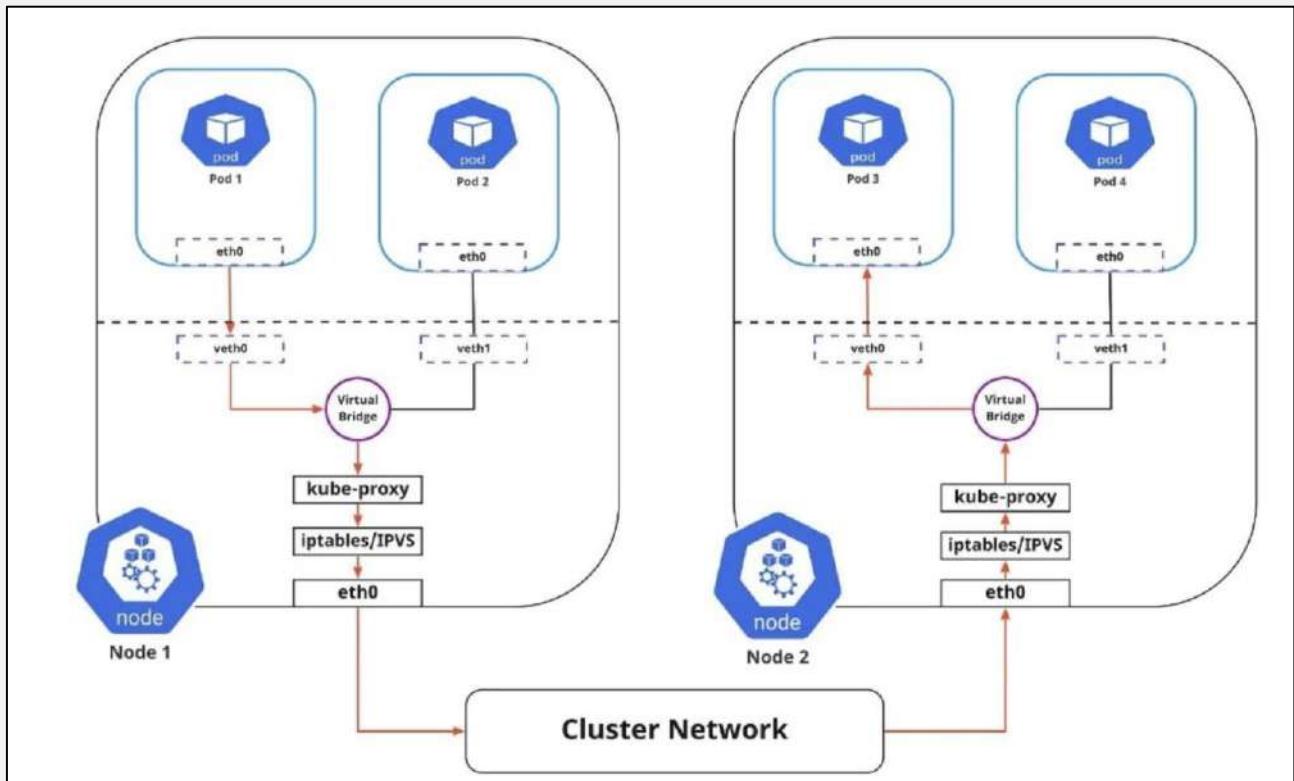
Abbiamo configurato i pod in modo tale che ciascuno di essi disponga del proprio Network namespace, in modo da credere di avere il proprio dispositivo ethernet, indirizzo IP e che siano connessi al Root namespace del nodo.

Ora, vogliamo che i pod parlino tra loro attraverso il Root Network namespace e per questo utilizziamo un bridge di rete.

Quando un pod effettua una richiesta all'indirizzo IP di un altro nodo, effettua tale richiesta tramite la propria interfaccia eth0.

La richiesta segue il tunneling alla rispettiva interfaccia vethX del nodo.
Ogni pod Kubernetes include un container pause che avvia il pod e stabilisce, tra le altre cose, il namespace prima che vengano creati i singoli container.





L'immagine del container pause è sempre presente, quindi l'allocazione delle risorse del pod avviene istantaneamente quando vengono creati i container.

I dispositivi veth sono dispositivi ethernet virtuali, possono fungere da tunnel tra Network namespace differenti, realizzando una sorta di ponte virtuale.

Possono anche essere utilizzati come dispositivi di rete autonomi.

Dal punto di vista del pod, veth esiste nel proprio Network namespace e deve comunicare con altri Network namespace dello stesso nodo.

Fortunatamente, i namespace possono essere collegati utilizzando un dispositivo Linux Virtual Ethernet o una coppia veth costituita da due interfacce virtuali che possono essere distribuite su più namespace. Per connettere i namespace dei pod, possiamo assegnare un lato della coppia veth al Root namespace e l'altro lato al Network namespace del pod. Ogni coppia veth funziona come un cavo patch, collegando i due lati e consentendo un flusso di traffico tra loro. Questa configurazione può essere replicata per tutti i pod della macchina.

Container Network Interface (CNI)

Il CNI fornisce un'API comune per connettere i container alla rete esterna.

Viene utilizzato per creare e configurare le interfacce di rete per i container e i pod all'interno di un cluster di nodi.

È un framework per la configurazione dinamica delle risorse di rete. La rete del cluster fornisce la

comunicazione tra i diversi pod.

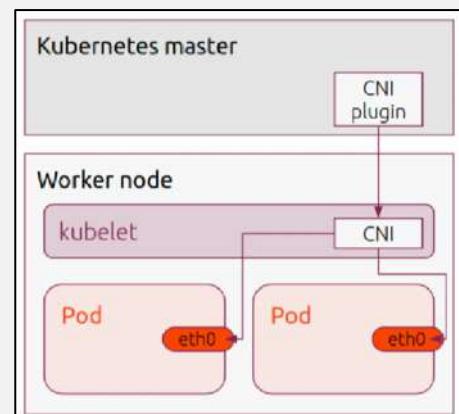
Il modello di rete Kubernetes richiede che gli IP dei pod siano raggiungibili attraverso la rete, ma non specifica come farlo.

Sono stati stabiliti alcuni schemi per renderlo più semplice.

In generale, ad ogni nodo del cluster viene assegnato un blocco CIDR che alloca gli indirizzi IP disponibili per i pod in esecuzione su un nodo col minimo spreco. Una volta che il traffico destinato a quel blocco CIDR raggiunge il nodo, è responsabilità del nodo inoltrare il traffico al pod corretto.

Il plugin CNI dev'essere implementato come eseguibile e deve poter essere richiamato dal master.

Un plugin CNI è responsabile dell'inserimento di un'interfaccia di rete nel Network namespace del container (ad esempio un'estremità di una coppia veth) e dell'apportare le modifiche necessarie sull'host (ad esempio collegando l'altra estremità del veth a un bridge).



Il provider di rete di base di Kubernetes è **kubenet**, un semplice plugin di rete che funziona con vari provider di servizi cloud, ma presenta molte limitazioni. In sintesi: un plugin CNI collega coppie di interfacce di rete per connettere i pod ai custom bridge fornendo una connessione virtuale tra essi.

Kubenet funziona bene con i pod locali ma quando ha a che fare con pod su cluster distribuiti non ce la fa. Alcuni plugin di terze parti più famosi sono Flannel, Weave, Calico e AWS VPC. Si mettono in opera attraverso i file yaml.

I Servizi (pod-to-service)

In questa sezione viene analizzato il **networking tra i pod e il mondo esterno**. Il networking è possibile se e solo se tutti i pod che sono messi in opera sono visti dall'esterno come un unicum.

Ci sono quindi due tipologie di networking: una che mostra un unicum all'interno del cluster (quanto visto finora) e una che mostra un unicum all'esterno.

In entrambi i casi si necessita di una qualche entità che metta ordine.

I servizi di Kubernetes fanno questo lavoro "sporco". Rete Pod to Service.

Il networking tra i pod di un cluster è molto buono ma non può garantire un sistema affidabile e duraturo. A differenza del mondo "Non Kubernetes", dove un amministratore di sistema potrebbe configurare le applicazioni specificando esattamente gli indirizzi IP dei server che contengono un servizio nel file di configurazione del client, nel mondo Kubernetes non è possibile specificare un indirizzo IP esatto perché i pod sono delle entità effimere che potrebbero esserci o non esserci in qualsiasi momento. Supponiamo, ad esempio, che un pod venga rimosso da un nodo per fare spazio ad altri pod o perché qualcuno ha ridotto il numero di pod o perché un nodo del cluster ha avuto esito negativo.

Se si utilizza l'indirizzo IP di un pod come endpoint allora non c'è alcuna garanzia che l'indirizzo non cambi la prossima volta che il pod verrà ricreato; cosa che potrebbe accadere per una serie di motivi. Kubernetes assegna un indirizzo IP a un pod dopo che il pod è stato schedulato su un nodo e prima che venga avviato. I client, quindi, non possono conoscere l'indirizzo IP del server pod in anticipo. Inoltre, ciò non sarebbe conveniente in termini di scalabilità orizzontale: il ridimensionamento orizzontale significa che più pod possono fornire lo stesso servizio. Ciascuno di questi pod ha il proprio indirizzo IP.

Ai client non dovrebbe interessare di quanti pod supportino il servizio e quali siano i loro indirizzi IP. Non dovrebbero tenere un elenco di tutti i singoli IP dei pod. Invece, tutti quei pod dovrebbero essere accessibili tramite un singolo indirizzo IP. Questo problema viene risolto attraverso uno dei

servizi di Kubernetes, il **LoadBalancer**. Fondamentalmente, il traffico viene instradato al load balancer che funge da proxy per poi essere instradato ai pod corretti.

Il client si connette solo con il **LoadBalancer**, sarà lui a mantenersi in memoria una lista dei pod in salute e funzionanti a cui instradare le richieste fatte dal client. Ciò implica che il proxy LoadBalancer sia **durevole** e **fault tolerance**, che abbia una **lista di server sempre aggiornata** a cui poter instradare le richieste dei client e che abbia un qualche **sistema di notifica di aggiornamento** dello stato di salute live dei vari pod-server (nel caso un pod cada, non può essere più preso in considerazione per instradare richieste). Chiaramente deve avere un'interfaccia di rete verso l'esterno e verso i server virtuali.

Un servizio può essere visto come un'astrazione che definisce un **set logico di pod** e una **policy** con cui accedervi. Hanno il compito di scoprire a chi inviare il traffico, utilizzano i **Selector** per individuare i pod corretti.

I servizi assicurano che il traffico venga sempre instradato al pod corretto all'interno del cluster, indipendentemente dal nodo in cui esso gira.

Ogni servizio espone un indirizzo IP e può anche esporre un endpoint DNS per mettere in relazione il nome dei pod con l'indirizzo IP del nodo.

È un DNS che sta dentro al cluster e a cui fare riferimento per contattare i singoli endpoint. IP e DNS del servizio non cambieranno mai.

I client interni o esterni che devono comunicare con un set di pod utilizzeranno l'indirizzo IP del servizio o il suo endpoint DNS (più generalmente noto).

I servizi Kubernetes sono un'astrazione che consentono di esporre un'applicazione in esecuzione su un set di pod come un servizio di rete.

Ciò consente l'esecuzione di un set di pod utilizzando un singolo nome DNS e soprattutto consente il bilanciamento del carico delle richieste su tali pod.

Il ciclo di vita dei pod e dei servizi **NON** dipende l'uno dall'altro.

Se un pod muore, un servizio può rimanere in vita e può sostituire il pod morto con un pod nuovo sostituendo il vecchio indirizzo IP con l'IP del pod nuovo.

Per creare pod-server si può utilizzare l'oggetto Deployment.

```

kind: Deployment
apiVersion: apps/v1
metadata:
  name: service-test
spec:
  replicas: 2
  selector:
    matchLabels:
      app: service_test_pod
  template:
    metadata:
      labels:
        app: service_test_pod
    spec:
      containers:
        - name: simple-http
          image: python:2.7
          command: ["/bin/bash"]
          args: ["-c", "echo < p>Hello from $(hostname)</p>" > index.html; python -m SimpleHTTPServer 8080"]
      ports:
        - name: http
          containerPort: 8080

```

kubectl apply, i pod sono in esecuzione nel cluster.

L'insieme di pod che riceveranno il traffico è determinato dal **Selector**, che corrisponde alle etichette assegnate ai pod quando vengono creati.

Una volta creato il servizio, possiamo vedere come ad esso sia stato assegnato un indirizzo IP e un numero di porta 80 dove accetterà le richieste.

Quando il kind di un servizio non viene specificato, gli viene associato di default il tipo cluster IP. Ciò rende il servizio raggiungibile solo all'interno del cluster.

In particolare, può essere raggiunto solo da tutti i pod all'interno del cluster.

Non è possibile effettuare richieste al servizio (pod) dall'esterno del cluster. In quest'esempio il servizio va ad utilizzare tutti i pod che hanno come label la chiave "app" con valore "servirce_test_pod".

In Kubernetes, quando si crea un servizio, occorre impostare le proprietà "port" e "targetPort". Il tag "port" si riferisce alla porta del container esposta da un pod mentre il tag "targetPort" si riferisce alla porta sulla macchina host a cui è diretto il traffico.

Kubernetes fornisce anche un servizio DNS interno in modo da far comunicare i vari container e pod tra loro mediante un nome e non con l'indirizzo IP.

Un esempio di un client di tipo pod che utilizza il DNS Kubernetes.

```

apiVersion: v1
kind: Pod
metadata:
  name: service-test-client
spec:
  restartPolicy: Never
  containers:
    - name: test-client2
      image: alpine
      command: ["/bin/sh"]
      args: ["-c", "echo 'GET / HTTP/1.1\r\n\r\n' | nc service-test 80"]

```

Esempio di Deployment che crea due pod-server HTTP che rispondono sulla porta 8080 con l'hostname in cui sono in esecuzione.

Dopo aver creato questa distribuzione, utilizzando

```

kind: Service
apiVersion: v1
metadata:
  name: service-test
spec:
  selector:
    app: service_test_pod
  ports:
    - port: 80
      targetPort: http

```

Questo è un pod dentro lo stesso cluster che fa utilizzo dei servizi sopra citati.

Questo pod va a chiedere al DNS Kubernetes l'indirizzo IP su cui si interfacciano i pod che hanno il servizio denominato "service-test" sulla porta 80 per connettersi e inviare una richiesta di tipo GET a tale servizio. Una volta acquisito, li contatta.

L'indirizzo IP utilizzato per accedere al servizio appartiene ad un'altra subnet ancora, diversa dalle altre. L'indirizzo IP sta dentro al database e basta.

Lo spazio degli indirizzi riservato ai servizi prende il nome di **service network**.

Il service network non esiste, almeno non come interfacce connesse.

Tuttavia, funziona. Com'è possibile?

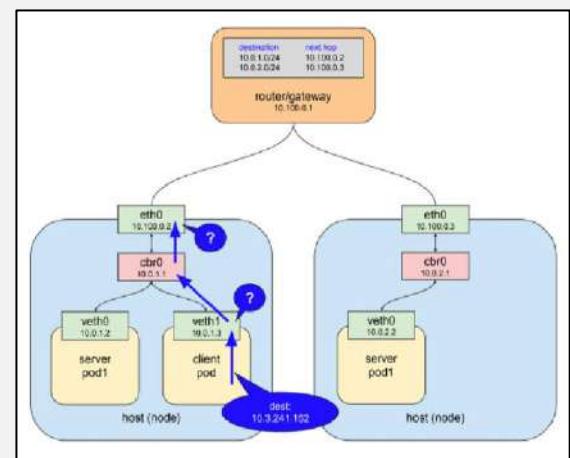
Come ogni cosa all'interno di Kubernetes, un servizio è solo una risorsa.

Un record in un database centrale che descrive come configurare del software per fare qualcosa.

Si supponga che un pod-client effettui una richiesta HTTP ad un servizio utilizzando il nome DNS service-test. Il sistema DNS del cluster traduce il nome nell'indirizzo IP associato al servizio nel cluster, supponiamo **10.3.241.152**.

Il pod-client va quindi a inviare una richiesta HTTP che comporta l'invio di alcuni pacchetti con **quell'IP** nel campo di destinazione.

Le reti IP sono generalmente configurate con rotte tali che quando un'interfaccia non può consegnare un pacchetto alla sua destinazione perché non esiste localmente alcun dispositivo con quell'indirizzo specifico, inoltre il pacchetto al suo "superiore", il gateway.



In quest'esempio ci sono 3 pod, 2 in un nodo e 1 in un altro.

Hanno indirizzi IP diversi. Ci sono 2 **bridge** e 2 interfacce **eth0**.

Supponiamo che uno dei due pod nello stesso nodo faccia richiesta (diventa il pod-client) per un servizio ad un altro pod (diventa il pod-server).

Grazie ad una richiesta CURL è possibile accedere a tale servizio o direttamente dal pod-server che si trova nel suo stesso nodo o dall'altro.

La richiesta verso 10.3.241.152 viene inoltrata al next op, ovvero al bridge interno perché la subnet dell'IP di destinazione è diversa dalla sua.

Tale bridge però non sa dove sia la subnet richiesta, non sa quindi che fare. Quindi quando un dispositivo non sa che fare usa la rotta di default, verso **eth0**. Inoltre la richiesta a quest'ultimo. A questo punto **eth0** legge la destinazione, ma nemmeno lui la conosce. Inoltre quindi la richiesta ancora più verso l'alto, ovvero al **router/gateway**. Ma quando la richiesta arriva a quest'ultimo c'è una differenza sostanziale: **l'indirizzo di destinazione è stato modificato**.

Ciò che accade realmente è che il pacchetto viene catturato durante il transito, **modificato** e reindirizzato a uno dei pod-server.

Il motivo di questo comportamento risiede nell'intervento del **kube-proxy**.

Il kube-proxy, così come tutti i proxy, ha bisogno di un'interfaccia sia per ascoltare le connessioni client che per connettersi ai server back-end. Il kube-proxy risiede in ogni nodo worker. Kubernetes utilizza un kernel Linux chiamato **netfilter** e un'interfaccia dello spazio utente chiamata **IPTABLES**.

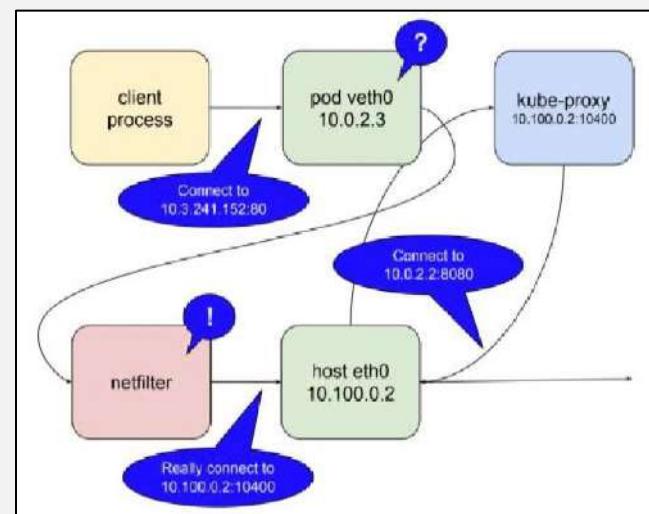
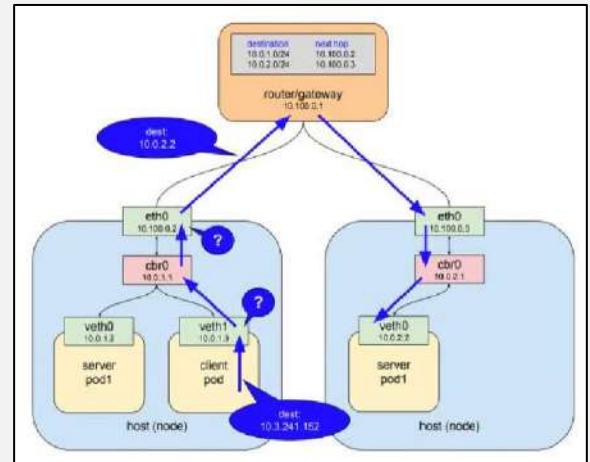
In questo modo **kube-proxy** apre una porta (la 10400 nell'esempio in figura) sull'interfaccia dell'host locale per ascoltare le richieste al "service-test" e va ad inserire delle regole di netfilter per reindirizzare i pacchetti destinati all'IP del servizio verso la propria porta ed infine re-inoltra tali richieste con IP e porta di destinazione corretti all'host locale. È così che una richiesta a 10.3.241.152:80 diventa magicamente una richiesta a 10.0.2.2:8080.

Riassumendo, quindi, si ha che nella catena di input è presente una configurazione che cambia l'indirizzo di destinazione. Questa configurazione è il netfilter che sta in ascolto su eth0. Quando gli arriva un pacchetto cambia l'indirizzo IP e ci mette come destinazione l'indirizzo del kube-proxy.

Il kube-proxy si prende il pacchetto cambiandogli il socket (IP + porta) e lo rilancia nell'interfaccia dell'host.

Gli mette come destinazione l'indirizzo giusto con la porta 8080 e così il pacchetto può arrivare al pod-server nell'altro nodo in maniera corretta.

Purtroppo, la comunicazione con il kube-proxy può diventare un collo di bottiglia.



Nelle ultime versioni il kube-proxy ha solo il compito di inoltrare a netfilter, poi sarà lui a cambiare l'indirizzo. In questo modo resta tutto nel kernel space rendendo il tutto più veloce (questo perché kube-proxy sta nello spazio utente).

in quanto restiamo dentro al cluster.

In queste nuove versioni, il lavoro del kube-proxy è più o meno quello di mantenere sincronizzate le regole di netfilter.

Il kube-proxy ascolta il kube-apiserver del Master per le modifiche nel cluster, le quali includono modifiche ai servizi e agli endpoint.

Quando il kube-proxy riceve gli aggiornamenti, utilizza IPTABLES per mantenere sincronizzate le regole di netfilter. Quando viene creato un nuovo servizio e i relativi endpoint vengono popolati, il kube-proxy riceve la notifica e crea le regole necessarie. Allo stesso modo, rimuove le regole quando i servizi vengono eliminati. I controlli di integrità sugli endpoint vengono eseguiti da kubelet, un altro componente che viene eseguito su ogni nodo worker, e quando vengono rilevati endpoint non integri, kubelet invia una notifica al kube-proxy tramite l'API Server. In questo modo le regole di netfilter vengono modificate per rimuovere tali endpoint fino a quando non diventano nuovamente integri.

Networking dall'esterno verso i servizi (External-to-Service)

Il kind di default ClusterIP funziona solo per le richieste che hanno origine all'interno del cluster, ovvero per comunicazioni pod-to-service.

Qualsiasi sia il meccanismo utilizzato, per consentire ai client esterni di chiamare i pod occorre utilizzare la stessa infrastruttura di routing.

I client esterni devono chiamare l'IP e la porta del cluster, perché questo è il "front-end" per tutte le macchine, il che rende possibile non preoccuparsi di dove sia in esecuzione un pod in un dato momento.

Il ClusterIP di un servizio è raggiungibile solo dall'interfaccia eth di un nodo.

Niente al di fuori del cluster sa cosa fare con gli indirizzi di quell'intervallo.

Come possiamo inoltrare il traffico da un endpoint pubblicamente visibile ad un IP raggiungibile solo quando il pacchetto è già in un nodo?

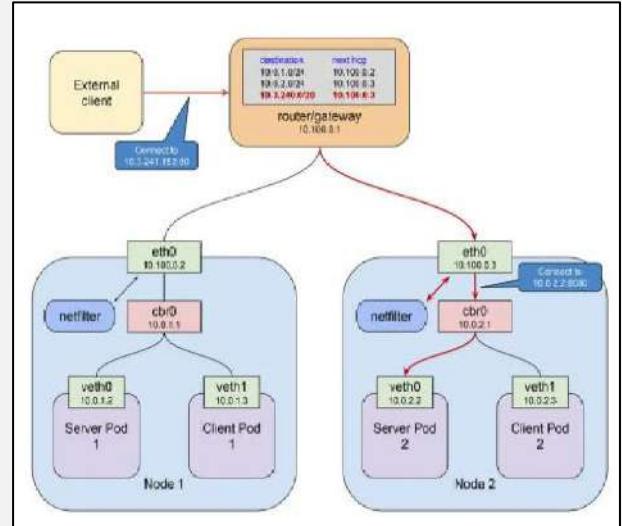
Potremmo semplicemente fornire ai client il ClusterIP, magari assegnandogli un nome di dominio descrittivo e quindi aggiungere un percorso per portare quei pacchetti a uno dei nodi.

I client chiamano l'IP del cluster, i pacchetti seguono un percorso fino a un nodo, per poi essere inoltrati ad un pod.

Questa soluzione soffre di alcuni seri problemi.

Il primo è semplicemente che i nodi sono effimeri. Non sono così effimeri come i pod, ma possono essere migrati a una nuova macchina virtuale, i cluster possono aumentare e diminuire, ecc. I router non distinguono i servizi sani da quelli non sani. Si aspettano

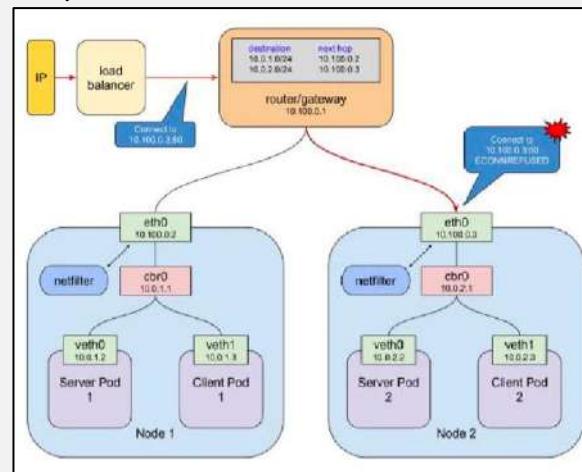
che il salto successivo nel percorso sia stabile e disponibile. Se il nodo diventa irraggiungibile, nella maggior parte dei casi il percorso si interromperà e rimarrà interrotto per un tempo significativo. Anche se il percorso fosse duraturo, che tutto il traffico esterno passi attraverso un singolo nodo non è ottimale.



Per poter utilizzare il LoadBalancer distribuendo il traffico dei client ai nodi di un cluster occorre un IP pubblico a cui i client possano connettersi e occorrono degli indirizzi sui nodi stessi a cui il load balancer possa inoltrare poi le richieste. Per i motivi discussi, non è possibile creare facilmente un percorso statico stabile tra il gateway e i nodi del cluster utilizzando la service network. Gli unici indirizzi rimasti disponibili sono nella rete a cui sono connesse le interfacce ethernet dei nodi (nell'esempio, 10.100.0.0/24).

Il router del gateway sa già come inviare i pacchetti a queste interfacce e le richieste inviate dal LoadBalancer al router arrivano nel posto giusto.

Tuttavia, se un client desidera connettersi ad un servizio in ascolto sulla porta 80 non può semplicemente inviare pacchetti con quella porta sulle interfacce dei nodi poiché non vi è alcun processo in ascolto su 10.100.0.3:80 (o se c'è è quello sbagliato) e le regole di netfilter, che speravamo intercettassero la richiesta e la indirizzassero a un pod, non corrispondono a quell'indirizzo di destinazione.



Corrispondono solo al ClusterIP sulla rete del servizio in 10.3.241.152:80.

Quindi, quei pacchetti non possono essere consegnati quando arrivano su quell'interfaccia e il kernel risponde con ECONNREFUSED.

La soluzione è creare qualcosa chiamato NodePort.

Un servizio di tipo NodePort è un servizio ClusterIP ma con una capacità aggiuntiva: è raggiungibile sia all'indirizzo IP del nodo host che all'indirizzo ClusterIP assegnato ad una service network.

Quando Kubernetes crea un servizio NodePort, il kube-proxy alloca una porta in un intervallo compreso tra [30000 e 32767] e la apre sull'interfaccia **eth0** di tutti i nodi del cluster (da qui il nome NodePort).

Le connessioni a questa porta vengono inoltrate al ClusterIP del servizio.

Il risultato è l'unione della rete dei nodi con la rete dei pod, garantendo la stabilità del servizio.

Se ci sono tre nodi nel cluster, Kubernetes aprirà una porta su tutti e tre i nodi del cluster, consentendo agli utenti di accedere al servizio utilizzando l'indirizzo IP di uno qualsiasi dei tre nodi più la porta utilizzata.

In sostanza, il NodePort è un modo per fornire un accesso dall'esterno del cluster ad un servizio all'interno del cluster Kubernetes.

Una volta fatto ciò, si dispone di una pipeline completa per il bilanciamento del carico delle richieste esterne dei client verso tutti i nodi interni al cluster.

Il servizio LoadBalancer in Kubernetes estende le funzionalità del servizio NodePort fornendo un **indirizzo IP esterno** che può essere utilizzato per accedere al servizio da parte di utenti o applicazioni esterne al cluster. Volendo, è possibile registrare su un DNS

```
kind: Service
apiVersion: v1
metadata:
  name: service-test
spec:
  type: NodePort
  selector:
    app: service_test_pod
  ports:
    - port: 80
      targetPort: http
```

esterno tale IP in modo da utilizzare un nome a scelta può essere raggiunto da chiunque.

Ogni volta che si vuole esporre un servizio verso il mondo esterno, occorre creare un nuovo LoadBalancer e ottenere un indirizzo IP.

I Servizi senza selettori: Servizi Esterni

Esistono scenari in cui è possibile che un servizio di Kubernetes abbia i nodi di calcolo dentro al cluster ma il database di riferimento all'esterno del cluster.

È possibile che una parte dei servizi sia dentro un cluster mentre un'altra parte dei servizi sia fuori, ecc.

In questi casi, non esiste più un Selector perché non ci sono più dei pod interni a cui fare riferimento.

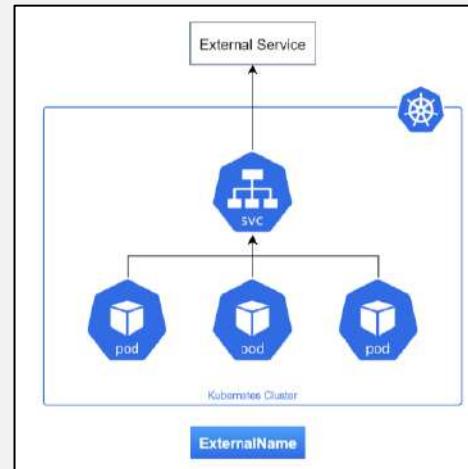
Si definiscono quindi dei servizi senza specificare un selettore per la corrispondenza con i pod.

Esistono casi in cui occorre esporre servizi esterni

tramite la funzionalità dei servizi Kubernetes. Invece di far in modo che il servizio reindirizzi le connessioni ai pod nel cluster, si cerca di reindirizzare verso IP e porte esterni. Si basa sui nomi DNS e sul reindirizzamento.

Il servizio **ExternalName** effettua un mapping di un particolare servizio a un nome DNS. Il servizio **ExternalName** funge da proxy, consentendo ad un pod di reindirizzare le richieste a un servizio che risiede all'esterno del cluster.

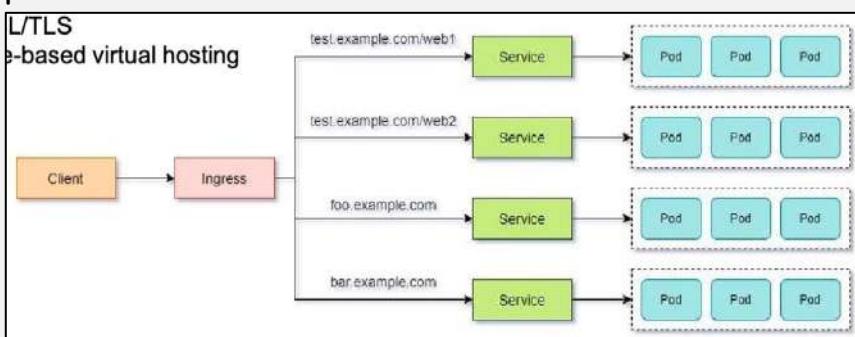
Caso d'uso: migrazione di tutte le applicazioni verso Kubernetes. Lavoro che sarebbe da fare in modo graduale. Quindi viene mantenuto un database esterno da cui i nuovi container gestiti da Kubernetes possano recuperare i dati. Tuttavia, il database risiede fuori dal cluster quindi i pod non hanno idea di cosa sia. Questo problema si risolve con il servizio ExternalName.



```
apiVersion: v1
kind: Service
metadata:
  name: my-database-svc
  namespace: prod
spec:
  type: ExternalName
  externalName: my.database.example.com
```

Ingress Service: può essere visto come un potenziamento dei servizi visti precedentemente. È come se si stesse costruendo un guscio attorno ai servizi di prima. Grazie ad Ingress vengono criptate le informazioni secondo uno strato aggiuntivo attraverso una password. Ingress Service utilizza i NodePort per inoltrare il traffico al servizio corrispondente, sfruttando il LoadBalancer interno fornito dalle regole iptables installate su ciascun nodo dal kube-proxy. Il parametro `secretName` è di fatto la password. Per il momento, supponiamo che chiunque acceda al file veda tale password.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    kubernetes.io/ingress.class: "gce"
spec:
  tls:
    - secretName: my-ssl-secret
  rules:
    - host: testhost.com
      http:
        paths:
          - path: /*
            backend:
              serviceName: service-test
              servicePort: 80
```



Occorre un file yaml per ogni pod, un file yaml per ogni servizio ed un file yaml per l'Ingress.
--> Servizio pronto e cifrato.
Riassumendo, Ingress

Service è un oggetto che permette di esporre i vari servizi all'esterno del cluster. In pratica, agisce come un livello d'ingresso al cluster ricevendo le richieste dei client. Supporta molte funzionalità avanzate come la gestione del carico e il routing basato su DNS.

Un client vuole solo l'hostname/indirizzo IP col quale accedere al servizio e non vuole ricordare tutte le altre informazioni, come le porte.

Per risolvere si utilizza Ingress Service di Kubernetes.

Il ciclo di vita di un pacchetto che passa attraverso Ingress è molto simile a quella di un pacchetto che passa attraverso il LoadBalancer.

Le differenze principali sono che l'**Ingress è a conoscenza del percorso dell'URL** (potendo instradare il traffico ai servizi in base al loro percorso) e che la connessione iniziale tra l'Ingress e il nodo avviene attraverso la porta esposta sul nodo per ciascun servizio.

Per aumentare la sicurezza è possibile creare degli oggetti ulteriori specifici chiamati **Secret** il cui unico compito è quello di gestire i file sensibili.

In questo modo si evita di esporre le password in chiaro. Aumenta la robustezza ma non è ancora sufficiente perché si fa sempre riferimento a dei file presenti nel filesystem del master e si potrebbe ancora violare la sicurezza se si è esperti. Il problema principale nella sicurezza è il fatto che nel database principale del master, etcd, i dati sono in chiaro se non si fa nulla.

Quindi si può avere accesso a tutte le informazioni.

Attraverso degli script si possono definire dei parametri per cifrare queste informazioni mediante degli algoritmi. Tuttavia, anche gli algoritmi necessitano di una chiave, e chi ce l'ha? Non dipende da Kubernetes.

Protezione di un Ingress specificando un SecretName che contiene una chiave privata e un certificato TLS. ----->

In un Ingress, il riferimento ad un Secret sta ad indicare di proteggere il canale dal client al LoadBalancer utilizzando il certificato TLS.



Volumi Kubernetes

Esistono due tipologie di volumi in Kubernetes: **effimeri** e **persistenti**.

I volumi effimeri non sono considerati risorse di primo livello come i pod ma considerati come parte di un pod, condividendo con lui lo stesso ciclo di vita.

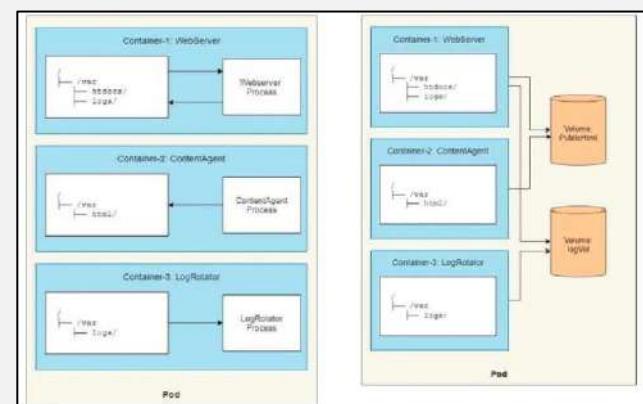
Ciò significa che un volume viene creato all'avvio di un pod e viene distrutto quando il pod viene eliminato. Per questo motivo, il contenuto di un volume persiste tra i vari riavvii dei container. Dopo il riavvio di un container, il nuovo container può visualizzare tutti i file scritti nel volume dal container precedente. Inoltre, se un pod ha più di un container allora il volume può essere utilizzato da tutti loro contemporaneamente.

Ad esempio, se il volume di un pod contiene i dati di log generati dai diversi container presenti allora ognuno di essi vedrà tutti i file di log complessivi a patto che il volume sia montato in ogni container.

Quello che non è permesso fare è collegare un pod di un nodo X a un volume di un nodo Y.

Inoltre, i volumi non sono oggetti Kubernetes autonomi e pertanto non possono essere creati o eliminati autonomamente.

Invece, i volumi persistenti sopravvivono anche a fronte della caduta di un pod.



Kubernetes Horizontal Pod Autoscaler (HPA)

Un oggetto HPA monitora il consumo di risorse dei pod gestiti da un controller (Deployment, ReplicaSet o StatefulSet) a determinati intervalli di tempo e controlla le repliche confrontando l'obiettivo desiderato specificato in determinate metriche con il loro utilizzo reale.

Esempio: supponiamo di avere inizialmente un oggetto controller Deployment con 2 pod che utilizzano in media 1.000 m di CPU. Vogliamo che la percentuale di CPU sia di 200 m per pod. L'HPA associato calcolerà quanti pod sono necessari per il target desiderato con la formula $2 * (1000 \text{ m} / 200 \text{ m}) = 10$.

Ciò significa che regolerà di conseguenza le repliche del Deployment a 10 pod.

Kubernetes si occuperà del restante per programmare gli 8 nuovi pod.

HPA è un componente fondamentale, istanziato con un suo file yaml.

Monitora più o meno continuamente lo stato del cluster.

HPA acquisisce lo stato del cluster dal

Metrics Server (processo demone che

necessita di essere installato

manualmente). In particolare, controlla i

pod nei vari replica set.

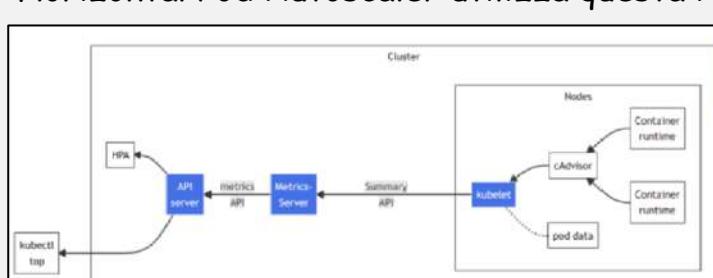
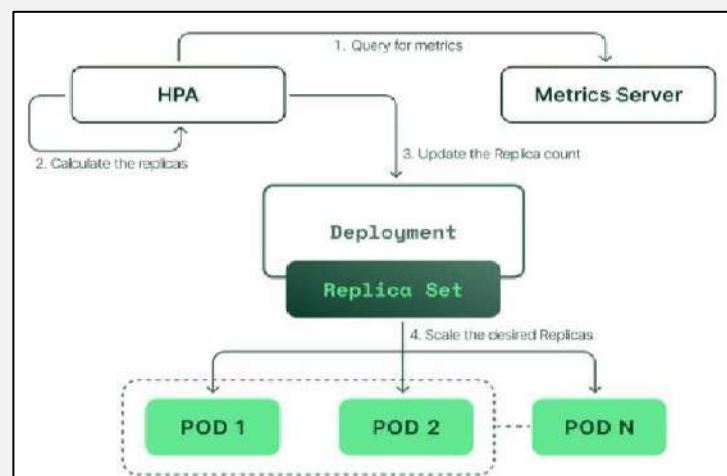
In base all'utilizzo delle risorse raccolte, HPA calcolerà il numero desiderato di repliche richieste.

L'autoscaling orizzontale non viene applicato agli oggetti che non possono essere scalati per ragioni tecniche come per esempio il DaemonSet.

HPA gira nel piano di controllo di Kubernetes.

È obbligatorio disporre di un Metrics Server installato e in esecuzione sul cluster Kubernetes. Esso fornirà le metriche tramite la sua API.

Horizontal Pod Autoscaler utilizza questa API per raccogliere le metriche.



Architettura della pipeline delle metriche per le risorse.

cAdvisor: demone per la raccolta, l'aggregazione e l'esposizione delle metriche dei container.

Inoltre le info a Kubelet.

HPA funziona in modo intermittente, non è un processo continuo.

L'intervallo predefinito è di 15 secondi.

Dal punto di vista più elementare, il controller Horizontal Pod Autoscaler opera sul rapporto tra il valore metrico desiderato e il valore metrico corrente:

```
desiredReplicas = ceil[currentReplicas * ( currentMetricValue / desiredMetricValue )]
```

Si moltiplica il numero attuale di repliche (stiamo parlando di pod) per il rapporto tra il valore di metrica corrente e quella desiderata.

Bisogna stare attenti a queste soglie, sennò **si rischia che HPA scali sempre all'infinito visto che il numero di pod potrebbe sempre cambiare in breve tempo.**

Ad esempio, se il valore della metrica corrente è 200m e il valore desiderato è 100m, il numero di repliche verrà raddoppiato, poiché $200.0 / 100.0 = 2.0$.

Se il valore corrente è invece 50m, HPA dimezza il numero di repliche, poiché $50.0 / 100.0 = 0.5$. Il piano di controllo ignora qualsiasi azione di ridimensionamento se il rapporto è sufficientemente vicino a 1,0 (entro una tolleranza configurabile a livello globale, 0.1 per impostazione predefinita).

Con questa metrica il controller HPA manterrà l'utilizzo medio dei pod nel target di ridimensionamento al 60% ----->

```
type: Resource  
resource:  
  name: cpu  
target:  
  type: Utilization  
  averageUtilization: 60
```

La sezione delle risorse come CPU e memoria deve essere specificata durante la realizzazione delle specifiche di un pod.

La **stabilizationWindow** si trova dentro la sezione **behavior**, sia per le impostazioni di **scaleDown** che di **scaleUp**.

Quando si fa **scaleDown** non lo si fa subito ma si applica un'isteresi. In questo esempio si aspettano 300 secondi.

Se il massimo rispetto al minimo dei valori monitorati ogni 15 sec richiede l'abbassamento, allora si scala. ScaleDown.

Si aspetta quindi un po' per vedere quello che succede.

Per evitare il fenomeno del **flapping**.

Per la **scaleUp** invece la **stabilizationWindow** è di 0.

Viene cioè eseguita istantaneamente.

ScaleUp e scaleDown possono essere configurati diversamente l'uno dall'altro.

La sezione **policy** va a specificare quello che

effettivamente occorre fare: il valore per i pod da scalare va tra il **Max** e il valore che, per questo esempio 4, viene fuori dalla formula precedente.

Questa è la policy di base ma si può modificare in base ai parametri d'interesse, come il numero delle risorse, la latenza desiderata, etc.

È possibile creare un HPA anche in maniera imperativa da linea di comando.

```
behavior:  
  scaleDown:  
    stabilizationWindowSeconds: 300  
    policies:  
      - type: Percent  
        value: 100  
        periodSeconds: 15  
  scaleUp:  
    stabilizationWindowSeconds: 0  
    policies:  
      - type: Percent  
        value: 100  
        periodSeconds: 15  
      - type: Pods  
        value: 4  
        periodSeconds: 15  
    selectPolicy: Max
```

`kubectl autoscale rs foo --min=2 --max=5 --cpu-percent=80` creerà un HPA per il ReplicaSet foo, con un utilizzo della CPU fino ad un massimo dell'80% e con un numero di repliche compreso tra 2 e 5.

Example: Autoscaling applications using HPA for CPU Usage

Create deployment

```
[root@controller ~]# cat nginx-deploy.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    type: dev
  name: nginx-deploy
```

```
[root@controller ~]# kubectl create -f nginx-deploy.yaml
deployment.apps/nginx-deploy created
```

```
spec:
replicas: 1
selector:
matchLabels:
  type: dev
template:
  metadata:
    labels:
      type: dev
  spec:
    containers:
      - image: nginx
        name: nginx
      ports:
        - containerPort: 80
    resources:
      limits:
        cpu: 500m
      requests:
        cpu: 200m
```

Example: Autoscaling applications using HPA for CPU Usage

Create deployment

```
[root@controller ~]# cat nginx-deploy.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    type: dev
  name: nginx-deploy
```

```
[root@controller ~]# kubectl create -f nginx-deploy.yaml
deployment.apps/nginx-deploy created
```

