



Riassunto di
Virtual Networks and Cloud Computing

CORSO DI LAUREA IN INGEGNERIA INFORMATICA E ROBOTICA – A.A. 2023-2024

DIPARTIMENTO DI INGEGNERIA

docente

Prof. Gianluca REALI

Virtual Networks and Cloud Computing

studenti

Alex	Ardelean	alexnicolae.ardelean@studenti.unipg.it
Giuseppe	Gallo	giuseppe.gallo2@studenti.unipg.it

Indice

1 Virtualizzazione	3
1.1 Introduzione	3
1.2 Docker	3
1.2.1 Architettura ad alto livello	4
1.2.2 Architettura a basso livello	4
1.2.3 Creazione di container	5
1.2.4 Dockerfile: creazione di immagini	5
1.2.5 Applicazioni multi container	6
1.2.6 Volumi	6
1.2.7 Rete	7
1.3 Kubernetes	10
1.3.1 Overview	10
1.3.2 Oggetti Kubernetes	11
1.3.3 Namespace kubernetes	12
1.3.4 Architettura master-worker	13
1.3.5 Pod	15
1.3.6 Workload Resources	16
1.3.7 Networking	18
1.3.8 Volumi	24
1.3.9 HPA	25
1.4 VxLAN	27
1.4.1 VLAN	27
1.4.2 Datacenter	27
1.4.3 Multitenancy	28
1.4.4 Problemi nei datacenter	29
1.4.5 Soluzione VxLAN	29
1.4.6 Header VxLAN	30
1.4.7 Meccanismi di forwarding	30
1.4.8 VNI inter-networking	33
2 Cloud Computing	35
2.1 Modello di servizio	35
2.2 Modello di deployment	36
2.3 Sicurezza negli ambienti cloud	37
2.4 Shared responsibility model	37
2.5 Politica Zero Trust	38
2.6 Design Pattern	38
2.6.1 DP per la gestione dei dati	38
2.6.2 DP implementativi	42
2.6.3 DP per l'affidabilità	43
2.6.4 DP per la disponibilità	44
2.6.5 DP per la messaggistica	45
2.7 OpenStack	46
2.7.1 Architettura	46
2.7.2 Networking	48
2.7.3 Zone gerarchiche	49
2.7.4 Servizi	49
2.8 Serverless e OpenFaaS	59

2.8.1	Serverless	59
2.8.2	OpenFaaS	59
3	Teoria delle code	63
3.1	Sistema a coda	63
3.1.1	Caratterizzazione matematica	63
3.1.2	Grandezze limite	64
3.1.3	Notazione di Kendall	64
3.1.4	Legge di Little	65
3.1.5	Fattore di utilizzazione	66
3.1.6	Stato del sistema	66
3.2	Catene di Markov	66
3.2.1	Equazioni di Chapman-Kolmogorov	66
3.2.2	Probabilità di uno stato	69
3.2.3	Catene di Markov omogenee	70
3.2.4	Catene di Markov irriducibili	70
3.2.5	Catene di Markov ergodiche	70
3.2.6	Soluzione alle equazioni di C-K per catene omogenee	71
3.2.7	Catene di Markov omogenee in equilibrio	71
3.3	Processi di nascita e morte	71
3.3.1	Definizione	71
3.3.2	Probabilità di stato	72
3.4	Processi di Poisson	72
3.4.1	Probabilità di stato	72
3.4.2	Caratterizzazione statistica degli stati	73
3.4.3	Caratterizzazione statistica del tempo di interarrivo	74
3.4.4	Probabilità di k eventi	74
3.4.5	Distribuzione di Erlang	75
3.5	Sistemi di servizio	75
3.5.1	Parametri prestazionali	76
3.5.2	Processi di nascita e morte in equilibrio statistico	76
3.6	Sistema a coda $M/M/1/\infty/\infty$	77
3.6.1	Probabilità limite di stato	78
3.6.2	Numero medio di utenti nel sistema	78
3.6.3	Tempo di permanenza nel sistema	78
3.6.4	Parametri prestazionali	78
3.6.5	Distribuzione in equilibrio statistico	78
3.6.6	Tempo di attesa in coda	79
3.6.7	Tempi di permanenza nel sistema	79
3.7	Sistema a coda $M/M/\infty$	80
3.7.1	Probabilità limite di stato	80
3.8	Sistema a coda $M/M/m$	81
3.8.1	Probabilità limite di stato	81
3.8.2	Formula C di Erlang	81
3.9	Sistema a coda $M/M/m/\infty$	82
3.9.1	Probabilità limite di stato	82
3.9.2	Probabilità di blocco di servizio	82
3.9.3	Parametri prestazionali	83
3.9.4	Dimensionamento del sistema	83
3.10	Sistema a coda $M/M/1/K$	83
3.10.1	Probabilità limite	83
3.11	Proprietà PASTA	84
3.11.1	Proprietà dei processi di Poisson	84
3.12	Teorema di Burke	85
3.13	Teorema di Jackson	85
3.14	Ipotesi di indipendenza di Kleinrock	86

Capitolo 1

Virtualizzazione

1.1 Introduzione

La **virtualizzazione** è una tecnica utilizzata per astrarre le risorse hardware per renderle disponibili al software come risorse virtuali (ad esempio CPU, RAM, disco, ...) che possono essere in generale diverse dalle risorse effettive della macchina. Una **macchina virtuale** è l'insieme delle risorse virtuali.

Esistono due tipi di virtualizzazione:

- **tradizionale**: ogni VM esegue un OS isolato. Fa uso di un **hypervisor**, ovvero un programma capace di creare ed eseguire VM e virtualizzare l'hardware. Esistono due tipologie di hypervisor:
 - tipo 1 (**bare-metal**): eseguiti direttamente sull'hardware;
 - tipo 2 (**hosted**): eseguiti all'interno di un OS (ad esempio VirtualBox);

L'hypervisor è composto da tre moduli: dispatcher, allocator, interpreter.

- **a livello di OS**: fa uso di container per virtualizzare l'OS. I container condividono lo stesso kernel dell'OS host, ma eseguono applicazioni in spazi isolati chiamati **namespace**. Ogni container include solo le librerie e le dipendenze necessarie per eseguire l'applicazione.

La **tecnica tradizionale** offre un livello elevato di **isolamento** a scapito di un **overhead** alto e tempi di avvio lunghi. La **virtualizzazione a livello di OS** consente di ottenere un **overhead minore** a scapito di un **isolamento minore**.

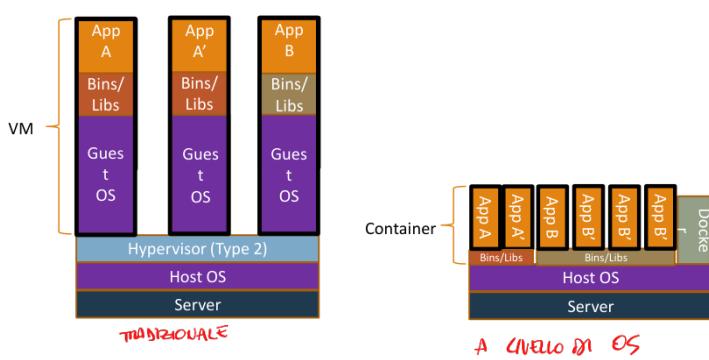


Figura 1.1: VM vs Container

1.2 Docker

Docker è una piattaforma open-source progettata per automatizzare il deployment, la scalabilità e la gestione di applicazioni. Utilizza la virtualizzazione a livello di sistema operativo per eseguire le applicazioni in ambienti isolati chiamati **container**. Un container Docker, a differenza di una macchina virtuale, non include un sistema operativo separato. Al contrario, utilizza alcune funzionalità del kernel Linux, come ad esempio i **namespace** per creare ambienti separati per l'elaborazione dei processi, oppure **cgroups** che permette di gestire l'allocazione di

risorse fisiche ai processi. Un container è quindi una sorta di contenitore che può eseguire una o più applicazioni fornendo loro lo “stretto necessario” per essere eseguite in maniera corretta.

1.2.1 Architettura ad alto livello

Docker utilizza un’**architettura client-server** in cui il **client Docker** comunica con il **Daemon Docker** (`dockerd`) che esegue il lavoro pesante di **compilazione**, **esecuzione** e **distribuzione** dei container. Essi comunicano tramite un’**API REST**.

Docker Registry è un archivio di immagini docker.

Docker Desktop consente la **creazione** e la **condivisione** di applicazioni e microservizi containerizzati.

Un’**immagine Docker** è un pacchetto software leggero, autonomo ed eseguibile, che include tutto il necessario per eseguire un’applicazione: codice, runtime, strumenti di sistema, librerie e impostazioni. Le immagini dei container diventano container in fase di esecuzione.

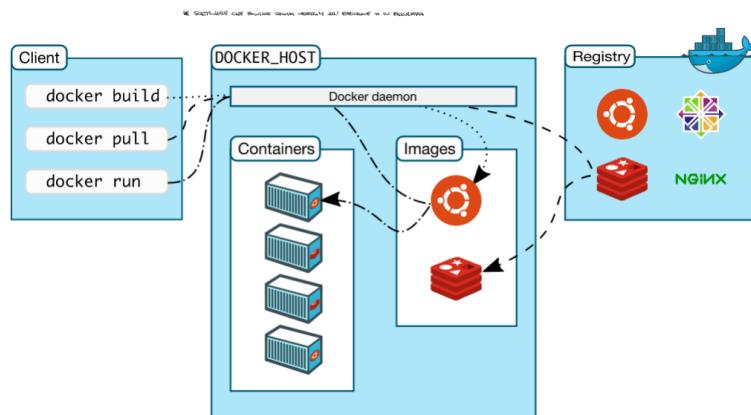


Figura 1.2: Architettura Docker ad alto livello

1.2.2 Architettura a basso livello

Docker usa **containerd** come container runtime, si tratta di un processo demone che permette di creare, avviare, fermare e distruggere i container. È possibile aggiungere **plugin** a containerd per estendere le sue funzionalità.

Containerd interagisce con l’**OCI runtime** che si occupa di creare i **namespace** e i **cgroups** necessari. OCI runtime fa quindi uso di funzionali linux per l’isolamento e il controllo delle risorse necessarie alla containerizzazione.

I **namespace** consentono di isolare gruppi di processi. In ogni namespace i processi hanno istanze isolate di risorse globali (utenti, PID, network, filesystem, ...).

I **cgroup** consentono di limitare le capacità e l’uso delle risorse da parte dei processi. Ad esempio è possibile limitare l’uso della CPU e della RAM, oppure limitare l’accesso a determinate porzioni del filesystem. Per ogni container creato il runtime crea un nuovo cgroup.

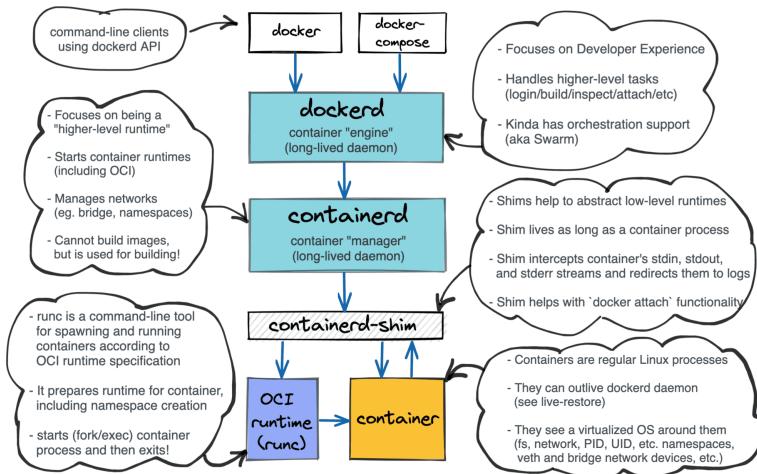


Figura 1.3: Architettura Docker a basso livello

1.2.3 Creazione di container

Il comando `docker run -i -t ubuntu /bin/bash` avvia un container in modalità interattiva basato sull'immagine `ubuntu` ed esegue il comando `/bin/bash` all'interno del container. La modalità interattiva permette di interagire con il container tramite terminale.

Quando viene creato un container con questa modalità vengono eseguite le seguenti operazioni:

1. se l'immagine non è presente viene scaricata dal Docker registry;
2. viene creato un nuovo container relativo all'immagine;
3. viene creato un filesystem associato al container;
4. viene creata un'interfaccia di rete associata al container e le viene assegnato un indirizzo IP;
5. viene avviato il container e viene eseguito il comando `/bin/bash`;

1.2.4 Dockerfile: creazione di immagini

Un dockerfile è un file di testo contenente una serie di istruzioni utilizzate da Docker per automatizzare la creazione di immagini Docker. Esempi di istruzioni base sono:

- **FROM** : specifica l'immagine base del container;
- **COPY** : copia file e directories dall'ambiente di compilazione del dockerfile nel file system del container;
- **RUN** : esegue un comando (ad esempio l'installazione di un'applicazione);
- **EXPOSE** : informa Docker che il container è in ascolto su una porta;
- **CMD** : comando eseguito di default quando si avvia il container, può essere sovrascritto specificando un comando all'avvio del container;
- **ENTRYPOINT** : comando che viene eseguito sempre all'avvio del container e non può essere sovrascritto da eventuali comandi aggiuntivi nel comando di creazione del container stesso;

A partire da un dockerfile è possibile creare un'immagine docker con il comando:

```
docker build -t nome_immagine:tag ./percorso
```

Per eseguire l'immagine creata con il dockerfile si usa il comando

```
docker run nome_immagine
```

Da notare che il container eseguito con quest'ultimo comando non esegue i comandi **RUN** specificati nel dockerfile, essi infatti vengono eseguiti soltanto in fase di build. Il container tuttavia esegue in fase di avvio i comandi specificati tramite le istruzioni **CMD** e **ENTRYPOINT**. Per sovrascrivere i comandi **CMD** è sufficiente specificare un comando nell'istruzione `docker run` nel seguente modo:

```

docker run nome_immagine echo "ciao"
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

```

Listing 1.1: Esempio di dockerfile

1.2.5 Applicazioni multi container

Con Docker è possibile definire applicazioni che fanno uso di più container contemporaneamente. È necessario avere applicazioni multicontainer nel caso in cui essa è composta da servizi che interagiscono, in questo modo si assegna ad ogni container un servizio.

Per creare un applicazione multicontainer è possibile utilizzare due modalità:

- **metodo naïve**: si crea un container alla volta e si connettono manualmente fra di loro. In questo modo bisogna stare attenti alle dipendenze e all'ordine di avvio dei vari container.
- **Docker compose**: è uno strumento di Docker con il quale è possibile definire l'applicazione multi container all'interno di un file yaml. In questo modo si può fare il deployment dell'intera applicazione attraverso un singolo comando senza preoccuparsi delle dipendenze e dell'ordine di avvio. Si utilizza un file yaml per la definizione dei servizi, delle reti e dei volumi necessari per l'applicazione. Ogni servizio definito nel file rappresenta un container Docker che fa parte dell'applicazione, è quindi possibile definire più servizi e specificare come interagiscono tra loro. Docker Compose gestisce automaticamente la rete tra i servizi che possono quindi comunicare.

```

version: "2"
services:
  my-application:
    build: ./my-application
    ports:
      - "8000:8000"
    environment:
      - CONFIG_FILE
  db:
    image: postgres
  redis:
    image: redis
    command: redis-server --save "" --appendonly no
    ports:
      - "6379"

```

Listing 1.2: Esempio di file Compose

Il precedente file compose consente la creazione di un'applicazione multicontainer composta da 3 servizi:

- my-application: utilizza un dockerfile per effettuare il build dell'immagine ed espone la porta 8000 del container sulla 8000 dell'host, in questo modo il servizio è accessibile anche dall'esterno dell'host. CONFIG_FILE permette di definire una variabile d'ambiente o un file di configurazione per il container.
- db: utilizza l'immagine *postgres* (db relazionale) per il servizio.
- redis: utilizza l'immagine *redis* (db key-value) per il servizio ed espone la porta 6379 solo all'interno della rete Docker.

1.2.6 Volumi

Docker Volumes consente di mantenere i dati al di fuori del ciclo di vita dei container, consentono inoltre di condividere dati fra più container.

Oltre ai volumi è possibile memorizzare i dati in maniera persistente tramite la modalità bind mount che permette l'accesso da parte del container ad una porzione del filesystem host tramite il suo percorso assoluto. Tuttavia è preferibile utilizzare i volumi dato che sono gestibili da CLI e sono più sicuri.

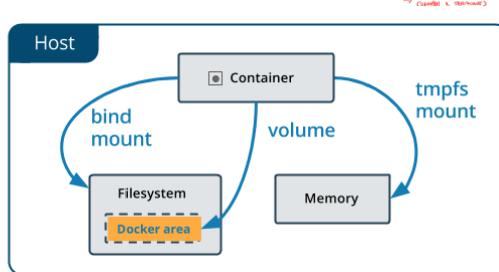


Figura 1.4: Volumi Docker

Per creare un volume è possibile utilizzare il comando:

```
docker volume create my-volume
```

Per montare un volume in un container è possibile utilizzare il comando:

```
docker run --mount source=[volume_name],destination=[path_in_container]
```

È possibile anche creare e montare volumi in maniera dichiarativa

```
services:
  web:
    image: nginx
    volumes:
      - nome_volume: percorso_mount
volumes:
  nome_volume:
  ...
  ...
```

Listing 1.3: Creazione e mount di volumi in modalità dichiarativa

1.2.7 Rete

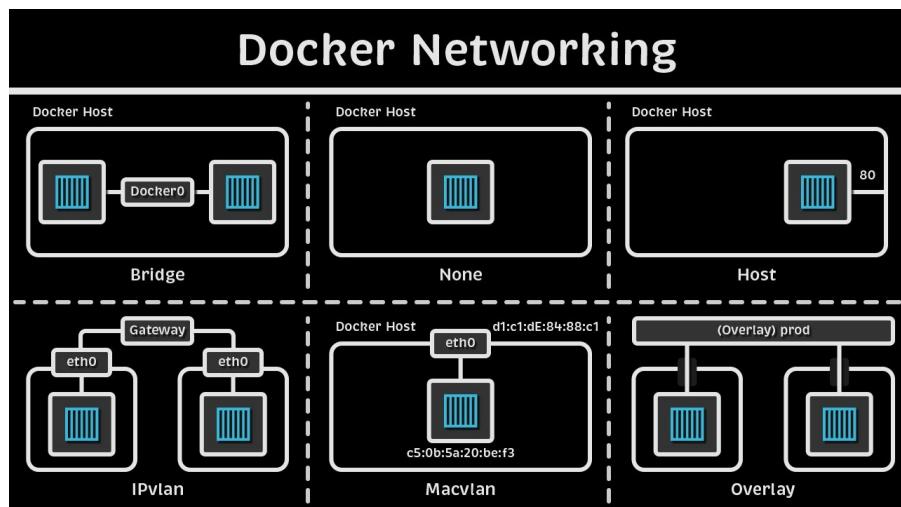


Figura 1.5: Docker networking

Docker offre vari tipi di configurazione di rete per i container:

- **Bridge:** crea un bridge (dispositivo Link Layer che inoltra il traffico tra i segmenti di rete grazie ad una tabella MAC e separa quindi il dominio di collisione) software all'interno dell'host e consente ai

container connessi allo stesso bridge di comunicare tra loro, isolando così i container non connessi allo stesso bridge. Viene quindi creata una rete privata isolata (ad esempio 192.17.0.0/16) in cui i container possono comunicare direttamente tra loro tramite indirizzo IP. È anche possibile utilizzare nomi di dominio per la comunicazione grazie al DNS (non supportato dal bridge default creato da Docker). È possibile esporre un container all'esterno tramite port-forwarding. Per la comunicazione con l'esterno si usa il NAT.

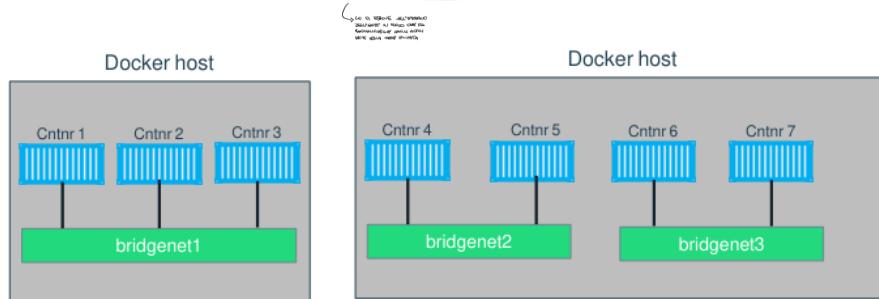


Figura 1.6: Docker bridge

Per creare una rete bridge è possibile utilizzare il seguente comando Docker:

```
docker network create -d bridge --subnet=192.168.0.0/16 --name bridgenet1
```

con cui si specifica anche il range di indirizzi IP da assegnare ai container all'interno della rete bridge.

Per connettere un container ad una rete bridge è possibile utilizzare il seguente comando:

```
docker network connect bridgenet1 my-container
```

Con questa modalità per comunicare dall'esterno con i container è necessario effettuare il port forwarding, cioè consente l'inoltro dei pacchetti tra il container e il kernel dell'host.

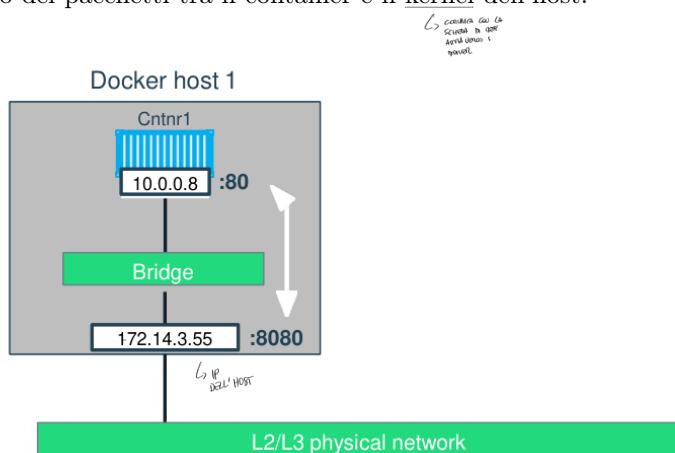


Figura 1.7: Port forwarding

Per effettuare il port forwarding è possibile utilizzare il seguente comando:

```
docker container run -p 8080:80 ...
```



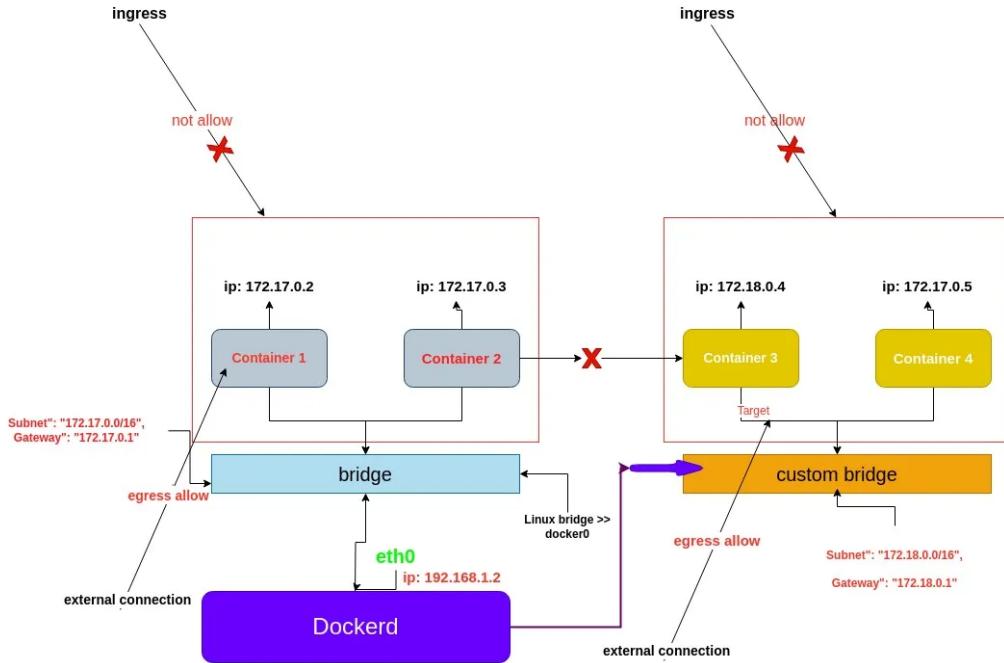


Figura 1.8: Esempio di rete bridge

- **Host:** i container usano lo stesso network namespace dell'host, ciò implica che hanno lo stesso IP dell'host e più container non possono usare la stessa porta o una porta usata dall'host. Ha performance migliori della modalità bridge dato che non richiede il NAT, tuttavia non garantisce nessun isolamento a livello di rete. Le applicazioni risultano come eseguite direttamente sull'host.

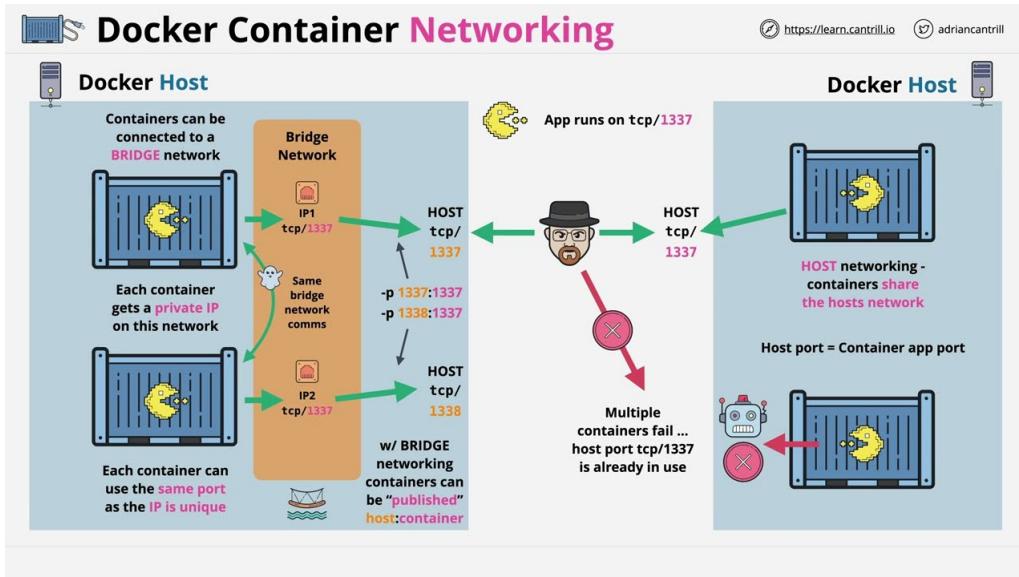


Figura 1.9: Differenza rete bridge/host

- **macvlan:** assegna un indirizzo MAC all'interfaccia virtuale del container facendo apparire il container come un dispositivo fisico direttamente connesso alla rete fisica. Consente inoltre di assegnare un indirizzo IP nella stessa subnet dell'host evitando il natting. Alcune applicazioni potrebbero richiedere di essere connesse direttamente sulla rete fisica, ad esempio applicazioni legacy o di monitoraggio del traffico.
- **ipvlan:** i container ricevono un indirizzo IP nella stessa subnet dell'host, non è quindi necessario il port-forwarding per esporre una porta di un container. A differenza di macvlan tutti i container ricevono lo stesso indirizzo MAC dell'host. Prevede due modalità:
 - L2: l'host agisce da switch;

- L3: l'host agisce da router;

La L3 isola completamente la rete dei container ma necessita di impostare le rotte per il routing.

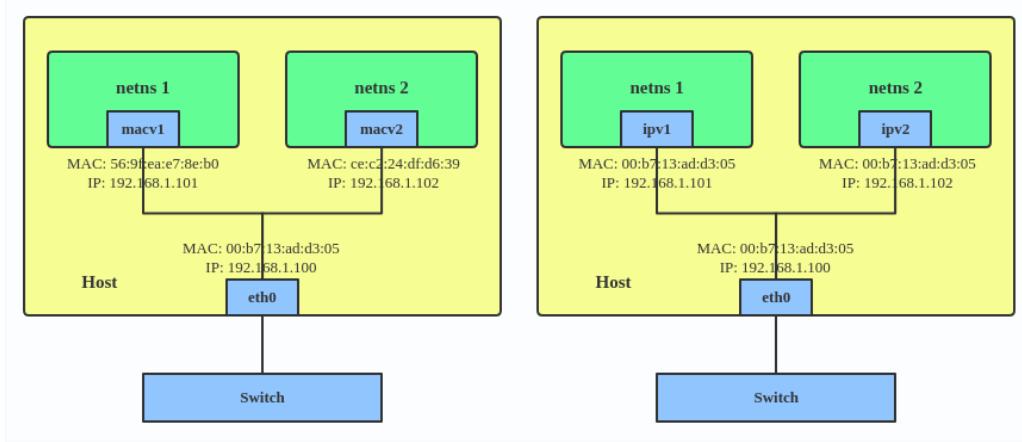


Figura 1.10: Differenza rete macvlan/ipvlan

- **Overlay:** crea una rete distribuita tra vari Docker Daemon, tale rete si appoggia sulla rete fisica degli host, per questo motivo si chiama rete overlay. Tale rete permette di far comunicare container su host diversi come se fossero sulla stessa rete. Docker si occupa del forwarding del traffico.

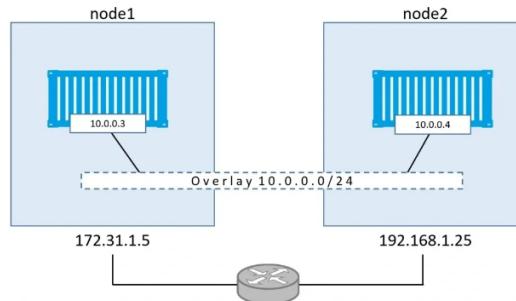


Figura 1.11: Rete overlay

- **none:** disconnette il container da qualsiasi rete. Si utilizza per lavori di batch.
- **Network plugin:** è possibile installare plugin di rete di terze parti

1.3 Kubernetes

L'orchestrazione dei container è il processo di gestione automatizzata di container software, che include il loro deployment, la gestione, lo scaling e il networking. I più popolari strumenti per l'orchestrazione sono Docker Swarm e Kubernetes.

Kubernetes permette di:

- controllare e automatizzare gli aggiornamenti e i deployment delle applicazioni
- montare e aggiungere storage per eseguire applicazioni stateful
- scalare le risorse delle applicazioni all'interno dei container
- attuare health check e safe heal delle applicazioni

1.3.1 Overview

Le entità fondamentali in Kubernetes sono:

- **Pod:** gruppo di uno o più container che condividono risorse di rete, volumi e ciclo di vita.

- **Nodo**: macchina fisica o virtuale che fa parte di un cluster kubernetes, i nodi forniscono le risorse computazionali su cui vengono eseguiti i pod.
- **Cluster**: insieme di nodi che eseguono applicazioni containerizzate. Ogni pod ha un IP unico all'interno del cluster.

All'interno dei cluster ci sono due tipologie di nodi:

- **master**: unico per ogni cluster e gestisce i nodi worker
- **worker**: ospita i pod che implementano le applicazioni

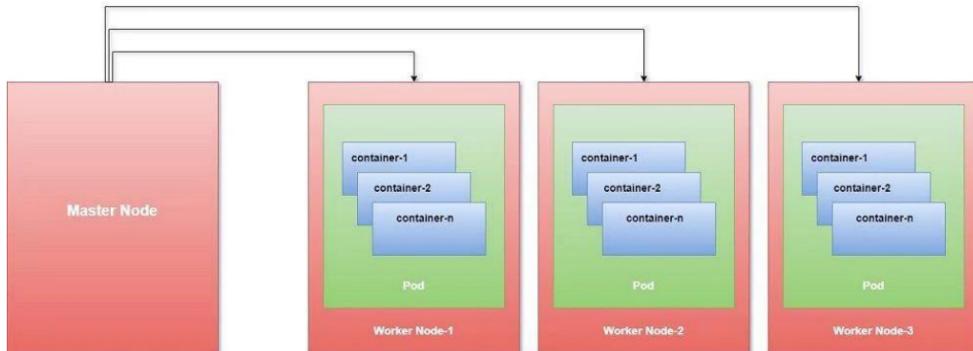


Figura 1.12: Componenti kubernetes

1.3.2 Oggetti Kubernetes

Gli **oggetti** sono entità persistenti che descrivono lo stato del cluster. In particolare, possono descrivere:

- cosa eseguono le applicazioni containerizzate e su quale nodo;
- le risorse disponibili alle applicazioni;
- le policy (restart, upgrade, fault-tolerance, ...) delle applicazioni;

Ogni oggetto Kubernetes è composto da 2 campi:

- **spec**: stato desiderato;
- **status**: stato corrente;

Il piano di controllo di Kubernetes (master node) si occupa di mantenere lo stato corrente coerente con quello desiderato. Ad esempio un oggetto di tipo Deployment rappresenta un'applicazione in esecuzione nel cluster. Tramite il campo spec è possibile specificare che si vogliono 3 istanze in esecuzione in contemporanea, mentre nel campo status ci sarà il numero effettivo di istanze in esecuzione.