```
spec:
replicas: 1
selector:
matchLabels:
  type: dev
template:
  metadata:
    labels:
      type: dev
  spec:
    containers:
      - image: nginx
        name: nginx
      ports:
        - containerPort: 80
    resources:
      limits:
        cpu: 500m
      requests:
        cpu: 200m
```

Kubeadm: strumento creato per fornire kubeadm init e kubeadm join come "percorsi rapidi" di best practice per la creazione di un cluster Kubernetes. Esegue le azioni necessarie per rendere operativo un cluster minimo valido.

Capitolo 4: Cloud Networks

Quando si parla di Cloud ci si riferisce a grandi moli di dati conservati in strutture apposite sparse per il mondo, ciascuna delle quali contiene al suo interno migliaia di server. Una Cloud non è un concetto astratto ma un vero e proprio spazio fisico con i suoi confini, requisiti e limitazioni.

Al giorno d'oggi ci si aspetta che un utente paghi solo per quello di cui ha bisogno e in base al tempo effettivo di utilizzo della rete Cloud.

Una Cloud fornisce ad ogni utente un proprio ambiente con dentro tutto il servizio richiesto, l'utente ha la sensazione che la Cloud sia tutta per sé.

Le reti Cloud risultano utili in quanto offrono una serie di vantaggi a livello economico. Infatti, grazie ad esse non occorre pagare per la gestione del sistema, per l'affitto di grandi server, per la gestione di grandi reti.

Sono tutti aspetti che vengono offerti da chi offre la Cloud come servizio, non occorre preoccuparsi della maggior parte dei problemi che invece sarebbero presenti nel caso in cui si decidesse di realizzare e mantenere tutto in proprio. Inoltre, una Cloud offre scalabilità in base al traffico di utenti (ha un costo). Un altro vantaggio è quello di avere una garanzia sul mantenimento dei propri dati, in quanto essi vengono salvati in diverse località evitando problemi di perdita dovuti a disastri naturali, guerre, ecc.

Inoltre, una Cloud offre garanzie in termini di aggiornamenti software con patch di sicurezza annesse (cosa che richiederebbe un costo nel caso in cui si facesse tutto da soli), in quanto è premura dei provider di queste reti per attirare sempre più utenti e aziende. Ovviamente ci sono anche garanzie in termini di sicurezza fisica per i server cosa che invece sarebbe da aggiungere alla lista dei costi nel caso in cui si facesse tutto in proprio.

Una Cloud può comunque presentare una serie di svantaggi in determinati contesti, ad esempio, se un'applicazione è particolarmente monolitica con un afflusso di richieste standard e fisse che non richiedono una particolare scalabilità, oppure se un'applicazione gira particolarmente bene in locale e richiede una particolare attenzione per la riscrittura della stessa per il trasferimento in Cloud.

Gli inconvenienti più importanti sono la privacy e la gestione dei propri dati. Sebbene i provider abbiano molta premura nel garantire una sicurezza per i dati degli utenti e aziende che richiedono il loro servizio, ciò non toglie il fatto che i dati in questione vengono gestiti dal provider e non dagli interessati, fiducia e privacy sono sempre da tenere in considerazione.

Ci sono casi in cui, per legge o per accordi sulla privacy, un individuo desidera cancellare le proprie informazioni private. Come ci si può assicurare che il fornitore di servizi Cloud lo faccia effettivamente?

Qual è la politica del provider di servizi Cloud in merito alla notifica in caso di violazione dei dati? Quanto tempo occorrerebbe per tornare alla normalità? C'è e, se c'è, qual è la nostra responsabilità nel notificare al provider di servizi Cloud che i nostri dati sono stati violati?

Nelle grandi aziende Cloud con data center situati in altri Paesi (forse non così amichevoli con il nostro), l'ubicazione fisica dei dati può essere un grosso problema. Come facciamo a sapere che il provider di servizi Cloud non estrae i nostri dati privati per i propri scopi e non vende queste informazioni a terzi? Qual è la responsabilità dell'azienda se il provider Cloud fa questo?

Aspetti Generali

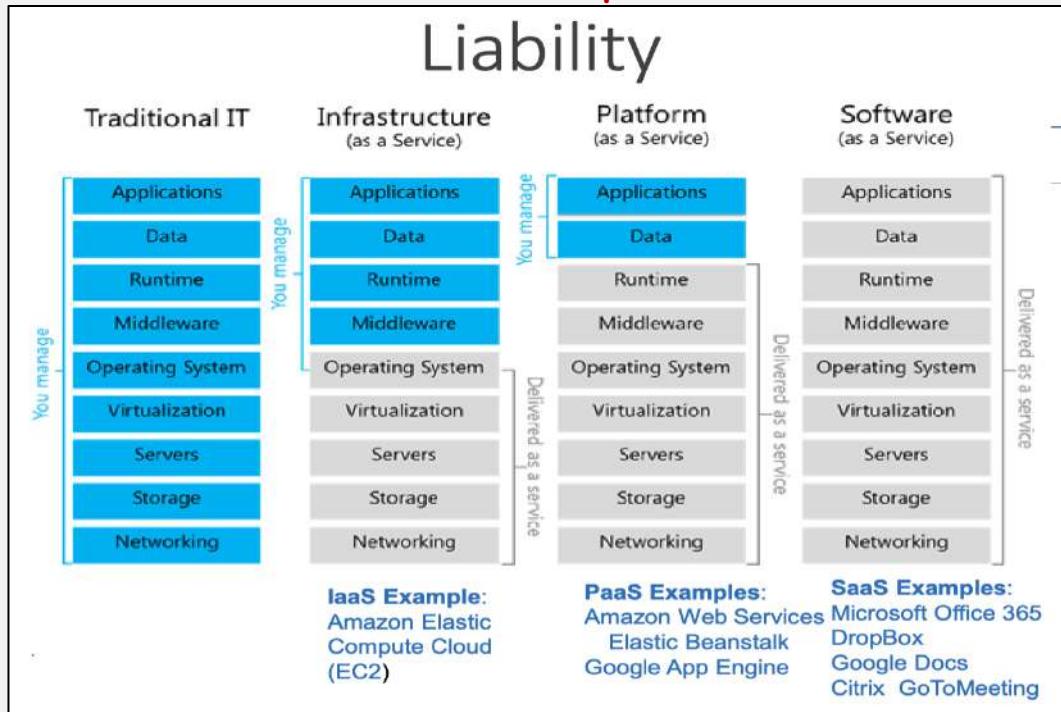
Oltre agli aspetti già analizzati, una rete Cloud include dei **modelli di servizio** e dei **modelli di distribuzione**. Per quanto riguarda i **Service Models** abbiamo:

- **Software as a Service** (SaaS): software erogato come servizio in cloud. Offre un'applicazione al client, come se un utente utilizzasse Word non sul suo PC ma accedendovi dal browser, il servizio offerto gira in cloud.
- **Platform as a Service** (PaaS): piattaforma erogata come servizio, magari già con del software preinstallato, ecc. Si ha un controllo sulle configurazioni dell'ambiente di hosting delle applicazioni.
- **Infrastructure as a Service** (IaaS): offre un collegamento alla Cloud da remoto, fornendo una macchina virtuale con la quale l'utente può fare ciò che vuole, inclusa la scelta del sistema operativo con cui farla girare.

Queste tre categorie sono le più generali per i modelli di servizio in cloud.

- **Function as a Service** (FaaS): invece di offrire all'utente una VM con tutti i processi annessi si offrono le function, ovvero delle micro-VM che istanziano dei micro-container con solo le applicazioni necessarie. Si paga solo per quello che si utilizza e non per tutto il sistema operativo. Di base potrebbe essere tutto spento, l'accensione si ha quando arriva una richiesta con la conseguente attivazione del microservizio desiderato. Conosciuto anche come modello **serverless**, se il servizio non fa guadagnare allora si spegne. Categoria in diffusione.
L'infrastruttura (server e hardware di rete) è completamente gestita dal provider, i developer si concentrano solo sul codice e sviluppo app.
Esempio: Amazon Lambda. Conviene quando l'utilizzo è a singhiozzi, se il server è sempre attivo allora conviene la modalità tradizionale.

Gestione delle responsabilità



Trade off tra ciò che in Cloud gestisce l'utente e ciò che gestisce il provider.

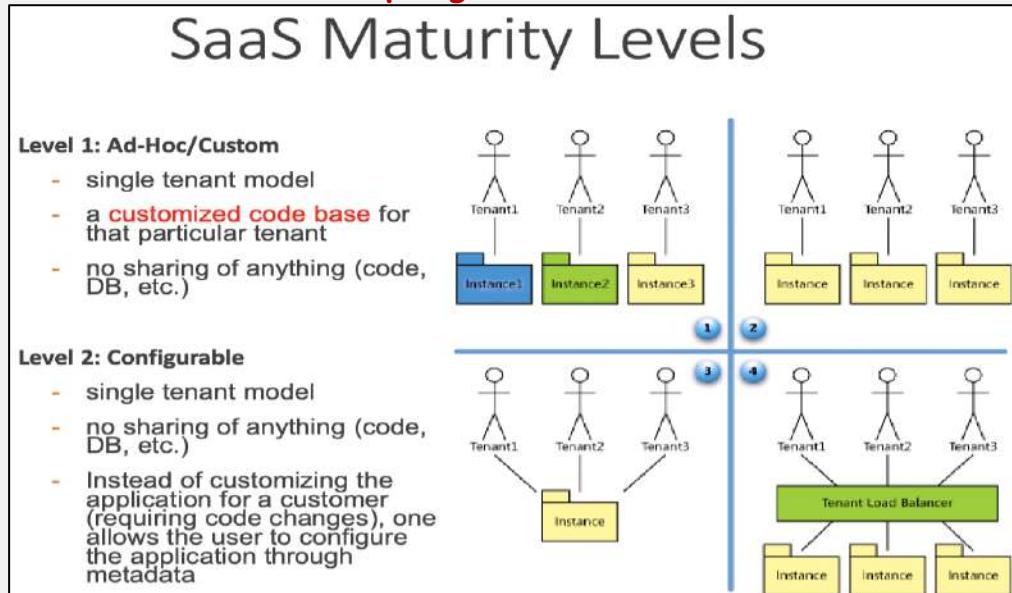
In **Cloud private** (Traditional IT) l'utente o azienda è responsabile di tutto.

L'estremo opposto è **Software as a Service**, dove gestisce tutto il provider.

Le **vie di mezzo** sono **IaaS**, dove il provider gestisce fino al S.O. e l'utente è responsabile di quello che ci mette sopra, e **PaaS**, dove il provider gestisce tutto tranne Data e Applicazioni che sono gestite dall'utente o azienda.

Runtime descrive le istruzioni che vengono eseguite mentre un programma è in esecuzione, in particolare quelle istruzioni che non sono scritte esplicitamente ma che sono necessarie per la corretta esecuzione del codice.

Tipologie di offerte



1. Vengono fornite delle istanze di applicazioni customizzate per ogni singolo utente. Non offre scalabilità, poco efficiente per il provider.
2. Viene offerta la stessa istanza a più utenti, effetto sub lineare.
3. Viene migliorato l'utilizzo delle risorse. Vengono condivise le risorse della stessa istanza ma con diversi input. Ogni utente ha la sensazione di usare l'applicazione in esclusiva ma di fatto è condivisa.
4. **Usata attualmente:** consente di istanziare le istanze indipendentemente dal numero di utenti. Si utilizza un Load Balancer che distribuisce le risorse in base alle richieste che arrivano. Non c'è nessuna forma di preallocazione, è tutto dinamico. Tenant x <- Load Balancer -> risorse. È altamente scalabile ed elastico. Comporta enormi risparmi sui costi.

Dalla 1 verso la 4 il costo diminuisce a fronte della stessa offerta.

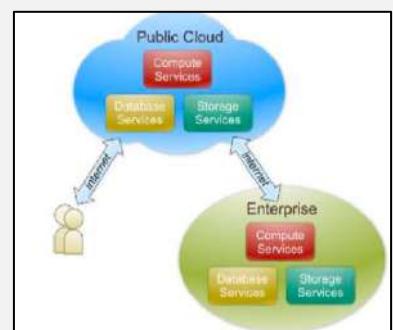
Per quanto riguarda i **Deployment Models** si hanno: Cloud pubbliche, Cloud private, Cloud Community e Cloud Ibride. Numericamente, le più diffuse sono le Cloud Ibride. Le loro **differenze risiedono principalmente nell'ambito e nell'accesso ai servizi** cloud offerti.

Le Cloud pubbliche sono infrastrutture omogenee.

Non si ha interesse nell'acquistare cose eterogenee, altrimenti occorrerebbe più personale da pagare.

Policy di gestione comune: **isolare gli utenti e ottenere una certa economia di scala** offrendo spazio di storage, potenza di calcolo, etc.

Adatta al grande pubblico o a grandi industrie.

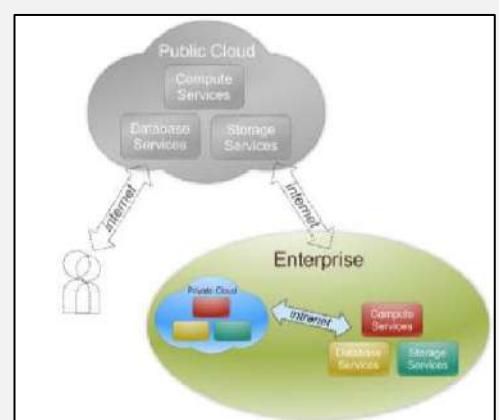


Le Cloud private sono infrastrutture tipicamente eterogenee.

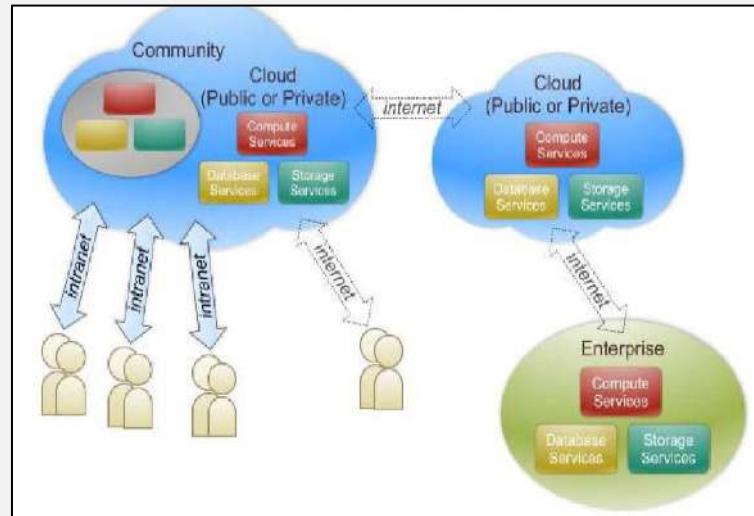
Hanno l'obiettivo di realizzare privatamente una risorsa di **accesso disponibile solo per i dipendenti o un numero di persone stabilito**.

Si costruisce la Cloud privata e gli utenti vi accedono attraverso un endpoint che sta dentro l'azienda.

Amazon si presta a creare Cloud private mediante pagamento.



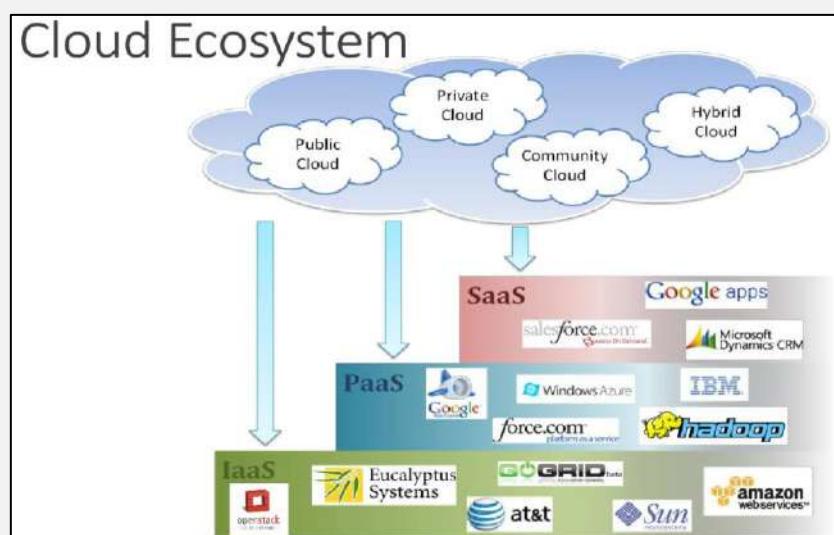
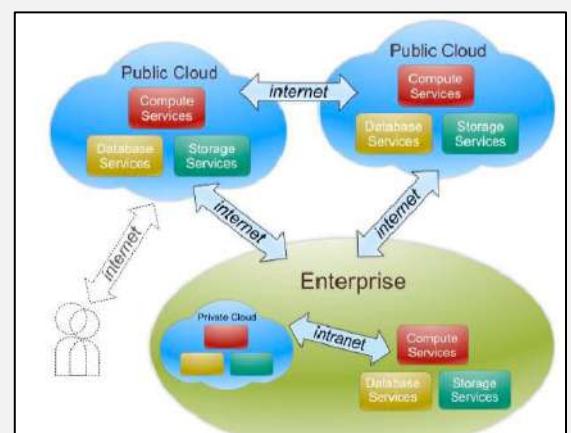
Le Cloud community sono infrastrutture private ma che non si limitano a offrire servizi ad una singola organizzazione ma a più. Se, ad esempio, gli ospedali dell'Umbria si coalizzassero, potrebbero implementare una loro Cloud Community per condividere tutto il software all'interno di una rete accessibile solo a loro. Una sorta di **Cloud condivisa a specifiche organizzazioni con obbiettivi comuni**. Uno degli obbiettivi è quello di condividere le risorse per risparmiare denaro.



Le Cloud ibride sono **infrastrutture composte da due o più Cloud di diversa natura** (pubblica, privata, community). È come avere una Cloud privata con delle risorse limitate e utilizzabili solo dall'organizzazione. **Se però durante l'utilizzo le risorse non sono abbastanza allora si può appoggiare ad una o più Cloud pubbliche**.

Ad esempio, le API di Amazon o Microsoft potrebbero programmare Kubernetes per fare Cloud Bursting ovvero irrompere in una Cloud pubblica quando la domanda di elaborazione aumenta e occorre scalare verso l'alto.

Si paga in base a se e quanto le cloud pubbliche vengono utilizzate.



Problemi di sicurezza

I problemi di sicurezza sono sempre presenti anche quando si parla di ambienti Cloud. Occorre analizzare e comprendere i vari scenari.

Perdita di dati: può succedere disgraziatamente che il provider a cui un utente si appoggia per lo storage dei suoi dati li perda o peggio ancora li renda accessibili a terzi. In quel caso cosa succede?

Downtimes: potrebbero verificarsi casi in cui per un certo periodo di tempo il provider non è in grado di fornire il servizio. In quel caso cosa succede?

Phishing: frode informatica eseguita mascherando un sito malevolo come un normale sito web a cui si accede, con lo scopo di rubare soldi tramite password.

Password Cracking: potrebbe accadere che utenti malevoli sfruttino le risorse di calcolo di una rete Cloud per eseguire attacchi contro altre entità.

Botnets and Malware: rete composta da dispositivi infettati da malware, detti bot o zombie, che agiscono tutti sotto controllo di un unico dispositivo master, aumentando esponenzialmente le capacità offensive dell'attaccante.

Virtualization Security

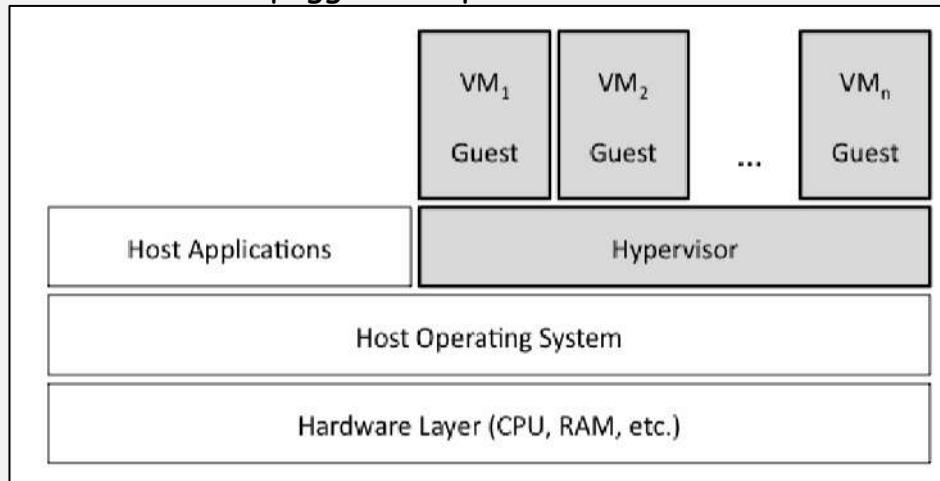
Caratteristiche: isolamento, istantanee (snapshot).

Problemi: ripristino dello stato, ridimensionamento, fugacità, durata dei dati.

Usare una VM per ogni singola applicazione garantisce l'isolamento.

È meglio avere l'isolamento che girare due applicazioni sullo stesso server.

L'isolamento tramite VM è peggiore rispetto all'esecuzione su due server fisici.

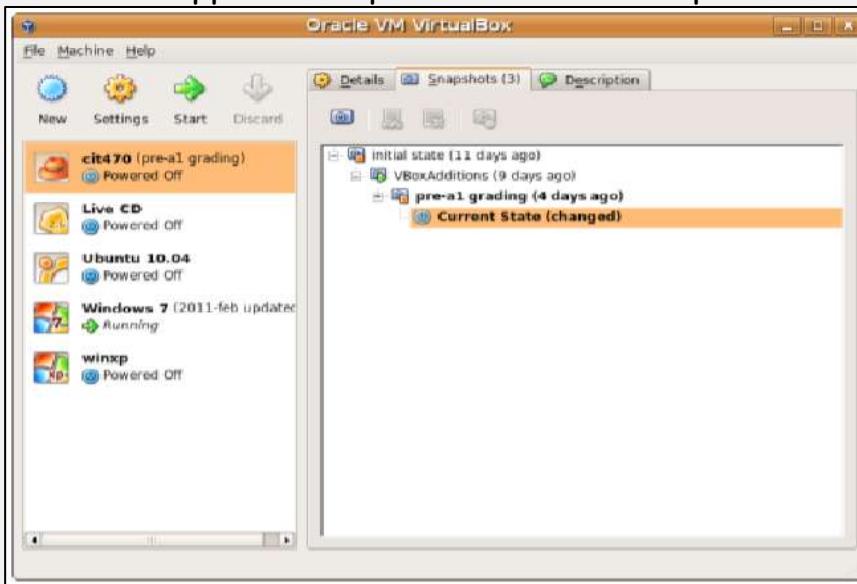


Features: Isolation

Meltdown è una vulnerabilità hardware che colpisce microprocessori Intel e ARM e permette a programmi e potenziali attaccanti di accedere ad aree protette di memoria di un computer.

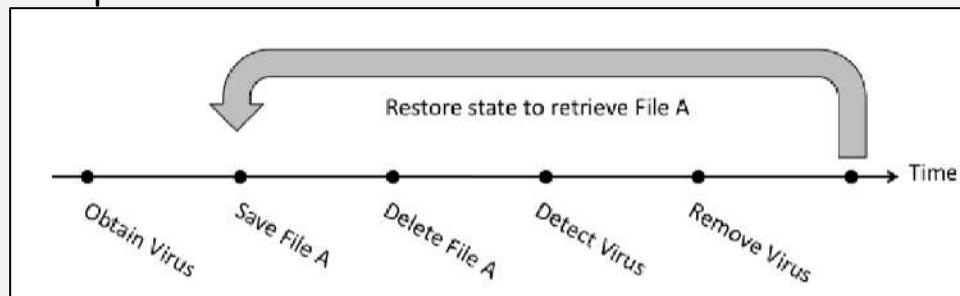
Le VM possono registrare degli stati. In caso di problemi è possibile ritornare ad una versione precedente attraverso uno stato del sistema precedente.

Dobbiamo ricordarci di applicare le patch di correzione per evitare ricadute.



Vari stati di un sistema operativo, i backup di VirtualBox.

State Restore: un possibile problema del recupero di uno stato precedente è che la VM si potrebbe infettare senza accorgersene. Quindi eseguendo i backup si riavrebbe comunque uno stato infettato da un virus. Le patch applicate vengono annullate perché il virus persiste nelle profondità della VM causando una ripetizione continua di uno stato della macchina infetta.



Scaling: l'aumento delle macchine fisiche non sempre può essere fatto, magari per limiti di budget o di tempo per la configurazione. Aumentare le macchine virtuali, invece, è facile come copiare un file, portando ad una crescita esplosiva del loro numero. Scalare troppo rapidamente può essere sconveniente se si eccede la capacità di organizzarsi a livello di sicurezza di sistema.

Esistono anche altre **vulnerabilità di diversa natura** da risolvere, come:

- Responsabilità.
- Nessun perimetro di sicurezza.
- Superficie di attacco più ampia.
- Nuovi canali laterali.
- Mancanza di verificabilità.
- Conformità normativa.

Per quanto riguarda la **responsabilità**, per stabilire chi deve detenerla si va a guardare dove è ospitato il carico di lavoro.

La prima cosa da considerare quando c'è un problema è quella di capire di chi è la colpa tra il provider e il cliente. Se succede qualcosa come virus, accesso non consentito, perdita dei dati etc. **Chi paga per i danni?**

Se il cliente si fa le cose in casa allora la colpa sarà tutta sua.

Se il cliente accede ad una Cloud IaaS allora il cloud provider è il responsabile dell'infrastruttura fisica, mentre il client è responsabile di tutto il resto a partire dall'uso del sistema operativo.

Se il cliente accede ad una Cloud PaaS allora il cloud provider è il responsabile fino alle librerie e applicazioni di base, mentre il cliente è responsabile di tutto ciò che va ad installare.

Se il cliente accede ad una cloud SaaS allora il cloud provider è il responsabile di tutta l'infrastruttura fisica, del sistema operativo, delle applicazioni che sono installate e del meccanismo di autenticazione del cliente, mentre il cliente è il responsabile solo nel caso in cui non gestisca nel modo migliore le proprie password causando un'intrusione nella rete cloud.

Ci sono anche delle responsabilità che sono sempre del cliente o dell'azienda, ovvero i dati con le proprie informazioni, dispositivi, gli account e le identità.

In generale, **il vantaggio del modello di responsabilità condivisa è che le aziende hanno chiare le proprie responsabilità e quelle del fornitore di servizi cloud.**

	Responsibility				On-Prem	
	SaaS	PaaS	IaaS			
Mode	Information and data					RESPONSIBILITY ALWAYS RETAINED BY CUSTOMER
	Devices (Mobile and PCs)					
	Accounts and identities					
	Identity and directory infrastructure					
	Applications					
	Network controls					
	Operating system					
	Physical hosts					
	Physical network					
	Physical datacenter					
 Microsoft Provider Customer						

Per quanto riguarda la **sicurezza perimetrale** c'è da dire che in una rete cloud non c'è un vero e proprio perimetro in quanto potrebbe essere distribuita in tutto il mondo con macchine virtuali che vanno e vengono.

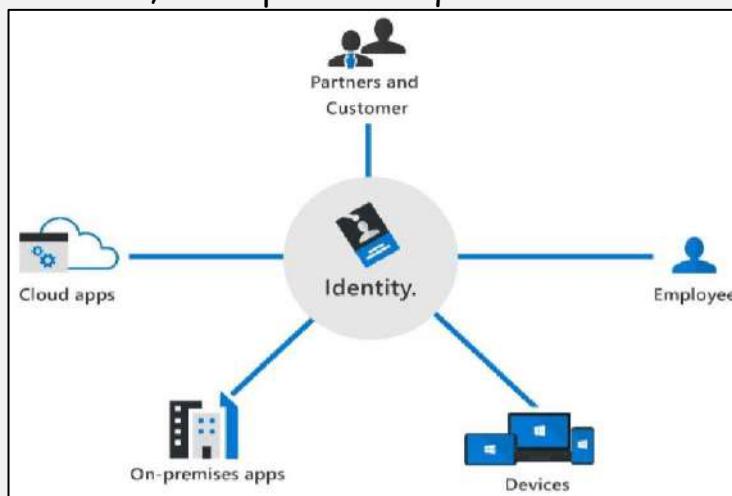
Poco controllo sulla posizione fisica o di rete delle macchine virtuali delle istanze cloud. L'accesso alla rete dovrebbe essere controllato host per host. Il provider espone gli **endpoint del servizio**, ma se quest'ultimi vengono progettati dall'azienda o dal cliente allora occorre pensarci da sé.

Quando si parla di perimetro nelle reti cloud non bisogna pensare alle macchine fisiche ma a dove si accede.

L'identità è il primo perimetro di sicurezza.

Ma cosa intendiamo per identità?

Un'identità è il modo in cui qualcuno o qualcosa può essere verificato e autenticato per essere chi dice di essere. Un'identità può essere associata a un utente, un'applicazione, un dispositivo o qualcos'altro.



Il modello che sta emergendo è il **modello Zero Trust** ovvero non fidarsi di nessuno. I principi su cui si basa questo approccio sono tre:

- 1) **Verifica esplicita di tutto**: mai dare nulla per scontato, si opera con il presupposto che qualcosa di imprevisto possa succedere.
Utilizzare autenticazione a due o più fattori.
- 2) **Garantire livelli di privilegi minimi e indispensabili**: agli operatori si danno privilegi minimi per fare il loro lavoro. Se uno deve usare solo un dataset è inutile dargli l'accesso a tutto il database.
- 3) **Assumere che un breach (violazione) possa esserci in qualsiasi momento**: segmentare l'accesso per rete, utente, dispositivi e applicazione. Utilizzare la crittografia per proteggere i dati.
Rilevare attraverso delle analisi eventuali minacce per migliorare la sicurezza, quindi verificare tutto e andare a monitorare l'attività di chi può accedere all'infrastruttura.

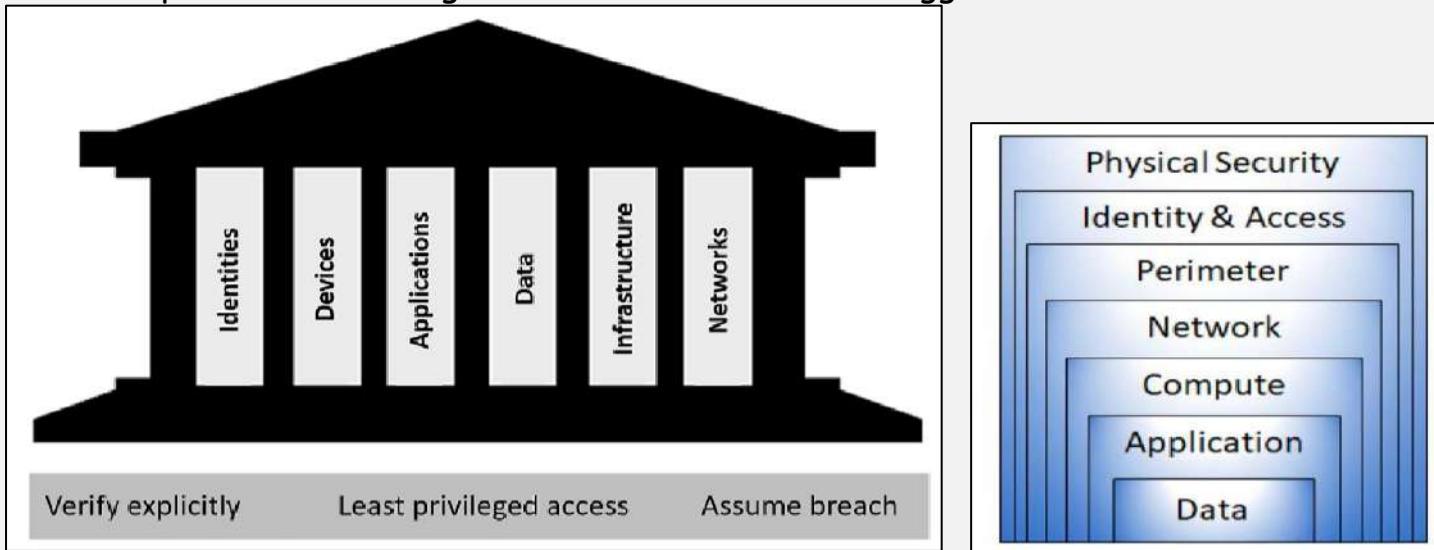
In generale, le vulnerabilità si evolvono nel tempo.

Nel modello Zero Trust, tutti gli elementi lavorano insieme per fornire sicurezza end-to-end.

Questi sei elementi sono i **pilastri fondamentali del modello Zero Trust**:

- Le **identità** possono essere utenti, servizi o dispositivi. Quando un'identità tenta di accedere ad una risorsa, deve essere verificata con un'autenticazione forte e seguire i principi di accesso con privilegi minimi.
- I **dispositivi** creano un'ampia superficie di attacco mentre i dati fluiscono dai dispositivi ai carichi di lavoro locali e al cloud.
Il monitoraggio dei dispositivi per la salute e la conformità è un aspetto importante della sicurezza.
- Le **applicazioni** sono il mezzo con cui i dati vengono consumati.
Ciò include il rilevamento di tutte le applicazioni utilizzate e la gestione delle autorizzazioni e degli accessi.
- I **dati** devono essere classificati, etichettati e crittografati in base ai relativi attributi. Gli sforzi per la sicurezza riguardano la protezione dei dati e la garanzia che rimangano al sicuro quando lasciano dispositivi, applicazioni, infrastrutture e reti controllate dall'organizzazione.
- L'**infrastruttura**, sia locale che basata su cloud, rappresenta un vettore di minaccia. In caso di cloud locale occorre piazzare telecamere e scanner. Ciò consente di bloccare o segnalare automaticamente comportamenti sospetti e intraprendere azioni protettive.

- Le **reti** dovrebbero essere segmentate tra di loro e al loro interno. Inoltre, dovrebbero essere utilizzate la protezione dalle minacce in tempo reale, la crittografia end-to-end, il monitoraggio e l'analisi.



Modello a cipolla: i dati sono la cosa più preziosa e devono essere protetti. Ogni livello superiore assume una forma di protezione.

Un esempio di Cloud provider è **Amazon Cloud EC2** il quale fornisce all'utente un servizio web con tutta la potenza di calcolo che occorre per soddisfare le sue esigenze. Offre letteralmente un server del suo datacenter che è possibile utilizzare per ospitare il software utente. Si accede mediante GUI, linea di comando o API. Si paga solo per le risorse effettive che si utilizzano.

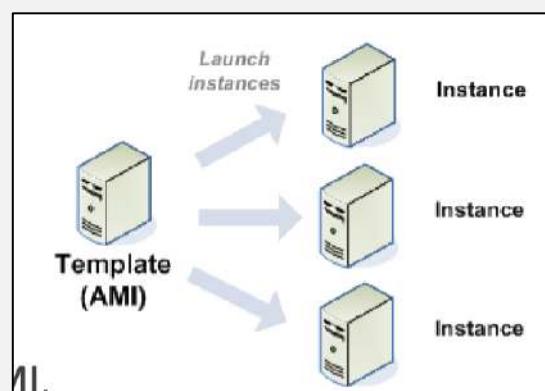
È un servizio di tipo IaaS, molto elastico, se c'è un aumento del traffico allora vengono istanziate diverse macchine virtuali.

Tra i principali componenti di EC2 (ci saranno anche in OpenStack) abbiamo l'immagine delle VM, le istanze, le regioni e zone disponibili, i servizi di storage, il monitoring, il load balancing, etc.

Le Amazon Machine Image (AMI) sono dei Template (file) con all'interno tutta la configurazione software necessaria per istanziare macchine virtuali.

Un equivalente del file ".ova" di VirtualBox.

AMI è quella che in OpenStack prende il nome di immagine, ovvero gli stampini. Utilizzate per realizzare delle VM. Quando si realizza una VM è possibile realizzare tutte le istanze che si vogliono e in forma customizzata a livello di core, RAM, memoria, porte aperte utilizzate, etc. Le istanze continuano a funzionare fino a quando non vengono interrotte o non si guastano. Se un'istanza non funziona, è possibile avviarne una nuova partendo dall'AMI.



È possibile utilizzare una o più AMI differenti a seconda delle esigenze.
Da una singola AMI è possibile, come detto, avviare diversi tipi di istanze.
Amazon pubblica molte AMI che contengono configurazioni software comuni per un utilizzo pubblico.

Amazon dispone di moltissimi datacenter sparsi in tutto il mondo.

Ciò permette di utilizzare EC2 in differenti Paesi tranquillamente.

Per la gestione dei dati occorre prestare attenzione ai vari garanti della privacy come il GDPR, a volte capita di dover garantire che i dati restino all'interno di una certa area geografica.

Amazon, organizzando la sua rete in regioni molto ampie, come uno Stato o un continente, gestisce queste problematiche.

All'interno di queste zone ci sono le **Availability Zones**.

Vengono realizzate

principalmente per associarle ai contratti degli utenti.

Ogni zona di disponibilità è progettata in modo da essere isolata dai guasti di altre zone di disponibilità. Un cliente o organizzazione può avviare le istanze della sua applicazione in zone di disponibilità separate in modo da proteggerle dai guasti di una singola posizione.

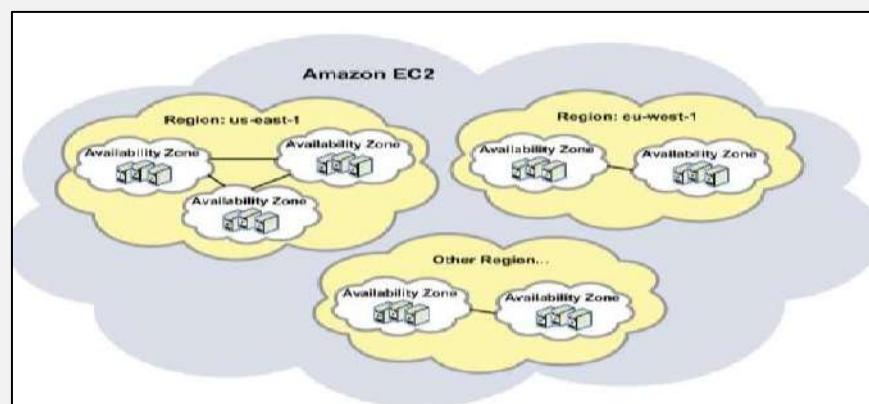
Quando si lancia un'istanza in Amazon occorre associargli uno **spazio di storage** per i dati. Esistono tre livelli di storage Amazon (e anche in OpenStack):

- **Storage legato all'istanza della VM**: lo spazio dove si appoggia il FS è la RAM del server. **Se si generano o scambiano dati, essi saranno presenti finché la VM è accesa**, una volta spenta viene perso tutto in quanto appunto lo storage è effimero perché gira sulla RAM.
- **Storage legato a volumi**: lo spazio dove si appoggia il FS è permanente. **Se la VM viene spenta i dati persistono e restano organizzati**.

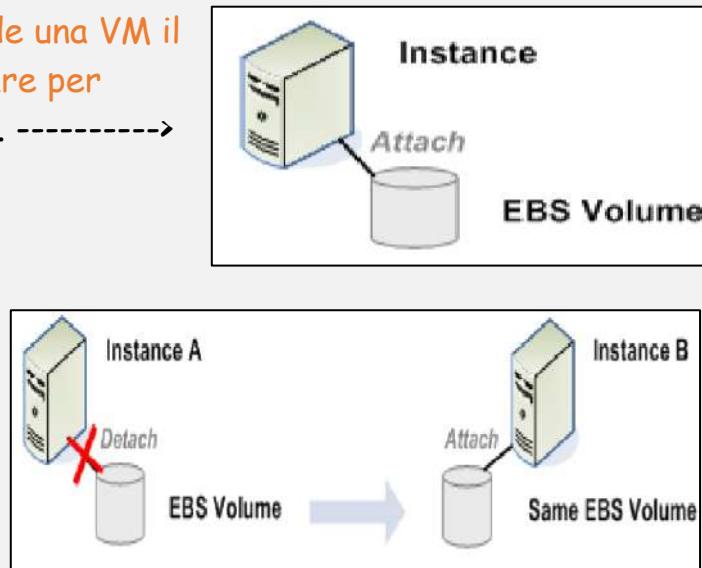
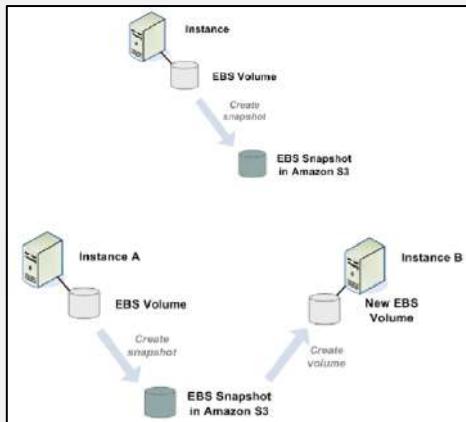
È uno spazio esterno che può essere attaccato all'istanza di una o più VM qualsiasi entrando a far parte del FS di quest'ultima.

Il volume è una componente indipendente e pertanto può essere replicato, backuppato, inserito, staccato e riattaccato ad un'altra macchina.

Sono volumi diversi da quelli presenti in Kubernetes. Lì, infatti, i volumi sono attaccati ai pod, eliminando un pod cade anche il volume.



Qui no, i volumi sono a sé stanti, **se cade una VM il volume con i dati resta e si può utilizzare per altro**. Prendono il nome di Amazon EBS.



- **Storage di tipo S3:** sono storage ad oggetti come Google Drive, OneDrive o DropBox. Si deposita un file (oggetto) e il servizio fornisce un API REST, quindi un URL, che fa riferimento a tale oggetto. Al giorno d'oggi sono molto diffusi. Utile per eseguire backup, condivisioni, distribuzioni, Data Analysis, etc.

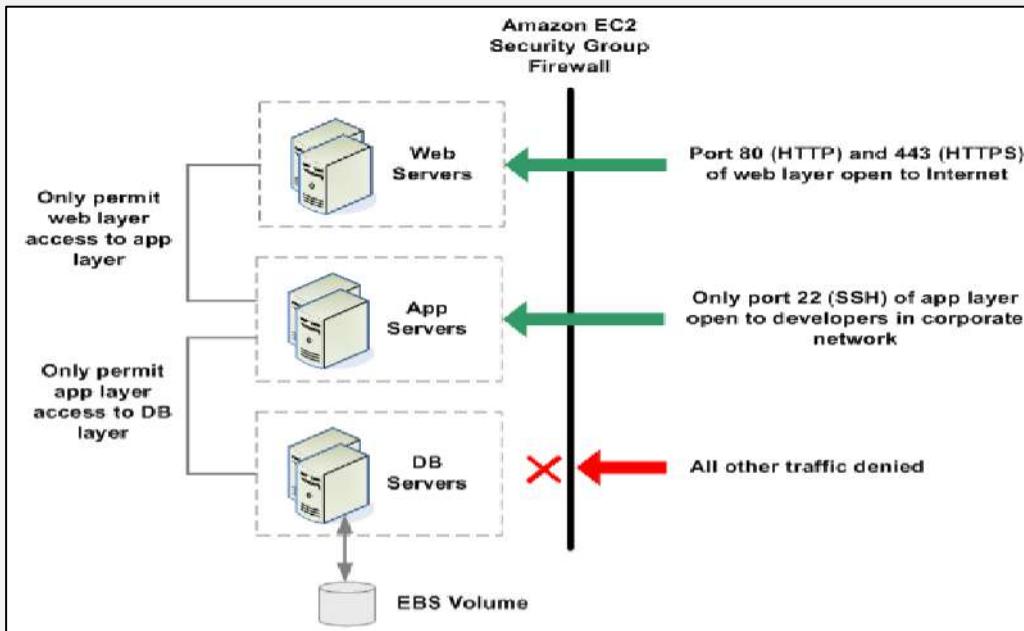
Se un'applicazione in esecuzione su EC2 necessita di un database allora possiamo utilizzare due metodi principali per implementarlo:

- Amazon Relational Database Service (Amazon RDS) per ottenere un database relazionale gestito in cloud.
- Avviare un'istanza di un'AMI di un database e utilizzare tale istanza EC2.

Amazon Cloud networking and security: quando si istanza una VM partendo dalla sua immagine (in Amazon, AMI) occorre specificare dei parametri per associarla a un **security group**, ovvero una sorta di firewall di tipo chiave-valore.

Ogni istanza che viene avviata nello spazio di rete Amazon EC2 è assegnata a un indirizzo IP pubblico. Se un'istanza ha esito negativo e viene riavviata, quest'ultima avrà un indirizzo IP pubblico diverso.

I **security group** vengono utilizzati per controllare l'accesso alle istanze utente. Sono analoghi a un firewall di rete in entrata e consentono ad un utente di specificare i protocolli, le porte e gli intervalli IP a cui è consentito raggiungere le istanze utente. Un utente può creare più security group con regole diverse. Possiamo vederli come dei firewall personalizzabili per ogni singola immagine. Ogni istanza può essere assegnata a uno o più security group che determineranno il traffico consentito verso l'istanza. Sono oggetti a sé stanti.

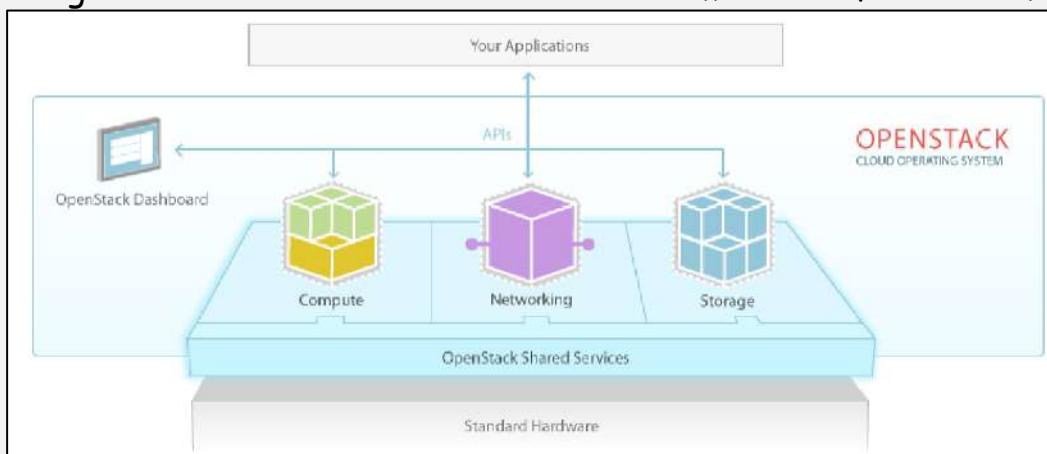


Utili per garantire che solo determinati indirizzi IP abbiano accesso all'istanza.

OpenStack (breve introduzione, il capitolo è più avanti)

OpenStack è un **progetto IaaS** open source utilizzato per **creare e gestire infrastrutture cloud**. Consente di creare e gestire macchine virtuali, archivi di dati, reti e altri servizi per l'elaborazione e lo storage dei dati.

OpenStack è progettato per essere altamente scalabile e può essere utilizzato per costruire cloud private, pubbliche o ibride. Il tutto gestito tramite una dashboard che offre agli amministratori il controllo e allo stesso tempo permette agli utenti di utilizzare delle risorse tramite interfaccia web.



Piuttosto che essere concepite come monoliti, le applicazioni sono scomposte in piccoli microservizi decentralizzati. Questi microservizi comunicano tramite API utilizzando la **gestione degli eventi** o la **messaggistica asincrona**. Le applicazioni scalano aggiungendo nuove istanze secondo necessità. I dati gestiti in cloud sono molto superiori all'ordine dei GB. Scala tutto enormemente rispetto ad un'ambiente locale.

Progettare applicazioni Cloud Native

Le progettazioni in Cloud differiscono in base al tipo di servizio utilizzato (IaaS, PaaS, SaaS, FaaS) e in base al tipo di storage utilizzato.

Sono diversi i database utilizzabili: relazionali, chiave-valore, documenti, ecc.

In base alla natura dei dati le query possono impiegare più o meno tempo per restituire una risposta. Per lo storage ci si affida al database che meglio si adatta alla loro manipolazione.

Un'applicazione cloud per essere scalabile, resiliente e gestibile deve soddisfare una serie di requisiti (visti anche in Kubernetes):

- **Proprietà self-healing**: in caso di errori, l'applicazione deve sapersi correggere e ripartire da sola.
- **Redundant**: ridondanza per evitare single point of failure.
- **Minimize coordination**: ridurre il coordinamento tra servizi applicativi per ottenere una migliore scalabilità.
- **Horizontal Scaling**: progettare l'applicazione in modo da poter aggiungere nuove istanze in caso di necessità.

OpenStack eroga servizi di tipo IaaS. L'obiettivo principale agli inizi era quello di erogare VM con un SO ai vari utenti in giro per il mondo.

Ad oggi, permette anche di erogare servizi basati su container.

OpenStack è un sistema distribuito accessibile dall'esterno attraverso reti IP.

L'interazione verso l'esterno avviene con client specifici di tipo REST o attraverso linea di comando. Ogni servizio è accessibile dall'esterno.

L'applicazione dev'essere pensata per evolversi, ogni decisione che dev'essere presa dev'essere giustificata da opportuni requisiti associati all'azienda.

Bisogna conoscere come vengono elaborate le richieste e il tipo di servizio che una specifica azienda fornisce.

Design Patterns

Possiamo considerare i Design Pattern come delle **best practices da applicare per una realizzazione ottimale delle applicazioni**. Una delle più grandi sfide è la gestione delle grandi moli di dati, sparsi in diverse località e che devono sempre essere pronti all'uso garantendo le proprietà viste precedentemente. Inoltre, i dati devono sempre essere protetti: nel riposo, nel transito, nell'uso.

Data Management Patterns

Pattern	Summary
Cache-Aside	Load data on demand into a cache from a data store
CQRS	Segregate operations that read data from operations that update data by using separate interfaces.
Event Sourcing	Use an append-only store to record the full series of events that describe actions taken on data in a domain.
Index Table	Create indexes over the fields in data stores that are frequently referenced by queries.
Materialized View	Generate prepopulated views over the data in one or more data stores when the data isn't ideally formatted for required query operations.
Sharding	Divide a data store into a set of horizontal partitions or shards .
Static Content Hosting	Deploy static content to a cloud-based storage service that can deliver them directly to the client.
Valet Key	Use a token or key that provides clients with restricted direct access to a specific resource or service.

Pattern per la gestione di grandi moli di dati di un sistema in una Cloud

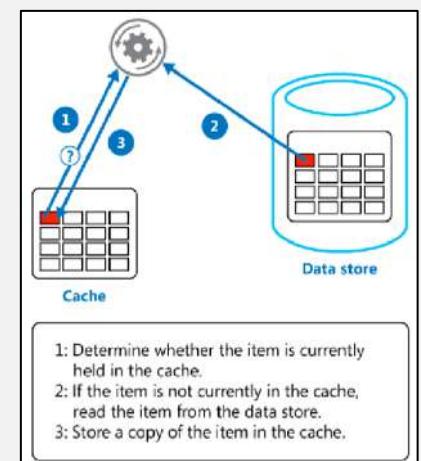
Cache-Aside: le cache sono dei depositi temporanei di dati, spesso vicini al cliente. Sono di piccole dimensioni, non possono esserci troppi riferimenti al database originale. La cache favorisce migliori prestazioni in quanto **il tempo per l'individuazione dei dati importanti, o più utilizzati, diminuisce**.

Tuttavia, **a volte si rinuncia alla consistenza**: le memorie cache dovrebbero essere mantenute aggiornate nel caso in cui i dati al loro interno invecchino o subiscano modifiche significative. Non sempre è possibile farlo in tempo reale e potrebbe verificarsi il caso in cui l'utente acceda ad una loro vecchia versione. L'applicazione dovrebbe controllare se i dati sono nella cache o eventualmente prenderli dal repository originario. **Se sono importanti o usati spesso** e sulla cache non ci sono allora si possono inserire, altrimenti no. L'aggiornamento della cache varia a seconda degli algoritmi utilizzati. **Write-Behind**: quando il client aggiorna i dati dev'esserci un riflesso sia nella cache che nel repository, oppure si eliminano dalla cache e si aggiorna solo il repository. Se queste funzioni non sono implementate nella cache allora l'applicazione deve farlo da sola.

Un'applicazione può emulare questa funzionalità della cache implementando il design pattern Cache-Aside.

Questa strategia carica i dati nella cache su richiesta.

In generale, se si conosce la popolarità dei contenuti allora è possibile fare caching come si vuole, ma in caso contrario, o in caso di cambiamenti veloci, cosa si inserisce nella cache?



È **consigliabile** utilizzare questo design pattern quando le richieste sono imprevedibili, garantendo un caricamento dei dati nella cache su richiesta, e quando la cache non è fornita nativamente di operazioni Read/Write through ed è quindi l'applicazione che deve pensare ad aggiornare i dati.

Non occorre utilizzarlo se il set di dati contenuto nella cache è statico.

CQRS: quando si eseguono operazioni di lettura e scrittura su applicazioni complesse possono insorgere dei problemi di consistenza dei dati.

Nella gestione delle operazioni ci possono essere problemi di contesa dei dati. In tal caso, i dati possono essere compromessi. Subentrano anche problemi di differenza di prestazioni e di requisiti di scala per tali operazioni read/write.

In generale, **occorre separare le operazioni di lettura da quelle di scrittura**.

Il design pattern CQRS (Command and Query Responsibility Segregation) si occupa proprio di separare le due operazioni di lettura e scrittura attraverso la realizzazione di due modelli separati, utilizzando determinati comandi per aggiornare i dati e determinate query per leggerli.

Come sempre, meglio implementare mediante approcci dichiarativi.

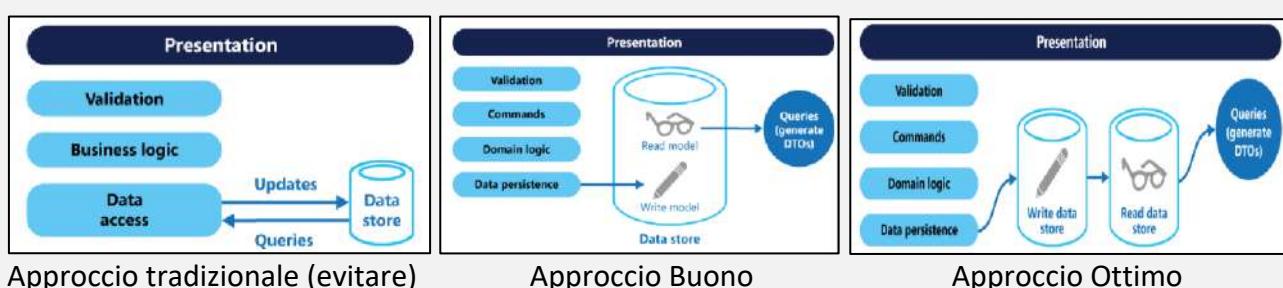
Separando le due operazioni si va a semplificare la logica delle applicazioni.

Le operazioni dovrebbero essere basate sui task e non sui dati e dovrebbero essere inserite in una coda per un'elaborazione asincrona piuttosto che essere elaborate in modo sincrono. Le query non dovrebbero modificare mai il db.

Per un isolamento ottimale si potrebbero separare fisicamente i dati riservati alla lettura da quelli riservati alla scrittura, offrendo delle strutture personalizzate per tali scopi. CQRS offre questa possibilità.

Ad esempio, il Read Data Store potrebbe utilizzare le Materialized View dei dati ed essere di tipo Document Database mentre il Write Data Store potrebbe essere di tipo Database Relazionale.

In cloud replicare il dataset per intero non è un problema, non è il massimo ma comunque si può fare. Si separano i dataset per la lettura e per la scrittura. Quando si effettua una replicazione del database, solitamente vengo apportate delle modifiche in modo da velocizzare entrambe le operazioni.



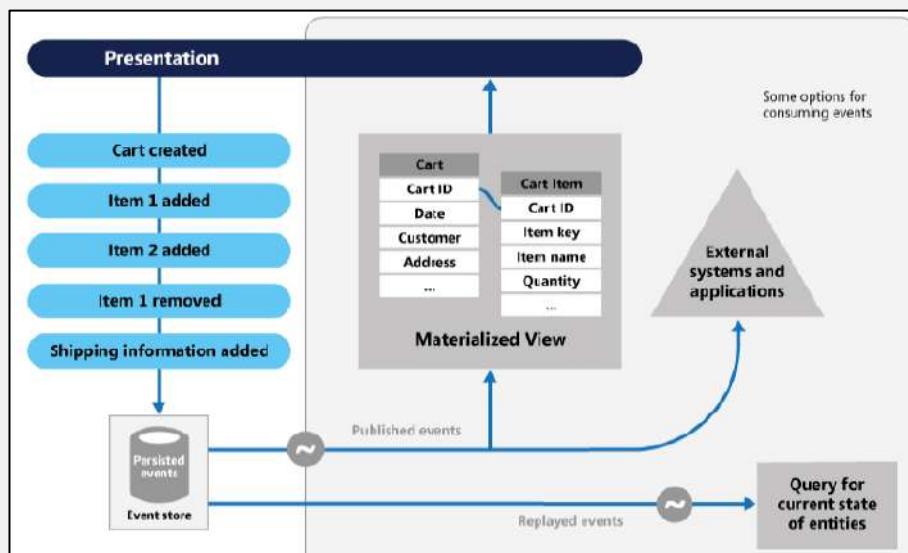
In generale, questo Design Pattern è sempre raccomandato (domini collaborativi, interfacce basate su task, ottimizzazione delle letture, evoluzione e integrazione dei sistemi, cambiamenti nelle regole di business).

Event Sourcing: immaginiamo uno scenario in cui delle applicazioni devono manipolare costantemente i dati, c'è un dataset per le prenotazioni dei biglietti aerei con tanti operatori che ricevono gli ordini e devono gestire l'utenza. Tali operazioni devono essere coordinate, come si verificano i posti disponibili? L'approccio tradizionale prevede che l'applicazione mantenga lo stato corrente dei dati aggiornandolo man mano che gli operatori vi lavorano attraverso le operazioni CRUD (create, read, update, delete).

Tuttavia, quest'approccio è molto limitante in quanto le operazioni di aggiornamento dei dati vanno ad intaccare direttamente il database, rallentandone le performance. In ambienti collaborativi ci potrebbero essere dei conflitti durante l'aggiornamento dello stesso dato e, salvo meccanismi appositi, si andrebbe a perdere la cronologia delle operazioni.

Il design pattern Event Sourcing definisce un approccio per la gestione delle operazioni sui dati guidato da una sequenza di eventi, ognuna delle quali è registrata in un archivio di sola aggiunta (non permette operazioni di delete).

L'applicazione invia una serie di eventi, che descrivono in modo imperativo ogni azione che si è verificata sui dati, ad un archivio degli eventi, il quale dispone di una memoria permanente e può essere sempre consultato.



In qualsiasi momento, per l'applicazione è possibile leggere la cronologia degli eventi.

Il pattern Event Sourcing è legato al concetto del Teorema CAP: risoluzione dei conflitti, rollback agli stati precedenti, ecc.

In generale, **Event Sourcing è sempre utilizzato in Cloud** con un approccio di tipo Eventual Consistency (accettiamo che magari il sistema non sia consistente per un breve periodo a patto che torni ad esserlo entro un tot di tempo).

Questo Design Pattern **potrebbe non essere utile se** si ha a che fare con piccoli o semplici domini che hanno poca o nessuna logica aziendale o sistemi che funzionano bene con i tradizionali meccanismi CRUD per la gestione dei dati. Oppure, in sistemi in cui sono richiesti coerenza e aggiornamenti in tempo reale delle visualizzazioni dei dati. O ancora, in sistemi in cui non sono richieste cronologie o funzionalità che eseguono rollback per ripetere delle azioni. O per sistemi che aggiungono prevalentemente dati piuttosto che aggiornarli.

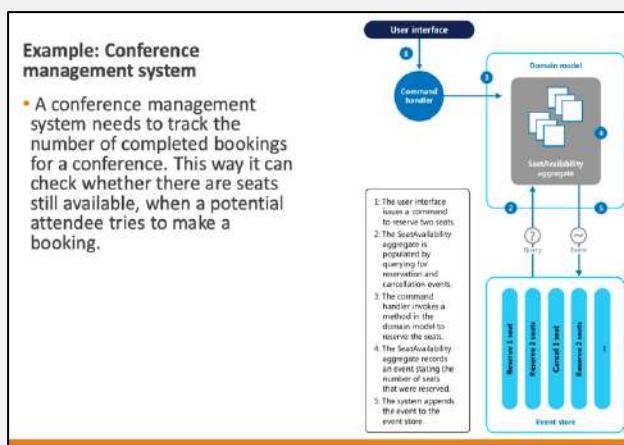
Index Table: pattern che **si occupa di creare degli indici in alcuni campi dei dati archiviati a cui fanno spesso riferimento le query.**

Sebbene la chiave primaria sia preziosa per le query, in quanto recuperano i dati in base al valore di questa chiave, un'applicazione potrebbe non essere in grado di utilizzarla se deve recuperare i dati in base a un altro campo.

Esempio: l'ID è la chiave primaria utilizzata per recuperare i dati dei clienti, ma un'applicazione interroga i dati esclusivamente facendo riferimento al valore di qualche altro attributo, come la CITTÀ in cui si trova il cliente. A questo punto l'applicazione per ottenere i dati dovrebbe recuperare ed esaminare ogni record del cliente, impiegando molto tempo.

Molti sistemi di gestione di database relazionali supportano le chiavi secondarie, ma la maggior parte dei database NoSQL utilizzati dalle applicazioni cloud non forniscono una funzionalità equivalente.

Questo pattern va a migliorare le prestazioni delle query consentendo alle applicazioni di individuare più rapidamente i dati da recuperare da un database. Ad esempio, se non è presente una chiave secondaria, o qualcosa di simile, allora il pattern va ad implementarla attraverso un'emulazione manuale.



Per strutturare un Index Table vengono comunemente utilizzate **tre strategie**, in base al numero di indici secondari richiesti e della natura stessa delle query eseguite da un'applicazione.

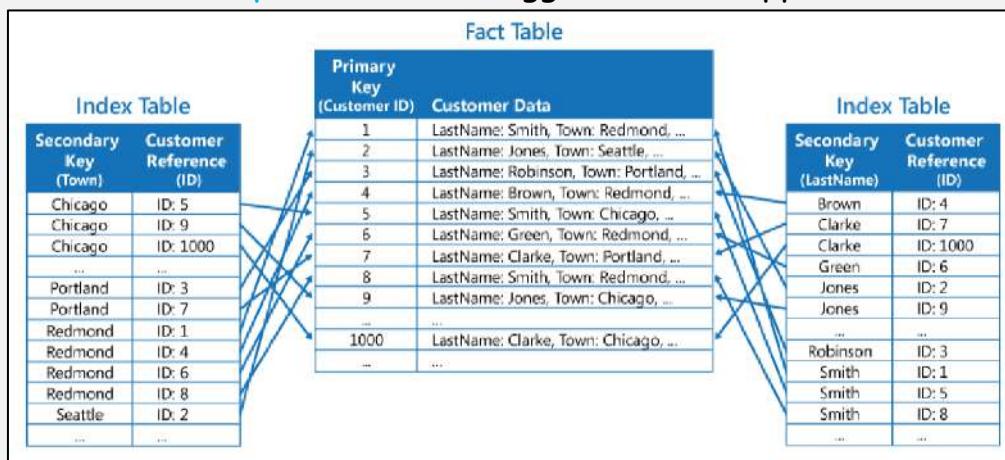
La prima è un approccio di replica. Ad esempio, abbiamo la chiave primaria che è l'ID dell'utente ma in base all'utilizzo dei dati ci si accorge che la città di origine e il cognome sono molto più utilizzati, allora è possibile replicare i dati assegnando come chiave secondaria il cognome o la città di origine.

Secondary Key (Town)	Customer Data	Secondary Key (LastName)	Customer Data
Chicago	ID: 5, LastName: Smith, Town: Chicago, ...	Brown	ID: 4, LastName: Brown, Town: Redmond, ...
Chicago	ID: 9, LastName: Jones, Town: Chicago, ...	Clarke	ID: 7, LastName: Clarke, Town: Portland, ...
Chicago	ID: 1000, LastName: Clarke, Town: Chicago, ...	Clarke	ID: 1000, LastName: Clarke, Town: Chicago, ...
...	...	Green	ID: 6, LastName: Green, Town: Redmond, ...
Portland	ID: 3, LastName: Robinson, Town: Portland, ...	Jones	ID: 2, LastName: Jones, Town: Seattle, ...
Portland	ID: 7, LastName: Clarke, Town: Portland, ...	Jones	ID: 9, LastName: Jones, Town: Chicago, ...
Redmond	ID: 1, LastName: Smith, Town: Redmond,
Redmond	ID: 4, LastName: Brown, Town: Redmond, ...	Robinson	ID: 3, LastName: Robinson, Town: Portland, ...
Redmond	ID: 6, LastName: Green, Town: Redmond, ...	Smith	ID: 1, LastName: Smith, Town: Redmond, ...
Redmond	ID: 8, LastName: Smith, Town: Redmond, ...	Smith	ID: 5, LastName: Smith, Town: Chicago, ...
Seattle	ID: 2, LastName: Jones, Town: Seattle, ...	Smith	ID: 8, LastName: Smith, Town: Redmond, ...
...

Con questo approccio si denormalizzano completamente i dati di una tabella, cioè i dati di più tabelle vengono combinati in un'unica tabella, in modo da velocizzare il loro recupero. L'obiettivo è quello di semplificare le operazioni di interrogazione ed elaborazione dei dati, anche se questa pratica può portare a ridurre l'integrità e la coerenza dei dati stessi.

La seconda è un approccio di tabella degli indici normalizzata. Si mantengono i dati di origine con la chiave primaria e si realizzano delle tabelle ausiliarie con delle chiavi secondarie che vengono mappate sulla tabella principale.

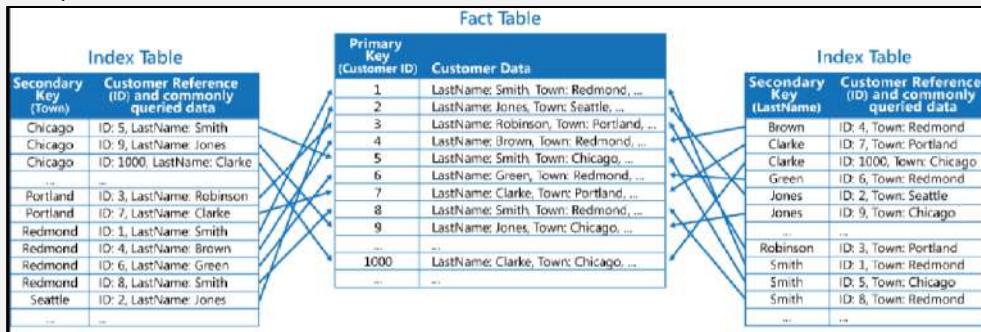
L'obiettivo è quello di risparmiare spazio e ridurre il sovraccarico dovuto al mantenimento di dati duplicati. Lo svantaggio è che un'applicazione deve



eseguire due operazioni di ricerca per trovare i dati utilizzando una chiave secondaria. Trova la chiave primaria per i dati nell'Index Table grazie alla secondaria e poi utilizza tale chiave per cercare i dati nella Fact Table. Offre una gestione completa ma ha problemi legati ai riferimenti e alla latenza.

La terza è un approccio che fondamentalmente consiste nel realizzare delle Index Table più articolate con più informazioni, utilizzando più memoria ma risolvendo il problema del doppio accesso (passando da 2 a 1).

Chiaramente, occorre conoscere l'utilizzo dei dati.



Realizza una chiave ausiliaria unendo più informazioni, come città e cognome. Tutti questi indici generano overhead, sono codici da annettere in più. In generale, questo design pattern dev'essere valutato prima dell'uso.

Materialized View: uno dei pattern principali, uovo di colombo.

Creiamo una view, ovvero una nuova visione. Una sorta di nuovo dataset più piccolo, con solo le informazioni che interessano in un certo contesto.

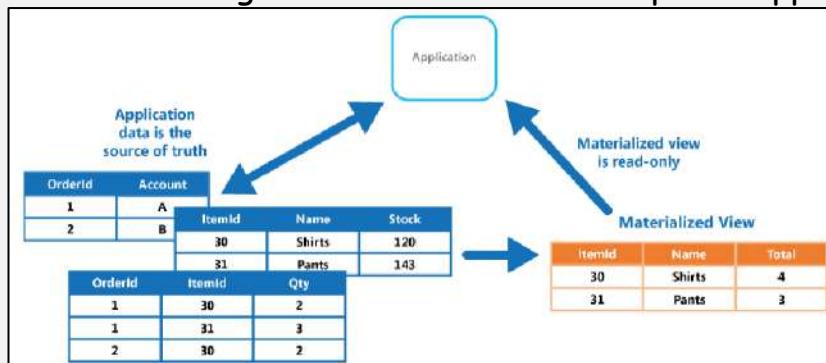
Utilizzato per migliorare le prestazioni delle query in un database.

In sostanza, invece di eseguire una query complessa ogni volta che viene richiesta, viene creata una vista materializzata pre-calcolata che viene aggiornata solo quando i dati sottostanti vengono modificati.

In questo modo, le query successive possono accedere alla vista materializzata invece che hai dati di origine, riducendo il tempo di esecuzione e migliorando le prestazioni complessive del database.

Potrebbero esserci dei classici problemi di riferimenti.

Le Materialized View contengono solo i dati necessari per le applicazioni.



Materialized View può essere combinato con Event Sourcing, si va a recuperare una vista materializzata di una catena di eventi per facilitare le operazioni da compiere. Le view devono essere utilizzate solo per un numero limitato di query, altrimenti si tornerebbe al database originale.

Bisogna tenere traccia degli aggiornamenti, se il database originale si aggiorna si deve aggiornare anche la Materialized View. Si rinuncia un po' alla consistenza perché l'aggiornamento non sempre avviene istantaneamente.

Bisogna anche stabilire dove salvare queste view, magari vicino agli utilizzatori, indicizzando correttamente i dati.

In generale, questo pattern è molto utile quando è difficile fare query sui dati. Garantisce la partiton tollerant e aiuta nella gestione della sicurezza e privacy, con la costante presenza del mostro a nove teste GDPR da considerare. Le modifiche si fanno nelle materialized view dei dati, dove si fanno le query e non nel database originale sennò subentrano problemi. Di solito si adattano le Materialized View alle normative vigenti nello Stato in cui si trovano.

Questo pattern è raramente sconsigliato e solo nei casi in cui praticamente è impossibile rinunciare alla consistenza anche solo per un breve intervallo, o quando le query sono molto facili, o quando i dati cambiano troppo velocemente e aggiornare sempre le Materialized View diventa complesso e oneroso.

Sharding: pattern utilizzato per distribuire i dati di un database su più server o nodi in modo da migliorare le prestazioni e scalare il sistema.

In sostanza, invece di conservare tutti i dati in un singolo server, vengono suddivisi in più "shard" e archiviati su più database in nodi diversi.

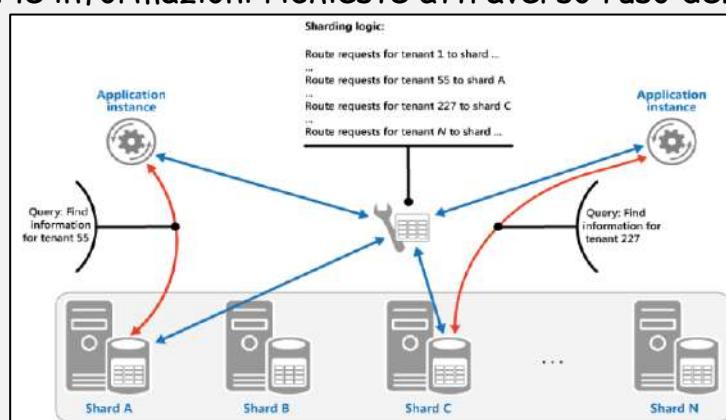
Questo consente al sistema di gestire un volume maggiore di dati e di elaborare le query in modo più efficiente, in quanto ogni nodo gestisce una parte dei dati e le query possono essere distribuite tra i database dei vari nodi.

Inoltre, il design pattern Sharding attraverso la ridondanza dei dati, aumenta la disponibilità del sistema in caso di guasti hardware o problemi di rete.

In generale, per utilizzare i vari shard si utilizza una shard key.

Principalmente, sono tre le strategie che vengono utilizzate per decidere come distribuire i dati nelle varie partizioni (shard).

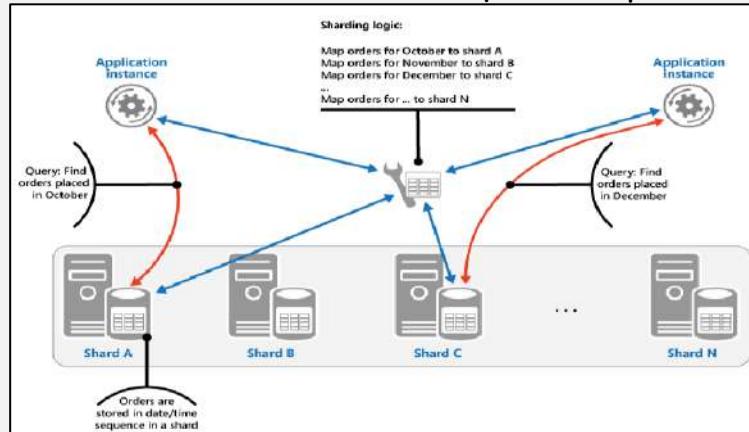
Strategia Lookup: si implementa una mappa che reindirizza le richieste agli shard contenenti le informazioni richieste attraverso l'uso della shard key.



Ovviamente, un singolo server può ospitare più shard.

C'è molto controllo sulla gestione della configurazione degli shard.

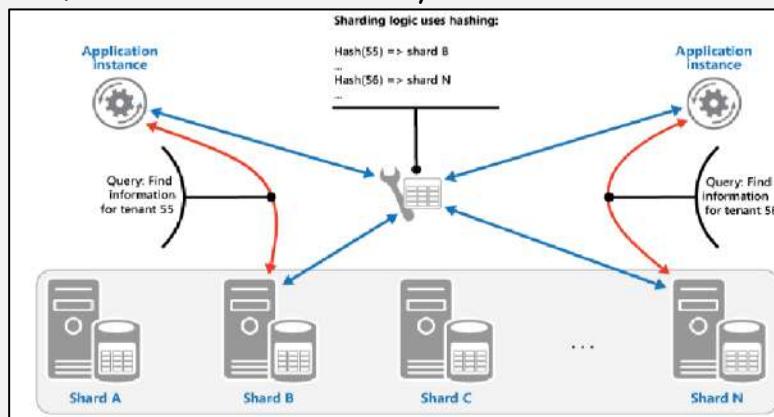
Strategia Range: utilizza un approccio sequenziale dove i dati simili o legati tra loro vengono mappati nello stesso shard. Ad esempio, lo shard A contiene tutti i dati del mese di ottobre, lo shard B contiene quelli di aprile, etc.



Facile da implementare, ma si hanno problemi di bilanciamento del carico.

Strategia Hash: utilizza un approccio che distribuisce i dati tra gli shard in modo da raggiungere un equilibrio a livello di dimensioni di ogni shard e di carico medio che ogni shard affronta. Per decidere in quale shard archiviare un dato si calcola una funzione in base all'hash di uno o più attributi del dato stesso.

La funzione di hash prende in input un dato di lunghezza arbitraria (es. una stringa o un file) e lo mappa in una sequenza di bit di lunghezza fissa (es. 128/256 bit), come se fosse l'impronta digitale del dato. Il risultato viene utilizzato per determinare a quale nodo del sistema dev'essere assegnato il dato. Se, ad esempio, si hanno 8 shard allora il range viene diviso in 8 parti e ognuno di essi fa riferimento a uno shard, rendendo la distribuzione equa.



Questa tecnica è molto vantaggiosa in termini di carico distribuito.

Infatti, a differenza della strategia Range, dove i dati vengono archiviati a blocchi nei vari shard, rischiando un sovraccarico su alcuni e nulla su altri in base all'attività della fonte di dati gestita (es. un tenant), nella strategia Hash

una fonte di dati viene assegnata ad uno shard sulla base di una funzione di hash basata magari sull'ID, rendendo probabile che due ID di due fonti contigue vengano assegnati a shard diversi, distribuendo il carico tra essi. Lo svantaggio di questa tecnica è che, una volta eseguita, rende l'ambiente quasi cristallizzato. È difficile, infatti, reindirizzare tutto.

Static Content Hosting: pattern utilizzato per ospitare e distribuire contenuti statici come immagini, file audio, pagine HTML, script JavaScript attraverso un server web. In sostanza, invece di memorizzare i contenuti su un back-end dinamico, i contenuti statici vengono memorizzati in una posizione di archiviazione a basso costo come, ad esempio, Amazon S3 e consegnati direttamente agli utenti. Questo consente di ridurre i costi di gestione e migliorare la velocità di caricamento dei contenuti per gli utenti.

Valet Key: i programmi client e i browser web spesso devono leggere o scrivere file o flussi di dati da e verso l'archiviazione di un'applicazione.

In generale, l'applicazione gestisce lo spostamento dei dati, recuperandoli dallo storage e trasmettendoli al client oppure leggendo il flusso caricato dal client e archiviandolo nell'archivio dati. Tuttavia, quest'approccio assorbe risorse preziose come elaborazione, memoria e larghezza di banda.

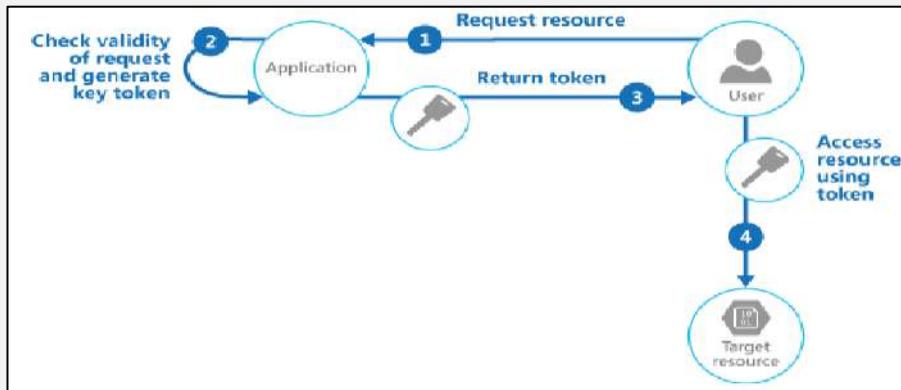
Il design pattern Valet Key utilizza un token che fornisce ai client un accesso diretto, ma limitato, a una risorsa specifica, al fine di alleggerire il trasferimento dei dati all'applicazione. Ciò è particolarmente utile nelle applicazioni che utilizzano sistemi o code di archiviazione ospitati in cloud per ridurre al minimo i costi e massimizzare la scalabilità e le prestazioni.

I token sono stringhe di bit che rappresentano l'accessibilità dell'utente ai vari contenuti e servizi della rete cloud, rendendo la rete sicura e scalabile.

Ci sono tutta una serie di autorizzazioni tra i servizi, quando collaborano.

Chiaramente tutto deve andare molto velocemente e finché i token sono validi ci si autentica con essi. In pratica, questo significa che l'applicazione fornisce una chiave "valet" agli utenti permettendo loro di accedere solo a una parte limitata delle funzionalità dell'applicazione e dei dati sensibili.

In questo modo, l'applicazione può limitare l'accesso ai dati sensibili e proteggere la privacy degli utenti. Le applicazioni devono essere in grado di controllare in modo sicuro e granulare l'accesso ai dati ma al tempo stesso ridurre il carico sul server impostando questa connessione e quindi consentendo ai client di comunicare direttamente con il database per eseguire le operazioni di lettura o scrittura di cui necessitano.



Il design pattern Valet Key è spesso utilizzato in applicazioni web o mobile che devono fornire l'accesso ai dati degli utenti a terze parti, come ad esempio servizi di pagamento o di analisi dei dati, ma al tempo stesso devono garantire la sicurezza e la privacy dei dati degli utenti. In sintesi, l'utente fa richiesta per una risorsa, l'applicazione controlla la validità di tale richiesta e in caso di esito positivo genera e consegna al client un key token. Il client, una volta in possesso del token, accede direttamente alla risorsa desiderata senza passare dall'applicazione ogni volta, alleggerendo il tutto. Non c'è rischio di accedere ad altro perché la chiave fa solo quello.

Cloud Design Patterns

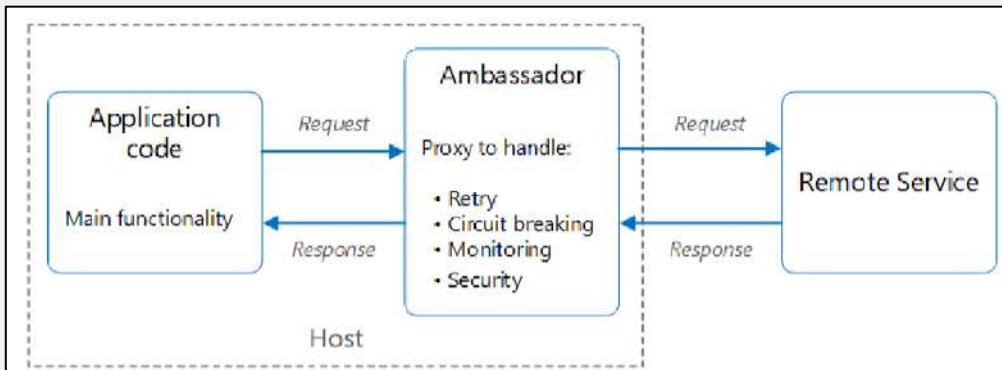
- Design and implementation patterns

Pattern	Summary
Ambassador	Create helper services that send network requests on behalf of a consumer service or application.
Gateway Aggregation	Use a gateway to aggregate multiple individual requests into a single request.
Gateway Routing	Route requests to multiple services using a single endpoint.

Pattern per la gestione dell'architettura di un sistema in una Cloud

Ambassador: pattern che consente di separare la logica di accesso ai servizi remoti dalla logica di business dell'applicazione, semplificando la gestione della comunicazione tra i componenti dell'applicazione.

Può essere visto come una specie di proxy che gestisce tutti gli aspetti tecnici della comunicazione con i servizi remoti, come l'apertura e la chiusura delle connessioni, la gestione degli errori di rete e il caching delle risposte.



Principalmente gestisce la comunicazione verso l'esterno ma l'Ambassador si occupa anche di altro. Può essere implementato in modo diverso dall'applicazione principale e anche programmato con un linguaggio diverso. Di solito, è preferibile sviluppare l'Ambassador nello stesso ambiente dell'host.

È proprio il concetto visto nei Pod, è un container di supporto istanziato accanto al container principale, all'interno dello stesso Pod.

L'utilizzo di questo pattern, proprio per via della sua natura da proxy, va ad aumentare, in generale, la latenza.

L'Ambassador è in grado di gestire operazioni ripetute ma che è sconsigliato fare salvo che le operazioni da riprocessare non siano idempotenti (evento la cui ripetizione non cambia il risultato), in tal caso possiamo permettercelo.

Durante la fase di sviluppo, bisogna considerare se creare un'istanza di Ambassador per ogni client o se invece utilizzarne una per gestirne molti.

Questo pattern è consigliato quando si necessita di creare un set comune di funzionalità per la connettività per più utenti, linguaggi, framework, ecc.

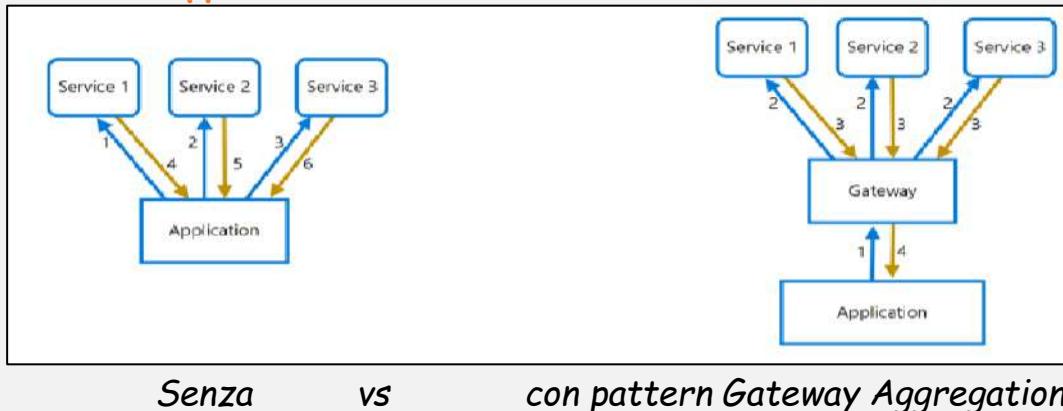
Potrebbe non essere adatto se la latenza è un aspetto critico, se i client si connettono al servizio sempre attraverso lo stesso modo o se gli aspetti di connettività non possono essere generalizzati per tutti i client.

Gateway Aggregation: all'interno di una Cloud i servizi sono implementati come microservizi. Essi permettono un facile aggiornamento, una facile gestione, ecc. Non sono comunque esenti da problemi. Uno dei principali è che, essendo dei microservizi, richiedono molte più comunicazioni rispetto al servizio stesso realizzato come un unicum. C'è bisogno quindi di un coordinamento.

Il pattern Gateway Aggregation si occupa di gestire e orchestrare le varie richieste dei client verso i vari microservizi del servizio.

Il Gateway Aggregation riceve le richieste dei client, le invia ai vari sistemi di back-end interessati; quindi, aggrega i risultati e li manda al client richiedente.

Il principale beneficio di questo pattern è che riduce il numero di richieste effettuate dall'applicazione ai servizi back-end, andando a migliorare le prestazioni dell'applicazione su reti ad alta latenza.



La richiesta è di tipo aggregato, nel **Gateway Aggregation** è presente solo la logica dei vari servizi. Il pattern, in pratica, svolge il lavoro sporco.

L'applicazione in questo modo si concentra solo su sé stessa.

Kubernetes si adatta molto bene a questa logica di funzionamento.

Anche qui ci sono una serie di considerazioni e problemi da tenere a mente.

Innanzitutto, per svolgere un lavoro del genere, il **Gateway Aggregation** dev'essere dotato di abbastanza risorse di RAM, Storage e Core.

Altrimenti, si rischia di avere problemi di Single-Point-Of-Failure.

Bisogna considerare che occorre un accoppiamento tra le parti. Tuttavia, non dev'essere eccessivo, devono essere poco dipendenti tra loro altrimenti subentrerebbero una serie di problematiche in caso di malfunzionamenti.

Inoltre, se possibile, il **Gateway Aggregation** dev'essere vicino al back-end.

In ultimo, ma non per importanza, se un servizio non riesce a rispondere subito il **Gateway Aggregation** dev'essere in grado di rispondere comunque al client, magari con una versione vecchia di informazioni che ha a disposizione.

Questo pattern è utile quando un client deve comunicare con più servizi di back-end per eseguire un'operazione o quando si utilizzano reti con latenza significativa, come le reti cellulari, etc.

Questo pattern potrebbe non essere adatto quando magari si desiderano ridurre il numero di chiamate tra un client e un singolo servizio o quando il client o l'applicazione si trova vicino ai servizi di back-end e la latenza non è un fattore significativo che viene preso in considerazione.

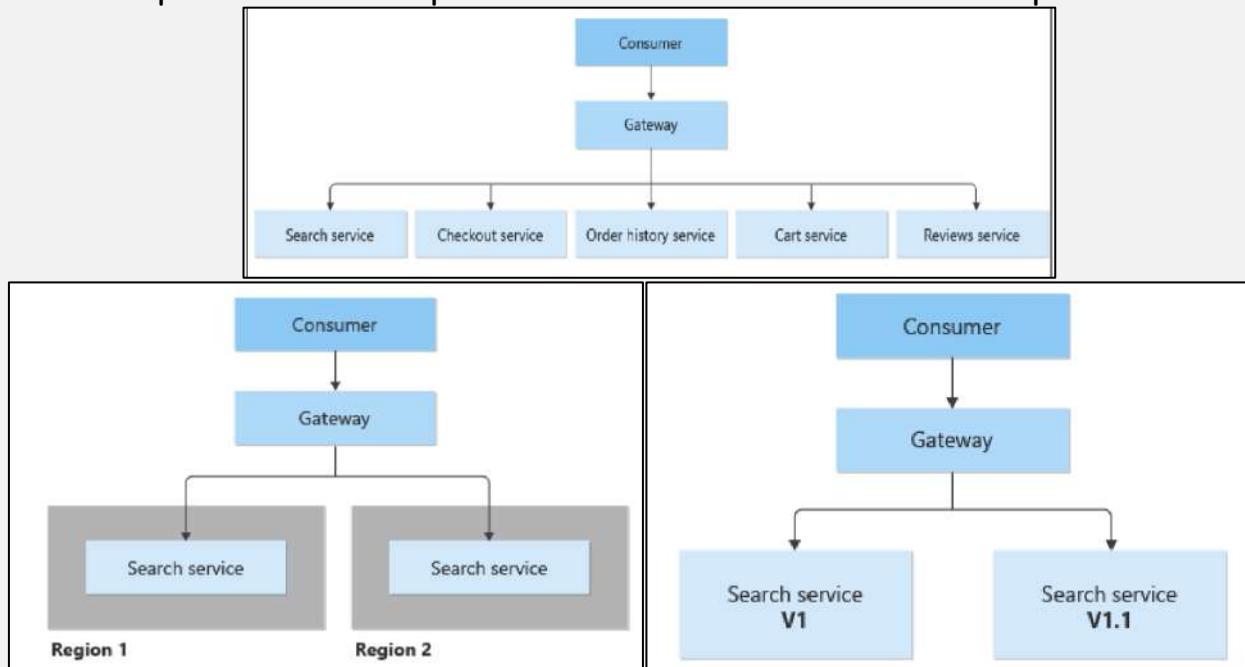
Gateway Routing: pattern che ha il compito di instradare le richieste a più servizi o a più istanze di un servizio utilizzando un singolo endpoint.

Proprio come fa Ingress di Kubernetes.

Utile quando si vogliono esporre più servizi con un singolo endpoint per poi instradare la richiesta al servizio corretto.

Utile quando si vogliono esporre più istanze dello stesso servizio su un unico endpoint per favorire il bilanciamento del carico e la disponibilità.

Utile quando si vogliono esporre diverse versioni di uno stesso servizio su un unico endpoint e inoltrare poi la richiesta ad una delle versioni specifiche.



Come in ogni caso analogo, affidare tutto ad un gateway significa andare incontro a un possibile Single-Point-Of-Failure, o rischiare di avere un collo di bottiglia in caso di un eccessivo aumento delle richieste.

Questo pattern opera al livello 7 (applicativo) della pila protocollo, può essere regionale o globale. Si può considerare la possibilità di limitare l'accesso alla rete pubblica ai servizi back-end, rendendo i servizi accessibili solo tramite il Gateway Routing o tramite una rete virtuale privata.

Questo pattern è utile quando un client deve utilizzare più servizi a cui è possibile accedere attraverso un gateway o quando si desidera magari semplificare le applicazioni client utilizzando un singolo endpoint.

Quando è necessario instradare le richieste da endpoint indirizzabili esternamente ad endpoint virtuali interni, ad esempio esponendo delle porte su una macchina virtuale verso indirizzi IP virtuali del cluster, ecc.

Cloud Design Patterns

Reliability

Pattern	Summary
Deployment Stamps	Deploy multiple independent copies of application components, including data stores.
Geodes	Deploy backend services into a set of geographical nodes, each of which can service any client request in any region.
Health Endpoint Monitoring	Implement functional checks in an application that external tools can access through exposed endpoints at regular intervals.
Queue-Based Load Leveling	Use a queue that acts as a buffer between a task and a service that it invokes, to smooth intermittent heavy loads.
Throttling	Control the consumption of resources by an instance of an application, an individual tenant, or an entire service.

Pattern per la gestione dell'affidabilità di un sistema in una Cloud

Deployment Stamps: repliche a nastro indipendenti dei componenti applicativi.

Geodes: repliche dei nodi (back-end) in diverse aree geografiche.

Health Endpoint Monitoring: endpoint che vengono monitorati costantemente ad intervalli regolari.

Queue-Based Load Leveling: utilizzo di code come un buffer tra i task e i servizi per andare a smussare il carico di lavoro dei servizi.

Throttling: strozzatura di un'istanza di un'applicazione, di un tenant o di un intero servizio che consuma troppe risorse.

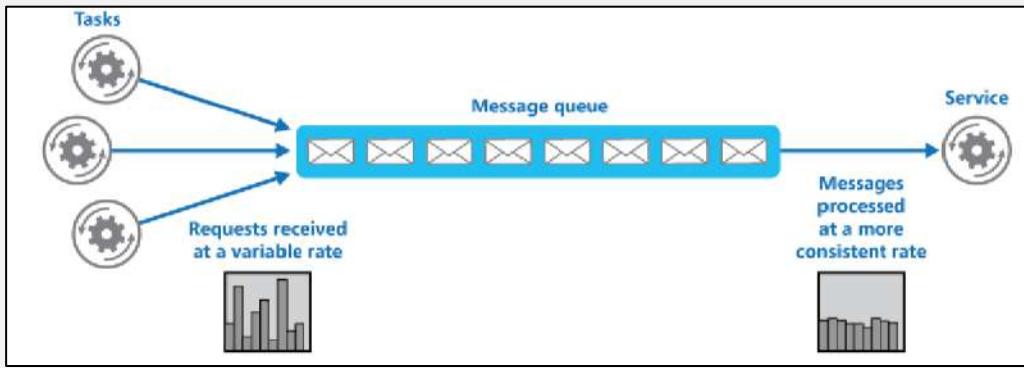
Il quarto e il quinto design pattern possono essere combinati insieme.
Analizzeremo quest'ultimi.

Queue-Based Load Leveling: utilizza la teoria delle code.

Le code fungono da "ammortizzatori". Ovvero, il server risponde normalmente alle richieste fintato che riesce. Dopodiché si inseriscono le richieste in una coda per evitare il crash del servizio o di dover scartare le richieste dei client.

Ovviamente, la speranza è che il server smaltisca la coda in tempi accettabili.

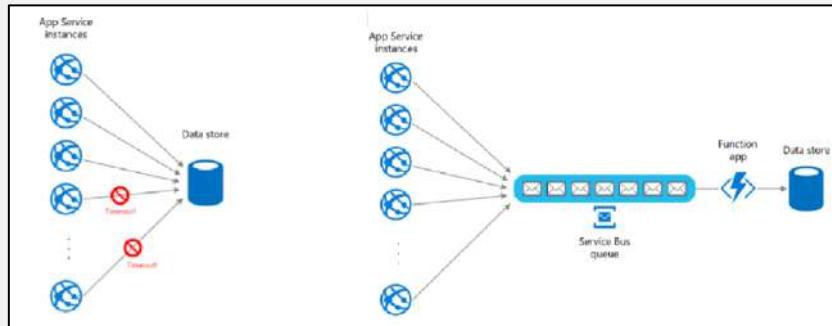
La coda va ad assorbire dei picchi di traffico, seppur in maniera limitata. Questo per evitare di riempire anche la coda. Se si arriva a quel punto, allora qualcosa va scartato e perso per sempre, necessariamente.



Esempio: **Apache Kafka** è una piattaforma open source di stream processing. Il progetto mira a creare una piattaforma a bassa latenza ed alta velocità per la gestione di feed dati in tempo reale.

La coda va a disaccoppiare i task dal servizio, il quale potrà gestire le richieste con le sue tempistiche. Questo va ad aumentare il tempo di disponibilità del servizio, in quanto, anche nel caso di disservizi per brevi intervalli, il client può comunque continuare ad inviare in coda le sue richieste. Se ci sono molte richieste esse vengono diluite, a scapito del tempo di risposta. Aumentando il numero di code a disposizione si va ad aumentare la scalabilità. Il comportamento della coda dev'essere oggetto di analisi in base alla soluzione che si sta cercando di implementare (possibile perdita di informazioni, comunicazione unidirezionale, autoscaling, etc.).

Questo pattern è utile per tutte le applicazioni che utilizzano dei servizi soggetti a sovraccarico. **Meno utile se** l'applicazione prevede una risposta da parte del servizio con una latenza trascurabile.



Esempio di più istanze di una web app che fanno richieste contemporanee a un database esterno. Si rischia di dover troncare qualche richiesta, con la coda no.

Throttling: pattern che prevede di scartare le richieste di servizio.

Quello che fa è strozzare le richieste in alcune situazioni (non belle, applicabile solo in casi particolari). **Essenzialmente, i picchi di traffico, invece di essere gestiti, si vanno a strozzare.** A volte occorre mantenere la qualità del servizio piuttosto che soddisfare tutte le richieste. Il pattern Throttling **si utilizza per monitorare l'uso delle risorse da parte di client, applicazioni o tenant e valutare la loro terminazione in caso di un utilizzo eccessivo.**

Questo pattern può essere utilizzato in alternativa o con l'autoscaling.

Il sistema potrebbe implementare **diverse strategie di limitazione**, come tagliare fuori utenti che effettuano troppi accessi in un tot di tempo o rinunciare a delle funzionalità di servizio non fondamentali.

Ad esempio, per un servizio di streaming è meglio rinunciare all'alta definizione per qualche minuto piuttosto che interrompere lo streaming.

Quando un *Gateway Routing* regola gli accessi potrebbe dover attendere diverse componenti di servizio erogate da altri; in tal caso, possono essere tagliati se tardano a rispondere evitando di riempire il log di errori.

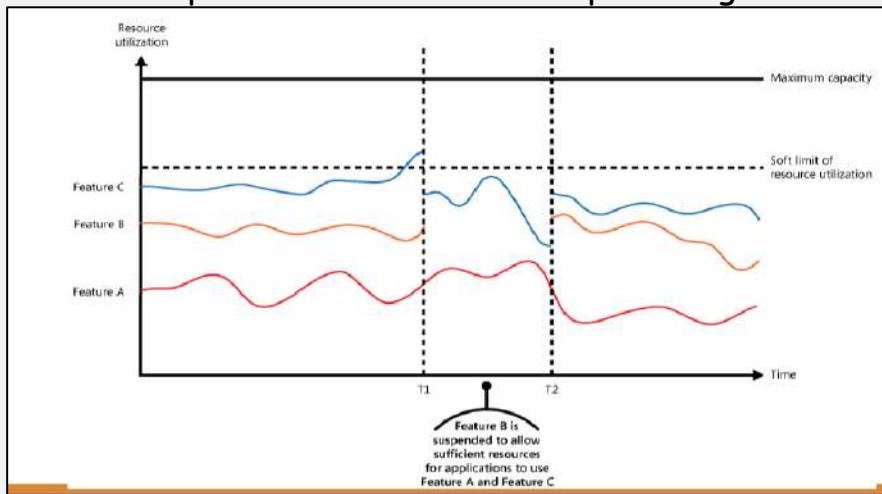


Illustrazione del concetto di throttling

L'ascissa (asse x) rappresenta il tempo.

L'ordinata (asse y) rappresenta l'utilizzo complessivo delle risorse (generico).

L'occupazione delle risorse da parte di una features è l'area sottesa dalla curva.

C'è un **limite soft** (linea tratteggiata) e un **limite hard** (linea continua), proprio come in Kubernetes. Nel caso in cui una feature eccede troppo nell'uso delle risorse, è possibile tagliarla momentaneamente fuori dal sistema.

Appena prima del tempo T1, le risorse totali assegnate a tutte le applicazioni che utilizzano queste funzionalità raggiungono il limite soft. A questo punto, le applicazioni rischiano di esaurire le risorse disponibili. In questo sistema d'esempio, la feature B è meno critica delle feature A e C, quindi viene temporaneamente disabilitata e le risorse che utilizzava vengono rilasciate.

Nell'intervallo [T1, T2] le applicazioni che utilizzano la feature A e la feature C continuano a funzionare normalmente. Alla fine, l'utilizzo delle risorse di queste due feature diminuisce al punto che, all'istante T2, c'è capacità sufficiente per abilitare nuovamente la feature B.

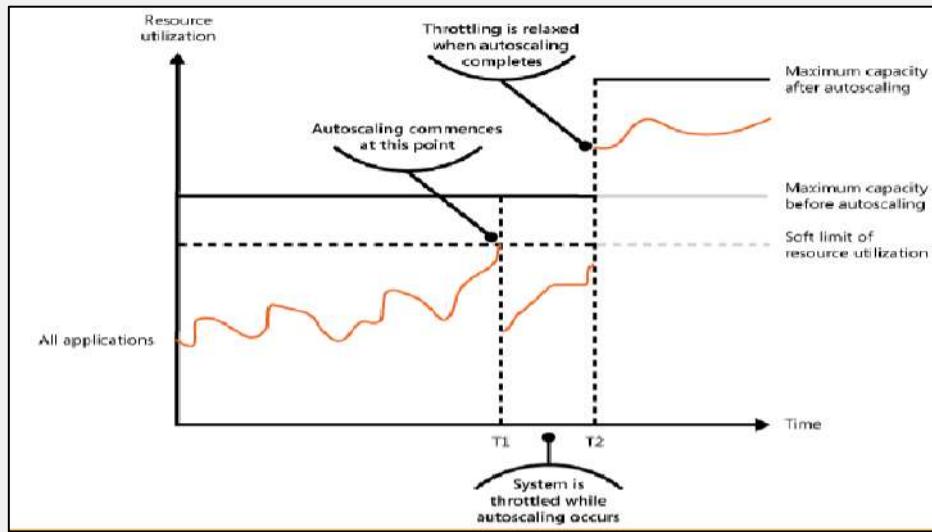
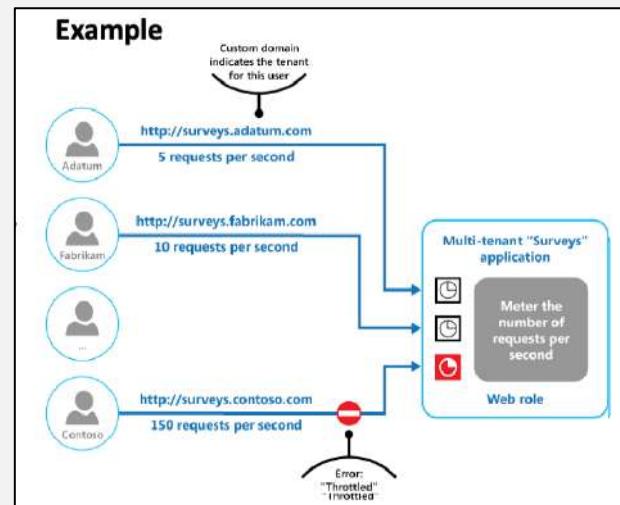


Illustrazione del concetto di throttling combinato con l'autoscaler

C'è un cluster che fa girare i suoi servizi e la curva rossa indica l'occupazione delle risorse. Raggiunto il limite soft al tempo T1, si utilizza un autoscaler. Durante l'intervallo di scaling, si implementa anche un Throttler per strozzare il servizio. Al tempo T2, il Throttler viene rimosso in quanto sono stati aumentati i limiti soft e hard perché l'infrastruttura adesso è scalata verso l'alto e dispone di più risorse di calcolo.

Implementare una strategia di questo tipo richiede un continuo monitoraggio delle risorse in uso, inoltre il sistema dev'essere in grado di cambiare stato velocemente, aumentando o diminuendo la gestione delle richieste.

Questo pattern è tuttavia molto utile nel caso in cui si vogliono mantenere i costi del servizio offerto a un certo livello (se si strozzano e non si aumentano le risorse non c'è un aumento dei costi), se si vogliono mantenere qualità fisse del servizio, se si vuole evitare che un utente mangi tutte le risorse e per gestire picchi di traffico.



Cloud Design Patterns

High Availability

Pattern	Summary
Deployment Stamps	Deploy multiple independent copies of application components, including data stores.
Geodes	Deploy backend services into a set of geographical nodes, each of which can service any client request in any region.
Health Endpoint Monitoring	Implement functional checks in an application that external tools can access through exposed endpoints at regular intervals.
Bulkhead	Isolate elements of an application into pools so that if one fails, the others will continue to function.
Circuit Breaker	Handle faults that might take a variable amount of time to fix when connecting to a remote service or resource.

Pattern per la gestione di un'elevata disponibilità di un sistema in una Cloud

Tendenzialmente, sono pattern che applicano delle ridondanze.

Lo scopo è quello di evitare di far morire i servizi.

Vediamo nel dettaglio gli ultimi due.

Bulkhead: il concetto è quello di prendere il deployment e **fare in modo che, in caso di problemi, non si abbiano ripercussioni su tutto il servizio.**

Un po' come negli scafi delle navi, dove, se entra dell'acqua, si va ad isolare solo lo scafo problematico, lasciando il resto intatto e garantendo la nave a galla.

Il pattern Bulkhead offre proprio questa tolleranza alle partizioni.

Un'applicazione basata su cloud può includere più servizi, con ogni servizio che ha uno o più consumatori. Un carico eccessivo o un errore in un servizio potrebbe avere un impatto su tutti i consumatori del servizio.

Inoltre, un consumatore potrebbe inviare richieste a più servizi contemporaneamente, utilizzando risorse per ogni richiesta.

Quando il consumatore invia una richiesta a un servizio configurato in modo errato o che non risponde, le risorse utilizzate dalla richiesta del client potrebbero non essere liberate in modo tempestivo. Man mano che le richieste al servizio continuano, tali risorse potrebbero esaurirsi. Ad esempio, il pool di connessioni del client potrebbe esaurirsi. A quel punto, anche le richieste del consumatore ad altri servizi sono "infettate". Alla fine, il client non può più inviare richieste a nessun servizio, non solo al servizio originale che non risponde. Il pattern **punta ad isolare i servizi e i consumatori in modo da evitare fallimenti in cascata. Si vanno a preservare le funzionalità principali dei servizi.**

Garantisce agli utenti una qualità differenziata in base all'applicazione.

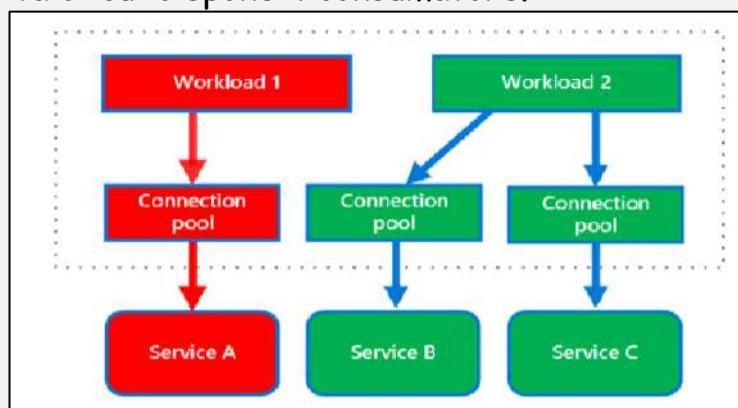
Ad esempio, per una partita in streaming si accetta una latenza di qualche secondo, l'importante è che si veda e non si blocchi.

Per un'operazione di home-banking, invece, la probabilità di errore dev'essere zero. In questo caso si accetta anche una connessione lenta, a patto che l'operazione vada a buon fine, si sta parlando di soldi.

Il pattern Bulkhead va ad eseguire un **partizionamento delle istanze dei servizi** in differenti gruppi, in base al carico di richieste e ai requisiti di disponibilità.

In questo modo si predispongono i fallimenti isolati, permettendo di continuare ad offrire le funzionalità dei servizi ad altri utenti.

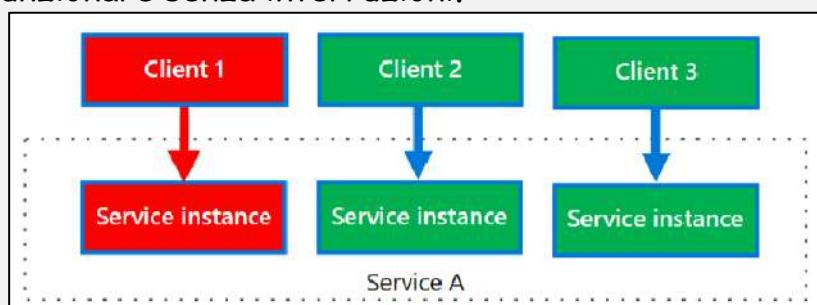
Inoltre, è possibile sviluppare servizi che offrono differenti qualità di servizio in base alla priorità di cui dispone il consumatore.



Esempio in cui si separano le connection pool ai vari servizi

Una volta che una connessione è stata creata, viene messa nella connection pool e viene riutilizzata più volte in modo tale da non dover ricreare una connessione ad ogni richiesta di dati. Si evita così la latenza dovuta alla connessione e si accede direttamente al servizio. Se il servizio A fallisce o causa altri problemi, il pool di connessione è isolato. Quindi sono interessati solo i carichi di lavoro che utilizzano il pool di thread assegnato al servizio A.

I carichi di lavoro che utilizzano i servizi B e C non sono interessati e possono continuare a funzionare senza interruzioni.



Esempio in cui si separano le istanze di un servizio

Il client 1 ha effettuato troppe richieste e ha sovraccaricato la sua istanza.

Poiché ogni istanza del servizio è isolata dalle altre, gli altri client possono continuare a effettuare richieste normalmente.

Quest'approccio è sconveniente in termini di risorse, ma vantaggioso in termini di deployment.

Occorre definire le partizioni in base ai requisiti aziendali e tecnici dell'applicazione. Si può valutare la sua combinazione con altri design pattern.

Bisogna capire come fare Bulkhead in base a cosa si sta utilizzando.

I servizi che comunicano tramite messaggi asincroni possono essere isolati tramite diversi set di code. Quando si partizionano i servizi in Bulkhead (paratie), prendere in considerazione la possibilità di distribuirli in macchine virtuali, container o processi separati. I container offrono un buon equilibrio tra l'isolamento delle risorse e un sovraccarico piuttosto basso.

In generale, è sempre conveniente utilizzare questo pattern.

Bulkhead pattern

Example

The following Kubernetes configuration file creates an isolated container to run a single service, with its own CPU and memory resources and limits.