La definizione di un oggetto consiste nello specificare il suo tipo, il suo nome e lo stato desiderato nel cluster (spec). Tale definizione avviene tramite un file manifest di tipo yaml del tipo:

```
apiVersion: apps/v1
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
```

Listing 1.4: Esempio di file manifest per la creazione di un oggetto

Il formato del campo spec dipende dal tipo di oggetto che si vuole creare. Alcune tipologie di oggetti sono: Namespace, Node, Pod, ReplicationController, Secret, Service, DaemonSet, Deployment, ReplicaSet, StatefulSet, HorizontalPodAutoscaler, CronJob, Job, DaemonSet, Ingress, Volume,

Per creare l'oggetto si usa il comando:

```
kubectl apply -f nome_manifest.yaml
```

Ogni oggetto Kubernetes ha un **nome** unico tra gli oggetti della stessa tipologia e un **UID** unico in tutto il cluster.

Le **label** sono coppie chiave/valore che possono essere associate agli oggetti (anche più di una per oggetto). Vengono utilizzate per organizzare e selezionare un sottoinsieme di oggetti.

Tramite un **selettor** è possibile identificare un sottoinsieme di oggetti. Esistono due tipologie di selettori:

- **equality-based**: tramite gli operatori `=` e `!=` si specifica i valori delle coppie chiave-valore;
- **set-based**: tramite gli operatori `in`, `notin` e `exists` si specifica il sottoinsieme dei valori associati ad una particolare chiave;

È possibile utilizzare l'operatore `AND` o `OR`, per concatenare più condizioni. Ad esempio per la seguente label `environment: production` è possibile utilizzare il seguente selettor equality-based `environment = production`, oppure il seguente selettor set-based `environment in (production, qa)`.

1.3.3 Namespace kubernetes

I **namespace** Kubernetes sono un meccanismo utile a suddividere un singolo cluster in più ambienti virtuali, ciascuno dei quali può avere le proprie risorse, utenti e politiche di accesso.

I namespace definiscono la visibilità dei nomi delle risorse, ogni nome deve essere unico all'interno del namespace ma non globalmente. Ciò è utile quando si vuole avere la stessa configurazione in ambienti diversi (ad esempio Development, Staging, Production).

Quando si crea un Service (ovvero quando si espone un'applicazione in esecuzione su uno o più pod tramite un unico endpoint) viene creata una entry DNS del tipo `<service-name>. <namespace-name>. svc.cluster.local`, se un container usa solo `<service-name>` si assumerà che tale servizio si trova nello stesso namespace del container.

Kubernetes include alcuni namespace predefiniti:

- **default**: namespace con il quale kubectl interagisce di default
- **kube-system**: contiene i componenti creati da Kubernetes (DNS, API-Server, ControllerManager, Scheduler, ...)
- **kube-public**: visibile e leggibile da tutti, contiene configurazioni e informazioni pubbliche
- **kube-node-lease**: contiene oggetti lease associati ai nodi per il controllo della loro salute. Gli oggetti lease sono utilizzati per gestire la leader election o il lock di componenti o processi distribuiti all'interno del cluster. Il leader ha il compito di gestire operazioni critiche e/o di coordinate gli altri pod o gruppi di nodi, in modo tale da evitare conflitti o operazioni duplicate.

Per visualizzare i namespace disponibili è possibile utilizzare il comando

```
kubectl get ns
```

Per creare un namespace è possibile utilizzare due modalità:

- imperativa: tramite il comando

```
kubectl create ns nome-ns
```

- dichiarativa: tramite un file yaml

```
kind: Namespace
metadata:
  name: nome.ns
```

Listing 1.5: file yaml per la creazione di namespace

e per crearlo si usa

```
kubectl create -f nomefile.yaml
```

1.3.4 Architettura master-worker

Un cluster Kubernetes consiste in un piano di controllo e un insieme di macchine worker, chiamate nodi, che eseguono le applicazioni containerizzate. I nodi worker ospitano i pod che sono le componenti dell'applicazione in esecuzione. Il piano di controllo gestisce i nodi e i pod, effettua decisioni globali e risponde agli eventi del cluster.

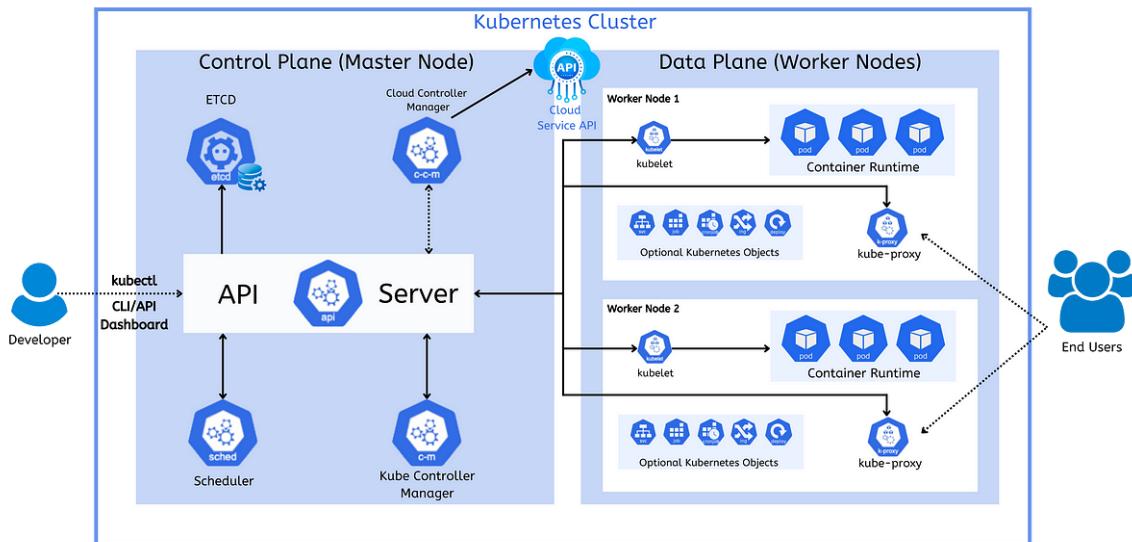


Figura 1.13: Architettura Kubernetes

Architettura nodo master

Il nodo Master presenta le seguenti componenti:

- **API Server**: espone un'API REST. Si ricorda che REST è uno stile architettonico per sistemi distribuiti che si basa su HTTP, è stateless, prevede l'uso di URL per identificare le risorse e implementa le operazioni GET, POST, PUT, PATCH e DELETE.
- **etcd**: archivio distribuito di tipo chiave-valore che memorizza lo stato del cluster. Un esempio di informazione memorizzata in etcd è lo stato dei pod (in esecuzione, in attesa, terminato, ...). La chiave è data dal percorso univoco che identifica la risorsa nel cluster (ad esempio /registry/pods/default/nginx-pod), mentre il valore può essere un file json contenente lo stato della risorsa.
- **scheduler**: decide dove eseguire un nuovo pod in base alle risorse richieste e disponibili. Considera vincoli e preferenze, come ad esempio:
 - **node affinity**: preferenza sulla collocazione dei pod sui nodi, ad esempio si può preferire che un pod venga eseguito su nodi con una certa etichetta
 - **taints**: vengono applicati ai nodi e impediscono lo scheduling su di essi
 - **tolerations**: vengono applicati ai pod e permettono di tollerare alcuni taint

- **controller manager**: esegue i controller (processi) che mantengono lo stato desiderato per una specifica risorsa. Ogni controller monitora lo stato delle risorse e adotta le azioni necessarie per ottenere lo stato desiderato. Esempi di controller: Node controller, Job controller, ServiceAccount controller,
- **cloud controller manager**: esegue controller gestiti dal provider del cloud. Se non c'è alcun provider tale componente non è presente.

Architettura nodo worker

Il nodo Worker presenta le seguenti componenti:

- **kubelet**: garantisce che i container siano in esecuzione correttamente, comunica con l'API Server e gestisce i volumi e la rete.
- **kube-proxy**: gestisce le regole di rete che permettono la comunicazione con i pod dall'interno o dall'esterno del cluster. Effettua il routing e il load balancing tra i pod. Se si usa un plugin di rete non è necessario utilizzare il kube-proxy.
- **container runtime**: il kubelet comunica tramite gRPC (Remote Procedure Call) Client con il gRPC Server del CRI (Container Runtime Interface). Il CRI consente di interagire con diversi runtime di container, ad esempio Docker, containerd e CRI-O.

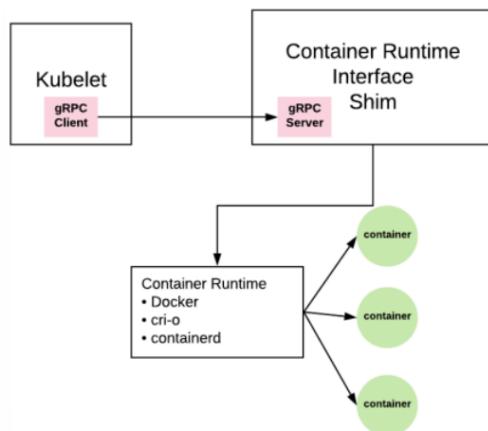


Figura 1.14: CRI

- **addons**: componenti esterne che estendono le funzionalità base di kubernetes. L'unico addon obbligatorio è *cluster DNS* che mantiene i record dei servizi kubernetes.

È possibile aggiungere un nodo al cluster in due modi:

- il kubelet del nodo si auto-registra al nodo master;
- tramite un oggetto di tipo *Node*:

```

1 {
2   "kind": "Node",
3   "apiVersion": "v1",
4   "metadata": {
5     "name": "10.240.79.157",
6     "labels": {
7       "name": "my-first-k8s-node"
8     }
9   }
10 }
  
```

Listing 1.6: Esempio di file json per la creazione di nodi

Due nodi non possono avere lo stesso nome e ogni nodo presenta uno stato in cui sono memorizzate le seguenti informazioni:

- Addresses: HostName, ExternalIP (disponibili all'esterno del cluster), InternalIP (disponibile all'interno del cluster);
- Conditions: Ready, DiskPressure, MemoryPressure, PIDPressure, NetworkUnavailable, ...;
- Capacity and Allocatable: descrive le risorse (CPU, memoria, pod eseguibili) disponibili sul nodo;
- Info: kernel version, kubelet and kube-proxy version, container runtime, ...;

1.3.5 Pod

Un workload è un'applicazione in esecuzione su Kubernetes. I workloads vengono eseguiti in un insieme di pod. Un pod è un insieme di uno o più container che condividono le risorse di storage e di rete. I container nello stesso pod sono quindi altamente accoppiati. Un pod può essere visto come un insieme di container con un namespace e un filesystem condiviso che eseguono un'unica applicazione.

I pod possono essere utilizzati in due modi:

- un container per pod: il pod funge da wrapper per il container e Kubernetes gestisce il pod invece di gestire direttamente il container;
- più container per pod: ogni pod contiene più container altamente accoppiati di un'applicazione;

Creazione di Pod

La creazione di un pod consiste nel definire un file di configurazione del pod e inviare tale configurazione al cluster Kubernetes tramite l'API Server.

```
kind: Pod
metadata:
  name: nginx
  namespace: default
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 80
```

Listing 1.7: Esempio di file yaml per la creazione di pod

Per creare il pod è possibile utilizzare il comando

```
kubectl apply -f nginx-pod.yaml
```

In genere non si creano pod con questa modalità ma si preferisce utilizzare le workload resources.

Per ottenere una lista dei pod in esecuzione si può usare il comando

```
kubectl get pods
```

Per ottenere una descrizione di un pod si usa il comando

```
kubectl describe pod name-pod
```

Per effettuare port forwarding da una porta locale ad un porta del pod si può usare il comando

```
kubectl port-forward <resource-type/resource-name> [local_port]:<pod_port>
```

Tale comando configura un proxy che inoltra tutto il traffico dalla porta locale specificata alla porta del pod.

Risorse condivise in un pod

In ogni pod è possibile specificare un insieme di volumi condivisi tra i vari container nel pod. Tali volumi sono persistenti, i dati non vengono quindi persi una volta eliminati i pod.

I container in un pod condividono il network namespace, hanno quindi lo stesso indirizzo IP (quello assegnato al pod) e le stesse porte. All'interno dei pod i container possono comunicare tramite localhost. Per comunicare con container in altri pod possono invece utilizzare la rete IP.

↳ https://kubernetes.io/docs/concepts/storage/persistent-volumes/

Init e sidecar container

Gli init container sono container che vengono eseguiti prima dei container dedicati all'applicazione. I container dell'applicazione vengono avviati quando gli init container terminano.

I sidecar container sono container ausiliari che forniscono un servizio secondario all'applicazione di interesse nel pod. Ad esempio si considera un web server (app container) che fornisce i dati memorizzati in un volume e un file puller (sidecar container) che aggiorna i dati nel volume.

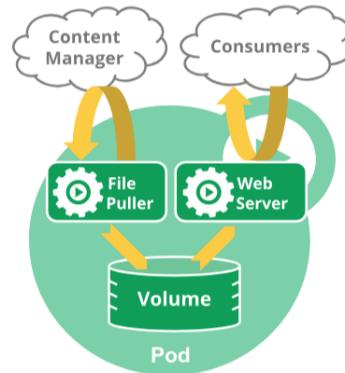


Figura 1.15: Sidecar container

Ciclo di vita dei pod

Ogni pod può trovarsi in 5 fasi:

- Pending: il pod è stato accettato dal cluster ma uno o più container non sono ancora in esecuzione. Il pod si trova in questa fase anche quando è in attesa dello scheduling o si stanno scaricando le immagini relative ai container.
- Running: il pod è stato associato ad un nodo e almeno un container si trova nello stato running, starting o restarting.
- Succeeded: tutti i container nel pod hanno terminato con successo la loro esecuzione.
- Failed: tutti i container nel pod hanno terminato la loro esecuzione e almeno uno ha fallito la sua esecuzione.
- Unknown: stato non noto a causa di problemi di comunicazione con il nodo.

I container possono trovarsi in 3 diversi stati:

- Waiting: il container non ha ancora completato lo start-up.
- Running: in esecuzione senza problemi.
- Terminated: esecuzione terminata (anche a causa di un errore).

Nel campo spec di un pod è possibile definire una restartPolicy con tre possibili valori:

- Always: riavvia il container automaticamente dopo la terminazione.
- OnFailure: riavvia il container solo dopo una terminazione dovuta ad un errore.
- Never: non riavvia il container dopo la terminazione.

1.3.6 Workload Resources

Il ciclo di vita dei pod viene gestito tramite le **workload resources**, tali risorse configurano i controller che si assicurano che il giusto numero di pod sia in esecuzione. Alcuni esempi di workload resources sono: Deployment, ReplicaSet, StatefulSet, DaemonSet, Job, CronJob,

Le workload resources definiscono un PodTemplate che viene usato come base dai controller per creare repliche dei pod.

Alcune delle workload resources più comuni sono:

- ReplicaSet: mantiene operativi un certo numero di pod replicati. Nel caso in cui il nodo sia a corto di risorse nel momento della duplicazione di un pod, verrà replicato su un altro nodo del cluster disponibile. Deployment invece è un oggetto che gestisce e aggiorna i ReplicaSet in modo dichiarativo.

```

apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # modify replicas according to your case
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
        - name: php-redis
          image: us-docker.pkg.dev/google-samples/containers/gke/gb-frontend:
            ↗ v5

```

Listing 1.8: Esempio di file yaml per la creazione di ReplicaSet

- StatefulSets: come Deployment esegue un gruppo di pod replicati, ma a differenza sua ogni pod ha una propria identità e non sono intercambiabili. Può risultare utile ad esempio quando un pod deve mantenere un volume persistente e ad un suo fallimento è possibile recuperare tale volume con l'id del pod.
- DaemonSet: assicura che tutti i nodi esegano una copia del pod. Ciò può essere utile per pod che forniscono funzionalità locali ai nodi. Ad esempio si possono usare per effettuare logging su tutti i nodi o per effettuare monitoring sui nodi.
- Job e Cronjob: permette di creare dei pods che hanno all'interno una certa logica, assicurando che verrà completata un numero specificato di volte. Con job il task viene eseguito solo una volta, con cronjob invece il task viene eseguito più volte in base a una schedule. Quando si crea un job va definita anche una restart policy, settata su OnFailure (quando il container fallisce viene reistanziato sullo stesso pod) o Never (viene reistanziato in un nuovo pod).

Per creare diversi tipi di Job, si settano due parametri: completions (quante volte il task deve essere completato) e parallelism (quanti pod vanno istanziate contemporaneamente).

1.3.7 Networking

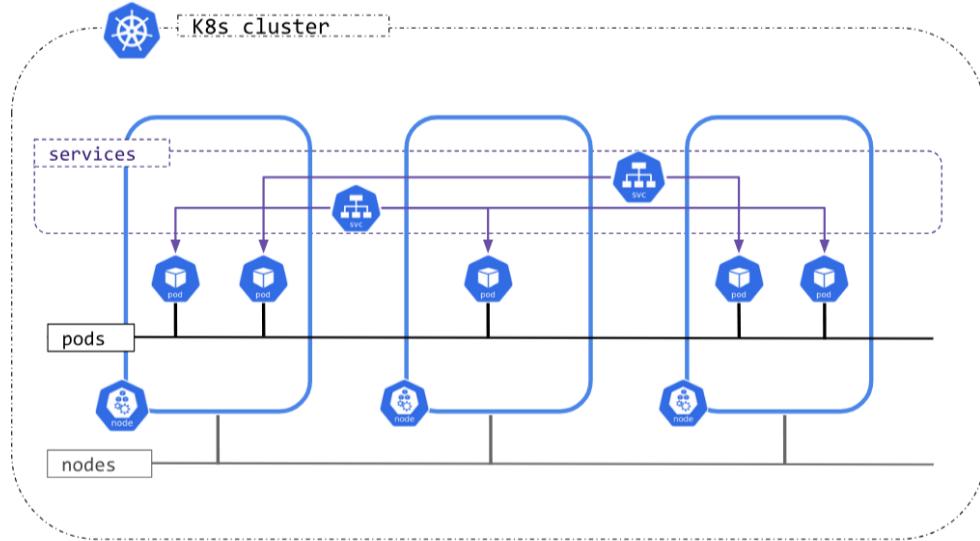


Figura 1.16: Kubernetes networking model

In un cluster Kubernetes è necessario allocare indirizzi IP univoci per pod, servizi e nodi a partire da un range di indirizzi configurati nelle seguenti componenti:

- il network plugin del container runtime è configurato per assegnare indirizzi IP ai pod;
- il kube-apiserver è configurato per assegnare indirizzi IP ai servizi;
- il kubelet è configurato per assegnare indirizzi IP ai nodi;

Il modello di networking di Kubernetes risolve i seguenti problemi di comunicazione:

- container-to-container;
- pod-to-pod;
- pod-to-service;
- external-to-service;

Comunicazione container-to-container

Docker di default assegna un indirizzo IP ad ogni container e connette i container tra di loro e all'host tramite un bridge.

Dato che Kubernetes assegna un unico indirizzo IP ai pod (tutti nella sottorete del cluster), i container all'interno di un pod devono utilizzare lo stesso indirizzo IP per la comunicazione. Al fine di poter comunicare tra di loro i container nello stesso pod fanno uso delle porte. Due container nello stesso pod potrebbero dunque comunicare utilizzando i seguenti indirizzi: `http://localhost:80` e `http://localhost:8080`. Due container nello stesso pod non possono dunque utilizzare la stessa porta.

Al fine di creare e mantenere la configurazione di rete condivisa da tutti i container in un pod, kubernetes crea in ogni pod un container speciale chiamato `pause`, il cui unico scopo è quello di mantenere la connettività tra i container all'interno del pod condividendo il suo network namespace con tutti i container.

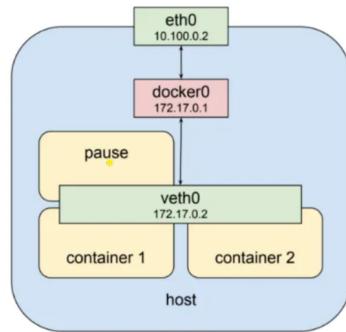


Figura 1.17: Comunicazione container-to-container

Comunicazione pod-to-pod

L'implementazione della comunicazione tra pod dipende dal particolare plugin CNI (Container Network Interface) in uso. Le specifiche del networking Kubernetes tuttavia impongono che la comunicazione tra pod (anche su nodi diversi) deve poter avvenire senza l'utilizzo di NAT. Il plugin CNI di base di Kubernetes è *kubenet*, esso si occupa di effettuare le seguenti operazioni:

- Crea un bridge linux chiamato *cbr0*.
- Crea una coppia di veth per ogni pod al fine di connettere i pod al bridge *cbr0*. Le veth sono dispositivi ethernet virtuali usati per connettere due network namespace diversi (in questo caso il network namespace del pod con il network namespace root del nodo).
- Assegna un indirizzo IP alla estremità lato pod della veth a partire da un range di indirizzi assegnato al nodo dal controller-manager.
- Imposta le rotte invocando il plugin IPAM (IP Address Management).

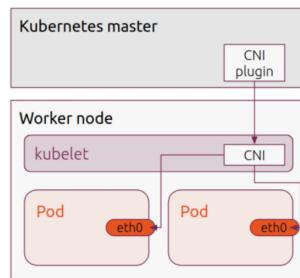


Figura 1.18: CNI

La comunicazione tra pod nello stesso nodo avviene quindi grazie al bridge software presente nel nodo. La comunicazione tra pod in nodi diversi avviene invece grazie ad un router/gateway esterno ai nodi che si occupa di inoltrare i pacchetti tra i vari nodi.

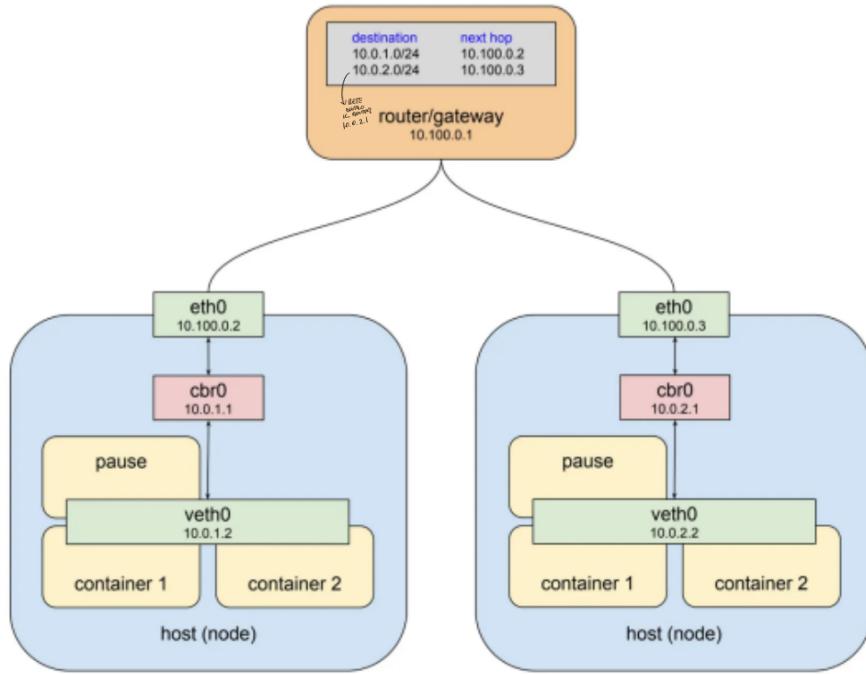


Figura 1.19: Comunicazione pod-to-pod

Oltre al plugin kubenet è possibile utilizzare anche altri plugin, ad esempio:

- Flannel: crea una VxLAN su cui vengono connessi tutti i nodi tramite una veth.
- Weave
- Calico: utilizza l'incapsulamento Ip-in-Ip per creare una rete overlay.
- AWS VPC

Service

Utilizzare l'indirizzo IP di un pod come endpoint (punto della rete in cui vengono resi disponibili servizi) non è una buona pratica dato che i pod sono effimeri (devono poter scalare e possono essere cancellati). L'indirizzo IP di un endpoint deve essere unico e non deve cambiare, lo scaling orizzontale (pratica per cui un servizio viene reso disponibile su più pod identici contemporaneamente) rende impraticabile l'uso degli IP dei pod come endpoint.

Per risolvere questo problema di solito si adotta un reverse-proxy, i client si connettono al proxy e il proxy si occupa di inoltrare la richiesta ai server. Il proxy deve quindi soddisfare i seguenti requisiti:

- deve essere sempre disponibile;
- deve mantenere una lista di server a cui inoltrare le richieste;
- deve conoscere lo stato di salute dei server;

Kubernetes implementa una soluzione chiamata Service che soddisfa i tre requisiti precedentemente elencati. Un Service consente quindi di esporre un'applicazione contenuta in un insieme di pod tramite una particolare policy. L'insieme di pod che il Service astrae viene individuato tramite un selettore. Il Service si occupa di instradare il traffico verso il pod corretto indipendentemente dal nodo in cui si trova. Ogni Service espone un indirizzo IP e un nome di dominio risolvibile tramite DNS.

Esistono varie tipologie di Service:

- ClusterIP: espone il servizio su un indirizzo IP interno al cluster, rendendo quindi il servizio disponibile solo all'interno del cluster. È possibile esporre il servizio all'esterno usando un Ingress o un Gateway.
- NodePort: espone l'applicazione sull'IP di ogni pod su una porta specifica, statica (la NodePort). Per fare sì che la porta sia raggiungibile, viene creato automaticamente un ClusterIP.

- **LoadBalancer**: Espone il servizio esternamente utilizzando un load balancer. Kubernetes non offre direttamente una componente load balancer, quindi andrà fornita in altri modi
- **ExternalName**: mappa le richieste al Service su un dominio esterno al cluster

Comunicazione pod-to-service: clusterIP

Al fine di contattare un Service un pod utilizza il servizio DNS per ottenere il suo indirizzo IP (chiamato clusterIP). Una volta ottenuto il suo indirizzo IP utilizza il router per instradare la richiesta al nodo in cui è contenuto il pod che implementa il servizio. Tuttavia il router presenta nella tabella di routing solamente le regole necessarie ad instradare i pacchetti verso lo spazio degli indirizzi dei pod e il clusterIP appartiene allo spazio degli indirizzi dei Service.

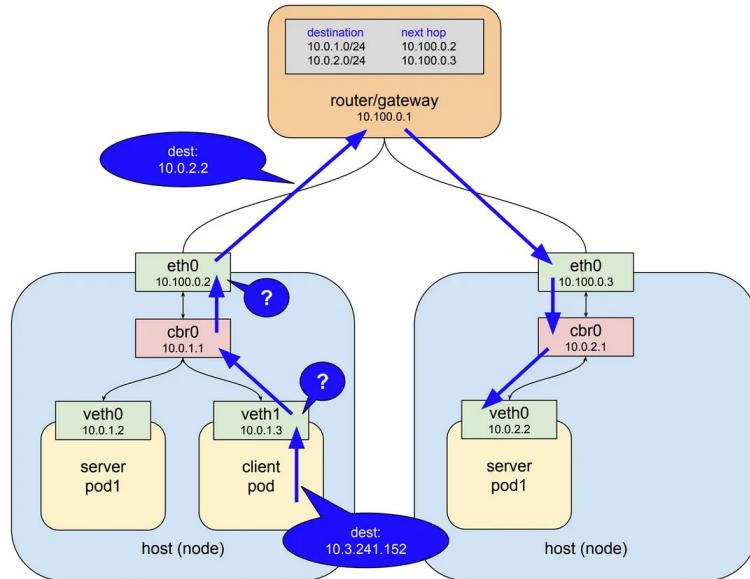


Figura 1.20: Instradamento pod-to-service

È quindi necessario "tradurre" la richiesta verso un clusterIP ad una richiesta verso un podIP. Per fare ciò esistono 2 modalità:

- **Kubernetes legacy:**

- si fa uso di **kube-proxy** che apre una porta per ogni Service nel cluster;
- sull'interfaccia del pod è configurato netfilter che traduce le richieste a clusterIP in richieste al proxyIP + porta associata al servizio;
- il proxy fornisce l'ip di un pod su cui è implementato il Service;
- la richiesta si è quindi tradotta in una normale comunicazione pod-to-pod;

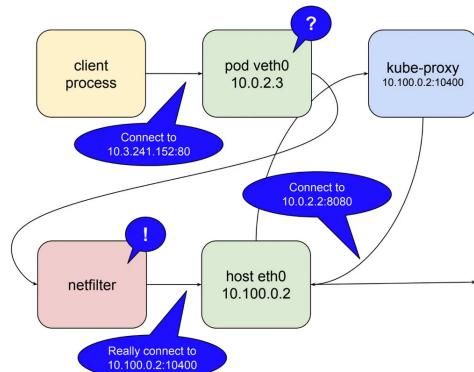


Figura 1.21: Traduzione di richieste a clusterIP in richieste ai pod

- Kubernetes 1.2, al fine di evitare il passaggio dal kernel-space al kube-proxy (user-space) e di nuovo al kernel-space si utilizza il seguente approccio:
 - il kube-proxy configura (tramite iptables) netfilter in modo che esso traduca gli indirizzi clusterIP in indirizzi podIP;
 - la richiesta si è quindi tradotta in una normale comunicazione pod-to-pod direttamente da netfilter;

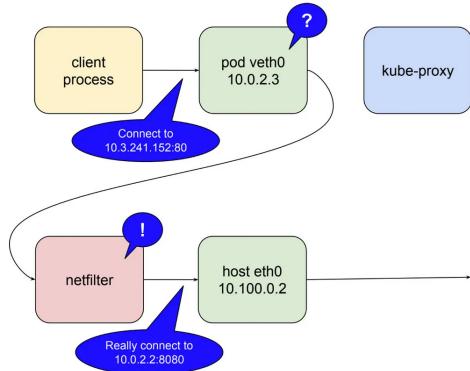


Figura 1.22: Traduzione di richieste a clusterIP in richieste ai pod

In entrambi i casi il kube-proxy è configurato dal master node.

Comunicazione external-to-service

Si è visto che tramite ClusterIP e kube-proxy + netfilter un pod all'interno di un nodo può contattare un Service. Si vuole ora analizzare la situazione in cui un dispositivo al di fuori di un nodo del cluster vuole contattare un Service.

Al di fuori del cluster

Per risolvere tale problema basterebbe aggiungere una regola al router che inoltra il traffico diretto alla subnet dei Service verso un nodo qualunque, in questo modo da dentro il nodo sarebbe possibile contattare il Service con il meccanismo ClusterIP visto in precedenza. Tuttavia ciò non è possibile perché i nodi sono effimeri. È quindi necessario che al router arrivi direttamente come destinazione un indirizzo di un nodo.

Una soluzione che permetta di contattare un Service dall'esterno di un nodo deve risolvere i seguenti problemi:

- Tradurre un indirizzo ClusterIP in un indirizzo di nodo prima che la richiesta arrivi al router. Tale problema è risolto dal Service di tipo LoadBalancer.
- Una volta che il router ha inoltrato la richiesta all'interno di un nodo del cluster, si traduce tale richiesta in una richiesta clusterIP al fine di usare la normale comunicazione pod-to-service già vista. Tale problema è risolto dal Service di tipo NodePort.