```
apiVersion: v1
kind: Pod
metadata:
  name: drone-management
spec:
  containers:
    - name: drone-management-container
      image: drone-service
      resources:
        requests:
          memory: "64Mi"
          cpu: "250m"
        limits:
          memory: "128Mi"
          cpu: "1"
```

Circuit Breaker: pattern che ha lo scopo di "interrompere il circuito".

Si pensi ad un servizio composito, magari mediato da un orchestratore.

Una delle componenti ha un problema, cosa si fa?

Potrebbe valere la pena riprovare e attendere, a volte, infatti, i problemi sono temporanei. Tuttavia, non si può attendere eccessivamente.

Nel caso in cui un servizio impieghi troppo tempo a rispondere, si termina.

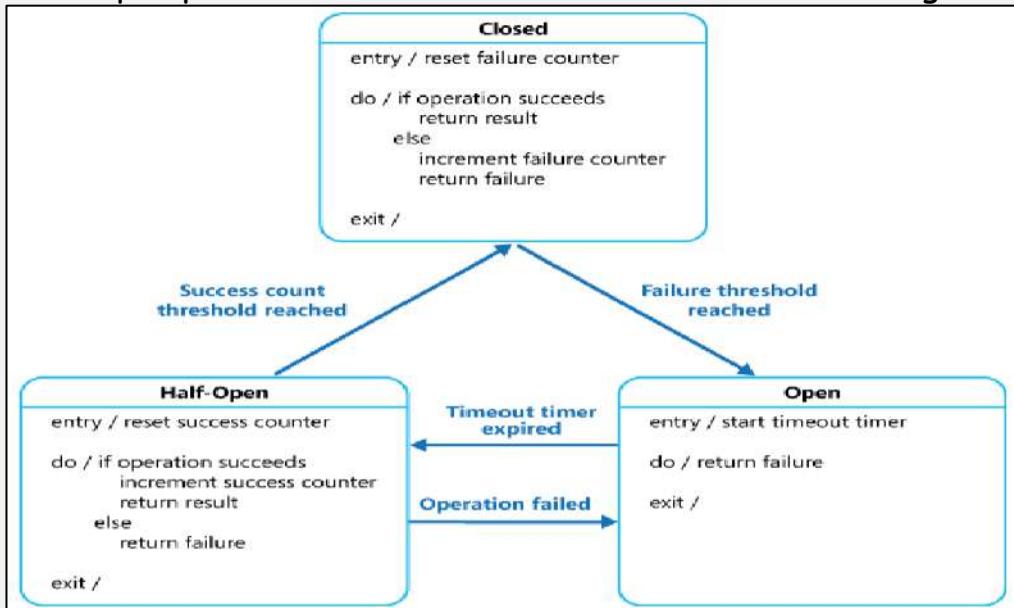
Il pattern Circuit Breaker **impedisce ad un'applicazione di tentare ripetutamente di eseguire un'operazione che potrebbe non riuscire, permettendogli di continuare senza attendere la correzione dell'errore o sprecare cicli della CPU mentre determina che l'errore è di lunga durata**.

Il pattern Circuit Breaker consente, inoltre, ad un'applicazione di rilevare se l'errore del servizio è stato risolto. Se il problema sembra essere stato risolto, l'applicazione può tentare di rieseguire la richiesta.

Può essere visto come un proxy che esegue operazioni potenzialmente fallimentari al posto dell'applicazione.

Per farlo, **va a monitorare il numero di errori recenti** che si sono verificati e utilizza queste informazioni per decidere se consentire all'operazione di procedere o se semplicemente restituire immediatamente un'eccezione.

Il pattern Circuit Breaker si implementa mediante una macchina a stati finiti, per capire quando interrompere il servizio (analoga con un circuito elettrico che si interrompe aprendo l'interruttore o di un filo che si va a tagliare).



Nello specifico, prevede l'implementazione di tre stati: IF, THEN, ELSE. Sono tre stati che vanno a modellare le possibili situazioni.

Lo stato Closed: in questo stato, la richiesta dall'applicazione viene instradata correttamente al servizio (il circuito è chiuso, quindi si scorre dentro).

Il proxy mantiene un conteggio del numero di errori recenti e, se la chiamata all'operazione non riesce, il proxy incrementa questo conteggio.

Se il numero di errori recenti supera una soglia specificata entro un determinato periodo di tempo, il proxy viene posto nello stato Open.

Lo stato Open: in questo stato si è tagliato il filo (si solleva l'interruttore creando un'interruzione nel circuito, non si scorre più) e di conseguenza le richieste non finiscono più al servizio perché si è stimato che è meglio interrompere. Si restituisce immediatamente un'eccezione all'applicazione.

Lo stato Half-Open: in questo stato siamo in una vita di mezzo, sperando che la situazione migliori col tempo. Viene fatto passare un numero limitato di richieste dell'applicazione che possono richiamare l'operazione. Se queste richieste vanno a buon fine, si presume che l'errore che in precedenza causava il guasto sia stato riparato e di conseguenza si torna allo stato Closed (il contatore dei guasti viene azzerato). Se una qualsiasi richiesta di quelle fatte passare (esploratorici) fallisce, Circuit Breaker presuppone che il guasto sia ancora presente e di conseguenza torna allo stato Open, riavviando il timer di timeout per dare al sistema un ulteriore periodo di tempo per riprendersi dal guasto. Scaduto il timer, riprova ad inviare richieste esploratoriche.

Questo pattern sono 20 righe di codice.

Bisogna considerare le eccezioni e come esse debbano essere gestite per prendere una specifica decisione. Molte altre considerazioni da fare.

Messaging	
Pattern	Summary
Choreography	Have each component of the system participate in the decision-making process about the workflow of a business transaction, instead of relying on a central point of control.
Priority Queue	Prioritize requests sent to services so that requests with a higher priority are received and processed more quickly than those with a lower priority.
Publisher-Subscriber	Enable an application to announce events to multiple interested consumers asynchronously, without coupling the senders to the receivers.

Pattern per la gestione della messaggistica di un sistema in una Cloud

Choreography: lo scenario è quello dei microservizi.

Tante piccole componenti che operano insieme per un servizio più grande.

Chi gestisce i vari microservizi, facendoli operare al meglio tra loro?

Ci sono **due possibili approcci**.

Utilizzo di un orchestratore: **un cervello centrale** che gestisce tutta la logica. Dice ai vari microservizi cosa fare, quando aspettare per poi fare qualcosa, etc. Non è la soluzione migliore in quanto affidare tutta la logica ad una singola componente rappresenta un **Single-Point-Of-Failure** in caso di guasti o sovraccarichi.

L'altro approccio, migliore, prevede l'utilizzo del pattern **Choreography**.

Questo pattern lascia ad ogni servizio la decisione di come e quando elaborare un'operazione aziendale, invece di dipendere da un orchestratore centrale.

Un modo per implementare **Choreography** consiste nell'utilizzare il **modello di messaggistica asincrono** per coordinare le operazioni aziendali.

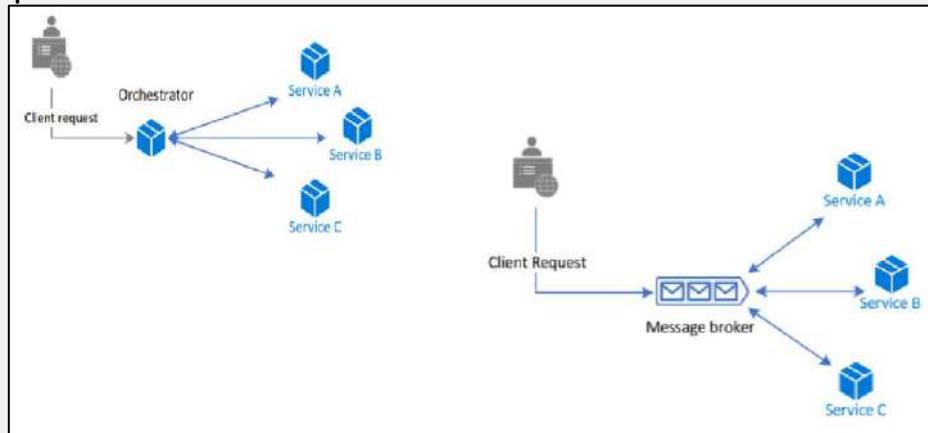
Si può quindi **distribuire l'intelligenza ad ogni microservizio**, in modo da renderli indipendenti e renderli in grado di sapere cosa fare in base alle situazioni.

Come fanno però a capire quando eseguire le operazioni?

Per fare ciò, occorre un bus distribuito, ovvero, un **broker message**, dove pubblicare i vari messaggi. Il client invia la sua richiesta al broker.

In base al topic, i microservizi leggono quello che è di loro competenza.

Dopo aver elaborato una richiesta, il servizio o esegue altre operazioni (inviando magari a sua volta dei messaggi nel broker che saranno utilizzati da altri) oppure semplicemente termina il suo lavoro.



Primo approccio vs Secondo approccio

Poiché non esiste una comunicazione point-to-point, **questo pattern consente di ridurre l'accoppiamento tra i servizi**.

Inoltre, va a rimuovere il collo di bottiglia delle prestazioni causato dall'agente di orchestrazione che deve, invece, gestire tutte le transazioni.

Questo pattern è consigliato se si prevede di aggiungere, aggiornare o rimuovere servizi frequentemente. Grazie al pattern Choreography il tutto viene fatto senza troppi problemi o disturbi verso gli altri servizi.

Questo pattern si sposa molto bene con architetture **serverless** (modello di sviluppo software in cui il fornitore del servizio cloud gestisce l'infrastruttura necessaria per eseguire e scalare l'applicazione) **dove tutti i servizi possono avere vita breve**. I servizi possono essere attivati a causa di un evento, svolgere la propria attività ed essere rimossi al termine dell'attività.

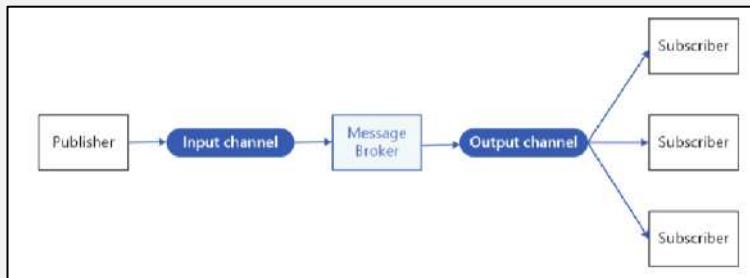
In ogni caso, **non è sempre conveniente**, la logica da applicare è complessa.

Inoltre, è difficile risolvere problemi perché occorre capire dove sono le cause di quest'ultimi. Quindi, in generale, c'è una gestione complessa.

Publisher-Subscriber: consente a un'applicazione di annunciare eventi a più consumatori interessati in modo asincrono, senza accoppiare i mittenti ai destinatari ed evitando che il mittente debba aspettare la risposta.

È un pattern utile nello scenario Cloud in quanto la generazione di eventi con conseguenti cause è sempre all'ordine del giorno.

Ad esempio, **questo pattern è utilizzato all'interno di OpenStack** in quanto incrementa la scalabilità e l'affidabilità oltre al disaccoppiamento.



Un esempio di publisher-subscriber è **Apache Kafka**.

I canali di input, di output e il broker lavorano insieme per consentire ai publisher di inviare messaggi ai subscriber in modo disaccoppiato e scalabile.

Il publisher invia i messaggi all'input channel, specificando il **topic** a cui il messaggio è destinato. Il messaggio viene quindi ricevuto dal broker attraverso l'input channel. Il broker gestisce la ricezione del messaggio, lo memorizza e lo invia a tutti i subscriber registrati a quel topic specifico. Questo processo di invio del messaggio ai subscriber avviene attraverso l'output channel.

I subscriber, d'altra parte, si registrano per un particolare topic attraverso l'input channel. Quando il broker riceve un messaggio su un topic a cui il subscriber si è registrato, il messaggio viene inviato al subscriber attraverso l'output channel. Il subscriber riceve il messaggio e lo elabora come necessario.

In questo modo, i **channel input e output consentono ai publisher e ai subscriber di interagire con il sistema in modo disaccoppiato**, senza conoscere gli altri componenti del sistema. Il broker gestisce la comunicazione tra i publisher e i subscriber, consentendo ai messaggi di essere instradati ai subscriber corretti in modo affidabile e scalabile. Ci sono delle considerazioni da fare.

Innanzitutto, è sempre meglio utilizzare tecnologie già sviluppate piuttosto che creare il proprio sistema di messaggistica. Inoltre, bisogna considerare le funzionalità di disiscrizione, la capacità di usare comunicazioni bidirezionali e, importante, bisogna tener conto che non tutti i messaggi sono uguali, **alcuni messaggi potrebbero essere più urgenti o importanti**, vanno gestiti.

Una componente che svuota il broker nel caso in cui alcuni messaggi rimangano troppo tempo all'interno del canale è necessaria.

In generale, **questo pattern conviene utilizzarlo quando** l'applicazione si gestisce molto bene con questo approccio. Quando si necessita di comunicare con più componenti. Quando l'interazione tra applicazione e utente non è real-time. Quando la consistenza non è un grosso problema.

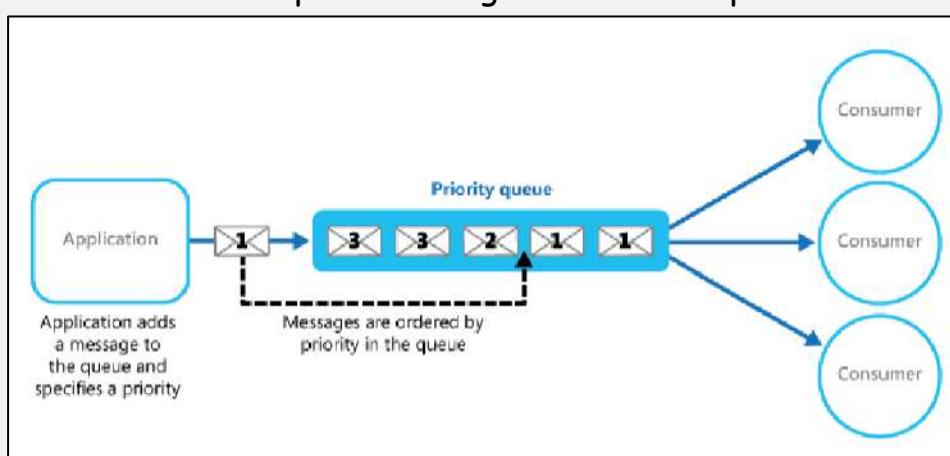
Meno conveniente quando ci sono pochi subscriber o quando la latenza richiesta è infinitesima. Ad esempio, la guida autonoma non può utilizzare questo pattern.

Priority Queue: pattern che **ha come obiettivo la gestione della priorità**.

La sua presenza è molto importante perché se ci fossero dei messaggi di un'importanza per lo sviluppo dell'applicazione maggiore di altri (ad esempio, un messaggio di reset è più importante di un messaggio di accesso), essi devono in qualche modo "scavalcare la coda".

Una coda, se non ci sono altre policy, di default è di tipo first-in first-out.

Chiaramente una politica del genere non sempre è accettabile.



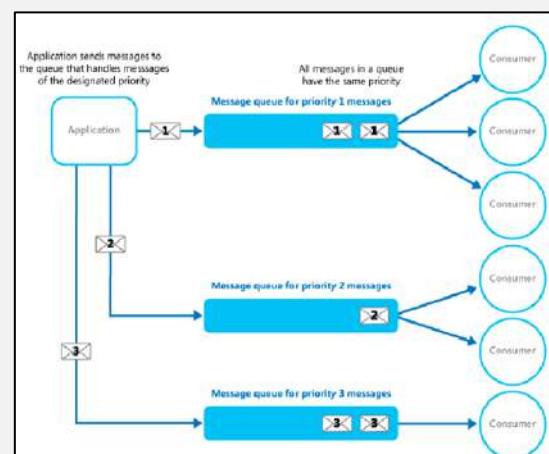
Quello che si può fare è **incrementare la priorità di alcuni messaggi**. Se ci sono messaggi con la stessa priorità allora si mettono in coda tra loro.

Nei sistemi che non supportano code di messaggi basate sulla priorità, una soluzione alternativa consiste nel mantenere una coda separata per ogni priorità. L'applicazione è responsabile dell'inserimento dei messaggi nella coda appropriata. Ogni coda può avere un pool separato di consumer. Le code con priorità più alta possono avere un pool più ampio di consumer in esecuzione su hardware più veloce rispetto alle code con priorità più bassa. La figura illustra l'utilizzo di code di messaggi separate per ciascuna priorità.

Si implementano più code di priorità in parallelo.

I messaggi vengono discriminati in base alla modalità con cui sono trattate le singole code.

Questo pattern è utile quando vogliamo discriminare gli utenti.



Quando vogliamo modellare il servizio per minimizzare i costi.

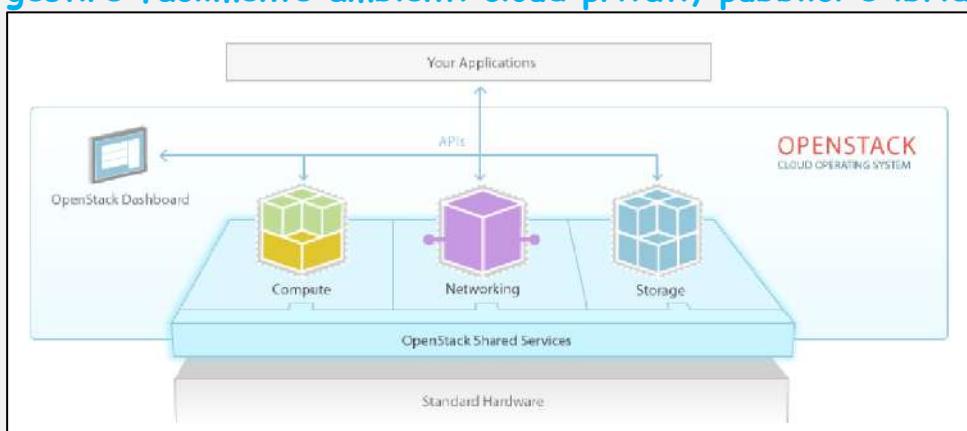
Quando si vogliono garantire prestazioni differenti in base alla priorità.

Nonostante si basi sulla priorità, bisogna comunque gestire i messaggi di priorità inferiore, che devono comunque essere processati prima o poi.

Capitolo 5: OpenStack

OpenStack è una piattaforma open-source **progettata per la creazione e la gestione di infrastrutture cloud di tipo Infrastructure as a Service**.

Consiste in un insieme di servizi interconnessi che consentono di controllare e orchestrare risorse di calcolo, archiviazione e rete su un'ampia varietà di hardware e fornitori di servizi in cloud. **L'obiettivo principale di OpenStack è fornire un'infrastruttura flessibile e scalabile che consenta agli utenti di creare e gestire facilmente ambienti cloud privati, pubblici e ibridi.**



Integra le risorse fisiche distribuite su tutto il datacenter e le fornisce in forma disaggregata agli utenti che ne fanno richiesta.

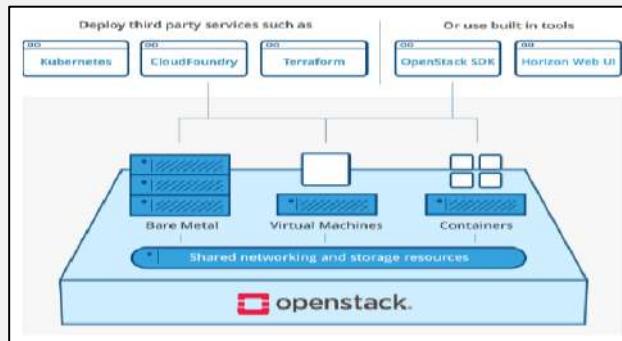
Per accedere ad OpenStack è possibile utilizzare la **Dashboard Horizon** fornita dal servizio stesso che permette di accedere a quasi tutti i servizi.

È possibile accedere anche mediante riga di comando, da terminale.

Normalmente, i gestori utilizzano la Dashboard.

Si può accedere anche indirettamente attraverso delle applicazioni.

Horizon è, comunque, lo strumento principale in quanto consente agli amministratori di gestire l'infrastruttura e ai client di accedere ai vari servizi, il tutto attraverso un'interfaccia web.

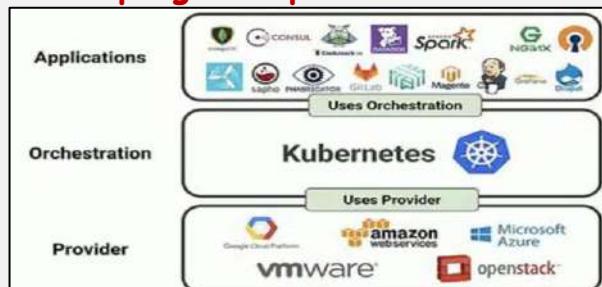


OpenStack è composto da una serie di componenti, installabili anche separatamente, che lavorano insieme per fornire ciò di cui una rete Cloud ha bisogno. Tra i componenti (servizi) principali ci sono: **Compute (Nova)**, **Identity Service (Keystone)**, **Networking (Neutron)** e **Image Service (Glance)**.

Attraverso Kubernetes è possibile utilizzare i servizi messi a disposizione da OpenStack. Inizialmente, OpenStack nasce con l'obiettivo di mettere in distribuzione servizi di tipo IaaS, ovvero **offrire un collegamento alla rete cloud mediante un'interfaccia web per gestire delle VM in modo personalizzato**.

Ad oggi, OpenStack non mette solo a disposizione VM ma anche Bare Metal (host for user) e Container. Lo scenario non è quindi banale.

Come si dispongono OpenStack e Kubernetes?

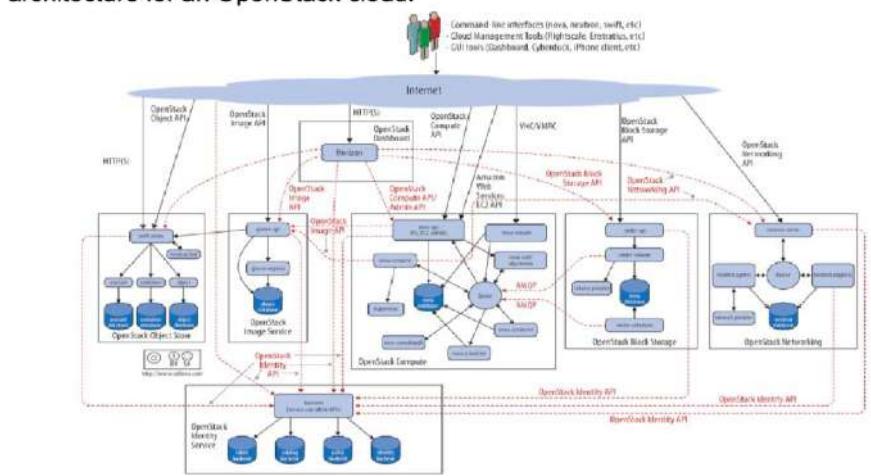


OpenStack fornisce la piattaforma su cui può girare Kubernetes per orchestrare i container contenenti le applicazioni dei client.

OpenStack si trova dunque sotto Kubernetes.

Sebbene entrambi possano essere utilizzati separatamente, insieme danno il meglio per realizzare un'infrastruttura cloud definitiva e stabile.

The following diagram shows the most common, but not the only possible, architecture for an OpenStack cloud:

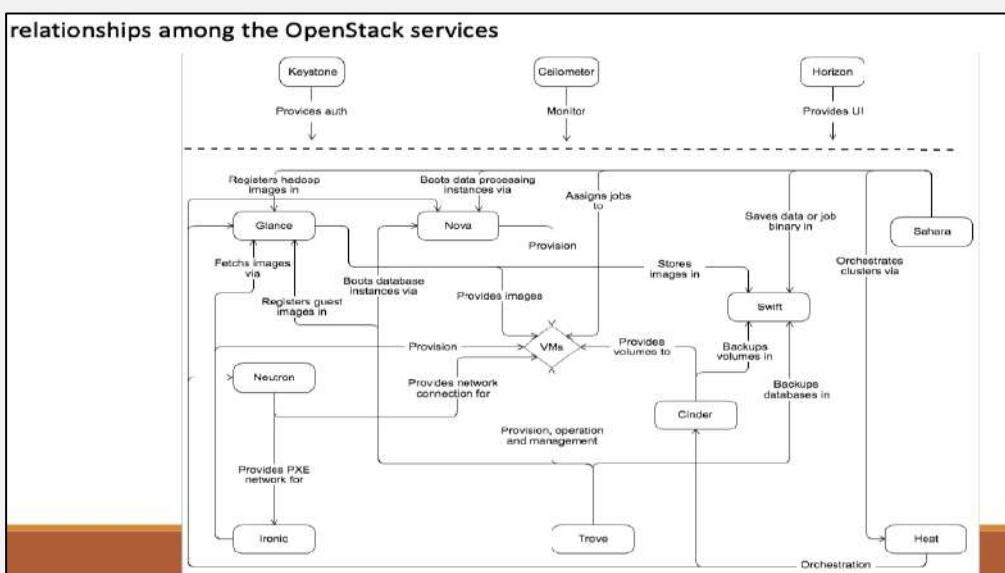


Ogni servizio ospita al suo interno almeno un **database relazionale**, il cui compito è quello di mantenere lo stato del servizio stesso.

Inoltre, ogni servizio è dotato di **un'API REST per comunicare con l'esterno** e di **un'API per comunicare all'interno** dell'infrastruttura con altri servizi.

Per lo scambio dei messaggi interni si utilizza il protocollo AMQP.

Quindi, in generale, abbiamo la **presenza di Broker e di API**.



In alto sono presenti i servizi "generali" che interagiscono con tutti gli altri. Sotto di essi vi è la logica di funzionamento principale.

Al centro ci sono le VM che vengono fornite a chi ne fa richiesta.

La componente che si occupa di fornire le VM si chiama Nova.

La componente che si occupa di fornire il networking alle VM si chiama Neutron.

La componente che orchestra i volumi, le reti e le immagini si chiama Heat.

In generale, i vari componenti sono indipendenti e prendono il nome di servizi.

Tutti i servizi effettuano l'autenticazione tramite un Identity Service comune.

Internamente, ogni servizio è composto da una serie di processi, tra i quali almeno uno dev'essere sempre attivo, ovvero quello che si occupa di ascoltare le richieste API, rielaborarle e passarle ad altri processi del servizio.

Come detto, ogni servizio interno comunica con gli altri attraverso delle API.

L'interazione interna ed esterna si basa su un approccio di messaggistica di tipo Oriented Middleware. RabbitMQ è il broker utilizzato di default.

Ogni singola componente interna di un servizio deve avere un suo processo RabbitMQ attivo, fondamentale altrimenti non è in grado di comunicare.

A volte il processo di RabbitMQ non si avvia in automatico. Occorre, dunque, verificare la sua esecuzione e, eventualmente, lanciarlo da terminale.

Si fa uso di code per evitare di dover perdere dei messaggi.

Il broker si prende la responsabilità di garantire che i messaggi arriveranno a destinazione correttamente, anche se il destinatario è temporaneamente non disponibile per un qualche motivo. Ciò rende la comunicazione affidabile.

Con questo sistema, tutti i servizi, seppur debolmente, sono accoppiati.

OpenStack: Aspetti Generali

La Triade

Dashboard / Horizon: interfaccia per accedere ai servizi interni di OpenStack, come può essere l'avvio di un'istanza o l'assegnamento di un indirizzo IP.

Compute / Nova: gestisce tutto il ciclo di vita delle VM.

Networking / Neutron: gestisce tutta la parte della connettività per i servizi di OpenStack, ad esempio fornisce la rete a OpenStack Compute.

È possibile creare delle reti virtuali che fungono da estensione della rete fisica.

Ovvero, è possibile collegare le VM alla rete provider e, se si è in grado, di farla funzionare come se fosse una connessione di strato 2. In questo modo, tutte le VM attaccate a quella LAN possono scambiarsi dati tra di loro e avere degli indirizzi IP compatibili. In questo modo, si crea una rete virtuale dentro la Cloud completamente isolata a cui è possibile collegare le VM di un tenant. In questo modo, tutte le VM di un tenant sono collegate alla rete.

Si potrebbe successivamente generare un router virtuale per farle navigare anche su internet, attraverso la realizzazione di due interfacce: una verso la rete provider interna e una verso la rete esterna.

Viene così gestito uno spazio di indirizzamento completamente separato.

Servizi di condivisione

Identity service / Keystone: fornisce un'autenticazione e un'autorizzazione a tutti i servizi di OpenStack.

Definisce dei ruoli, ogni utente ha associato un ruolo.

OpenStack garantisce l'autorizzazione e gli accessi in base ad essi.

È possibile utilizzare un approccio più flessibile rispetto ai ruoli ma più difficile da gestire. Tale approccio consiste nell'associare ogni componente a degli attributi e combinarli a delle funzioni per poi decidere cosa un utente può fare.

OpenStack di default funziona mediante l'utilizzo dei ruoli.

Image service / Glance: servizio che fornisce le immagini. Ha come obiettivo quello di gestire delle versioni di sistemi operativi rimestati per Cloud.

Una sorta di gestione di file .iso "cloudificati".

Una volta istanziata, l'immagine può essere riutilizzata, come uno stampino.

Glance le crea e le fornisce a Nova, quando occorrono, mediante un endpoint di tipo REST (le prende e le carica verso Nova).

Telemetry / Ceilometer: servizio non essenziale ma che serve per il monitoraggio delle varie componenti. Si utilizza a livello commerciale per fare, ad esempio, il logging delle risorse utilizzate da parte degli utenti.

Servizi di archiviazione

Block Storage / Cinder: servizio che mette a disposizione dei volumi, ovvero delle memorie permanenti, per ogni VM che ne ha bisogno.

Object Storage / Swift: possibile alternativa. Memorizza e recupera oggetti di dati non strutturati (immagini, audio, ...) tramite un'API REST basata su HTTP.

Servizi di livello superiore

Orchestration / Heat: servizio che fornisce un'orchestrazione.

È possibile orchestrare anche delle operazioni di **cloud bursting** per gestire i picchi delle richieste. Se una rete cloud privata di un'organizzazione raggiunge il 100% della sua capacità, in termini di risorse, allora il traffico in eccesso viene indirizzato a una cloud pubblica, senza alcuna interruzione dei servizi.

Il vantaggio principale del bursting nel cloud, oltre alla flessibilità e alla funzionalità self-service, è il risparmio economico. Si paga per le risorse aggiuntive solo quando servono. Non occorre spendere per avere una capacità aggiuntiva che non si utilizza o tentare di prevedere le variazioni e i picchi della domanda. Quindi, dentro OpenStack si realizzano anche Cloud ibride.

Esempio di una piccola cloud privata per la gestione di una piccola azienda.

C'è un **nodo Controller** che gestisce la rete, è la logica centrale, il cuore.

Le risorse come la RAM, ecc. possono aumentare in base alle esigenze.

Le interfacce fisiche, **NIC**, devono essere almeno due: una sul piano di controllo e una sul piano utente.

C'è un **nodo Compute** all'interno del quale avviene il deployment delle VM.

Al suo interno gira un hypervisor.

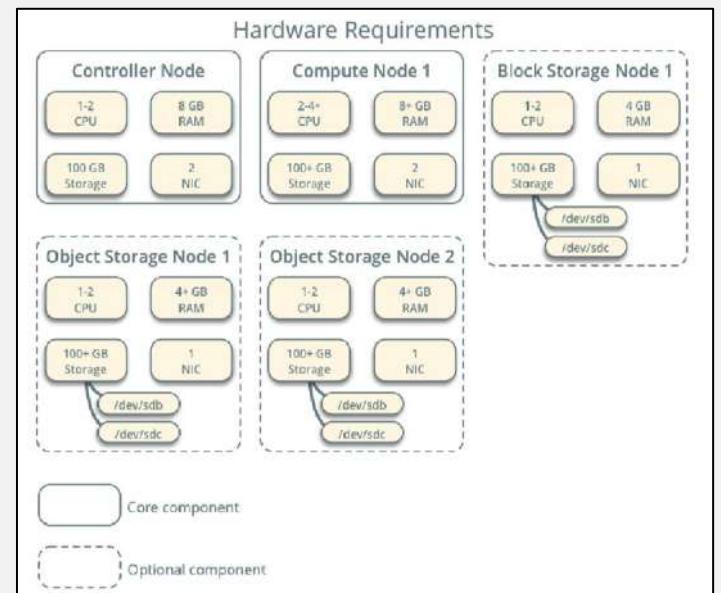
I Compute possono essere 1 o N.

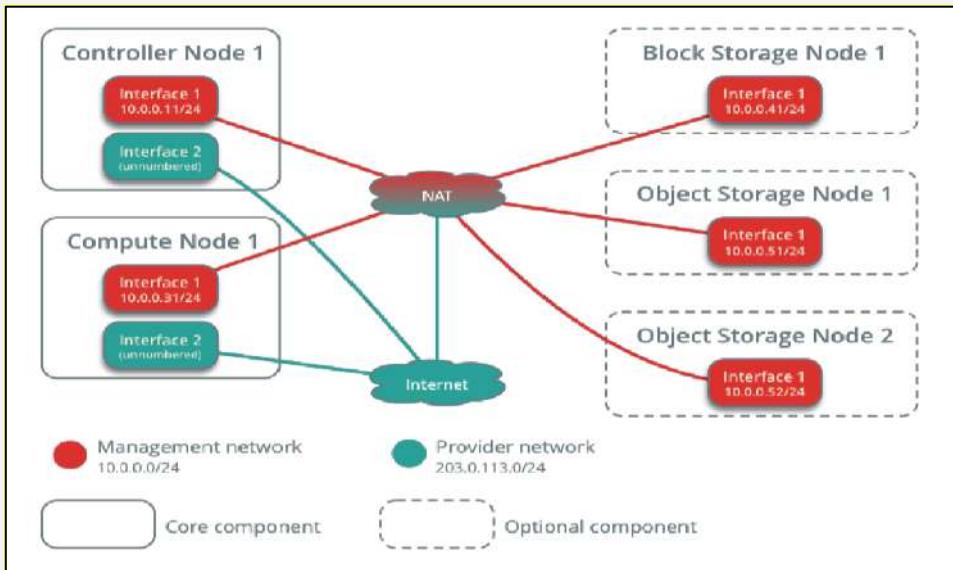
1 nodo Controller + 1 nodo Compute = Cloud più piccola e realizzabile possibile.

Il **nodo Block Storage** fornisce spazio di archiviazione che viene gestito dal sistema operativo come un dispositivo di archiviazione fisico ed è utilizzato per fornire uno spazio di archiviazione persistente per le istanze dei server, permettendo loro di salvare i dati anche dopo la loro terminazione.

Dispone di un solo NIC perché è sufficiente inviare i volumi alle immagini.

I nodi **Object Storage** servono per depositare i vari tipi di file. Anche in questo caso la memoria è di tipo permanente. **Si prevengono Disaster Recovery.**





Network Layout

La **rete in verde** rappresenta la rete fisica dell'ambiente (laboratorio IngSfw). Siccome, come detto, i nodi Controller e Compute hanno due interfacce, un'interfaccia sarà quella sulla rete verde, **la rete provider** che indica la rete fisica esistente e un'interfaccia sarà quella sulla **rete rossa**, la **rete di management**, che nessuno impedisce possa essere una rete di tipo VxLAN oppure un'interfaccia fisica. La nuvola mezza rossa e mezza verde rappresenta **il router**, un dispositivo di strato 3 che **collega le due reti e permette di operare in internet anche alla rete di management**.

Si potrebbero scaricare dalla rete immagini virtuali da usare poi all'interno della Cloud, quindi il collegamento è un aspetto chiave.

Se una rete Cloud dispone di 10.000 server ci dev'essere una rete che li collega tutti. **Se all'interno della rete Cloud qualcuno vuole utilizzare 30 macchine virtuali allora esse dovranno scambiarsi il traffico tra di loro.**

Ma che rete usano? Ci sono due opzioni.

La prima opzione, da evitare, consiste nell'attaccare le macchine virtuali sulla rete provider; cioè generare delle VM all'interno di computer che si collegano alla rete provider (rete verde) mediante la scheda di rete e navigare dunque in internet. Tuttavia, se si creano 30 VM che devono scambiarsi traffico tra loro non occorre accedere all'interfaccia fisica.

La seconda opzione prevede di utilizzare delle **reti overlay virtuali**, ovvero reti apposite realizzate dentro la Cloud sopra una rete fisica esistente.

Sono viste come reti proprietarie ma di fatto non esistono.

Lo standard di riferimento è **VxLAN**, il cliente vede le VM interconnesse.

Questa tipologia di rete è detta anche rete **self-service** poiché realizzata appositamente per il cliente. Si creano così più ambienti separati.

Quando si gestisce l'interfaccia sulla rete provider occorre fare **pass-through**.

Ovvero, **farla funzionare a strato 2** facendo passare tutto il traffico.

Deve funzionare **come fosse un bridge, senza assegnare indirizzi IP**.

Abbiamo visto dunque come creare reti private virtuali (VLAN) e sottoreti per isolare il traffico tra le VM. È anche possibile creare router virtuali per connettere queste reti private alla rete esterna e consentire l'accesso a Internet. Inoltre, è possibile utilizzare strumenti di sicurezza, come i **security groups**, per limitare l'accesso alle VM.

Come utilizzare OpenStack

Attraverso Horizon si fa richiesta per creare delle istanze.

In OpenStack, per istanza si intende una macchina virtuale che gira e che un utente può utilizzare da remoto. L'istanza viene creata utilizzando un'immagine, ovvero una copia del sistema operativo. Quando un'istanza viene avviata, deve disporre di un qualche tipo di sistema operativo per l'esecuzione. Occorre associare all'istanza di OpenStack una chiave pubblica e privata per poter accedere. È possibile creare queste chiavi mediante Horizon.

La chiave pubblica viene registrata in OpenStack mentre la chiave privata dovrebbe essere salvata e conservata in locale per aumentare la sicurezza.

Flavor: costituisce un insieme di variabili che specificano come dev'essere implementata un'immagine. Definisce in pratica le risorse a livello numerico per istanziare un'immagine: quanta RAM, quanti CORE, ecc., tutta la configurazione fisica. In un pool di Flavor, se ne sceglie uno per ogni VM in base alle necessità.

OpenStack permette di fare **overcommitting** della CPU e della RAM.

In pratica, permette di allocare più risorse di quelle effettivamente disponibili nell'host fisico, in modo da creare più macchine virtuali di quante potrebbero essere supportate dalle risorse fisiche disponibili.

Ciò causa una diminuzione delle performance delle varie istanze.

Sebbene sia possibile, non conviene fare overbooking.

La memoria RAM dev'essere considerata una risorsa esclusiva.

OpenStack Services

OpenStack Identity Service: Keystone

Keystone è il servizio OpenStack che memorizza molte informazioni per il suo lavoro. Ha quindi molti backend a disposizione (intesi come repository).

Autentica e autorizza l'utilizzo di tutte le risorse all'interno della Cloud.

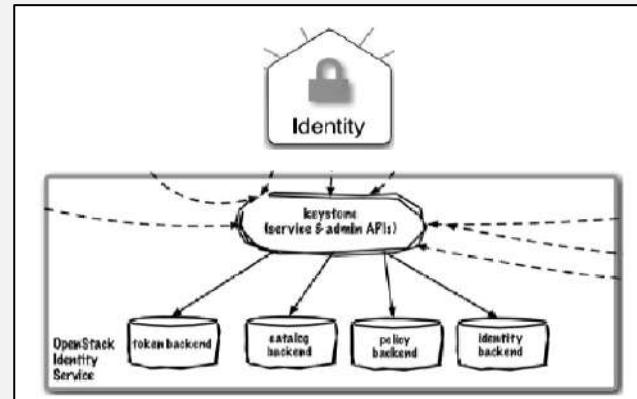
Autentica gli utenti attraverso un username e una password e poi con dei token.

Keystone genera, gestisce e scambia i token. Anche i servizi devono autenticarsi tra loro. È un continuo autorizzarsi.

Controllo degli accessi basato sul ruolo.

Gestisce gli endpoint dei vari servizi di tipo REST e altre tipologie per ogni servizio.

Implementa l'API Identity di OpenStack.



Architettura di Keystone

Keystone è organizzato come un gruppo di servizi interni esposti su uno o più endpoint. Molti di questi servizi sono utilizzati in modo combinato dal frontend.

Identity Service: fornisce ad ogni entità (utenti e gruppi di utenti) **delle credenziali di accesso e le valida quando c'è bisogno.**

Gli utenti rappresentano degli API individuali, un consumer.

Ogni nome utente dev'essere unico all'interno di un cosiddetto dominio.

Possiamo vedere un dominio come un namespace: è un gruppo di nomi dentro i quali il singolo nome non può essere ripetuto, nome utente unico all'interno del dominio. Possono essere ripetuti in domini differenti.

I gruppi di utenti sono dei contenitori di utenti, **utili quando si devono indicare delle configurazioni di policy che accomunano/servono a un tot di consumer.**

Il nome per un gruppo di utenti funziona come per l'utente singolo.

L'Identity Service è il primo servizio con cui l'utente interagisce.

Utenti e servizi possono localizzarsi e autenticarsi reciprocamente utilizzando **il catalogo dei servizi** gestito dall'Identity Service.

Ogni servizio in OpenStack ha tre tipologie di API REST (endpoint):

- Admin: per utilizzi amministrativi.
- Public: per l'utilizzo di tutti.

- Internal: una via di mezzo, cosa può fare lo decide l'amministratore.

Il servizio Identity è costituito da tre componenti:

- **Server**: è il servizio stesso. Centralizzato ed accessibile attraverso un'interfaccia REST che autentica le richieste degli utenti. Per fare ciò, il server utilizza i metadati degli utenti memorizzati in un qualche db.
- **Driver**: componente integrata nel server centralizzato. Fornisce la connettività con i vari database o altri strumenti esterni (ad esempio, un database SQL), altrimenti il server non può lavorare.
- **Moduli**: pezzi di software che intercettano le richieste e le girano al server. Infine, girano le risposte ai client.

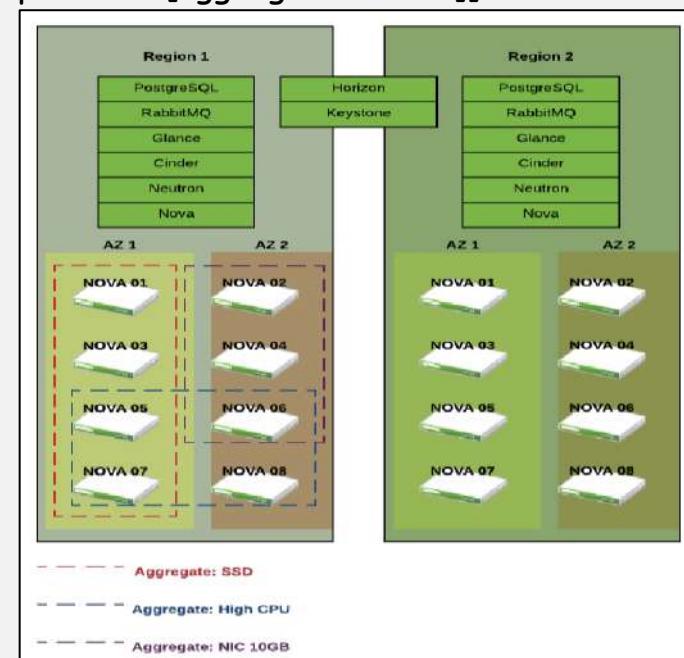
Resource Service: fornisce un'organizzazione delle risorse all'interno della Cloud. Quando si definisce un servizio da erogare ad un utente esterno, esso viene inquadrato in un Project (o Tenant, inquilino). Anche in questo caso i nomi dei vari Project devono essere univoci all'interno di un singolo dominio ma non a livello globale. **Per dominio si intende uno spazio privato per il cliente.**

I nomi dei domini devono essere univoci a livello globale.

I Project devono essere per forza inclusi in un dominio, altrimenti vengono assegnati a quello di default.

Partition Service: fornisce un supporto per suddividere la rete Cloud OpenStack in varie zone. Una rete Cloud OpenStack è divisa, infatti, in tre principali **zone gerarchiche**: regioni[zone di disponibilità[aggregati di host]].

- **Regions**: è il livello più grande. Costituisce l'intero deployment di OpenStack escludendo Horizon e Keystone.
- **Availability zones**: sono suddivisioni logiche all'interno di una regione. Ogni zona di disponibilità è costituita da un insieme di host fisicamente separati ma collegati da una rete. Garantiscono un'elevata disponibilità.
- **Host Aggregates**: gruppi di host che condividono caratteristiche comuni. Solo gli amministratori possono visualizzare o creare aggregati di host.



Ogni zona è separata dalle altre e se una di esse presenta dei problemi essi restano confinati al suo interno senza interferire altrove.

L'utente vede solo la Dashboard, non ha idea di dove siano i nodi.

Assignment Service: fornisce l'assegnazione dei ruoli.

Rappresenta il ruolo che l'utente ha all'interno della Cloud.

I ruoli vengono assegnati attraverso tuple di 3 elementi: ruolo, risorsa, identità.

Authentication Service: fornisce differenti tipologie di autenticazione.

Tale autenticazione avviene attraverso due meccanismi: [username, password] o Token. L'autenticazione mediante token avviene in automatico, senza che l'utente se ne accorga.

Possono esserci diversi tipi di token, vengono utilizzati uno per volta.

Di solito sono quattro le tipologie: UUID (Universally Unique Identifier), PKI (Public Key Infrastructure), Fernet e JWS (JSON Web Signature).

Le prime due sono deprecate, mentre Fernet è il più diffuso.

Keystone convalida e gestisce i token utilizzati per l'autenticazione delle richieste dopo che le credenziali di un utente sono già state verificate.

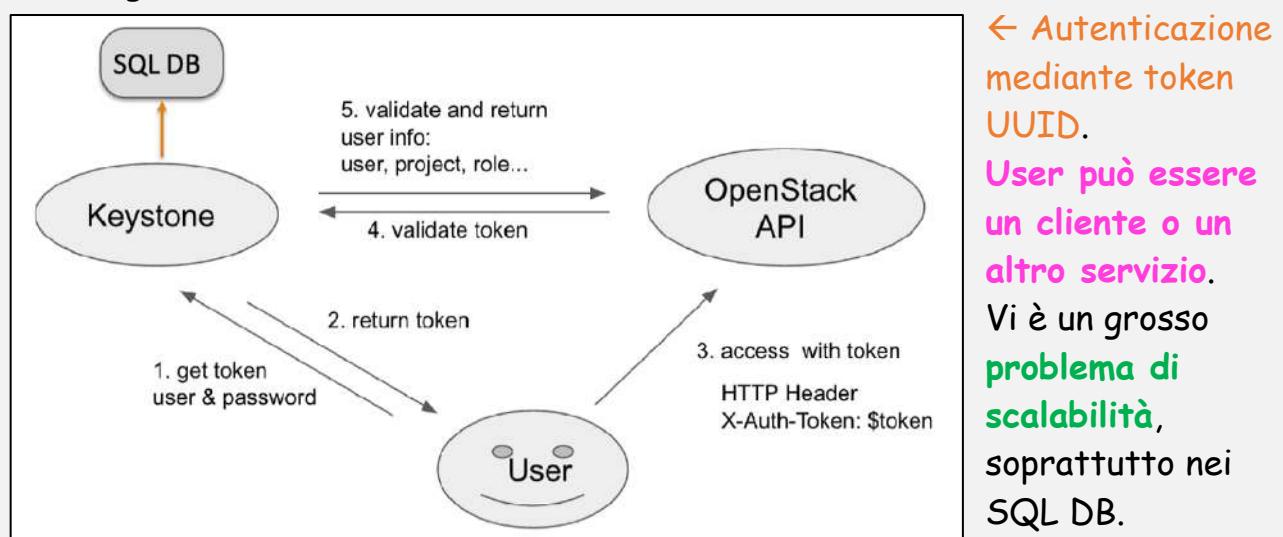
I Token vengono associati ad un utente o servizio per operare nella Cloud.

La prima cosa da fare è contattare Keystone per avere un Token.

Ogni token Fernet ha una scadenza.

Ogni volta che un utente accede ad un servizio, Keystone verifica se il token in suo possesso è ancora valido.

Quando genera i token, Keystone li memorizza (problema di scalabilità incoming) e deve verificare che siano validi continuamente.



Token UUID: il problema principale risiede nella **dimensione dei token** (varia in base alla tipologia utilizzata) che va a diminuire le performance in trasmissione sulla rete. Inoltre, **più il database aumenta** in dimensioni, per contenere le varie chiavi che si aggiungono, **più Keystone dovrà effettuare** delle operazioni di ricerca **sempre più approfondite**. Aumentando il tempo.

Inoltre, **Keystone deve periodicamente rimuovere i token dal database** che sono scaduti o in caso in cui le autorizzazioni degli utenti vengano cambiate.

Tutte queste operazioni degradano incrementalmente le performance.

Token PKI: dal momento che Keystone diventa un collo di bottiglia, **si fa in modo di non chiedere sempre la validità del token**.

In questa modalità Keystone agisce come un **controllore di validità**. Gli viene assegnata una coppia di chiavi e quando l'utente richiede un token, Keystone risponde inviandogli un token firmato **con la sua chiave di cifratura privata**.

Il token contiene l'intero catalogo dei servizi, con tutti gli endpoint a cui l'utente può accedere. Tutto firmato digitalmente.

Di conseguenza, **i vari servizi devono avere la chiave pubblica di Keystone**.

Di conseguenza, **occorre configurare tutta la rete Cloud**.

Così facendo, ogni API di un servizio **può validare il token di un utente in locale**.

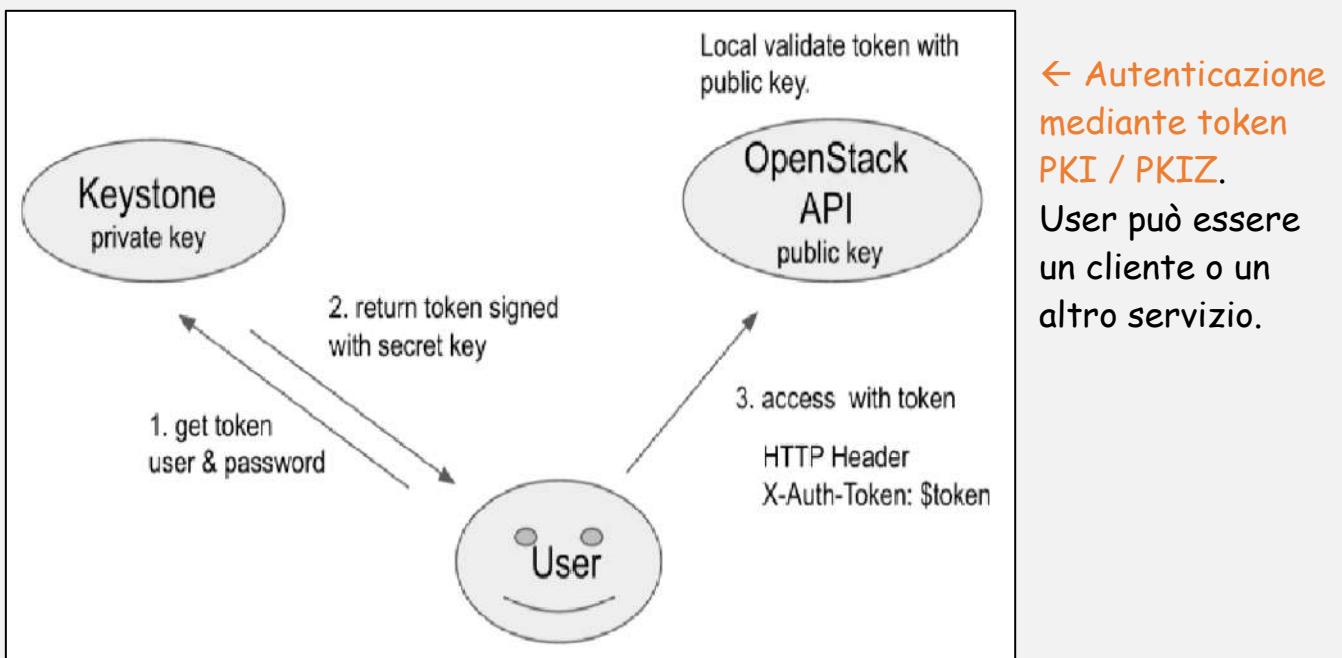
Invece di far girare i token, si fanno girare le chiavi.

È un buon compromesso ma non risolve tutti i problemi.

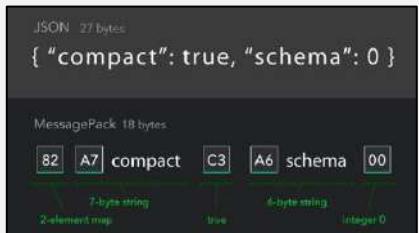
Infatti, si riducono gli scambi ma sono enormi.

Allora si procede zippando le chiavi, senza perdita. **PKIZ**.

Tuttavia, a causa dell'entropia, non si può comprimere eccessivamente.



Token Fernet: sono dei token effimeri, di breve durata, che hanno una scadenza. Dispongono di un **payload** con poche informazioni per verificare la loro validità. Keystone li verifica **senza** doverli memorizzare. Fernet consente di scambiare dati attraverso differenti linguaggi, tra cui JSON che è leggero e veloce, seppur leggibile da umani.



Questi token quando sono inviati in rete vengono **cifrati con una chiave simmetrica**. Tali chiavi sono segrete e non possono essere sempre le stesse, occorre aggiornarle per garantire la sicurezza. Queste chiavi le usa Keystone. Le chiavi sono conservate in un **repository di chiavi** che Keystone passa a una libreria che gestisce la crittografia e la decrittografia dei token.

Queste chiavi **hanno di un ciclo di vita prestabilito** e in base al momento possono essere primarie, secondarie, o staged (entrano in scena).

La **chiave primaria** è autorizzata a **crittografare e decrittografare i token**.

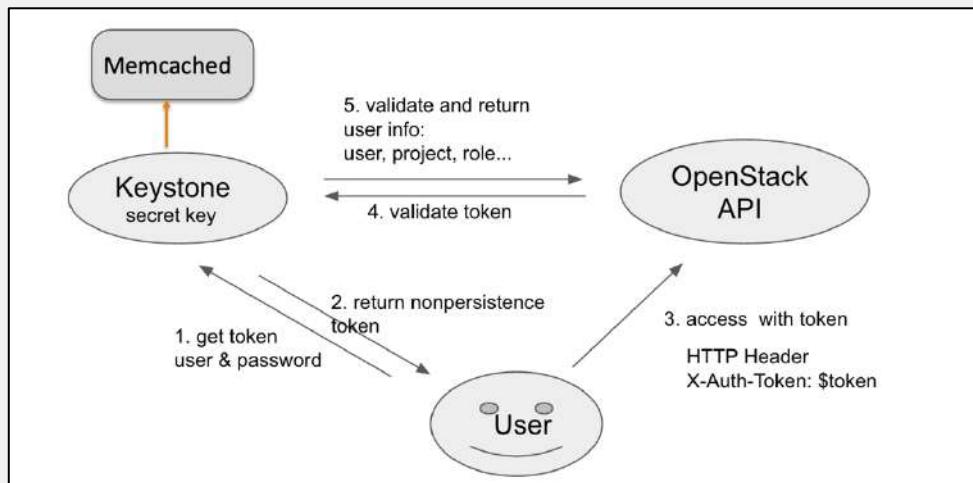
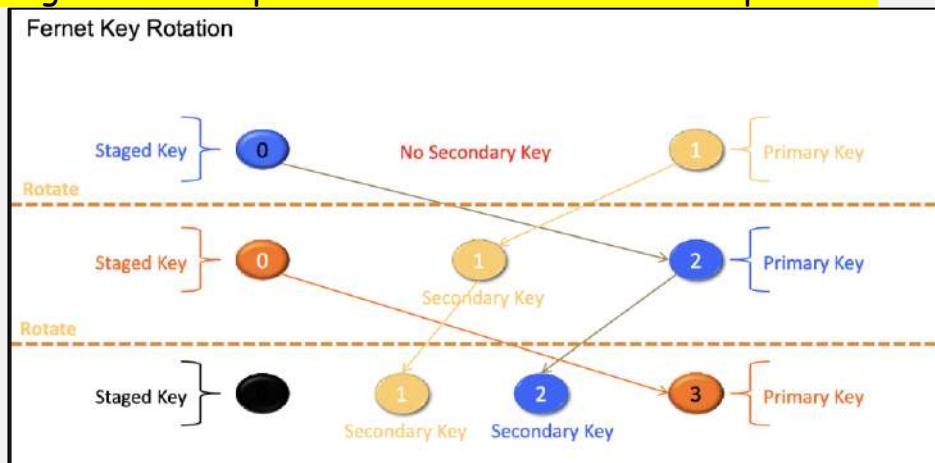
La **chiave secondaria** è una chiave che in un certo momento era una chiave primaria ma che è stata retrocessa al posto di un'altra nuova chiave primaria. Con la chiave secondaria è **consentito solo decrittografare i token**.

Keystone, infatti, dev'essere in grado di decrittografare i token creati con le vecchie chiavi primarie.

La **chiave staged** è una chiave speciale che entra in scena quando occorre sostituire la chiave primaria corrente. Proprio come le chiavi secondarie, le

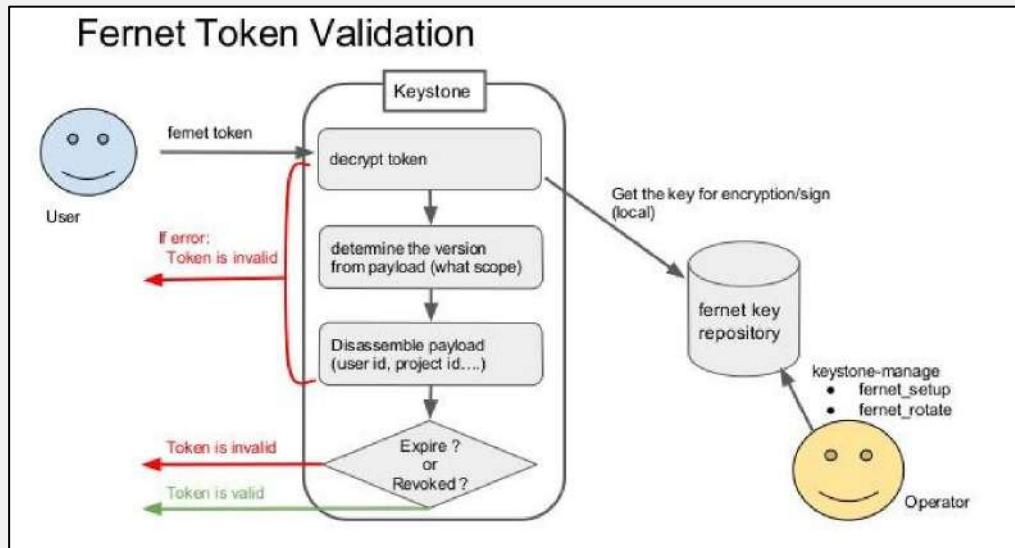
chiavi staged hanno la capacità di decrittografare i token. A differenza delle chiavi secondarie, però, non sono mai state una chiave primaria.

La chiave staged sarà sempre la successiva nuova chiave primaria.



← Autenticazione mediante token
Fernet. User può essere un cliente o un altro servizio.

L'utente ha il suo token in forma criptata e può utilizzarlo per autenticarsi con altri servizi. Nemmeno il servizio stesso può leggere il token, lo rimanda dunque a Keystone che lo decripta e verifica che sia ancora valido (verificando la scadenza e altre informazioni). Se è valido risponde al servizio comunicandoglielo, si sblocca l'operazione. Keystone conserva necessariamente le chiavi, se la Cloud è piccola non le condivide con nessuno.

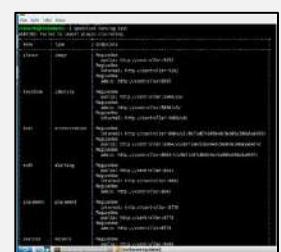


Anche se i token fernet funzionano in modo molto simile ai token UUID, **non richiedono persistenza**. Il database dei token di **Keystone** non subisce più **problemi di sovradimensionamento** come effetto collaterale dell'autenticazione. **L'eliminazione dei token scaduti dal database dei token non è più necessaria** quando si utilizzano i token Fernet poiché non richiedono persistenza, non devono essere replicati. Finché ogni nodo Keystone condivide lo stesso repository di chiavi, i token Fernet possono essere creati e convalidati istantaneamente tra i nodi. Anche i token Fernet presentano comunque dei problemi, di natura politica: non sono degli standard. Fuori da OpenStack non sono universali e altri sistemi non li usano.

Per la compatibilità della Cloud con altri servizi esterni non sono ottimali.

Token JWS: sono dei token **effimeri come i token Fernet**, il che significa che non sovraccaricano il database né richiedono la replica tra i nodi. Tuttavia, uno dei problemi principali è che **sono dei token decifrabili da tutti**.

Catalog Service: fornisce tutti gli endpoint dei vari servizi a cui l'utente può accedere. Per questo prende il nome di catalogo. Consente alle API dei client di scoprire e navigare all'interno dei servizi della Cloud.



Policy Service: fornisce le specifiche dei diritti d'uso per le risorse delle varie componenti di OpenStack. Sono dei file JSON presenti nei file system delle VM che specificano i diritti d'uso.

OpenStack Image Service: Glance

Glance è il servizio di OpenStack che **memorizza le immagini delle macchine virtuali su possibili backend**, ad esempio nel filesystem dove c'è il Controller. Glance interagisce con Keystone e con qualunque altro client, espone un'API. Gli scambi per le autenticazioni verso Keystone sono sempre presenti.

Glance **utilizza un database di tipo relazionale**.

Il **Domain Controller** è il cuore di Glance, il quale gestisce i diritti d'uso, d'autorizzazione, le policy di utilizzo, le quota risorse per le immagini delle VM. Dentro Glance vi è un database per memorizzare i metadati relativi ad un servizio. Per accedere al database di Glance il Domain Controller deve far uso di un **Database Abstraction Layer** cioè una sorta di API che unifica le comunicazioni tra Glance e i vari database evitando di rendere il Domain Controller dipendente dal singolo database. Nella maggior parte delle implementazioni viene utilizzato un database relazionale ma con questa API il funzionamento del Controller è sempre lo stesso.

Glance eroga un servizio a Nova ma necessita di altri servizi, come Keystone per autenticare i token, quindi c'è un API REST per essere contattati dall'esterno. Per la comunicazione con Keystone, Glance utilizza un middleware interno e non l'API REST.

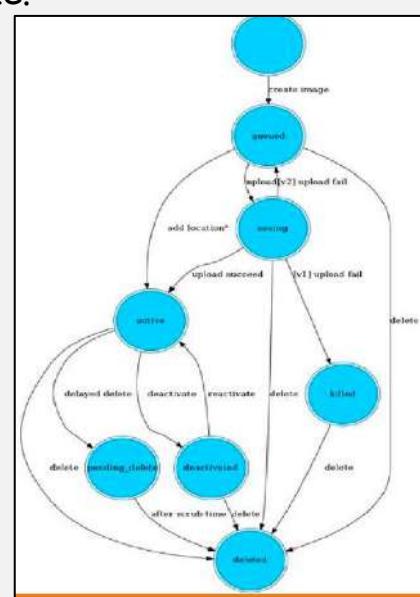
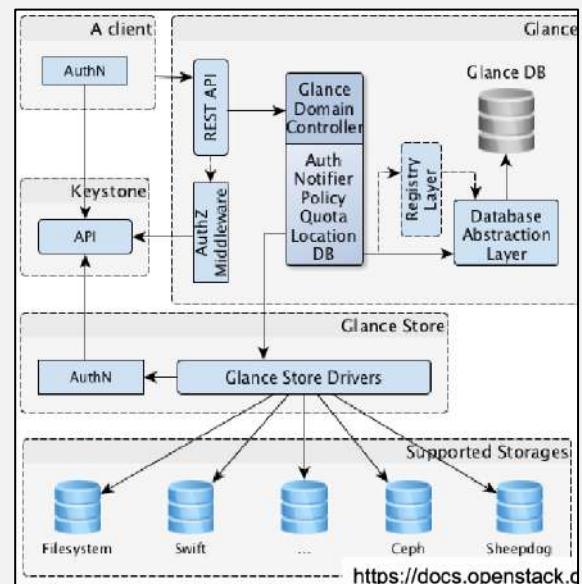
Il **Registry Layer** è un elemento opzionale per rendere sicure le operazioni con il database. Il **Glance Database** memorizza i metadati delle immagini ed è possibile scegliere un database in base alle proprie preferenze.

Le immagini vengono create attraverso l'utilizzo di file ISO o scaricandole direttamente dalla rete.

Una volta create, vengono messe in coda e poi salvate.

A questo punto possono entrare nello stato active per essere utilizzate. Le immagini hanno anche dei diritti d'uso associati: se l'immagine è **pubblica** allora chiunque può utilizzarla e sarà presente nella lista di default per tutti gli utenti. Se è **privata** allora può essere utilizzata solo dal proprietario. Se è **community** allora può essere utilizzata da tutti gli utenti ma non compare nella lista di default.

Se è **shared** allora può essere utilizzata solo da un gruppo di utenti selezionato dal proprietario e la vedranno nella lista di default. Quando si creano le immagini è possibile specificare la loro visibilità.



OpenStack Nova

Nova è il servizio di OpenStack che riceve da Keystone, per l'autenticazione, e da Glance, per le immagini, ed offre un servizio completo per andare ad implementare le macchine virtuali.

I servizi visti finora sono implementati nel Controller.

Nova è, invece, distribuito in parte nel Controller e in parte nei Compute. Nella componente che sta nei Compute sarà presente un hypervisor, per far girare le macchine virtuali.

---->

Nova può lavorare con diversi tipi di hypervisor.

Nova Compute è colui che mette in opera le VM utilizzando le immagini fornитogli da Glance e instaurando la rete attraverso il supporto di OpenStack Neutron.

Il Conductor è un intermediario utilizzato per disaccoppiare il Compute dal database che viene dunque utilizzato attraverso di lui, aumentando la sicurezza.

Il database contiene i metadati delle VM stanziate.

Lo scheduler decide in quali nodi istanziare le VM.

All'interno di Nova abbiamo degli scambi orizzontali (interni alla Cloud con gli altri servizi) attraverso l'utilizzo di RabbitMQ con protocollo AMQP.

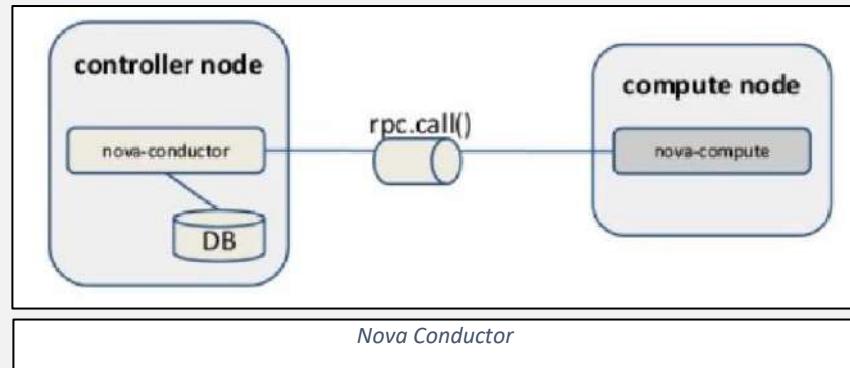
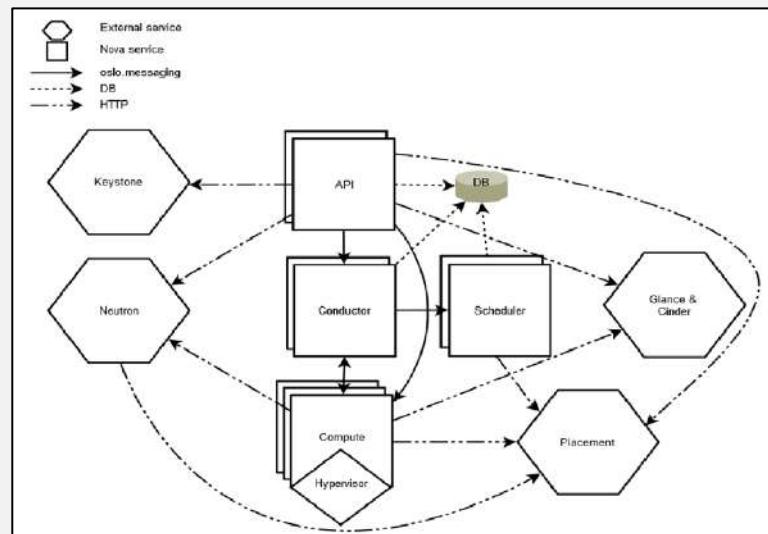
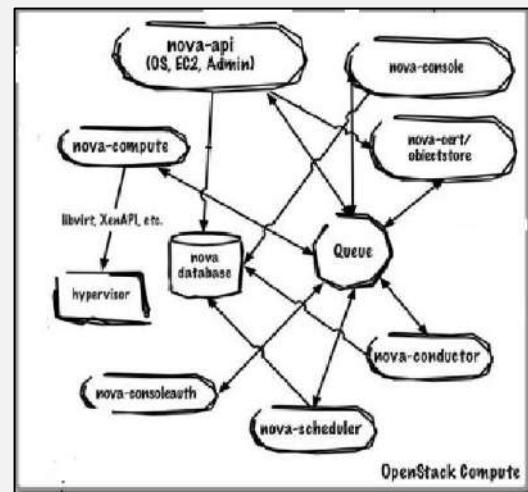
RabbitMQ utilizza delle RPC (chiamate di procedura remota).

I messaggi finiscono nei vari topic di RabbitMQ e Nova Compute li legge da lì e non direttamente.

C'è il disaccoppiamento.

Quando a Nova viene chiesto di istanziare una VM ma l'operazione non va a buon fine viene tenuta traccia della richiesta.

Le richieste non esaudite sono conservate in un altro database relazionale.



Celle: fin quando le Cloud sono piccole le macchine virtuali si istanziano nei pochi nodi Compute che sono disponibili. Ma quando crescono è necessario andare a partizionarle anche per evitare la diffusione globale di possibili problemi che da una parte si riflettono sull'altra. **Questo partizionamento prende il nome di celle.** I nodi Compute vengono organizzati in celle che operano in maniera indipendente. Le celle sono configurate come un albero.

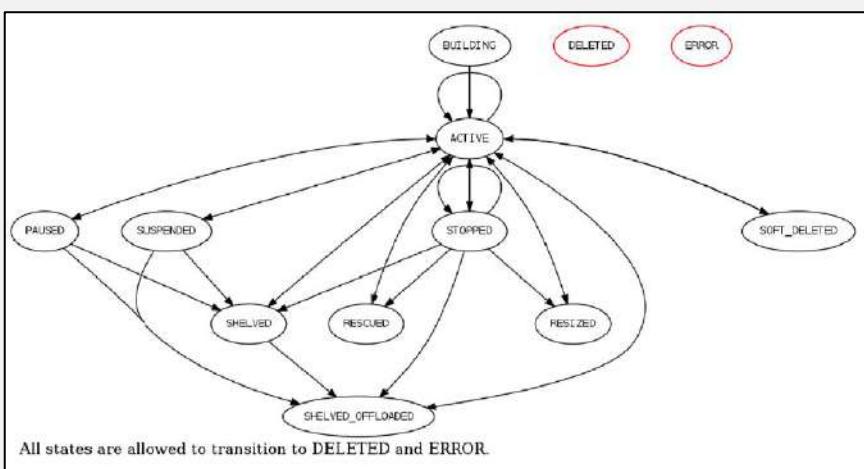
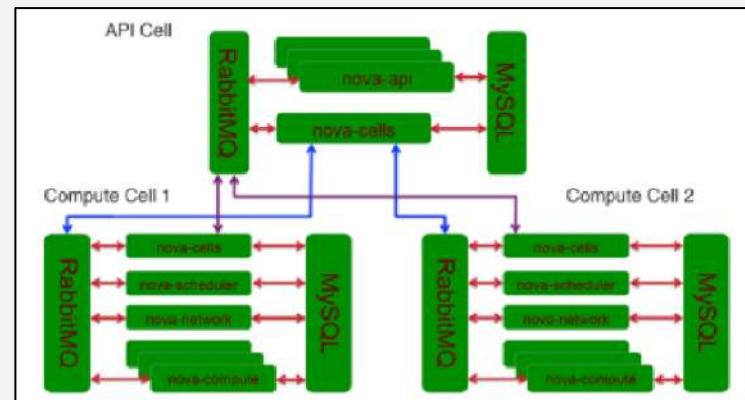
Possiamo dunque vedere le celle come gruppi di VM.

Possiamo associare le celle alle VM con un'unica istruzione.

Le celle sono degli aggregati di VM.

La differenza tra una **cella singola** e una **cella multipla** è nell'organizzazione a livello gerarchico.

Nella multipla ci sono più celle, si utilizza un super Conductor che interagisce con i Conductor delle varie celle. Anche RMQ è replicato per ogni cella. L'intera Cloud quindi si realizza in diverse porzioni. La cella di primo livello dovrebbe avere un host che esegue un servizio nova-api, ma nessun servizio nova-compute. Ogni cella figlia esegue tutti i tipici servizi nova-* in un normale nodo Compute ad eccezione di nova-api. Le celle possono essere viste come una normale distribuzione di calcolo in quanto ogni cella ha il proprio db, server e broker con code di messaggi.



L'istanza di una VM può trovarsi in vari stati. Quando viene lanciata l'istanza essa va in stato di **building** per poi passare, in caso di successo, allo stato di **active**. Da qui, gli stati dove l'istanza può finire sono molteplici. Può finire nello stato **suspended, stopped, shelved** (ovvero messa da parte),

rescued (ovvero recuperata), **resized** (ovvero riorganizzata), **self deleted** (ovvero verso la cancellazione).

All'interno di ogni stato ci sono delle operazioni che consentono di passare da uno stato all'altro.

Nova Scheduler: definisce quali sono i nodi dove le istanze delle VM devono essere messe in opera, successivamente Nova passa le immagini a tali nodi. Abbiamo un **gruppo di host** che possono ospitare le VM, altro non sono che i nodi Compute della Cloud. Viene eseguita **un'operazione di filtraggio** che ha come obiettivo quello di individuare i nodi che possono ospitare la/le VM di cui si è richiesto l'istanziamento.

L'operazione di filtraggio altro non fa che individuare gli host che hanno le risorse adeguate.

Si sposa bene con i **Flavor**.

Fatto il filtraggio, quello che segue è **un'operazione di "pesatura"**.

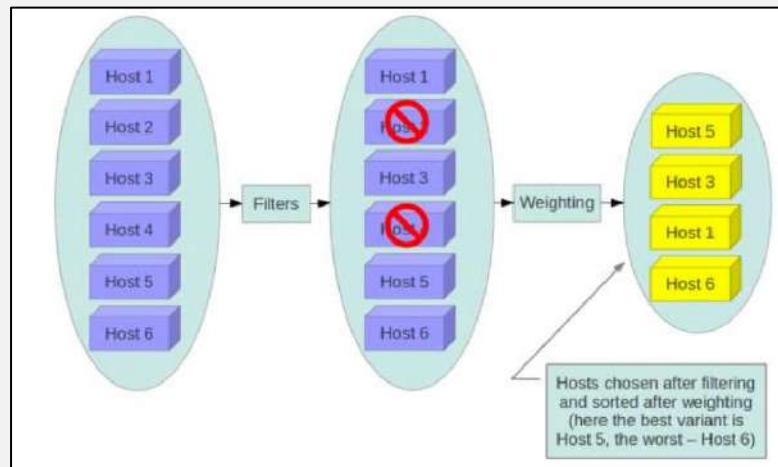
In un grande datacenter la politica potrebbe essere quella di condensare più VM possibili all'interno dello stesso RACK così che elettricità, raffreddamento ecc. andrebbero gestiti solo in quell'armadio, riducendo il costo rispetto ad utilizzarne un altro. **Pesatura = ordino gli host per capacità**.

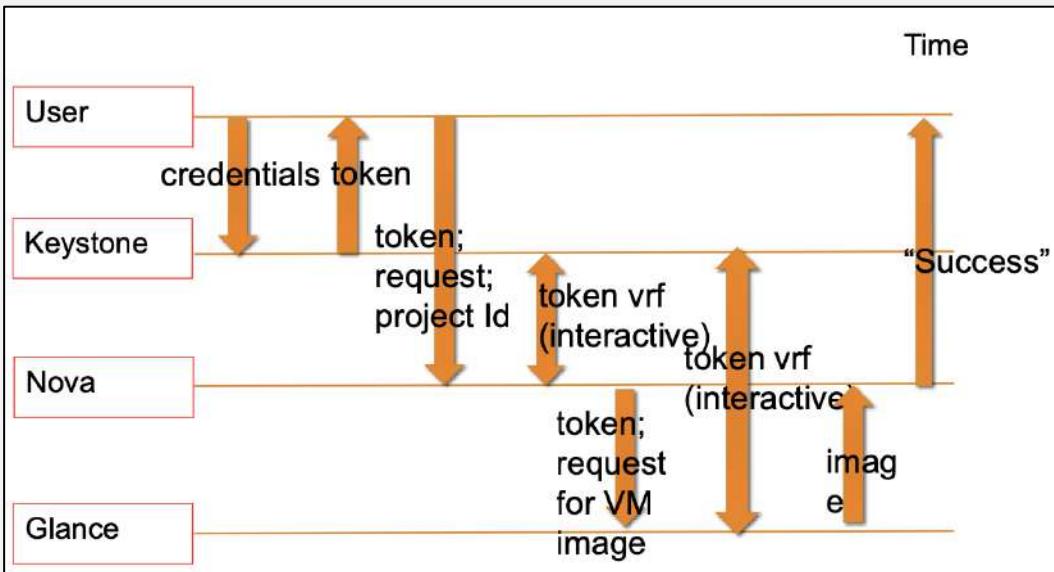
Un'altra politica potrebbe essere quella di fare Node Balancing, caricando i server in maniera equa.

L'opzione di default di OpenStack è quella di distribuire equamente l'utilizzo della RAM sui vari nodi. Si inserisce la VM nel nodo che ha più RAM a disposizione. **Questo però aumenta lo sparpagliamento**.

Riassumendo, il Nova Scheduler esegue operazioni di filtraggio + pesatura.

Sommario delle operazioni eseguite per istanziare un progetto di un utente





Quest'immagine mette **insieme i concetti** visti fino ad ora.

Quando **l'utente** richiede una VM, ha bisogno di un token.

Quindi effettua l'autenticazione con **Keystone** tramite le credenziali e, **se ne ha diritto**, riceve il token. A questo punto l'utente contatta **Nova** chiedendo una VM specificando l'ID del suo progetto e inviando il suo token per autenticarsi. A questo punto Nova contatta Keystone per verificare che il token che ha ricevuto dall'utente sia valido e, **se la risposta è positiva**, successivamente va a chiedere l'immagine della VM a **Glance** (utilizzando sempre il token per l'autenticazione). Glance, a sua volta, ricevuta la richiesta prende il token ricevuto da Nova e contatta Keystone per verificarne la validità e, **se la risposta è positiva**, successivamente invia l'immagine a Nova.

A questo punto Nova è in grado d'istanziare l'immagine della VM e invia una risposta di **successo** all'utente.

OpenStack Neutron

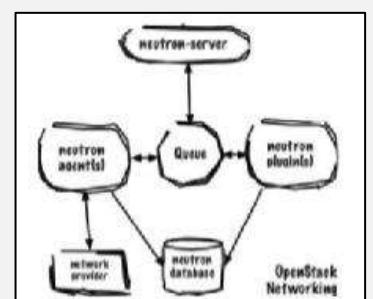
Neutron è il servizio di OpenStack che fornisce delle reti virtuali per interconnettere le schede di rete delle istanze che vengono messe in opera.

Le reti possono essere completamente interne alla Cloud o interfacciarsi con dei router virtuali alla rete del datacenter.

Possono essere anche un'estensione naturale della rete fisica.

L'obiettivo è quello di mettere a disposizione dei tenant contenenti delle VM connesse da una rete.

Ogni utente può configurare la propria topologia di rete.



Neutron è caratterizzato da quattro principali elementi:

- **Network**: inteso come un oggetto virtuale che può essere creato. Il Network fornisce la rete ad ogni tenant presente nella Cloud. Una rete è l'equivalente di uno switch con porte virtuali che può essere dinamicamente creato o eliminato in base alle esigenze di utilizzo. Quando si parla di rete si fa riferimento allo strato 2.
- **Subnet**: un pool di indirizzi IP. Due subnet differenti comunicano tra loro attraverso un router. Quando si parla di sottorete si fa riferimento ad oggetti associati alla rete per fornire supporto di stato 3. Quindi, per avere una connessione di strato 3, occorre una rete e una sottorete.
- **Port**: entità a sé tramite la quale router e VM si attaccano alla rete.
- **Router**: entità virtuale che può essere creato ed eliminato, si occupa di selezionare le reti in cui spedire i dati. Gestisce i **Floating IP**.

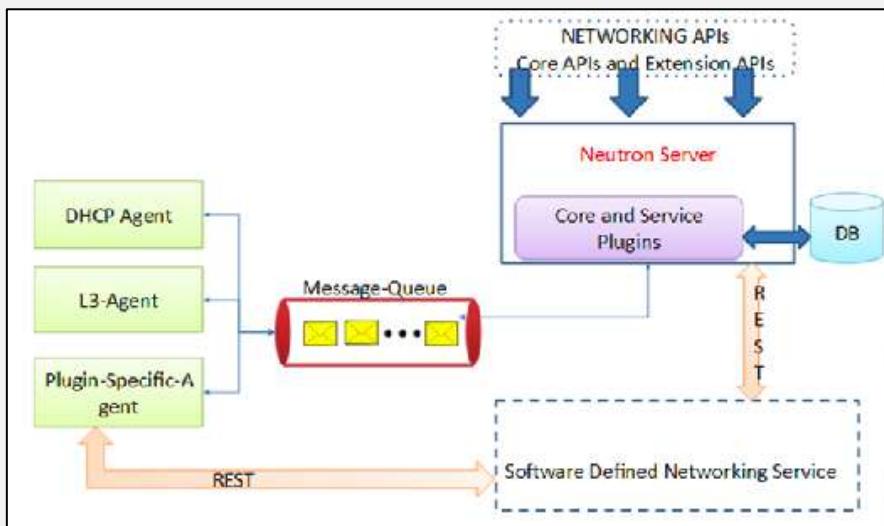
Neutron, come Nova, è un servizio distribuito e pertanto dev'essere sia nel Controller sia nei vari Compute.

Nel Controller è presente il demone **neutron-server**, la principale entità software di Neutron, che si occupa di esporre un'API REST per connettersi con il mondo e di implementare dei meccanismi di networking attraverso dei Plugin. È dunque l'elemento centrale del servizio.

Sulla base delle richieste attiva il Plugin richiesto.

Neutron fa uso di RabbitMQ per comunicare con le altre componenti OpenStack inserendo i messaggi in una coda attraverso il protocollo AMQP. Viene inoltre avviato il servizio Neutron RPC in modo che i server Neutron possano comunicare con gli agenti. Il servizio RPC carica di fatto i Plugin.

Architettura di Neutron



All'interno del **Neutron Server** ci sono delle entità chiamate **Plugin**.

Plugin Core: offrono funzionalità di strato 2 creando reti per le VM, anche se non sono istanziate nello stesso nodo. Nulla vieta ad un cliente di avere un tenant composto da tre Compute: uno a Milano, uno a Sidney e uno a Londra. Per connetterli la tecnologia utilizzata è **VxLAN**, sono dei tunnel di rete.

Plugin Service: offrono funzionalità di strato 3 gestendo le comunicazioni con delle interfacce REST. I Plugin di OpenStack Neutron forniscono funzionalità specifiche per la gestione delle reti e delle risorse di rete, consentendo a Neutron di supportare una vasta gamma di tecnologie di rete e di essere **personalizzato** in base alle esigenze specifiche dell'utente o dell'applicazione.

Fuori dal **Neutron Server** ci sono delle entità chiamate **Agenti**.

Sono dei programmi remoti situati all'interno dei nodi **Compute**.

Quando il Neutron Server riceve un'operazione per una configurazione di rete o per una richiesta di gestione delle risorse di rete da un utente o da un altro componente di OpenStack, il Server determina quali agenti di Neutron devono essere coinvolti nell'operazione e invia loro gli ordini necessari da svolgere.

Gli agenti di Neutron, quindi, eseguono le operazioni richieste sulla macchina in cui risiedono, comunicando con il Neutron Server per fornire informazioni sullo stato corrente delle operazioni e per ricevere ulteriori ordini.

C'è anche la possibilità di far pilotare il networking implementando il servizio Software Defined Networking (SDN).

Attraverso l'SDN il Nodo Controller va a configurare in maniera remota i nodi. È una sorta di cervello centrale che invia i comandi di configurazione agli Agenti remoti sulla base di quello che gli comunica il Neutron Server.

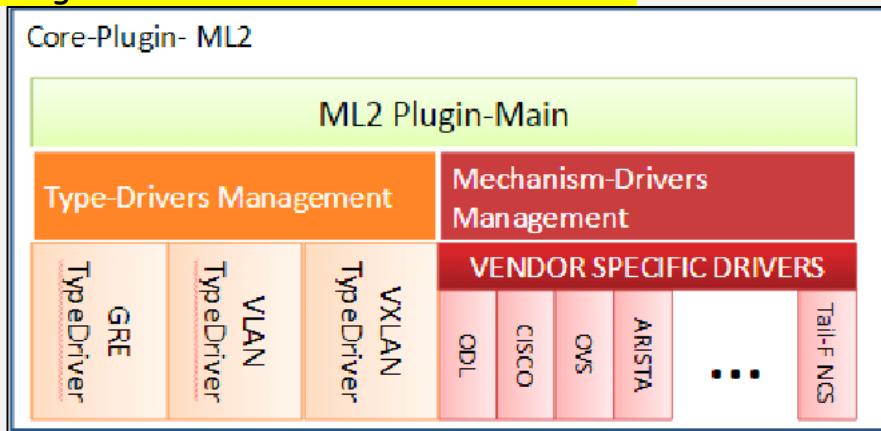
Così come ogni altro servizio in OpenStack, **Neutron dispone di un proprio database relazionale** contenente, in questo caso, le informazioni dei vari Plugin, i dati di configurazione e le relazioni tra reti, sottoreti, porte e router.

Uno dei principali Plugin di Neutron è il **Modular Layer 2 (ML2)**.

Viene installato direttamente insieme a Neutron.

È un **Plugin di tipo Core**, pertanto fornisce connettività di strato 2.

Include tutta una serie di driver per gestire la connettività verso un qualcosa di fisico o di remoto. La sua principale caratteristica è quella di permettere a diverse tecnologie di diversi vendori di coesistere.



Le istanze delle macchine virtuali avranno le proprie interfacce (porte virtuali) e si dovranno attaccare ad una rete. Le reti messe a disposizione sono quelle implementate dal Plugin ML2. Alcuni esempi sono VLAN e VxLAN. Queste sono le famiglie di reti che attacchiamo alle macchine virtuali ma possono essere customizzate e possono emulare le tecnologie più importanti dei vendori.

ML2 utilizza due tipologie di driver: i **Type-Drivers**, che stabiliscono le operazioni e le configurazioni di base per la tecnologia che dev'essere utilizzata (sono del tutto generici, non fanno riferimento a un venditore particolare), e i **Mechanism-Drivers**, che stabiliscono delle configurazioni che dipendono dal venditore. Perché occorre utilizzare la tecnologia di altri?

Perché se, ad esempio, si utilizza un datacenter con un'infrastruttura CISCO, Neutron avendolo già a disposizione aiuta i vari vendori.

Tutti i vendori compatibili con lo standard fanno internetworking.

Tra i Service Plugins, i più famosi ed utilizzati sono:

- FWaaS: Firewall as a Service.
- VPNaas: VPN as a Service.
- LBaaS: Load Balancing as a Service.

Essendo di tipo Service, offrono tutti funzionalità di strato 3.

Configurazione di una rete

Per configurare una rete occorre, come detto, una **network**, una **subnet**, delle **porte e router** insieme a una serie di **istruzioni** per permettere ai nodi Compute di attaccarsi alle porte di queste reti. Sono due le tipologie di reti:

- **Reti virtuali e interne** dette project o **self-service**: sono reti dell'utente che, senza altri meccanismi, **non possono scambiare dati con l'esterno**. Creano una rete tra le istanze del tenant, spesso con il protocollo VxLAN.
- **Reti provider**: reti virtuali ma **in connessione con la rete fisica**, con l'infrastruttura dov'è stanziatato il datacenter. Come se fossero un'estensione della rete LAN del laboratorio. Queste reti si connettono, o **mappano**, alle reti di strato 2 esistenti nel datacenter. Devono disporre di indirizzi IP compatibili con i quali è possibile navigare in internet.

Basta semplicemente che le VM nei nodi Compute utilizzino un Hypervisor che permetta la connettività alle macchine virtuali.

Ad esempio, **con VirtualBox si creano delle interfacce virtuali di tipo bridge**.

Non devono avere un indirizzo IP, **FONDAMENTALE**, dev'essere di strato 2.

Quindi, **nei nodi Compute si attiva l'interfaccia di rete ma senza configurazione IP**, deve operare a strato 2. Il suo funzionamento è di tipo **pass-through**, così le VM attraverso il nodo Compute, che funge da bridge, accendono alla rete e possono navigare. Questa è una rete provider.

Come far navigare le reti self-service?

Attraverso un router si connettono le reti self-service con la rete provider.

Occorrono delle operazioni di natting.

Se volessimo creare un server e metterlo in una rete Cloud, come assicuriamo agli utenti che sono fuori di accedere al server nella Cloud?

Si fa natting. **Neutron assegna degli IP pubblici alle VM che sono pre-allocati nel server ma di cui non sono a conoscenza**.

Quando passa il traffico nel router esso viene mappato con il Floating IP (FIP) associato alle macchine interne.

In questo modo i server sono raggiungibili attraverso l'esterno.

I FIP sono sconosciuti ai server dentro alla Cloud OpenStack.

I **Floating IP** (o indirizzi IP fluttuanti) sono indirizzi IP temporanei che possono essere assegnati a un'istanza di un server o a una risorsa di rete all'interno di un ambiente di cloud computing.

L'obiettivo dei floating IP è quello di consentire una maggiore flessibilità e disponibilità dei servizi, **consentendo di spostare rapidamente un indirizzo IP da una risorsa all'altra senza interruzioni del servizio per gli utenti**.

In pratica, un Floating IP viene assegnato a una risorsa, come ad esempio un server web, e quando questa risorsa si sposta o viene sostituita da un'altra, l'indirizzo IP fluttuante viene spostato sulla nuova risorsa senza che gli utenti debbano modificare l'indirizzo IP a cui si connettono.

In questo modo, il servizio risulta continuo e senza interruzioni.

Il traffico in uscita dalle VM viene quindi mappato con un Floating IP (FIP), che è associato alla VM internamente ma sconosciuto all'interno della Cloud

OpenStack. Il FIP consente di indirizzare il traffico esterno verso la VM corretta, consentendo ai server di essere raggiungibili dall'esterno.

I Floating IP sono spesso utilizzati in ambienti di cloud computing come Amazon Web Services (AWS) o Microsoft Azure per fornire servizi altamente disponibili e scalabili.

OpenStack Data Storage

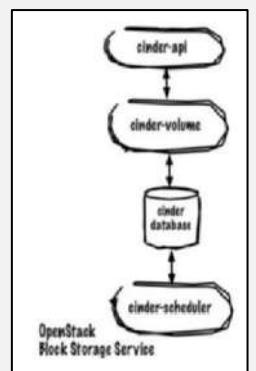
La memorizzazione dei dati in OpenStack può avvenire in 3 possibili modi:

- **Effimero**: ad ogni VM è assegnato uno storage per eseguire le varie operazioni e memorizzare i dati. Il ciclo di vita dei dati è pari al ciclo di vita della VM, quindi, se la VM muore muoiono anche i dati.
- **Block Storage**: alle VM vengono attaccati dei volumi, per gestire i propri dati. I volumi risiedono dello spazio fisico del disco, o in altri spazi fisici. Il gestore di questa metodologia è **Cinder**, il quale deve girare come all'interno dei server.
- **Object Storage**: i dati vengono conservati come oggetti binari che possono essere ricevuti o scritti mediante richieste HTTP. Permette di accedere ai dati mediante un URL. Sono repository che possono essere ritirate quando se ne ha bisogno, tramite servizio web. Il gestore di questa metodologia è **Swift**, il migliore per gestire dati non strutturati come e-mail, video, immagini, etc.

Cinder crea dei volumi permanenti e li associa alle VM.

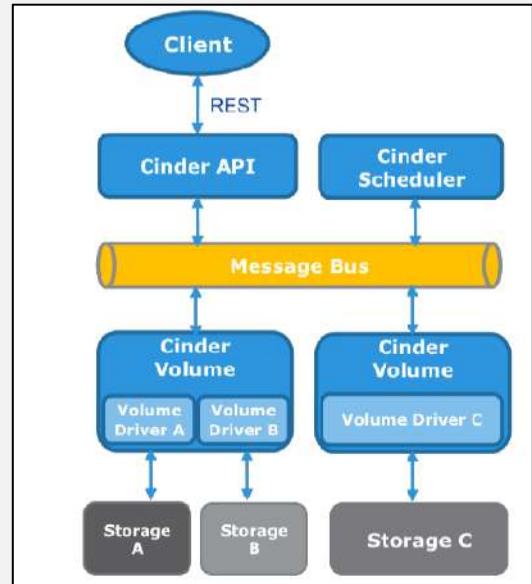
Implementa servizi e librerie per un accesso su richiesta ai dati e alle risorse sulle reti self-service. È progettato per presentare le risorse di archiviazione agli utenti finali che possono essere utilizzate da OpenStack Compute (Nova). Questo viene fatto attraverso l'uso di un'implementazione di riferimento (LVM).

Cinder virtualizza la gestione dei volumi e fornisce agli utenti finali un'API self-service per richiedere e consumare tali risorse senza bisogno di alcuna conoscenza su dove sia effettivamente distribuito il loro storage o su quale tipo di dispositivo sia.



Cinder scheduler è la parte di Cinder che decide quali macchine fisiche utilizzare per fare il deployment dei volumi. Decide tra tutti i volumi disponibili, quali utilizzare per metterli a disposizione come istanze di OpenStack. La richiesta di istanziare i volumi avviene attraverso l'API di Cinder.

Cinder Volume è la parte di Cinder che attraverso LVM (Logical Volume Manager) gestisce i volumi. Essenzialmente, raggruppa (mappa) partizioni fisiche in un unico volume logico. Come tutti i servizi visti, utilizza una coda di messaggi per la comunicazione.



Swift è il servizio di storage a oggetti, più complesso di Cinder. È formato da una serie di elementi, i principali sono gli storage servers Account, Container e Object.

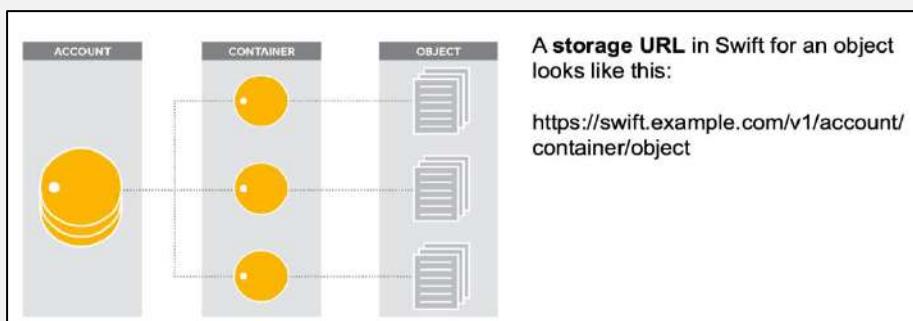
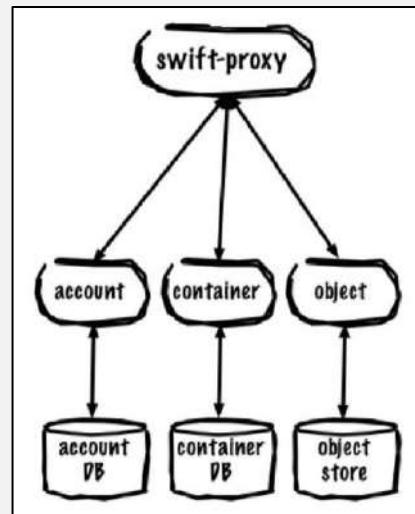
Swift utilizza i server per mettere a disposizione **in modo gerarchico** account, container e oggetti.

Swift fornisce un semplice servizio di archiviazione per le applicazioni che utilizzano interfacce RESTful, fornendo la massima disponibilità dei dati e capacità di archiviazione.

Per accedere ai servizi di Swift bisogna parlare col proxy server **Swift-Proxy**. Solo attraverso di esso è possibile accedere ai servizi di Swift. È l'unico in contatto col mondo.

In Swift esistono 3 categorie da archiviare:

- Account: può essere visto come un namespace, è l'ambiente isolato che ogni utente dispone per i propri file. Un account è un account utente.
- Container: può essere visto come un contenitore che contiene i file, i quali sono nella forma chiave-valore. La chiave è il Container mentre il valore è un puntatore al file conservato dentro Object.
- Object: risorse vere e proprie, pescate con un'interfaccia REST.



Livello gerarchico.
Account → Container → Object.

Gestione della consistenza

Swift si basa su una struttura di archiviazione distribuita, in cui i dati sono replicati su più nodi di archiviazione per garantirne la disponibilità e la tolleranza ai guasti. In quanto tale, deve fare i conti con il **teorema CAP**.

Tra consistenza, disponibilità e tolleranza alla partizione è possibile garantirne solo 2 su 3 contemporaneamente.

Quella che si lascia indietro è la consistenza perché le altre due sono fondamentali per i sistemi di oggi. Ovviamente, non è che viene abbandonata. Si cerca di offrire l'**eventual consistency**, ovvero un modello di consistenza dei dati utilizzato nei sistemi distribuiti, in cui i dati possono non essere immediatamente coerenti in tutte le repliche del sistema.

In pratica, quando un dato viene aggiornato in una replica del sistema, non viene immediatamente propagato a tutte le altre repliche, ma viene invece propagato gradualmente nel tempo.

Ciò significa che in un dato momento, le diverse repliche possono contenere copie del dato con valori diversi, ma eventualmente i valori convergeranno verso uno stato coerente. Quindi, **Swift può raramente offrire dati non aggiornati**.

Ci sono dei server che fanno monitoraggio della rete e sulla base di quello che osservano prendono provvedimenti pensati per ottimizzare la consistenza.

Swift utilizza anche dei processi in background per gestire la consistenza.

Auditor: scansionano continuamente le partizioni per vedere se funzionano.

Se una partizione non funziona allora si vanno a replicare i dati al suo interno in un'altra partizione.

Updater: verificano che gli elenchi di account, container e object siano corretti. Verificano che i puntatori siano corretti.

Replicator: replicano i dati. La politica di default è di tre repliche per dato.

Per garantire la disponibilità e la partizione le copie di uno stesso dato non devono essere inserite all'interno dello stesso nodo.

Per questo motivo in Swift è stato introdotto il concetto di **zona**.

Per zona si intende una porzione del cluster fisicamente separata dalle altre.

In questo modo se c'è un problema in una zona esso non si ripercuote sulle altre.

Swift fa uso del Consistent Hashing.

Il Consistent Hashing è un algoritmo di hashing distribuito che consente di assegnare in modo uniforme le richieste di lavoro ai nodi di un cluster, garantendo al contempo una buona scalabilità e tolleranza ai guasti del sistema. L'algoritmo funziona utilizzando un anello di hash, in cui ogni nodo del cluster è rappresentato da un punto sull'anello.

Per assegnare una richiesta a un nodo, l'algoritmo di Consistent Hashing applica una funzione di hash sulla chiave della richiesta, che restituisce un valore che viene poi mappato sull'anello.

La richiesta viene quindi assegnata al nodo il cui punto sull'anello segue il valore restituito dalla funzione di hash, nella direzione oraria.

In questo modo, il Consistent Hashing garantisce che ogni richiesta venga assegnata in modo coerente al nodo corretto, anche in presenza di guasti o di nuovi nodi che vengono aggiunti al cluster. Inoltre, poiché ogni richiesta viene assegnata a un nodo specifico, l'algoritmo consente di implementare la cache distribuita in modo efficace, poiché ogni nodo può mantenere una cache per le richieste che riceve regolarmente.

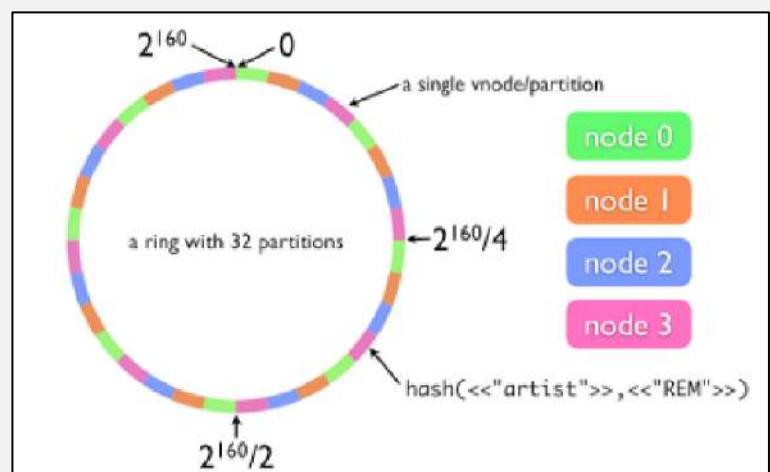
Infine, il Consistent Hashing consente anche di bilanciare il carico del sistema, poiché i nodi vengono distribuiti uniformemente sull'anello di hash, in modo che il carico di lavoro sia distribuito in modo equo tra tutti i nodi del cluster.

Per ogni account, container e object si ha un ring. Per ogni oggetto si usa un hash (mappaggio pseudo-casuale) di tale oggetto. La numerazione è ciclica. 2^{160} combinazioni sull'anello.

A questo punto si spezza l'anello in fette e si associa ai nodi di storage.

Ovviamente è impensabile interrompere il servizio per ricalcolare l'hash ogni volta.

Per evitare ciò si fanno molte più fette di quanti sono i nodi. Quando un nodo si aggiunge alla lista, gli vengono assegnate fette di altri nodi, senza interrompere il servizio e in modo sicuro.



Capitolo 6: OpenStack Orchestration

Ceilometer è un servizio, opzionale, di telemetria e **monitoraggio** di OpenStack che raccoglie dati sulle risorse della Cloud come le istanze, i volumi e le reti. In particolare, **Ceilometer monitora l'utilizzo delle risorse**, i tempi di attività, la capacità di archiviazione e la larghezza di banda della rete. Questi dati possono essere utilizzati per creare report, fatturazione e per garantire conformità delle politiche di sicurezza.

I **compute agent** vengono eseguiti su ogni nodo compute, dove girano le VM. Eseguono il polling per le statistiche sull'utilizzo delle risorse.

I **central agent** vengono eseguiti nei nodi controller.

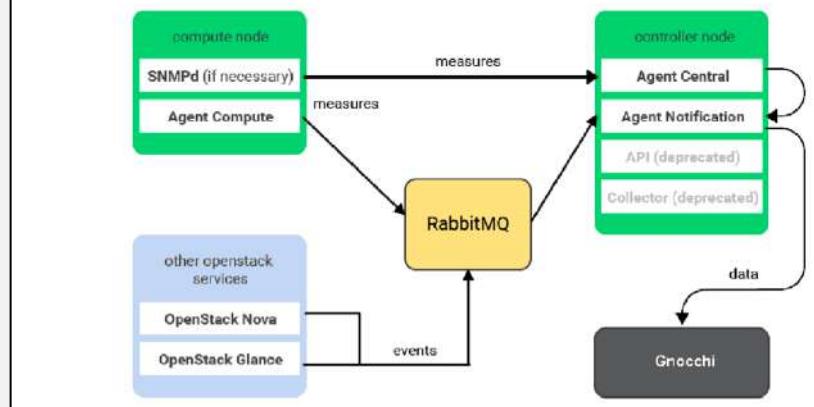
Il **notification agent** viene utilizzato per lo scambio di informazioni.

Le informazioni vengono scambiate con il meccanismo interno di OpenStack, ovvero RabbitMQ, e alla fine vengono depositate in Gnocchi, un database temporale dove le informazioni vengono associate a dei timestamp automatici.

Nel nodo controller abbiamo l'agente centrale e l'agente di notifica mentre nei nodi compute c'è l'agente compute.

Nella figura sono anche presentati degli elementi che sono stati deprecati. -----> Ceilometer una volta acquisite le informazioni, le utilizza per triggerare degli eventi.

Ceilometer Architecture



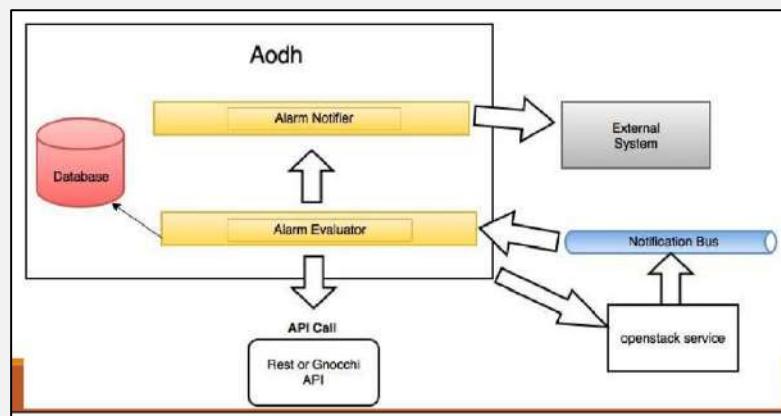
Aodh è un servizio, opzionale, di **allarme** il cui compito è decidere se, come e quando scatenare degli allarmi in base alle informazioni di cui dispone. Gli allarmi vengono associati al superamento di determinate **soglie**.

Aodh è formato da diverse componenti che lavorano insieme.

API server: fornisce l'accesso al servizio stesso e alle informazioni immagazzinate nel proprio database.

Alarm Evaluator: stabilisce quando l'allarme si accende, cioè determina il superamento della soglia impostata. Di default, valuta se lanciare gli allarmi una volta al minuto. **SE**.

Notification listener: determina quando scatenare gli allarmi. **QUANDO**.



Alarm Notifier: stabilisce la notifica degli allarmi. COME.

Tutto questo mette a disposizione per gli utenti un servizio di Monitoring as a Service. Gli allarmi possono trovarsi in **tre possibili stati**:

- **OK**: non ci sono allarmi, il sistema è in salute.
- **ALARM**: viene lanciato l'allarme, si è superata una certa soglia.
- **INSUFFICIENT DATA**: non ci sono abbastanza dati dentro Gnocchi per decidere, in base alla politica predefinita, se lanciare un allarme oppure no.

Come definire un allarme: bisogna specificare un valore di soglia in base a cosa si va a monitorare, ad esempio lo stato della CPU, lo stato di occupazione della memoria, etc. e definire una politica di quando la metrica dev'essere valutata. La sliding window è la politica più popolare e consiste nell'utilizzare una finestra temporale per indicare quanto indietro nel passato (recente) si vuole guardare.

Monitorare costantemente è una pratica sconveniente, in quanto potrebbe essere troppo oneroso.

Esempio: generazione di un allarme per Gnocchi al superamento di una soglia di utilizzo della CPU da parte di una particolare istanza.

Soglia impostata al 70%, oltre si passa allo stato ALARM.

Il periodo di valutazione è di 10 minuti per 3 volte.

```
$ aodh alarm create \
--name cpu_hi \
--type gnocchi_resources_threshold \
--description 'instance running hot' \
--metric cpu_util \
--threshold 70.0 \
--comparison-operator gt \
--aggregation-method mean \
--granularity 600 \
--evaluation-periods 3 \
--alarm-action 'log://' \
--resource-id INSTANCE_ID \
--resource-type instance
```

Heat è un servizio, opzionale, per l'**orchestrazione** dell'infrastruttura Cloud attraverso la definizione di template che semplificano la creazione, la gestione e la modifica di risorse come macchine virtuali, reti, volumi di archiviazione e servizi. **Permette di coordinare la creazione e la gestione di tali risorse in modo automatizzato**, riducendo il rischio di errori umani e garantendo una maggiore coerenza e prevedibilità nella gestione dell'infrastruttura Cloud.

Bisogna configurarlo con un file yaml.

Heat utilizza il concetto di **Stack**.

Uno Stack rappresenta **tutte le risorse necessarie** per distribuire un'applicazione.

Può essere semplice come un'unica istanza e le sue risorse o complesso come più istanze con tutte le dipendenze delle risorse che compongono un'applicazione multilivello. Queste risorse possono essere istanze, router, interfacce, reti, floating IP, tutto ciò che deve essere creato esplicitamente e insieme concorrono ad un servizio richiesto da un tenant.

Heat mette in opera un insieme di risorse offerte poi ad un utente.

Istanzia un'istanza.

Script più facile che possiamo
creare ----->