Comunicazione external-to-service: NodePort

*LOAD
BALANCER*

Se la richiesta al Service proviene dall'esterno del nodo essa avrà un nodeIP come destinazione (per i motivi spiegati in precedenza). È quindi necessario tradurre tale richiesta in una richiesta clusterIP. Per distinguere i normali pacchetti destinati al nodo da quelli che sono richieste al Service si usa una particolare porta nel range 30000-32767 (diversa per ogni servizio). Una richiesta proveniente dall'esterno diretta verso il Service sarà nella forma *nodeIP:ServicePort*, il kube-proxy sarà quindi in ascolto sulla porta *ServicePort* e tradurrà *nodeIP:ServicePort* in *clusterIP:port*. In questo modo il proxy traduce una richiesta esterna al nodo in una richiesta equivalente ad una interna al nodo. È quindi possibile utilizzare la normale comunicazione pod-to-service.

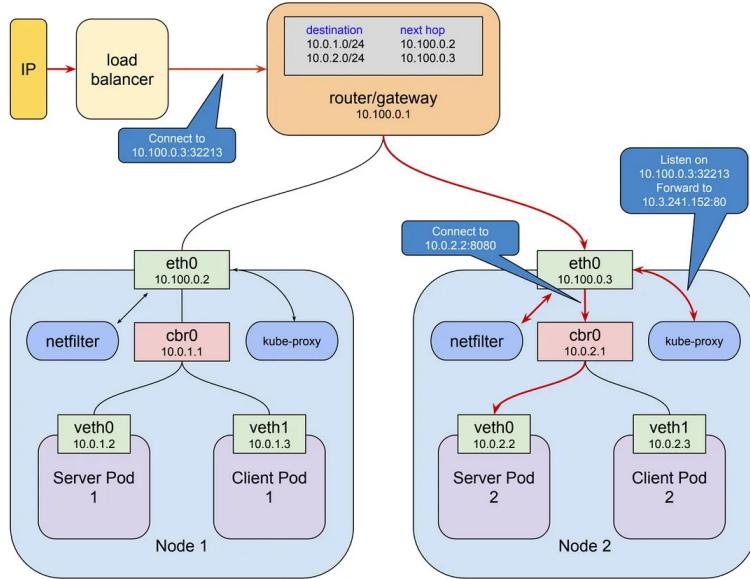


Figura 1.23: NodePort

Comunicazione external-to-service: LoadBalancer

Il Service di tipo LoadBalancer permette di esporre un servizio su un IP pubblico (externalIP). Il LoadBalancer si occupa quindi di tradurre le richieste provenienti dall'esterno del cluster in richieste ad un nodo disponibile (healthy) e con la NodePort associata al servizio richiesto. Il LoadBalancer può gestire un solo servizio alla volta dato che lavora a strato 3, non può quindi per esempio esporre una porta per ogni servizio e tradurla nella corrispondente NodePort. Il LoadBalancer quindi può solo effettuare “traduzioni” $externalIP-nodeIP:NodePort$ e non $externalIP:port-nodeIP:NodePort$.

Da notare che LoadBalancer fa uso di NodePort, tuttavia è possibile utilizzare NodePort senza fare uso di un LoadBalancer, in questo caso però è necessario conoscere l'IP di un nodo healthy. Quindi NodePort senza LoadBalancer è utilizzato di solito solo in fase di test in cui gli sviluppatori si occupano di fornire l'IP di un nodo healthy. In fase di produzione invece si preferisce sempre usare LoadBalancer dato che gli utenti non hanno modo di individuare l'IP di un nodo healthy.

Service di tipo ExternalName

Si è fin'ora visto come esporre servizi implementati in un pod tramite Service. In alcuni casi potrebbe essere utile esporre servizi esterni al cluster utilizzando sempre un oggetto Service. Ad esempio se si sta migrando un'applicazione all'interno di kubernetes è buona pratica spostare una componente alla volta e testarla singolarmente, in questo caso è possibile rendere disponibili le altre componenti come servizi esterni al cluster. Per fare ciò kubernetes offre la tipologia di Service chiamata ExternalService. ExternalService invece di redirigere le richieste per un servizio verso un pod interno al cluster le redirige verso un nome DNS. In questo modo è possibile utilizzare un nome locale interno al cluster per mascherare un servizio esterno.

Comunicazione external-to-multipleServices: Ingress

Si è visto che il LoadBalancer non ha la capacità di esporre più di un servizio al di fuori del cluster dato che non ha accesso al layer 4. L'oggetto di tipo Ingress tuttavia è capace di definire un load balancer layer 4, invece di effettuare un mapping $externalIP-nodeIP:NodePort$ è capace di effettuare un mapping $externalIP:URL-nodeIP:NodePort$. Tramite Ingress è quindi possibile esporre all'esterno del cluster più servizi, assegnando ad ogni servizio un URL.

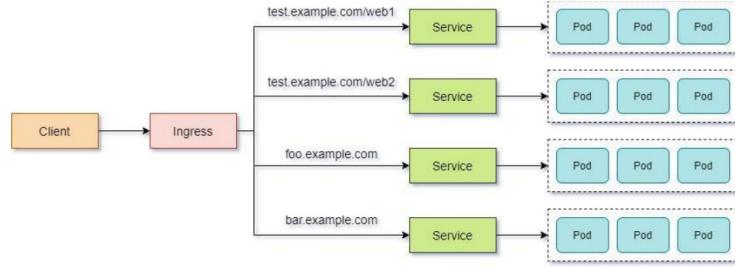


Figura 1.24: Ingress

Al fine di criptare la comunicazione tra il client esterno e Ingress è possibile utilizzare un Secret. Un Secret è un oggetto kubernetes che consente di non dover memorizzare informazioni confidenziali all'interno del codice dell'applicazione. Nella definizione di un Ingress è quindi possibile specificare l'uso di un Secret, tale specifica farà sì che la comunicazione client-Ingress sarà criptata tramite TLS.

1.3.8 Volumi

I container all'interno dei pod sono effimeri e il loro filesystem non è persistente, in alcune applicazioni tuttavia si vorrebbero memorizzare dati al di fuori del ciclo di vita dei container. Inoltre, in alcune applicazioni i container all'interno di un pod necessitano di condividere file. Si consideri un pod composto da 3 container:

- WebServer: rende disponibili pagine html e salva i log di accesso;
- ContentAgent: genera nuove pagine html;
- LogRotator: processa i log;

Il WebServer deve poter accedere alle nuove pagine generate dal ContentAgent e il LogRotator deve poter accedere ai log scritti dal WebServer.

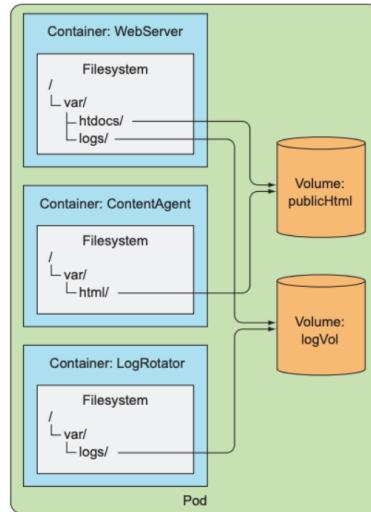


Figura 1.25: Volumi Kubernetes

Per risolvere questo problema Kubernetes fa uso di **volumi**, ovvero di una risorsa di storage interna ai pod persistente alle cancellazioni di container.

I dati memorizzati nei volumi possono essere persistenti alla cancellazione dei pod oppure effimeri dipendentemente dalla tipologia di volume. Esistono varie tipologie di volumi, alcuni esempi sono:

- **EmptyDir**: directory temporanea dipendente dal ciclo di vita del pod.
- **HostPath**: permette di accedere ad una directory nel filesystem dell'host.
- **NFS**: permette di accedere ad una directory in un host remoto.
- **Secret**: utilizzato per memorizzare informazioni riservate come password

```

apiVersion: v1
kind: Pod
metadata:
  name: shared-volume-emptyDir
spec:
  containers:
    - name: alpine1
      image: alpine
      command: ["/bin/sleep", "999999"]
      volumeMounts:
        - mountPath: /alpine1
          name: data
    - name: alpine2
      image: alpine
      command: ["/bin/sleep", "999999"]
      volumeMounts:
        - mountPath: /alpine2
          name: data
  volumes:
    - name: data
      emptyDir: {}

```

Listing 1.9: Esempio di file yaml per la creazione e il mounting di un volume di tipo EmptyDir

1.3.9 HPA

L'HPA (Horizontal Pod Autoscaler) aggiorna le workload resources (ad esempio un Deployment) allo scopo di adattarsi ad un particolare carico di lavoro. Nello scaling orizzontale il numero di repliche di un pod che implementano un'applicazione viene aggiornato dinamicamente in base al numero di richieste a tale applicazione, nello scaling verticale invece vengono adattate le risorse (CPU, memoria, ...) disponibili ai singoli pod.

Per implementare l'HPA si definisce un oggetto di tipo *HorizontalPodAutoscaler*, tale oggetto andrà a definire il comportamento di un HPA Controller, ovvero un processo in esecuzione nel master node, che periodicamente andrà ad aggiustare il numero di repliche per un particolare workload target.

```

apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: guestbook-frontend
  namespace: default
spec:
  maxReplicas: 10
  metrics:
    - resource:
        name: cpu
        target:
          averageUtilization: 50
          type: Utilization
        type: Resource
  minReplicas: 1
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: frontend

```

Listing 1.10: Esempio di file yaml per la creazione di un HPA

Funzionamento di HPA

HPA funziona nel seguente modo:

1. Si interroga periodicamente il metric server per ottenere informazioni sull'uso delle risorse dei pod.

2. Si calcola il numero di repliche necessarie per soddisfare i requisiti di risorse imposti (ad esempio si vuole che i pod consumino 200m di CPU). Il numero di repliche desiderato viene calcolato nel seguente modo:

$$\text{desiredReplicas} = \text{ceil}(\text{currentReplicas}(\text{currentMetricValue}/\text{desiredMetricValue}))$$

3. Si aggiorna il workload resource che controlla il numero di repliche dei pod.

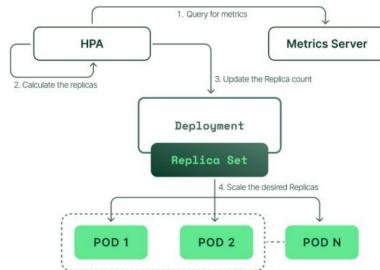


Figura 1.26: Funzionamento di HPA

Architettura di HPA

Per poter funzionare in un cluster, l'HPA necessita delle seguenti componenti:

- **cAdvisor**: processo demone in esecuzione nei nodi che raccoglie e aggrega le informazioni sul consumo di risorse dai container e le rende disponibili al kubelet.
- **kubelet**: raccoglie le statistiche dal cAdvisor e le rende disponibili tramite *Summary API*.
- **metrics-server**: raccoglie e aggrega le informazioni dai kubelet e le rende disponibili tramite *Metric API*. L'HPA controller e kubectl possono accedere alle statistiche tramite l'API Server che sfrutta la Metric API del metric server.

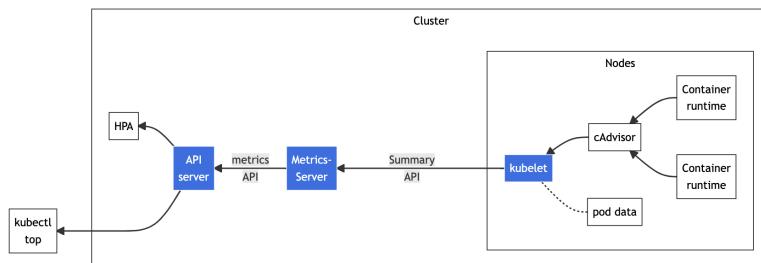


Figura 1.27: Architettura di HPA

Personalizzazione dell'HPA

L'intervallo di funzionamento dell'HPA è personalizzabile tramite il parametro `--horizontal-pod-autoscaler-sync-period` del kube-controller-manager. La risorsa da monitorare si imposta tramite il parametro `scaleTargetRef` mentre i pod da monitorare si individuano tramite il selettore specificato in `spec.selector`.

Le statistiche di utilizzo possono essere raccolte in modalità raw oppure come percentuale rispetto al valore desiderato.

Utilizzando la v2 dell'HPA è possibile utilizzare il campo `behavior` per configurare separatamente lo `scaleUp` e lo `scaleDown`. È anche possibile specificare più behavior contemporaneamente e l'HPA utilizzerà quella che effettua il maggior numero di cambiamenti. Il parametro `stabilizationWindowSeconds` permette di ridurre il flapping, ovvero fluttuazioni troppo frequenti nel numero di pod. La window stabilisce una finestra temporale in cui vengono fatte proposte di aggiornamento del numero di repliche e per lo scaling up sceglie la proposta con il numero più alto di repliche, mentre per lo scaling down sceglie la proposta con il numero più basso di repliche.