```
heat_template_version: 2015-04-30
```

```
description: Simple template to deploy a single compute instance
```

```
resources:
```

```
my_instance:
```

```
  type: OS::Nova::Server
```

```
  properties:
```

```
    key_name: my_key
```

```
    image: cirros
```

```
    flavor: m1.nano
```

Riassumendo: Ceilometer, Aodh e Heat sono tre servizi opzionali che possono lavorare insieme per fornire funzionalità di monitoraggio, allarme e orchestrazione all'interno dell'infrastruttura Cloud OpenStack.

Capitolo 7: Markov Models

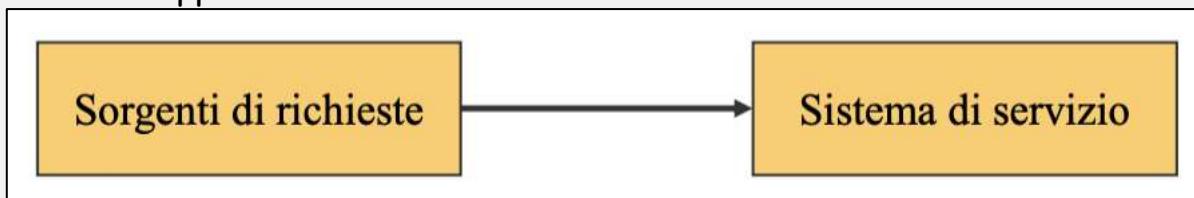
L'obiettivo è **analizzare quantitativamente le prestazioni di un sistema dinamico**.

Valuteremo l'efficienza, il fattore di utilizzazione, il ritardo nel ricevere un servizio, possibili perdite delle richieste di servizio e i tempi di inattività.

Sono dei numeri che rappresentano le prestazioni di un sistema.

Alla fine del capitolo occorre tirare fuori dei numeri.

Per valutare quantitativamente le prestazioni di un sistema dinamico è necessario rappresentare in modo astratto le sue funzionalità.



Possiamo vedere il modello come: Chi Chiede → Chi Eroga.

Nel nostro caso, chi chiede sono i client, chi eroga sono i server.

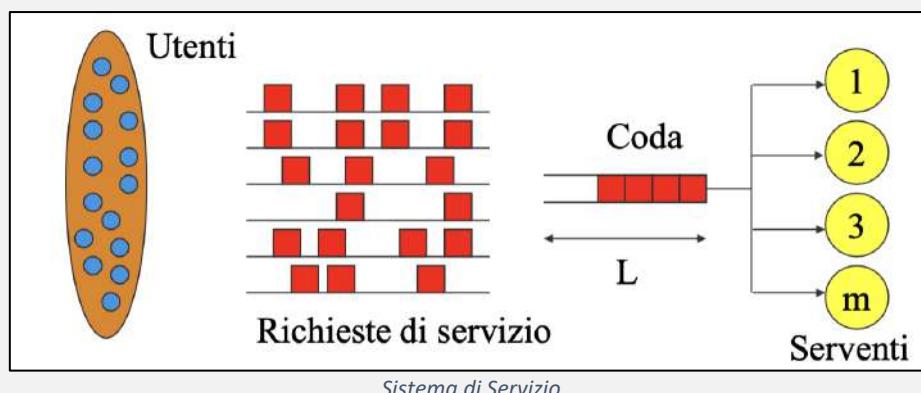
A noi interessa la coda di questi processi.

Distinguiamo due sistemi:

- **Sistemi a perdita**: sistemi che modellano delle situazioni dove non è possibile aspettare. Di conseguenza, se non ci sono **serventi** disponibili, la richiesta viene scartata direttamente.
- **Sistemi ad attesa**: sistemi che permettono di accodare le richieste ed aspettare che un servente si liberi per soddisfarle.

Nei sistemi a perdita ci interessa il carico smaltito, la frequenza con cui le richieste vengono rifiutate e la durata dei periodi di congestione.

Nei sistemi ad attesa ci interessa sapere per quanto tempo gli utenti restano in attesa, per quanto tempo permangono gli utenti all'interno del sistema e il numero di richieste in attesa. Se si va ad esplodere il modello sopra vediamo le componenti che occorre modellare matematicamente con sistemi analitici.



Entrambi i sistemi sono **sistemi a coda**.

Si parte dal **numero di utenti** in grado di richiedere un servizio.

Se il numero è molto elevato allora si parla di limite perché non ha senso andare ad analizzare il dettaglio quando le richieste sono moltissime.

Se, invece, il numero è molto basso allora è importante sapere in quanti stanno già utilizzando il servizio e in quanti invece potrebbero ancora richiederlo.

La differenza, in questo caso, si apprezza.

Consideriamo la **cardinalità** della popolazione infinita quando l'eventuale presenza di utenti nel sistema a coda altera significativamente il processo di arrivo delle richieste verso i serventi.

Ci sono poi le **richieste di servizio** da parte di ogni singolo utente che possono avere diverse evoluzioni con il passare del tempo.

Il loro insieme, ovvero tutte le combinazioni, costituisce un processo aleatorio che è possibile modellare matematicamente.

C'è poi la **coda** che se trova dei serventi liberi allora li contatta.

Potrebbe essere la RAM del router o la sala d'attesa dal medico di base.

Se la probabilità che arrivando una richiesta essa trovi sia la coda che i serventi tutti occupati è bassa, allora diventa inutile considerarla e di conseguenza si imposta un limite ad **infinito**.

Se, invece, la probabilità d'attesa è alta allora la coda è finita e di conseguenza la si modella con un numero **finito L**.

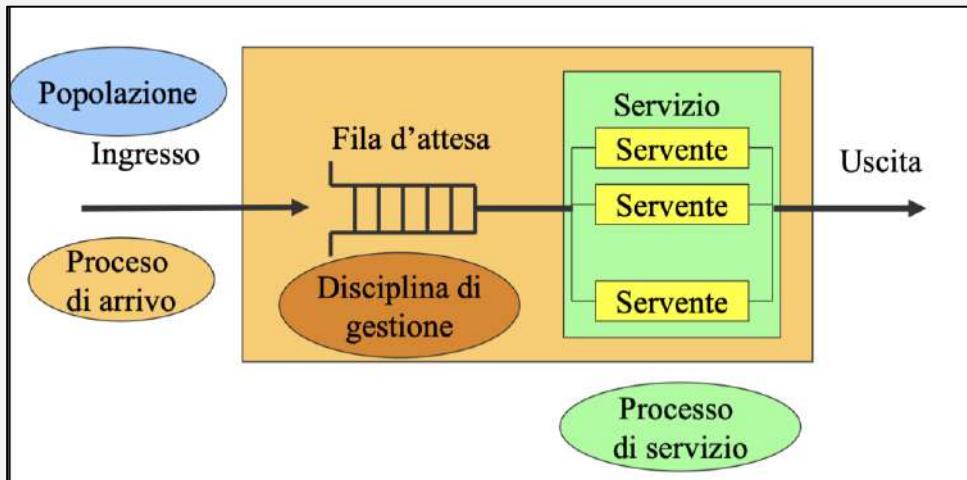
Occorre anche capire che **politica di gestione utilizzare per la coda**, se con priorità, first in first out, meno tempo di servizio, ecc.

Infine, ci sono i **serventi**. Se si considera una Cloud Amazon allora il numero dei serventi m tende ad **infinito**. Se, invece, occorre modellare una Cloud in un laboratorio allora la m dei serventi è un numero **finito**.

Risoluzione delle contese di utilizzazione:

- Sistema a perdita pura ($L=0$).
- Sistema orientato alla perdita (L piccolo).
- Sistema orientato al ritardo (L grande).
- Sistema a ritardo senza perdita ($L \rightarrow \infty$ oppure $L \geq$ utenti).

Indichiamo con $L + m$ la capacità del sistema di gestire gli utenti prima di scartarli inevitabilmente.



Riassunto degli elementi descrittivi di un sistema a coda

Cardinalità della popolazione:

- finita, infinita

Processo di Arrivo :

- frequenza media, varianza ...

Accodamento:

- dimensione della coda:

 - finita, infinita

- numero di code

Disciplina di gestione, o Selezione:

- discipline di coda
 - primo arrivato primo servito (FIFO)
 - shortest job first (SJF)
 - ...
- classi di priorità

Numero dei serventi

- $n = \text{cardinalità della popolazione}$
- $L = \text{dimensione della coda}$
- $m = \text{numero di serventi}$
- $C = m + L = \text{capacità del sistema}$

Se $n \leq C$ e $L > 0 \rightarrow$ sistema ad attesa (senza perdita). Es: Cloud pubblica.

Se $n > C$ e $L > 0 \rightarrow$ sistema a perdita (con attesa). Es: Cloud privata.

Se $n > C$ e $L = 0 \rightarrow$ sistema a perdita pura (senza attesa).

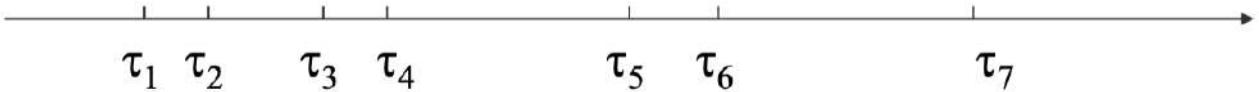
Come parametro prestazionale abbiamo il

$$\text{Tempo di sistema (s)} = \text{tempo di attesa (w)} + \text{tempo di servizio (x)}$$

Caratterizzazione della domanda

La domanda è caratterizzata da richieste presentate dagli utenti del sistema.

- consideriamo una sequenza di istanti di richiesta di servizio (τ_i)



Gli istanti tau sono variabili completamente aleatorie.

Conviene lasciare la variabile del tempo come continua perché possono verificarsi degli eventi in qualsiasi momento.

La conoscenza esatta del tempo di arrivo di una richiesta di servizio non è così fondamentale. Invece, è più utile sapere quanto frequentemente arrivano le richieste, ovvero sapere quando arriva τ_2 dopo che è arrivata la richiesta τ_1 .

Indichiamo con il termine **tempo di interarrivo** l'intervallo che intercorre tra l'istante di presentazione di una richiesta $i - 1$ -esima e quello della richiesta i esima. Il tempo che passa tra il tempo di arrivo della richiesta τ_{i-1} e τ_i .

$$f_T(t_i) = \lambda l_i^{-\lambda t} \quad t_i = \tau_i - \tau_{i-1} \text{ con } i = 1, 2, \dots$$
$$f_T(t_i) = 1 - l_i^{-\lambda t} \quad \text{con } t \geq 0$$

Tempo di servizio

Indichiamo con L_i il servizio, attività, o 'lavoro', in temini quantitativi, che i serventi del sistema devono erogare per soddisfare la richiesta i -esima.

Definiamo l' i -esimo tempo di servizio x_i come **l'intervallo di tempo che un servente deve dedicare per soddisfare la richiesta i -esima**.

Se supponiamo che la capacità C di ogni servente del sistema di soddisfare le richieste sia identica allora per servire la richiesta i -esima sarà necessario un tempo uguale a:

$$x_i = \frac{L_i}{\Gamma} \quad \text{con } i = 1, 2, \dots$$

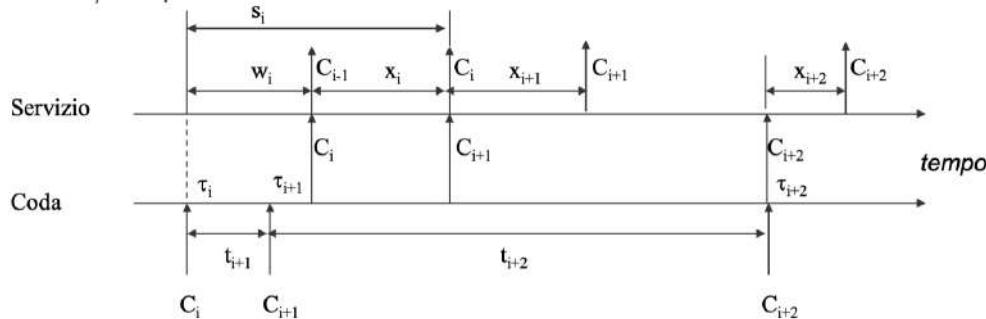
Quindi x_i è una variabile aleatoria temporale che modella il tempo necessario per soddisfare la richiesta del cliente.

Quindi, il tempo che il servente impiega per soddisfare la richiesta.

L_i potrebbe essere, ad esempio, il numero di bit di un pacchetto.

Diagramma temporale del sistema

- C_i : *i-esima* richiesta di servizio (o cliente) ad entrare nel sistema
- $N(t)$: numero di richieste di servizio nel sistema al tempo t
- $U(t)$: attività che resta da svolgere al tempo t
- Grandezze temporali:
 - τ_i : tempo di arrivo *i-esima* richiesta di servizio
 - t_i : tempo di interarrivo tra la richiesta *(i-1)-esima* e la *i-esima*
 - w_i : tempo di attesa in coda della *i-esima* richiesta di servizio
 - x_i : tempo di servizio dell'*i-esima* richiesta



Supponiamo che ci sia già un servente in lavoro prima dell'asse del tempo in foto. Arriva C_i , ovvero la richiesta di servizio del customer i . Non può andare direttamente al servente in quanto è occupato. Passa del tempo w_i (wait) ovvero il tempo di attesa per trovare il servente disponibile.

Quando il servente si libera, la richiesta di servizio C_i viene processata in un tempo x_i . Nel mentre entra il customer C_{i+1} che si inserisce in coda dopo di C_i .

Quando C_i esce dal sistema perché la sua richiesta è conclusa, entra C_{i+1} .

Nel frattempo, non è successo nulla. Di conseguenza, quando C_{i+1} esce dal sistema il servente diventa libero e resta in attesa che arrivi qualcun altro.

È importante che questo scenario si verifichi perché indica che la probabilità di svuotamento del sistema è > 0 , cosa fondamentale se un sistema vuole essere considerato stabile. Altrimenti, significherebbe che un sistema è sempre pieno di richieste e che non riesce mai a soddisfare tutte.

Le richieste che arrivano dovranno essere a un certo punto scartate.

Dunque, è fondamentale che il sistema si svuoti di tanto in tanto.

Quando arriva C_{i+2} , entra subito perché il servente è libero.

La sua richiesta viene soddisfatta in un tempo x_{i+2} e poi esce dal sistema.

Processi di ingresso e di servizio

La sequenza dei tempi di interarrivo $\{t_i\}$ e quella dei tempi di servizio $\{x_i\}$ costituiscono delle realizzazioni di due processi stocastici:

- il processo di ingresso
- il processo di servizio

Ogni valore t_i e x_i è una realizzazione di una variabile aleatoria

Normalmente si suppone i due processi siano stazionari, almeno WSS, e statisticamente indipendenti

Grandezze limite

Comportamento al limite delle variabili aleatorie:

- tempi di interrivo

$$\tilde{t} = \lim_{n \rightarrow \infty} t_n$$

$$P[t_n \leq t] = A_n(t) \xrightarrow{n \rightarrow \infty} P[\tilde{t} \leq t] = A(t)$$

$$E[t_n] = \bar{t}_n \xrightarrow{n \rightarrow \infty} E[\tilde{t}] = \bar{t} = \frac{1}{\lambda}$$

Analogamente si possono definire le stesse grandezze per

- i tempi di attesa

$$\tilde{w} = \lim_{n \rightarrow \infty} w_n, \quad E[\tilde{w}] = \bar{w} = W$$

- i tempi di servizio

$$\tilde{x} = \lim_{n \rightarrow \infty} x_n, \quad E[\tilde{x}] = \bar{x} = \frac{1}{\mu}$$

- i tempi di sistema

$$\tilde{s} = \lim_{n \rightarrow \infty} s_n, \quad E[\tilde{s}] = \bar{s} = T$$

\tilde{t} non è il risultato del limite ma una variabile aleatoria.

Si presume che col passare di un tempo sufficiente lungo, il sistema vada in una forma di regime permanente.

P è la funzione di distribuzione della probabilità della variabile aleatoria, il tempo di interattivo a regime.

E è il tempo medio di interarrivo ed equivale a $1/\lambda$.
 λ nel suo valore medio.

μ è la frequenza media di servizio. λ frequenza media di interarrivo.

Caratterizzazione di un sistema a coda

Una coda è definita da:

- | | |
|----------------------------------|---------------------------|
| ◦ processo degli arrivi (D.d.p.) | $A(t)$ |
| ◦ tempi di servizio (D.d.p.) | $B(t)$ |
| ◦ numero di serventi | m |
| ◦ dimensione del sistema | L Dimensione della coda |
| ◦ cardinalità della popolazione | n |
| ◦ disciplina di servizio | |

Notazione sintetica di Kendall ($A/B/m/L/n$)

- A e B posono assumere i valori:
 - M esponenziale negativa o "Markoviana"
 - D deterministica o costante
 - E_i erlangiana con i stadi
 - H_i iper-esponenziale con i stadi
 - G generale

La notazione di Kendall serve a caratterizzare un sistema a coda.

I primi tre sono la base minima per un sistema.

La disciplina di servizio di default che si utilizza è la FIFO.

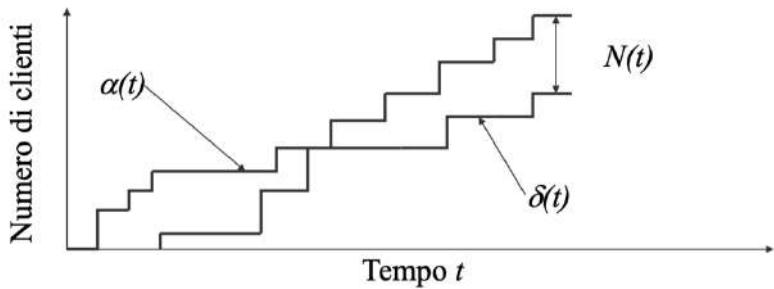
m, L ed n sono delle costanti e sono opzionali.

Quando non vengono indicate, le si considera con un valore di infinito.

Legge di Little

Ipotesi: sistema senza perdita

- $\alpha(t)$: numero di arrivi in $[0,t]$
- $\delta(t)$: numero di partenze in $[0,t]$
- $N(t) = \alpha(t) - \delta(t)$
- $\gamma(t)$: l'area tra le due curve rappresenta il tempo che tutti i clienti hanno trascorso nel sistema nell'intervallo $[0,t]$



È una legge che vale per i sistemi senza perdita ed è completamente generica poiché non fa nessun ipotesi sul processo degli arrivi e sul processo di servizio.

Quando i due gradini coincidono significa che il sistema si è svuotato.

$N(t)$ corrisponde al numero di utenti presenti all'interno del sistema.

Arrivi - partenze = dentro.

Legge di Little

Definiamo:

- frequenza media di interarrivo in $[0,t]$: $\lambda_t = \alpha(t)/t$ (a)
- tempo medio di sistema per ogni utente in $[0,t]$: $T_t = \gamma(t)/\alpha(t)$
- numero medio di clienti nel sistema in $[0,t]$: $N_t = \gamma(t)/t$

(b)

Quindi si ottiene:

- $N_t = \gamma(t)/t$; (dalla a) $N_t = \gamma(t) \cdot \lambda_t / \alpha(t)$; (dalla b) $N_t = T_t * \lambda_t$
- assumendo che esistano per il sistema i valori limite:

- si ottiene:

$$\lambda = \lim_{t \rightarrow \infty} \lambda_t, \quad T = \lim_{t \rightarrow \infty} T_t$$

$$\bar{N} = \lambda T = \lambda W + \lambda \bar{x} = \bar{N}_q + \bar{N}_s$$

Risultato indipendente da A(t), B(t) e m

IPOTESI: ASSENZA DI PERDITE!

Si suppone che il sistema sia chiuso, una volta che l'utente entra può uscire solo dopo aver completato la sua richiesta.

Il numero medio di utenti nel sistema è uguale al tempo medio di permanenza nel sistema per la frequenza media di interarrivo nel sistema.

Se si utilizzano i valori limite di queste variabili, e quindi si osservano valori limite, si ottiene che il numero medio di utenti nel sistema è uguale alla frequenza media di arrivo nel sistema per il tempo medio di permanenza nel sistema. Ovvero:

$$\bar{N} = \lambda T \text{ con tutti e tre i valori medi}$$

Fattore di utilizzazione dei serventi: definito come il rapporto tra il ritmo (frequenza) con cui il "lavoro" entra nel sistema e quello con il quale il sistema riesce a smaltirlo. Se si assume che il tasso di servizio dei serventi sia indipendente da qualsiasi altro parametro del sistema allora si può scrivere che

- caso 1 servente: $\rho = \lambda / \mu = \lambda \bar{x}$
- caso m serventi: $\rho = \lambda / (m\mu) = \lambda \bar{x} / m$

Nel caso di 1 solo servente, se $\rho = 1$ allora siamo nel caso d'instabilità.

$\frac{\lambda}{\mu}$, ossia la frequenza con cui il lavoro entra nel sistema normalizzata alla capacità del singolo servente, rappresenta l'intensità di traffico.

Viene espressa in **Erlang**.

1 Erlang è inteso come un servizio continuativo, rapportato alla capacità del singolo servente. Di conseguenza se il servente eroga servizio, è occupato.

M Erlang è inteso come M serventi che erogano servizio e sono quindi occupati.

Fattore di utilizzazione

Se $0 \leq \rho < 1$:

- può essere interpretato come:

frazione di serventi occupati

- rappresenta la condizione di stabilità del sistema
- Infatti, nel caso G/G/m, risulta quanto segue:
 - dato un intervallo τ sufficientemente grande, con probabilità tendente a 1 il numero di arrivi sarà pari a $\lambda \tau$
 - quindi un servente sarà occupato per un tempo pari a $\tau(1-p_0)$, definendo p_0 come la probabilità di trovare il server libero in un generico istante di tempo
 - il numero di utenti serviti in tale intervallo sarà $m(\tau - \tau p_0)\mu$
 - eguagliando il numero di arrivi con il numero di utenti serviti, si ottiene:

$$\lambda \tau \cong m(\tau - \tau p_0)\mu \xrightarrow{\tau \rightarrow \infty} \rho = 1 - p_0$$

$\tau(1 - p_0)$ è il tempo in cui il servente lavora, ovvero 1 - la probabilità che il servente sia libero (assumendo che il sistema sia stabile).

Quindi p_0 rappresenta la stabilità del sistema. **Dev'essere per forza maggiore di zero**. Questo perché se il sistema come assunto è stabile, allora **ogni tanto deve potersi svuotare**.

Concetto di stato: in questo corso lo **STATO** è il numero di utenti presenti nel sistema. **Non ci interessa conoscere la storia passata** perché non influenza. L'informazione che ci interessa, a prescindere da quanto essa sia vecchia, è la più recente possibile.

La probabilità che il sistema si trovi in un generico stato j al tempo t è uguale a:

$$\Pi_j(t) = P[X(t)=j] \quad \text{J state i}$$

Sia la variabile t sia lo stato $X(t)$ possono assumere valori in un insieme continuo o in un insieme discreto (finito o numerabile)

Noi considereremo processi **continui** nel tempo e **discreti** nelle realizzazioni

Chiaramente $\sum_{j=0,1,\dots} \Pi_j(t) = 1$

Catene

La somma delle probabilità che il sistema al tempo t si trovi in un generico stato $0, 1, \dots$ è uguale a 1.

discreto e

Un processo aleatorio è detto Catena d
gode della proprietà di Markov

Proprietà di Markov:

- dato un insieme di variabili aleatorie $\{X_n\}$, questo forma una catena di Markov se la probabilità di trovarsi in un tempo futuro in un determinato stato può essere espressa in funzione solo dello stato assunto al tempo corrente e non occorre specificare quali stati sono stati assunti in precedenza;
- lo stato attuale riassume tutta la storia del sistema
- la conoscenza più recente dello stato del sistema rende inutile la conoscenza degli stati assunti in precedenza
- La conoscenza del passato non ci consente di predire quanto tempo il processo debba rimanere nello stato in cui si trova
 - la distribuzione del tempo che il processo rimane in uno stato è "senza memoria", e nell'ipotesi d tempo continuo quest porta alla **distribuzione esponenziale** del tempo di permanenza nello stato.

Se riusciamo a modellare un sistema reale attraverso Markov allora per determinare il sistema stesso ci occorre solo la conoscenza più recente.

Formalmente:

- il processo aleatorio $X(t)$ forma una catena di Markov tempo-continua se per tutti gli interi n e per una sequenza di istanti temporali $t_1 < t_2 < \dots < t_n < t_{n+1}$ risulta

$$P[X(t_{n+1})=j | X(t_n)=i_n, X(t_{n-1})=i_{n-1}, \dots, X(t_1)=i_1] = P[X(t_{n+1})=j | X(t_n)=i_n]$$

Classificazione:

i.i.d.

- Catene di Markov tempo continuo:
 - distribuzione del tempo di stato esponenziale $p_T(t) = \lambda e^{-\lambda t}$, $t \geq 0$
- Catene di Markov tempo discreto
 - distribuzione del tempo di stato geometrica $p_N(n) = (1-p)^{n-1}p$, $n \geq 0$

La distribuzione del tempo di stato è il tempo in cui un sistema resta in un determinato stato senza cambiare.

Lo stato è la j , il numero di utenti, la previsione degli utenti nel sistema.

$p_N(n) = (1 - p)^{n-1} p$ è la probabilità che non si verifichi una transizione di stato fino ad $n-1$ per la probabilità che la transizione si verifica con probabilità p .

Questa è una proprietà di "mancanza di memoria" associata alla distribuzione del tempo che un processo di Catena di Markov rimane in uno stato.

Quando si dice che la distribuzione del tempo di permanenza nello stato è "senza memoria", significa che **la probabilità che il sistema rimanga in uno stato per un certo periodo di tempo non dipende dalla durata del tempo trascorso fino a quel momento nello stato**. In altre parole, **la probabilità di uscire da uno stato non dipende da quanto tempo il sistema abbia già trascorso nello stato**.

Nell'ipotesi di tempo continuo, questa mancanza di memoria porta alla **distribuzione esponenziale** del tempo di permanenza nello stato.

La distribuzione esponenziale è una distribuzione di probabilità continua che descrive il tempo tra eventi indipendenti di un processo di Poisson.

Nel contesto delle Catene di Markov, il tempo che il processo rimane in uno stato segue una distribuzione esponenziale, il che significa che la probabilità di uscire da uno stato è costante nel tempo e non dipende dalla durata trascorsa nello stato.

In sostanza, **la proprietà di mancanza di memoria implica che la distribuzione del tempo di permanenza nello stato è esponenziale, il che fornisce un modo per modellare e calcolare la probabilità di transizione tra gli stati in un processo di Catena di Markov**.

Probabilità di transizione da uno stato $i \rightarrow j$:

$$p_{ij}(s,t) = P[X(t) = j | X(s) = i] \text{ per } t \geq s$$

È la probabilità che al tempo t il sistema si trovi nello stato j dato che l'evento condizionante al tempo s si trova nello stato i . Tempo t temporalmente dopo s . Con $t = s$ e $i \neq j$ la probabilità è 0. Con $t = s$ e $i = j$ la probabilità è 1.

Per passare dallo stato i con istante s , allo stato j con istante $t > s$, il processo dovrà passare per uno stato intermedio k ad un certo istante intermedio u .

$$\begin{aligned} p_{ij}(s,t) &= \sum_k P[X(t) = j | X(u) = k, X(s) = i] = \\ &= \sum_k P[X(u) = k | X(s) = i] P[X(t) = j | X(s) = i, X(u) = k] = \\ &= \sum_k p_{ik}(s,u) p_{kj}(u,t) \end{aligned}$$

Equazioni di Chapman-Kolmogorov

$$\sum_B P(A, B | C) = P(A | C) \quad P(A | B, C) P(B | C) = \frac{P(A, B, C)}{P(B, C)} \frac{P(B, C)}{P(C)} = P(A, B | C)$$

Queste espressioni sono due probabilità di transizione di stato:

- 1) Passare da i a k dal tempo s al tempo u .
- 2) Passare da k a j dal tempo u al tempo t .

Abbiamo ricavato le **equazioni di Chapman-Kolmogorov**.

Il risultato è un prodotto matriciale riga per colonna.

Possiamo costruire due matrici: una con la transizione da s a u e l'altra con transizioni da u a t . Successivamente si va a fare il prodotto.

$$\begin{aligned} p_{ij}(s,t) &= \sum_k p_{ik}(s,u) p_{kj}(u,t) = \\ &= \begin{bmatrix} p_{11}(s,u) & p_{12}(s,u) & \dots & p_{1k}(s,u) & \dots \\ p_{21}(s,u) & p_{22}(s,u) & \dots & p_{2k}(s,u) & \dots \\ \vdots & \vdots & & \vdots & \vdots \\ p_{i1}(s,u) & p_{i2}(s,u) & \dots & p_{ik}(s,u) & \dots \end{bmatrix} \begin{bmatrix} p_{11}(u,t) & p_{12}(u,t) & \dots & p_{1j}(u,t) & \dots \\ p_{21}(u,t) & p_{22}(u,t) & \dots & p_{2j}(u,t) & \dots \\ \vdots & \vdots & & \vdots & \vdots \\ p_{i1}(u,t) & p_{i2}(u,t) & \dots & p_{ij}(u,t) & \dots \end{bmatrix} \end{aligned}$$

$$H(s,t) = H(s,u) H(u,t)$$

Equazioni di C-K in forma matriciale

$$H(s,t) = \begin{bmatrix} p_{11}(s,t) & p_{12}(s,t) & \dots & p_{1k}(s,t) & \dots \\ p_{21}(s,t) & p_{22}(s,t) & \dots & p_{2k}(s,t) & \dots \\ \vdots & \vdots & & \vdots & \vdots \\ p_{i1}(s,t) & p_{i2}(s,t) & \dots & p_{ik}(s,t) & \dots \end{bmatrix}$$

In forma matriciale diventa:

$$H(s,t) = [p_{ij}(s,t)] = H(s,u)H(u,t), \text{ con } H(t,t) = I$$

definendo $P(t) = H(t, t + \Delta t) = [p_{ij}(t, t + \Delta t)]$ si ottiene:

$$\begin{aligned} H(s,t) - H(s,t - \Delta t) &= H(s,t - \Delta t)H(t - \Delta t, t) - H(s,t - \Delta t) = \\ &= H(s,t - \Delta t)P(t - \Delta t) - H(s,t - \Delta t) = H(s,t - \Delta t)(P(t - \Delta t) - I) \end{aligned}$$

dividendo per Δt e facendo il limite tendente a zero:

$$\lim_{\Delta t \rightarrow 0} \frac{H(s,t) - H(s,t - \Delta t)}{\Delta t} =$$

$$\frac{\partial H(s,t)}{\partial t} = \lim_{\Delta t \rightarrow 0} \left(H(s,t - \Delta t) \frac{(P(t - \Delta t) - I)}{\Delta t} \right) = H(s,t)Q(t)$$

$$Q(t) = \lim_{\Delta t \rightarrow 0} \frac{P(t - \Delta t) - I}{\Delta t} = [q_{ij}(t)], \text{ con } q_{ij}(t) = \begin{cases} \lim_{\Delta t \rightarrow 0} \frac{p_{ii}(t - \Delta t, t) - 1}{\Delta t} & i = j \\ \lim_{\Delta t \rightarrow 0} \frac{p_{ij}(t - \Delta t, t)}{\Delta t} & i \neq j \end{cases}$$

$H(t,t)$ è la matrice nello stesso stato. Non c'è un tempo di cambiamento, quindi, è una matrice d'identità con tutti 1 nelle diagonali e 0 altrove.

$P(t)$ è la matrice che contiene tutte le probabilità di transizione da t a $t + \Delta t$.

L'istante intermedio u è $t - \Delta t$. Applichiamo la prima formula di Kolmogorov. Nella seconda riga andiamo ad utilizzare la seconda relazione, quella con $P(t)$, sostituendola ad $H(t - \Delta t, t)$.

Abbiamo un rapporto incrementale, diviso per Δt e con il limite che tende a 0.

Stiamo facendo la derivata rispetto alla variabile temporale t .

Per definizione, questa quantità è la derivata parziale rispetto a t di $H(s,t)$.

$$\frac{\partial H(s,t)}{\partial t} = H(s,t)Q(t)$$

Equazioni di C-K in forma differenziale

$$Q(t) = [q_{ij}(t)], \quad q_{ij}(t) = \begin{cases} \lim_{\Delta t \rightarrow 0} \frac{p_{ii}(t - \Delta t, t) - 1}{\Delta t} & i = j \\ \lim_{\Delta t \rightarrow 0} \frac{p_{ij}(t - \Delta t, t)}{\Delta t} & i \neq j \end{cases}$$

Q è nota come **generatore infinitesimale** or **rate matrix** del processo di Markov.

È evidente che

$$\sum_j q_{ij}(t) = 0 \quad \Rightarrow q_{ii}(t) = - \sum_{j \neq i} q_{ij}(t) < 0$$

Il -1 dipende dalla matrice di identità. Nel primo caso c'è, nel secondo caso no. Tutti gli elementi presenti sono positivi in quanto le probabilità vanno da 0 a 1, mentre Δt è un tempo quindi dev'essere positivo.

q_{ij} dipende dalle probabilità di transizione tra gli stati.

Le probabilità di transizione aumentano se le frequenze di transizione tra gli stati aumentano. Se un sistema a coda transita dallo stato 3 allo stato 5 significa che ci sono stati due eventi (2 nuovi utenti nel sistema a coda).

Maggiore è la frequenza di arrivo degli utenti maggiore sarà la probabilità di transizione. Se un sistema passa dallo stato 7 allo stato 6 vuol dire che un utente è stato servito e che lo stato transita verso un valore più basso.

Una volta nello stato i da qualche parte occorre andare:

restare nello stesso stato, andare avanti, andare indietro.

La cosa importante è che la somma delle probabilità di transizione degli eventi deve fare 1. Sommando rispetto a j tutte le quantità q_{ij} si sta sommando tutte le probabilità di transizione, quindi il totale è 1.

Prendendo la matrice Q , la somma di tutti i q_{ij} per ogni riga da come risultato 0 (rispetto alla formula con il -1 al numeratore).

Gli elementi sulla diagonale principale q_{ii} possono essere portati fuori dalla sommatoria.

Sulla diagonale principale abbiamo tutti valori negativi che vanno a bilanciare tutti i valori positivi presenti sulle varie righe.

Pertanto:

$$Q(t) = \begin{bmatrix} -\sum_{j \neq 1} q_{1j}(t) & q_{12}(t) & \dots & q_{1N}(t) \\ q_{21}(t) & -\sum_{j \neq 2} q_{2j}(t) & \dots & q_{2N}(t) \\ \vdots & \vdots & & \vdots \\ q_{N1}(t) & q_{N2}(t) & \dots & -\sum_{j \neq N} q_{Nj}(t) \end{bmatrix}$$

Domanda: perché la matrice $Q(t)$ è detta "rate" matrix?

Risposta: perché il valore degli elementi q_{ij} sono in relazione con la frequenza degli eventi che determinano l'evoluzione del processo di Markov.

Ci interessano il processo degli arrivi e il processo di servizio.

Esempio in cui ci sono lo stato i e lo stato j e q_{ij} è l'elemento che connette i a j.

$$q_{ij}(t) = \begin{cases} \lim_{\Delta t \rightarrow 0} \frac{p_{ii}(t - \Delta t, t) - 1}{\Delta t} & i = j \\ \lim_{\Delta t \rightarrow 0} \frac{p_{ij}(t - \Delta t, t)}{\Delta t} & i \neq j \end{cases} = \begin{cases} \lim_{\Delta t \rightarrow 0} \frac{p_{ii}(t - \Delta t, t) - p_{ii}(t, t)}{\Delta t} = 1 & i = j \\ \lim_{\Delta t \rightarrow 0} \frac{p_{ij}(t - \Delta t, t) - p_{ij}(t, t)}{\Delta t} & i \neq j \end{cases}$$

Nel primo limite, al posto di 1 si scrive $p_{ii}(t, t)$ che fa 1 poiché in un tempo nullo il sistema rimane sicuramente nello stesso stato.

Nel secondo limite, è come avere -0 poiché in un tempo nullo non è possibile passare dallo stato i allo stato j . Alla fine, sono delle derivate e possiamo utilizzare i e j per approssimare una funzione.

Quindi:

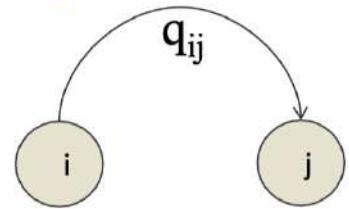
$$p_{ij}(t, t + \Delta t) \approx |q_{ij}(t)| \Delta t$$

$$p_0(0, t + \Delta t) = p_0(0, t)(1 - p_{ij}(t, t + \Delta t)) = p_0(0, t)(1 - |q_{ij}(t)| \Delta t)$$

$$\lim_{\Delta t \rightarrow 0} \frac{p_0(0, t + \Delta t) - p_0(0, t)}{\Delta t} = \frac{dp_0(0, t)}{dt} = -|q_{ij}(t)| p_0(0, t)$$

$$p_0(0, t) = e^{-\int_0^t |q_{ij}(t)| dt} \quad p_{ij}(0, t) = 1 - e^{-\int_0^t |q_{ij}(t)| dt}$$

$$\text{if } q_{ij}(t) = q_{ij} \quad p_{ij}(0, t) = 1 - e^{-|q_{ij}|t}, \quad \bar{T}_{ij} = \frac{1}{|q_{ij}|}$$



q_{ij} risulta essere uguale alla frequenza media di transizione di stato

Funzione di distribuzione, il valor medio è $\frac{1}{|q_{ij}|}$.

Valor medio di T , ovvero del tempo di transizione.

Valor medio del tempo di permanenza dentro lo stato.

q_{ij} è la frequenza media di arrivo degli utenti, è la frequenza degli eventi che determina l'evoluzione del processo e dei sistemi a coda.

Sono valori di input del nostro problema.

Se abbiamo questo valore, allora possiamo determinare le probabilità di transizione o la probabilità di stato attraverso la formula di Kolmogorov.

Equazioni di Chapman-Kolmogorov:

- forma matriciale
- in avanti

$$\frac{\partial H(s,t)}{\partial t} = H(s,t)Q(t) \quad s \leq t$$

- all'indietro

$$\frac{\partial H(s,t)}{\partial s} = -Q(s)H(s,t) \quad s \leq t$$

- termine a termine

- in avanti

$$\frac{\partial p_{ij}(s,t)}{\partial t} = q_{jj}(t)p_{ij}(s,t) + \sum_{k \neq j} q_{kj}(t)p_{ik}(s,t) \quad \text{con } p_{ij}(s,s) = \begin{cases} 1 & j = i \\ 0 & j \neq i \end{cases}$$

- all'indietro

$$\frac{\partial p_{ij}(s,t)}{\partial s} = -q_{ii}(t)p_{ij}(s,t) - \sum_{k \neq i} q_{ik}(s)p_{kj}(s,t) \quad \text{con } p_{ij}(t,t) = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

Le catene di Markov possono anche essere "riavolte", andando cioè all'indietro.

Noi utilizziamo più spesso la prima formula, in avanti.

Sono equazioni differenziali matriciali di prim'ordine (derivata 1^a del tempo t).

$p_{ij}(s,s)$ è la probabilità di transizione al tempo nullo.

Se $i = j$ allora la probabilità è 1, zero altrimenti.

Soluzioni

$$H(s,t) = e^{\int_s^t Q(u)du}$$

$$e^A = I + A + \frac{A^2}{2!} + \frac{A^3}{3!} + \dots$$

La soluzione dell'equazione differenziale di prim'ordine (soluzione di Chapman-Kolmogorov) è un esponenziale di matrice. Quindi, come argomento dell'esponenziale abbiamo una matrice. Per definire una funzione esponenziale che ha come argomento una matrice, si usa l'equivalenza formale tra lo sviluppo in serie dell'esponenziale reale e l'espressione dell'esponenziale di una matrice.

In generale, se abbiamo un'espressione esponenziale e^x allora la possiamo scrivere come somme, rispetto ad un indice k, di $\frac{x^k}{k!}$.

Sfruttando ciò, al posto della x scriviamo una matrice e, per definizione, la chiamiamo esponenziale di matrice.

La probabilità di fare una transizione da $i \rightarrow j$ nell'intervallo $(t, t + \Delta T)$ è data da:

$$p_{ij}(t, t + \Delta t) = q_{ij}(t)\Delta t + o(\Delta t) \quad o(\Delta t) \Leftrightarrow \lim_{\Delta t \rightarrow 0} \frac{o(\Delta t)}{\Delta t} = 0$$

$o(\Delta t)$ è un infinitesimo di ordine superiore pari al grado a cui ci si ferma.

In questo caso, il termine noto è zero perché c'è solo il termine lineare e di conseguenza l'errore è dato da $o(\Delta t)$. Meglio si approssima più piccolo è l'errore.

La probabilità di uscire dallo stato i -esimo nell'intervallo $(t, t + \Delta T)$ è data da:

$$1 - p_{ii}(t, t + \Delta t) = \Delta t \sum_{j \neq i} q_{ij}(t) + o(\Delta t) = -q_{ii}(t)\Delta t + o(\Delta t)$$

Ovvero 1 - la probabilità di stare dentro lo stesso stato.

Finora ci si è focalizzati sulle probabilità di transizione tra gli stati, introducendo un generatore infinitesimale i cui elementi sono legati alla frequenza degli eventi che accadono nel sistema a coda.

Ora ci si va a focalizzare sulle probabilità di stato.

La probabilità che il sistema a coda sia nello stato j al tempo t .

$$\pi_j(t) = P[X(t)=j]$$

È immediato scrivere che per $t \geq s$

$$\pi_j(t) = \sum_i \pi_i(s)p_{ij}(s,t) = [\pi_1(s) \quad \pi_2(s) \quad \dots] \begin{bmatrix} p_{1j}(s,t) \\ p_{2j}(s,t) \\ \vdots \\ p_{ij}(s,t) \end{bmatrix}$$

$$\pi^T(t) = \pi^T(s)H(s,t)$$

$$\frac{d\pi^T(t)}{dt} = \pi^T(s) \frac{\partial H(s,t)}{\partial t} = \pi^T(s)H(s,t)Q(t) = \pi^T(t)Q(t)$$

Forward Chapman-Kolmogorov equations

$$\frac{d\pi(s)}{ds} = Q(s)\pi(s) \quad \text{Backward Chapman-Kolmogorov equations}$$

Somme per tutti i possibili stati al tempo s ($\pi^T(s)$) per le probabilità di transizione dal tempo s al tempo t ($H(s,t)$). Prodotto di 2 vettori, riga colonna.

Noi usiamo i vettori colonna, indichiamo i vettori riga con il trasposto.

L'assenza del pedice indica che la variabile è un vettore.

La presenza del pedice indica che la variabile è un elemento del vettore.

Derivata rispetto a t , il primo termine non la ha quindi resta uguale mentre per il secondo termine si fa la derivata parziale.

È l'equazione differenziale che si riferisce alle probabilità di stato.

Una catena di Markov tempo-continua è detta **omogenea** se la probabilità di transizione da uno stato all'altro rimane costante nel tempo, cioè non dipende dal momento in cui si verifica la transizione ma solo dalla sua durata.

Per catene di Markov omogenee

$$\begin{aligned} p_{ij}(s,t) &= p_{ij}(\tau) \Rightarrow H(s,t) = H(\tau) = [p_{ij}(\tau)], \quad \tau = t - s \\ q_{ij}(t) &= q_{ij} \quad i,j = 1,2,\dots \Rightarrow Q(t) = Q = [q_{ij}] \end{aligned}$$

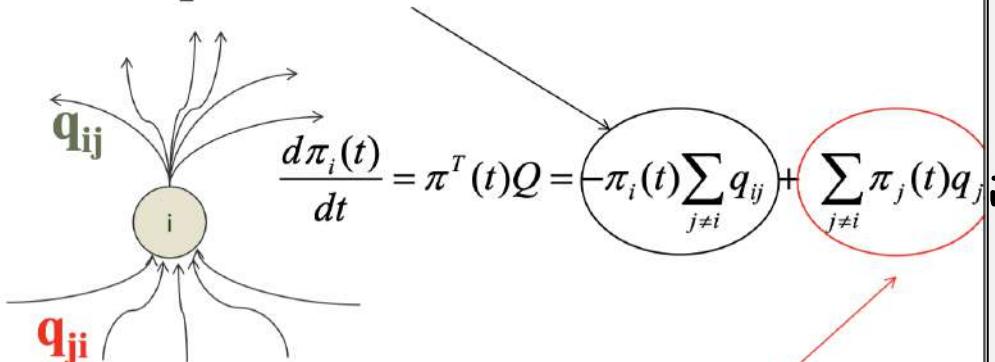
Le equazioni di Chapman-Kolmogorov diventano

$$\frac{d\pi^T(t)}{dt} = \pi^T(t)Q$$

Ci interessa solo l'intervallo di tempo, quindi la dipendenza dal tempo sparisce. Se la frequenza di transizione fra due stati in un certo intervallo di tempo non dipende dall'istante considerato allora la frequenza di transizione di stato si mantiene costante nel tempo.

Le equazioni di Chapman-Kolmogorov si possono scrivere facilmente mediante ispezione diretta:

Flusso di probabilità verso altri stati



Flusso di probabilità dagli altri stati

Consideriamo uno stato generico i e delle potenziali transizioni verso quello stato partendo da altri stati con probabilità q_{ji} .

Analogamente, consideriamo la probabilità di uscire dallo stato q_{ij} .

q_{ji} entra, q_{ij} esce.

Gli eventi in ingresso influenzano positivamente mentre gli eventi in uscita negativamente. Possiamo considerarli come flussi di probabilità in ingresso e uscita. Otteniamo l'equazione di Kolmogorov bilanciando i due flussi.

Catena di Markov **irriducibile**:

- ogni stato è raggiungibile da qualsiasi altro: $p_{ij}(t) > 0$

$$\lim_{t \rightarrow \infty} p_{ij}(t) = p_j \text{ (i)}$$

Catena di Markov **ergodica**:

- le probabilità di stato convergono ad un valore limite, indipendente dalla distribuzione iniziale degli stati

$$\lim_{t \rightarrow \infty} \pi_j(t) = \pi_j = p_j \text{ (i)}$$

Una catena di Markov omogenea si dice **irriducibile** se la probabilità di andare a finire nel lungo periodo in qualsiasi stato è diversa da zero (non nulla).

Una catena di Markov omogenea si dice **ergodica** se lo stato finale non dipende dallo stato di partenza. Se le probabilità di stato convergono ad un valore limite indipendentemente da dove si parte allora significa che le caratteristiche statistiche dell'evoluzione della catena di Markov non dipendono dallo stato di partenza. Osservare una singola realizzazione nel tempo, è come osservare tutti i possibili valori su tutte le possibili realizzazioni.

L'irriducibilità garantisce che tutti gli stati debbano avere una probabilità positiva di essere visitati in qualche istante mentre l'ergodicità garantisce che il comportamento statistico della catena non dipende dallo stato di partenza.

In questo modo il comportamento statistico di tutte le realizzazioni del processo di Markov è lo stesso, pertanto le statistiche temporali coincidono con quelle d'insieme.

Soluzione delle equazioni di Chapman Kolmogorov per catene omogenee.

$$\frac{d\pi(t)}{dt} = \pi(t)A \quad \text{A matrice generica}$$

Caso monidimensionale, $A = a$, scalare: $\pi(t) = \pi_0 e^{at}$

$$\frac{de^{at}}{dt} = e^{at}a = ae^{at}$$

$$e^{at} = 1 + at + \frac{1}{2!}(at)^2 + \cdots \frac{1}{k!}(at)^k + \cdots = \sum_{k=0}^{\infty} \frac{(at)^k}{k!}$$

$$k! = k \times (k-1) \times \cdots \times 2 \times 1$$

$$\begin{aligned} \frac{de^{at}}{dt} &= \frac{d(1 + at + \frac{1}{2}(at)^2 + \cdots + \frac{1}{k!}(at)^k + \cdots)}{dt} \\ &= \frac{0 + a + \frac{2}{2}(at)a + \cdots + \frac{k}{k!}(at)^{k-1}a + \cdots}{dt} = ae^{at} \end{aligned}$$

Soluzione delle equazioni di Chapman Kolmogorov per catene omogenee.

Caso multidimensionale: $\pi(t) = \pi_0 e^{At}$

L'esponenziale di matrice e^{At} è definito come:

$$e^{At} = I + At + \frac{1}{2!}A^2t^2 + \cdots + \frac{1}{k!}A^k t^k + \cdots = \sum_{k=0}^{\infty} \frac{A^k t^k}{k!}$$

$$\frac{de^{At}}{dt} = \frac{de^{At}}{d(At)} \frac{d(At)}{dt} = Ae^{At} = e^{At} A \quad \text{CHAIN RULE}$$

Soluzione delle equazioni di Chapman Kolmogorov per catene omogenee.

$$\begin{aligned}
 \frac{d}{dt} e^{At} &= \frac{d}{dt} (I + At + \frac{1}{2!} A^2 t^2 + \cdots + \frac{1}{k!} A^k t^k + \cdots) \\
 &= 0 + A + \frac{1}{2} 2tA^2 + \cdots + \frac{1}{k!} kt^{k-1} A^k + \cdots \\
 &= A + A^2 t + \cdots + \frac{1}{(k-1)!} A^k t^{k-1} + \cdots \\
 &= A(I + At + \cdots + \frac{1}{(k-1)!} A^{k-1} t^{k-1} + \cdots) \\
 &= A \sum_{k=0}^{\infty} \frac{A^k t^k}{k!} = Ae^{At} \\
 &= (I + At + \cdots + \frac{1}{(k-1)!} A^{k-1} t^{k-1} + \cdots)A \\
 &= \left(\sum_{k=0}^{\infty} \frac{A^k t^k}{k!} \right) A = e^{At} A
 \end{aligned}$$

Soluzione delle equazioni di Chapman Kolmogorov per catene omogenee.

 $\frac{d\pi(t)}{dt} = \pi(t)A \rightarrow \pi(t) = \pi_0 e^{At}$

$$s\pi(s) - \pi_0 = \pi(s)A$$

$$\pi(s) = \pi_0 (sI - A)^{-1}$$

$$\boxed{\pi(t) = \pi_0 L^{-1}[(sI - A)^{-1}]}$$

$$e^{At} = L^{-1}[(sI - A)^{-1}]$$

$$(sI - A)^{-1} = L[e^{At}] = L[I + At + \cdots + \frac{1}{k!} A^k t^k + \cdots]$$

$$= \frac{I}{s} + \frac{A}{s^2} + \frac{A^2}{s^3} + \cdots + \frac{A^k}{s^{k+1}} + \cdots = \sum_{k=0}^{\infty} \frac{A^k}{s^{k+1}}$$

L^{-1} è la trasformata inversa di Laplace.

Se si ha una matrice A e si vuole calcolare e^{At} allora basta calcolare questa.

Proprietà dell'esponenziale di matrice

$$e^{At} \Big|_{t=0} = I \quad [e^{At}]^{-1} = e^{-At}$$

$$e^{A(t_1+t_2)} = e^{At_1} e^{At_2} = e^{At_2} e^{At_1}$$

$$e^{A(t-t)} = e^{At} e^{-At} = e^{-At} e^{At} = I$$

$$e^{At} e^{Bt} = e^{(A+B)t} \quad \text{solo se A e B commutano}$$

Esempio

Si consideri la matrice sottostante.

$$\dot{x} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} x \quad A = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$$

Si ha che

$$A^2 = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 2^2 \end{pmatrix}, \quad A^3 = \begin{pmatrix} 1 & 0 \\ 0 & 2^2 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 2^3 \end{pmatrix}, \dots$$

In generale,

$$A^n = \begin{pmatrix} 1 & 0 \\ 0 & 2^n \end{pmatrix}$$

Quindi

$$e^{At} = \sum_{n=0}^{\infty} \frac{A^n t^n}{n!} = \sum_{n=0}^{\infty} \begin{pmatrix} 1/n! & 0 \\ 0 & 2^n/n! \end{pmatrix} t^n = \begin{pmatrix} e^t & 0 \\ 0 & e^{2t} \end{pmatrix}$$

Esempio

$$\dot{x} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} x \quad [sI - A] = \begin{bmatrix} s-1 & 0 \\ 0 & s-2 \end{bmatrix}$$

$$[sI - A]^{-1} = \begin{bmatrix} \frac{s-2}{(s-1)(s-2)} & 0 \\ 0 & \frac{s-1}{(s-1)(s-2)} \end{bmatrix}$$

$$\Phi(t) = L^{-1}\{[sI - A]^{-1}\} = \begin{bmatrix} e^t & 0 \\ 0 & e^{2t} \end{bmatrix}$$

Catene di Markov Omogenee in Equilibrio

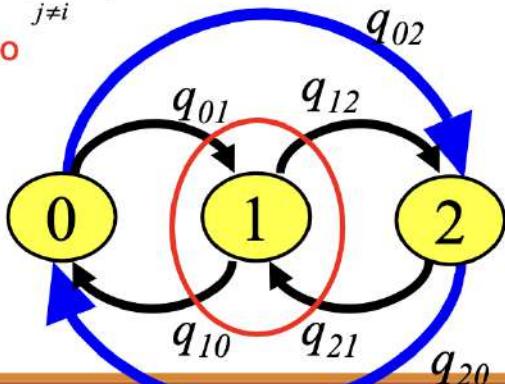
Le catene di Markov omogenee in equilibrio sono modelli matematici utilizzati per descrivere come un sistema passa casualmente da uno stato all'altro nel tempo, e come esso raggiunge uno stato stazionario in cui le probabilità di trovarsi in ogni stato rimangono costanti.

Le probabilità limite (ossia in condizione di equilibrio statistico) sono le soluzioni del sistema lineare:

$$\begin{cases} q_{jj}\pi_j + \sum_{k \neq j} q_{kj}\pi_k = 0 \\ \sum_j \pi_j = 1 \end{cases} \quad \text{oppure} \quad \begin{cases} \pi Q = 0 \\ \sum_j \pi_j = 1 \end{cases}$$

$$q_{ii}(t) = -\sum_{j \neq i} q_{ij}(t) < 0$$

Esempio



$$\begin{cases} q_{00}\pi_0 + q_{10}\pi_1 + q_{20}\pi_2 = 0 \\ q_{11}\pi_1 + q_{01}\pi_0 + q_{21}\pi_2 = 0 \\ q_{22}\pi_2 + q_{02}\pi_0 + q_{12}\pi_1 = 0 \\ \pi_0 + \pi_1 + \pi_2 = 1 \end{cases}$$

$$\begin{cases} q_{00} = -q_{01} - q_{02} \\ q_{11} = -q_{10} - q_{12} \\ q_{22} = -q_{20} - q_{21} \end{cases}$$

Troviamo le tre equazioni, una per ogni stato. Per impostarle occorre guardare lo stato attuale + i flussi di probabilità entranti in tale stato, ovvero **lo stato attuale + le frecce entranti verso lo stato**. La quarta equazione rappresenta la somma degli stati che deve fare 1. Ora si risolve il sistema a tre incognite.

Processi di nascita e morte

I processi di nascita e morte sono un tipo specifico di catene di Markov.

In questi processi, il numero di individui può aumentare (nascita) o diminuire (morte) a seguito di eventi casuali.

Non è possibile fare salti più lunghi rispetto agli stati adiacenti perché nella coda viene gestito un evento alla volta. Non ha senso parlare ad esempio da i a $i+2$. **Ci spostiamo numericamente solo di un'unità**: da i a $i+1$ oppure da i a $i-1$.

Catene di Markov in cui sono permesse, da un generico stato j , soltanto transizioni verso gli stati $j+1$ (nascita) e $j-1$ (morte), definendo

$$P_k(t) = p_k(t) = P[X(t)=k]$$

Consideriamo il caso di una catena di Markov omogenea; definiamo:

$$\begin{cases} \lambda_k = q_{k,k+1} \\ \mu_k = q_{k,k-1} \\ q_{k,k} = -(\lambda_k + \mu_k) \text{ da } \sum_j q_{kj} = 0 \\ q_{k,j} = 0 \text{ per } |k-j| > 1 \end{cases} \quad Q = \begin{bmatrix} -\lambda_0 & \lambda_0 & 0 & 0 \dots \\ \mu_1 - (\lambda_1 + \mu_1) & \lambda_1 & 0 \dots \\ 0 & \mu_2 - (\lambda_2 + \mu_2) & \lambda_2 \dots \\ \vdots & & & \end{bmatrix}$$

Nella matrice Q , la somma delle righe deve fare zero.

Per ogni riga, ad eccezione della prima, si ha $\mu_i - (\lambda_i + \mu_i) = \lambda_i$.

La distribuzione di probabilità degli stati si ottiene risolvendo le seguenti equazioni:

$$\frac{d\pi^T(t)}{dt} = \pi^T(t)Q$$

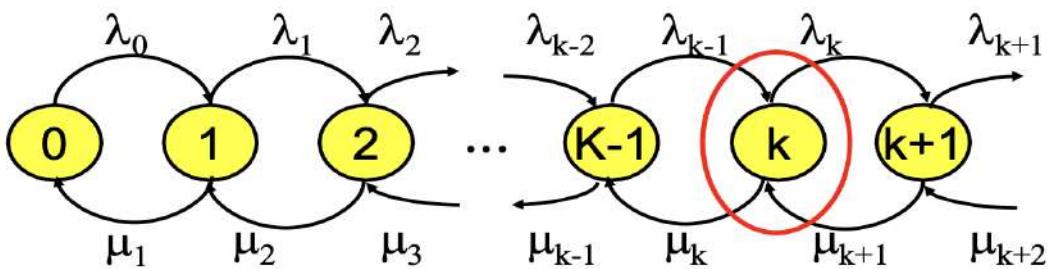
$$\begin{cases} \sum_{k=0}^{\infty} P_k(t) = 1 \\ \frac{dP_k(t)}{dt} = -(\lambda_k + \mu_k)P_k(t) + \lambda_{k-1}P_{k-1}(t) + \mu_{k+1}P_{k+1}(t) \\ \frac{dP_0(t)}{dt} = -\lambda_0P_0(t) + \mu_1P_1(t) \\ \text{Condizioni iniziali } P_k(0), k = 0, 1, \dots \end{cases}$$

... che si possono vedere anche per ispezione visiva, facendo il bilancio dei flussi di probabilità:

- Flusso entrante nello stato k : $\lambda_{k-1}P_{k-1}(t) + \mu_{k+1}P_{k+1}(t)$
- Flusso uscente dallo stato k : $-(\lambda_k + \mu_k)P_k(t)$

La differenza tra queste due quantità rappresenta il tasso di variazione di probabilità dello stato k :

$$\frac{dP_k(t)}{dt} = -(\lambda_k + \mu_k)P_k(t) + \lambda_{k-1}P_{k-1}(t) + \mu_{k+1}P_{k+1}(t)$$



La derivata rispetto al tempo di P_k (probabilità di trovarsi in uno stato k).

Occorre il tempo t perché non siamo a regime e quindi le probabilità di trovarsi in ogni stato non sono costanti e cambiano nel tempo.

Tale derivata equivale alla somma delle probabilità dei flussi entranti nello stato k per la probabilità di stare nello stato $k-1$ o $k+1$ meno la somma delle probabilità dei flussi uscenti dallo stato k per la probabilità di stare in stato k .

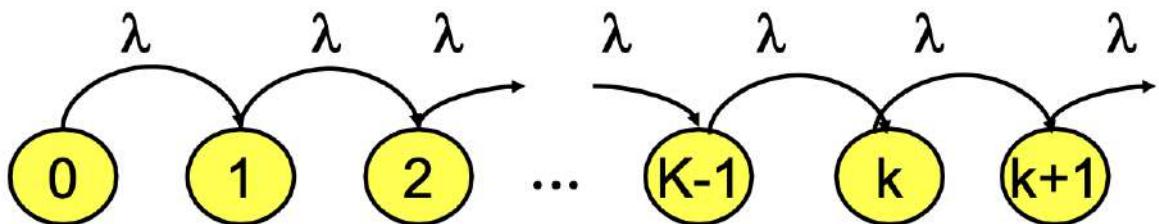
Diagramma degli stati di un sistema di nascita e morte dallo stato 0 fino ad arrivare alla fine o all'infinito, in base al modello.

Processi di Poisson

Focalizziamo l'attenzione su un **sistema con solo processi di nascita** in cui il tasso di variazione di probabilità di uno stato è costante al variare degli stati. Ciò significa che **gli arrivi non dipendono dallo stato**.

Nella pratica, il numero di utenti che sono già arrivati all'interno del sistema non influisce sul numero di utenti che devono ancora arrivare.

Caratteristiche: catena di Markov di pura nascita a tasso costante (λ)



Dopo un tempo infinito, la probabilità di trovarsi in uno stato specifico in un sistema di sole nascite è 0. È un sistema di accumulo, ad esempio lo stato 2 utenti nel sistema dopo un tempo infinito avrà probabilità 0 perché ormai è impossibile che ci siano solo 2 utenti nel sistema.

Stiamo modellando la frequenza degli arrivi. Possiamo farlo con Poisson.

I processi di Poisson sono un tipo di processo stocastico utilizzato per modellizzare l'arrivo di eventi casuali in un sistema. In particolare, si assumono che gli eventi si verifichino in modo casuale e indipendente nel tempo, senza che l'occorrenza di un evento influenzi la probabilità che se ne verifichi un altro.

$$\begin{cases} \frac{dP_0(t)}{dt} = -\lambda P_0(t) \\ \frac{dP_k(t)}{dt} = -\lambda P_k(t) + \lambda P_{k-1}(t), & k \geq 1 \end{cases}$$

$$P_0(0) = 1 \quad \text{Condizioni iniziali}$$

$$P_0(t) = e^{-\lambda t} \rightarrow \frac{dP_1(t)}{dt} = -\lambda P_1(t) + \lambda e^{-\lambda t} \quad t \geq 0$$

$$P_1(t) = \lambda t e^{-\lambda t} \rightarrow \frac{dP_2(t)}{dt} = -\lambda P_2(t) + \lambda^2 t e^{-\lambda t} \dots$$

$$P_2(t) = \frac{(\lambda t)^2}{2} e^{-\lambda t} \rightarrow \dots$$

$$P_k(t) = \frac{(\lambda t)^k}{k!} e^{-\lambda t} \quad t \geq 0 \quad \text{Probabilità di } k \text{ arrivi in } (0-t)$$

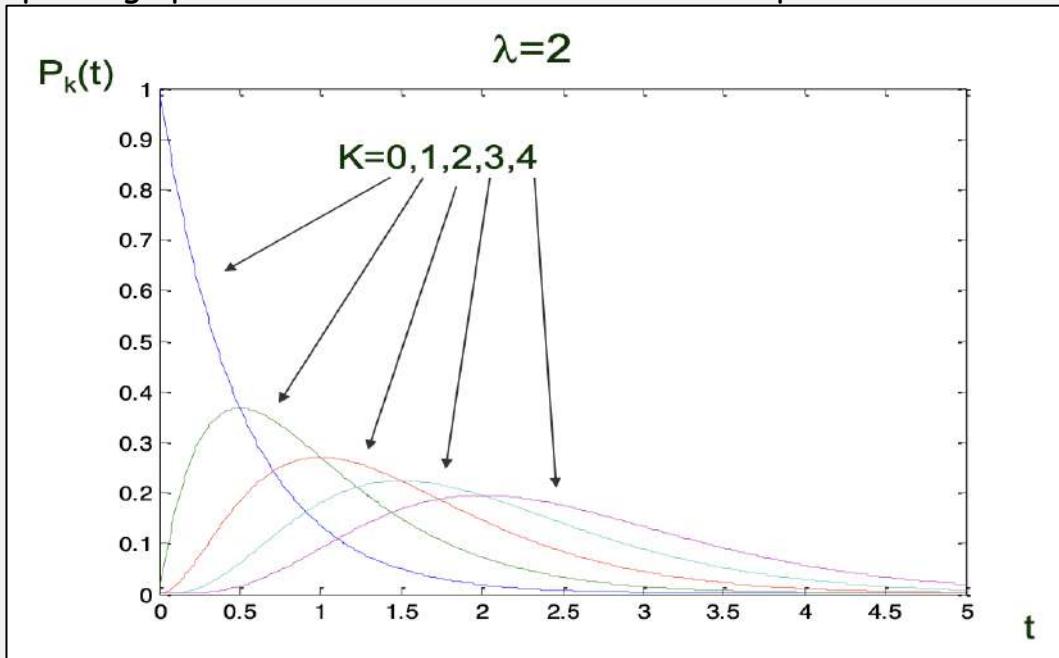
Condizioni iniziali: sistema vuoto, ossia si considerano 0 arrivi al tempo zero.

Iterando k volte queste equazioni si arriva all'ultima.

È la distribuzione di Poisson.

Fisicamente, tale distribuzione **modella la probabilità che nell'intervallo di tempo che va da 0 a t ci siano k arrivi.**

All'aumentare degli arrivi, il picco dei massimi si sposta sempre più verso destra schiacciandosi sempre più. Questo perché c'è bisogno di un intervallo di tempo sempre più lungo per ottenere un valore di arrivi K sempre crescente.



Unico processo aleatorio nel continuo in cui il valor medio è uguale alla varianza.
Se per un lungo periodo si osservano due valori simili allora la probabilità che il processo sia di Poisson è molto alta.

Sia la media che la varianza sono uno sviluppo in serie di un'esponenziale.

$$E[k] = \sum_{k=0}^{\infty} k \frac{(\lambda t)^k}{k!} e^{-\lambda t} = e^{-\lambda t} \sum_{k=0}^{\infty} k \frac{(\lambda t)^k}{k!} = e^{-\lambda t} \sum_{k=1}^{\infty} \frac{(\lambda t)^k}{(k-1)!} = \\ e^{-\lambda t} \lambda t \sum_{k=1}^{\infty} \frac{(\lambda t)^{k-1}}{(k-1)!} = e^{-\lambda t} \lambda t \sum_{j=0}^{\infty} \frac{(\lambda t)^j}{j!} = \lambda t$$

$$\sigma_k^2 = E[(k - E[k])^2] = E[k^2 - 2kE[k] + E[k]^2] = \\ = E[k^2 + k - k - 2kE[k] + E[k]^2] = E[k(k-1)] + E[k] - (E[k])^2 \\ E[k(k-1)] = \sum_{k=0}^{\infty} k(k-1) \frac{(\lambda t)^k}{k!} e^{-\lambda t} = e^{-\lambda t} (\lambda t)^2 \sum_{k=2}^{\infty} \frac{(\lambda t)^{k-2}}{(k-2)!} = (\lambda t)^2 \\ \sigma_k^2 = (\lambda t)^2 + \lambda t - (\lambda t)^2 = \lambda t$$

Valor medio = varianza

Ipotizziamo che il processo degli arrivi sia di Poisson.

statistica dei tempi di interarrivo

$$P(x(s, s+t) = k) = P_k(t) = \frac{(\lambda t)^k}{k!} e^{-\lambda t} \quad k \geq 0, t \geq 0, \forall s$$

$$A(t) = p(\tilde{t} \leq t) = 1 - p(\tilde{t} > t) = 1 - P_0(t)$$

$$A(t) = 1 - e^{-\lambda t} \quad t \geq 0$$

$$a(t) = \lambda e^{-\lambda t} \quad t \geq 0$$

$$E(t) = \frac{1}{\lambda}; \quad E(t^2) = \frac{2}{\lambda^2}; \quad \sigma_t^2 = \frac{1}{\lambda^2}$$

$$A(s) = \int_0^\infty \lambda e^{-\lambda t} e^{-st} dt = \frac{\lambda}{s + \lambda}$$

Proprietà “**memoryless**” della distribuzione esponenziale
se il tempo di permanenza in uno stato è una v.a. esponenziale, il tempo trascorso
nello stato non è utilizzabile per predire quanto si resterà ancora nello stato stesso.

\tilde{t} rappresenta il tempo di interarrivo.

$A(t)$ è la probabilità che il tempo di interarrivo sia minore uguale a t , che a sua volta è uguale a $1 - P_0(t)$.

Questo perché se il tempo di interarrivo è maggiore del tempo t allora sicuramente non ci sono arrivi nel tempo t e k è 0, quindi $P_0(t)$.

Ma $P_0(t)$ sappiamo essere pari a $e^{-\lambda t}$.

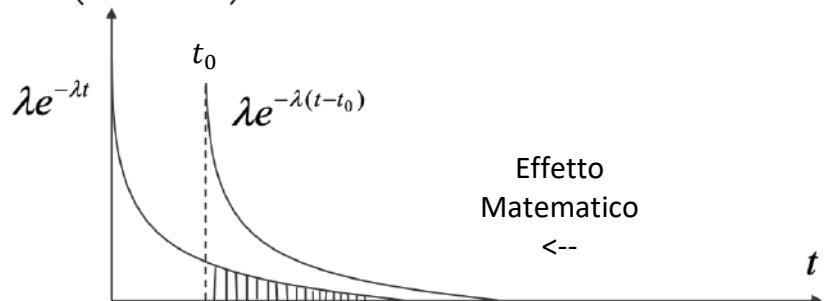
Quindi, se il processo degli arrivi è quello di Poisson allora il tempo di interarrivo è una distribuzione esponenziale.

Momenti principali della distribuzione esponenziale.

Supponiamo di fissare un istante $t=0$ di riferimento in corrispondenza di un arrivo. Se al tempo t_0 non vi è stato nessun altro arrivo, ci chiediamo quale sia la probabilità che il prossimo arrivo si verifichi dopo t a partire da t_0 .

In pratica, ci chiediamo quanto tempo t dobbiamo ancora aspettare se nell'intervallo di tempo t_0 ancora non è arrivato nessuno.

$$\begin{aligned} P[\tilde{t} \leq t + t_0 | \tilde{t} > t_0] &= \frac{P[t_0 < \tilde{t} \leq t + t_0]}{P[\tilde{t} > t_0]} = \frac{P[\tilde{t} \leq t + t_0] - P[\tilde{t} \leq t_0]}{P[\tilde{t} > t_0]} = \\ &= \frac{A(t+t_0) - A(t_0)}{1 - A(t_0)} = \frac{1 - e^{-\lambda(t+t_0)} - (1 - e^{-\lambda(t_0)})}{1 - (1 - e^{-\lambda(t_0)})} = \frac{-e^{-\lambda(t+t_0)} + e^{-\lambda(t_0)}}{e^{-\lambda(t_0)}} = \\ &= 1 - e^{-\lambda(t)} = A(t) \end{aligned}$$



In questi calcoli si dimostra che se il tempo di interarrivo è esponenziale allora aver aspettato t_0 non dà alcuna informazione!

Quanto occorre aspettare ancora se si è già aspettato per t_0 ?

Risposta: come se non si fosse aspettato t_0 !

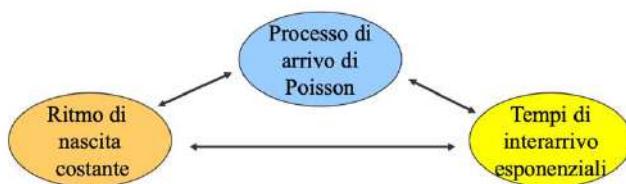
Si perde memoria di ciò che è successo in passato e un tempo di attesa non ci consente di fare alcuna previsione su quanto ancora occorre aspettare.

Riassumendo

Caratteristiche: catena di Markov di pura nascita a ritmo costante (λ)

- $\lambda_k = \lambda$ $k=0,1,2,\dots$
- $\mu_k = 0$
- $E[K] = \lambda t$, $\sigma_K^2 = \lambda t$

$$P_k(t) = \frac{(\lambda t)^k}{k!} e^{-\lambda t} \quad k \geq 0, t \geq 0$$



Sono 3 proprietà dello stesso fenomeno, osservato in 3 modi diversi.

Focalizzando l'attenzione sull'intervallo $[0, t]$, ipotizzando sempre di lavorare con processi di Poisson.



Ci siano k arrivi in $[0-t]$ (evento condizionante B_k). Suddividiamo l'intervallo $0-t$ in intervalli generici di tipo α_i , senza arrivi, e in intervalli di tipo β_i , con un singolo arrivo (evento A_k)

$$P_1(\beta_i) = \lambda \beta_i e^{-\lambda \beta_i}$$

$$P_0(\alpha_i) = e^{-\lambda \alpha_i}$$

$P_1(\beta_i)$ rappresenta la probabilità che nell'intervalle beta ci sia un solo arrivo ed equivale alla formula di Poisson con $k=1$.

Analogamente, $P_0(\alpha_i)$ rappresenta la probabilità che nell'intervalle alpha non ci siano arrivi ed equivale alla formula di Poisson con $k=0$.

Tenendo conto del fatto che i processi sono i.i.d. e che la somma di tutti gli intervalli alpha e beta è t , abbiamo che la probabilità è pari a:

$$\begin{aligned} P(A_k | B_k) &= \frac{e^{-\lambda \alpha_1} \lambda \beta_1 e^{-\lambda \beta_1} e^{-\lambda \alpha_2} \lambda \beta_2 e^{-\lambda \beta_2} \dots e^{-\lambda \alpha_k} \lambda \beta_k e^{-\lambda \beta_k} e^{-\lambda \alpha_{k+1}}}{(\lambda t)^k k! e^{-\lambda t}} = \\ &= \frac{\lambda^k (\beta_1 \beta_2 \dots \beta_k) e^{-\lambda t}}{(\lambda t)^k k! e^{-\lambda t}} = \frac{(\beta_1 \beta_2 \dots \beta_k) k!}{t^k} \end{aligned}$$

Adesso facciamo un'altra supposizione: supponiamo di distribuire in modo aleatorio k punti nell'intervallo $[0, t]$ con una distribuzione uniforme.

È semplice ricavare che:

$$P(A_k | B_k) = \left(\frac{\beta_1}{t} \frac{\beta_2}{t} \frac{\beta_3}{t} \dots \frac{\beta_k}{t} \right) k!$$

Essendo uniforme, la probabilità di cascare su un intervallo beta qualsiasi è uguale per tutti. Quindi sarebbe β_i fratto t moltiplicato per k fattoriale in quanto ci sono tutte le possibili combinazioni di distribuzione dei k punti nei vari intervalli beta. Scopriamo che le due probabilità condizionate sono uguali.

Questo significa che parlare di processi di arrivi di Poisson vuol dire parlare di processi di arrivi che si distribuiscono uniformemente nel tempo.

Dati k arrivi in $0-t$, se tali arrivi sono generati da un processo di Poisson, allora sono distribuiti uniformemente nell'intervallo $0-t$.

Si supponga di accumulare k arrivi di un processo di Poisson. A tal fine sarà necessario un tempo pari alla somma di k v.a. esponenziali, la cui funzione caratteristica sarà pari a:

$$\Pi(s) = \left(\frac{\lambda}{s + \lambda} \right)^k \quad \text{Funzione di densità di probabilità nel dominio di Laplace.}$$

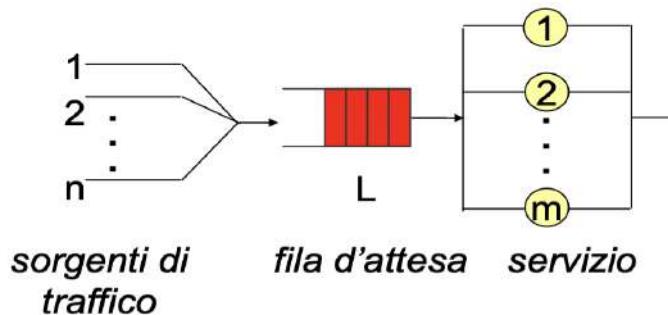
la cui anti-trasformata è pari a

$$f_x(x) = \frac{\lambda(\lambda x)^{k-1}}{(k-1)!} e^{-\lambda x} \quad x \geq 0$$

DISTRIBUZIONE DI ERLANG (k)

Dominio diretto è la convoluzione, dominio trasformato è il prodotto.

Sistemi di servizio



- Il sistema è descritto attraverso variabili aleatorie:
- k = numero di utenti nel sistema
- l = numero di utenti nella sola fila d'attesa
- h = numero di serventi contemporaneamente occupati
- x = tempo di servizio
- s = tempo di permanenza nel sistema (tempo di coda o di ritardo)
- w = tempo di permanenza nella fila d'attesa

La variabile aleatoria k è caratterizzata attraverso la sua probabilità limite

$\pi_k = p_k$ = probabilità che in un generico istante di osservazione **in regime permanente** siano presenti k utenti (richieste di servizio) all'interno del sistema

In termini generali, **la probabilità limite si riferisce alla probabilità di un evento che si verifica quando il numero di prove o di osservazioni diventa infinito.**

In altre parole, la probabilità limite di un evento è la probabilità che si verifichi l'evento quando vengono effettuati un numero infinito di tentativi.

Una volta che un sistema raggiunge il regime permanente, la distribuzione di probabilità delle sue variabili non cambia nel tempo o cambia solo in modo prevedibile e ciclico. Questo significa che la probabilità di osservare determinati valori delle variabili del sistema rimane costante nel tempo.

Parametri prestazionali

- Probabilità di sistema bloccato (m serventi)

$$S_p = \Pr\{k = L + m\} = p_{L+m}$$

- Probabilità di rifiuto
 - Data una richiesta di servizio offerto (r.s.o.)

$$\Pi_p = \Pr\{\text{sistema bloccato/r.s.o.}\} = S_p \frac{\Pr\{\text{r.s.o./sistema bloccato}\}}{\Pr\{\text{r.s.o.}\}} = S_p$$

$$P(A|B) = \frac{P(A, B)}{P(B)} = \frac{P(A)P(B|A)}{P(B)}$$

Per sistema bloccato si intende un sistema sempre pieno.

Per rifiuto si intende un sistema che scarta una richiesta da un utente.

La probabilità di rifiuto è più bassa rispetto a quella di blocco perché intuitivamente bisogna passare prima dal blocco per avere il rifiuto.

Se il processo degli arrivi è modellabile con Poisson allora la probabilità di sistema bloccato è uguale alla probabilità di rifiuto. Quindi, numeratore e denominatore si semplificano numeratore lasciando solo S_p .

Analogamente alle probabilità di prima, abbiamo:

- Probabilità di servizio bloccato (m serventi)

$$S_r = \Pr\{k \geq m\}$$

- Probabilità di ritardo

- Data una richiesta di servizio accolta (r.s.a.)

$$\Pi_r = \Pr\{\text{servizio bloccato/r.s.a.}\} = S_r \frac{\Pr\{\text{r.s.a./servizio bloccato}\}}{\Pr\{\text{r.s.a.}\}} = S_r$$

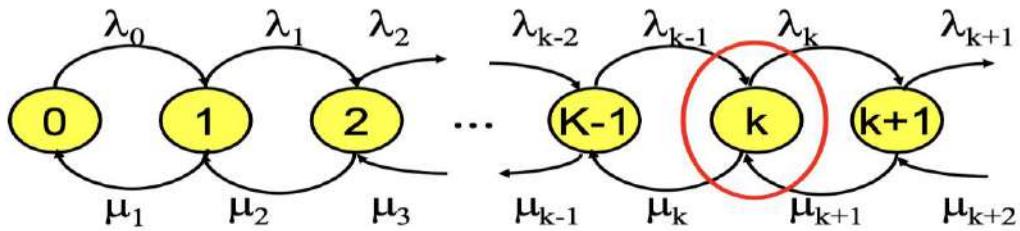
$$P(A|B) = \frac{P(A, B)}{P(B)} = \frac{P(A)P(B|A)}{P(B)}$$

La probabilità di servizio bloccato è la probabilità che il sistema sia bloccato e che quindi la richiesta viene accolta e messa in coda.

Se il processo degli arrivi è modellabile con Poisson allora la probabilità di mettersi in coda è uguale alla probabilità che il servizio è bloccato.

Processi di nascita e morte in equilibrio statistico

$$\left\{ \begin{array}{l} \sum_{k=0}^{\infty} P_k = 1 \\ 0 = -(\lambda_k + \mu_k)P_k + \lambda_{k-1}P_{k-1} + \mu_{k+1}P_{k+1} \\ 0 = -\lambda_0P_0 + \mu_1P_1 \end{array} \right.$$



$$\mu_{k+1}P_{k+1} - \lambda_kP_k = \mu_kP_k - \lambda_{k-1}P_{k-1}$$

$$\text{sia } \alpha_k = \mu_kP_k - \lambda_{k-1}P_{k-1}$$

risulta

$$\alpha_k = \text{costante} = \mu_1P_1 - \lambda_0P_0 = 0, \quad \text{quindi}$$

$$P_k = \frac{\lambda_{k-1}}{\mu_k} P_{k-1} \Rightarrow P_k = P_0 \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}$$

siccome

$$P_0 + P_0 \sum_{k=1}^{\infty} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}} = 1 \Rightarrow P_0 = \frac{1}{1 + \sum_{k=1}^{\infty} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}}$$

La probabilità di ogni stato la trovo a partire dalla probabilità dello stato precedente. La probabilità di tutti gli stati tranne P_0 .

Per ricavare P_0 si sfrutta il fatto che la somma delle probabilità di tutti gli stati deve fare 1.

Sistema a coda M/M/1/ ∞/∞

Ipotesi:

- tempi di interarrivo i.i.d. con distribuzione esponenziale negativa di parametro λ (ingresso di Poisson);
- tempi di servizio i.i.d. con distribuzione esponenziale negativa di parametro μ ;
- processi di arrivo e di servizio statisticamente indipendenti.
- singolo servente;
- spazio infinito per la fila di attesa.

Il processo di coda $K(t)$ è descrivibile mediante un processo di Markov di nascita e morte con spazio di stato $\{0,1,\dots\}$

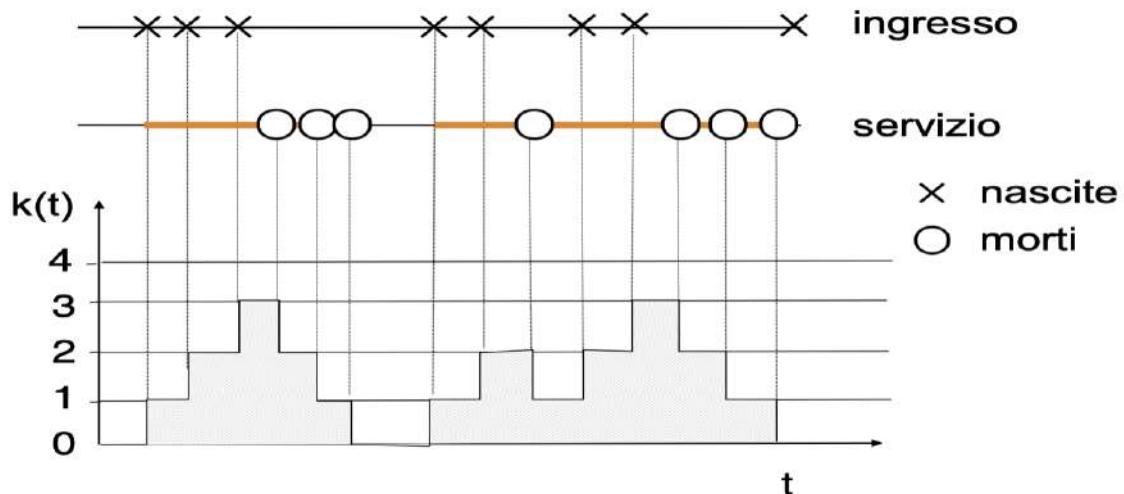
Il processo di coda $K(t)$ è ergodico se $\lambda/\mu < 1$

Il processo è stabile solo se $\lambda < \mu$.

Se in media vi sono più arrivi di quanti il sistema può servirne, la coda crescerà indefinitamente e non ci sarà equilibrio.

Se chiamiamo il loro rapporto $\rho = \frac{\lambda}{\mu}$ allora la condizione diventa $\rho < 1$.

Evoluzione temporale



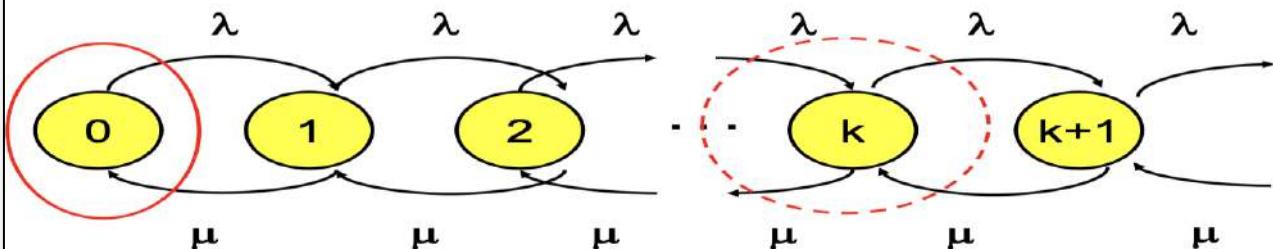
Frequenze di transizione di stato

$$\lambda_k = \lambda \quad \text{per } k \geq 0$$

frequenza di nascita

$$\mu_k = \mu \quad \text{per } k \geq 1$$

frequenza di morte



Probabilità limite di stato

Sostituendo λ e μ nella soluzione generale:

$$P_k = P_0 \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}} \quad \text{dove} \quad P_0 = \frac{1}{1 + \sum_{k=1}^{\infty} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}}$$

Nel nostro caso
 λ e μ sono costanti,
 spariscono i pedici.

$$P_0 = \frac{1}{1 + \sum_{k=1}^{\infty} \rho^k} = 1 - \rho$$

Portando 1 dentro la sommatoria.
 È la serie geometrica.

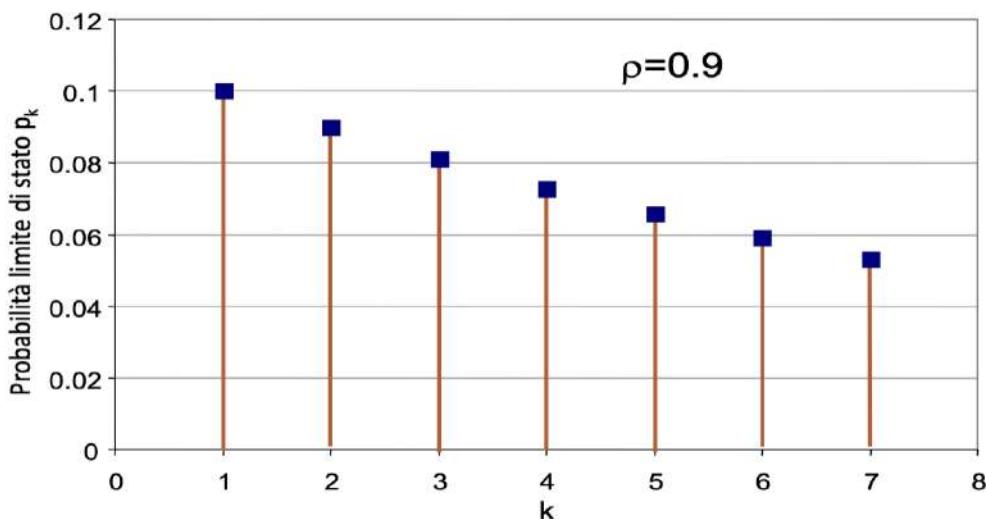
dove $\rho = \lambda/\mu$, $\rho < 1$

$$P_k = (1 - \rho) \rho^k$$

Probabilità degli stati a regime.

$k=0,1,2, \dots$ (distribuzione geometrica)

Probabilità limite di stato



La distribuzione è di tipo geometrico con parametro ρ

Probabilità limite di stato

Il numero medio di utenti nel sistema è

$$\begin{aligned}
 E[K] &= \bar{k} = \sum_{k=0}^{\infty} k \cdot p_k = \sum_{k=0}^{\infty} k(1-\rho)\rho^k = (1-\rho) \sum_{k=0}^{\infty} k\rho^k = \\
 &\quad \boxed{\text{Portando un } \rho \text{ fuori dalla } \sum \text{ rimane } \rho^{k-1}.} \\
 &= (1-\rho)\rho \sum_{k=0}^{\infty} k\rho^{k-1} = (1-\rho)\rho \sum_{k=1}^{\infty} k\rho^{k-1} = (1-\rho)\rho \left(\frac{\partial}{\partial \rho} \sum_{k=0}^{\infty} \rho^k \right) = \\
 &= (1-\rho)\rho \left(\frac{\partial}{\partial \rho} \frac{1}{1-\rho} \right) = \boxed{\frac{\rho}{1-\rho}} \quad \begin{array}{l} \text{Media del numero di utenti presenti} \\ \text{all'interno del sistema.} \end{array}
 \end{aligned}$$

Il tempo di permanenza medio nel sistema è (Legge di Little)

$$T = \frac{\bar{k}}{\lambda} = \frac{\rho}{\lambda(1-\rho)} = \frac{1/\mu}{(1-\rho)} = \frac{1}{\mu - \lambda}$$

Se il processo degli arrivi o dei servizi non è deterministico allora si crea sempre un effetto coda e il sistema si blocca, occorre quindi essere sempre un po' distanti da 1 (carico massimo) perché l'effetto asintoto c'è sempre e se ci si avvicina troppo a 1 schizza verso l'alto e probabilmente il sistema si blocca.

Se $\rho \rightarrow 0$ allora il numero medio di utenti presenti nel sistema tende a zero.

Quando arriva un utente, seppur ogni morto di papà, comunque deve ricevere il servizio e il tempo medio di servizio è $\frac{1}{\mu}$.

Parametri prestazionali

In condizioni di equilibrio statistico l'intensità media di richieste smaltite A_s coincide con l'intensità di richieste di servizio offerte A_o

Il traffico smaltito è uguale al traffico offerto ed è uguale a ρ che è uguale a $1 - p_0$ con p_0 strettamente maggiore di zero.

$$A_s = A_o = \rho = 1 - p_0$$

La probabilità di servizio bloccato S_r coincide con la probabilità di ritardo nel ricevere servizio Π_r

$$S_r = \Pi_r = (1 - \rho) \sum_{k=1}^{\infty} \rho^k = \rho$$

Σ per k che va da 1 a infinito. In tutti questi stati il server è occupato.

ρ = prob. che il servente sia occupato = la percentuale temporale di occupazione del servente = la prob. che una richiesta in arrivo sia costretta ad attendere in coda

Distribuzioni in equilibrio statistico

l = lunghezza della fila d'attesa=numero di utenti nella fila d'attesa

$$Pr\{l = j\} = \begin{cases} (1 - \rho) + \rho(1 - \rho) = 1 - \rho^2 & j = 0 \\ (1 - \rho) \cdot \rho^{j+1} & j \geq 1 \end{cases}$$

$$\bar{l} = \frac{\rho^2}{1 - \rho}$$

h =numero di serventi impegnati

$$Pr\{h = j\} = \begin{cases} 1 - \rho & j = 0 \\ \rho & j = 1 \end{cases}$$

$$\bar{h} = \rho$$

il numero medio di utenti all'interno del sistema è quindi

$$\bar{k} = \bar{l} + \bar{h} = \frac{\rho}{1 - \rho}$$

$Pr\{l = j\}$ è la probabilità che la coda l sia lunga j .

Ad esempio, la probabilità che la coda sia lunga 5 si ha quando il sistema si trova nello stato 6, dove 1 utente viene servito e gli altri 5 aspettano.

Di conseguenza, $Pr\{l = 0\}$ si ha quando il sistema si trova nello stato 0 o 1.

Tempi di attesa in coda

Ora andiamo a lavorare sulla distribuzione dei tempi di attesa in coda.
L'utente $k+1$ deve aspettare che terminino i primi k utenti, la somma di k variabili aleatorie temporali esponenziali, il tempo complessivo.
Quindi, sarebbe una convoluzione che nel dominio trasformato diventa il prodotto delle rispettive densità di probabilità.
Quindi, in media si ottiene:

Si supponga che un utente trovi, al suo arrivo, il sistema nello stato k , ossia vi sono altri k utenti presenti nel sistema (uno in servizio e $k-1$ nella fila di attesa). Nel caso di disciplina **FIFO**, l'utente, prima di essere servito dovrà attendere un tempo pari alla somma di k v.a. esponenziali. Questo avverrà con probabilità $\rho^k(1-\rho)$. In media si avrà che:

Il modulo di questa frazione dev'essere < 1.

$$\begin{aligned} W(s) &= \sum_{k=0}^{\infty} \left(\frac{\mu}{s+\mu} \right)^k \rho^k (1-\rho) = (1-\rho) \frac{1}{1 - \frac{\mu}{s+\mu} \rho} = \\ &= (1-\rho) \frac{s+\mu}{s+\mu - \mu\rho} = \frac{(1-\rho)(s+\mu + \lambda - \lambda)}{s+\mu(1-\rho)} = (1-\rho) + \frac{\lambda(1-\rho)}{s+\mu(1-\rho)} \end{aligned}$$

$$w(t) = (1-\rho)\delta(t) + \lambda(1-\rho)e^{-\mu(1-\rho)t}$$

$W(s)$ è la densità di probabilità media dell'utente che arriva nel sistema a coda, espressa nel dominio di Laplace s .

L'anti-trasformata di una costante è la costante per l'impulso delta(t) mentre il secondo termine è l'anti-trasformata di un'esponenziale.

Quindi la funzione di densità di probabilità dei tempi di attesa in coda è uguale ad un impulso matematico di un certo coefficiente più un'esponenziale.

W sta per Wait (attesa) mentre $w(t)$ è il tempo di attesa in coda.

Tempi di attesa in coda

Probabilità che il tempo di attesa in coda sia minore o uguale a t è dato dall'integrale di w(t).

$$F_w(t) = Pr(w \leq t) = 1 - \rho \cdot e^{-(1-\rho) \cdot \mu \cdot t}$$

$$W = \frac{\rho}{\mu} \cdot \frac{1}{1 - \rho}$$

Detto inoltre w_r l' $r/100$ -percentile del tempo di attesa (cioè quel valore che non è superato per una percentuale di tempo uguale a $r\%$)

$$Pr(w \leq w_r) = \frac{r}{100}$$

$$w_r = \frac{W}{\rho} \ln \left(\frac{100\rho}{100-r} \right)$$

Tempi di permanenza nel sistema

Nel caso di disciplina **FIFO**, l'utente, prima di essere servito dovrà attendere un tempo pari alla somma di $k+1$ v.a. esponenziali. Questo avverrà con probabilità $\rho^k(1-\rho)$. In media si avrà che:

$$S(s) = \sum_{k=0}^{\infty} \left(\frac{\mu}{s+\mu} \right)^{k+1} \rho^k (1-\rho) = \frac{\mu}{s+\mu} (1-\rho) \frac{1}{1 - \frac{\mu}{s+\mu} \rho} =$$

$$= \frac{\mu(1-\rho)}{s+\mu(1-\rho)} \quad \Rightarrow \quad s(t) = \mu(1-\rho) e^{-\mu(1-\rho)t}$$

C'è il tempo di servizio dell'utente stesso da considerare quindi abbiamo la stessa equazione di prima ma al posto di k abbiamo $k+1$.

La s minuscola sta per System.

L'impulso matematico non c'è più perché non può essere mai nullo.

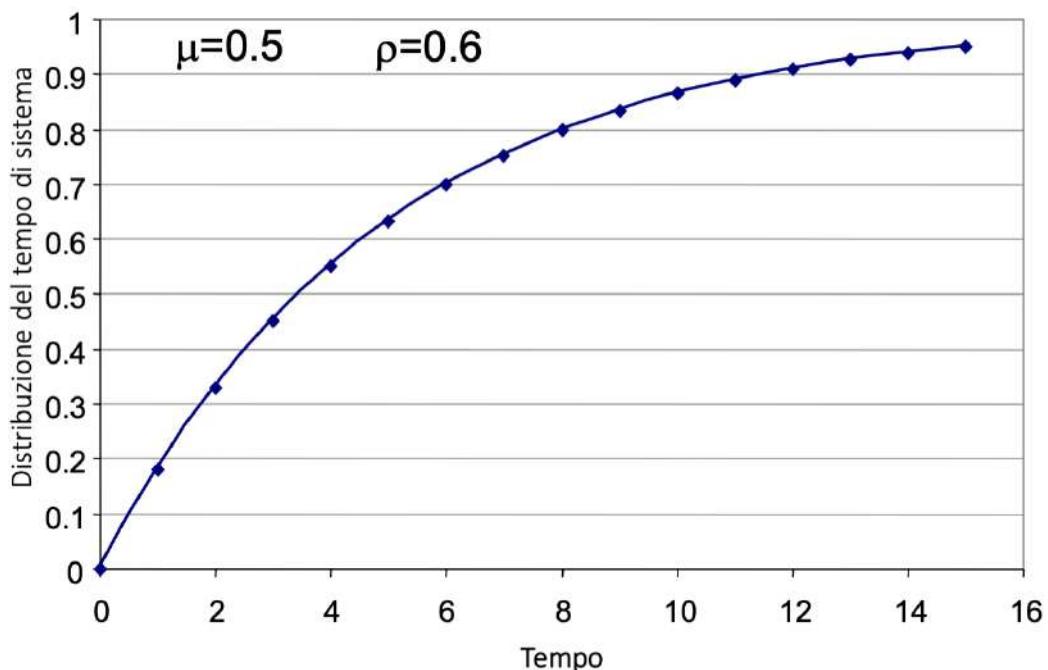
$$F_s(t) = Pr(s \leq t) = 1 - e^{-(1-\rho)\mu \cdot t}$$

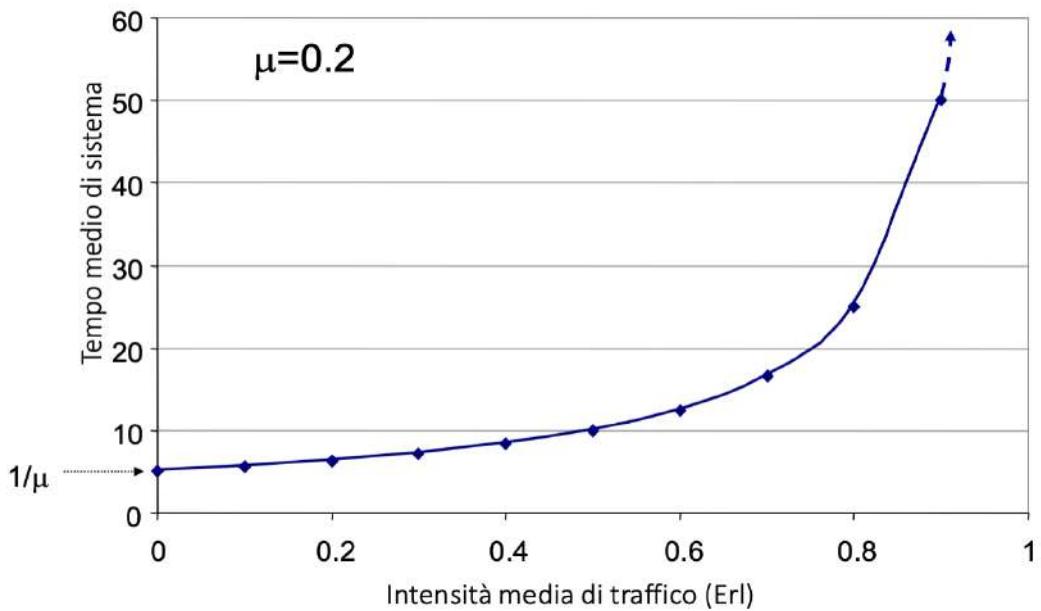
$$T = \frac{1}{\mu \cdot (1-\rho)} = \frac{1}{\mu - \lambda} = \bar{w} + \frac{1}{\mu}$$

detto inoltre s_r il percentile $r\%$ del tempo di coda

$$Pr(s \leq s_r) = \frac{r}{100}$$

$$s_r = T \frac{-\ln(1-r/100)}{1-\rho}$$





Al crescere dell'intensità di traffico il tempo di coda tende all'infinito

Sistema a coda M/M/ ∞

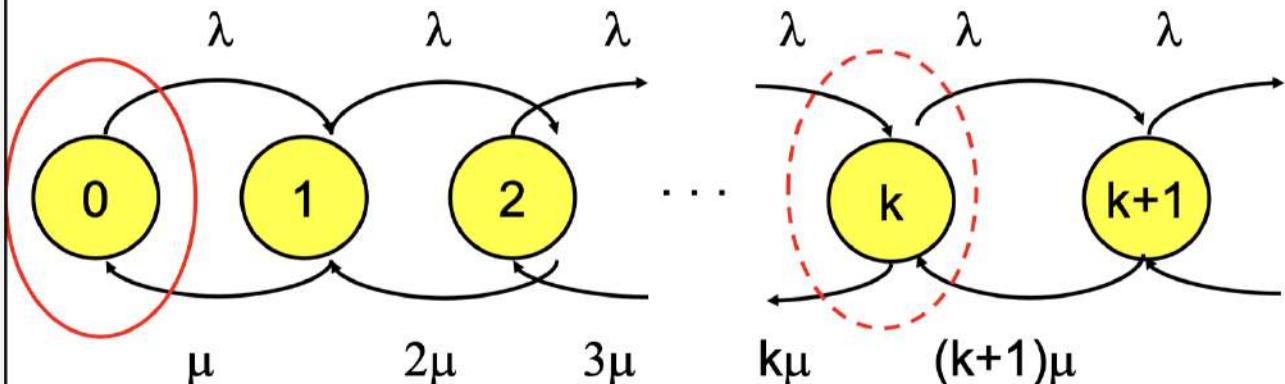
Markoviano, Markoviano, numero dei serventi infinita.

Stiamo modellando una Cloud pubblica, come Amazon. Si potrebbe utilizzare per calcolare quanto costerebbe esportare un'azienda dentro Amazon.

Sistema a coda M/M/ ∞

$$\lambda_k = \lambda \quad \text{per } k \geq 0 \quad \text{frequenza di nascita}$$

$$\mu_k = k\mu \quad \text{per } k \geq 0 \quad \text{frequenza di morte}$$



$$\frac{\lambda}{\mu} < \infty \quad P_k = P_0 \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}} \quad \text{dove} \quad P_0 = \frac{1}{1 + \sum_{k=1}^{\infty} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}}$$

Siccome i serventi sono infiniti, in ogni stato k si hanno k tempi di servizio.
Il sistema non è mai instabile.

Amazon ha così tanti (soldi) server che possiamo considerarli infiniti.

$$P_0 = \frac{1}{1 + \sum_{k=1}^{\infty} \prod_{i=0}^{k-1} \frac{\lambda}{(i+1)\mu}} = \frac{1}{1 + \sum_{k=1}^{\infty} \left(\frac{\lambda}{\mu}\right)^k \frac{1}{k!}} = e^{-\frac{\lambda}{\mu}}$$

Il risultato è $e^{-\frac{\lambda}{\mu}}$
 perché è uno sviluppo in serie con argomento $\frac{\lambda}{\mu}$.

$$P_k = e^{-\frac{\lambda}{\mu}} \prod_{i=0}^{k-1} \frac{\lambda}{(i+1)\mu} = \left(\frac{\lambda}{\mu}\right)^k \frac{1}{k!} e^{-\frac{\lambda}{\mu}}, \quad k = 0, 1, 2, \dots$$

P_k è dato da P_0 per la Π .

La distribuzione di probabilità degli stati è una distribuzione di Poisson discretizzata

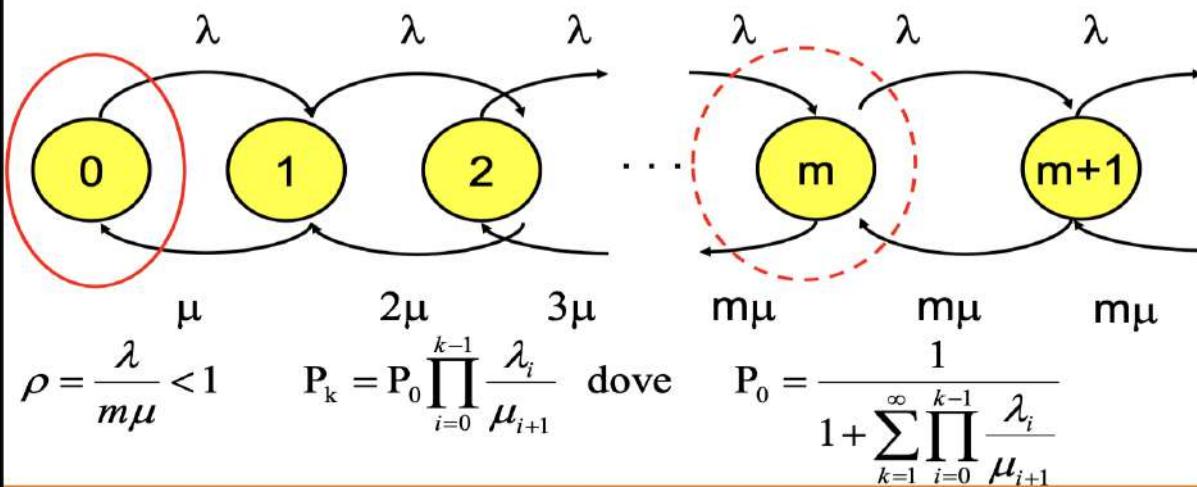
$$\bar{N} = \frac{\lambda}{\mu} \quad T = \frac{1}{\mu}$$

La media \bar{N} è $\lambda * \frac{1}{\mu} = \frac{\lambda}{\mu}$.
 È la legge di Little.

Analizziamo adesso una Cloud privata, i serventi passano da infinito ad m .

Sistema a coda M/M/m

$$\lambda_k = \lambda \quad \mu_k = \min(k\mu, m\mu) = \begin{cases} k\mu, & 0 < k < m \\ m\mu, & m \leq k \end{cases}$$



Dallo stato $m+1$ in poi, si ha sempre $m * \mu$ perché i serventi sono finiti e sono sempre quelli.

Sistema a coda M/M/m

- 1) P_k in funzione di P_0 per $k \leq m$.
- 2) P_k in funzione di P_0 per $k \geq m$.
- 3) Trovo P_0 .

$$P_k = P_0 \prod_{i=0}^{k-1} \frac{\lambda}{(i+1)\mu} = P_0 \left(\frac{\lambda}{\mu} \right)^k \frac{1}{k!} = P_0 \frac{(m\rho)^k}{k!}, \quad k \leq m$$

$$P_k = P_0 \prod_{i=0}^{m-1} \frac{\lambda}{(i+1)\mu} \prod_{i=m}^{k-1} \frac{\lambda}{m\mu} = P_0 \left(\frac{\lambda}{\mu} \right)^k \frac{1}{m! m^{k-m}} = P_0 \frac{\rho^k m^m}{m!}, \quad k \geq m.$$

$$P_0 = \frac{1}{1 + \sum_{k=1}^{\infty} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}} = \frac{1}{1 + \sum_{k=1}^{m-1} \left(\frac{\lambda}{\mu} \right)^k \frac{1}{k!} + \sum_{k=m}^{\infty} \left(\frac{\lambda}{\mu} \right)^k \frac{1}{m! m^{k-m}}} =$$

$$= \frac{1}{1 + \sum_{k=1}^{m-1} \frac{(m\rho)^k}{k!} + \sum_{k=m}^{\infty} \frac{(m\rho)^k}{m!} \frac{1}{m^{k-m}}} = \frac{1}{\sum_{k=0}^{m-1} \frac{(m\rho)^k}{k!} + \frac{(m\rho)^m}{m!} \frac{1}{1-\rho}}$$

Calcoliamo i punti 1 e 2 in modo da avere la distribuzione di probabilità di tutti gli stati per calcolare poi P_0 .

Nel calcolare P_0 splittiamo la sommatoria: una va da 1 a $m-1$ e una va da m a infinito. Il conto diventa analogo al precedente, porto 1 dentro la sommatoria. $\frac{1}{1-\rho}$ esce fuori dalla serie geometrica con la sommatoria che va da m a infinito.

Sistema a coda M/M/m

Un utente che arriva in ingresso al sistema ha la necessità di accodarsi con probabilità pari a:

$$P[\text{coda}] = \sum_{k=m}^{\infty} P_k = \sum_{k=m}^{\infty} P_0 \frac{(m\rho)^k}{m!} \frac{1}{m^{k-m}} = P_0 \frac{(m\rho)^m}{m!} \frac{1}{1-\rho}$$

$$P[\text{coda}] = \frac{(m\rho)^m}{m!} \frac{1}{1-\rho} \\ \sum_{k=0}^{m-1} \frac{(m\rho)^k}{k!} + \frac{(m\rho)^m}{m!} \frac{1}{1-\rho}$$

FORMULA DI ELRLAG C,
indicata come $C(m, \lambda/\mu)$

E' utilizzata per determinare la **probabilità di attesa** nell'accesso a una risorsa condivisa di m serventi disponibili. Ad esempio, può essere utilizzata nei call center per calcolare il numero di operatori necessari per gestire le chiamate entranti posto un certo livello di servizio.

Lega la probabilità che un utente è costretto ad accordarsi in funzione del numero di serventi e del rapporto $\frac{\lambda}{\mu}$ ovvero ρ .

Quindi conoscendo i parametri del sistema siamo in grado di capire la probabilità di essere costretti ad accordarci dato un numero di serventi m .

Esempio

Si consideri un centralino telefonico operante ad attesa. Si assuma che:

- a un fascio di giunzioni all'uscita dell'autocommutatore sia offerto un traffico poissoniano entrante con intensità media di 25 Erl;
- tale fascio sia composto da 30 giunzioni;
- la durata di una conversazione telefonica sia distribuita con legge esponenziale negativa e con valore medio di 3 min.

Si determini la probabilità che una chiamata venga accodata

In quest'esempio, abbiamo 25 serventi che in media offrono traffico continuativo. 30 è il numero totale dei serventi.

Per studiare questo sistema e determinare la probabilità che una chiamata venga accodata utilizziamo il modello M/M/30.

Modello M/M/m

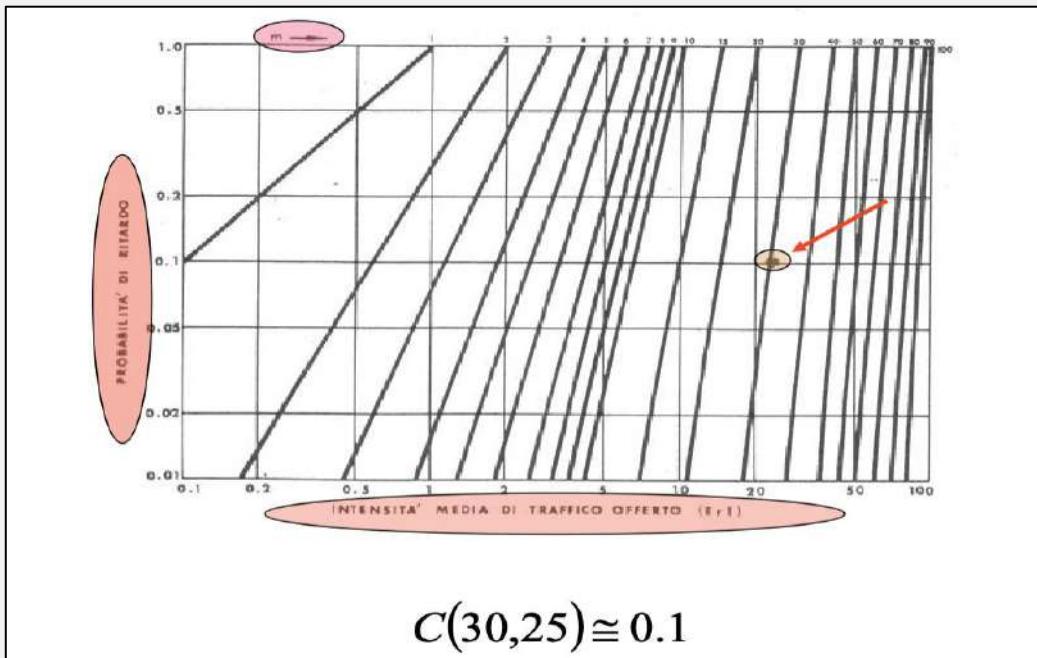
La probabilità di entrare in coda (cioè di subire un ritardo) è data da:

$$C\left(m, \frac{\lambda}{\mu}\right) = \frac{\frac{(m\rho)^m}{m!} \frac{1}{1-\rho}}{\sum_{k=0}^{m-1} \frac{(m\rho)^k}{k!} + \frac{(m\rho)^m}{m!} \frac{1}{1-\rho}} \quad \text{C-ERLANG}$$

$$m = 30$$

Il traffico offerto A_0 è pari a 25 Erlang

$$\rho = \frac{\lambda}{m\mu} = \frac{25}{30} = 0.83 \quad \text{coefficiente di utilizzazione}$$



Nel nostro caso m è 30. Vedendo graficamente dove si interseca si ottiene che la probabilità di mettersi in coda che è pari a 0.1.
Fine esempio.

Sistema a coda M/M/m/m/ ∞

Ipotesi:

- tempi di interarrivo i.i.d. con distribuzione esponenziale negativa (λ);
- tempi di servizio i.i.d. con distribuzione esponenziale negativa (μ);
- processi di arrivo e di servizio statisticamente indipendenti.
- m serventi, statisticamente identici ed indipendenti;
- capacità nulla della fila d'attesa.

Il processo di coda è descrivibile mediante un processo di Markov di nascita e morte con spazio di stato $\{0, \dots, m\}$.

Il processo di coda è ergodico per ogni valore positivo di λ e μ (coda a perdita)

Questo sistema è molto utilizzato.

L' ∞ finale indica che il numero della popolazione è infinitamente grande.

Assumiamo che i processi di arrivo e di servizio siano i.i.d., non è obbligatorio ma come assunzione ci può stare.

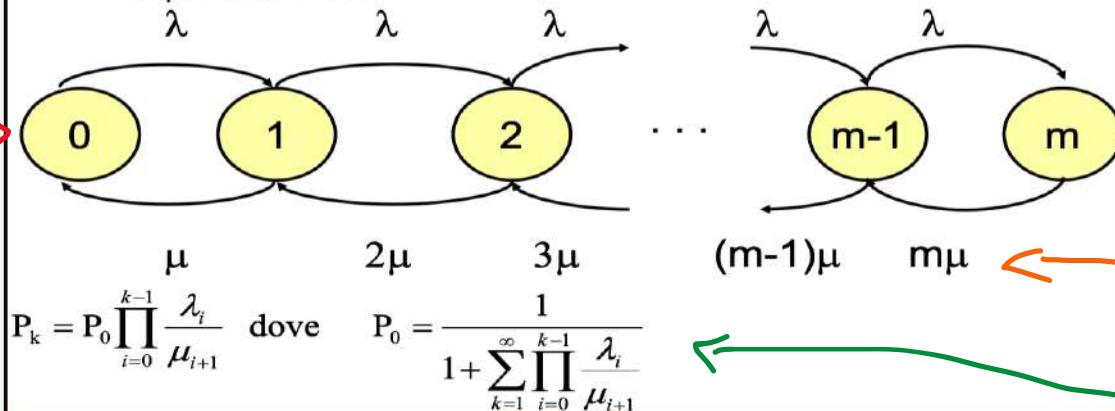
La quarta m indica la dimensione del sistema, se è uguale al numero di serventi, la terza m , allora non c'è una capacità di attesa.

Quindi, quando un utente arriva o viene servito o viene scartato.

Frequenze di transizione di stato

$$\lambda_k = \lambda \quad \text{per } 0 \leq k \leq m-1 \\ \text{frequenza di nascita}$$

$$\mu_k = k\mu \quad \text{per } 1 \leq k \leq m \\ \text{frequenza di morte}$$



Nello stato 0 tutti i serventi sono liberi mentre nello stato m sono tutti i serventi si assumono occupati.

Il tasso di servizio μ arriva fino ad un massimo di $m\mu$.