`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

```

Listing 1.11: Esempio di file yaml per la creazione di un HPA

1.4 VxLAN

1.4.1 VLAN

VLAN è una tecnologia di rete che consente di suddividere una rete fisica in più reti logiche separate.

Le VLAN sfruttano i tag nei frame Ethernet per identificare il traffico appartenente a una VLAN specifica.

Gli switch VLAN-capable sono capaci di gestire il traffico tra VLAN diverse o instradare i dati verso una VLAN specifica. Le porte dello switch vengono divise in due categorie:

- Access Port: assegnata a una singola VLAN; è usata per collegare dispositivi come PC o stampanti.
- Trunk Port: trasporta il traffico di più VLAN; è usata per collegare switch tra loro o con un router.

Le VLAN sono progettate per essere logicamente separate: i dispositivi in VLAN diverse non possono comunicare direttamente. La comunicazione tra VLAN richiede un dispositivo di livello 3, come un router o uno switch L3, per inoltrare i pacchetti tra reti diverse.

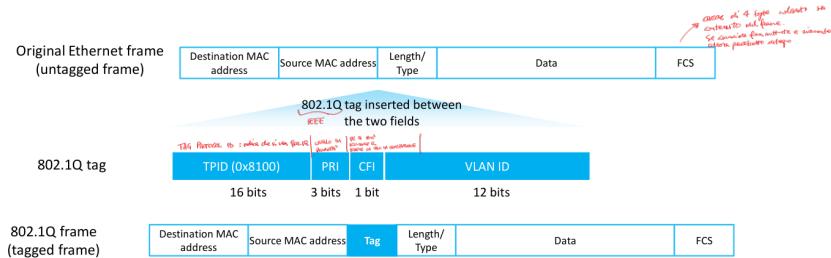


Figura 1.28: Tag VLAN nei frame ethernet

1.4.2 Datacenter

Un datacenter è una struttura fisica che ospita server, storage, dispositivi di rete ed ha lo scopo di supportare applicazioni, servizi web e l'archiviazione dei dati.

Architettura tradizionale

L'architettura tradizionale di un datacenter è composta da 3 livelli:

- Core layer: composto da router o switch L3 che effettuano l'instradamento tra diverse VLAN e gestiscono il traffico tra il datacenter e il mondo esterno.
- Aggregation layer: switch L2/L3 connessi direttamente al livello di accesso. Inviano il traffico dall'access layer al core layer e implementano funzionalità di sicurezza come firewall e IDS. Uno switch aggregation connette i dispositivi della stessa VLAN.

- Access layer: switch L2 il cui scopo è fornire una connessione diretta ai dispositivi finali (server, storage, ...).

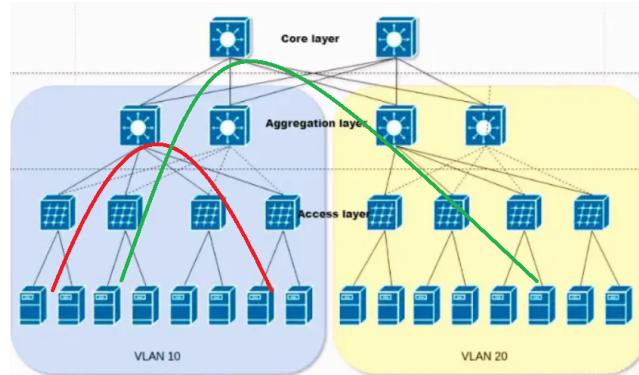


Figura 1.29: Architettura tradizionale nei datacenter

L'architettura tradizionale è ottima per il flusso di traffico nord-sud, ovvero per le comunicazioni esterno-dispositivo finale, dato che il core layer si occupa di instradare in modo efficiente il traffico verso la porzione di rete corretta. Tuttavia, il traffico est-ovest, ovvero quello tra dispositivi finali dello stesso datacenter, risulta inefficiente a causa dell'elevato numero di salti necessari per raggiungere i dispositivi finali. Molte interfacce dello strato aggregation inoltre vengono bloccate dal protocollo STP al fine di evitare i loop introducendo un notevole spreco soprattutto nel caso di datacenter di dimensioni elevate.

Tale architettura non si adatta quindi alle necessità introdotte dalla virtualizzazione. La virtualizzazione dei dispositivi finali infatti introduce un notevole incremento del traffico est-ovest e la necessità di dover migrare continuamente una VM mantenendo la sua configurazione di rete.

Architettura spine-leaf

L'architettura spine-leaf di un datacenter è composta da 2 livelli:

- Leaf: switch L2 che connettono i dispositivi finali dello stesso rack.
- Spine: switch L3 che connettono tutti i leaf.

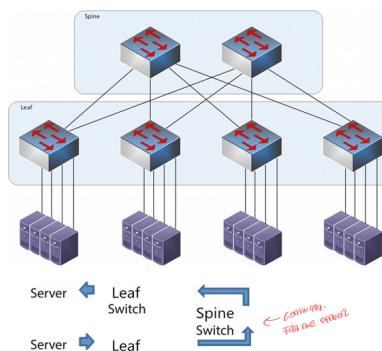


Figura 1.30: Architettura spine-leaf nei datacenter

Grazie all'architettura leaf-spine il dominio di broadcast L2 è limitato ai leaf, non è quindi necessario utilizzare STP. Per aggiungere un nuovo dispositivo di rete è sufficiente aggiungere un leaf e connetterlo a tutti gli spine. Il traffico est-ovest è a bassa latenza dato che sono necessari solo 3 hop per la comunicazione tra dispositivi finali.

1.4.3 Multitenancy

Un tenant è un entità che utilizza una o più risorse all'interno di un datacenter. In quest'ottica ci sono due modalità per gestire le risorse del DC:

- **single tenant:** le macchine virtuali istanziate su una certa macchina fisica sono utilizzate dallo stesso tenant

- **multi tenant**: le macchine virtuali sono utilizzate da più tenant contemporaneamente. In questo caso è necessario che si preveda un architettura capace di separare i dati per ciascun tenant.

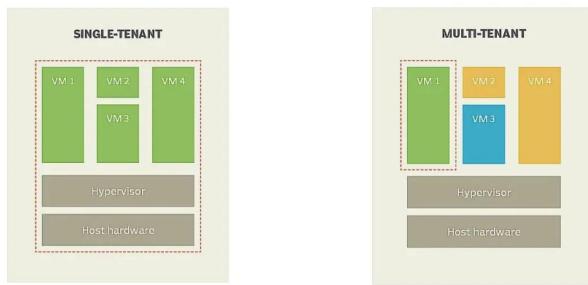


Figura 1.31: Architettura single vs multi tenant

1.4.4 Problemi nei datacenter

Le VLAN presentano diversi problemi quando si tratta di applicarle a datacenter anche di piccole dimensioni. Infatti, il tag VLAN è costituito da soli 12 bit, il che significa che è possibile identificare un massimo di 4096 VLANs. E' impossibile quindi soddisfare i requisiti di isolamento per un numero elevato di tenant.

Al fine di mantere il servizio che offrono, le VM necessitano di mantenere la configurazione di rete (IP e MAC) durante una migrazione. Per fare ciò in un datacenter è possibile adottare due soluzioni:

- si effettua la migrazione a strato 2, si ha però un dominio di migrazione troppo piccolo vincolato dall'architettura di rete;
- si riconfigura la rete, troppo oneroso da effettuare ad ogni migrazione (molto frequenti in ambienti virtualizzati);

Le VxLAN risolvono i due problemi precedentemente presentati nel seguente modo:

- Utilizza un identificatore di 24 bit chiamato VNI (VxLAN Network Id) sufficiente anche in ambienti multitenant.
- Il raggio di migrazione delle VM è potenzialmente illimitato dato che VxLAN incapsula le trame ethernet in pacchetti IP creando una rete a strato 2 estendibile a piacere.

1.4.5 Soluzione VxLAN

Una **overlay network** è una rete virtuale costruita sopra una rete fisica per fornire un'infrastruttura logica indipendente dalla topologia fisica. Nell'ambito di VxLAN, una overlay network permette di estendere segmenti di rete Layer 2 su una rete Layer 3, fornendo un modo scalabile e flessibile per interconnettere risorse distribuite, come data center o macchine virtuali.

VxLAN utilizza un meccanismo di incapsulamento per trasportare frame Ethernet su una rete IP. Questo è reso possibile incapsulando i pacchetti Ethernet originali in un header VxLAN, che a sua volta è incapsulato in UDP/IP.

Gli elementi chiave di una overlay network VxLAN sono i VTEP (VxLAN Tunnel Endpoints), che si occupano di:

- Incapsulare i frame Ethernet nei pacchetti VxLAN quando entrano nella rete overlay.
- Decapsulare i pacchetti VxLAN in frame Ethernet quando escono dalla rete overlay.

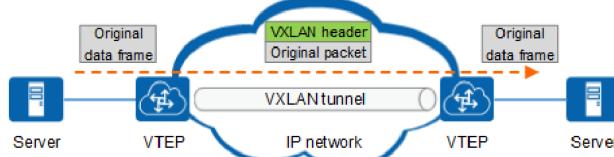
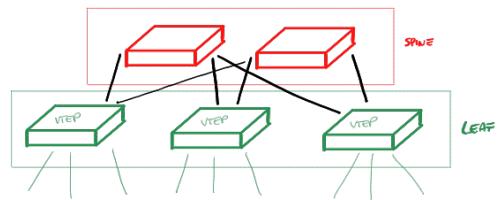


Figura 1.32: Funzionamento di VxLAN



Nella architettura spine-leaf:

- gli switch spine non sono a conoscenza della VxLAN;
- gli switch leaf fungono da VTEP;

Ogni overlay network in VxLAN è identificata da un VNI (VxLAN Network Identifier) a 24 bit, che consente di creare fino a 16 milioni di reti logiche isolate. La rete sottostante (underlay network) si occupa solo del routing e della consegna dei pacchetti a livello IP. La topologia e i dettagli della rete overlay sono completamente astratti dall'infrastruttura fisica.

1.4.6 Header VxLAN

Quando un pacchetto entra nella rete overlay:

- Il VTEP riceve un pacchetto Ethernet dalla rete locale.
- Il VTEP incapsula il frame Ethernet aggiungendo l'intestazione VXLAN e incapsulando il tutto in un pacchetto UDP.
- Il pacchetto VXLAN risultante viene inviato nella rete Layer 3 fisica (underlay).

Quando un pacchetto esce dalla rete overlay:

- Il VTEP riceve un pacchetto VXLAN dalla rete sottostante.
- Il VTEP rimuove l'intestazione VXLAN estraendo il frame Ethernet originale.
- Il frame Ethernet viene inoltrato verso la destinazione nella rete locale.

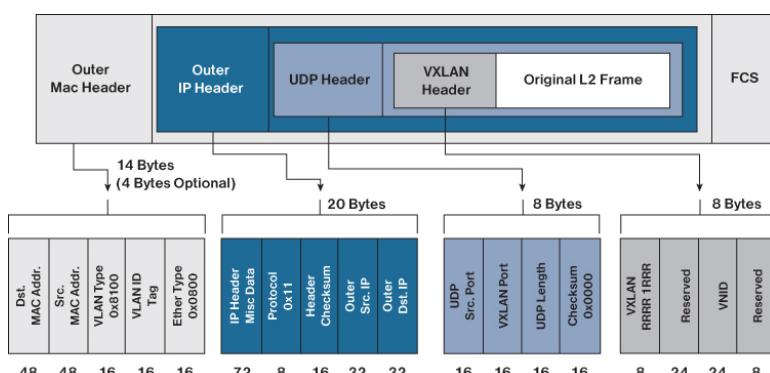


Figura 1.33: Header VxLAN

La destination port UDP è sempre fissata a 4789. Gli IP sorgente e destinazione sono quelli del VTEP sorgente e destinazione. Il MAC sorgente è quello del VTEP, mentre quello di destinazione è del next hop.

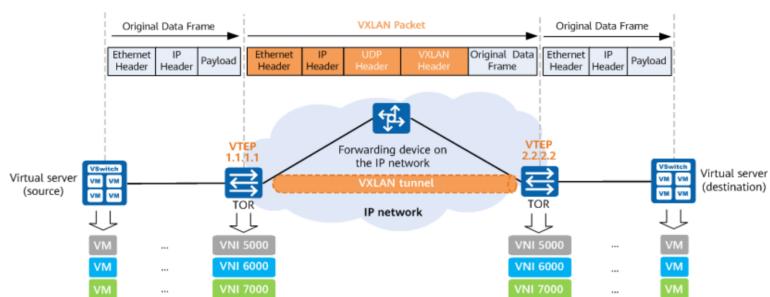


Figura 1.34: Tunneling VxLAN

1.4.7 Meccanismi di forwarding

L'utilizzo di VxLAN introduce un notevole overhead negli header dato che il traffico di segnalazione raddoppia (doppiate intestazioni L4/L3/L2 + header VxLAN). È quindi necessario configurare il forwarding nei VTEP al fine di ottimizzare al massimo la trasmissione dei dati.

Nelle reti VxLAN i pacchetti vengono classificati in due tipologie in base alla modalità di forwarding:

- **BUM:**

- Broadcast: pacchetti inviati a tutti gli host in una rete Layer 2 (indirizzo di destinazione MAC FF:FF:FF:FF:FF:FF).
- Unknown Unicast: pacchetti indirizzati a un MAC sconosciuto alla tabella di forwarding del VTEP.
- Multicast: pacchetti destinati a un gruppo specifico di host identificati da un indirizzo multicast.

- **Known unicast:** il destinatario è un unico utente ed è presente una regola di forwarding per poterlo raggiungere;

Tabella MAC-to-VTEP

La tabella MAC-to-VTEP associa l'indirizzo MAC del dispositivo finale (identifica un host specifico nella rete) all'indirizzo IP del VTEP responsabile di gestire il traffico destinato a quel dispositivo. Questa mappatura è necessaria perché i dispositivi nella rete overlay comunicano utilizzando indirizzi MAC mentre il traffico deve essere trasportato sulla rete fisica sottostante (underlay) utilizzando gli indirizzi IP dei VTEP.

Quando un VTEP riceve un frame Ethernet da un host locale (nella rete underlay del VTEP), consulta la tabella MAC-to-VTEP per trovare il VTEP remoto (indirizzo IP) corrispondente al MAC di destinazione. Il frame Ethernet viene incapsulato in un pacchetto VxLAN con destinazione IP impostata sull'indirizzo del VTEP remoto. Quando un VTEP riceve un pacchetto VxLAN dalla rete underlay, verifica il VNI e il MAC di destinazione contenuti nel payload e inoltra il frame Ethernet verso l'host locale corrispondente.

Ad esempio:

- Host A con MAC 00:11:22:33:44:55 è connesso al VTEP 1 (IP 10.1.1.1).
- Host B con MAC 66:77:88:99:AA:BB è connesso al VTEP 2 (IP 10.2.2.2).
- VTEP 1 invia un pacchetto a Host B:
 - VTEP 1 consulta la tabella MAC-to-VTEP, trova che 66:77:88:99:AA:BB è associato al VTEP 2 (10.2.2.2).
 - Incapsula il frame Ethernet in un pacchetto VxLAN con destinazione IP 10.2.2.2.
- VTEP 2 riceve il pacchetto VxLAN, decapsula il frame Ethernet e lo inoltra a Host B.

Forwarding dei pacchetti BUM

VXLAN supporta tre principali modalità per gestire il forwarding dei BUM packets, ciascuna con vantaggi e svantaggi a seconda dell'architettura della rete.

Nella modalità **Ingress Replication** il VTEP che riceve o genera un pacchetto BUM si occupa di replicare il pacchetto per ogni VTEP remoto che partecipa alla stessa VNI. Tale modalità è compatibile con qualsiasi rete underlay, tuttavia ogni VTEP deve gestire molteplici repliche, aumentando il consumo di risorse di CPU e banda.

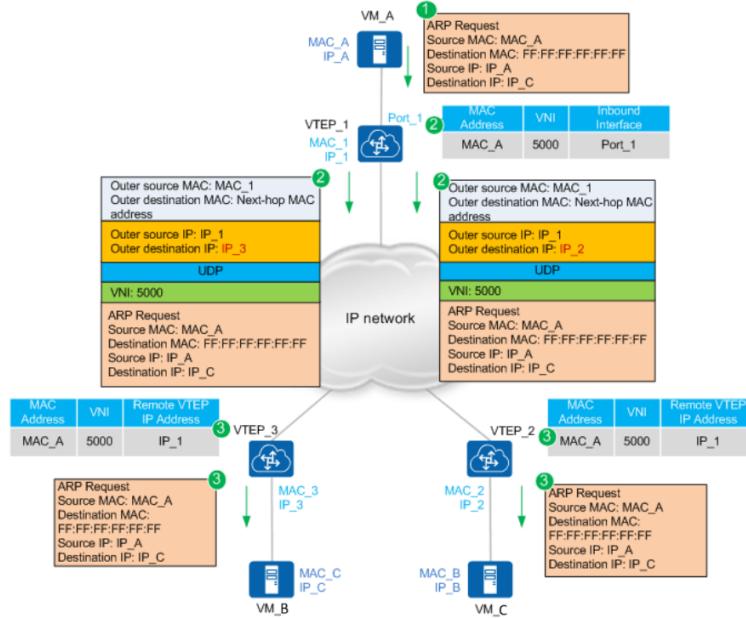


Figura 1.35: VxLAN ingress replication

Nella modalità **Multicast Replication** ogni VNI viene mappato a un gruppo multicast IP, e i pacchetti BUM vengono distribuiti tramite tale gruppo. Il VTEP sorgente invia un unico pacchetto VXLAN incapsulato all'indirizzo IP del gruppo multicast associato al VNI. I router multicast nella rete underlay distribuiscono il pacchetto ai VTEP che si sono uniti al gruppo. Risulta più efficiente in termini di banda della modalità ingress replication dato che non richiede la duplicazione dei pacchetti da parte del VTEP sorgente. Tuttavia c'è una dipendenza dall'infrastruttura underlay che deve supportare e gestire gruppi multicast.

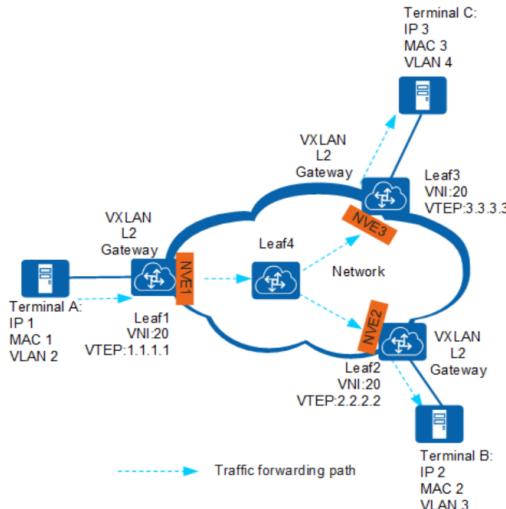


Figura 1.36: VxLAN multicast replication

Nella modalità **centralized Replication** si utilizza un nodo centrale per gestire il forwarding dei pacchetti BUM. Il VTEP sorgente invia il pacchetto al nodo centrale, che si occupa di replicarlo e distribuirlo agli altri VTEP. Ha come vantaggio il fatto che il VTEP sorgente non deve replicare direttamente i pacchetti tuttavia introduce un singolo punto di fallimento e il nodo centrale rappresenta un collo di bottiglia potenziale in caso di traffico BUM elevato.

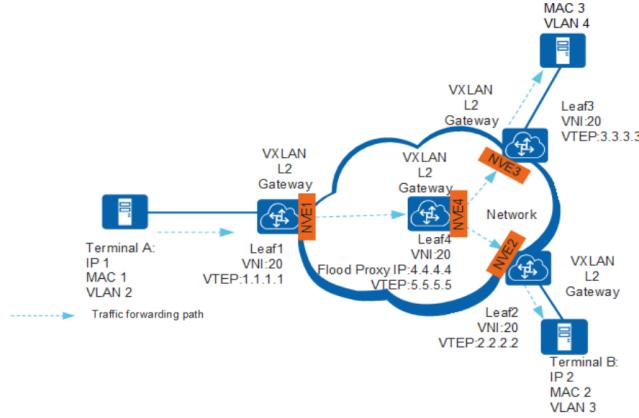


Figura 1.37: VxLAN centralized replication

Forwarding dei pacchetti known unicast

Il forwarding dei pacchetti known unicast in una rete VxLAN avviene quando l'indirizzo MAC di destinazione è già conosciuto dal VTEP sorgente, ovvero è presente nella tabella di mapping MAC-to-VTEP. Il VTEP sorgente consulta la sua tabella MAC-to-VTEP per trovare il VTEP remoto (indirizzo IP) associato all'indirizzo MAC di destinazione. Se il MAC è già presente, significa che il VTEP remoto è noto grazie a un apprendimento precedente (Flood and Learn) o a un control plane come EVPN.

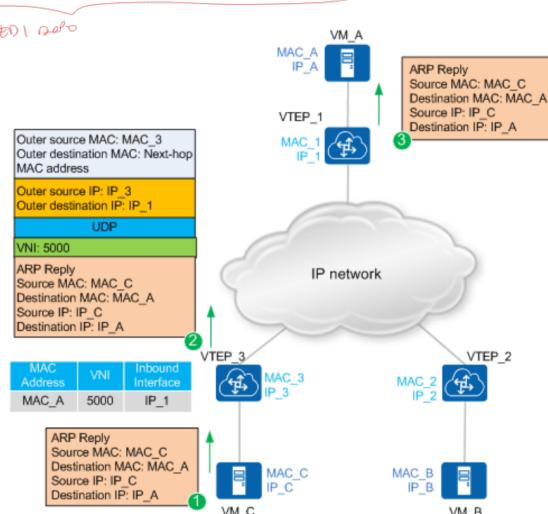


Figura 1.38: VxLAN known unicast forwarding

Costruzione della tabella MAC-to-VTEP

La costruzione della tabella MAC-to-VTEP può avvenire in due modalità:

- Flood and Learn:** il VTEP apprende il mapping MAC-to-VTEP tramite il flooding iniziale di un pacchetto BUM. Una volta che il MAC-to-VTEP mapping è noto, i pacchetti known unicast possono essere inoltrati direttamente.
- EVPN (Ethernet Virtual Private Network):** il control plane distribuisce in anticipo il mapping MAC-to-VTEP a tutti i VTEP che partecipano al VNI. Quando un host si connette a un VTEP, quest'ultimo apprende l'indirizzo MAC e/o IP dell'host. Il VTEP pubblica questa informazione agli altri VTEP attraverso BGP. Gli altri VTEP aggiornano le proprie tabelle di forwarding.

1.4.8 VNI inter-networking

Un **VxLAN Gateway** è un dispositivo che permette la comunicazione tra VNI diverse e tra dispositivi all'interno della VxLAN ed esterni alla VxLAN.

I VxLAN Gateway possono operare in due modalità:

- **Centralizzata:** i nodi leaf agiscono da VTEP mentre i nodi spine agiscono sia da VTEP che da gateway. Tale modalità permette una gestione semplice delle rotte inter-VNI, tuttavia non offre il path ottimale per i pacchetti. Inoltre i gateway potrebbero dover memorizzare tabelle ARP molto grandi.
- **Distribuita:** le funzioni di gateway sono distribuite tra i VTEP. Ogni nodo leaf agisce quindi sia da VTEP che da VxLAN Gateway. In questa modalità le rotte inter-VNI sono ottime tuttavia è necessario un piano di controllo (EVPN) per configurare i nodi leaf per i routing.

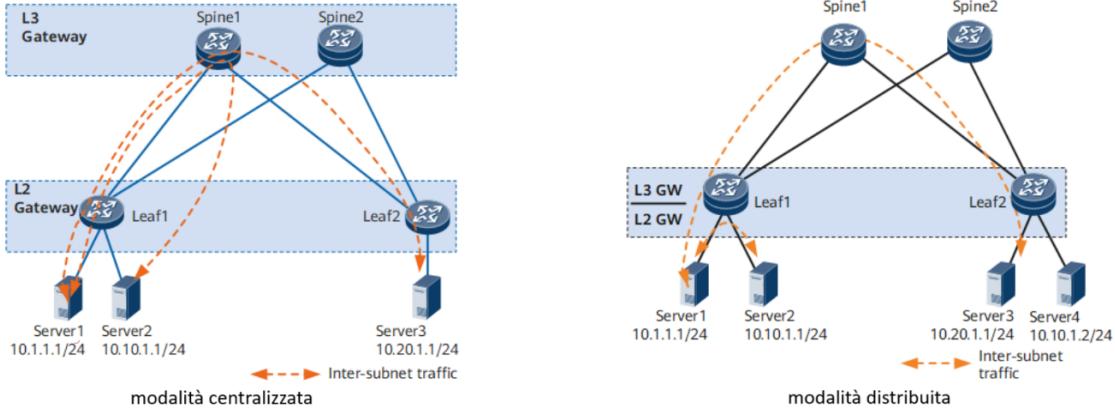


Figura 1.39: Modalità di routing centralizzata e distribuita

Nel caso di routing VxLAN è necessario utilizzare due tipologie di VNI:

- **L2VNI:** utilizzato per estendere reti di livello 2. (come VLAN, MPLS)
- **L3VNI:** usato per instradare traffico tra diversi segmenti di rete virtuale. Ad ogni L3VNI è associata una tabella di routing VRF (Virtual Routing and Forwarding). Quando un pacchetto appartiene a un determinato L3VNI, il dispositivo di rete usa la VRF associata al VNI per determinare come fare il routing.

Nel caso in cui sia il VTEP di ingresso che quello di uscita effettuano routing si parla di **Symmetric IRB**, nel caso in cui solo il VTEP di ingresso effettua il routing si parla di **Asymmetric IRB**. IRB (Integrated Routing and Bridging) è una funzionalità che consente di combinare routing e bridging in un'unica configurazione su un dispositivo di rete. In pratica, IRB permette a un dispositivo di instradare il traffico tra diverse subnet e bridgiare (estendere) segmenti di rete a livello di Layer 2, tutto in modo integrato.

L'IRB asimmetrico è più comune quando si utilizzano soluzioni in cui diversi VTEP sono specializzati in determinate operazioni (come routing o bridging), e quindi c'è un design che separa questi compiti tra i dispositivi. Questo approccio è utile per ottimizzare le risorse di rete e per separare in modo più chiaro le funzioni di bridging e routing, soprattutto in ambienti complessi multi-tenant.

SIMMETRIC: I VTEP SUL INGRESSO E USCITA FANNO ROTTINO, CIOE INDIFERENZA CHE SI CAMBIERA UNA O DUE, USCIRÀ DAL PRIMO VTEP TUTTO LA VIA DI PERCORRENZA LA LINEA CHE FERMAI ROTTINO VERSO LA VIA DI PERCORRENZA DEL VTEP IN USCITA CAMBIANDO DA NUOVO UNI TUTTOFINO QUELLA DI DESTINAZIONE

ASIMMETRIC: TUTTO LA UNI DA DESTINAZIONE GLI' DAL PRIMO VTEP

Capitolo 2

Cloud Computing

Il **cloud computing** è un modello di erogazione di servizi informatici che permette di accedere a risorse computazionali tramite internet. Invece di possedere infrastrutture proprie, le organizzazioni e gli individui possono utilizzare servizi forniti da terzi.

Il cloud computing offre numerosi vantaggi: scalabilità, costo, accessibilità, affidabilità, manutenzione. Presenta anche alcuni svantaggi: la scalabilità non è particolarmente utile per applicazioni con architettura monolitica, l'applicazione potrebbe essere difficile da adattare per il cloud, i dati potrebbero essere gestiti dal provider, la locazione fisica dei dati potrebbe non essere nota, problemi di responsabilità.

2.1 Modello di servizio

I modelli di servizio definiscono come le risorse e i servizi sono forniti agli utenti attraverso la rete.

I tre principali modelli sono:

- **IaaS** (Infrastructure as a Service): fornisce risorse di elaborazione di base come server virtuale, storage e reti. Gli utenti hanno controllo totale sui SO, sulle applicazioni e sulle configurazioni, ma non sull'infrastruttura sottostante (hardware fisico).
- **PaaS** (Platform as a Service): fornisce un ambiente di sviluppo e distribuzione di applicazioni con tutti i componenti necessari, come server di runtime, database e strumenti di sviluppo.
- **SaaS** (Software as a Service): fornisce applicazioni complete a cui gli utenti accedono tramite internet attraverso un browser web. Può essere di 4 livelli:
 1. single-tenant (custom): ogni client ha la propria istanza separata dell'applicazione e del database.
 2. single-tenant (configurable): ogni client ha la propria istanza dell'applicazione che però è configurabile tramite metadati.
 3. multi-tenant (configurabile): un'unica istanza dell'applicazione e del database serve più client, noti come tenant, ma i loro dati sono isolati.
 4. multi-tenant (scalabile): un'unica istanza dell'applicazione serve più tenant, ma è progettata per scalare automaticamente.

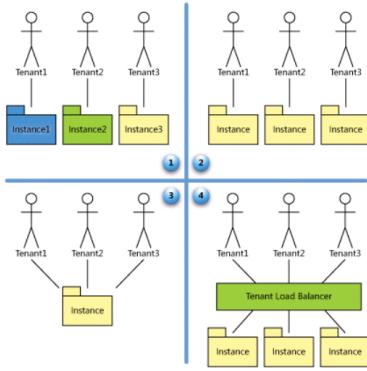


Figura 2.1: Livelli di SaaS

- **FaaS** (Function as a Service): consente agli sviluppatori di eseguire codice in risposta ad eventi senza dover gestire l'infrastruttura. Utilizzano un modello per pay-per-use in cui si paga solo il tempo di esecuzione delle funzioni.

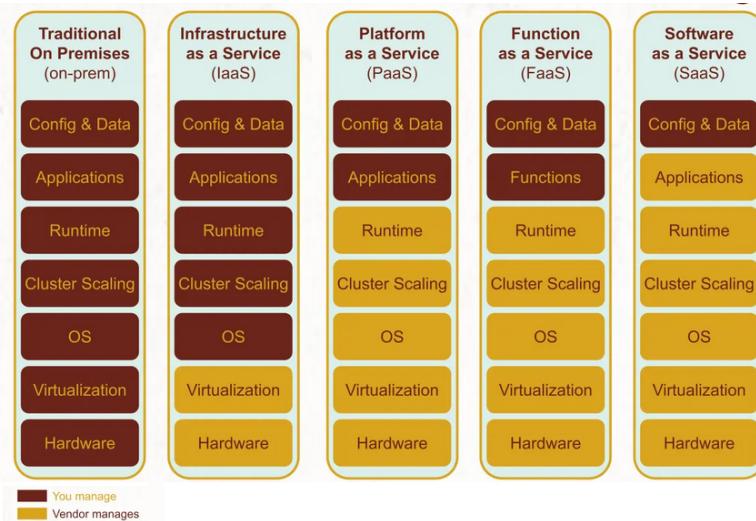


Figura 2.2: Modelli di servizio

2.2 Modello di deployment

I modelli di deployment definiscono come le risorse di infrastruttura sono gestite. Esistono 4 modelli di deployment:

- **Public Cloud**: le risorse sono di proprietà di un fornitore di terze parti che gestisce l'infrastruttura. Le risorse sono accessibili da qualsiasi luogo, c'è scalabilità, pay-per-use e condivisione delle risorse (multi-tenant).
- **Private Cloud**: risorse utilizzate da una singola organizzazione, può essere ospitato sia internamente all'azienda che da un provider terzo.
- **Hybrid Cloud**: combina l'infrastruttura del cloud pubblico e privato consentendo la portabilità dei dati e delle applicazioni tra i due ambienti. Le organizzazioni possono mantenere il controllo su risorse sensibili su un cloud privato e sfruttare i vantaggi del cloud pubblico per carichi di lavoro meno critici.
- **Community Cloud**: risorse cloud condivise da diverse organizzazioni con interessi comuni (requisiti normativi, standard di sicurezza, ...).

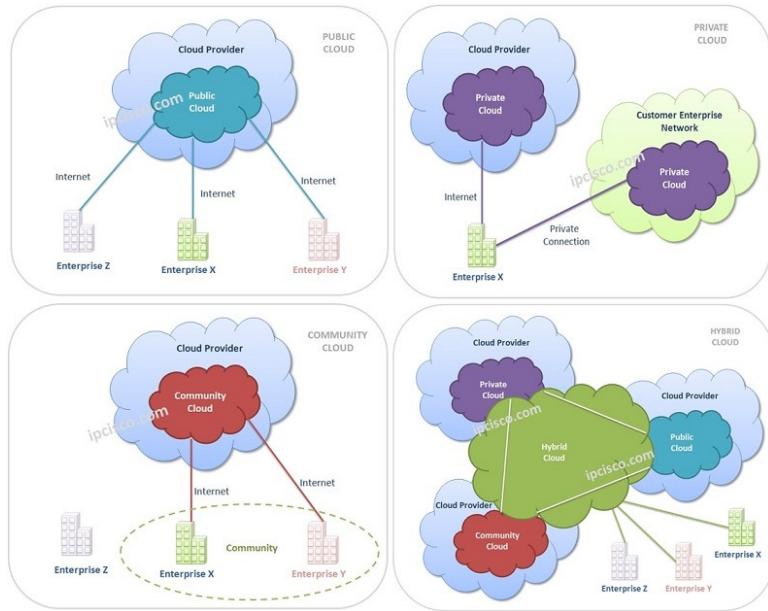


Figura 2.3: Modello di deployment

2.3 Sicurezza negli ambienti cloud

Gli ambienti cloud non sono immuni ai problemi di sicurezza “classici”: data loss, downtime, phishing, password cracking, botnet, malware,

Gli ambienti virtualizzati offrono due principali vantaggi:

- isolamento: un problema su una VM non influisce direttamente sulle altre;
- snapshot: copia dell'intero stato di una VM che consente il suo ripristino;

presenta però anche alcuni svantaggi:

- state restore: ripristinare una VM pò reintrodurre vulnerabilità e problemi già risolti;
- larger attack surface: l'aumento di VM aumenta il numero di punti di accesso vulnerabili;
- accountability: difficoltà nel determinare chi è responsabile della sicurezza di specifiche risorse o dati;
- new side channels: ad esempio si potrebbe sfruttare la vulnerabilità in una VM per attaccarne un'altra;
- lack of auditability: accesso limitato ai log;
- regulatory compliance: normative e standard da rispettare;

2.4 Shared responsibility model

Il **shared responsibility model** definisce le responsabilità di sicurezza del provider cloud e del cliente cloud. In generale, il fornitore di cloud è responsabile della sicurezza dell'infrastruttura sottostante che affitta ai suoi clienti, mentre il cliente è responsabile della sicurezza delle ariee dell'infrastruttura cloud su cui ha il controllo.

L'esatta suddivisione delle responsabilità varia a seconda del tipo di servizio cloud offerto:

- IaaS:
 - cloud: sicurezza fisica dei data center;
 - client: SO, patch, runtime;
- PaaS:
 - cloud: infrastruttura, SO, runtime;
 - client: app, dati, accessi;

- SaaS:
 - cloud: infrastruttura, app;
 - client: dati, accesso;

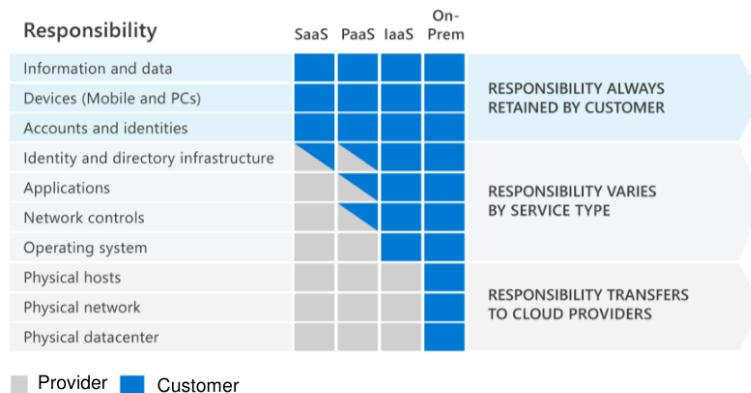


Figura 2.4: Shared responsibility model

2.5 Politica Zero Trust

Nei sistemi cloud il perimetro fisico di sicurezza non esiste. È quindi necessario utilizzare delle politiche di sicurezza basate sull'identità. Tramite l'autenticazione è possibile verificare l'identità dell'utente e tramite l'autorizzazione è possibile determinare a quali risorse l'utente può accedere. Il modello zero trust presuppone che non ci sia un perimetro di sicurezza affidabile e quindi richiede che ogni accesso, interno o esterno, sia autenticato e autorizzato.

I principi fondamentali del modello zero trust sono:

- verifica esplicita: ogni richiesta di accesso deve essere autenticata e autorizzata.
- accesso con privilegi minimi: gli utenti dovrebbero avere solo i privilegi necessari per poter eseguire il task.
- assume breach: si assume che una violazione sia già avvenuta o potrebbe avvenire.

La difesa nel cloud usa un approccio a strati in cui ogni strato offre un meccanismo di sicurezza. Ci sono 6 strati: fisico, identità, perimetro, network, compute, application, data.

2.6 Design Pattern

Nello sviluppo di applicazioni cloud-native invece di concepire applicazioni monolitiche, esse vengono scomposte in servizi decentralizzati che comunicano tra loro tramite API, gestione degli eventi e messaggi asincroni. Le applicazioni possono così scalare aggiungendo nuove istanze in base alle necessità.

I design pattern vengono usati per creare applicazioni cloud affidabili, scalabili e sicure.

2.6.1 DP per la gestione dei dati

Per motivi di performance, scalabilità e disponibilità i dati devono essere hostati su più server, anche in posizioni diverse. È quindi necessario provvedere meccanismi per mantenerne la consistenza. Inoltre, è necessario proteggere i dati memorizzati e in transito per assicurare confidenzialità, integrità e disponibilità.

Cache-Aside

Le operazioni di lettura possono essere lente se effettuate su un DB. Nel DP Cache-aside i dati vengono memorizzati in una cache e il servizio li recupera da lì se presenti, altrimenti li carica dal DB e aggiorna la cache.

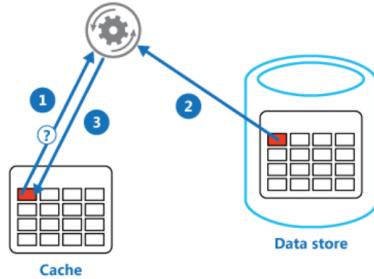


Figura 2.5: DP Cache-Aside

CQRS (Command and Query Responsability Segregation)

Nel caso in cui il carico di lavoro di lettura e scrittura sono asimmetrici è conveniente utilizzare CQRS. Tale pattern separa i modelli di lettura e scrittura utilizzando due interfacce distinte per gestire le operazioni di comando (scrittura) e di query (lettura). In questo modo è possibile utilizzare un modello che velocizza le operazioni di lettura.

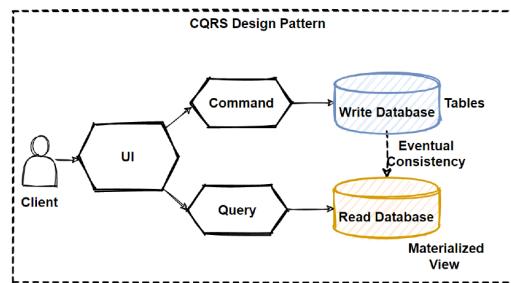


Figura 2.6: DP CQRS

Event Sourcing

Nel caso in cui è necessario una storicizzazione dello stato dell'applicazione è possibile utilizzare il pattern Event Sourcing. Tale pattern memorizza tutte le modifiche allo stato come una sequenza di eventi anziché lo stato corrente dell'applicazione. L'applicazione ricostruisce lo stato a partire da questi eventi. È spesso utilizzato insieme al CQRS, in questo caso il command model gestisce le richieste di modifica dello stato e genera eventi, mentre il query model genera delle view interrogabili a partire dalla sequenza di eventi.

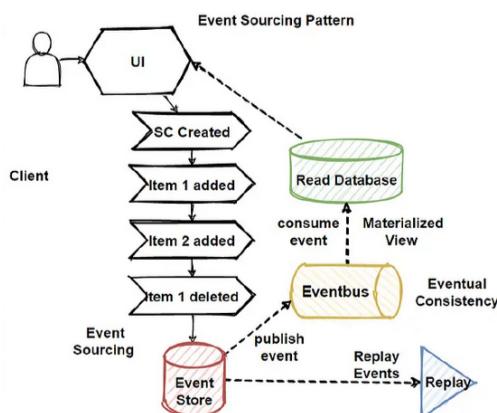


Figura 2.7: DP CQRS

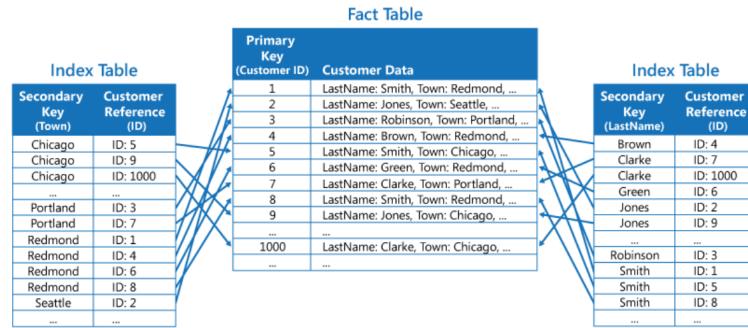
Index Table

Le query su grandi volumi possono essere lente e poco efficienti. Il DP Index Table prevede la creazione di tabelle con indici secondari su cui si effettuano le query, possono essere di 3 tipi:

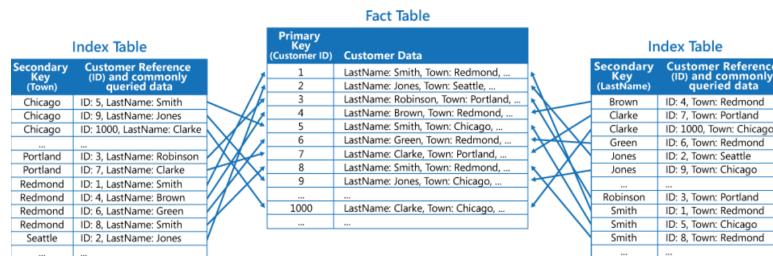
- duplicazione di tutti i dati con una chiave diversa;

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, ...
...	

- tabella contenente l'associazione tra la nuova chiave e la chiave primaria;



- combina i primi due approcci duplicando parzialmente i dati più comunemente usati e facendo uso di chiavi secondarie;



MaterIALIZED View

Il formato dei dati del database potrebbe non essere particolarmente adatto per alcune query, oppure alcune query che generano dati aggregati potrebbero essere ripetute inutilmente. Il DP Materialized View precalcola e memorizza i risultati di una query in una view che potrà poi essere utilizzata per effettuare delle query più semplici.

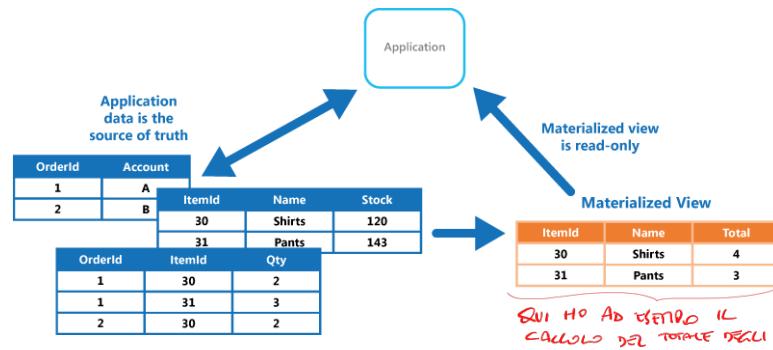
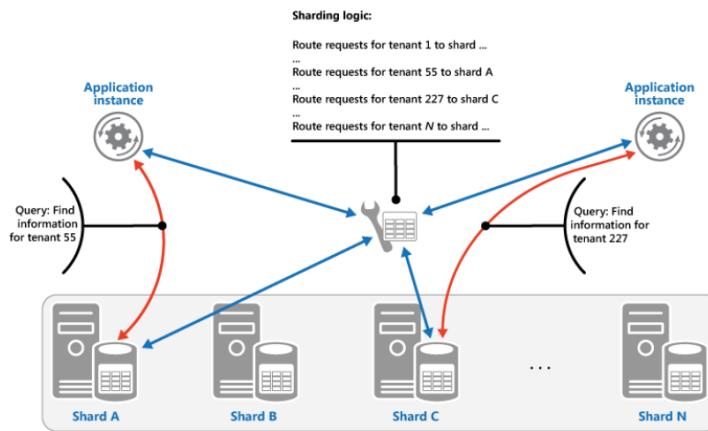


Figura 2.8: Materialized View

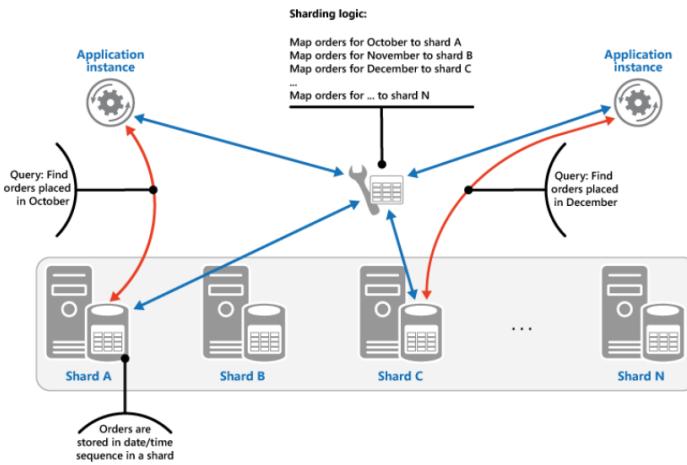
Sharding

DB di grandi dimensioni sono un collo di bottiglia a causa di limitazioni di scalabilità. Lo Sharding pattern divide il DB in più parti (shard) distribuite su server diversi. Si possono usare 3 strategie di sharding:

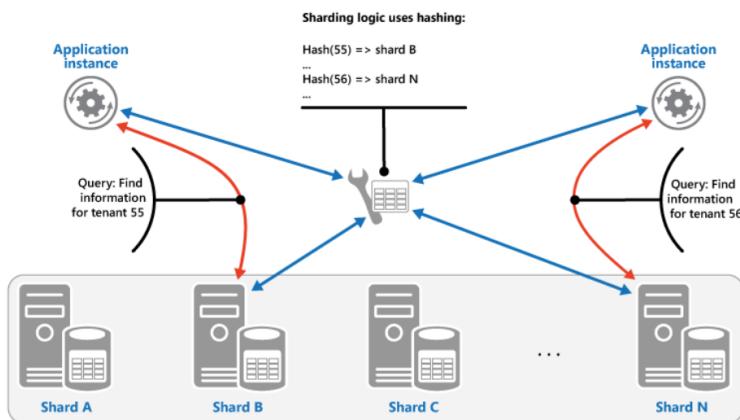
- **Lookup strategy:** i dati di un tenant vengono memorizzati in un unico shard e si effettua un'associazione tra tenant e shard, usando quindi il tenant come shard key.



- **Range strategy:** raggruppa dati legati nello stesso shard e usa una shard key sequenziale per recuperare i dati.



- **Hash strategy:** distribuisce i dati sugli shard in base al digest ottenuto su un particolare attributo dei dati, in questo modo si evita di sovraccaricare alcuni shard contenenti dati molto richiesti.



Static content hosting

Le risorse statiche come immagini e video possono gravare su server applicativi. Il DP Static Content Hosting prevede l'utilizzo di servizi di hosting per distribuire il contenuto statico.

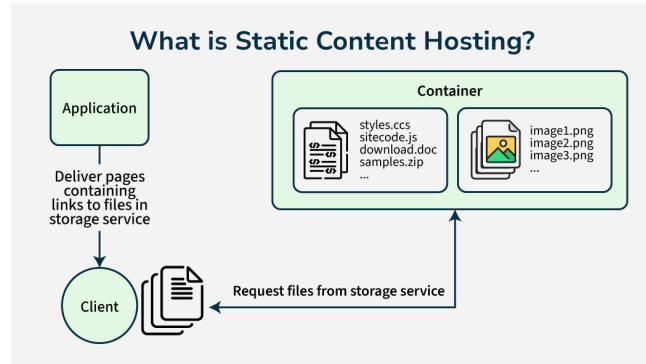


Figura 2.9: Static content hosting

Valet key

Si vuole controllare l'accesso ad un data store nel caso in cui il data store non supporta l'autenticazione e l'autorizzazione. Il DP Valet Key risolve questo problema facendo uso di una chiave temporanea (valet key) per accedere alla risorse privata.

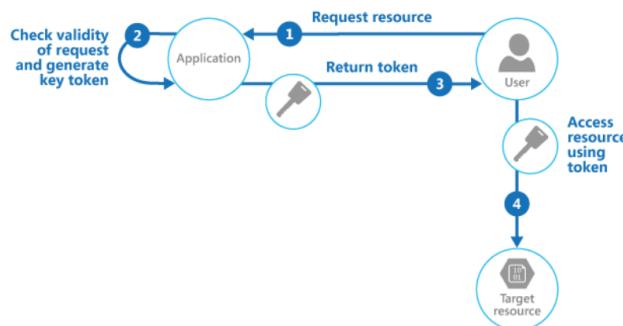


Figura 2.10: Valet Key

2.6.2 DP implementativi

Ambassador

Nel caso in cui si deve gestire la comunicazione tra un'applicazione e un servizio remoto si usa il DP Ambassador. L'Ambassador funge da intermediario (proxy) tra l'applicazione client e il servizio remoto. Permette di implementare politiche di gestione delle richieste avanzate come: retry, caching, o circuit breaker (interrompe temporaneamente le richieste a un servizio remoto quando rileva troppi fallimenti consecutivi per prevenire il sovraccarico del servizio).

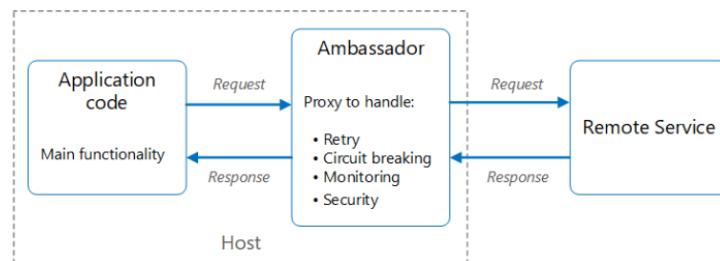


Figura 2.11: Ambassador

Gateway Aggregation

Nel caso in cui un'applicazione necessita di effettuare richieste che coinvolgono l'accesso a dati da più servizi conviene utilizzare il DP Gateway Aggregation. Il gateway si occupa di ricevere la richiesta, contattare i

vari servizi e fornire una risposta unificata. Ciò permette di risparmiare lo scambio di messaggi da parte dell'applicazione.



Figura 2.12: Gateway Aggregation

Gateway Routing

Nel caso in cui si vogliono esporre più servizi su un unico endpoint, fare load balancing tra più istanze dello stesso servizio o avere più versioni dello stesso servizio su un singolo endpoint si usa il DP Gateway Routing. Il gateway si occupa di instradare le richieste verso il servizio appropriato.

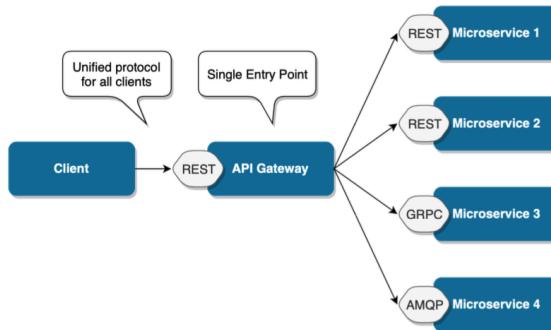


Figura 2.13: Gateway Routing

2.6.3 DP per l'affidabilità

Queue-based Load Leveling

Nel caso in cui picchi di traffico sovraccaricano un servizio è possibile utilizzare il DP Queue-Based Load Leveling. Esso prevede l'utilizzo di code che permettono ai servizi di processare le richieste a un ritmo costante. Tale tecnica può aumentare la latenza nelle risposte.

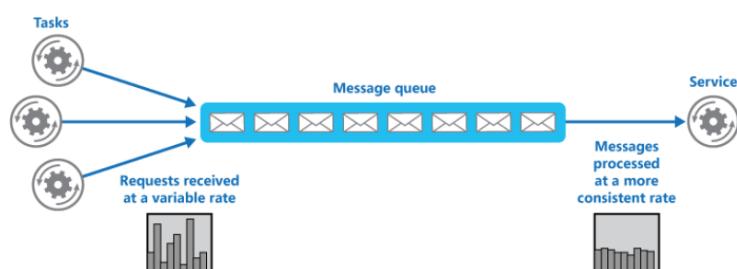


Figura 2.14: Queue-based Load Leveling

Throttling

Nel caso in cui alcuni utenti effettuano un numero di richieste eccessive che sovraccaricano il sistema è possibile limitare tale richieste tramite il DP Throttling. Il Throttling viene spesso combinato con l'autoscaling al fine di

limitare temporaneamente le richieste effettuabili da un utente in un arco temporale, ciò permette all'autoscaling di scalare per poter soddisfare le richieste.

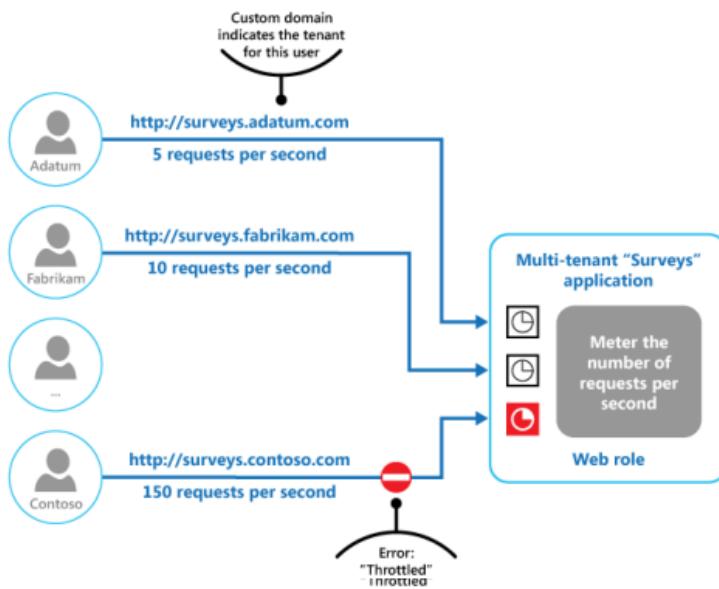


Figura 2.15: Throttling

2.6.4 DP per la disponibilità

Bulkhead

Al fine di evitare che un guasto in una parte del sistema influisca sull'intero sistema si isolano le parti del sistema in compartimenti separati (bulkheads) in modo che continuino a funzionare nonostante il guasto.

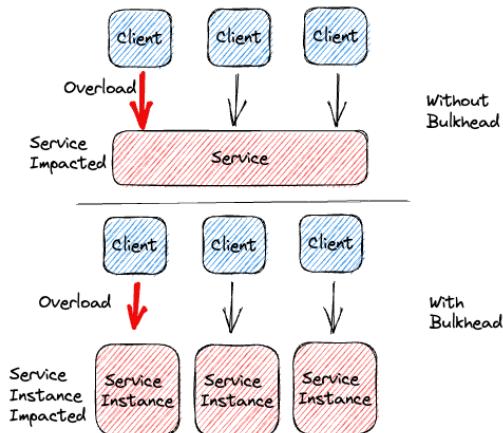


Figura 2.16: Bulkhead

Circuit Breaker

Il DP Circuit Breaker consente di individuare guasti in un servizio e permette di non far bloccare il sistema in seguito al guasto. Per fare ciò viene usato un proxy che agisce da tramite per le richieste al servizio ed implementa una macchina a stati a 3 stati:

- **Closed:** le richieste vengono inoltrate al servizio e si mantiene un counter con il numero di fallimenti recenti. Se il counter supera un soglia si passa nello stato Open e viene avviato un timer, alla scadenza di tale timer si passa nello stato di Half-open.
- **Open:** la richiesta non viene inoltrata al servizio e viene inviato un messaggio di errore al client.

- **Half-open:** solo un numero limitato di richieste vengono inoltrate al servizio e se tali richieste vanno a buon fine si torna nello stato Closed altrimenti si torna nello stato Open.

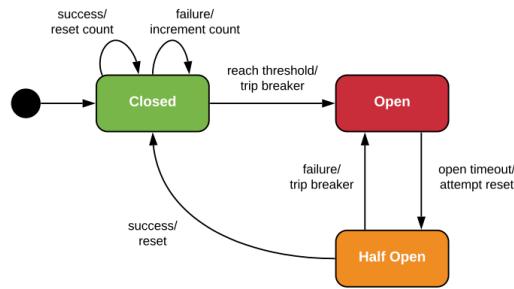


Figura 2.17: Circuit Breaker

2.6.5 DP per la messaggistica

Choerography

Utilizzare un orchestratore centrale che smista i messaggi verso i servizi può risultare difficile da mantenere nel caso in cui i servizi vengono rimossi e aggiunti continuamente. Il DP Coerography prevede un sistema di messaggistica asincrona che consiste nell'accumulare le richieste in una coda e farle gestire ai servizi che sono interessate ad esse. Le risposte vengono poi inviate alla stessa coda o ad una coda diversa.

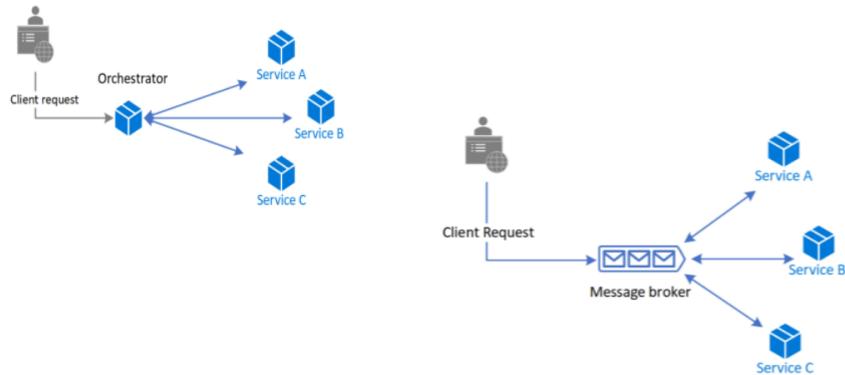


Figura 2.18: Coerography

Publisher-Subscriber

Nel DP Publisher-Subsciber i servizi che effettuano richieste inviano messaggi su un topic. I servizi che consumano le richieste si iscrivono ai topic di interesse ed elaborano le richieste. Il tutto è gestito da un intermediario chiamato broker (ad esempio RabbitMQ).

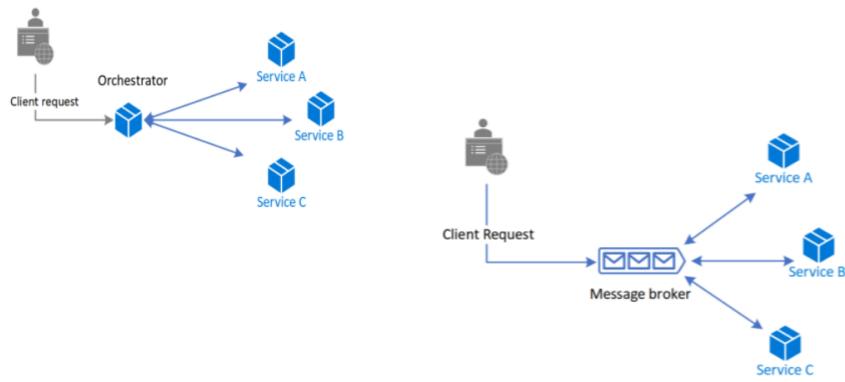


Figura 2.19: Publisher Subscriber

Priority Queue

Nel caso in cui alcuni messaggi devono essere elaborati prima di altri si utilizza il DP Priority Queue. Tale DP prevede l'uso di code di priorità, ovvero code i cui elementi sono ordinati in base ad un livello di priorità.

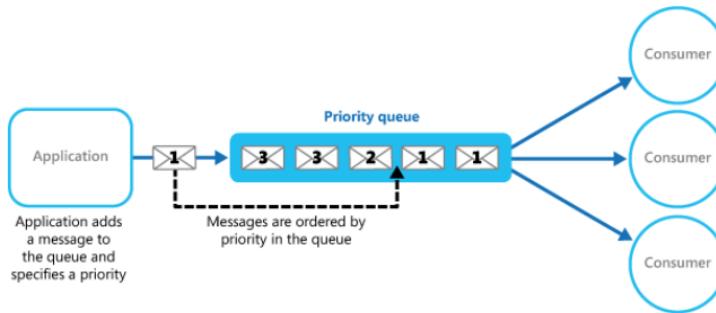


Figura 2.20: Priority Queue

2.7 OpenStack

OpenStack è una piattaforma open-source per la gestione di infrastrutture cloud. Fornisce strumenti per costruire e gestire ambienti cloud pubblici e privati. Le sua architettura è modulare, composta da vari componenti che lavorano insieme per offrire servizi come: Compute (Nova), Storage (Cinder e Swift), Networking (Neutron), Identity (Keystone), Dashboard (Horizon), Orchestration (Heat).

2.7.1 Architettura

Ogni servizio di OpenStack opera in modo indipendente, ma può interagire con gli altri servizi tramite API pubbliche. Ciasun servizio di OpenStack è costituito da diversi processi, e in ogni servizio è presente un API Service, il cui compito è ascoltare le richieste API, preprocessarle, e inoltrare agli altri processi interni dal servizio stesso.

OpenStack adotta un'architettura MOM (Message Oriented Middleware) per facilitare la comunicazione tra i vari servizi. Tale architettura consente ai componenti di essere debolmente accoppiati. I messaggi tra i servizi vengono gestiti tramite una coda con il protocollo AMQP (Advanced Message Queueing Protocol). OpenStack utilizza RabbitMQ come broker di messaggi, che implementa AMPQ, e che gestisce la ricezione, la memorizzazione temporanea e la distribuzione dei messaggi tra i diversi componenti, facilitando la gestione delle operazioni asincrone e consentendo l'elaborazione distribuita.

Essendo OpenStack progettato con un'architettura debolmente accoppiata è possibile aggiornare, scalare e sostituire le componenti del sistema senza influire sugli altri.

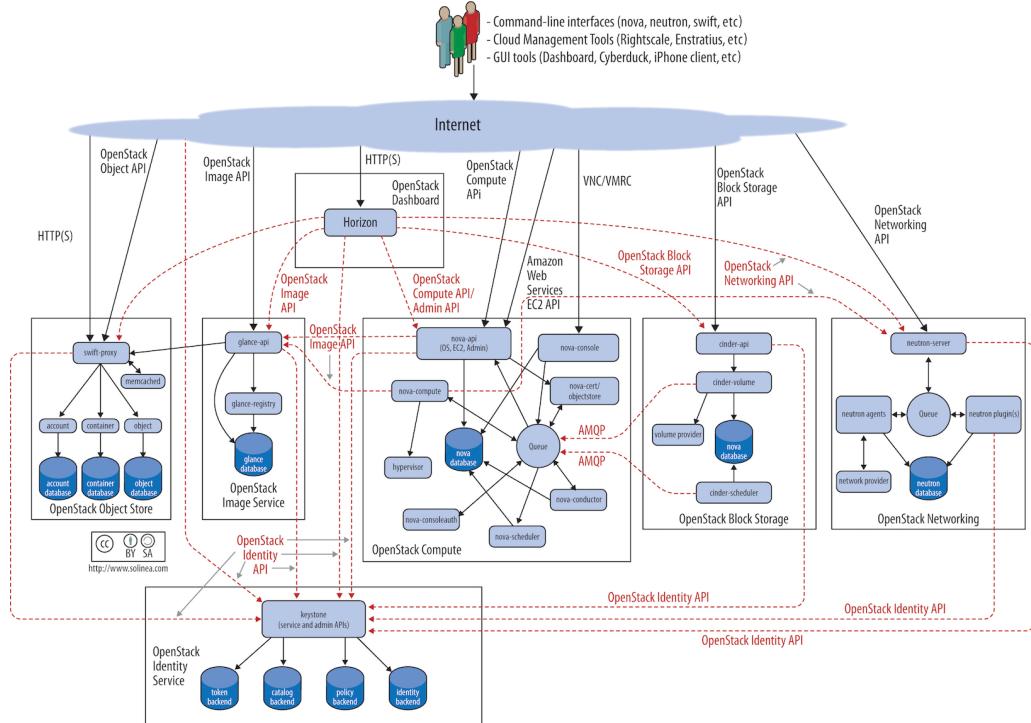


Figura 2.21: Architettura OpenStack

OpenStack offre alcuni servizi base:

- **Horizon** (dashboard): portale web per interagire con i servizi OpenStack.
- **Nova** (compute): gestisce il ciclo di vita delle istanze computazionali.
- **Neutron** (networking): fornisce la connettività per i servizi e una API che consente agli utenti di definire reti.

alcuni servizi condivisi:

- **Keystone** (identità): fornisce un servizio di autenticazione e autorizzazione per i servizi OpenStack.
- **Glance** (image): fornisce immagini di VM usate da Compute.
- **Ceilometer** (telemetry): monitors e fornisce metriche su OpenStack.

servizi di storage:

- **Swift** (object storage): permette di archiviare file senza l'utilizzo di un file system tradizionale e di recuperarli tramite una API REST.
- **Cinder** (block storage): fornisce volumi di storage a blocchi, che possono essere montati come unità disco virtuali dai sistemi operativi.

servizi di alto livello:

- **Heat** (orchestrazione)

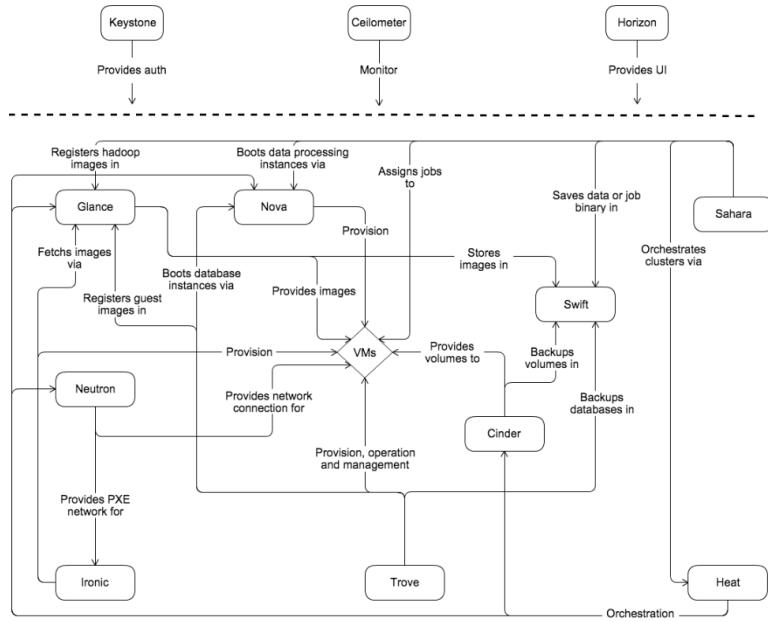


Figura 2.22: Interazione tra le componenti OpenStack

Un tipico sistema OpenStack è composta da:

- **Nodo Controller:** esegue i servizi di Identity e Image, la parte gestionale di Compute e di Networking e la Dashboard. Presenta servizi di supporto come SQL DB, message queue e NTP (Network Time Protocol). Può eseguire parti di servizi opzionali come Block Storage, Object Storage, Orchestration e Telemetry. Necessita di almeno 2 NIC (rete management e provider).
- **Nodo Compute:** esegue la parte di Compute dedicata all'hypervisor (KVM). È possibile avere più nodi compute e ognuno di essi necessita di almeno 2 NIC (rete management e provider).
- **Nodo Block Storage (opzionale):** contiene i dischi usati dal servizio Block Storage. È possibile avere più nodi Block Storage e ognuno di essi necessita di almeno 1 NIC (rete management).
- **Nodo Object Storage (opzionale):** contiene i dischi usati dal servizio Object Storage. È possibile avere più nodi Block Storage e ognuno di essi necessita di almeno 1 NIC (rete management).

2.7.2 Networking

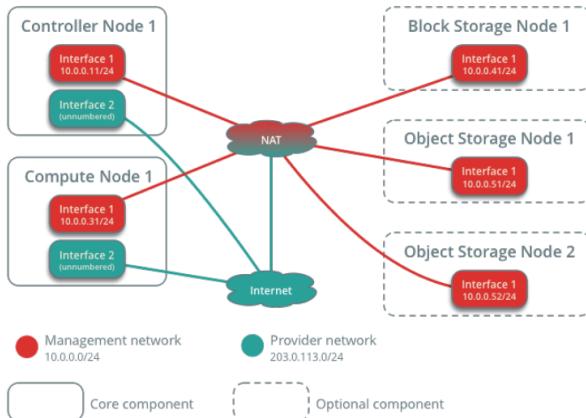


Figura 2.23: OpenStack Network

Ogni nodo richiede l'accesso a Internet per scopi amministrativi come l'installazione di pacchetti, aggiornamenti di sicurezza, DNS e NTP. Ogni nodo presenterà quindi una NIC connessa alla rete management. Spesso tale rete è una rete privata, e quindi per consentire l'accesso ad Internet si utilizza il natting.

Ogni istanza (macchina virtuale) presente sui nodi di calcolo è connessa ad una rete che consente ai tenant di comunicare con l'esterno e può essere configurata in due modi:

- Provider Network: le istanze OpenStack sono collegate direttamente alla rete fisica e il traffico viene gestito tramite switch/bridge. Si utilizzano le VLAN per la segmentazione del traffico e il routing è gestito dalla struttura fisica e non da OpenStack. Questa soluzione si usa nel caso di cloud semplici dove è presente una infrastruttura di rete fisica solida.
- Self-Service Network: la rete viene segmentata usando VxLAN, che consente l'isolamento tra reti virtuali e fisiche. Questa soluzione si usa in ambienti multi-tenant complessi e cloud su larga scala.

2.7.3 Zone gerarchiche

OpenStack è stato progettato per poter scalare anche fino a migliaia di nodi dislocati anche in datacenter e regioni geograficamente diverse. È quindi possibile dividere il cloud OpenStack in 3 zone gerarchiche:

- Regions: ogni regione ha il suo deployment OpenStack completo che include un API endpoint, la rete e le risorse di calcolo. Regioni diverse condividono i servizi Keystone e Horizon.
- Availability Zone (AZ): raggruppamento logico di nodi compute all'interno di una regione, che consente l'isolamento degli errori confinandoli all'interno della zona. Quando si lancia una nuova VM è possibile specificare in quale zona effettuare il deployment.
- Host Aggregates: raggruppamento logico di nodi compute basato su caratteristiche hardware specifico degli host (ad esempio SSD/HDD). È possibile effettuare aggregati anche fra AZ diverse.