Alle formule generali di P_k e P_0 andiamo a sostituire λ_i con un λ costante e μ_{i+1} con $(i+1)\mu$. Inoltre, nella sommatoria di P_0 sostituiamo ∞ con m .

Otteniamo che la distribuzione di probabilità per ogni stato è pari a:

Probabilità limite di stato

Per l'equilibrio dei flussi si ha (formule generali):

$$P_k = P_0 \prod_{i=0}^{k-1} \frac{\lambda}{(i+1)\mu} \quad \text{dove} \quad P_0 = \frac{1}{\sum_{k=0}^m \left(\frac{\lambda}{\mu}\right)^k \frac{1}{k!}}$$

posto $A_o = \lambda/\mu$: traffico offerto al sistema, risulta

$$P_k = P_0 \left(\frac{\lambda}{\mu}\right)^k \frac{1}{k!} = P_0 A_o^k \frac{1}{k!} \quad 0 < k \leq m$$

$$P_0 = \frac{1}{\sum_{j=0}^m (A_o)^j \frac{1}{j!}}$$

$$P_k = \frac{A_o^k \frac{1}{k!}}{\sum_{j=0}^m A_o^j \frac{1}{j!}}$$

Probabilità di blocco di servizio

Nel caso di processo di ingresso di Poisson, dato che la probabilità di r.s.o. è indipendente dallo stato, si ha:

$$\Pi_p = S_p \frac{\lambda_m}{A_o} = S_p$$

Nel caso di sistema a coda M/M/m/m (per k=m)

$$\Pi_p = S_p = \frac{A_o^m \frac{1}{m!}}{\sum_{j=0}^m A_o^j \frac{1}{j!}}$$

FORMULA B
DI ERLANG

Nella formula utilizziamo m perché a noi interessa m .

Quindi, otteniamo la probabilità di blocco, alias di rifiuto.

Mette in relazione la probabilità di blocco o di rifiuto con il valore di A_o , che è il traffico offerto nel sistema con m numero di serventi.

Quindi, possiamo calcolare la probabilità di rifiuto.

Potremmo utilizzarla per stabilire la dimensione di un centralino telefonico.

Formula B di Erlang

L'espressione della probabilità di sistema bloccato e di rifiuto per un sistema a coda M/M/m/m a perdita in senso stretto è denominata anche funzione di Erlang del 1° tipo di ordine m e di argomento A_o .

Godere inoltre della proprietà di calcolo di tipo ricorsivo, infatti:

$$E_{1,m}(A_o) = \frac{A_o^m \frac{1}{m!}}{\sum_{j=0}^m A_o^j \frac{1}{j!}} = \frac{A_o E_{1,m-1}(A_o)}{m + A_o E_{1,m-1}(A_o)}$$

In questo modo si evita il fattoriale.

- con il primo elemento pari a:

$$E_{1,1}(A_o) = \frac{A_o}{1 + A_o}$$

Per evitare l'effetto overflow del fattoriale, è possibile esprimere la formula in **forma ricorsiva**.
Il pedice 1 di E sta ad indicare che utilizza la formula B di Erlang.

La grande importanza della formula B di Erlang risiede anche nel fatto che essa risulta valida per qualsiasi distribuzione dei tempi di servizio (resta necessaria l'ipotesi di i.i.d.).

In condizioni di equilibrio statistico la distribuzione del numero di utenti nel sistema è funzione del solo tempo medio di servizio $1/\mu$ e non della distribuzione del tempo di servizio stesso.

Gode della proprietà "insensitiva" rispetto al processo di servizio.

Ovvero è una formula generale che può valere per qualunque distribuzione del tempo di servizio e questo è top perché se l'applichiamo in sistemi a noi più vicini come VNCC per un sistema in cloud allora, quando osserviamo i servizi che il processo degli arrivi sia markoviano ci sta, ma che i processi di servizio siano un'esponenziale negativa non è scontato.

Ma se questa formula vale, allora l'applicabilità diventa generale.

Parametri prestazionali

Intensità media di traffico o «lavoro» smaltito A_s , che rappresenta il numero medio di serventi contemporaneamente occupati, dipende da A_o e dal numero di serventi m :

$$A_s = \sum_{k=1}^m kP_k = A_o [1 - E_{1,m}(A_o)]$$

Il traffico smaltito A_s è uguale al traffico offerto A_o per la probabilità che il sistema non sia bloccato.

Intensità media di traffico o «lavoro» rifiutato:

$$A_p = A_o - A_s = A_o E_{1,m}(A_o)$$

A_p è il traffico perso.

Coefficiente di utilizzazione del servente:

$$\rho = \frac{A_s}{m} = \frac{A_o}{m} [1 - E_{1,m}(A_o)]$$

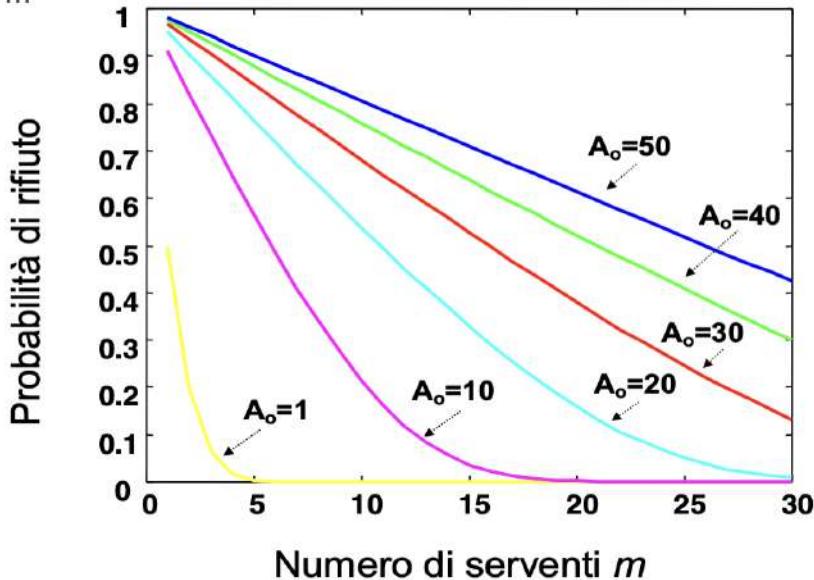
ρ, m ed A_o sono strettamente legati. Possono essere utilizzati per valutare le prestazioni o per valutare se occorre un aggiornamento del sistema, ecc. Mostrano quanto efficientemente sono utilizzati i serventi, che dipende da quanti serventi si utilizzano. Più se ne usano, più i serventi lavorano peggio.

All'aumentare del traffico offerto aumenta la probabilità di rifiuto.

All'aumentare del numero di serventi diminuisce la probabilità di rifiuto.

Probabilità di rifiuto in funzione di m

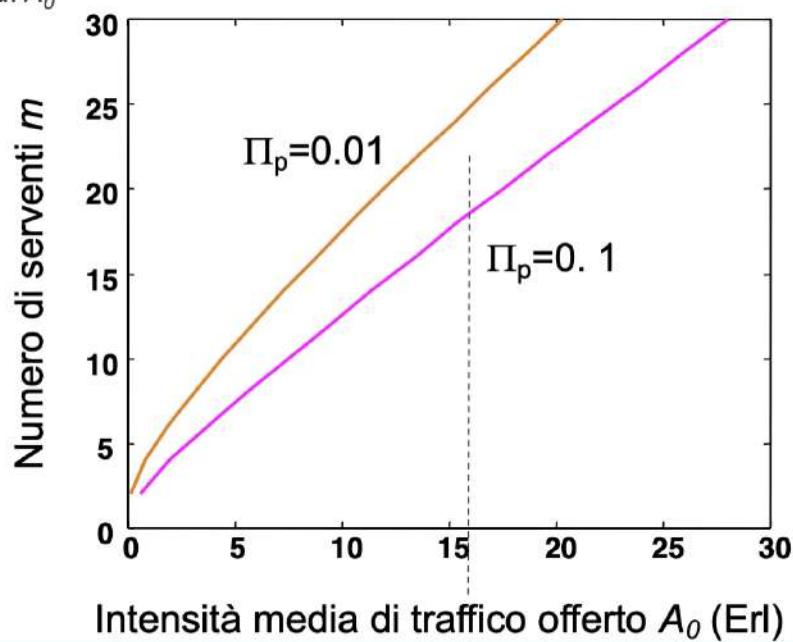
La probabilità di rifiuto, a parità di A_o , decresce al crescere del numero di serventi m



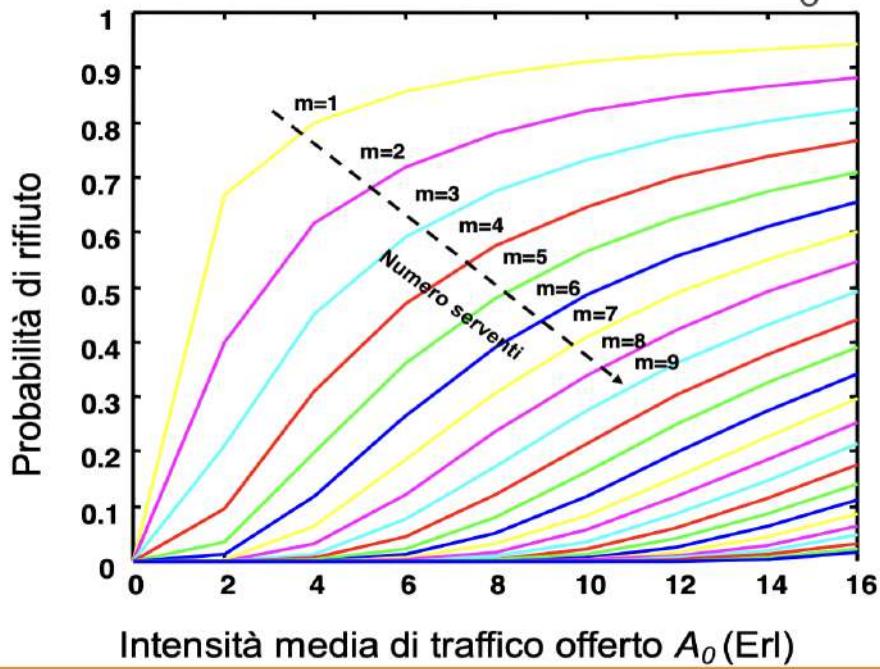
Quanti serventi occorrono per soddisfare un traffico offerto A_o di 16 con probabilità di rifiuto 0.1.

Dimensionamento di m in funzione di Π_p

La probabilità di rifiuto è, a parità di m , una funzione monotona crescente di A_o



Probabilità di rifiuto in funzione di A_0

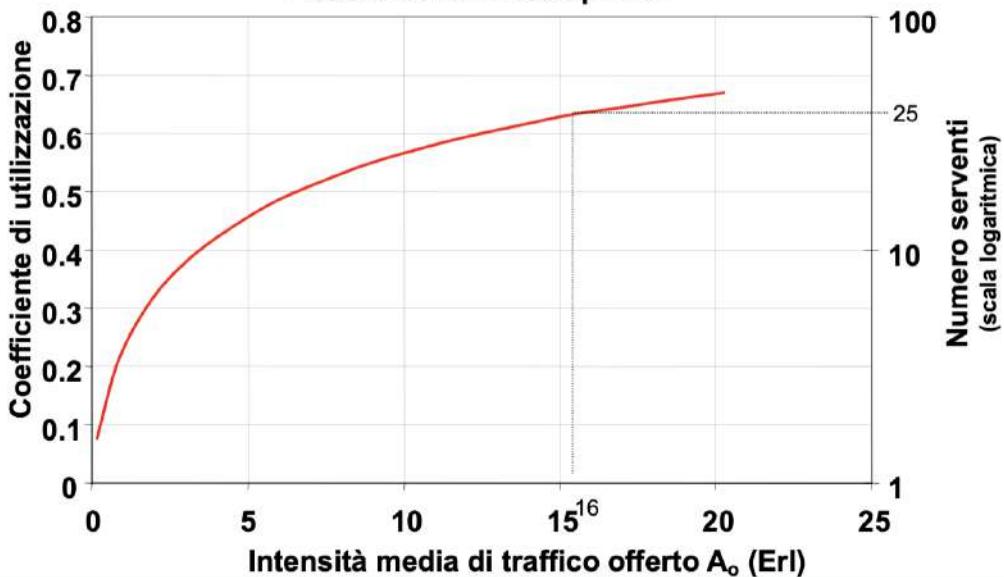


Viene preso il valore m di serventi appena superiore a quello che si interseca.

ρ in funzione di A_0

- A parità di congestione di chiamata, sistemi con elevato numero di serventi presentano, in condizioni di equilibrio statistico, un rendimento **MIGLIORE** rispetto a sistemi con pochi serventi.

Probabilità di rifiuto $\Pi p=0.01$



B di Erlang: dimensionamento del sistema

Dimensionamento del sistema: stimato il traffico offerto A_o e fissato il valore massimo per la probabilità di congestione di chiamata Π_{max} , determinare m :

- trovare il più piccolo valore di m tale per cui

$$E_{I,m}(A_o) \leq \Pi_{max}$$

- tale valore può essere facilmente determinato per tentativi a partire da $m=1$
- il valore effettivo della congestione di chiamata potrà risultare inferiore a Π_{max}

Esempio

Si assume che in media 100 indirizzi IP vengano costantemente utilizzati.

- Intensità di richieste a un server DHCP $A_o = 100$ Erl
- Tale traffico è offerto ad un unico server in modo tale che la probabilità di rifiuto sia minore dell'1%

$$E_{I,m}(A_o) \leq 0.01 \quad \rightarrow \quad m = 117 \text{ indirizzi IP necessari}$$

- Si supponga di ripartire tali richieste uniformemente su n subnet, con $n=2, 4, 10, 25, 50, 100$
- Si può notare come all'aumentare di n aumenta il numero di indirizzi necessari e diminuisce il p di ogni singolo fascio

n	$A_{oi} = (A_o / n)$	m_i	$m = m_i * n$	Π_p	p
1	100	117	117	0.0098	0.8463
2	50	64	128	0.0084	0.7747
4	25	36	144	0.0080	0.6889
10	10	18	180	0.0071	0.5516
25	4	10	250	0.0053	0.3979
50	2	7	350	0.0034	0.2847
100	1	5	500	0.0031	0.1994

Applicazione della Formula B di Erlang.

Conosciamo A_o , conosciamo la probabilità di rifiuto, vediamo chi è m .

m è 117, occorrono dunque 117 IP per ottenere come probabilità di rifiuto 0.01.

In tabella abbiamo come colonne rispettivamente il numero di subnet n , l'intensità di traffico offerto normalizzato al numero di serventi A_{oi} , il numero di serventi calcolati da Erlang in ogni subnet m_i , il numero di serventi

effettivamente utilizzato m , la probabilità d rifiuto Π_p e il coefficiente di utilizzo ρ . La prima riga della tabella rappresenta i valori calcolati nell'esempio. In questo caso, l'effetto asintoto non c'è perché il DHCP serve se ha un IP libero, altrimenti scarta la richiesta.

Più scendiamo nella tabella all'aumentare di n e più il ρ peggiora e di conseguenza vengono sprecate più risorse.

Quando si ha un fascio di richieste di servizio con un numero limitato di serventi e la coda non ha attesa, è bene creare una coda unica e inoltrare le richieste di servizio ad uno dei serventi che a mano a mano di libera.

Questo perché se si utilizzassero code separate magari un servente quando si libera rimane fermo mentre magari avrebbe potuto continuare a lavorare se la coda era unica. Guadagno di multiplazione.

B di Erlang: valutazione delle prestazioni

Valutazione delle prestazioni: dato il numero dei serventi ed il traffico offerto, determinare la probabilità di congestione di chiamata:

- Occorre notare che solitamente è noto il traffico smaltito A_s^* e il numero di serventi m da cui si può stimare A_o attraverso la relazione seguente

$$A_o [1 - E_{I,m}(A_o)] = A_s^*$$

L'asterisco in A_s è solo una notazione, non ha un significato particolare.

- Una volta calcolato A_o si calcola la probabilità di congestione di chiamata

$$\Pi_p = E_{I,m}(A_o)$$

Esempio (1/6)

- Si consideri un centralino telefonico automatico (PABX) di una grande azienda. Il centralino è collegato alla rete telefonica nazionale (RTN) tramite un certo numero di linee bidirezionali.
- Si consideri inoltre che:
 - nell'ora di punta gli utenti attestati al centralino formulano mediamente 140 chiamate dirette verso la RTN;
 - nell'ora di punta il numero di chiamate provenienti dalla RTN e dirette verso gli utenti del PABX è mediamente 180;
 - il flusso delle chiamate sia entranti sia uscenti è Poissoniano;
 - la distribuzione di probabilità delle durate delle conversazioni è di tipo esponenziale negativo con valor medio pari a 3 minuti;
 - la modularità delle linee è pari a 4, ovvero si possono inserire linee solo a gruppi di 4;
 - il PABX è del tipo a perdita pura.
- Si determini il numero di linee necessario a garantire un servizio con congestione di chiamata non superiore all'1%.
- Calcolare inoltre la frequenza massima delle chiamate consentita nell'ora di punta.

Esempio (2/6)

Il PABX può essere modellato con un sistema a coda del tipo $M/M/m/m$ in cui m è il numero di linee tra PABX e RTN

Si calcola il traffico globale offerto. Questo è pari alla somma del traffico uscente

$$A_u = \frac{140}{60} 3 = 7 \text{ Erl}$$

μ è $\frac{1}{3}$, in 3 minuti si smaltisce una chiamata. Quindi, si moltiplica per 3.

60 sono i minuti.

e del traffico entrante

$$A_e = \frac{180}{60} 3 = 9 \text{ Erl}$$

quindi

$$A_o = A_u + A_e = 16 \text{ Erl}$$

Esempio (3/6)

Per calcolare il numero di linee necessario a garantire una probabilità di congestione di chiamata minore dello 0.01 si deve determinare il minimo valore di m tale che

$$E_{1,m}(A_o) \leq 0.01$$

Si ottiene in tal caso $m=25$

A causa del vincolo sulla modularità il numero di linee da inserire sarà pari quindi a $m=28$

Dato tale numero di linee la congestione di chiamata sarà notevolmente inferiore a quella richiesta infatti

$$\Pi_{p,\text{effettivo}} = E_{1,28}(16) = 0.0019$$

1: formula B di Erlang.
28: m serventi (linee).
16: traffico offerto calcolato prima.

Esempio (4/6)

Per determinare la frequenza massima delle chiamate consentita nell'ora di punta si calcola prima il valore di $A_{o,max}$ tale che

$$E_{1,28}(A_{o,max}) \leq 0.01$$

da cui si ricava $A_{o,max} = 18.64$

per cui

$$\lambda_{max} = A_{o,max} \frac{60}{3} \cong 373 \text{ chiamate / ora}$$

Moltiplichiamo $A_{o,max}$ per $\frac{60}{3}$ perché è in minuti, dividiamo per 3 che sono le chiamate da smaltire e quindi si ottengono 373 chiamate all'ora servibili con questa probabilità di blocco. Se le chiamate aumentano allora la probabilità di blocco aumenta.

Esempio(5/6)

Si consideri il PABX dimensionato con 28 linee bidirezionali che lo connettono alla Rete Telefonica Nazionale.

A distanza di tempo dalla sua installazione si vuole valutare la qualità di servizio offerta sapendo che a seguito di una campagna di misure si è riscontrato, nell'ora di punta, un valore di intensità media di traffico smaltito pari a circa 20.42 Erl.

Esempio (6/6)

Dato il traffico smaltito misurato si può ricavare il traffico offerto al sistema risolvendo l'equazione

$$A_o(1 - E_{1,28}(A_o)) = 20.42$$

20.42 è A_s .
Sappiamo la sua formula, invertendola troviamo A_o .

da cui si ha

$$A_o = 21 \text{ Erl}$$

21 Erlang di A_o sono tollerabili?

Per quanto riguarda il valore di congestione di chiamata, si ha

$$\Pi_p = E_{1,28}(21) = 0.0277$$

Per saperlo, inseriamo A_o nella formula. Π_p è diventata 0.0277, non va bene. Tocca aumentare il numero di linee telefoniche.

Il PABX non è più in grado di rispettare il vincolo sul grado di servizio. Le prestazioni sono variate, ad esempio, per un leggero incremento dell'utenza. Bisognerà quindi ridimensionare il numero di linee per riportare la probabilità di rifiuto sotto la soglia dello 0.01

Abbiamo appena visto un sistema orientato alla perdita.

Vediamo ora un sistema orientato al ritardo.

Che succede quando le richieste di servizio non vengono scartate ma accodate?

Esempio (1/3)

Si considerino N terminali di utente che possiamo modellare come sorgenti di traffico dati. Ognuna emette traffico poissoniano con ritmo binario medio pari a λ bit/s e lunghezza dei pacchetti con distribuzione esponenziale negativa di media L.

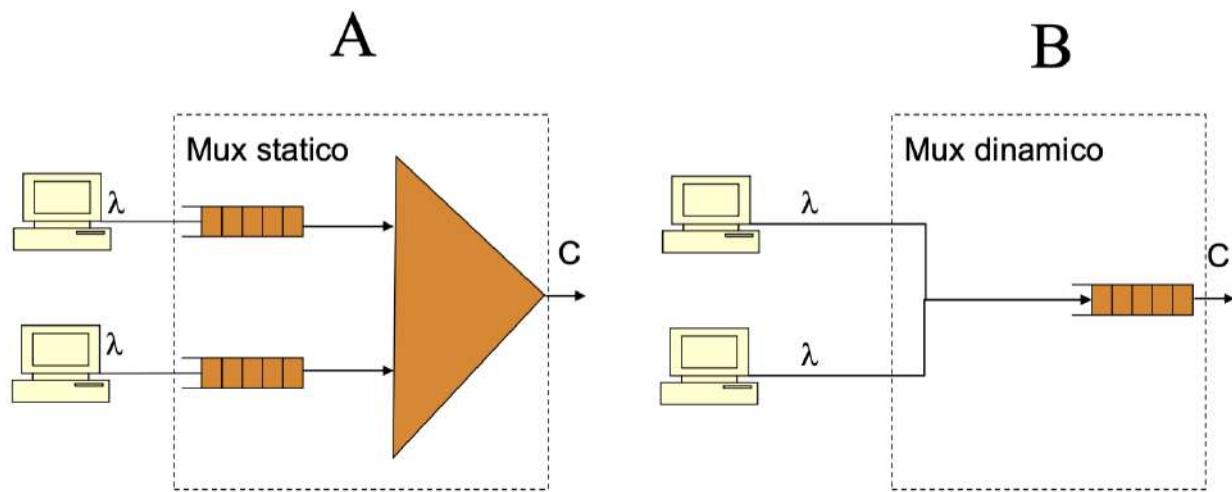
Il traffico prodotto è inoltrato verso un load balancer (multiplatore) con capacità di smaltimento complessiva pari a C.

Si considerino due tecniche di multiplazione, con $N^* \lambda < C$:

- Multiplazione statica: ad ogni utente è assegnato un buffer di dimensione infinita ed una capacità pari a C/N
- Multiplazione dinamica: tutta la capacità è dinamicamente condivisa tra tutte le sorgenti, che utilizzano un unico buffer di dimensione infinita

Si valuti e discuta la prestazioni delle due soluzioni in termini di coefficiente di utilizzazione della capacità C e del tempo medio di sistema T

Esempio (2/3)



Esempio (3/3)

- Assunzioni: flussi statisticamente indipendenti
- Gestione delle code di tipo FIFO

A - N code M/M/1

coefficiente di utilizzazione:

$$\rho = \frac{\lambda}{C} = \frac{\lambda NL}{C}$$

tempo medio di sistema: $T = \frac{\rho}{\lambda(1-\rho)}$

B - 1 coda M/M/1

coefficiente di utilizzazione:

$$\rho = \frac{N\lambda}{C} = \frac{\lambda NL}{C}$$

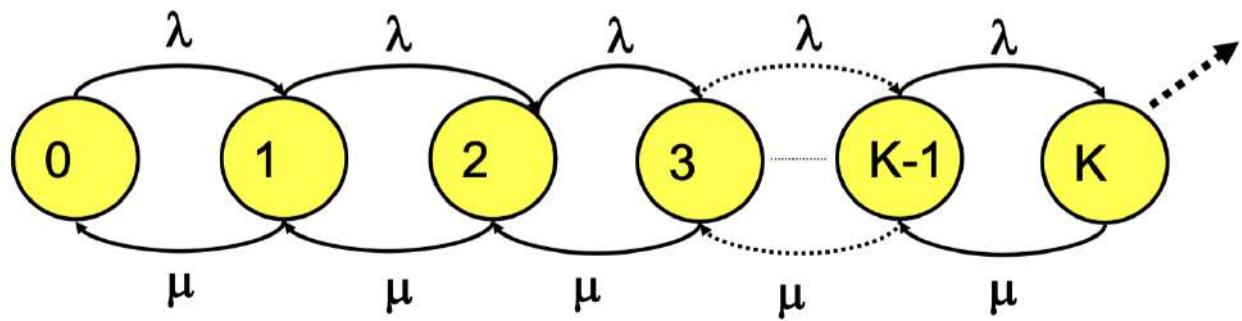
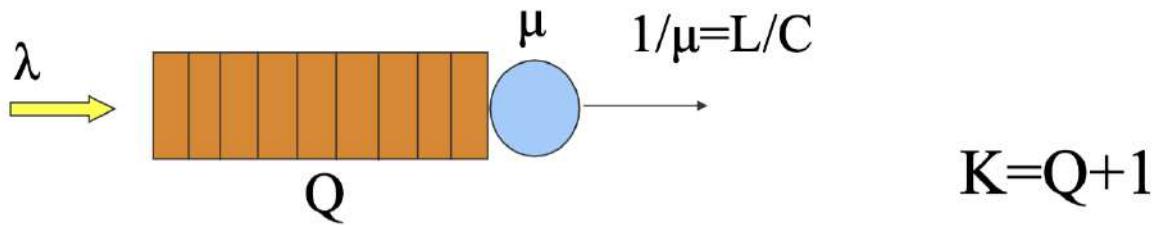
tempo medio di sistema: $T = \frac{\rho}{N\lambda(1-\rho)}$

In entrambi i casi, il coefficiente di utilizzazione è uguale perché le richieste non vengono scartate in nessun caso.

Però, il tempo medio di sistema T nel secondo caso diminuisce.

Lo osserviamo in T perché questo è un sistema orientato al ritardo.

Sistema a coda M/M/1/K



$$P_k = P_0 \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}} \quad \text{dove} \quad P_0 = \frac{1}{1 + \sum_{k=1}^{\infty} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}}$$

Questo è un sistema in cui c'è una coda finita, con un servente.

Se la coda è piena, allora quando arriva un utente potrebbe essere scartato.

È un sistema orientato alla perdita.

K è uguale al numero di utenti che possono stare in coda e in questo caso è pari a $Q + 1$ che è il servente, se c'erano 2 serventi allora era $Q + 2$.

Indicando con $A_0 = \lambda / \mu$ il traffico offerto, si ricavano le probabilità limite di stato

$$\begin{aligned} P_k &= P_0 A_0^k \quad \text{dove} \quad P_0 = \frac{1}{1 + \sum_{i=1}^K A_0^i} = \frac{1}{1 + \sum_{i=1}^K A_0^i} = \frac{1}{1 + \left(\sum_{i=1}^{\infty} A_0^i - \sum_{i=K+1}^{\infty} A_0^i \right)} = \frac{1 - A_0}{1 - A_0^{K+1}} \\ &= \frac{1}{1 + \left(\frac{A_0}{1 - A_0} - \frac{A_0^{K+1}}{1 - A_0} \right)} = \frac{1 - A_0}{1 - A_0^{K+1}} \end{aligned}$$

$$P_k = \begin{cases} \frac{1 - A_0}{1 - A_0^{K+1}} A_0^k & 0 \leq k \leq K \\ 0 & k > K \end{cases}$$

$$A_S = A_0 (1 - P_{rifiuto})$$

Proprietà “PASTA”

Catena di Markov: stazionaria

- Il processo si è avviato dalla condizione statistica di stazionarietà, o
- Il processo dura per un tempo $t \rightarrow \infty$

La probabilità che in un dato istante t il processo si trovi nello stato i è uguale alla probabilità stazionaria che

$$p_i = \lim_{t \rightarrow \infty} P\{N(t) = i\} = \lim_{t \rightarrow \infty} \frac{T_i(t)}{t}$$

$N(t) = i$ significa che ci sono i utenti al tempo t .

T_i = tempo di permanenza del processo nello stato i

$T_i(t)$ è il tempo di permanenza all'interno dello stato i .

Domanda: Per un sistema a coda M/M/1, se t è il tempo di un arrivo, qual è la probabilità che $N(t)=i$?

→ Risposta: Poisson Arrivals See Time Averages (PASTA).

p_n : probabilità stazionaria di essere nello stato n .

a_n : distribuzione di probabilità degli stati al tempo t^- lo stato che l'utente osserva quando arriva è N .

Dimostriamo che lo stato osservato dall'utente quando arriva è p_n .

Probabilità stazionarie:

$$p_n = \lim_{t \rightarrow \infty} P\{N(t) = n\}$$

Probabilità stazionarie in corrispondenza di un arrivo:

$$a_n = \lim_{t \rightarrow \infty} P\{N(t^-) = n \mid \text{arrival at } t\}$$

$t \rightarrow \infty$ sta ad indicare che siamo a regime.

Ipotesi LAA (Lack of Anticipation): I futuri tempi di interrivo e i tempi di servizio dei clienti arrivati precedentemente sono indipendenti

→ Teorema: In un sistema a coda che soddisfa l'ipotesi LAA:

1. Se il proceddo degli arrivi è di Poisson si ha che

$$a_n = p_n, \quad n = 0, 1, \dots$$

2. Quello di Poisson è il solo processo avente questa proprietà (condizione necessaria e sufficiente)

Proprietà PASTA

La proprietà PASTA si applica ad altri processi di arrivo?

Esempio:

Arrivi deterministici ogni 10 sec

Tempo di servizio deterministico di 9 sec

- ➡ In corrispondenza di ogni arrivo il sistema è vuoto $a_1=0$
- ➡ Il tempo medio in cui un solo utente è presente nel sistema $p_1=0.9$

p_0 è 0.1, $a_1 = 0$
e $a_0 = 1$.

La proprietà
PASTA in questo
caso non vale.

Le medie osservate dall'utente non sono necessariamente medie temporali del sistema

Quindi $a_n = p_n$
dimostrato.

L'aleatorietà non è di aiuto, a meno che sia generata da un processo di Poisson!

Proprietà PASTA: dimostrazione

Sia $A(t, t+\delta)$, l'evento in cui vi sia un arrivo in $[t, t+\delta]$

Se che un utente arriva in t , la probabilità $a_n(t)$ di trovare il sistema nello stato n è data da

$$P\{N(t^-) = n \mid \text{arrival at } t\} = \lim_{\delta \rightarrow 0} P\{N(t^-) = n \mid A(t, t+\delta)\}$$

$A(t, t+\delta)$ è indipendente dallo stato del sistema prima di t , $N(t)$

- $N(t)$ è determinato dai tempi di arrivo $< t$, e dai corrispondenti tempi di servizio
- $A(t, t+\delta)$ è indipendente dallo stato del sistema, ossia dagli arrivi $< t$ [Poisson] e dai tempi di servizio degli utenti arrivati $< t$ [LAA]

$$a_n(t) = \lim_{\delta \rightarrow 0} P\{N(t^-) = n \mid A(t, t+\delta)\} = \lim_{\delta \rightarrow 0} \frac{P\{N(t^-) = n, A(t, t+\delta)\}}{P\{A(t, t+\delta)\}}$$

$$= \lim_{\delta \rightarrow 0} \frac{P\{N(t^-) = n\} P\{A(t, t+\delta)\}}{P\{A(t, t+\delta)\}} = P\{N(t^-) = n\}$$

$$a_n = \lim_{t \rightarrow \infty} a_n(t) = \lim_{t \rightarrow \infty} P\{N(t^-) = n\} = p_n$$

Quando
arriva un
utente lo
stato che
osserva
dipende
dagli utenti
arrivati
prima di lui
e da come
sono stati
serviti.

Esempi

Esempio 1: Arrivi non di Poisson

Tempi di interarrivo IID distribuiti uniformemente fra 2 and 4 sec

Tempi di servizio deterministici di 1 sec

• In corrispondenza di ogni arrivo il sistema è vuoto, quindi $a_1 = 0$. e $a_0 = 1$

• $\lambda=1/3$, $T=1 \rightarrow N=T\lambda=1/3 \rightarrow p_0=2/3$, $p_1=1/3$ p_0 e p_1 probabilità che è vuoto e probabilità che c'è 1 utente.

Diverso dalle a , quindi non è PASTA.

Esempio 2: mancanza dell'ipotesi LAA

Arrivi di Poisson

Tempo di servizio dell'utente i : $S_i = \alpha T_{i+1}$, $\alpha < 1$

• In corrispondenza di ogni arrivo il sistema è vuoto, $a_1 = 0$.

• Il tempo medio in cui un solo utente è presente nel sistema $p_1 = \alpha$

Se manca l'indipendenza tra i due processi allora la proprietà PASTA non vale. Non basta avere processi di Poisson, occorre anche l'indipendenza tra tempi di arrivo e i tempi di servizio.

Distribuzione dopo la partenza

$$d_n = \lim_{t \rightarrow \infty} P\{X(t^+) = n \mid \text{departure at } t\}$$

PASTA al contrario.

Probabilità di stato stazionarie dopo una partenza:

Usando la stessa proprietà di Markov:

• I limiti di a_n e d_n esistono e coincidono

• $a_n = d_n$, $n = 0, 1, \dots$

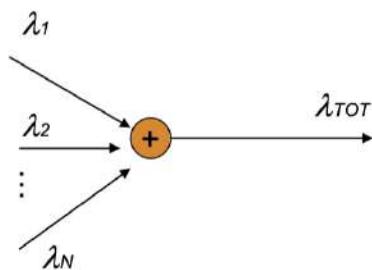
• In condizioni stazionarie, il sistema appare stocasticamente identico agli utenti che arrivano e che lasciano il sistema.

Arrivi di Poisson + LAA: un utente che arriva e un utente che lascia il sistema vedono il sistema stesso con la stessa statistica osservabile in un tempo aleatorio.

Ancora sui processi di Poisson...

L'aggregazione di N processi di Poisson indipendenti di parametro λ_i , $i=1\dots N$, è un processo di Poisson di parametro

$$\lambda_{TOT} = \sum_i \lambda_i$$

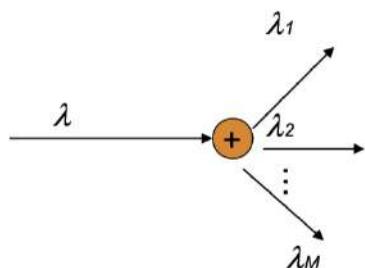


Ogni flusso di arrivo è un processo di Poisson.

Se li aggreghiamo, il risultato è ancora un processo di Poisson il cui parametro λ è il totale della somma dei vari λ dei vari flussi. Gli arrivi devono essere i.i.d.

Ancora sui processi di Poisson...

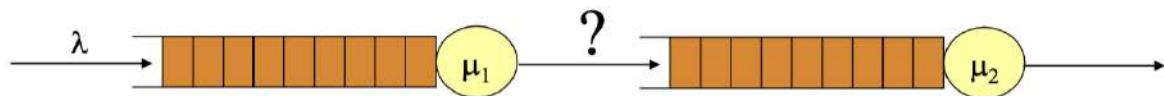
La separazione statistica di un processo di Poisson di parametro λ con probabilità p_1, p_2, \dots, p_M genera M processi di Poisson di parametro $\lambda_i = \lambda p_i$, $i=1,\dots,M$.



Dato un arrivo, lo inseriamo in una delle linee con probabilità $1/N$ in modo indipendente dall'arrivo precedente.
Se vale questo, allora i singoli flussi sono flussi di Poisson.

Teorema di Burke

Date due code in "tandem", qual è la distribuzione dei tempi di interarrivo alla seconda coda, se la prima è una M/M/1?



Tale distribuzione sarà equivalente a quella dei tempi di interpartenza ($D(t)$) dalla coda 1

Due code disposte una dietro l'altra. La seconda come possiamo studiarla?
Arrivano nella seconda coda quando sono stati serviti dalla prima, che è M/M/1.
Ma il successivo ancora quando arriva?
O quando esce il primo dalla coda 2 o quando c'è già uno dentro.
Nel primo caso, il tempo di interpartenza è uguale al tempo di uscita dalla seconda coda.
Nel secondo caso aspettiamo un tempo di interarrivo e poi uno di servizio.

Teorema di Burke

D(t) tempo di interarrivo alla coda 2.

Indichiamo con

- $D^*(s)$ la trasformata di Laplace di $D(t)$ Alla coda 1.
- $B^*(s)$ la trasformata di Laplace di $B(t)$ (distribuzione dei tempi di servizio) ↗

Quando un cliente parte da una coda (cioè ha ricevuto il suo servizio) può verificarsi uno solo dei seguenti eventi:

- un altro cliente è presente sulla coda 1 e sarà subito servito
- la coda 1 è vuota, quindi bisognerà attendere un tempo uguale alla somma di due contributi prima di un nuovo arrivo alla coda 2:
 - il tempo fino all'arrivo di un altro cliente
 - il suo tempo di servizio

Teorema di Burke

Nel primo caso risulta

È il tempo di servizio che determina il tempo di interarrivo.

- $D^*(s)|_{\text{coda non vuota}} = B^*(s)$

Nel secondo caso, ho la somma di due variabili aleatorie indipendenti, perciò la variabile aleatoria somma sarà la convoluzione delle pdf, quindi, nel dominio di Laplace, il prodotto delle trasformate delle distribuzioni

- $D^*(s)|_{\text{coda vuota}} = B^*(s) \lambda/(s+\lambda)$

Poiché il servente è di tipo esponenziale, si ha inoltre che

- $B^*(s) = \mu_I/(s+\mu_I)$

Teorema di Burke

Poiché la probabilità di avere il sistema non vuoto è pari $\rho = \lambda/\mu_I$, si ottiene che

- ◦ $D^*(s) = (1-\rho) D^*(s)|_{\text{coda vuota}} + \rho D^*(s)|_{\text{coda non vuota}}$ ρ è la probabilità che il sistema non sia vuoto.

Sostituendo i valori precedenti si ottiene

- $D^*(s) = (1-\rho) (\lambda/(s+\lambda)) (\mu_I/(s+\mu_I)) + \rho (\mu_I/(s+\mu_I))$

che risulta uguale a

- $D^*(s) = (\lambda/(s+\lambda)) = A^*(s) \Rightarrow D(t) = A(t) = 1 - e^{-\lambda t} \text{ per } t \geq 0$

Quindi i tempi di interpartenza sono distribuiti esponenzialmente con lo stesso parametro dei tempi di interarrivo:

- **la coda 2 può essere trattata come una M/M/1 indipendente dalla coda 1 !**

Burke estende questo risultato alle code M/M/m

Distribuzione di interarrivo nella seconda coda.

Tempo di interarrivo alla seconda coda è la trasformata di Laplace con parametro lambda.

Quindi se il tempo di servizio è esponenziale, allora la coda 2 può essere modellata come un modello M/M/1.

Teorema di Burke

Considerando le operazioni viste sui processi di Poisson, il Teorema di Burke implica che se si connettono dei server (cioè sistemi a coda) in modalità feed-forward allora ogni nodo della rete di code può essere esaminato SINGOLARMENTE come se fosse un sistema a coda indipendente.

Questo risultato è generalizzato dal Teorema di Jackson per le reti di code aperte

È fondamentale perché significa che è possibile fare come si pare con le code: sommarle, dividerle, allungare e se entra Poisson esce sempre Poisson, potendo analizzare come un modello $M/M/m$.

A patto che non ci siano dei feedback (verso l'indietro).

Il feedback dà fastidio perché se il traffico per metà entra e per metà esce, allora quello che rientra crea una correlazione tra i tempi di servizio e i tempi di interarrivo nella coda e se ciò avviene tutti i sistemi fatti si benedicono.

Si ricorre allora al Teorema di Jackson.

Teorema di Jackson

Si consideri una rete di code formata da K nodi (sistemi a coda) che soddisfano le seguenti tre condizioni:

- Ogni nodo contenga c_k serventi aventi tempo di servizio distribuiti esponenzialmente con parametro μ_k .
- Gli utenti provenienti dall'esterno giungono al generico nodo k secondo un processo di Poisson con parametro λ_k .
- Quando un utente è servito al nodo k è trasferito “istantaneamente” al generico nodo j con probabilità p_{kj} oppure esce dalla rete con probabilità $1 - \sum_j p_{kj}$

Teorema di Jackson

Il tasso degli arrivi al generico nodo k sarà:

$$\Lambda_k = \lambda_k + \sum_{j=1}^K p_{jk} \Lambda_j$$

Indicando con $p(n_1, \dots, n_K)$ la probabilità congiunta stazionaria di stato nei nodi, se

$$\Lambda_k < c_k \mu_k \quad \forall k, \quad \text{Quindi, sono code stabili.}$$

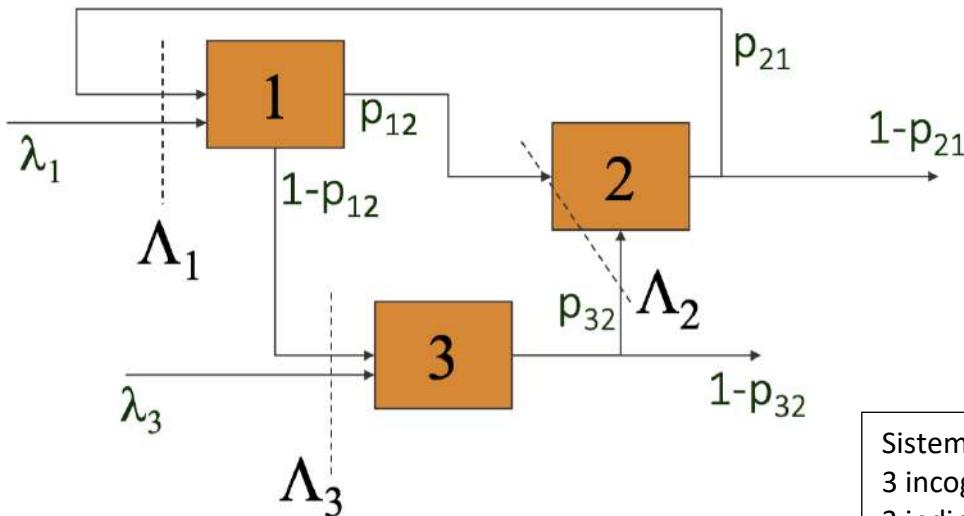
Allora $p(n_1, \dots, n_K) = p_1(n_1) p_2(n_2) \dots p_K(n_K)$ Utenti presenti in ogni singola componente della rete.

dove $p_i(n_i)$, $i=1, \dots, K$, è la probabilità stazionaria che il nodo i si trovi nello stato n_i , (vi siano n_i utenti nel sistema a coda i), modellandolo come un sistema $M/M/c_i$ con processo degli arrivi di Poisson con parametro Λ_i e tasso di servizio μ_i

La probabilità di avere n utenti nella prima coda, n utenti nella seconda, ecc. la possiamo scrivere come il prodotto delle singole probabilità. Come se ogni nodo fosse isolato, con modello $M/M/c$, come se fosse un processo di Poisson.

Teorema di Jackson

Rete generica con traffico di ogni tipo.



$$\begin{cases} \Lambda_1 = \lambda_1 + p_{21} \Lambda_2 \\ \Lambda_2 = p_{12} \Lambda_1 + p_{32} \Lambda_3 \\ \Lambda_3 = \lambda_3 + (1-p_{12}) \Lambda_1 \end{cases} \rightarrow \Lambda_1, \Lambda_2, \Lambda_3$$

Sistema a 3 equazioni con 3 incognite. Sono tutte e 3 indipendenti.
Una volta in possesso dei tre λ , possiamo studiare ogni singolo nodo 1, 2 o 3 e modellarlo come un sistema $M/M/c$.

Ipotesi di indipendenza di Kleinrock

- Tempi di interarrivo indipendenti alle varie code della rete
- Tempo di servizio di un generico pacchetto nelle varie code indipendente.
 - La lunghezza del pacchetto è random ogni volta che un pacchetto è trasmesso attraverso un collegamento.
- Tempo di servizio e tempi di interarrivo statisticamente indipendenti.

Le assunzioni sono state validate attraverso risultati sperimentali e simulazioni. In tal caso si ha che:

La distribuzione stazionaria degli stati approssima quella descritta dal teorema di Jackson

L'approssimazione è accettabile quando:

- Il processo degli arrivi dai punti in ingresso alla rete è un processo di Poisson.
- Il tempo di trasmissione del pacchetto è una variabile aleatoria approssimabile mediante un'esponenziale.
- Molto flussi di pacchetti sono multiplati in ogni collegamento.
- La rete è densamente connessa
- Vale per un'intensità di traffico da moderata a pesante.

Potremmo utilizzare il teorema di Jackson, si otterrebbero risultati verosimili. L'indipendenza in questione riguarda i tempi di servizio e i tempi di arrivo.

Capitolo 8: VxLAN (Virtual Extensible LAN)

Tecnologia che prevede una virtualizzazione più spinta dove vengono create reti LAN distribuite mettendole sopra al protocollo di trasporto.

Dobbiamo aspettarci un po' di overhead.

Vanno a risolvere però molti problemi, riducendo la complessità nella gestione delle reti. **Le reti VxLAN sono un'evoluzione delle VLAN e sono state introdotte per superare le limitazioni delle VLAN tradizionali in quanto con le Cloud, per via delle loro grandi dimensioni, non riescono a performare.**

I Data Center tradizionali prevedono l'utilizzo di tre livelli:

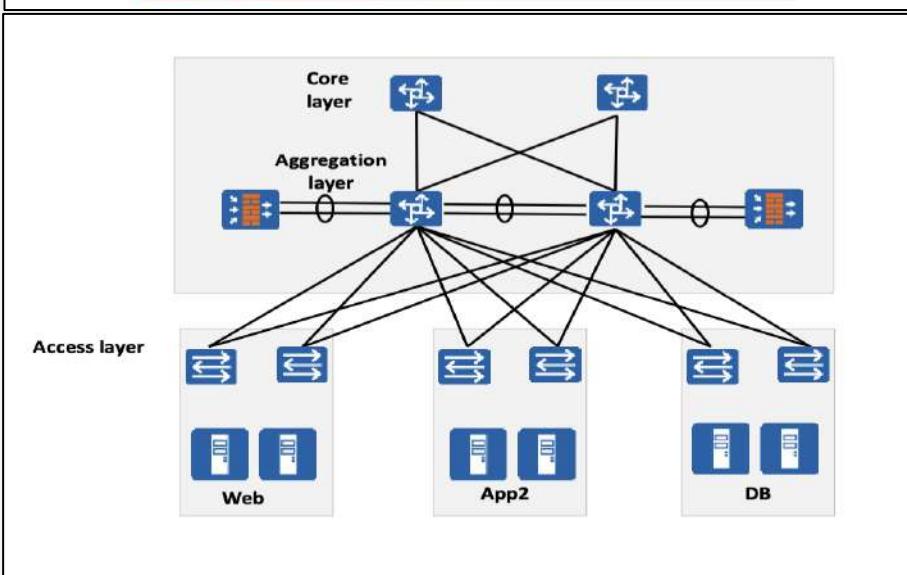
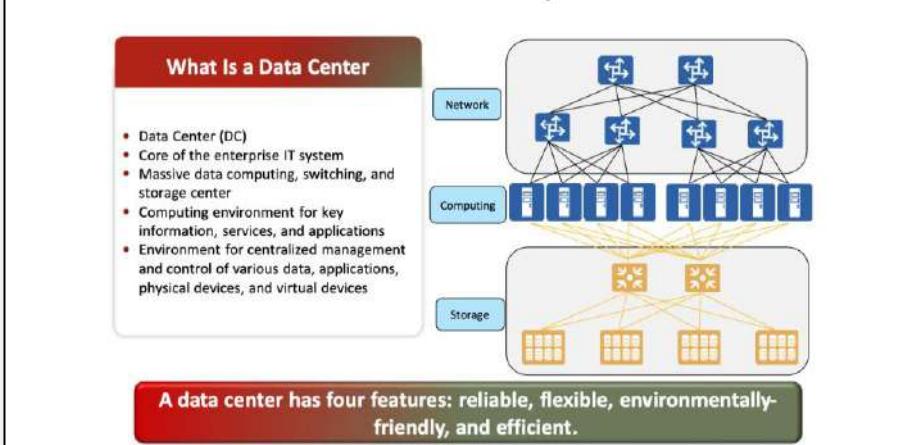
Access layer, Aggregation layer e Core layer.

Inoltre, prevedono principalmente un traffico Sud-Nord: verticale.

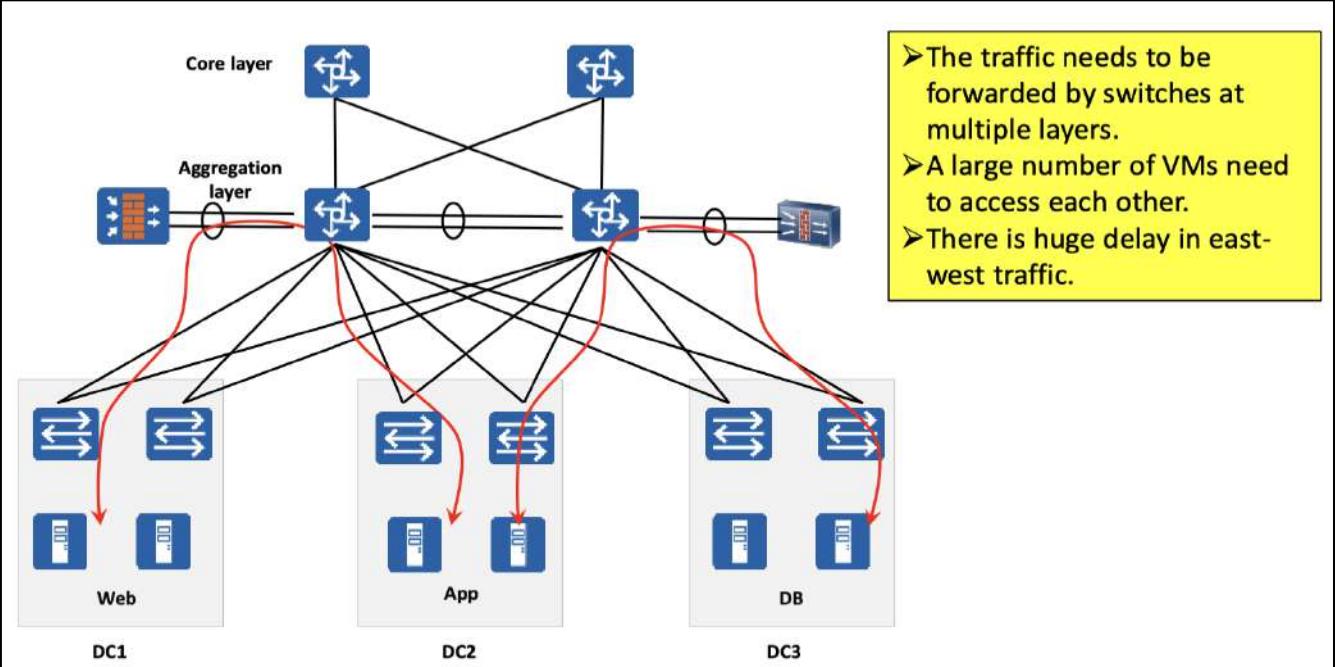
Questa è la struttura classica poi le cose sono scalate e sono comparse nuove sfide: è necessaria una bassa latenza, non soltanto per gli utenti ma anche per l'interazione tra le macchine virtuali all'interno del datacenter stesso.

Se si hanno tanti server in una certa area, non si possono gestire come un unico dominio di collisione ma occorre andare a segmentare la rete.

Data Center Basic Concepts and Features



1^a sfida: esigenza di avere una bassa latenza tra i nodi Compute



L'introduzione delle VM ha avuto conseguenze notevoli per i datacenter.

Non essendo fisiche, non sono legate a una posizione in un armadio.

Dove si possono spostare le VM? Fin dove?

Se restano nella stessa LAN, spostandole cambia poco; non occorre cambiare indirizzo IP se, ad esempio, ci spostiamo di un armadio.

Se, invece, occorre spostarle in un'altra rete con una subnet diversa, allora è un problema sia per la gestione della configurazione sia per gli endpoint del servizio che gli utenti vedono dall'esterno. Occorre l'intervento dei DNS.

I datacenter tradizionali si sposano male con le VM, peggio ancora con i container.

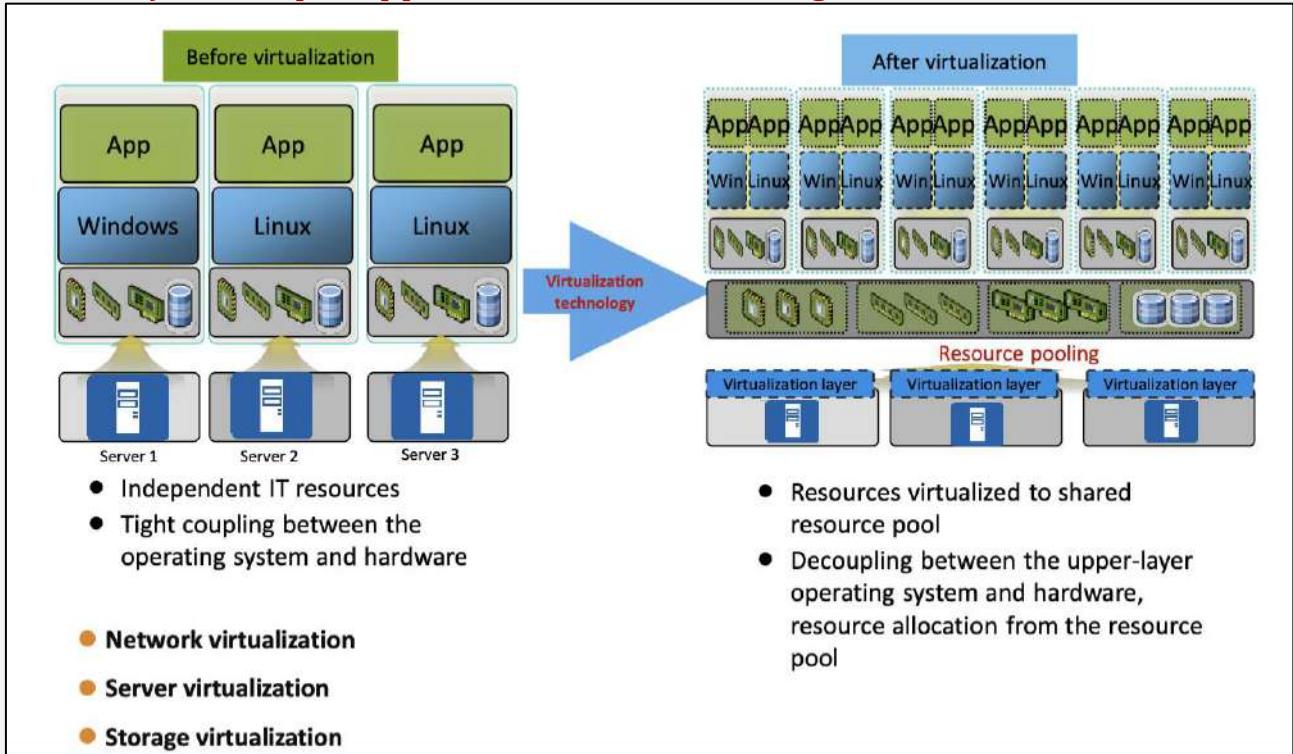
Occorre creare dei tunnel, degli applicativi sopra al protocollo di trasporto che vanno ad encapsulare il traffico, sbattendolo poi dall'altra parte e rimettendolo nella rete a strato 2. Bisogna coinvolgere tanti protocolli, incluso l'ARP.

Il protocollo ARP (Address Resolution Protocol) è utilizzato all'interno di una rete locale per mappare gli indirizzi IP degli host con i loro indirizzi MAC.

Attraverso **ARP request** e **ARP response**, le associazioni tra gli indirizzi IP e gli indirizzi MAC vengono risolte e memorizzate nelle tabelle ARP degli host per consentire un'efficace comunicazione tra di loro.

Le reti VxLAN semplificano il networking.

2^a sfida: ampia applicazione della tecnologia di virtualizzazione



Con l'emergere di nuove tecnologie e applicazioni, nuovi servizi come la migrazione delle macchine virtuali, la sincronizzazione dei dati, il backup dei dati e il calcolo collaborativo vengono implementati nei DC. Ad esempio, un utente prima di aggiornare e manutenere un server, può migrare le macchine virtuali sul server a un altro server per garantire la continuità del servizio durante l'aggiornamento. Dopo l'aggiornamento e la manutenzione del server, l'utente può eseguire la migrazione delle macchine virtuali sul server originale. **Tutto questo aumenta incredibilmente il traffico Est-Ovest: orizzontale**, che ad oggi è diventato predominante rispetto al traffico Sud-Nord.

Quando i microservizi comunicano tra loro generano traffico Est-Ovest.

Se si utilizzano centinaia di microservizi, viene da sé.

La capacità di isolamento delle reti tradizionali è limitata, il campo ID che utilizzano le reti VLAN è di soli 12 bit, il che significa che possono isolare e identificare univocamente al massimo 4096 reti VLAN.

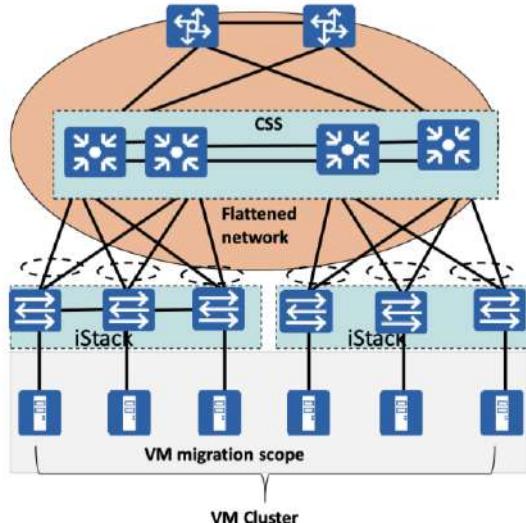
Pochi per gestire migliaia di Tenant.

Inoltre, **per garantire la continuità dei servizi durante la migrazione delle VM, gli indirizzi IP e gli indirizzi MAC delle VM devono rimanere invariati prima e dopo la migrazione**. Ciò significa che la migrazione delle VM deve avvenire in un dominio di strato 2. Tuttavia, **la migrazione delle VM in domini di livello 2 nelle reti di datacenter tradizionali è molto limitata**. Per

migrazione di una VM si intende che prima se ne accende un'altra e quando è pronta si spegne la prima.

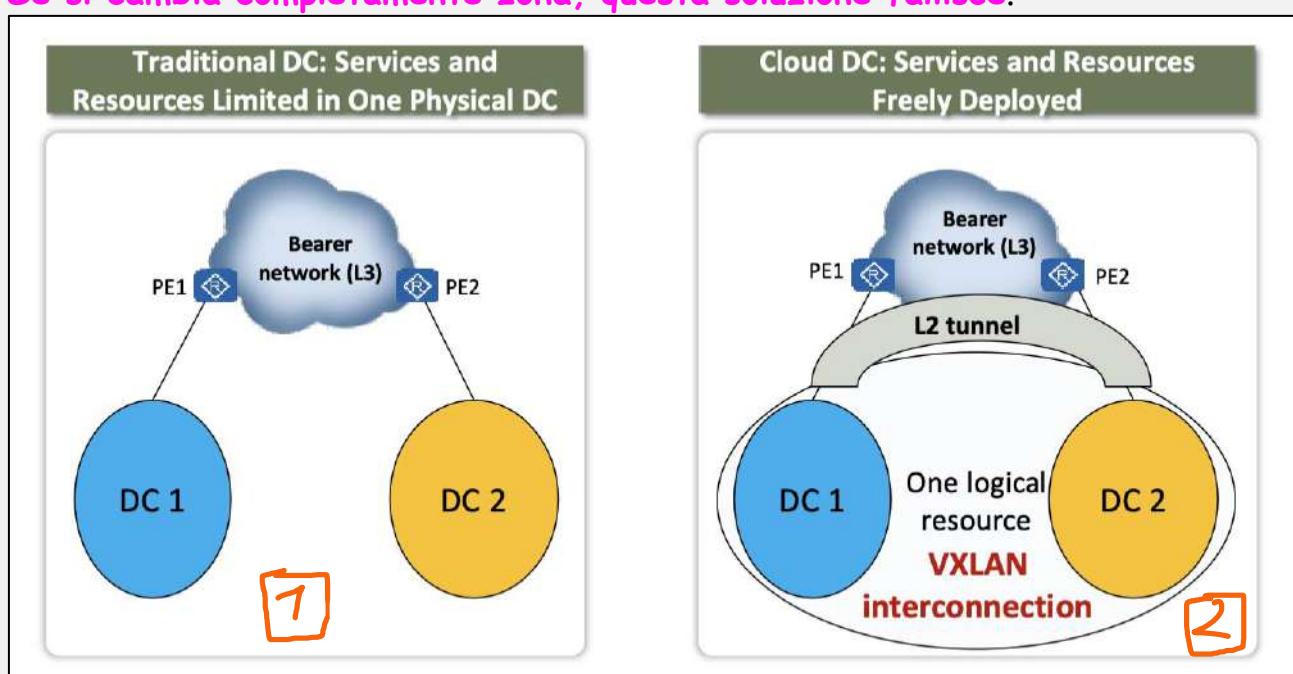
Sarebbe bene mantenere la configurazione che gli utenti esterni già vedono.

Possible Solution: Topology Simplification



- The cluster switch system (CSS) and iStack technologies are used to simplify the network topology.
- Servers are deployed in a flattened network architecture.
- VMs are deployed on a large Layer 2 network, so **VM IP addresses do not need to be changed** during the migration.
- Services are not affected, allowing for **flexible migration**.

Una delle prime soluzioni fu l'utilizzo del **CSS** (cluster switch system) che consiste nell'aggregare gli switch facendoli funzionare come un unico switch. Vengono visti come un unicum apparato, le VM anche se si spostano non devono cambiare indirizzi IP **ma sempre e solo se sono dentro lo stesso datacenter**. Se si cambia completamente zona, questa soluzione fallisce.



1) **DC Tradizionale**: 2 datacenter separati da una rete di stato 3.

Sono 2 domini, se si è solo dentro un DC ok, se si migra si hanno problemi.

2) **DC Cloud**: implementano dei **tunnel**. DC1 a Perugia, DC2 a Sidney e tunnel di strato 2. Non si opera a strato 2 ma si crea un'applicazione overlay.

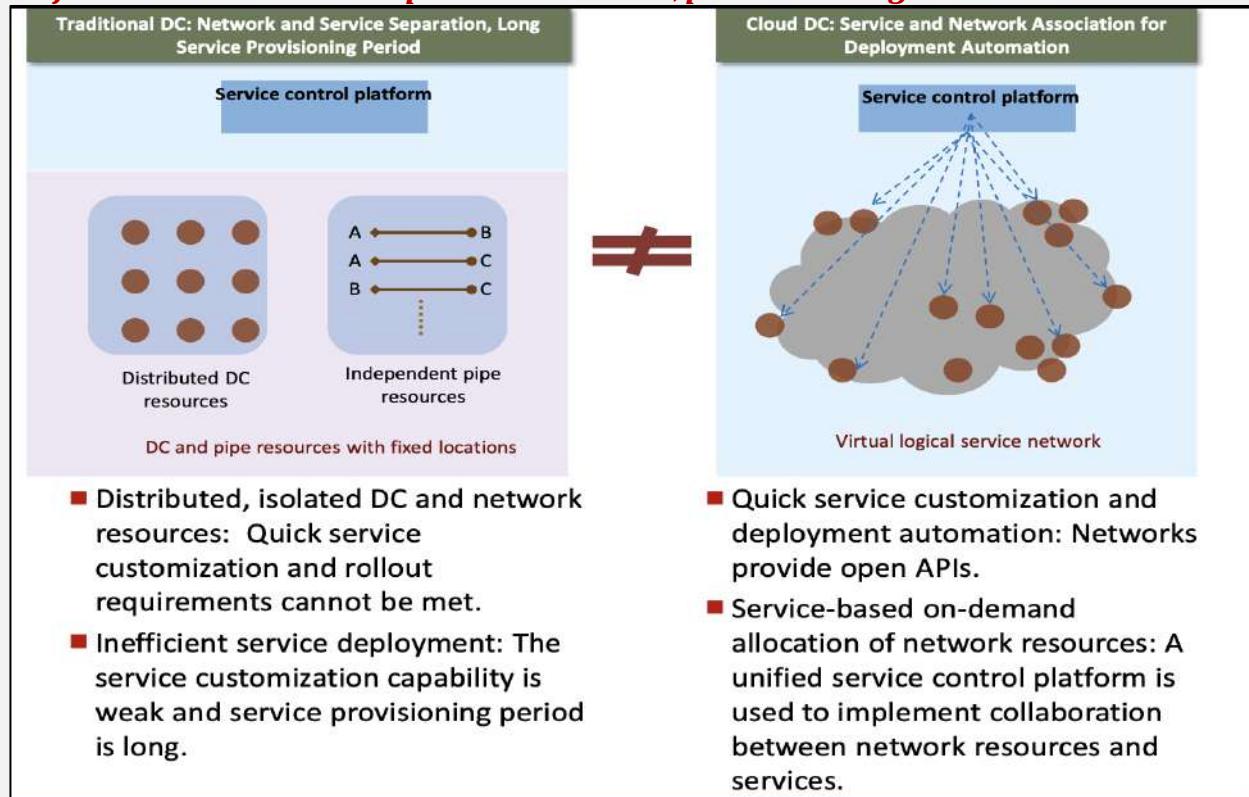
Gli endpoint del tunnel definiscono la complessità.

Se un utente nel DC1 invia un ARP request, allora riceverà l'ARP response a prescindere se il destinatario si trovi in DC1 o DC2.

Nei datacenter tradizionali, servizi e risorse sono limitati in un DC fisico.

Nei datacenter Cloud, servizi e risorse sono distribuiti liberamente.

3^a sfida: innovazione rapida dei servizi, provisioning automatico dei servizi



Soluzione a tutti i problemi: VxLAN

Le reti VxLAN consentono di superare le limitazioni delle VLAN tradizionali, come il numero limitato di VLAN ID disponibili e la limitata scalabilità.

Con VxLAN è possibile creare un numero molto maggiore di segmenti di rete virtuali rispetto alle VLAN tradizionali, in quanto si utilizza un'ID di rete da 24 bit che, in questo caso, prende il nome di VNI (VxLAN Network Identifier).

VxLAN incapsula i pacchetti Ethernet inviati dalle VM in pacchetti IP, trasmettendoli su rotte di rete IP per costruire una grande rete di strato 2.

La rete è a conoscenza solo dei parametri di rete incapsulati.

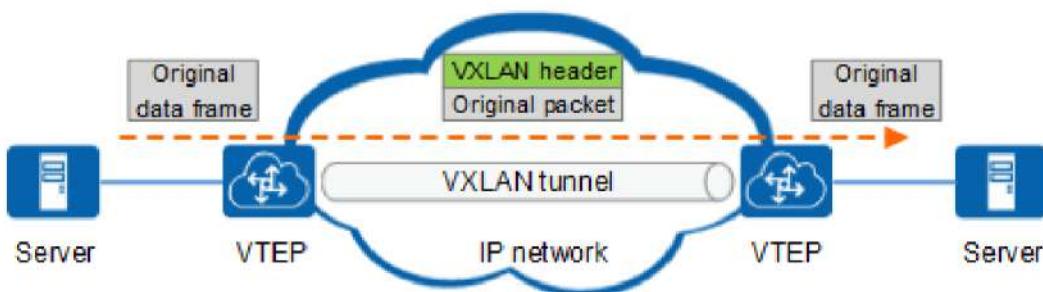
Ciò riduce il numero di indirizzi MAC richiesti dalle grandi reti di strato 2.

Pertanto, la migrazione delle VM non è più limitata dall'architettura di rete.

Il concetto principale è la creazione di opportuni tunnel virtuali

VXLAN network model

2 isole



- Overlay network: VXLAN
- Underlay network: IP

Abbiamo 2 VLAN differenti, in mezzo c'è internet.

I gateway di queste 2 VLAN fanno il mestiere di router. In ogni router viene eseguito un **demone** chiamato **VTEP**, ovvero l'**endpoint virtuale di un tunnel VXLAN** che serve a trasportare i pacchetti da una VLAN all'altra dando la sensazione all'utente di trovarsi sempre nello stesso dominio di broadcast.

Si modificano i pacchetti in transito, impostando un Header VxLAN.

Il frame di dati originale inviato da un server A viene incapsulato all'interno di pacchetti UDP con l'aggiunta dell'Header VXLAN, per poi essere inoltrato con la modalità di trasmissione di rete IP tradizionale. Dopo che il pacchetto UDP è arrivato al VTEP destinatario, tale endpoint rimuove l'intestazione esterna (l'Header VxLAN) e invia il frame di dati originale al server B di destinazione.

Dal punto di vista delle VM, sembra che le comunicazioni avvengano a livello 2.

Vengono così preservati gli indirizzi MAC delle VM.

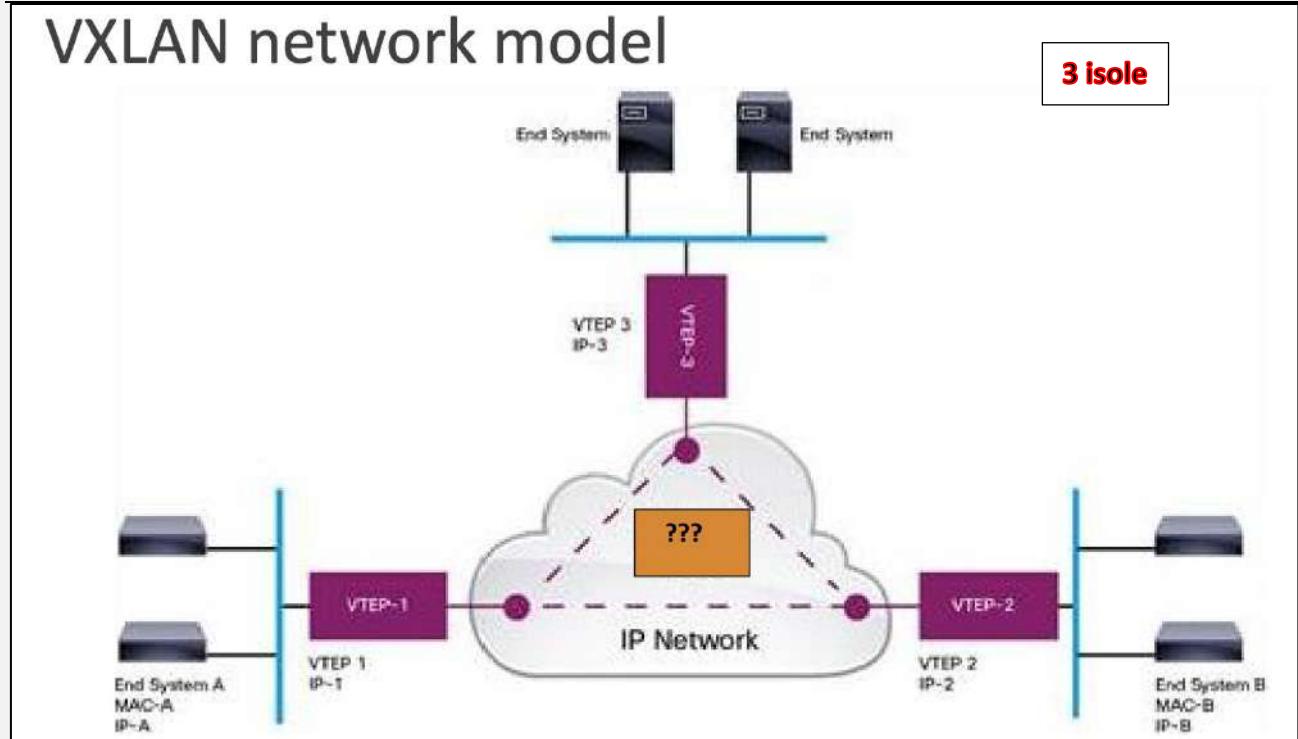
La rete sottostante è la rete IP, mentre la rete sovrastante è la rete VxLAN.

I VTEP possono essere fisici o virtualizzati via software.

Il VTEP è responsabile della creazione, configurazione e gestione dei tunnel VXLAN all'interno dell'infrastruttura di rete.

Essendo virtualizzabile, il VTEP può essere parte integrante di un sistema di gestione del network, come un controller SDN (Software Defined Networking), o di un orchestratore di rete, come OpenStack Neutron.

I VTEP forniscono un'interfaccia interna per la configurazione dei tunnel VXLAN e per la gestione delle operazioni di rete ad esse associate. Se, invece, i VTEP devono interagire, ad esempio, con Neutron allora vanno ad utilizzare le API fornite da Neutron stesso.

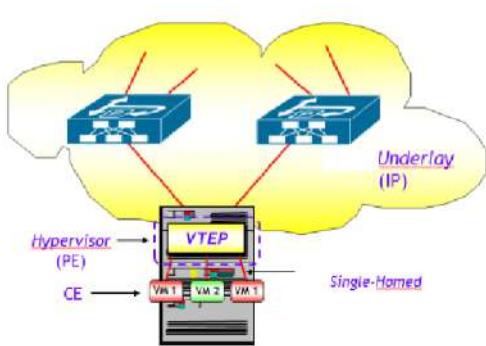


In caso di molteplici reti VLAN (isole) dove finisce l'ARP request?

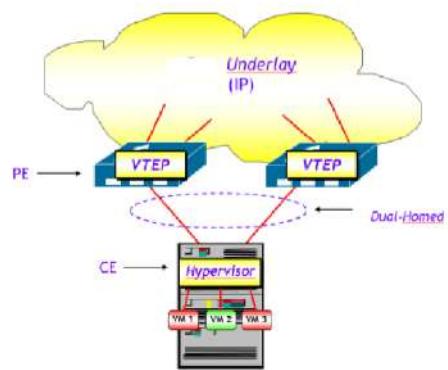
Come viene gestito in mezzo alla rete IP?

Ogni VLAN dispone del rispettivo VTEP che rappresenta il punto di interconnessione con le altre VLAN. Se si conoscessero gli indirizzi IP dei VTEP si risolverebbe il problema. Ma questo significherebbe che tutti i VTEP dovrebbero conoscere tutti gli endpoint delle varie isole. Problema serio.

Where is VTEP implemented?



The VTEP function is implemented by software and located in a Hypervisor (e.g. KVM, Hyper-V, VMware NSX, etc.).



The VTEP function is hardware implemented and located in a TOR switch

Dove può essere implementato il VTEP?

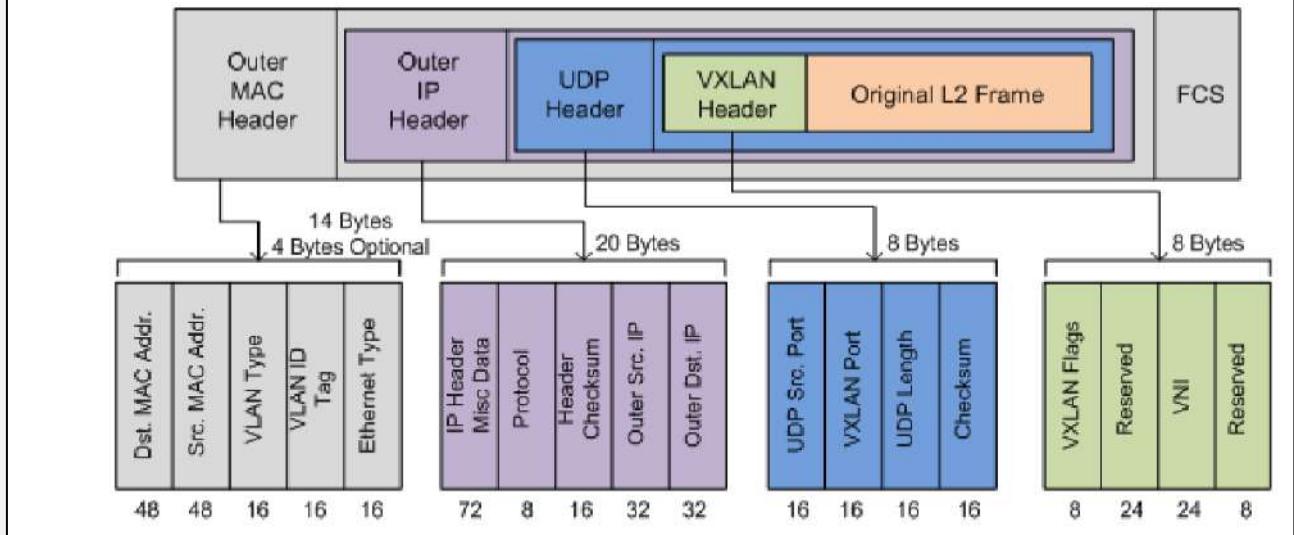
Potrebbe essere implementato via **software dentro l'hypervisor**, consentendo alle VM di collegarsi con le reti VLAN di tutto il mondo come se stessero utilizzando una sorta di VPN, in modo Single-Homed (il dispositivo ha una sola interfaccia di rete attiva per la connettività in rete).

In questo caso la rete underlay (IP) non è a conoscenza di VxLAN.

Potrebbe essere implementato via **hardware nel router della VLAN**, consentendo alle VM di un hypervisor di uscire a piacimento.

In questo caso i dispositivi non sono a conoscenza di VxLAN.

VXLAN packet format



Occorre posizionarsi sopra al protocollo di trasporto (overlay), spesso UDP.

In questo modo non viene posto alcun limite su ciò che c'è sotto.

Si ha la massima flessibilità. Tuttavia, **occorre portare dietro tutto lo strato protocollare per ogni singolo pacchetto**. Il pacchetto viene incapsulato con un **Header VXLAN** contenente l'identificativo della VXLAN, il VNI.

Il tutto si incapsula poi con un **Header UDP** (più leggero del TCP), per evitare di appesantire ulteriormente il pacchetto. Il tutto si incapsula con un **Header IP** contenente gli IP dei VTEP mittente e destinatario. Header MAC contiene il tutto e indica il campo **Dst. MAC che è il next hop verso il VTEP destinatario**.

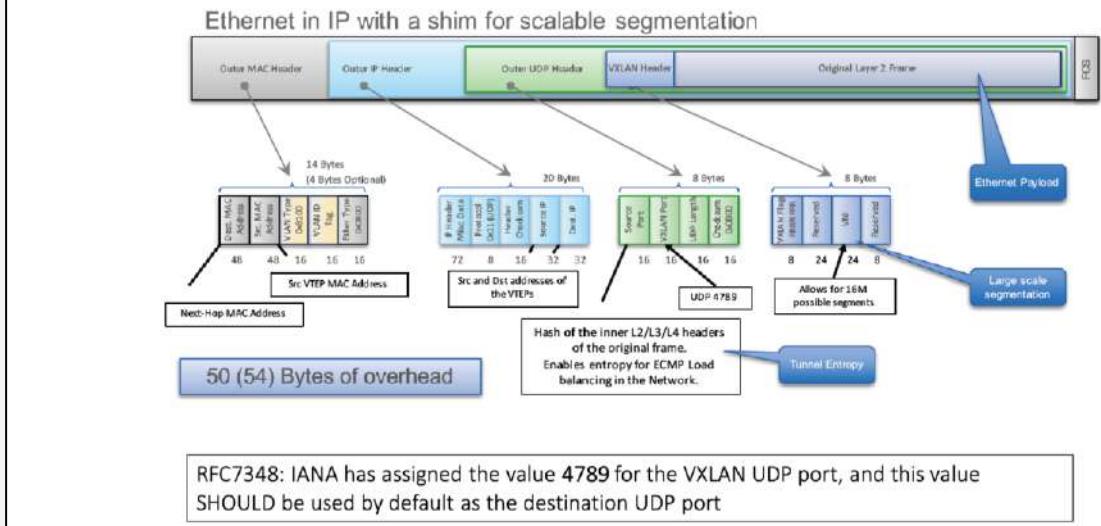
Alla fine, viene fuori un enorme pacchetto da 50 o 54 byte.

Per ogni pacchetto occorre portarsi tutte le intestazioni dietro.

Focus

↓

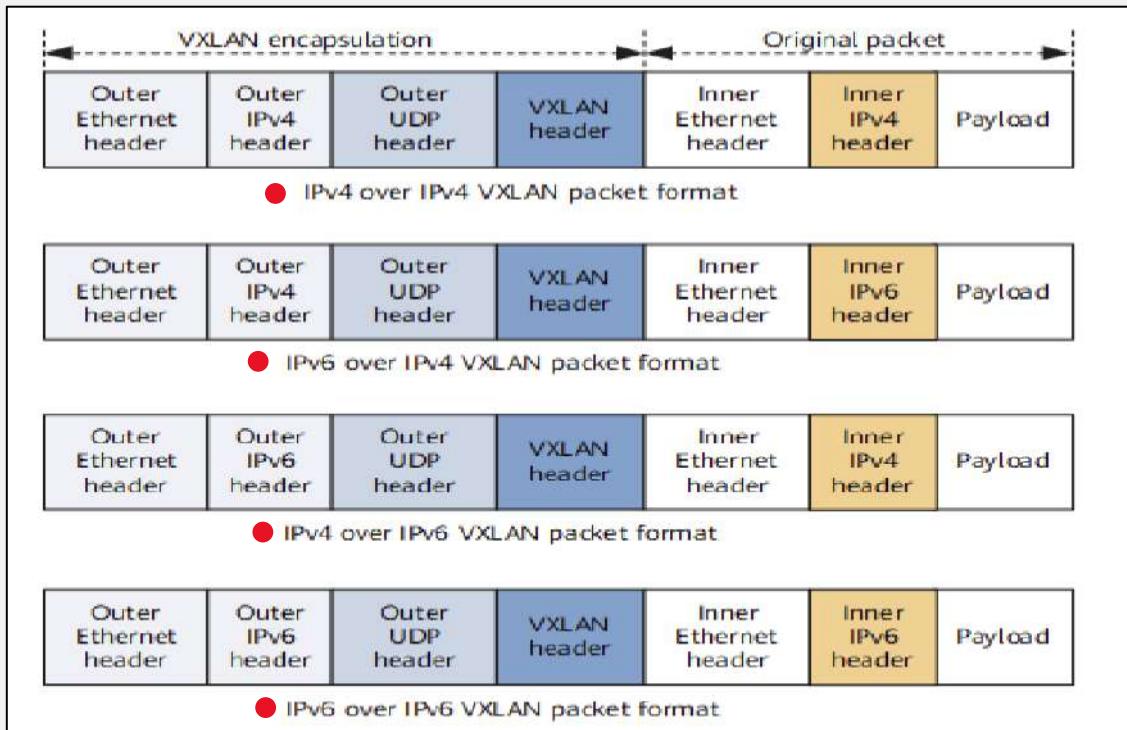
VXLAN packet format



Layer 2 Frame è la trama Ethernet originale generata all'interno della VLAN. Viene ricevuta dal VTEP che si accorge che non è destinata ad un MAC interno ma ad un altro VTEP destinatario. Quindi, mediante un applicativo, incapsula la trama inserendo l'**Header VXLAN**. Nell'intestazione ci sono 24 bit VNI che etichettano la VXLAN, 24 bit riservati e 8 bit di flag che ne condizionano il comportamento. Nel VTEP può essere configurata una o più VXLAN. Il tutto viene incapsulato all'interno di UDP e quindi viene inserito l'**header UDP**.

Poi viene tutto incapsulato dentro IP e quindi viene inserito l'**Header IP** dove ci sono vari campi, tra cui gli indirizzi IP del VTEP mittente e destinatario. Infine, il tutto è incapsulato dentro MAC e quindi viene inserito l'**Header MAC**. C'è un doppio stack protocollare, è un prezzo pesante da pagare.

La rete infrastrutturale su cui sono stabiliti i tunnel VXLAN è chiamata rete **underlay** mentre la rete di servizi trasferita sui tunnel VXLAN è chiamata rete **overlay**. Da quando esiste IPv6, il protocollo VXLAN è stato esteso anche a lui. Dunque, esistono tutte e 4 le possibili combinazioni tra IPv4 e IPv6.



L'inoltro di pacchetti in una rete VxLAN è classificato in due tipologie:

- Pacchetti broadcast, unknown unicast e multicast (**BUM**).
- Pacchetti known unicast (**quando si risponde**).

Il traffico unicast sconosciuto si verifica quando uno switch riceve traffico unicast destinato a un destinatario la cui destinazione non si trova nella sua base di informazioni di inoltro.

Gestione del traffico BUM

Esistono 3 modi per instradare i pacchetti BUM:

- **Ingress replication** (copie multiple di un pacchetto).
- **Multicast replication**.
- **Centralized replication**.

Il problema più importante nell'implementazione delle VxLAN è la determinazione delle **tabelle MAC-to-VTEP**.

Come accade con gli switch, ogni nodo foglia di una rete crea la sua **tabella**.

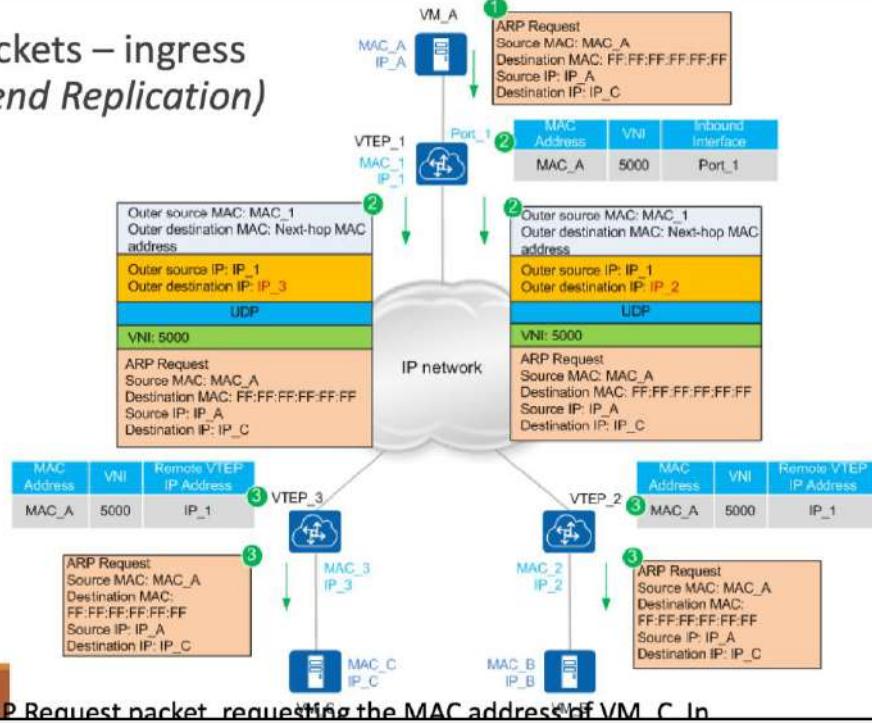
Le **tabelle contengono gli indirizzi MAC dei dispositivi collegati alle porte del nodo foglia**. Ciò che è diverso nelle VxLAN è la condivisione di queste **tabelle**.

A livello elementare, **ogni nodo foglia VTEP condivide la propria tabella, con le relative ad essa associate, con gli altri nodi foglia VTEP**.

Quando un dispositivo utente, ad esempio, invia un pacchetto in uscita al nodo foglia a cui è connesso, il nodo foglia controlla la sua tabella MAC-to-VTEP.

Ingress replication

Forwarding of BUM Packets – ingress replication (aka Head-end Replication)



VM_A broadcasts an ARP Request packet requesting the MAC address of VM_C. In

Emula la trasmissione multipla e simultanea attraverso trasmissioni unicast.

Abbiamo una VxLAN composta da 3 LAN fisiche.

VM_A invia un ARP request, il payload del pacchetto Ethernet contiene il MAC mittente (MAC_A), il MAC destinatario (tutti F, broadcast), l'IP (IP_A) mittente e l'IP destinatario (IP_C) di cui si vuole scoprire l'indirizzo MAC.

Essendo inviato in broadcast, arriva anche al VTEP_1 che popola la sua tabella con le informazioni ricevute e poi scopre che l'IP destinatario non appartiene alla sua rete. Lui conosce solo gli altri VTEP, non sa dove sia il destinatario.

Le 3 subnet in cui sono VM_A, VM_B e VM_C si riferiscono alla stessa VxLAN. Il destinatario (nell'esempio è VM_C) può trovarsi in una delle altre due subnet.

VTEP_1 inoltra la richiesta a tutti e due gli altri VTEP, 2 pacchetti separati.

Nel payload c'è la richiesta originaria a cui ci attacca un VNI (24 bit) e, come detto, **si autoconfigura** così quando riceverà la risposta sa già a chi mandarla. Incapsula il tutto in UDP e poi ci attacca l'intestazione IP con l'IP mittente che è il suo (IP_1) e con l'IP destinazione che rappresenta rispettivamente gli IP dei VTEP_2 (IP_2) e VTEP_3 (IP_3) nei rispettivi 2 pacchetti.

Infine, incapsula tutto con l'Header MAC con i vari campi di cui dispone.

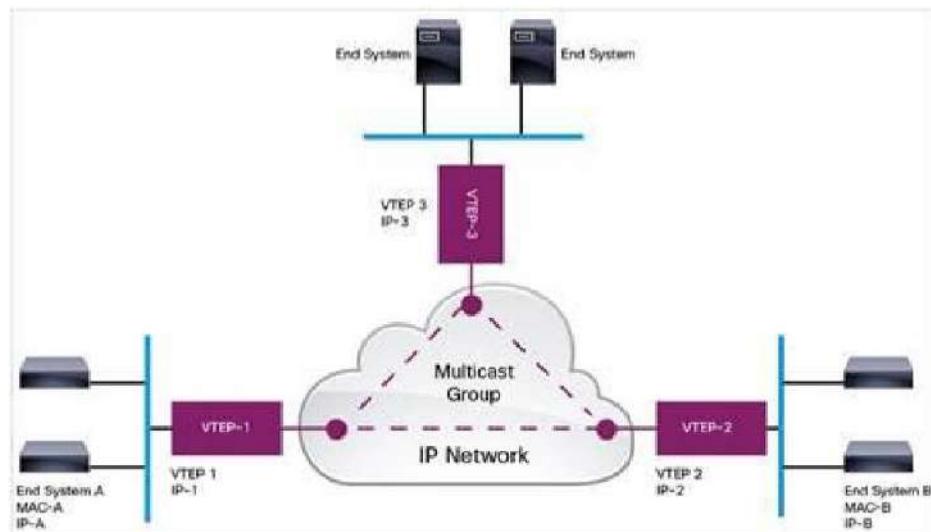
Il MAC mittente è il suo (MAC_1) mentre il destinatario è il MAC del nodo successivo che riceverà il pacchetto, il next-op che prenderà in consegna i due pacchetti mandati dal VTEP_1. Alla fine, **raggiungono i VTEP_2 e VTEP_3 che li decapsulano**. Vedono la VNI e si creano anche loro la propria tabella di

inoltro. Scoprono che MAC_A appartiene alla stessa VxLAN e che esso è raggiungibile attraverso l'IP del VTEP_1. Utilizzeranno tale IP per inviare la risposta.

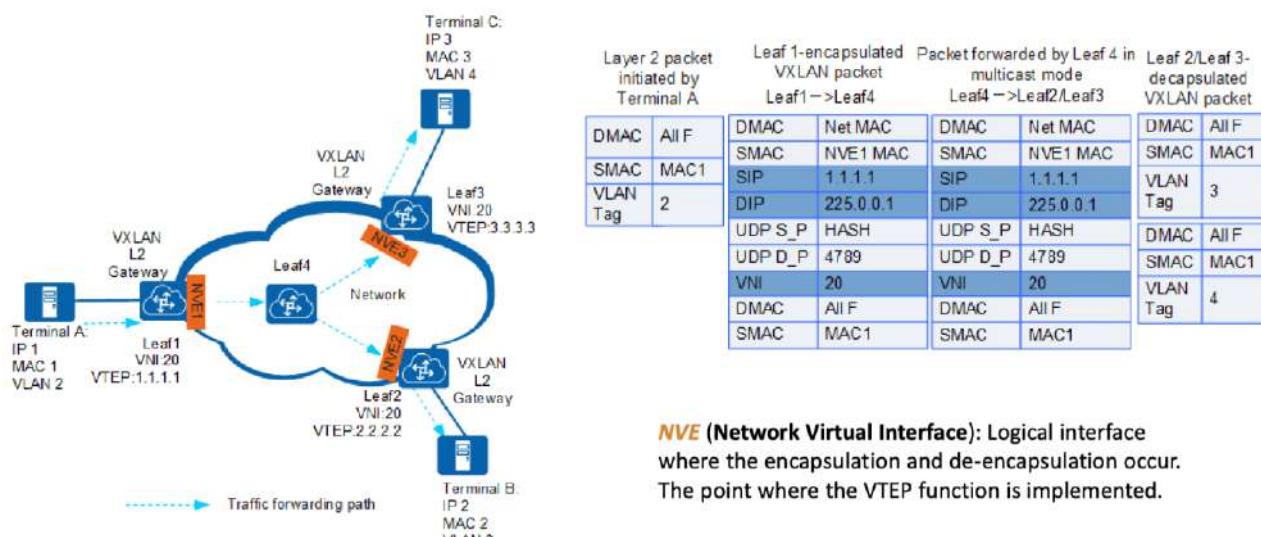
Alla fine, si scopre il destinatario in una delle due isole dove VTEP_2 e VTEP_3 inoltrano il pacchetto ARP request puro, nel corrispondente dominio di livello 2. **VM_C risponde** alla chiamata, riceve il pacchetto e invierà a sua volta un ARP replay che è un pacchetto di tipo known unicast poiché VM_C conosce l'indirizzo MAC di VM_A, in quanto risiedeva nel pacchetto ricevuto.

Multicast replication

Forwarding of BUM Packets – multicast replication



Forwarding of BUM Packets – multicast replication



Abbiamo una rete internet con 3 diverse VLAN appartenenti alla stessa VxLAN, con i rispettivi VTEP che sono implementati nei Gateway IP.

NVE è l'interfaccia dove si implementa il VTEP per incapsulare e decapsulare. È un dispositivo con varie interfacce, inclusa quella del VTEP.

Il terminale A invia un traffico BUM contenente il suo indirizzo MAC (MAC1) e come MAC destinatario utilizza tutti F.

Leaf1 incapsula il pacchetto con le varie intestazioni e utilizza come IP mittente il suo e come IP destinazione 225.0.0.1 (multicast).

Durante la comunicazione tra leaf1 e il nodo interno alla rete (leaf4) il pacchetto non viene alterato.

Dentro la rete (nuvola) non viene fatta nessun'operazione sul pacchetto.

Quando leaf4 rilancia il pacchetto a leaf2 e leaf3, il pacchetto resta uguale.

Viene poi decapsulato da leaf2 e leaf3, e inviato nei loro domini cercando il destinatario. Si fa riferimento anche ai Tag che differenziano le varie VLAN.

Si crea una rete locale distribuita composta dalle VLAN 2, 3 e 4.

Nella modalità multicast replication, tutti i VTEP con la stessa VNI si uniscono allo stesso gruppo multicast. Se un VTEP deve inviare un pacchetto ad un dispositivo che non sta nella propria VLAN, allora lo invia al gruppo multicast.

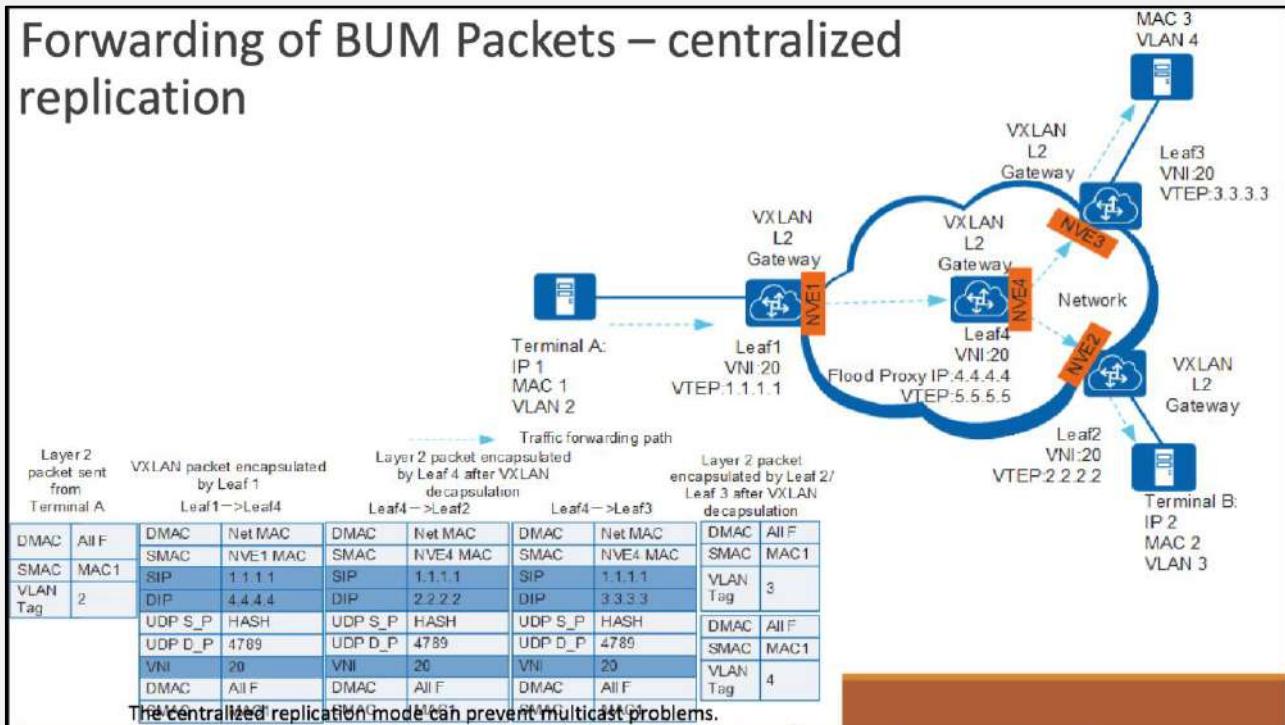
Anche in questa modalità avviene l'apprendimento del MAC con popolazione della tabella MAC-to-VTEP. Il nodo multicast dentro la rete IP inoltra il pacchetto, senza modificarlo, a tutti i leaf che sono in ascolto su quell'indirizzo.

Nel multicast replication, il leaf multicast non ha un suo tunnel.

Questo perché il pacchetto non viene modificato e l'indirizzo di destinazione resta sempre quello multicast.

Centralized replication

Forwarding of BUM Packets – centralized replication



In questa versione, si rendono tutti gli elementi all'interno della rete attivi.

Vengono inseriti dei VTEP all'interno del multicast group per agevolare l'inoltro del traffico di pacchetti. Pertanto, i pacchetti nella rete vengono modificati.

Le VxLAN terminano nella rete dove ci sarà un'istanza per arrivare ai leaf.

Dal terminale A al gateway leaf4 si utilizza come indirizzo MAC di destinazione tutti F e come indirizzo MAC mittente il MAC del terminale A (MAC1).

Da leaf 1 a leaf 4 si utilizza come indirizzo IP mittente 1.1.1.1, che è l'interfaccia VTEP di leaf1, e come indirizzo IP di destinazione 4.4.4.4 che è l'interfaccia VTEP di leaf4 dentro la rete.

Anche in questo caso supponiamo che la VxLAN sia unica (unico VNI).

Dalla rete all'esterno si modifica l'indirizzo IP di destinazione che diventa 2.2.2.2 o 3.3.3.3, mentre l'IP mittente resta uguale (quello di leaf1).

Successivamente, viene rilanciato il pacchetto nella VLAN 3 e 4.

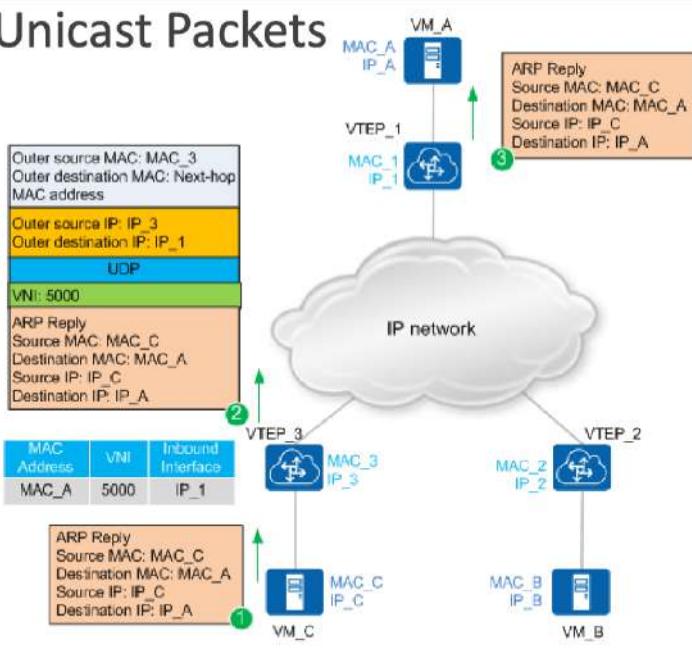
Questo è un approccio alternativo a quello precedente.

In entrambi i casi, non c'è nessun meccanismo di notifica che conferma la buona riuscita dell'operazione.

Gestione del traffico known unicast

Forwarding of Known Unicast Packets

VM_C sends an ARP Reply packet to VM_A with the source MAC address being MAC_C and the destination MAC address being MAC_A.



Rappresenta il percorso all'indietro, ovvero quando si risponde.

Si conosce già tutto. Il viaggio stavolta lo fa l'ARP reply.

Gli interlocutori pensano di scambiarsi ARP, invece c'è di mezzo il mondo.

VM_C invia un pacchetto di ARP reply a VM_A utilizzando come indirizzo MAC mittente il suo (MAC_C) e come indirizzo MAC di destinazione MAC_A.

Dopo aver ricevuto il pacchetto di ARP reply da VM_C, VTEP_3 leggendo la sua tabella determina la VNI ed esegue l'incapsulamento del pacchetto utilizzando come indirizzo IP di destinazione esterno l'indirizzo IP di VTEP_1.

Il pacchetto incapsulato viaggia nella rete finché non raggiunge VTEP_1.

Dopo che il pacchetto raggiunge VTEP_1, quest'ultimo lo decapsula per ottenere il pacchetto originale inviato da VM_C.

Infine, VTEP_1 invia il pacchetto decapsulato a VM_A.

EVPN in VxLAN

Occorre avere un piano di controllo per gestire i vari VTEP.

La tecnologia VxLAN non fornisce tale funzionalità.

EVPN aiuta ad evitare il fenomeno di inondazione del traffico nelle reti VxLAN.

Una delle problematiche nelle reti overlay è la diffusione indiscriminata del traffico BUM in tutta la rete. Questo può causare una congestione e/o causare una distribuzione inefficace del traffico.

Gli EVPN in VxLAN affrontano questo problema implementando una tecnica chiamata "Selective Forwarding" (inoltro selettivo).

Quest'approccio consente di instradare in modo selettivo i pacchetti BUM solo verso i dispositivi di rete che effettivamente richiedono tali pacchetti, invece che inondare l'intera rete.

Ciò riduce la quantità di traffico e migliora l'efficienza complessiva della rete.

EVPN sfrutta le informazioni di routing acquisite attraverso il protocollo BGP per instradare i pacchetti in modo ottimizzato verso i dispositivi della VxLAN.

BGP (Border Gateway Protocol) è un protocollo di routing utilizzato per lo scambio di informazioni tra i router di confine delle reti autonome (AS) all'interno di Internet. BGP gestisce la determinazione del percorso ottimale per instradare il traffico tra diverse reti.

In poche parole, BGP consente ai router di scambiare informazioni di routing e prendere decisioni intelligenti per instradare i pacchetti di dati attraverso la rete globale di Internet.

Il protocollo EVPN può essere visto come un'estensione del BGP, applicato per il networking di strato 2 e 3 nei datacenter.

Tutti gli endpoint dei tunnel sono configurati con EVPN permettendo loro di scambiarsi informazioni del tipo:

"Se dovete inviare un pacchetto a MAC_A, allora contattate me. Lo ho io".

NLRI (Network Layer Reachability Information) definisce le reti o i prefissi IP che sono raggiungibili attraverso un determinato router BGP.

EVPN NLRI definisce diversi tipi di rotte BGP EVPN che possono trasportare informazioni come l'indirizzo IP dell'host, l'indirizzo MAC, VNI, ecc.

Dopo che un VTEP ha appreso l'indirizzo IP e l'indirizzo MAC di un host connesso, può inviare tali informazioni agli altri VTEP tramite le rotte BGP.

In questo modo, l'apprendimento dell'indirizzo IP dell'host e delle informazioni sull'indirizzo MAC può essere implementato sul piano di controllo, eliminando il traffico in eccesso sul piano dati.

Esistono 5 tipologie di pacchetti EVPN.

Type 1	Ethernet auto-discovery (A-D) route
Type 2	<u>MAC/IP advertisement route</u>
Type 3	<u>Inclusive multicast Ethernet tag route</u>
Type 4	Ethernet segment route
Type 5	<u>IP prefix route</u>

Per VxLAN si utilizzano le tipologie 2, 3 e 5.

Il più importante è il 2 che invia pacchetti di segnalazione contenenti le combinazioni MAC e IP presenti in rete, distribuendole a tutti.

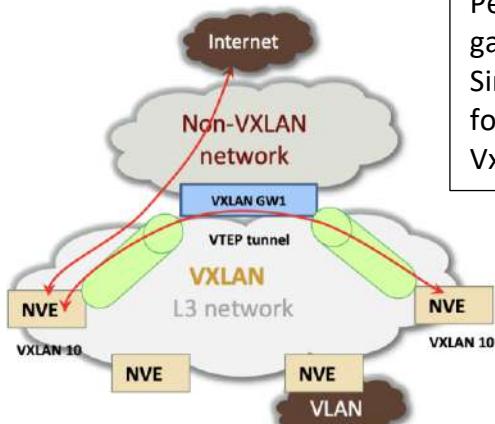
Riassumendo, EVPN in VxLAN facilita la creazione dei tunnel VxLAN, gestisce il traffico BUM in modo selettivo per ridurre la congestione di rete e utilizza le informazioni di routing BGP per instradare i pacchetti in modo ottimizzato.

Questo migliora l'efficienza complessiva della rete e consente la connessione di reti virtuali estese basate su Ethernet.

Architettura di tipo Leaf-Spine (spina dorsale)

Unico obiettivo: connettere tutti i leaf per facilitare lo scambio est-ovest.

VXLAN Gateways



Per implementare un'interoperabilità di livello 3, un gateway di livello 3 deve essere distribuito su una VxLAN. Simile a un VxLAN NVE, un gateway VxLAN di livello 3 fornisce un mappaggio tra l'intestazione del pacchetto VxLAN e l'intestazione del pacchetto IP.

Se due domini sono separati, allora devono essere connessi da un gateway.

Connessione tra reti VxLAN differenti: utilizzo di gateway VxLAN di livello 3.

Connessione tra reti VxLAN e reti non VxLAN: utilizzo sia di gateway VxLAN di livello 2 che gateway VxLAN di livello 3.

I gateway possono essere implementati in modalità centralizzata o distribuita.

Nella modalità centralizzata, i gateway di livello 3 sono configurati su un unico dispositivo centralizzato che si trova all'esterno delle reti VxLAN.

Funge da punto di ingresso e uscita per il traffico tra le reti.

L'intelligenza è centralizzata e pertanto non prevede l'utilizzo del protocollo EVPN sul piano di controllo in quanto sarebbe inutile.

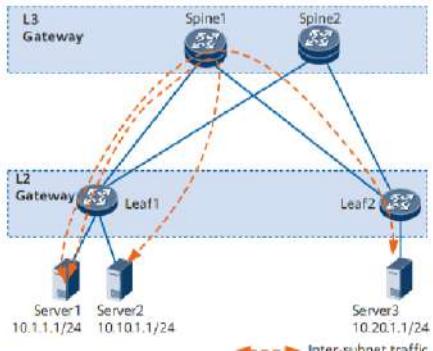
Non ci sono rotte alternative, si passa dal dispositivo centrale e basta.

Nella modalità distribuita, ciascun nodo foglia funziona sia come VTEP che come gateway di livello 3.

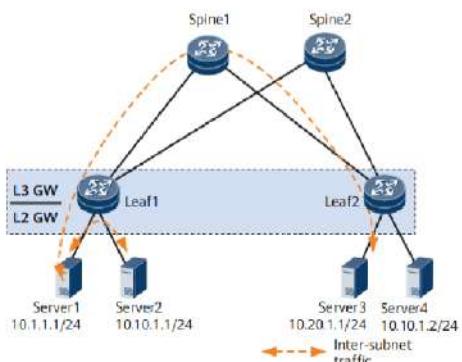
L'intelligenza è distribuita e pertanto prevede l'utilizzo di EVPN sul piano di controllo per stabilire il percorso migliore.

VXLAN Gateways

Versione centralizzata



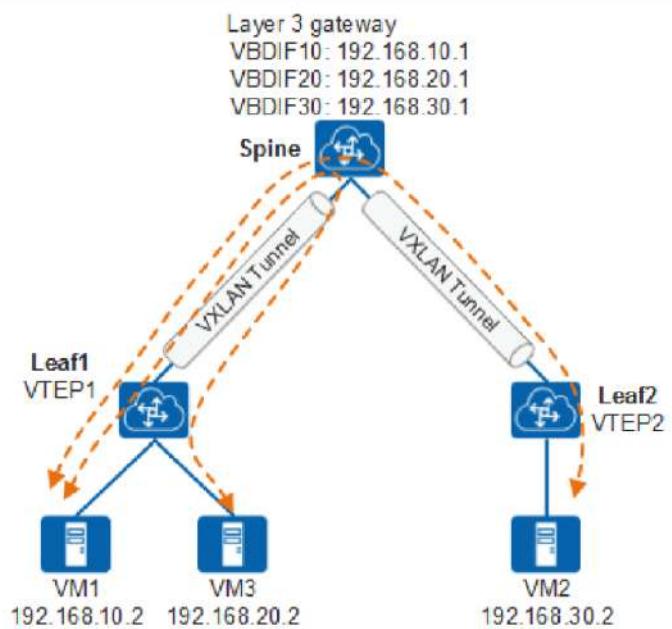
Versone distribuita



Centralized Gateway

Centralized VXLAN gateway deployment has its advantages and disadvantages.

- Advantage: Inter-segment traffic can be centrally managed, and gateway deployment and management is easy.
- Disadvantages:
 - Forwarding paths are not optimal. Inter-segment Layer 3 traffic of data centers connected to the same Layer 2 gateway must be transmitted to the centralized Layer 3 gateway for forwarding.
 - The ARP entry specification is a bottleneck. ARP entries must be generated for tenants on the Layer 3 gateway. However, only a limited number of ARP entries are allowed by the Layer 3 gateway, impeding data center network expansion.



Nella modalità centralizzata, il traffico su diverse sottoreti viene gestito centralmente. L'implementazione e la gestione dei gateway è semplificata. Tuttavia, anche il traffico tra sottoreti della stessa leaf (VTEP) deve essere inoltrato attraverso il nodo dorsale (Spine).

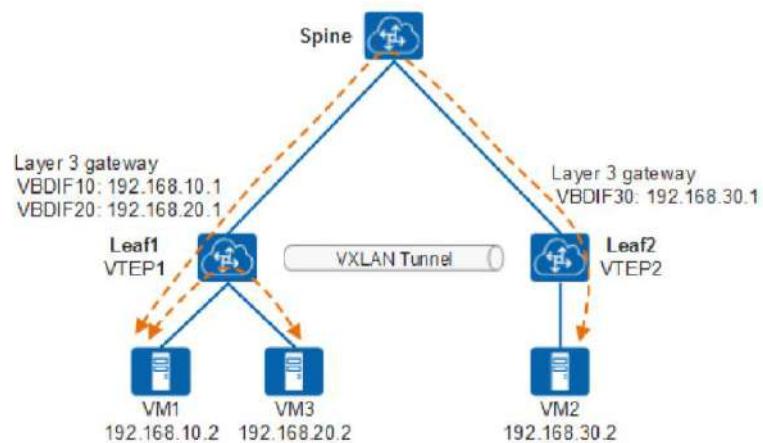
Ciò significa che il percorso di inoltro del traffico non è ottimale.

Non utilizza EVPN.

Gateway L2 nei nodi leaf. Gateway L3 nei nodi Spine.

Distributed Gateway

In the distributed gateway scenario, a control plane is required to transmit host routes between Layer 3 gateways to ensure communication between hosts. To meet this requirement, Ethernet VPN (EVPN) is introduced as the VXLAN control plane.



Nella modalità distribuita, i gateway di livello 3 sono distribuiti sui nodi leaf per implementare la comunicazione tra sottoreti sullo stesso nodo leaf.

Ciò consente al traffico di essere inoltrato direttamente dal nodo foglia senza passare attraverso il nodo spinale, conservando una grande quantità di larghezza di banda. I nodi foglia che fungono da gateway devono solo apprendere gli ARP entranti dalle VM ad esso collegate. Utilizza EVPN.

Gateway L2 e L3 entrambi sui nodi leaf.

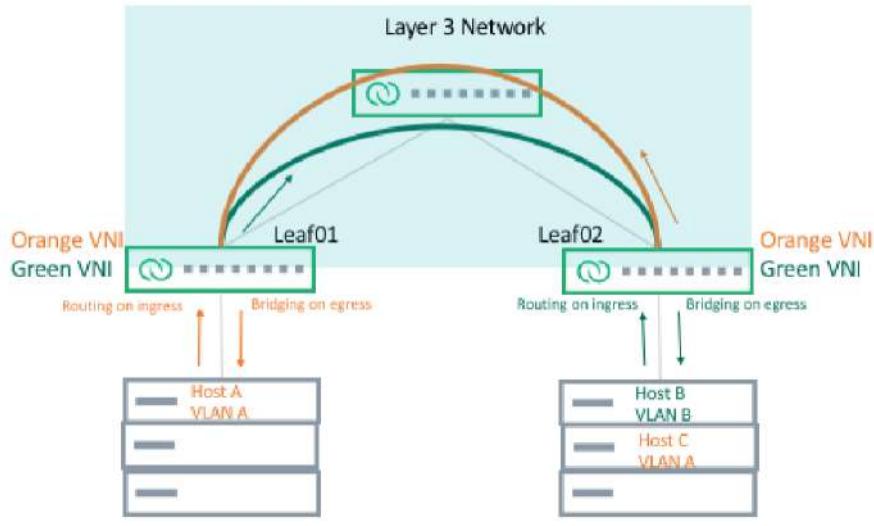
L'incapsulamento è identico, ciò che cambia è l'uso dei singoli tunnel.

Modalità di interazione tra due VxLAN

Ipotesi: ci sono 2 datacenter e 2 VxLAN attestate su di essi, una di colore arancione e una di colore verde. `vlan=vxlan` (errore di stampa).

Modello asimmetrico

VNI Types - asymmetric EVPN IRB



Quando host A invia un pacchetto a host C esso viene esaminato dal LEAF 01. LEAF 01 osserva che il destinatario ha lo stesso VNI del mittente; quindi, incapsula il pacchetto nella VxLAN verde, operando a strato 2 (bridging).

Quando host A invia un pacchetto a host B esso viene esaminato dal LEAF 01. LEAF 01 osserva che il destinatario non ha lo stesso VNI del mittente; quindi, incapsula il pacchetto nella VxLAN arancione, operando a strato 3 (routing). Una volta arrivato al LEAF 02, in base al valore del VNI (colore) verrà deciso il percorso da utilizzare verso i destinatari.

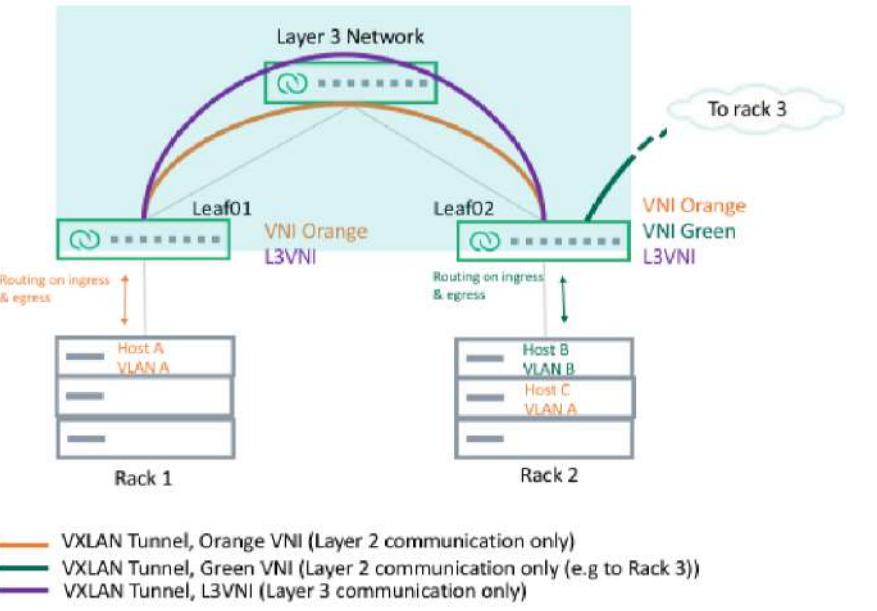
Il modello asimmetrico consente il routing e il bridging in ingresso al tunnel VxLAN, ma solo il bridging in uscita.

LEAF 02 rimuove l'intestazione VxLAN dal frame e poi lo invia all'Host C. Con il modello asimmetrico, tutti i valori di VNI di origine e destinazione richiesti (in questo caso VNI arancione e VNI verde) devono essere presenti su ogni nodo foglia, anche se un nodo foglia non ha un host.

L'instradamento tra i domini VxLAN è gestito solo da uno dei due gateway VxLAN. In questo caso, uno dei gateway, noto come ingress gateway, si occupa dell'instradamento in entrata verso il proprio dominio VxLAN e inoltro dei pacchetti provenienti da altri domini VxLAN. L'altro gateway, noto come egress gateway, si occupa solo dell'instradamento in uscita verso altri domini VxLAN.

Modello simmetrico

VNI Types - symmetric EVPN IRB



Il modello simmetrico consente il routing e il bridging sia in ingresso al tunnel VxLAN che in uscita.

Quando si riceve da un VNI diverso, il LEAF esegue operazioni di routing.

Quando si riceve da un VNI uguale, il LEAF esegue operazioni di bridging.

Con il modello simmetrico, i LEAF (switch) devono solo ospitare le VLAN e le VNI corrispondenti che si trovano sul proprio rack, nonché la L3VNI e la VLAN associata, poiché il LEAF di ingresso non ha bisogno di conoscere la VNI di destinazione. Tuttavia, la configurazione è più complessa in quanto sono necessari un tunnel VxLAN aggiuntivo e una VLAN nella rete.

Nel modello EVPN simmetrico, entrambi i gateway VxLAN sono coinvolti nell'instradamento del traffico bidirezionale tra i domini VxLAN.

Entrambi i gateway svolgono sia le funzioni di ingress gateway che di egress gateway per i pacchetti che attraversano i domini VxLAN.

Capitolo 9: SDN (Software Defined Networking)

SDN (Software Defined Networking) è un approccio all'architettura delle reti informatiche che **separa il controllo di rete dallo strato di inoltro dei dati**. Invece di avere dispositivi di rete tradizionali con intelligenza distribuita, SDN centralizza il controllo e permette di gestire la rete in modo più flessibile e programmabile.

In un'architettura SDN, lo strato di controllo è separato dai dispositivi di rete, come switch e router. Questo strato di controllo è gestito da un **controller** SDN centralizzato, **che prende decisioni sulle regole di inoltro del traffico e le distribuisce ai dispositivi di rete**.

I dispositivi di rete, noti come switch SDN, **diventano semplici inoltratori di pacchetti, eseguendo le istruzioni del controller**.

Il controller SDN rappresenta il "cervello" del sistema, fornendo una vista completa e centralizzata della rete.

Utilizzando un'interfaccia di programmazione applicativa (API) standardizzata, il controller può essere gestito e programmato da applicazioni di livello superiore o da amministratori di rete. Ciò consente di creare politiche di rete dinamiche e flessibili, adattabili alle esigenze specifiche dell'ambiente di rete.

Le reti SDN offrono diversi vantaggi.

Permettono una maggiore flessibilità nella gestione e configurazione della rete, semplificano le operazioni di rete e riducono la dipendenza dall'hardware specifico dei dispositivi di rete.

Inoltre, SDN facilita l'implementazione di nuove funzionalità di rete e di servizi, consentendo una rapida innovazione nella gestione delle reti.

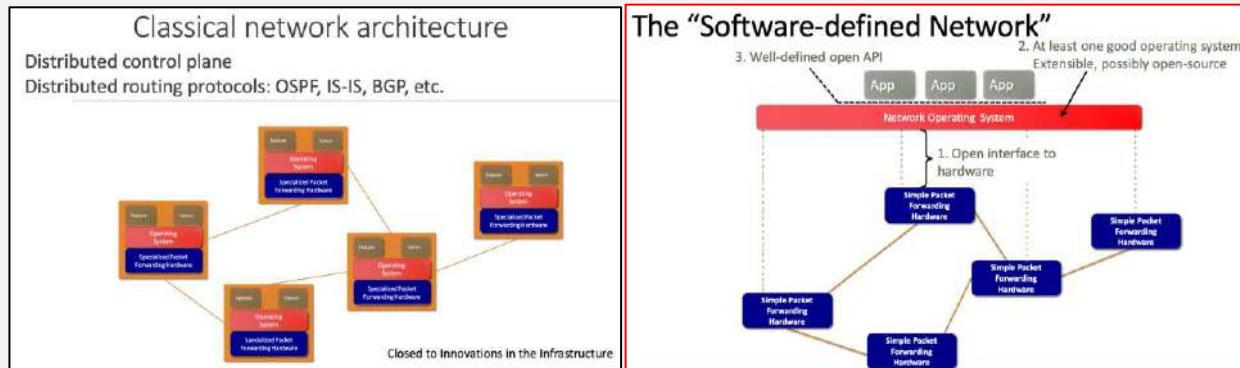
Nel complesso, **SDN offre un approccio più programmabile e flessibile alla gestione delle reti, consentendo di adattare rapidamente le reti alle esigenze in continua evoluzione delle applicazioni e dei servizi**.

Con SDN c'è una gestione della rete basata su software, forma di automatismo.

I dispositivi tradizionali sono dotati di tutti le tipologie di piani che servono:

- Piano utente: per ricevere e spedire i pacchetti.
- Piano di controllo: per configurare il dispositivo attraverso dei protocolli.
- Piano di management: per la gestione degli errori.

Sono in ogni benedetto dispositivo. Occorre evitare questa cosa.



Si va a centralizzare il cervello, i dispositivi diventano più leggeri ma stupidi.

Spostiamo anche le applicazioni, dentro resta solo il piano di Forwarding.

L'SDN prevede una separazione completa tra il piano di controllo e il piano d'utente attraverso la centralizzazione, dal punto di vista logico, del sistema operativo che diventa un sistema operativo di rete che gestisce l'intera rete e le applicazioni che non sono più implementate nei dispositivi.

I dispositivi di rete (switch, router, ecc.) prendono il nome di **dispositivi dumb**, ovvero sono senza cervello. La chiave è avere un'interfaccia di controllo standardizzata che parli direttamente con l'hardware.

Il controller va nel Network Operating System (NOS).

Modello molto diffuso nei Datacenter.

Si implementa un'interfaccia aperta interrogabile attraverso API di dominio pubblico cosicché tutti possano collegarsi sia dal basso che dall'alto, ovvero chi produce hardware in basso produce hardware aperto e in alto si consente a terze parti di sviluppare applicazioni come vogliono.

Esplode il networking. Ognuno programma e gestisce come vuole.

Non si è più legati alla logica chiusa dei singoli vendori.

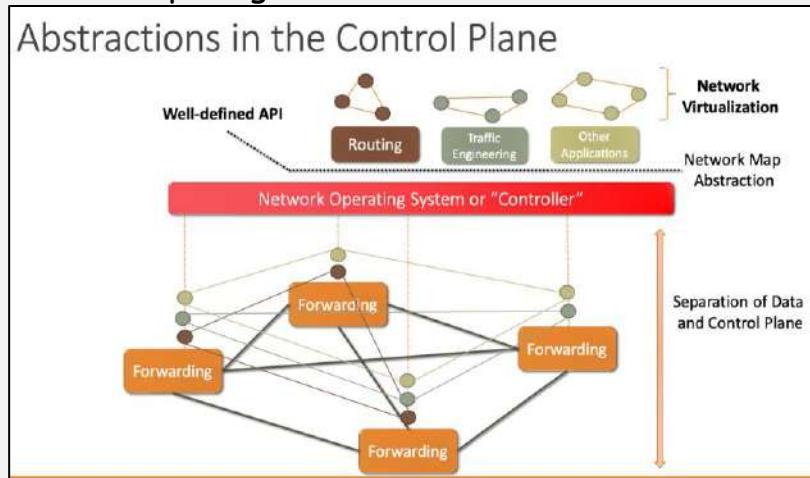
Northbound: direzione dai dispositivi dumb all'NOS.

Southbound: direzione dal controller NOS ai dispositivi dumb.

Per le interfacce viene utilizzato HTTP.



Ci sono diversi vendori per ogni livello, in un livello le scelte sono varie.



Slice: porzione di rete riservata appositamente per un utente.

Quindi, con SDN abbiamo:

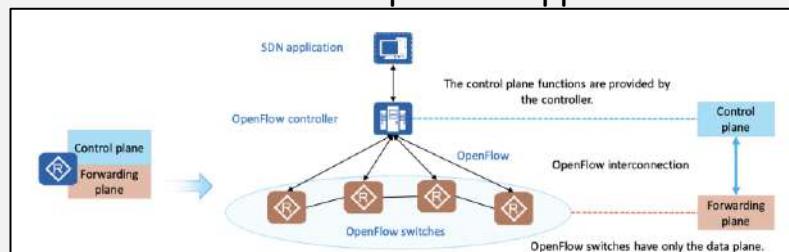
- Entità separate per piano di controllo e per piano utente.
- Intelligenza e stato della rete logicamente centralizzati.
- Infrastruttura di rete sottostante scollegata dalle applicazioni.
- Software del piano di controllo eseguibile su hardware generico.
- Disaccoppiamento da hardware di rete specifico.
- Piano utente programmabile e controllabile da un'entità centrale.
- Intera architettura di rete da controllare.

Per implementare SDN occorrono due elementi fondamentali:

- **Network Operating System** (NOS), detto anche **Controller**: sistema distribuito che crea una vista coerente e aggiornata della rete. Attraverso dei protocolli aperti riceve informazioni dagli elementi di forwarding e risponde loro inviando le regole da applicare.
- **OpenFlow**: protocollo per lo scambio di tipo southbound **per il controllo remoto della tabella di inoltro** di uno switch o di un router.

Nella fase iniziale SDN ha come obiettivo la separazione tra il piano di controllo e il piano d'inoltro, stabilire un controllo centralizzato della rete e realizzare interfacce programmabili aperte.

OpenFlow è il protocollo più popolare che agisce tra il controller e i dispositivi che spesso sono switch. **Data Path:** dispositivo appartenente al Piano d'inoltro.



Un esempio di come SDN rompe la logica chiusa del singolo venditore può essere fatto a riguardo della scelta degli switch di rete.

Nei modelli di rete tradizionali, l'acquirente è spesso vincolato a utilizzare gli switch di un singolo venditore specifico, poiché le funzionalità e il software di gestione sono fortemente integrati e dipendenti da quel particolare venditore. Con SDN, la situazione cambia. Poiché il piano di controllo è separato dagli switch di rete ed è gestito dal controller SDN, è possibile utilizzare switch di diversi vendori senza compromettere la coerenza e la gestione della rete.

Il controller SDN è in grado di comunicare con switch di diversi fornitori tramite interfacce standardizzate come OpenFlow.

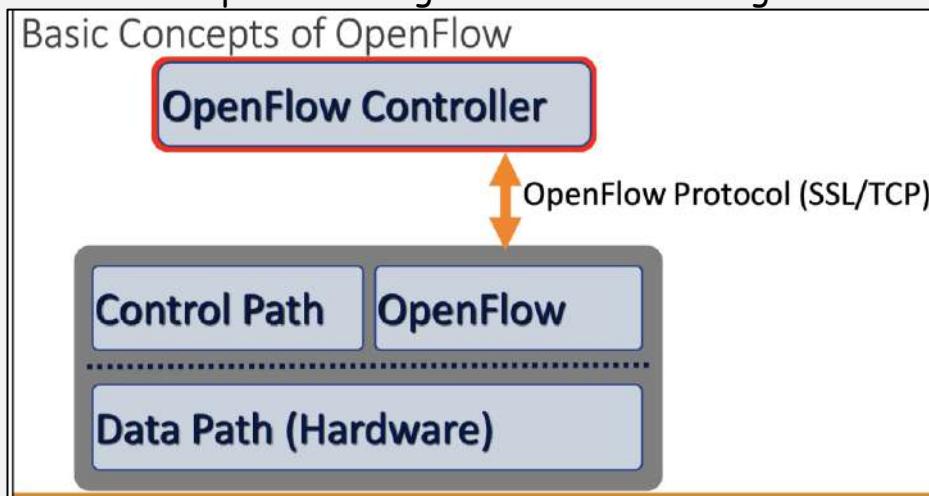
Esempio di controller SDN: **Ryu**.

Come detto, OpenFlow agisce sul traffico southbound.

OpenFlow si posiziona nello strato applicativo, sopra TCP.

Utilizza SSL per proteggere le comunicazioni più importanti.

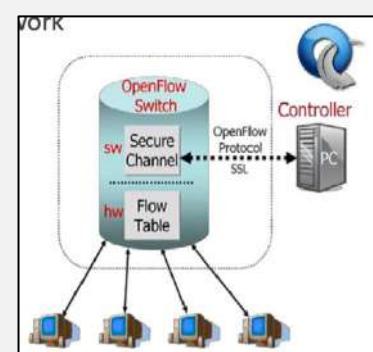
Nei dispositivi dumb ci dev'essere un agente compatibile per comunicare con il controller. Per sicurezza, una porzione dei dispositivi è riservata a Control Path cosicché in caso di problemi magari riescono ad autogestirsi da soli.



Obiettivo: definire le **Flow Table** all'interno degli switch.

Sono delle tabelle contenenti dei dati che identificano i **flussi** di traffico e impostano delle **regole** per la loro gestione.

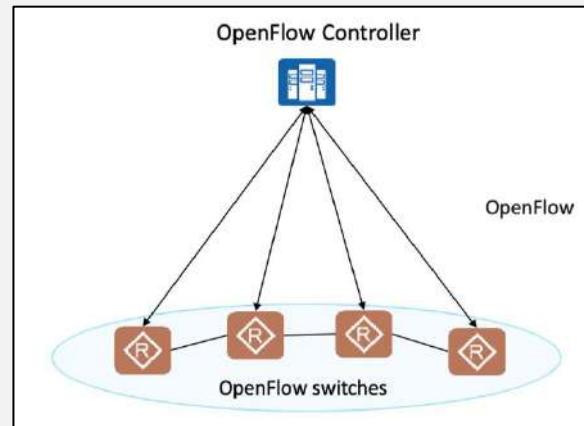
Nel Controller si gestisce il programma e le regole per i flussi che vengono poi iniettate nelle Flow Table dei dispositivi dumb. Uno switch OpenFlow è costituito da una Flow Table con le annessa azioni (regole) associate a ciascuna entità di flusso.



I messaggi tra il Controller OpenFlow e i Data Path possono essere di tre tipologie:

- **Da Controller a switch:** messaggi unidirezionali contenenti delle configurazioni.
- **Asincroni:** messaggi avviati da entrambi i lati.
- **Simmetrici:** messaggi di presentazione, come hello-x!

Nei messaggi è presente un **campo Type** che definisce mittente e destinatario: C->S o S->C.



Three-Way Handshake

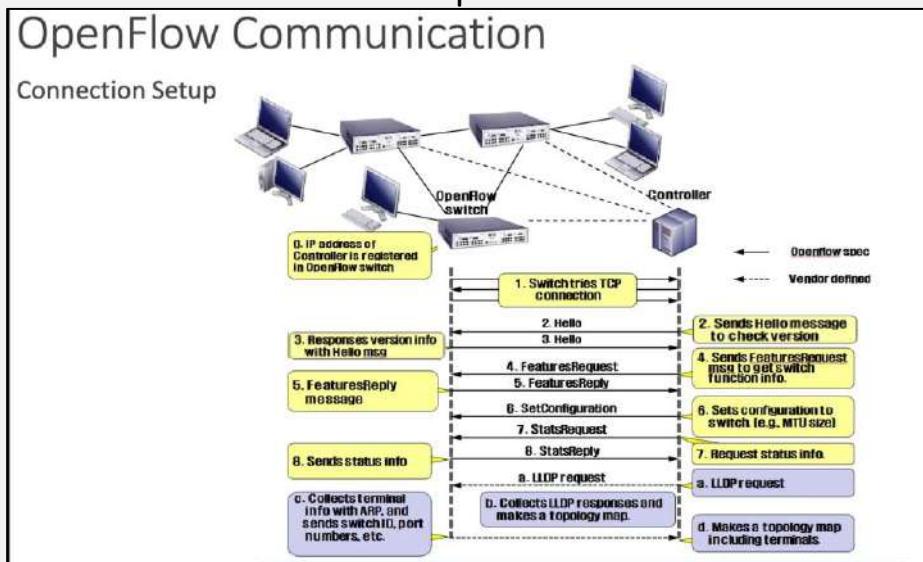
Networking che viene dopo la fase di configurazione.

Entità: dispositivi dumb, host e controller.

Gli switch si devono presentare al controller, richiedendo una connessione TCP.

Il controller, dal canto suo, si crea una mappa della rete.

È possibile decidere se inondare i dumb con le regole a nastro oppure aspettare che uno alla volta si presentino. Nel secondo caso il controller dev'essere più intelligente. Con i messaggi simmetrici (hello), Controller e Switch si accordano sulla versione di OpenFlow da utilizzare.



Gli switch inoltrano i pacchetti sulla base della Flow Table che hanno in pancia, applicando in base al traffico ricevuto delle regole, come scartare pacchetti, inoltrarli di nuovo, etc.

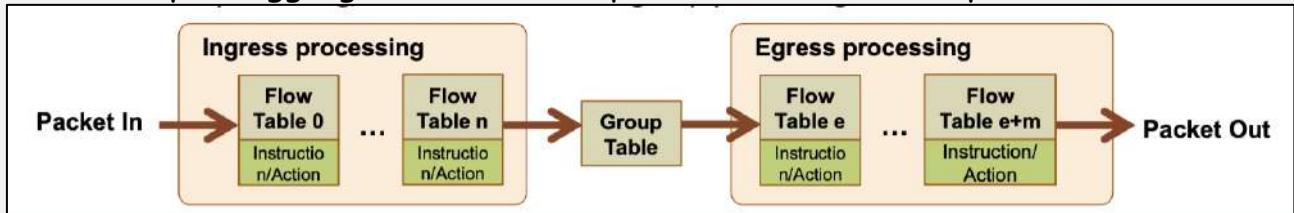
Match Fields: i vari campi rispetto ai quali viene confrontato un pacchetto, se i valori corrispondono allora si avvia l'azione correlata a quel flusso.

OpenFlow: Flow Table

Operation Mode	Switch Port	MAC src	MAC dst	Ether type	VLAN ID	Src IP	Dst IP	Proto No.	TCP S_port	TCP D_port	Action	Counter
Switching	*	*	00:1f..	*	*	*	*	*	*	*	Port1	243
Flow Switching	Port3	00:20..	00:2f..	0800	vlan1	1.2.3.4	1.2.3.9	4	4566	80	Port7	123
Routing	*	*	*	*	*	*	1.2.3.4	*	*	*	Port6	452
VLAN Switching	*	*	00:3f..	*	vlan2	*	*	*	*	*	Port6 Port7 Port8	2341
Firewall	*	*	*	*	*	*	*	*	*	22	Drop	544
Default Route	*	*	*	*	*	*	*	*	*	*	Port1	1364

Wild card (*) means "does not matter" – not important field

Gli switch SDN basati su OpenFlow possono utilizzare anche le Group Table, utilizzate per l'aggregazione e la manipolazione dei flussi di pacchetti.



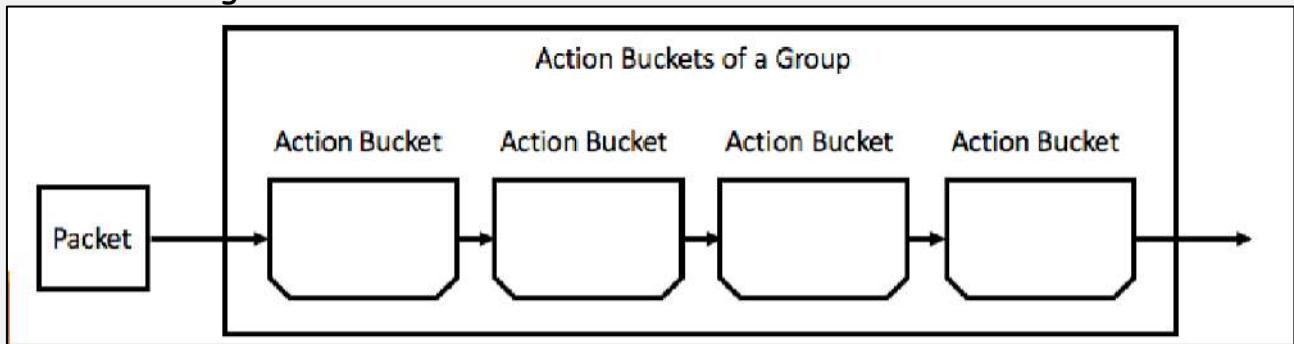
Le Group Table sono simili alle Flow Table, ma invece di definire singole azioni da intraprendere sui pacchetti, raggruppano una serie di azioni in un gruppo. Questo può includere l'inoltro del pacchetto a più porte di uscita contemporaneamente, la clonazione del pacchetto su più interfacce o l'invio del pacchetto a un gruppo di host specifico.

Una Group Table è composta da un insieme di **Bucket**.

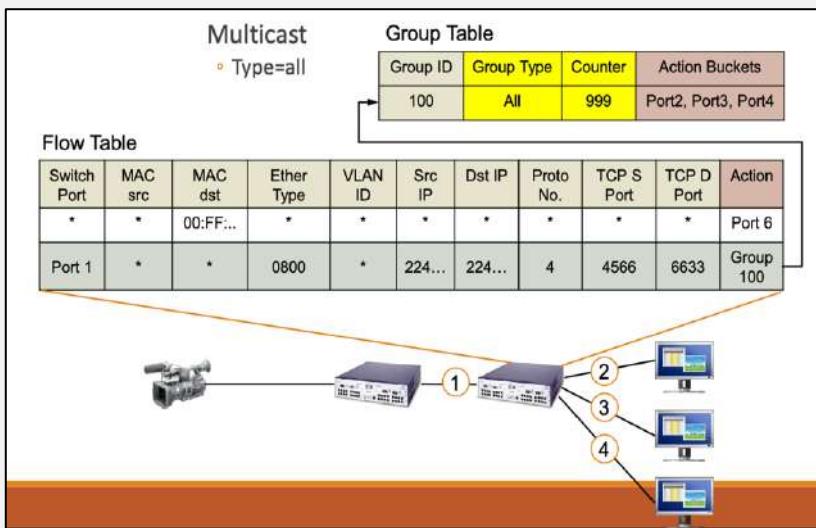
Ogni Bucket contiene un insieme di azioni che vengono applicate ai pacchetti aventi caratteristiche comuni.

Invece di definire regole separate per ciascun flusso, è possibile creare una Group Table che gestisce un insieme di flussi con un'unica azione.

Ciò semplifica la gestione e riduce la complessità delle regole di instradamento all'interno degli switch.

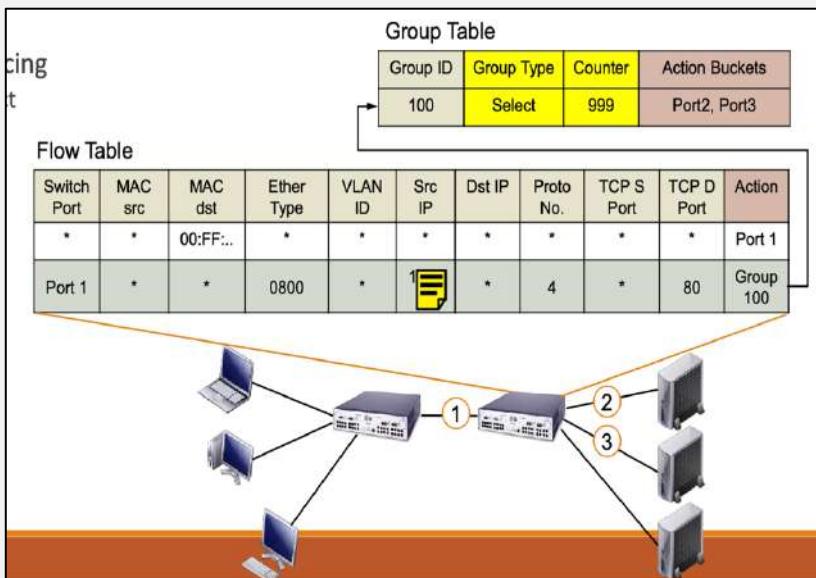


Sono 4 le tipologie di Group Table: All, Select, Indirect e Fast Failover.



Group Table di tipo "all".
Esegue tutti i Bucket del gruppo.

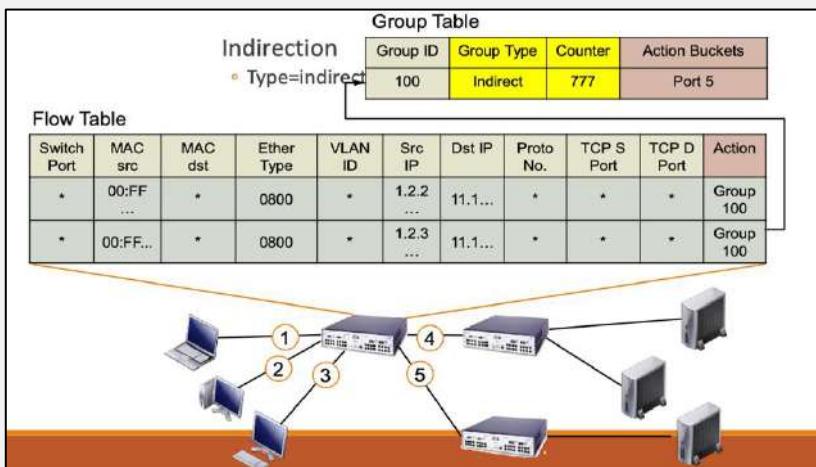
Esempio: invio di un video a tre dispositivi differenti.
Se c'è traffico che arriva nella porta 1 e gli altri campi sono uguali a quelli specificati nella Flow Table allora si inoltra il traffico verso le porte 2, 3 e 4.



Group Table di tipo "select".

Esegue solo alcuni Bucket del gruppo.

Ricevere flusso sulla porta 1 fa inoltrare il traffico alle porte 2 e 3.
Si effettua load balancing sui due server.



Group Table di tipo "Indirection". Indica un oggetto o un valore mediante un suo riferimento invece che direttamente.

OpenFlow Group Table

Fast Failover

- Type=fast-failover (ff)

Flow Table

Switch Port	MAC src	MAC dst	Ether Type	VLAN ID	Src IP	Dst IP	Proto No.	TCP S Port	TCP D Port	Action
Port 1	*	*	*	*	1.2.2	*	*	*	*	Port 7
Port 1	00:FF ...	*	0800	*	1.2.3 ...	11.1...	*	*	*	Group 100



Group Table di tipo

"fast failover".

Esegue il primo Bucket vivo.

Utilizzate per definire

azioni che stabiliscono percorsi secondari se il principale si interrompe.

In questo caso la porta principale è la 4. Se non funziona si inoltra su 5 e 6.

Inoltro dei pacchetti in OpenFlow

Inserimento flusso reattivo:

Quando un pacchetto non abbinato raggiunge lo switch OpenFlow, viene inviato al controller e in base alle informazioni nell'intestazione del pacchetto, un flusso appropriato verrà inserito. È sempre necessario interrogare il controller durante l'arrivo dei pacchetti → lento.

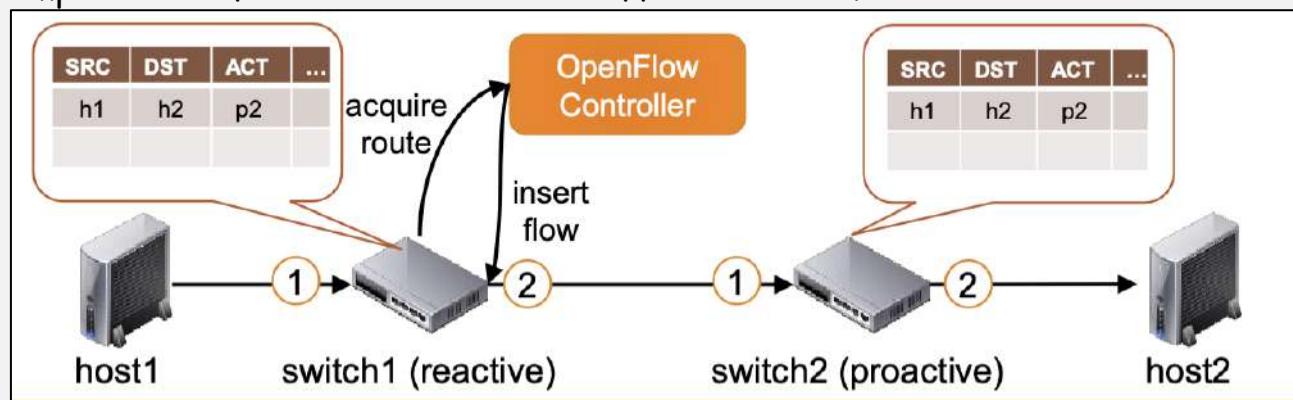
Può riflettere lo stato del traffico corrente

Inserimento flusso proattivo:

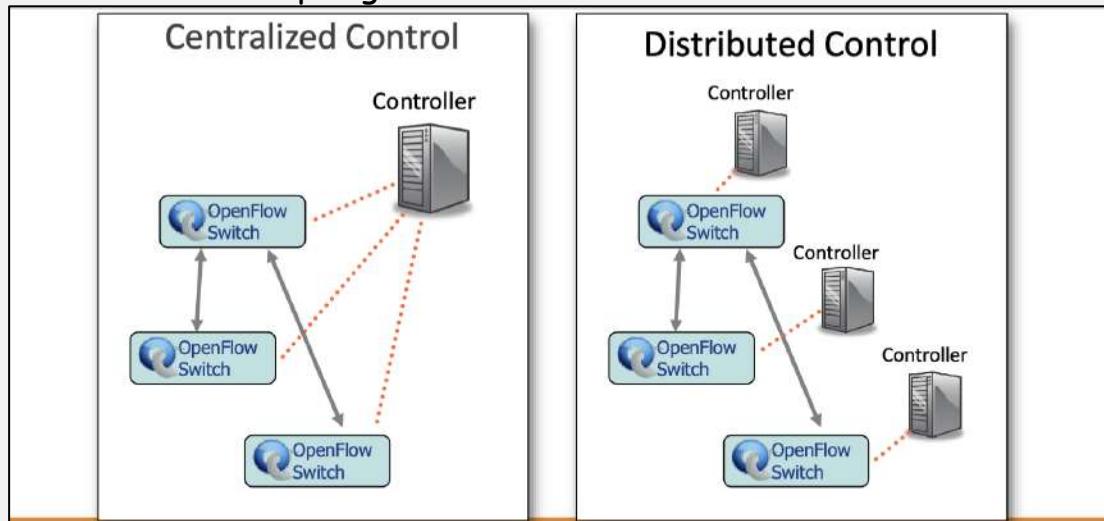
Il flusso può essere inserito in modo proattivo dal controller negli switch prima dell'effettivo arrivo del pacchetto.

Nessuna necessità di comunicare durante l'arrivo dei pacchetti → veloce.

Impossibile riflettere lo stato del traffico corrente.



Con l'utilizzo di OpenFlow il Controller SDN può essere sia centralizzato che distribuito. Nella modalità **centralizzata**, c'è un singolo controller SDN che ha il controllo completo sulla rete. Nella modalità **distribuita**, invece, il controllo della rete è distribuito su più controller SDN che collaborano tra loro per restare sincronizzati e per gestire l'intera rete.



Network Function Virtualization

La Network Function Virtualization (NFV) è un paradigma di virtualizzazione delle funzioni di rete che mira a **separare le funzioni di rete tradizionalmente implementate su hardware specializzato** (router, firewall, load balancer) dalla piattaforma hardware sottostante, trasformandole in software virtualizzato.

L'obiettivo principale della NFV è quello di migliorare l'agilità, la flessibilità e l'efficienza delle reti, consentendo la creazione di servizi di rete su piattaforme virtualizzate.

Le funzioni di rete virtualizzate, chiamate Virtual Network Functions (VNF), possono essere eseguite su server standard come macchine virtuali o container, anziché richiedere hardware dedicato.

SDN e NFV sono concetti complementari che spesso vengono utilizzati insieme per fornire soluzioni di rete più agili e flessibili.

SDN si concentra sulla separazione del piano di controllo dal piano di inoltro, consentendo la gestione centralizzata e programmabile della rete.

NFV si concentra sulla virtualizzazione delle funzioni di rete.

Esempio: un'organizzazione desidera implementare un firewall per proteggere il traffico di rete all'interno della sua infrastruttura.

Utilizzando NFV, invece di acquistare e configurare un nuovo dispositivo hardware dedicato, l'organizzazione può utilizzare un server esistente e creare una VNF di firewall.

Il server viene configurato per eseguire una macchina virtuale o un container che contiene il software del firewall. Il traffico di rete viene quindi instradato verso la VNF del firewall per essere analizzato e filtrato. La VNF del firewall può essere gestita centralmente attraverso un'interfaccia di gestione SDN, che consente di definire le regole di filtraggio e di aggiornarle in modo dinamico.