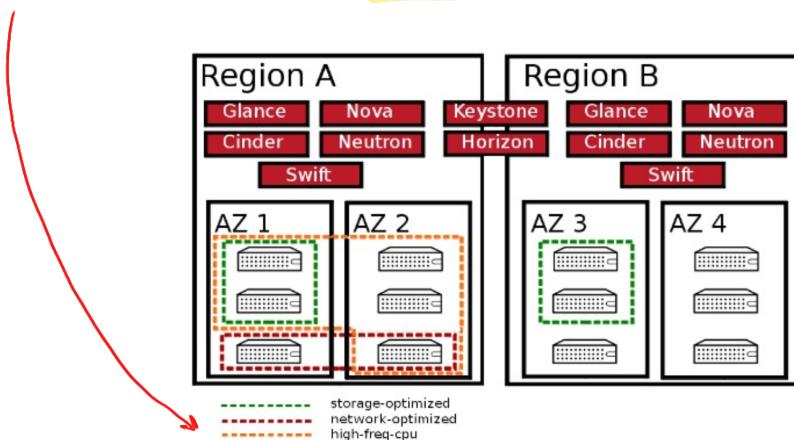


Figura 2.24: Zone gerarchiche

2.7.4 Servizi

Keystone

Keystone viene eseguito nel controller ed è organizzato come un insieme di servizi che cooperano per gestire l'autenticazione, l'autorizzazione e la gestione delle risorse.

Keystone è composto dai seguenti servizi interni:

- Identity Service: si occupa di validare le credenziali e di fornire dati sugli utenti.
 - Utenti: API consumer individuali, il loro nome è univoco all'interno del dominio ma non globalmente.
 - Gruppi: insiem; di utenti in un dominio, il loro nome è univoco all'interno del dominio ma non globalmente.

È il primo servizio con cui un utente interagisce. Una volta autenticato l'utente può accedere agli altri servizi. Ogni servizio OpenStack necessita di almeno una entry con il corrispettivo endpoint archiviato nell'Identity Service. Gli endpoint possono essere di 3 tipi:

- admin: permettono di effettuare modifiche agli utenti e ai tenant;
- internal: usato internamente per la comunicazione tra servizi;
- public: usato dagli utenti esterni;

L'Identity Service è composto dalle seguenti componenti:

- **Server:** fornisce autenticazione e autorizzazione tramite un interfaccia REST;
- **Drivers:** integrata nel server e permette di accedere alle informazioni sulle identità in repository esterne a OpenStack;
- **Modules:** intercetta le richieste, estrae le credenziali e le invia al server, agisce quindi da middleware;

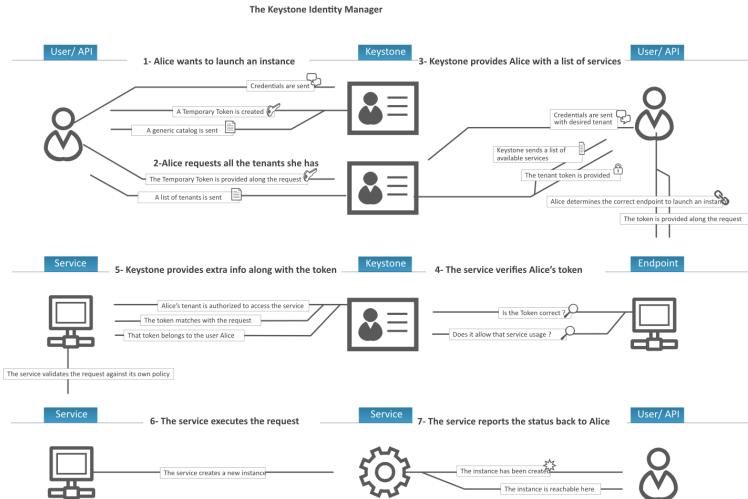


Figura 2.25: Identity Service

- **Resource Service:** fornisce dati sui progetti e sui domini. I progetti (tenants) raggruppano utenti, risorse e politiche di accesso, permettendo l'isolamento e la gestione autonoma delle risorse all'interno di un ambiente cloud. I domini sono l'unità organizzativa di livello più alto e contengono progetti, gruppi e utenti. I nomi dei domini devono essere univoci, quelli dei tenant sono univoci solo all'interno dei domini.
- **Assignment Service:** fornisce una mappatura tra utenti o gruppi e ruoli a livello di dominio e progetto. Un ruolo definisce quali azioni possono essere effettuate, OpenStack fornisce alcuni ruoli di default:

- **reader:** accesso read-only alle risorse;
- **admin:** controllo completo delle risorse;
- **member:** accesso intermedio alle risorse, dipende dal particolare servizio;

Un **Role Assignment** è una 3-tupla contenente:

- **Role:** definisce i permessi di accesso alle risorse (ad esempio admin o member).
- **Resource:** definisce il raggio di azione del ruolo (ad esempio un particolare dominio o tenant).
- **Identity:** definisce l'utente o il gruppo a cui è assegnato il ruolo.

- **Token Service:** gestisce i token che rappresentano sessioni di autenticazione. Un token è un oggetto temporaneo che un utente può usare per autenticarsi nei servizi senza dover fornire nuovamente le credenziali. Si usano diverse tipologie di token:

- **Token UUID** (Universally Unique Identifier): l'utente invia username/password a keystone che risponde con un token di 128 bit, il token viene poi inviato dall'utente insieme alle richieste dei servizi in modo che i servizi possano autenticare gli utenti. Questa modalità genera molto traffico tra i servizi e keystone, i servizi infatti necessitano di validare il token tramite keystone e di ricevere le informazioni di autenticazione assegnate al token.

I SERVIZI
NON SAVONO COSA DICONTE
CON QUESTI TOKEN

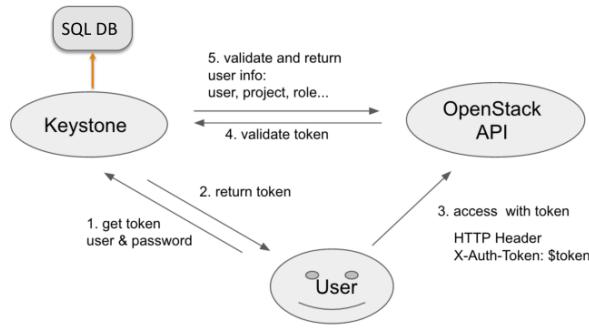


Figura 2.26: Token UUID

- **Token PKI:** il token restituito da keystone all'utente contiene già il catalogo dei servizi a cui l'utente può accedere. Il token viene firmato da keystone con una chiave privata e validato dal servizio richiesto con una chiave pubblica. Non è quindi necessario il passaggio di verifica tra servizio richiesto e keystone, tuttavia i token sono molto più grandi dato che contengono le informazioni sugli utenti (può arrivare anche a 8 KB). PKIZ utilizza la compressione ZIP per i token.

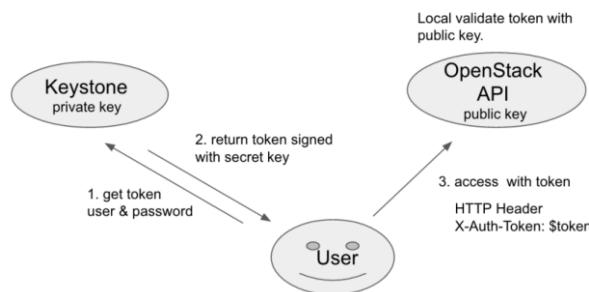


Figura 2.27: Token PKI

- **Token Fernet:** utilizza una chiave simmetrica (stessa chiave per cifrare e decifrare) e token effimeri (hanno durata limitata). Utilizza un formato serializzato binario per rappresentare le informazioni utente, critta tali informazioni con la chiave privata e genera una firma sempre con la stessa chiave privata. In questo caso, come nel caso UUID, il servizio richiesto e keystone si scambiano il token che viene decifrato da keystone e restituito al servizio.

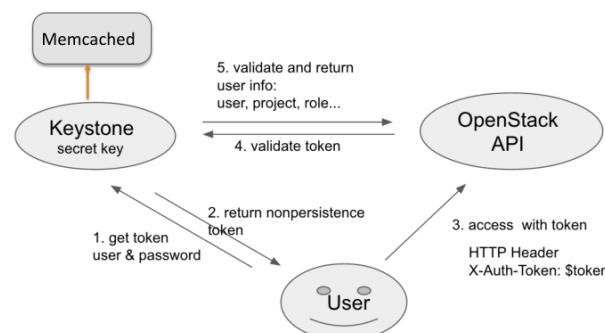


Figura 2.28: Token Fernet

Il vantaggio rispetto a UUID è che il DB di token keystone non è persistente e può essere in-memory (le informazioni sono contenute nel token), il vantaggio rispetto a PKI è che i token sono molto più piccoli grazie alla serializzazione. Al fine di garantire la sicurezza Fernet cambia periodicamente la chiave, per fare ciò mantiene un repository (memcached) contenente:

* primary key: unica e viene usata per cifrare e decifrare il token;

- * secondary key: non necessariamente unica, si tratta di primary key scadute e vengono usate per decifrare token cifrati con vecchie primary key;
- * staged key: unica e rappresenta la prossima primary key;

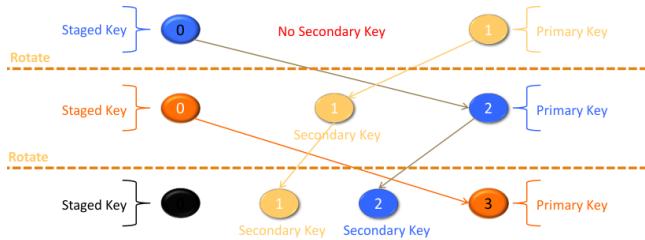


Figura 2.29: Rotazione delle chiavi Fernet

- Token JWS: utilizza una cifratura asimmetrica per firmare i token che vengono scambiati in chiaro. I token sono formati da: intestazione, dati, firma e possono essere inseriti all'interno di un header HTTP. Risolve il problema di Fernet della condivisione della chiave nel caso di keystone multinodo. La differenza fondamentale con il token PKI è che usa un formato basato su JSON e stringhe codificate in Base64 il che consente una compattezza maggiore.

- Catalog Service: mantiene e fornisce un catalogo dei servizi disponibili e dei relativi endpoint.
- Policy Service: gestisce le regole di autorizzazione degli utenti.

Glance

Glance è un servizio di gestione delle immagini che permette la scoperta, la registrazione e il recupero di immagini che possono essere utilizzate per avviare istanze di macchine virtuali.

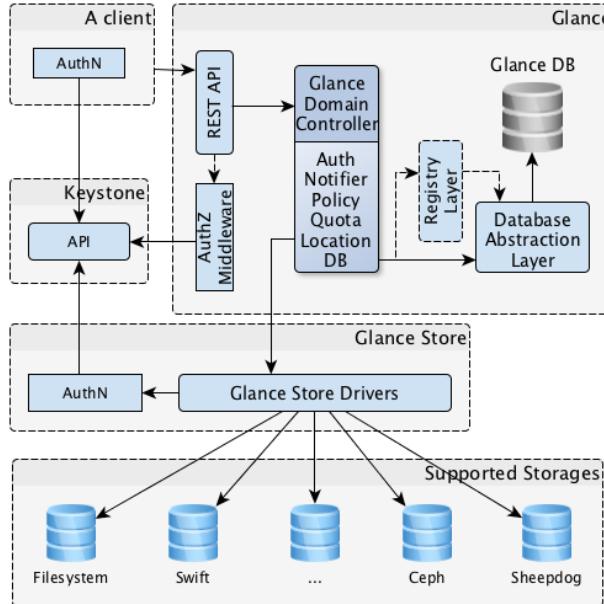


Figura 2.30: Architettura Glance

Glance presenta le seguenti componenti:

- Client: interfaccia CLI per interagire con l'API REST.
- API REST: permette di interagire con Glance. È possibile caricare, recuperare, aggiornare ed eliminare immagini.
- Database Abstraction Layer: permette di interagire con il DB dove sono memorizzati i metadati delle immagini in modo agnostico rispetto alla tecnologia DB utilizzata.

- **Glance Domain Controller:** orchestra le operazioni interne al server Glance, che sono: authorization, notification, policies, database connections.
- **Glance Store:** gestisce l'archiviazione effettiva delle immagini. Supporta vari backend (file system locale, Object Storage, Amazon S3, ...).

Le immagini possono essere salvate in vari formati disk (raw, vdi, iso, ...) e vari formati container (bare, ova, docker, ...).

Le immagini possono trovarsi in vari stati: queued, saving, active, killed,

La visibilità delle immagini può essere: public, community, shared, private.

Nova

Nova è il servizio che consente di avviare, gestire e monitorare le istanze delle VM. Si occupa della gestione del ciclo di vita delle VM, della schedulazione delle risorse e dell'interfacciamento con l'hypervisor.

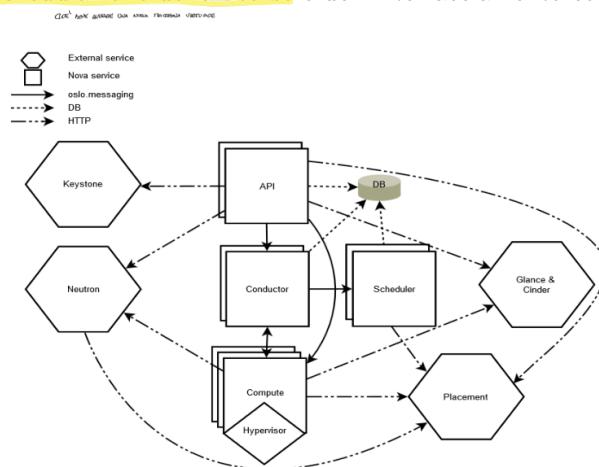


Figura 2.31: Componenti Nova

Nova presenta le seguenti componenti:

- **Database:** DB SQL contenente le informazioni di stato e configurazione relative alle istanze, ai nodi e alle risorse.
- **API:** riceve richieste HTTP, le converte in comandi e comunica con le altre componenti tramite la coda oslo.messaging o HTTP.
- **Scheduler:** decide su quale nodo avviare una nuova istanza. Per decidere il nodo si applica un filtro in base a criteri specificati nella richiesta di creazione (ad esempio minRAM). Una volta effettuato il filtraggio, il weight scheduler utilizza un insieme di pesi e algoritmi (ad esempio Round Robin, Least-Used) per decidere il nodo.
- **Compute:** interagisce con l'hypervisor per avviare, arrestare e gestire le VM.
- **Conductor:** intermediario tra compute e DB, viene eseguito sul controller (dove sta anche il DB) e viene invocato da Compute con delle rpc.call(), in questo modo Nova non deve preoccuparsi di come è implementato il DB.
- **Placement:** tracciano l'inventario e l'uso delle risorse

Per scambiare messaggi le componenti interagiscono tramite un paradigma publisher-subscriber usando AMQP al fine di ottenere componenti debolmente accoppiate.

OpenStack Cells permette di suddividere i nodi di calcolo in gruppi. Le celle sono una suddivisione logica e fisica di una grande installazione Nova e permette di separare ambienti di calcolo diversi (ad esempio test/produzione). Ogni cella ha un proprio controller che interagisce con un super-controller. Lo scheduler decide su quali celle distribuire le istanze. Ogni cella ha un proprio DB.

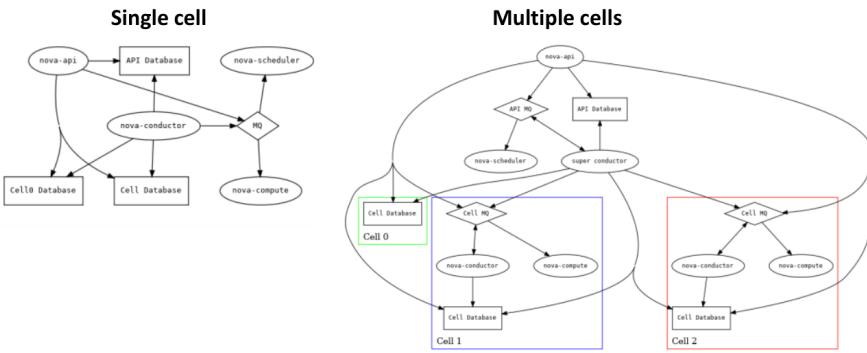


Figura 2.32: Celle Nova

L'avvio di una nuova istanza prevede i seguenti passaggi:

1. Autenticazione dell'utente tramite keystone.
2. Invio della richiesta da parte dell'utente a Nova specificando il tipo di istanza (flavour).
3. Richiesta di Nova a Glance per ottenere l'immagine.
4. Creazione dell'istanza.

Neutron

Il servizio Neutron consente di creare e gestire reti virtuali, subnet, router e altri elementi all'interno di un ambiente OpenStack.

Per connettere due VM sullo stesso host è possibile usare un bridge. Ma le VM sullo stesso host possono appartenere a tenant diversi, si usano quindi le VLAN per separarli. Per connettere VM su host diversi sono possibili 3 opzioni:

- Flat network: VM connesse sulla stessa Ethernet.
- VLAN: tramite VLANID globali.
- VxLAN

L'architettura Neutron prevede 3 componenti principali:

- Neutron server daemon: eseguito sul controller, avvia 2 componenti:
 - Servizio API REST: accetta richieste API e le inoltra al plug-in corretto.
 - Neutron plugin: classi python che permettono di estendere le funzionalità di Neutron. Sono di due tipi:
 - * Core: fornisce connettività L2 e indirizzi IP. Ad esempio il plugin ML2 permette l'integrazione di diverse tecnologie di rete L2 (VLAN, VxLAN, GRE, ...).
 - * Service: fornisce connettività L3 (routing), firewalling, load-balancing, Ad esempio FWaaS, VPNaas, LBaaS,
- Servizio RPC: eseguito sul controller e consente di comunicare con gli agents.
- Agents: eseguiti sui nodi compute, implementano la configurazione di rete ricevuta dal server. Ad esempio DHCP Agent, L3-Agent, plugin-specific-agent.

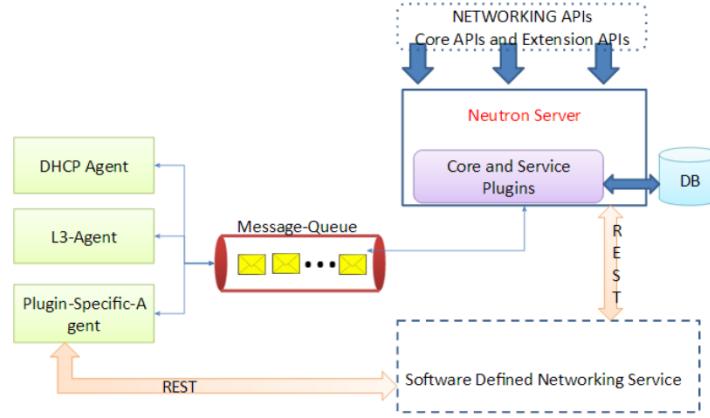


Figura 2.33: Architettura Neutron

Si hanno due tipologia di rete:

- Provider: connettività L2 tra le istanze con supporto al DHCP. Sfrutta direttamente la rete fisica del datacenter e usa le VLAN per separare i tenant. Dato che sfrutta la rete fisica la sua configurazione è effettuata dall'amministratore del datacenter.
- Self-Service: rete virtuale gestita dagli utenti internamente al cloud. Viene connessa alla rete fisica tramite NAT o router virtuali. Nel caso di IPv4 si usa FloatingIP (IP pubblico assegnato in modo dinamico alle istanze) per accedere dall'esterno alle istanze. Nel caso di IPv6 le istanze ricevono un IP pubblico.

SDN consente una gestione centralizzata e programmabile della rete, separando il piano di controllo da quello dati. ML2 può usare driver come OVS per interagire con un controller SDN.

Cinder

Cinder permette di creare e gestire volumi di storage che possono essere utilizzati dalle istanze VM nel cloud. Lo storage a blocchi consente di creare volumi aggiuntivi da montare sulle istanze.

Cinder è composto delle seguenti componenti:

- API REST.
- Scheduler daemon: decide quale nodo ospita il volume.
- Volume daemon: risponde alle richieste di lettura e scrittura e interagisce con diversi provider di storage.
- Backup daemon (opzionale): fornisce un servizio per il backup dei volumi.
- Queue: instrada i messaggi tra i vari processi.

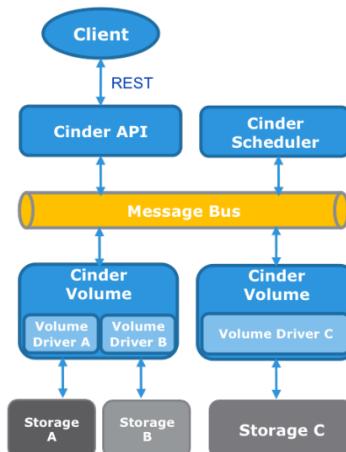


Figura 2.34: Cinder

Swift

Swift offre un servizio di storage per dati non strutturati (immagini, video, ...).

Le principali componenti Swift sono:

- **Proxy Server**: ascolta le richieste dei client, comunica con gli storage server e risponde ai client.
- **Storage Server**: memorizzano dati e metadati. Sono di 3 tipi: account, container e object.
- **Consistency Server**: si occupa di mantenere la consistenza.

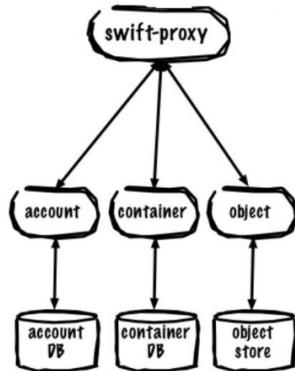


Figura 2.35: Swift

Le richieste alla REST API sono formate da 3 parti:

- **Comando HTTP**: GET, PUT, DELETE,
- **Informazioni di autenticazione**.
- **URL**: del tipo *https://swift.example.com/v1/account/container/object* in cui
 - *https://swift.example.com/v1* indica la locazione del cluster;
 - */account/container/object* indica la locazione di un oggetto;

Un **account** è un'area di storage con un nome univoco globalmente contenente informazioni sull'**account** stesso e una lista di **container** dell'**account**. I **container** sono un'area di storage con un nome univoco all'interno dell'**account** contenenti informazioni sui **container** stessi e una lista di **oggetti**. Gli oggetti sono memorizzati come file binari e metadati riguardanti i file.

Per il **CAP theorem** è impossibile per un sistema di archiviazione distribuito garantire simultaneamente:

- **Consistenza**: ogni lettura riceve la versione più recente dei dati oppure un messaggio di errore;
- **Disponibilità**: ogni richiesta effettuata riceve una risposta;
- **Tolleranza alle partizioni**: il sistema continua a funzionare anche quando c'è un guasto nella rete che impedisce la comunicazione tra alcuni nodi;

È possibile garantire al massimo 2 dei precedenti requisiti, Swift punta a garantire la disponibilità e la tolleranza al partizionamento rinunciando così alla consistenza.

I **Swift Consistency Server** sono responsabili del trovare e correggere errori causati sia da corruzione di dati che da problemi hardware. Utilizza i seguenti processi demone:

- **Auditor**: scannerizza il disco in cerca di dati corrotti, in caso di match sposta tali dati in quarantena e li sostituisce con una copia da Replication.
- **Updaters**: assicura che gli elenchi di account e container siano corretti.
- **Replicators**: assicura che i dati nel cluster siano dove dovrebbero essere e che ci siano abbastanza copie nel sistema.

Per garantire disponibilità e tolleranza al partizionamento di default Swift memorizza 3 copie di ogni oggetto. Gli oggetti vengono partizionati al fine di migliorare la scalabilità in caso di oggetti di grandi dimensioni. Il numero di partizioni memorizzate in un nodo è dunque dato da:

$$\#partNodo = \frac{\#repliche \times \#part}{\#nodi}$$

RICORDA: LA FUNZIONE DI HASH GENERA SEMPRE UN RISULTATO DELLA STESSA CUNIGEZZA (INLESSO O HASH)

Al fine di bilanciare il carico tra i nodi in modo efficace e tollerante ai guasti Swift usa il Consistent Hashing, ovvero utilizza un algoritmo di hashing per distribuire gli oggetti sui nodi, in modo che, se un nodo viene aggiunto o rimosso, solo una minima parte degli oggetti deve essere ridistribuita. Il consistent hashing prevede l'uso di una circonferenza che rappresenta lo spazio di tutti i possibili valori assunti dalla funzione di hash calcolata sugli indici dei nodi. La porzione di circonferenza precedente (in senso antiorario) al valore assunto dall'hash dell'indice del nodo appartiene a quel nodo. Per decidere su quale nodo memorizzare un oggetto di calcola l'hash della chiave di quell'oggetto e si controlla in quale settore della circonferenza appartiene. In questo modo l'aggiunta o la rimozione di un nodo prevede solo la ridistribuzione degli oggetti memorizzati nella porzione precedente (in senso antiorario) a quel nodo. Nell'hashing modulare invece l'indice del nodo in cui memorizzare l'oggetto viene calcolato come $indice = hash(chiave)\%N$ e quindi cambiando N è necessario ridistribuire molti più dati.

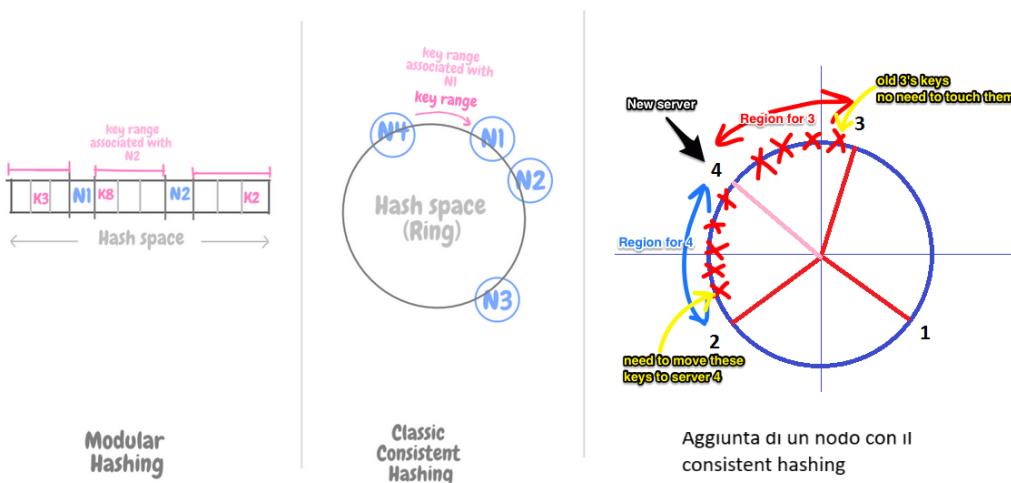


Figura 2.36: Consistent hashing

Ceilometer

Ceilometer fornisce un sistema di monitoraggio e metering per i servizi cloud. Il suo scopo è raccogliere e aggregare dati sull'utilizzo delle risorse al fine di essere usati per scopi di fatturazione, monitoraggio delle prestazioni o per generare statistiche sulle risorse consumate.

Ceilometer è formato dalle seguenti componenti:

- **Compute agent:** eseguito sui nodi compute e si occupa di raccogliere dati dall'infrastruttura di calcolo (istanze VM di Nova).
- **Central agent:** eseguito su un nodo controller e si occupa di raccogliere dati dai servizi centrali come Swift, Neutron e Cinder.
- **Notification agent:** legge i messaggi da una coda (RabbitMQ) genera eventi e statistiche e le invia a Gnocchi, ovvero un database con funzionalità di aggregazione dei dati.

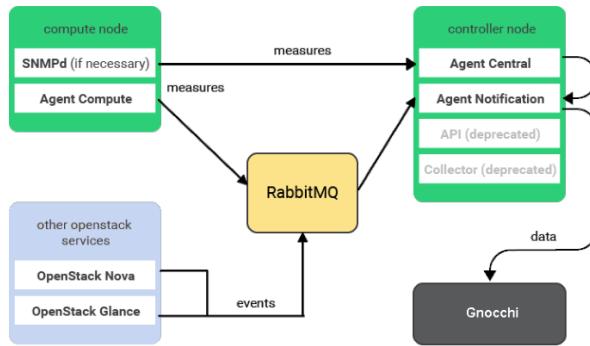


Figura 2.37: Ceilometer

AODH

AODH è il servizio di allarmistica utilizzato per monitorare e gestire allarmi basati su metriche raccolte da Ceilometer o Gnocchi. Le principali componenti sono:

- API Server: fornisce l'accesso ai dati degli allarmi memorizzati in un DB.
- Alarm evaluator: aziona allarmi basati su soglie.
- Notification Listener: aziona allarmi basati su eventi.
- Alarm notifier: permette di impostare allarmi.

Gli allarmi possono trovarsi in 3 stati: ok, alarm, insufficient data. Per impostare allarmi basati su soglie è necessario specificare: soglia con comparatore (minore/maggiore), statistica su cui applicare la soglia, sliding window (intervallo di tempo su cui calcolare la statistica).

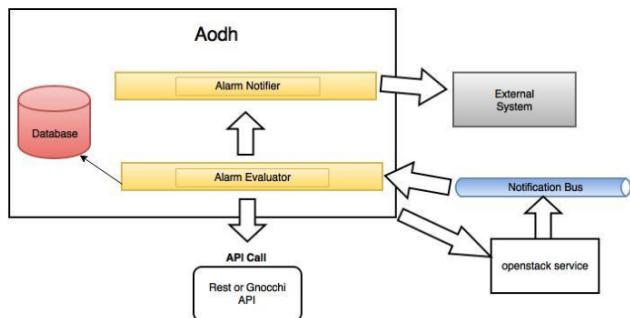


Figura 2.38: AODH

Heat

Heat consente di definire, distribuire e gestire applicazioni tramite template in formato yaml chiamati HOT (Heat Orchestration Template) che descrivono le risorse necessarie (VM, reti, volumi, ...) e le loro relazioni.

Nell'ambito di Heat uno stack è l'insieme delle risorse create e gestite da Heat in base al template.

Un file yaml HOT presenta i seguenti campi:

- description: breve descrizione dello scopo del template.
- parameter_groups: permette di raggruppare i parametri in categorie per facilitarne la lettura.
- parameter: parametri di input fornibili dall'utente durante la creazione dello stack.
- resources: risorse che heat creerà;
- outputs: definisce i valori che saranno restituiti alla fine della creazione dello stack.
- conditions: logica condizionale all'interno del template, permette di abilitare/disabilitare risorse in base a parametri.

```

heat_template_version: 2015-04-30
description: Simple template to deploy a single compute instance
parameters:
  key_name:
    type: string
    label: Key Name
    description: Name of key-pair to be used for compute instance
  image_id:
    type: string
    label: Image ID
    description: Image to be used for compute instance
  instance_type:
    type: string
    label: Instance Type
    description: Type of instance (flavor) to be used
resources:
  my_instance:
    type: OS::Nova::Server
    properties:
      key_name: { get_param: key_name }
      image: { get_param: image_id }
      flavor: { get_param: instance_type }

```

Listing 2.1: Esempio di file yaml HOT

2.8 Serverless e OpenFaaS

2.8.1 Serverless

Con il termine serverless si intende un modello di esecuzione cloud dove il provider del servizio cloud alloca le risorse della macchina appena queste vengono richieste. Quando un app non è in uso, nessuna risorsa viene consumata e il prezzo è basato solo sulle risorse utilizzate. Il termine serverless è fuorviante perché sono comunque utilizzati dei server in cloud, ma questi vengono astratti ai programmatore, che non devono più occuparsi di mantenimento o configurazione.

Conviene adottare una soluzione serverless quando:

- Applicazioni Event-Driven
- Necessità di scalabilità dinamica
- Microservizi e architettura modulare
- Elaborazione dati e automazione
- Applicazioni che richiedono bassa latenza su scala globale: è possibile eseguire il codice vicino agli utenti finali per ridurre la latenza.

Non conviene adottare una soluzione serverless quando:

- Applicazioni con carichi di lavoro continuativi
- Requisiti di stato persistente
- Vincoli di latenza da cold start
- Vincoli di latenza da cold start: funzioni che non possono tollerare ritardi introdotti dall'avvio di un'istanza fredda.

2.8.2 OpenFaaS

Si è visto che Kubernetes è uno strumento versatile, esso infatti non impone nessun vincolo sulla natura dei pod, che possono contenere anche software stateless. È quindi possibile utilizzare Kubernetes come strumento attuativo per le funzioni stateless.

OpenFaaS è un framework open-source che permette di creare e distribuire funzioni serverless sfruttando i container, si appoggia a Kubernetes per l'orchestrazione dei container.

OpenFaas permette di:

- eseguire funzioni su qualsiasi provider cloud;
- scrivere funzioni in qualsiasi linguaggio e impacchettarle in container docker;
- scalare il numero di funzioni in esecuzione in base alla richiesta (supporta anche lo scaling down to zero);

Per creare una funzione non è sufficiente scrivere il codice, aggiungere le dipendenze, creare il dockerfile ed effettuare il build, è infatti necessario avere una buona conoscenza di OpenFaaS per rendere il tutto compatibile con OpenFaaS. Per questo motivo OpenFaaS mette a disposizione dei template (distribuiti tramite un repository) per la maggior parte dei linguaggi di programmazione, è quindi sufficiente scaricare il template e modificare il codice della funzione.

Watchdog

Esistono due tipologie di template e differiscono in base alla tipologia di watchdog che usano. Il **watchdog** in OpenFaaS è un componente fondamentale per la gestione delle funzioni serverless. È un micro-server HTTP che agisce come un ponte tra il framework OpenFaaS e la funzione utente. Il suo scopo principale è quello di semplificare il processo di chiamata ed esecuzione delle funzioni, fornendo un'interfaccia uniforme per tutte le funzioni, indipendentemente dal linguaggio o dall'implementazione.

Il **classic watchdog** fa uso di *stdio* per la comunicazione con le funzioni e viene utilizzato dai template presenti nel repository openfaas. Il template che fa uso di classic watchdog inserisce nell'entrypoint o nel cmd del container il binary eseguibile del watchdog, in questo modo il processo init del container sarà proprio il watchdog. Quando una richiesta arriverà al watchdog essa effettuerà un fork creando il processo che esegue la funzione. Il processo watchdog e quello della funzione comunicano tramite *stdin*, *stdout* e *stderr*. Quando la funzione finisce la sua esecuzione il processo relativa ad essa termina. Questa tipologia di watchdog supporta i metodi POST, PUT, DELETE, UPDATE con body e GET senza body.

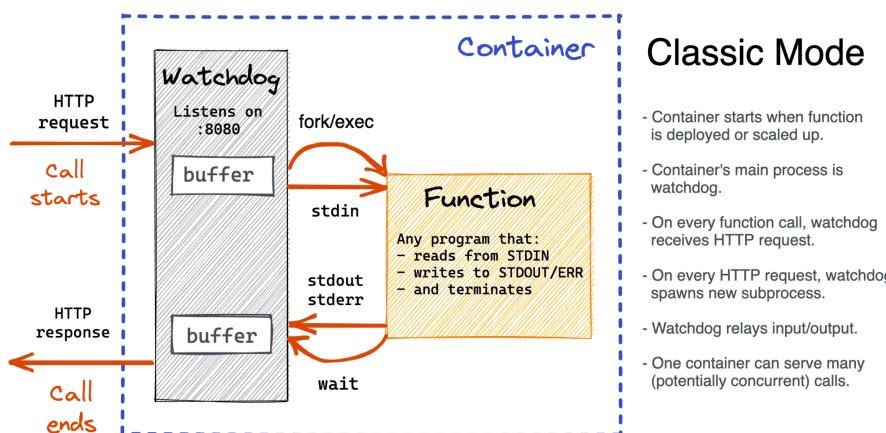


Figura 2.39: Watchdog classico

Il **of-watchdog** fa uso di *http* per comunicare con le funzioni e viene utilizzato dai template presenti in un repository GitHub. Questa tipologia di watchdog risolve i problemi di latenza dovuti al forking per la creazione del processo che esegue la funzione. Il processo che ospita la funzione viene avviato immediatamente dopo il watchdog e il watchdog funge da reverse proxy per inoltrare le richieste alle funzioni. Con questa modalità le funzioni sono implementate come piccoli server http che possono essere contattati tramite *localhost:3000*.

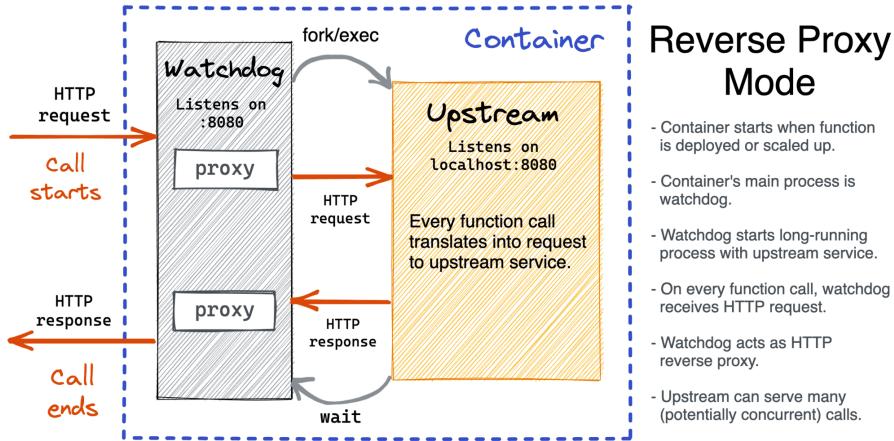


Figura 2.40: of-watchdog

Invocazione di funzioni

In OpenFaaS ogni funzione è distribuita come un Deployment e un Service Kubernetes. Il Service fornisce un endpoint in ascolto sulla porta 80 per accedere alla funzione. Le funzioni possono essere invocate tramite una richiesta HTTP al gateway OpenFaaS, specificando il path nell'URL, ad esempio `http://127.0.0.1:8080/function/NAME`.

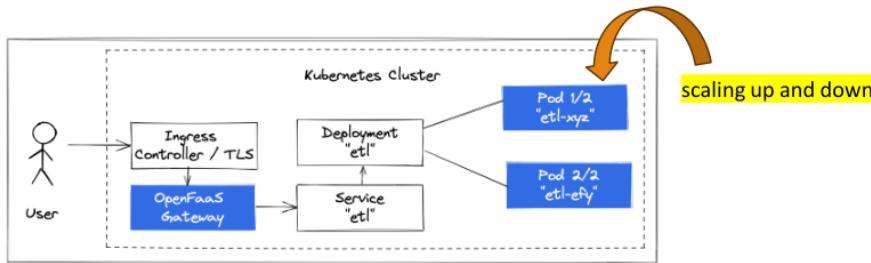


Figura 2.41: Invocazione di funzioni

La modalità sincrona di invocazione prevede che la richiesta HTTP venga inoltrata immediatamente dal gateway openfaas al pod che implementa la funzione.

È possibile invocare le funzioni anche in modalità asincrona utilizzando un URL del tipo `/async-function/NAME`. In questa modalità le richieste alle funzioni non vengono inoltrate immediatamente al pod, ma vengono accodate in una coda NATS, e viene inviato un messaggio di conferma di accodamento al client. Quando la richiesta uscirà dalla coda verrà effettuata una richiesta sincrona. Il client si sottoscrive per ottenere una risposta tramite un webhook. Il webhook permette al client di proseguire il lavoro senza rimanere in attesa della risposta del server. Consiste nell'inviare un URL al server e nell'impostare un evento che, una volta triggerato, permette al server di inviare la risposta al client all'URL configurato.

Eventi

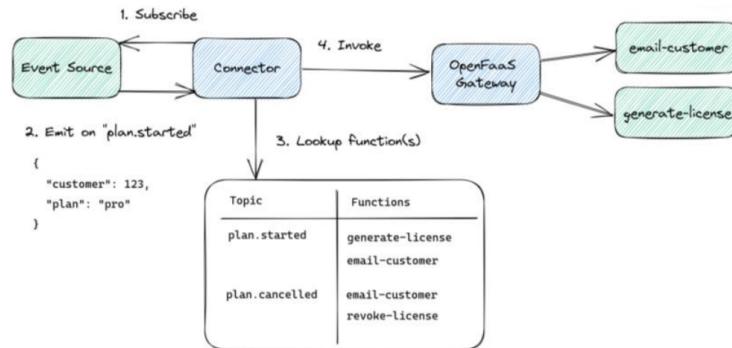


Figura 2.42: Eventi OpenFaaS

Il Pattern Event-Connector nel contesto di OpenFaaS si riferisce a un modello di progettazione in cui eventi provenienti da sistemi esterni attivano funzioni serverless distribuite sulla piattaforma OpenFaaS.

La sorgente degli eventi può essere qualsiasi sistema, come topic Kafka, broker MQTT, eventi AWS S3, webhook, ecc. Gli eventi sono generalmente messaggi codificati in JSON o payload simili.

Il connector è un componente intermedio che ascolta la sorgente degli eventi utilizza l'API del Gateway OpenFaaS per invocare la funzione corrispondente.

In questo modo è possibile integrare sorgenti di eventi diverse senza cambiare il design principale.

Autoscaling

Per effettuare l'autoscaling in OpenFaaS è possibile utilizzare la funzionalità di autoscaling offerta da Kubernetes, oppure utilizzare un autoscaler built-in di OpenFaaS. Tale autoscaler fa uso di Prometheus per il raccoglimento di statistiche e dell'Alert Manager per il triggering di allarmi.

OpenFaaS presenta tre modalità di autoscaling:

- **Capacity**: scala le funzioni in base al numero di richieste concorrenti in elaborazione rispetto alla capacità di una singola replica. Ogni funzione ha un limite di concorrenza (max_inflight), che definisce quante richieste una singola replica può gestire contemporaneamente. Se il numero di richieste in attesa supera questo limite, OpenFaaS scala orizzontalmente aggiungendo più repliche.
- **RPS**: scala in base al numero di richieste al secondo che arrivano alla funzione. Se il tasso di richieste supera una certa soglia, il sistema scala il numero di repliche.
- **CPU**: scala le funzioni OpenFaaS in base all'utilizzo della CPU o della memoria. Se l'utilizzo supera una soglia definita, Kubernetes scala automaticamente i pod.

OpenFaaS supporta la scalabilità a zero, ovvero disattivare tutte le repliche di una funzione quando non ci sono richieste per un certo periodo di tempo.

Capitolo 3

Teoria delle code

La teoria delle code è utilizzata per modellare sistemi dinamici in cui ci sono risorse limitate che devono servire una domanda variabile, con l'obiettivo di analizzare e migliorare le prestazioni del sistema. La teoria delle code si adatta quindi perfettamente in contesti di cloud computing in cui c'è bisogno di:

- determinare quanti server includere in un tenant;
- valutare l'effetto di un broker sulla latenza di accesso ad un servizio;
- determinare il numero di istanze nell'HPA;
- ...

3.1 Sistema a coda

Un sistema a coda è un sistema dinamico in cui:

- una popolazione di clienti effettua richieste;
- i serventi processano le richieste;
- quando le richieste superano la capacità di servizio immediata si formano code;

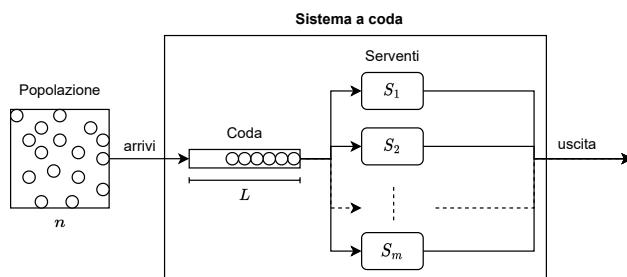


Figura 3.1: Sistema a coda

3.1.1 Caratterizzazione matematica

Si definisce:

- n cardinalità della popolazione;
- L dimensione della coda;
- m numero di serventi;
- $C = m + L$ capacità del sistema;

In base alle precedenti definizioni è possibile classificare il sistema in:

- sistema a perdita pura (senza attesa) se $n > C$ e $L = 0$;

- sistema a perdita (con attesa) se $n > C$ e $L > 0$;
- sistema ad attesa (senza perdita) se $n \leq C$ e $L > 0$;

Si definiscono inoltre le seguenti grandezze temporali:

- τ_i : tempo di arrivo dell' 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 dell' i -esima richiesta;
- x_i : tempo di servizio dell' i -esima richiesta;
- s_i : tempo di sistema dell' i -esima richiesta;

Il tempo di sistema è dato dalla somma del tempo di attesa in coda e dal tempo di servizio $s_i = w_i + x_i$. Inoltre se L_i è il lavoro da erogare per soddisfare l' i -esima richiesta (misurato in byte o #istruzioni) e Γ è la capacità del servente di svolgere lavoro (misurato in byte/sec o #istruzioni/sec) allora x_i , ovvero il tempo di servizio necessario per completare l' i -esima richiesta è $x_i = \frac{L_i}{\Gamma}$.

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 di t_i e x_i è una realizzazione di una variabile aleatoria. Normalmente si suppone che i due processi siano stazionari e statisticamente indipendenti.

3.1.2 Grandezze limite

Le variabili aleatorie t_i e x_i in generale hanno una statistica che si evolve nel tempo, ad un certo punto però tale dinamica può stabilizzarsi ottenendo così una situazione di regime. Si tratta di uno scenario realistico, si pensi ad esempio ad un servizio che viene messo in opera, inizialmente si avrà un improvviso aumento di traffico che con il tempo si stabilizzerà.

Al crescere di n t_i tenderà ad una v.a. $\tilde{t} = \lim_{n \rightarrow \infty} t_n$ (con statistica fissa). Con $A_n(t) = P(t_n < t)$ si indica la CDF di t_n , mentre con $A(t) = P(\tilde{t} < t)$ si indica la CDF di \tilde{t} . Per ogni n la v.a. t_n avrà la sua media $E\{t_n\} = \bar{t}_n$ e per $n \rightarrow \infty$ si avrà $E\{\tilde{t}\} = \bar{t} = \frac{1}{\lambda}$, dove λ è la frequenza media di interarrivo.

Analogamente è possibile definire le stesse grandezze per w_n , x_n e s_n . Di seguito una lista esaustiva:

- $\tilde{t} = \lim_{n \rightarrow \infty} t_n$, $E\{\tilde{t}\} = \bar{t} = \frac{1}{\lambda}$ dove λ è la **frequenza media di interarrivo**.
- $\tilde{w} = \lim_{n \rightarrow \infty} w_n$, $E\{\tilde{w}\} = \bar{w} = W$ dove W è il **tempo medio di attesa**.
- $\tilde{x} = \lim_{n \rightarrow \infty} x_n$, $E\{\tilde{x}\} = \bar{x} = \frac{1}{\mu}$ dove μ è la **frequenza media di erogazione del servizio**.
- $\tilde{s} = \lim_{n \rightarrow \infty} s_n$, $E\{\tilde{s}\} = \bar{s} = T$ dove T è il **tempo medio di permanenza nel sistema**.

3.1.3 Notazione di Kendall

La notazione di Kendall è un modo sintetico per descrivere un particolare modello del sistema a code. Essa prevede l'utilizzo di sei termini separati da uno /, del tipo 1/2/3/4/5/6, dove:

1. descrive la distribuzione di probabilità dei tempi di arrivo ($A(t)$);
2. rappresenta la distribuzione di probabilità dei tempi di servizio ($B(t)$);
3. numero di serventi (m);
4. dimensione del sistema ($C = m + L$);
5. cardinalità della popolazione (n);
6. politica di gestione della coda (se non indicata di default è FIFO);

Le distribuzioni $A(t)$ e $B(t)$ possono essere di vario tipo e per ognuna di esse si utilizza un particolare codice:

- M per l'esponenziale negativa (o Markoviana);

- D per una distribuzione deterministica (v.a. costante);
- E_i per l'erlangiana con i stadi;
- H_i per l'iper-esponenziale con i stadi;
- G per una distribuzione generale;

3.1.4 Legge di Little

La legge di Little stabilisce che il numero medio di clienti in un sistema (\bar{N}) è uguale al tasso medio di arrivo (λ) moltiplicato per il tempo medio nel sistema (T):

$$\bar{N} = \lambda T$$

e dato che $T = W + \bar{x}$ si ha che

$$\bar{N} = \lambda T = \lambda W + \lambda \bar{x} = \bar{N}_q + \bar{N}_s$$

dove \bar{N}_q e \bar{N}_s sono rispettivamente il numero medio di elementi in coda e il numero medio di elementi in servizio.

Tale legge assume che il sistema sia senza perdite e che il si trovi in una condizione di stazionarietà.

Dimostrazione.

Chiamando:

- $\alpha(t)$: numero di arrivi in $[0, t]$;
- $\delta(t)$: numero di partenze in $[0, t]$;
- $N(t) = \alpha(t) - \delta(t)$: numero di clienti nel sistema all'istante t ;

e utilizzando l'ipotesi di sistema senza perdite, è possibile dire che $\gamma(t) = \int_0^t N(t) dt$ rappresenta il tempo che tutti i clienti hanno trascorso nel sistema nell'intervallo $[0, t]$.

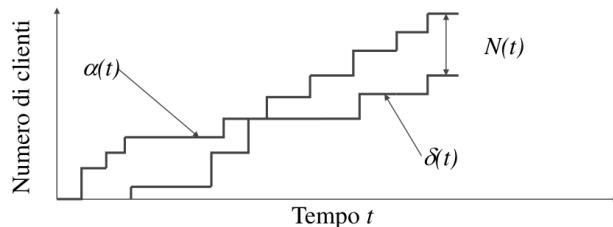


Figura 3.2: Numero di arrivi e di partenze (cumulativi) e numero di clienti (istantanei) nel sistema

Si definiscono ora le seguenti quantità:

- $\lambda_t = \frac{\alpha(t)}{t}$: frequenza media di interarrivo nell'intervallo $[0, t]$;
- $T_t = \frac{\gamma(t)}{\alpha(t)}$: tempo medio di sistema per ogni utente nell'intervallo $[0, t]$;
- $N_t = \frac{\gamma(t)}{t}$: numero medio di clienti nel sistema nell'intervallo $[0, t]$;

Dalla prima si ha $t = \frac{\alpha(t)}{\lambda_t}$ e sostituita nella terza si ottiene $N_t = \gamma(t) \frac{\lambda_t}{\alpha(t)}$. Dalla seconda è quindi possibile dire che $N_t = \lambda_t T_t$.

Utilizzando ora l'ipotesi di stazionarietà è possibile calcolare i valori al limite $\lambda = \lim_{t \rightarrow \infty} \lambda_t$ e $T = \lim_{t \rightarrow \infty} T_t$ da cui si ottiene che

$$\bar{N} = \lambda T$$

□

Da notare che tale risultato è indipendente dalle particolari distribuzioni $A(t)$ e $B(t)$ scelte e dal numero di serventi. È sufficiente infatti che il sistema sia senza perdite e che si raggiunga la condizione di stazionarietà.

3.1.5 Fattore di utilizzazione

Il **fattore di utilizzazione** è il rapporto tra la frequenza con la quale il “lavoro” entra nel sistema (λ) e quella con la quale i serventi riescono a smaltirlo (μ):

$$\rho = \frac{\lambda}{\mu}$$

Nel caso di m serventi si ha ovviamente $\rho = \frac{\lambda}{m\mu}$.

Il fattore di utilizzazione rappresenta quindi la frazione del tempo in cui un server è occupato.

Un sistema a code è instabile se il carico in arrivo supera la capacità totale del sistema $\rho \geq 1$. In questo caso la lunghezza della coda cresce indefinitamente, i tempi di attesa diventano teoricamente infiniti e il sistema non raggiunge uno stato stazionario.

Un sistema a code è stabile se il tasso di servizio totale è sufficiente a gestire il tasso di arrivo delle richieste $0 \leq \rho < 1$. In questo caso la lunghezza media della coda rimane finita, i tempi di attesa medi sono prevedibili e controllabili e il sistema raggiunge uno stato stazionario, dove le prestazioni possono essere analizzate matematicamente.

Per un intervallo di tempo τ sufficientemente grande il numero di arrivi sarà pari a $\lambda\tau$. Definendo p_0 come la probabilità di trovare il server libero in un generico istante di tempo, il server sarà occupato per un tempo pari a $\tau(1 - p_0)$. Il numero di utenti serviti in quel intervallo sarà quindi $m(\tau - \tau p_0)\mu$. Eguagliando il numero di arrivi con il numero di utenti serviti (condizione di stabilità), si ottiene

$$\lambda\tau \simeq m(\tau - \tau p_0)\mu \xrightarrow{\tau \rightarrow \infty} \rho = 1 - p_0$$

3.1.6 Stato del sistema

La velocità nel soddisfare ulteriori richieste dipende da quanti utenti si trovano già nel sistema. Dipendono quindi dalla storia più o meno recente del sistema. Lo stato del sistema è una traccia di quanto accaduto nel passato nel sistema e ne influenza il futuro.

Si utilizza la notazione $X(t)$ per indicare la v.a. che rappresenta lo stato del sistema all’istante t . Si considerano sistemi con stati $X(t)$ discreti e tempo continuo.

La probabilità che il sistema si trovi in un generico stato j al tempo t è indicata con:

$$\Pi_j(t) = P\{X(t) = j\}$$

3.2 Catene di Markov

Un processo aleatorio $\{X_n\}$ è detto catena di Markov se lo spazio degli stati è discreto e gode della proprietà di Markov, ovvero la probabilità di trovarsi in un tempo futuro in un determinato stato può essere espressa in funzione dello stato assunto al tempo corrente e non occorre specificare quali stati sono stati assunti in precedenza.

La conoscenza più recente dello stato del sistema rende inutile la conoscenza degli stati assunti in precedenza e quindi la conoscenza del passato non consente di predire quanto tempo il processo debba rimanere nello stato in cui si trova. La distribuzione del tempo che il processo rimanga in uno stato è senza memoria, e nell’ipotesi di tempo continuo ciò porta alla distribuzione esponenziale del tempo di permanenza nello stato.

La proprietà di markovianità può essere espressa nel seguente modo

$$P\{X(t_{n+1}) = j | X(t_n) = i_n, X(t_{n-1}) = i_{n-1}, \dots, X(t_1) = i_1\} = P\{X(t_{n+1}) = j | X(t_n) = i_n\}$$

3.2.1 Equazioni di Chapman-Kolmogorov

La probabilità di transizione da uno stato i ad uno stato j è indicata con:

$$p_{ij}(s, t) = P\{X(t) = j | X(s) = i\} \quad \text{per } t \geq s$$

e può essere calcolata come

$$\begin{aligned}
p_{ij}(s, t) &= \sum_k P\{X(t) = j, X(u) = k | X(s) = i\} = \\
&= \sum_k P\{X(u) = k | X(s) = i\} P\{X(t) = j | X(s) = i, X(u) = k\} = \\
&= \sum_k P\{X(u) = k | X(s) = i\} P\{X(t) = j | X(u) = k\} = \\
&= \sum_k p_{ik}(s, u) p_{kj}(u, t)
\end{aligned}$$

Forma scalare

L'equazione

$$p_{ij}(s, t) = \sum_k p_{ik}(s, u) p_{kj}(u, t)$$

è nota come **equazione di Chapman-Kolmogorov**. Tale equazione può essere scritta anche come prodotto scalare tra due vettori:

$$p_{ij}(s, t) = [p_{i1}(s, u) \quad p_{i2}(s, u) \quad \dots \quad p_{ik}(s, u) \quad \dots] \begin{bmatrix} p_{1j}(u, t) \\ p_{2j}(u, t) \\ \dots \\ p_{kj}(u, t) \\ \dots \end{bmatrix}$$

Forma matriciale

Definendo la matrice $H(s, t)$ come

$$H(s, t) = \begin{bmatrix} p_{11}(s, t) & p_{12}(s, t) & \dots & p_{1j}(s, t) & \dots \\ p_{21}(s, t) & p_{22}(s, t) & \dots & p_{2j}(s, t) & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \\ p_{i1}(s, t) & p_{i2}(s, t) & \dots & p_{ij}(s, t) & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

è possibile dire che

$$H(s, t) = H(s, u) H(u, t)$$

nota come **equazione di Chapman-Kolmogorov in forma matriciale**.

Forma differenziale

Definendo $P(t) = H(t, t + \Delta t)$ e osservando che

$$\begin{aligned}
H(s, t) - H(s, t - \Delta t) &= H(s, t - \Delta t) H(t - \Delta t, t) - H(s, t - \Delta t) = \\
&= H(s, t - \Delta t) P(t - \Delta t) - H(s, t - \Delta t) = \\
&= H(s, t - \Delta t) (P(t - \Delta t) - I)
\end{aligned}$$

è possibile dire che

$$\frac{\partial H(s, t)}{\partial t} = \lim_{\Delta t \rightarrow 0} \frac{H(s, t) - H(s, t - \Delta t)}{\Delta t} = \lim_{\Delta t \rightarrow 0} \left(H(s, t - \Delta t) \frac{(P(t - \Delta t) - I)}{\Delta t} \right) = H(s, t) Q(t)$$

dove

$$Q(t) = \lim_{\Delta t \rightarrow 0} \frac{P(t - \Delta t) - I}{\Delta t} = [q_{ij}(t)] \quad q_{ij}(t) = \begin{cases} \lim_{\Delta t \rightarrow 0} \frac{p_{ii}(t - \Delta t, t) - 1}{\Delta t} & i = j \\ \lim_{\Delta t \rightarrow 0} \frac{p_{ij}(t - \Delta t, t)}{\Delta t} & i \neq j \end{cases}$$

L'equazione

$$\frac{\partial H(s, t)}{\partial t} = H(s, t) Q(t)$$

è nota come **equazione di Chapman-Kolmogorov in forma differenziale**.

Generatore infinitesimale

La matrice Q è nota come **generatore infinitesimale** o rate matrix del processo di Markov e i suoi elementi q_{ij} rappresentano la frequenza degli eventi che determinano l'evoluzione del processo di Markov (dimostrazione più avanti).

Dato che $q_{ij}(t) = \begin{cases} \lim_{\Delta t \rightarrow 0} \frac{p_{ii}(t - \Delta t, t) - 1}{\Delta t} & i = j \\ \lim_{\Delta t \rightarrow 0} \frac{p_{ij}(t - \Delta t, t)}{\Delta t} & i \neq j \end{cases}$ si ha:

$$\begin{aligned} \sum_j q_{ij} &= \sum_{i \neq j} [q_{ij}(t)] + q_{ii}(t) \\ &= \lim_{\Delta t \rightarrow 0} \frac{\sum_{i \neq j} [p_{ij}(t - \Delta t, t)] + p_{ii}(t - \Delta t, t) - 1}{\Delta t} = \\ &= \lim_{\Delta t \rightarrow 0} \frac{\overbrace{\sum_{j \neq i} [p_{ij}(t - \Delta t, t)] + p_{ii}(t - \Delta t, t) - 1}^{=1}}{\Delta t} = \\ &= \lim_{\Delta t \rightarrow 0} \frac{1 - 1}{\Delta t} = 0 \end{aligned}$$

e dato che $\sum_j q_{ij} = \sum_{i \neq j} [q_{ij}(t)] + q_{ii}(t) = 0$ si ha che

$$q_{ii}(t) = - \sum_{i \neq j} q_{ij}(t) < 0$$

È quindi possibile riscrivere la rate matrix come

$$Q(t) = \begin{bmatrix} -\sum_{j \neq 1} q_{ij}(t) & q_{12}(t) & \dots & q_{1N}(t) \\ q_{21}(t) & -\sum_{j \neq 2} q_{2j}(t) & \dots & q_{2N}(s, t) \\ \vdots & \vdots & \ddots & \vdots \\ q_{N1}(t) & q_{N2}(2) & \dots & -\sum_{j \neq N} q_{Nj}(t) \end{bmatrix}$$

Interpretazione fisica della rate matrix

Dato che $q_{ij}(t) = \frac{dp_{ij}(t, t + \Delta t)}{dt}$ è possibile dire che

$$p_{ij}(t, t + \Delta t) \simeq |q_{ij}(t)|\Delta t$$

Nel caso in cui si ha un sistema con due stati i e j e ci si trova nello stato i all'istante 0 è possibile calcolare la probabilità che si rimanga nello stato iniziale i nell'intervallo $[0, t + \Delta t]$ (indicata con $p_0(0, t + \Delta t)$) come:

$$1 - p_{ij}(0, t + \Delta t) = p_0(0, t + \Delta t) = p_0(0, t)(1 - p_{ij}(t, t + \Delta t)) \simeq p_0(0, t)(1 - |q_{ij}(t)|\Delta t)$$

e quindi

$$\frac{dp_0(0, t)}{dt} = \lim_{\Delta t \rightarrow 0} \frac{p_0(0, t + \Delta t) - p_0(0, t)}{\Delta t} = \lim_{\Delta t \rightarrow 0} \frac{p_0(0, t)(1 - |q_{ij}(t)|\Delta t) - p_0(0, t)}{\Delta t} = -|q_{ij}(t)|p_0(0, t)$$

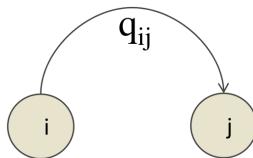


Figura 3.3: Sistema con due stati

E quindi dato che $\frac{dp_0(0, t)}{dt} = -|q_{ij}(t)|p_0(0, t)$ è possibile dire che

$$p_0(0, t) = e^{-\int_0^t |q_{ij}(t)| dt} \quad p_{ij}(0, t) = 1 - e^{-\int_0^t |q_{ij}(t)| dt}$$

In condizioni di stazionarietà si ha $q_{ij}(t) = q_{ij}$ e quindi $p_{ij}(0, t) = 1 - e^{-|q_{ij}|t}$. Dato che $p_{ij}(0, t) = 1 - e^{-|q_{ij}|t}$ rappresenta la probabilità che si verifichi un evento nell'intervallo $[0, t]$ essa è la CDF dell'evento. La sua pdf è quindi $f(t) = \frac{d(1 - e^{-|q_{ij}|t})}{dt} = |q_{ij}|e^{-|q_{ij}|t}$. Il tempo medio in cui si verifica un evento è quindi

$$E\{t\} = \int_0^\infty tf(t)dt = \int_0^\infty t|q_{ij}|e^{-|q_{ij}|t}dt = \frac{1}{|q_{ij}|} = \bar{T}_{ij}$$

La frequenza media di transizione è quindi data proprio da q_{ij} .

Forme alternative

Si è già vista la forma “in avanti” dell’equazione di Chapman-Kolmogorov

$$\frac{\partial H(s, t)}{\partial t} = H(s, t)Q(t) \quad s \leq t$$

mentre la forma “all’indietro” è

$$\frac{\partial H(s, t)}{\partial s} = -Q(s)H(s, t) \quad s \leq t$$

Soluzione all’equazione di Chapman-Kolmogorov

La soluzione all’equazione C-K in forma differenziale $\frac{\partial H(s, t)}{\partial t} = H(s, t)Q(t)$ è data da:

$$H(s, t) = e^{\int_s^t Q(u)du}$$

La probabilità di una transizione da $i \rightarrow j$ nell’intervallo $[t, t + \Delta t]$ è data da:

$$p_{ij}(t, t + \Delta t) = q_{ij}(t)\Delta t + o(\Delta t)$$

La probabilità di uscire dallo stato i -esimo nell’intervallo $[t, t + \Delta t]$ è data da:

$$\begin{aligned} 1 - p_{ii}(t, t + \Delta t) &= \sum_{j \neq i} p_{ij}(t, t + \Delta t) + o(\Delta t) = \\ &= \sum_{j \neq i} q_{ij}(t)\Delta t + o(\Delta t) = \\ &= \Delta t \sum_{j \neq i} q_{ij}(t) + o(\Delta t) = \\ &= -q_{ii}(t)\Delta t + o(\Delta t) \end{aligned}$$

3.2.2 Probabilità di uno stato

Si usa la notazione $\pi_j(t)$ per indicare la probabilità di trovarsi nello stato j all’istante t :

$$\pi_j(t) = P\{X(t) = j\}$$

Per $t > s$ si ha che la probabilità di trovarsi nello stato j all’istante t dipende dalla probabilità di trovarsi in uno stato i all’istante s e dalla probabilità di transitare da i a j nell’intervallo $[s, t]$ per ogni possibile stato i :

$$\pi_j(t) = \sum_i \pi_i(s)p_{ij}(s, t) = [\underbrace{\pi_1(s) \quad \pi_2(s) \quad \dots \quad \pi_i(s) \quad \dots}_\text{H.T.}] \begin{bmatrix} p_{1j}(s, t) \\ p_{2j}(s, t) \\ \vdots \\ p_{ij}(s, t) \\ \vdots \end{bmatrix} \xleftarrow{\text{probabilità}} \text{H.T.}$$

e quindi

$$\pi^T(t) = \pi^T(s)H(s, t) \quad \xrightarrow{\text{H.T.}}$$

Usando tale relazione è possibile derivare l’**equazione di C-K in avanti**:

$$\frac{d\pi^T(t)}{dt} = \pi^T(s) \frac{\partial H(s, t)}{\partial t} \xrightarrow{\text{cioè C-K in avanti}} = \pi^T(s)H(s, t)Q(t) = \pi^T(t)Q(t)$$

e analogamente per quella all’indietro $\frac{d\pi(s)}{ds} = Q(s)\pi(s)$.

$\frac{\partial H(s, t)}{\partial t} = H(s, t)Q(t) \quad s \leq t$
$\frac{\partial H(s, t)}{\partial s} = -Q(s)H(s, t) \quad s \leq t$

3.2.3 Catene di Markov omogenee

Una catena di Markov tempo-continua è detta omogenea se le probabilità di transizione fra due stati qualsiasi, in un dato intervallo di tempo, non dipende dall'istante di inizio di tale intervallo ma solo dalla sua durata.

Per catene di Markov omogenee si ha quindi che:

$$p_{ij}(s, t) = p_{ij}(\tau) \Rightarrow H(s, t) = H(\tau) = [p_{ij}(\tau)] \quad \tau = t - s$$

$$q_{ij}(t) = q_{ij} \Rightarrow Q(t) = Q = [q_{ij}]$$

$$q_{ij}(t) = \begin{cases} \lim_{\Delta t \rightarrow 0} \frac{p_{ij}(t - \Delta t, t) - 1}{\Delta t} & i = j \\ \lim_{\Delta t \rightarrow 0} \frac{p_{ij}(t - \Delta t, t)}{\Delta t} & i \neq j \end{cases}$$

L'equazione è
volumetrica
semplicificata

Quindi se la frequenza di transizione fra due stati in un certo intervallo di tempo non dipende dall'istante considerato si ha che la frequenza di transizione di stato si mantiene costante nel tempo.

L'equazione di C-K in avanti è scrivibile come:

$$\frac{d\pi^T(t)}{dt} = \pi^T(t)Q$$

La singola componente è calcolabile come:

$$\frac{d\pi_i^T(t)}{dt} = [\pi_1(t) \quad \pi_2(t) \quad \dots \quad \pi_i(t) \quad \dots] \begin{bmatrix} q_{1i} \\ q_{2i} \\ \vdots \\ q_{ii} \\ \vdots \end{bmatrix} =$$

$$= \pi_1(t)q_{1i} + \pi_2(t)q_{2i} + \dots + \pi_i(t)q_{ii} + \dots =$$

$$= \sum_{j \neq i} \pi_j(t)q_{ji} + \pi_i(t)q_{ii} \quad q_{ii} = -\sum_{i \neq j} q_{ij}$$

$$= \sum_{j \neq i} \pi_i(t)q_{ji} - \pi_i(t) \sum_{i \neq j} q_{ij}$$

PERICOLO ENTRANTE *FLUSSO USCENTE*

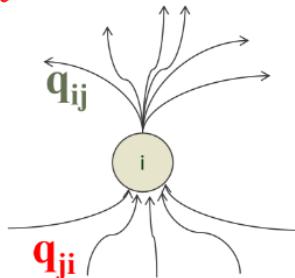


Figura 3.4: Flusso di probabilità in entrata ed in ingresso

La quantità $-\pi_i(t) \sum_{i \neq j} q_{ij}$ rappresenta il flusso di probabilità dallo stato i verso altri stati. La quantità $\sum_{j \neq i} \pi_i(t)q_{ji}$ invece rappresenta il flusso di probabilità dagli altri stati verso i .

3.2.4 Catene di Markov irriducibili

Una catena di Markov è detta irriducibile se ogni stato è raggiungibile da qualsiasi altro stato, ovvero $p_{ij}(t) > 0$. Per un tempo illimitato si avrà quindi che la probabilità di transitare in uno stato j a partire da un qualsiasi altro stato i sarà indipendente dallo stato di partenza ???

$$\lim_{t \rightarrow \infty} p_{ij}(t) = p_j$$

3.2.5 Catene di Markov ergodiche

Una catena di Markov è detta ergodica se le probabilità di stato convergono ad un valore limite, indipendentemente dalla distribuzione iniziale:

$$\lim_{t \rightarrow \infty} \pi_j(t) = \pi_j = p_j$$

3.2.6 Soluzione alle equazioni di C-K per catene omogenee

Si vuole risolvere un'equazione del tipo $\frac{d\pi(t)}{dt} = \pi(t)A$, dove A è una matrice generica.

$$\frac{d\pi(t)}{dt} = \pi(t)Q(t)$$

per catene omogenee

Caso scalare

Nel caso in cui A è uno scalare ($A = a$) la soluzione è $\pi(t) = ae^{at}$ infatti $\frac{de^{at}}{dt} = ae^{at}$ dato che

$$\begin{aligned}\frac{de^{at}}{dt} &= \frac{d(1 + at + \frac{1}{2}(at)^2 + \dots + \frac{1}{k!}(at)^k + \dots)}{dt} = \\ &= \frac{0 + a + \frac{2}{2}(at)a + \dots + \frac{k}{k!}(at)^{k-1}a + \dots}{dt} = ae^{at}\end{aligned}$$

Caso matriciale

Nel caso in cui A è una matrice la soluzione è $\pi(t) = Ae^{At}$ (dove $e^{At} = I + At + \frac{1}{2!}A^2t^2 + \dots + \frac{1}{k!}A^kt^k + \dots = \sum_{k=0}^{\infty} \frac{A^k t^k}{k!}$), infatti si ha che:

$$\frac{de^{At}}{dt} = \frac{de^{At}}{d(At)} \frac{d(At)}{dt} = Ae^{At}$$

3.2.7 Catene di Markov omogenee in equilibrio

Le probabilità limite $\pi_j = p_j$ (ossia in condizione di equilibrio statistico) sono le soluzioni del sistema lineare:

$$\begin{cases} q_{jj}\pi_j + \sum_{k \neq j} q_{kj}\pi_k = 0 & \forall j \\ \sum_j \pi_j = 1 \end{cases} \Leftrightarrow \begin{cases} \pi Q = 0 \\ \sum_j \pi_j = 1 \end{cases} \quad \frac{d\pi(t)}{dt} = 0 = \pi Q$$

dove q_{ii} può essere calcolato come $q_{ii} = -\sum_{j \neq i} q_{ij} < 0$.

DENSISSIMA
DELLA
PROB. DI
STATO
E UNA

Ad esempio se si considera il caso di una catena di Markov con 3 stati si ottiene che:

$$\begin{cases} q_{00}\pi_0 + q_{10}\pi_1 + q_{20}\pi_2 = 0 \\ q_{11}\pi_1 + q_{01}\pi_0 + q_{21}\pi_2 = 0 \\ q_{22}\pi_2 + q_{02}\pi_0 + q_{12}\pi_1 = 0 \\ \pi_0 + \pi_1 + \pi_2 = 1 \end{cases} \quad \text{dove } \begin{cases} q_{00} = -q_{01} - q_{02} \\ q_{11} = -q_{10} - q_{12} \\ q_{22} = -q_{20} - q_{21} \end{cases}$$

PROB. DI STARE IN
UNO STATO DIPENDE
DALLE PROB. DI PASSARE
IN QUANTO PESO
POTER

$q_{00}\pi_0 + q_{10}\pi_1 + q_{20}\pi_2$
prob. di rimanere
nello stesso stato
e anche
nella

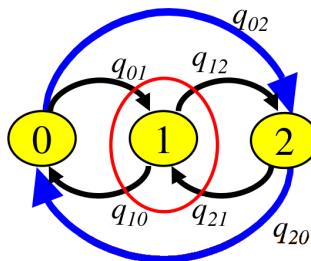


Figura 3.5: Esempio catena di Markov con 3 stati

3.3 Processi di nascita e morte

3.3.1 Definizione

Un processo di nascita e morte è una catena di Markov in cui sono permesse, da un generico stato j , soltanto transizioni verso gli stati $j+1$ (nascita) e $j-1$ (morte).

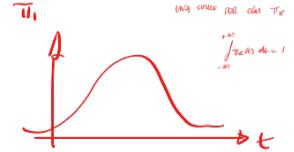
STATI DISSETTI
STATO FUTURO
SIA IN MIGRAZIONE

Nel caso di catena di Markov omogenea si definisce:

- $\lambda_k = q_{k,k+1}$ FREQUENZA NASCITA
- $\mu_k = q_{k,k-1}$ FREQUENZA MORTE

Si ha ovviamente che $q_{k,k} = -(\lambda_k + \mu_k)$, $q_{k,j} = 0$ per $|k-j| > 1$ e

$$Q = \begin{bmatrix} -\lambda_0 & \lambda_0 & 0 & 0 & \dots & \dots \\ \mu_1 & -(\lambda_1 + \mu_1) & \lambda_1 & 0 & \ddots & \vdots \\ 0 & \mu_2 & -(\lambda_2 + \mu_2) & \lambda_2 & 0 & \ddots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots \end{bmatrix}$$



3.3.2 Probabilità di stato

Si definisce

$$P_k(t) = \pi_k(t) = P\{X(t) = k\}$$

è possibile ricavare la distribuzione di probabilità risolvendo l'equazione C-K $\frac{d\pi^T(t)}{dt} = \pi^T(t)Q$, equivalente a

$$\begin{cases} \sum_{k=0}^{\infty} P_k(t) = 1 \\ \frac{dP_k(t)}{dt} = -(\lambda_k + \mu_k)P_k(t) + \lambda_{k-1}P_{k-1}(t) + \mu_{k+1}P_{k+1}(t) \\ \frac{dP_0(t)}{dt} = -\lambda_0P_0(t) + \mu_1P_1(t) \end{cases}$$

$$\begin{aligned} \text{Eq. 3.10} & \quad \text{Eq. 3.11} \\ \text{Eq. 3.12} & \quad \text{Eq. 3.13} \end{aligned}$$

è quindi necessario conoscere le condizioni iniziali $P_k(0) \forall k$. (eq. 3.14)

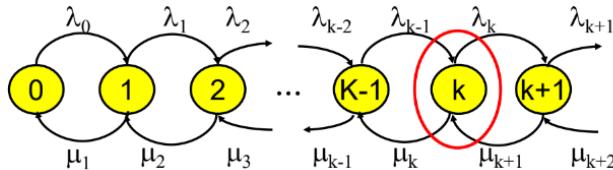


Figura 3.6: Processo di nascita e morte

Le precedenti equazioni possono essere ricavate ricordando che $\frac{dP_k(t)}{dt}$ è pari alla differenza tra il flusso di probabilità entrante nello stato k , ovvero $\lambda_{k-1}P_{k-1}(t) + \mu_{k+1}P_{k+1}(t)$, e il flusso uscente dallo stato k , ovvero $-(\lambda_k + \mu_k)P_k(t)$.

3.4 Processi di Poisson

Un **processo di Poisson** è una catena di Markov di pura nascita a tasso costante (λ).

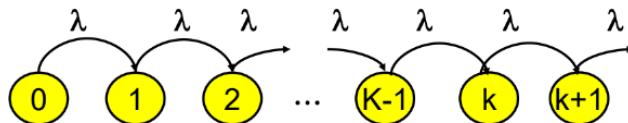


Figura 3.7: Processo di Poisson

I processi di Poisson vengono usati per modellare i processi di arrivo nelle code. Lo stato $X(t)$ rappresenta il numero di arrivi al tempo t e $P_k(t)$ rappresenta la probabilità che ci siano un numero k di arrivi nell'intervallo di tempo che va da 0 a t .

3.4.1 Probabilità di stato

In questo caso l'equazione C-K $\frac{d\pi^T(t)}{dt} = \pi^T(t)Q$ può essere scritta come

$$\begin{cases} \frac{dP_0(t)}{dt} = -\lambda P_0(t) \\ \frac{dP_k(t)}{dt} = -\lambda P_k(t) + \lambda P_{k-1}(t) \quad k \geq 1 \end{cases}$$

Se si suppone come condizione iniziale $P_0(0) = 1$, ovvero zero arrivi al tempo zero si ottiene la seguente soluzione:

$$\begin{cases} P_0(0) = 1 \\ \frac{dP_0(t)}{dt} = -\lambda P_0(t) \end{cases} \Rightarrow P_0(t) = e^{-\lambda t}$$

si ha quindi

$$\begin{cases} P_0(t) = e^{-\lambda t} \\ \frac{dP_1(t)}{dt} = -\lambda P_1(t) + \lambda P_0(t) \end{cases} \Rightarrow \frac{dP_1(t)}{dt} = -\lambda P_1(t) + \lambda e^{-\lambda t} \Rightarrow P_1(t) = \lambda t e^{-\lambda t}$$

Analogamente si ottiene:

$$P_2(t) = \frac{(\lambda t)^2}{2} e^{-\lambda t}$$

In generale quindi si ha che

$$P_k(t) = \frac{(\lambda t)^k}{k!} e^{-\lambda t} \quad \text{rossolana}$$

che rappresenta la probabilità di k arrivi in $[0, t]$.

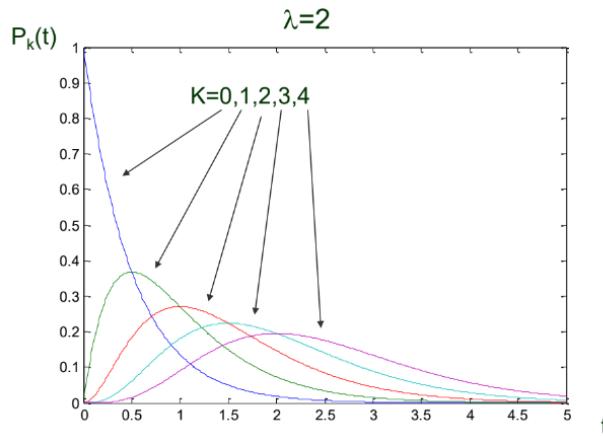


Figura 3.8: Distribuzione di Poisson

3.4.2 Caratterizzazione statistica degli stati

Per calcolare il numero medio di utenti che arrivano nell'intervallo $[0, t]$ è possibile calcolare il valore atteso di k :

$$\begin{aligned} E\{k\} &= \sum_{k=0}^{\infty} k \frac{(\lambda t)^k}{k!} e^{-\lambda t} = e^{-\lambda t} \sum_{k=0}^{\infty} k \frac{(\lambda t)^k}{k!} = \\ &= e^{-\lambda t} \sum_{k=1}^{\infty} \frac{(\lambda t)^k}{(k-1)!} = e^{-\lambda t} \lambda t \sum_{k=1}^{\infty} \frac{(\lambda t)^{k-1}}{(k-1)!} = e^{-\lambda t} \lambda t \sum_{j=0}^{\infty} \frac{(\lambda t)^j}{j!} = \lambda t \end{aligned}$$

La varianza invece è calcolabile come:

$$\begin{aligned} \sigma_k^2 &= E\{(k - E\{k\})^2\} = E\{k^2 - 2kE\{k\} + E\{k\}^2\} = \\ &= E\{k^2 + k - k - 2kE\{k\} + E\{k\}^2\} = E\{k(k-1)\} + E\{k\} - (E\{k\})^2 \end{aligned}$$

e dato che

$$E\{k(k-1)\} = \sum_{k=0}^{\infty} k(k-1) \frac{(\lambda t)^k}{k!} e^{-\lambda t} = e^{-\lambda t} (\lambda t)^2 \sum_{k=2}^{\infty} \frac{(\lambda t)^{k-1}}{(k-2)!} = e^{-\lambda t} (\lambda t)^2 \sum_{k=2}^{\infty} e^{\lambda t} = (\lambda t)^2$$

si ottiene

$$\sigma_k^2 = (\lambda t)^2 + \lambda t - (\lambda t)^2 = \lambda t$$

In questo caso varianza e valor medio coincidono.

3.4.3 Caratterizzazione statistica dei tempo di interarrivo

La probabilità che in un intervallo di tempo di durata t ci si trovi nello stato k è data da

$$P(X(s, s+t) = k) = P_k(t) = \frac{(\lambda t)^k}{k!} e^{-\lambda t} \quad k \geq 0, t \geq 0, \forall s$$

*una
mezz'ora / non un
valore*

Ricordando che la v.a. t_i rappresenta i tempi di interarrivo e $\tilde{t} = \lim_{n \rightarrow \infty} t_n$, allora la sua CDF sarà data da

$$A(t) = P\{\tilde{t} \leq t\} = 1 - P\{\tilde{t} > t\} = 1 - P_0(t)$$

dove $P\{\tilde{t} > t\}$ indica la probabilità che il primo arrivo avvenga dopo l'istante t , che corrisponde alla probabilità di trovarsi nello stato 0 al tempo t , ovvero $P_0(t)$.

Dato che la CDF del tempo di interarrivo è $A(t) = 1 - e^{-\lambda t}$ la sua pdf è $a(t) = \frac{dA(t)}{dt} = \lambda e^{-\lambda t}$. Si è già visto che il valore atteso di una tale v.a. è $E\{t\} = \frac{1}{\lambda}$ e quindi per quanto visto prima $\sigma_t = \frac{1}{\lambda}$.

Il tempo di interarrivo può essere interpretato anche come il tempo di permanenza in uno stato, dato che ad ogni arrivo si cambia stato. Una v.a. con una distribuzione esponenziale del tipo $a(t) = \lambda e^{-\lambda t}$ si dice essere senza memoria dato che il tempo trascorso nello stato non è utilizzabile per predire quanto si resterà ancora nello stato stesso. → Per mostrare ciò si supponga di fissare un istante di riferimento $t_0 = 0$ in corrispondenza di un arrivo. Se al tempo t_0 non vi è stato nessun altro arrivo, ci si chiede quale è la probabilità che il prossimo arrivo si verifichi dopo un tempo t a partire da t_0 . Si ha chiaramente che :

$$\begin{aligned} P\{\tilde{t} \leq \tilde{t} + t_0 | \tilde{t} > t_0\} &= \frac{P(A|B)}{P(B)} = \frac{P\{t_0 < \tilde{t} \leq \tilde{t} + t_0\}}{P(\tilde{t} > t_0)} = \\ &= \frac{P\{\tilde{t} \leq t + t_0\} - P\{\tilde{t} \leq t_0\}}{P\{\tilde{t} > t_0\}} = \\ &= \frac{A(t + t_0) - A(t_0)}{1 - A(t_0)} = \\ &= \frac{1 - e^{-\lambda(t+t_0)} - (1 - e^{-\lambda(t_0)})}{1 - (1 - e^{-\lambda(t_0)})} = \\ &= \frac{-e^{-\lambda(t+t_0)} + e^{-\lambda(t_0)}}{e^{-\lambda(t_0)}} = \\ &= 1 - e^{-\lambda(t)} = A(t) \end{aligned}$$

La quantità $P\{\tilde{t} \leq t + t_0 | \tilde{t} > t_0\}$ rappresenta la CDF del tempo di permanenza condizionata dal fatto che l'ultimo arrivo è avvenuto all'istante t_0 . Dato che tale CDF non dipende da t_0 risulta evidente che le proprietà statistiche di tale v.a. non sono influenzate dalla conoscenza dell'ultimo arrivo. Vale quindi la proprietà memoryless. Si può dimostrare che la distribuzione esponenziale è l'unica distribuzione che presenta tale proprietà.

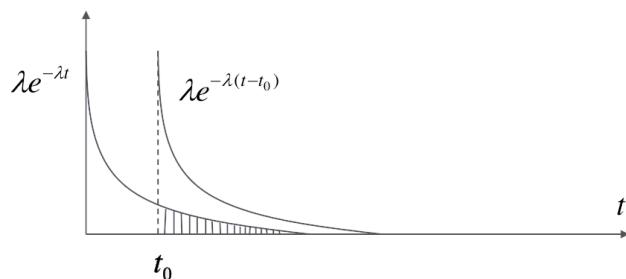


Figura 3.9: Proprietà memoryless

3.4.4 Probabilità di k eventi

Si vuole calcolare la probabilità di avere k arrivi nell'intervallo $[0, t]$. Per fare ciò si considera un intervallo $[0, t]$ e si suddivide tale intervallo in sottointervalli generici di durata α_i , senza arrivi, e in intervalli di durata β_i , con un singolo arrivo. La durata effettiva degli intervalli non è significativa a causa della proprietà memoryless.

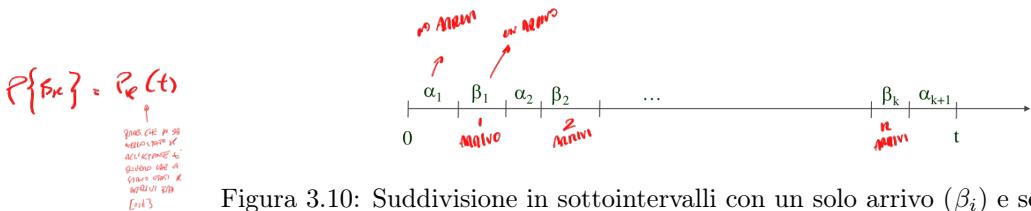
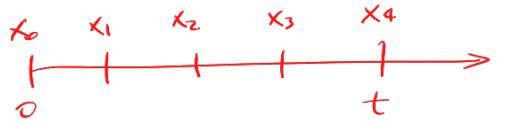


Figura 3.10: Suddivisione in sottointervalli con un solo arrivo (β_i) e senza arrivi (α_i)

Si definisce A_k come l'evento di singolo arrivo negli intervalli β_i e nessun arrivo negli intervalli α_i . Si definisce invece B_k come l'evento di k arrivi nell'intervallo $[0, t]$, si ha $P\{B_k\} = P_k(t) = \frac{(\lambda t)^k}{k!} e^{-\lambda t}$. La probabilità che non ci sia nessun arrivo in un intervallo di durata α_i è $P_0(\alpha_i) = e^{-\lambda \alpha_i}$. La probabilità che ci sia 1 arrivo in un intervallo di durata β_i è invece $P_1(\beta_i) = \lambda \beta_i e^{-\lambda \beta_i}$. È quindi possibile dire che

$$\begin{aligned} P\{A_k | B_k\} &= \frac{P\{A_k \cap B_k\}}{P\{B_k\}} = \frac{P_0(\alpha_1)P_1(\beta_1)P_0(\alpha_2)P_1(\beta_2)\dots P_0(\alpha_k)P_1(\beta_k)P_0(\alpha_{k+1})}{P\{B_k\}} = \\ &= \frac{(e^{-\lambda \alpha_1})(\lambda \beta_1 e^{-\lambda \beta_1})(e^{-\lambda \alpha_2})(\lambda \beta_2 e^{-\lambda \beta_2})\dots(e^{-\lambda \alpha_k})(\lambda \beta_k e^{-\lambda \beta_k})(e^{-\lambda \alpha_{k+1}})}{\frac{(\lambda t)^k}{k!} e^{-\lambda t}} = \\ &= \frac{\lambda^k (\beta_1 \beta_2 \dots \beta_k) e^{-\lambda(\alpha_1 + \beta_1 + \dots + \alpha_k + \beta_k + \alpha_{k+1})}}{\frac{(\lambda t)^k}{k!} e^{-\lambda t}} \quad \alpha_1 + \beta_1 + \dots + \alpha_k + \beta_k + \alpha_{k+1} = t \\ &= \cancel{\frac{\lambda^k (\beta_1 \beta_2 \dots \beta_k) e^{-\lambda t}}{\frac{(\lambda t)^k}{k!} e^{-\lambda t}}} \end{aligned}$$



Distribuendo aleatoriamente k punti nell'intervallo $[0, t]$ con distribuzione uniforme si ottiene la probabilità

$$\left(\frac{\beta_1}{t} \frac{\beta_2}{t} \frac{\beta_3}{t} \dots \frac{\beta_k}{t} \right) k!$$

$$\left(\frac{1}{5} \cdot \frac{2}{5} \cdot \frac{3}{5} \right) 3! \quad \frac{1}{5} \cdot \frac{2}{5} \cdot \frac{3}{5}$$

che è equivalente a $P\{A_k | B_k\}$ e siccome le due probabilità coincidono ne conseguì che dati k arrivi in $[0, t]$, se questi sono generati da un processo di Poisson, allora saranno distribuiti uniformemente nell'intervallo $[0, t]$.

3.4.5 Distribuzione di Erlang

Si è visto che la distribuzione dei tempi di interarrivo è data da $a(t) = \lambda e^{-\lambda t}$. Si vuole ora individuare la distribuzione di probabilità del tempo di interarrivo di k arrivi consecutivi. Per fare ciò si suppone di accumulare k arrivi di un processo di Poisson. Il tempo di arrivo w di k arrivi sarà una v.a. data dalla somma di k v.a. t_i con distribuzione $a(t)$:

$$w = t_1 + t_2 + \dots + t_k$$

La distribuzione di w è quindi la distribuzione di interesse. Per ricavare tale distribuzione si ricorda che la distribuzione di una somma di v.a. è pari alla convoluzione delle distribuzioni delle singole v.a., si ha quindi:

$$f_w(t) = \underbrace{a(t) * a(t) * \dots * a(t)}_{k \text{ volte}}$$

Ricordando inoltre che la convoluzione nel tempo diventa un prodotto in Laplace e ricordando che $\mathcal{L}(a(t)) = \mathcal{L}(\lambda e^{-\lambda t}) = \frac{\lambda}{s+\lambda}$ si ottiene

$$\mathcal{L}(f_w(t)) = \left(\frac{\lambda}{s+\lambda} \right)^k$$

e antitrasformando si ottiene

$$f_w(t) = \frac{\lambda(\lambda t)^{k-1}}{(k-1)!} e^{-\lambda t} \quad \text{ERLANGIANA}$$

3.5 Sistemi di servizio

Un sistema di servizio è caratterizzato da n sorgenti di traffico, una coda di lunghezza L e m serventi che elaborano le richieste.

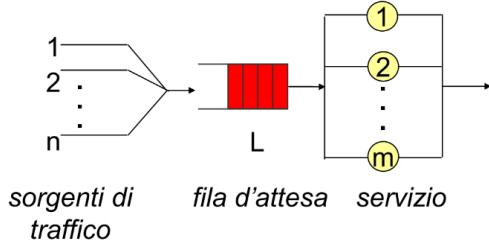


Figura 3.11: Sistema di servizio

Il sistema è descritto dalle v.a.:

- k : numero di utenti nel sistema;
- l : numero di utenti nella coda;
- h : numero di serventi contemporaneamente occupati;
- x : tempo di servizio;
- s : tempo di permanenza nel sistema;
- w : tempo di permanenza nella coda;

La variabile aleatoria k è caratterizzata attraverso la sua probabilità limite $\pi_k = p_k$, ovvero la probabilità che in un generico istante di osservazione in regime permanente siano presenti k utenti (richieste di servizio) all'interno del sistema.

3.5.1 Parametri prestazionali

La probabilità di **sistema bloccato** con m serventi è data da:

$$S_p = P\{k = L + m\} = p_{L+m}$$

La probabilità di **rifiuto** data una richiesta di servizio offerto (r.s.o.) è:

$$\Pi_p = P\{\text{sistema Bloccato} | \text{r.s.o.}\} = \frac{P\{A|B\}}{P\{B\}} = \frac{P\{A\}P\{B|A\}}{P\{B\}} = S_p \frac{P\{\text{r.s.o.} | \text{sistema Bloccato}\}}{P\{\text{r.s.o.}\}}$$

p{ r.s.o }

sistema non bloccato

dato che $P\{\text{r.s.o.} | \text{sistema Bloccato}\} = P\{\text{r.s.o.}\}$ se i due eventi sono indipendenti e in questo caso lo sono perché una richiesta di servizio non dipende dallo stato del sistema. Quindi la probabilità di rifiuto coincide con la probabilità di sistema bloccato.

La probabilità di **servizio bloccato** con m serventi è: (SERVENTI VINTI OCCUPATI)

$$S_r = P\{k \geq m\}$$

La probabilità di **ritardo** data una richiesta di servizio accolta (r.s.a.) è:

$$\Pi_r = P\{\text{servizio Bloccato} | \text{r.s.a.}\} = S_r \frac{P\{\text{r.s.a.} | \text{servizio Bloccato}\}}{P\{\text{r.s.a.}\}} = S_r$$

3.5.2 Processi di nascita e morte in equilibrio statistico

Un processo di nascita e morte in equilibrio statistico è descritto dalle equazioni:

$$\begin{cases} \sum_{k=0}^{\infty} P_k = 1 \\ 0 = -(\lambda_k + \mu_k)P_k + \lambda_{k-1}P_{k-1} + \mu_{k+1}P_{k+1} \quad \forall k \geq 1 \\ 0 = -\lambda_0P_0 + \mu_1P_1 \end{cases}$$

caso siamo 0



$$\lambda_0P_0 = \mu_1P_1$$

Dalla seconda è possibile dire che

$$\begin{aligned}\mu_{k+1}P_{k+1} - \lambda_k P_k &= \mu_k P_k - \lambda_{k-1} P_{k-1} \\ \mu_k P_k - \lambda_{k-1} P_{k-1} &= \mu_{k-1} P_{k-1} - \lambda_{k-2} P_{k-2}\end{aligned}$$

⋮

$$\mu_3 P_3 - \lambda_2 P_2 = \mu_2 P_2 - \lambda_1 P_1$$

$$\mu_2 P_2 - \lambda_1 P_1 = \mu_1 P_1 - \lambda_0 P_0 = 0$$

ovvero si ha che $\alpha_k = \mu_k P_k - \lambda_{k-1} P_{k-1} = 0$ dato che $\mu_1 P_1 - \lambda_0 P_0 = 0$.

Dato che $\mu_k P_k - \lambda_{k-1} P_{k-1} = 0$ si ha

$$P_k = \frac{\lambda_{k-1}}{\mu_k} P_{k-1}$$

e sostituendo ricorsivamente P_k si ottiene

$$P_k = \frac{\lambda_{k-1}}{\mu_k} \frac{\lambda_{k-2}}{\mu_{k-1}} P_{k-2} = \frac{\lambda_{k-1}}{\mu_k} \frac{\lambda_{k-2}}{\mu_{k-1}} \frac{\lambda_{k-3}}{\mu_{k-2}} P_{k-3} = \dots = \frac{\lambda_{k-1}}{\mu_k} \frac{\lambda_{k-2}}{\mu_{k-1}} \frac{\lambda_{k-3}}{\mu_{k-2}} \dots \frac{\lambda_0}{\mu_1} P_0 = P_0 \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}$$

Dalla nota proprietà della probabilità si ha

$$P_0 + \sum_{k=1}^{\infty} P_k = 1$$

Probabile così?
Voglio sommarmi per tutti i punti
perché sono finiti?

e sostituendo $P_k = P_0 \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}$ si ottiene

$$P_0 + P_0 \sum_{k=1}^{\infty} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}} = 1 \Rightarrow P_0 = \frac{1}{1 + \sum_{k=1}^{\infty} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}}$$

Individuando il denominatore *della* *sistema*

3.6 Sistema a coda $M/M/1/\infty/\infty$

Serventi *Cittadini*

Un sistema a coda $M/M/1/\infty/\infty$ presenta:

- M : processo degli arrivi markoviano;
- M : processo dei serventi markoviano;
- 1: numero serventi;
- ∞ : dimensione del sistema;
- ∞ : cardinalità popolazione;

Si ipotizza che:

- processo degli arrivi di Poisson i.i.d. con parametro λ ;
- processo di servizio di Poisson i.i.d. con 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 $\{0, 1, 2, \dots\}$. Il processo di coda $K(t)$ è ergodico se $\frac{\lambda}{\mu} < 1$.

versante positiva

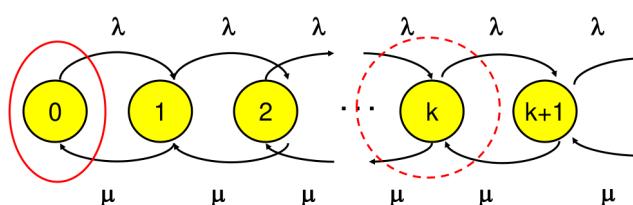


Figura 3.12: Sistema a coda $M/M/1/\infty/\infty$

3.6.1 Probabilità limite di stato

Si è visto che $P_k = P_0 \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}$ dove $P_0 = \frac{1}{1 + \sum_{k=1}^{\infty} \frac{\lambda_k}{\prod_{i=0}^{k-1} \mu_{i+1}}}$, ma per ipotesi $\lambda_k = \lambda$ e $\mu_k = \mu$, quindi:

$$P_0 = \frac{1}{1 + \sum_{k=1}^{\infty} \rho^k} = 1 - \rho \quad \text{dove } \rho = \frac{\lambda}{\mu} < 1$$

perché $\sum_{k=1}^{\infty} \rho^k = \frac{1}{1-\rho}$ (serie geometrica).

Quindi

$$P_k = P_0 \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}} = P_0 \rho^k = (1 - \rho) \rho^k$$

ovvero si ha una distribuzione geometrica.

3.6.2 Numero medio di utenti nel sistema

Il numero medio di utenti nel sistema è

$$\begin{aligned} E\{K\} = \bar{k} &= \sum_{k=0}^{\infty} kp_k = \sum_{k=0}^{\infty} k(1 - \rho)\rho^k = (1 - \rho) \sum_{k=0}^{\infty} k\rho^k = \\ &= (1 - \rho)\rho \sum_{k=0}^{\infty} k\rho^{k-1} = (1 - \rho)\rho \sum_{k=1}^{\infty} k\rho^{k-1} = (1 - \rho)\rho \left(\frac{\partial}{\partial \rho} \sum_{k=0}^{\infty} \rho^k \right) = \\ &= (1 - \rho)\rho \left(\frac{\partial}{\partial \rho} \frac{1}{1 - \rho} \right) = \frac{\rho}{1 - \rho} \end{aligned}$$

3.6.3 Tempo di permanenza nel sistema

Per la legge di Little si ha $\bar{k} = T\lambda$ e quindi

$$T = \frac{\bar{k}}{\lambda} = \frac{\rho}{\lambda(1 - \rho)} = \frac{1}{\mu(1 - \rho)} = \frac{1}{\mu - \lambda}$$

3.6.4 Parametri prestazionali λ, μ costanti

In condizioni di equilibrio statistico l'intensità media di richieste smaltite A_S coincide con l'intensità di richieste di servizio offerte A_O :

$$A_S = A_O = \rho = 1 - P_0$$

La probabilità di servizio bloccato S_r coincide con la probabilità di ritardo nel ricevere servizio Π_r :

$$S_r = \Pi_r = (1 - \rho) \sum_{k=1}^{\infty} \rho^k = \rho$$

Quindi la probabilità che il servente sia occupato è uguale alla probabilità che una richiesta in arrivo sia costretta ad attendere in coda, ovvero ρ .

3.6.5 Distribuzione in equilibrio statistico

Se l è il numero di utenti nella fila d'attesa e ricordando che la probabilità di avere k utenti nel sistema è $P_k = (1 - \rho)\rho^k$ si ha che la probabilità di avere 0 utenti in coda nel caso di 1 servente è pari a $P_0 + P_1 = (1 - \rho) + \rho(1 - \rho)$ (servente libero o servente occupato), e la probabilità di avere j utenti in coda è pari alla probabilità di avere $j + 1$ utenti nel sistema:

$$P\{l = j\} = \begin{cases} (1 - \rho) + \rho(1 - \rho) = 1 - \rho^2 & j = 0 \\ (1 - \rho)\rho^{j+1} & j \geq 1 \end{cases}$$

Il numero di medio di utenti in coda è

$$\bar{l} = \frac{\rho^2}{1 - \rho}$$

Si vuole calcolare la probabilità di avere j serventi impegnati. Se si indica con h il numero di serventi impegnati si ha $P\{h = 0\} = P_0$ e $P\{h = 1\} = 1 - P_0$ ovvero:

$$P\{h = j\} = \begin{cases} 1 - \rho & j = 0 \\ \rho & j = 1 \end{cases}$$

Il numero medio di serventi impegnati è

$$\bar{h} = \rho$$

Il numero medio di utenti nel sistema è

$$\bar{k} = \bar{l} + \bar{h} = \frac{\rho}{1 - \rho}$$

in conformità con quanto calcolato in precedenza.

3.6.6 Tempo di attesa in coda

Si supponga che un utente trovi, al suo arrivo, il sistema nello stato k , ciò avviene con probabilità $\rho^k(1 - \rho)$. In questo caso, se la coda è gestita con disciplina FIFO, l'utente dovrà attendere un tempo pari alla somma di k v.a. esponenziali.

Se $w = t_1 + t_2 + \dots + t_k$ e $t_i \sim a(t) = \lambda e^{-\lambda t}$ allora $w \sim \underbrace{\lambda e^{-\lambda t} * \lambda e^{-\lambda t} * \dots * \lambda e^{-\lambda t}}_{k \text{ volte}}$. E quindi

$$p_{w|k}(t) = P\{\text{tempo di attesa } w | k \text{ utenti}\}(t) = \underbrace{\lambda e^{-\lambda t} * \lambda e^{-\lambda t} * \dots * \lambda e^{-\lambda t}}_{k \text{ volte}}$$

$$p_w(t) = \sum_{k=0}^{\infty} p_{w|k}(t) p_k(t) = \sum_{k=0}^{\infty} \underbrace{\lambda e^{-\lambda t} * \lambda e^{-\lambda t} * \dots * \lambda e^{-\lambda t}}_{k \text{ volte}} \rho^k (1 - \rho)$$

$$W(s) = \mathcal{L}(p_w(t)) = \mathcal{L}\left(\sum_{k=0}^{\infty} p_{w|k} p_k\right) = \mathcal{L}\left(\sum_{k=0}^{\infty} \underbrace{(\lambda e^{-\lambda t} * \lambda e^{-\lambda t} * \dots * \lambda e^{-\lambda t})}_{k \text{ volte}} \rho^k (1 - \rho)\right) = \sum_{k=0}^{\infty} \left(\frac{\mu}{s + \mu}\right)^k \rho^k (1 - \rho)$$

e quindi

$$\begin{aligned} W(s) &= \sum_{k=0}^{\infty} \left(\frac{\mu}{s + \mu}\right)^k \rho^k (1 - \rho) = (1 - \rho) \frac{1}{1 - \frac{\mu}{s + \mu} \rho} = (1 - \rho) \frac{s + \mu}{s + \mu - \mu \rho} = \\ &= \frac{(1 - \rho)(s + \mu + \lambda - \lambda)}{s + \mu(1 - \rho)} = (1 - \rho) + \frac{\lambda(1 - \rho)}{s + \mu(1 - \rho)} \end{aligned}$$

per cui

$$w(t) = (1 - \rho)\delta(t) + \lambda(1 - \rho)e^{-\mu(1-\rho)t}$$

La CDF è

$$F_w(t) = P\{w \leq t\} = \int_0^t w(t) dt = 1 - \rho e^{-(1-\rho)\mu t}$$

Il tempo di attesa medio è

$$W = \frac{\rho}{\mu} \frac{1}{1 - \rho}$$

3.6.7 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 (k in coda + il tempo di servizio di esso stesso).

La trasformata di Laplace della distribuzione del tempo di permanenza del sistema è quindi

$$S(s) = \sum_{k=0}^{\infty} \left(\frac{\mu}{s + \mu}\right)^{k+1} \rho^k (1 - \rho) = \frac{\mu}{s + \mu} (1 - \rho) \frac{1}{1 - \frac{\mu}{s + \mu} \rho} = \frac{\mu(1 - \rho)}{s + \mu(1 - \rho)}$$

ROZ. ATTESA IN = VARI
 ATTESA IN = VARI
 ROZ. SERVIZIO
 NELLO STATO K
 (se il tempo di servizio
non dipende dal
tempo)

e quindi antitrasformando

$$s(t) = \mu(1 - \rho)e^{-\mu(1-\rho)t}$$

Il tempo medio di permanenza nel sistema è

$$\textcircled{T} = \frac{1}{\mu(1 - \rho)} = \frac{1}{\mu - \lambda} = \textcircled{\bar{w}} + \frac{1}{\mu}$$

3.7 Sistema a coda $M/M/\infty$

Un sistema a coda $M/M/\infty$ presenta:

- \textcircled{M} : processo degli arrivi markoviano; *processo degli arrivi in poisson con rate λ*
- \textcircled{M} : processo dei serventi markoviano;
- $\textcircled{\infty}$: numero serventi; *rate di servizio scalabile*
- $\textcircled{\infty}$: dimensione del sistema;
- $\textcircled{\infty}$: cardinalità popolazione;

Tale modello si adatta molto bene a descrivere il FaaS dato che:

- l'infrastruttura serverless è progettata per scalare automaticamente in modo teoricamente illimitato in base al numero di richieste;
- ogni richiesta può essere servita immediatamente senza dover attendere in coda;
- in molte applicazioni distribuite, il traffico delle richieste si comporta come un flusso casuale (Poisson);
- tempi di esecuzione delle funzioni FaaS spesso seguono una distribuzione esponenziale;

Nel sistema a coda $M/M/\infty$ si ha $\lambda_k = \lambda$ e $\mu_k = k\mu$.

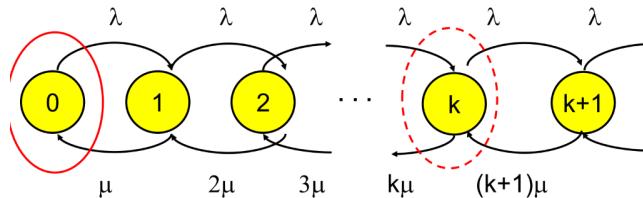


Figura 3.13: Sistema a coda $M/M/\infty$

In questo caso per avere una condizione di stabilità è sufficiente avere $\frac{\lambda}{\mu} < \infty$ e non $\frac{\lambda}{\mu} < 1$ dato che il numero di serventi è sempre sufficiente al numero di richieste (scaling).

3.7.1 Probabilità limite di stato

Si è visto che $P_k = P_0 \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}$ dove $P_0 = \frac{1}{1 + \sum_{k=1}^{\infty} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}}$, ma per ipotesi $\lambda_k = \lambda$ e $\mu_k = k\mu$, quindi:

$$P_0 = \frac{1}{1 + \sum_{k=1}^{\infty} \prod_{i=0}^{k-1} \frac{\lambda}{(i+1)\mu}} = \frac{1}{1 + \sum_{k=1}^{\infty} \left(\frac{\lambda}{\mu}\right)^k \frac{1}{k!}} = \frac{1}{\sum_{k=0}^{\infty} \left(\frac{\lambda}{\mu}\right)^k \frac{1}{k!}}$$

dato che $e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$.

Quindi

$$\textcircled{P_k} = P_0 \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}} = e^{-\frac{\lambda}{\mu}} \prod_{i=0}^{k-1} \frac{\lambda}{k\mu} = \boxed{e^{-\frac{\lambda}{\mu}} \left(\frac{\lambda}{\mu}\right)^k \frac{1}{k!}}$$

La distribuzione di probabilità degli stati è quindi una distribuzione di Poisson $\frac{(\lambda t)^k}{k!} e^{-\lambda t}$ valutata per $t = \frac{1}{\mu}$.

Si ha quindi che li numero medio di utenti nel sistema è $\textcircled{\bar{N}} = \lambda T = \textcircled{\frac{\lambda}{\mu}}$ e il tempo medio di permanenza nel sistema è $\textcircled{T} = \frac{1}{\mu}$ (non c'è nessuna coda).

3.8 Sistema a coda $M/M/m$

Un sistema a coda $M/M/m$ presenta:

- M : processo degli arrivi markoviano;
- M : processo dei serventi markoviano;
- m : numero serventi;
- ∞ : dimensione del sistema;
- ∞ : cardinalità popolazione;

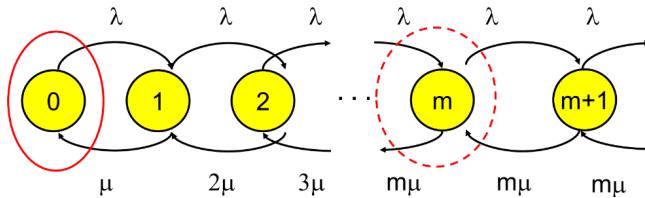


Figura 3.14: Sistema a coda $M/M/m$

In questo caso per avere una condizione di stabilità è sufficiente avere $\frac{\lambda}{m\mu} < 1$ dato che il numero di serventi è sufficiente al numero di richieste (scaling) fino ad un massimo di m .

3.8.1 Probabilità limite di stato

$$\text{RICORDA: CASO } m \text{ SERVENTI} \rightarrow \rho = \frac{\lambda}{m\mu}$$

TABELLA DI SVILUPPO DEL COEFFICIENTE

Per ipotesi $\lambda_k = \lambda$ e $\mu_k = \begin{cases} k\mu & k \leq m \\ m\mu & k > m \end{cases}$, quindi si ha:

$$P_k = P_0 \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}} = P_0 \prod_{i=0}^{k-1} \frac{\lambda}{(i+1)\mu} = P_0 \left(\frac{\lambda}{\mu}\right)^k \frac{1}{k!} = P_0 \frac{(m\rho)^k}{k!} \quad k \leq m$$

$$P_k = P_0 \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}} = P_0 \prod_{i=0}^{m-1} \frac{\lambda}{(i+1)\mu} \prod_{i=m}^{k-1} \frac{\lambda}{m\mu} = P_0 \left(\frac{\lambda}{\mu}\right)^k \frac{1}{m!m^{k-m}} = P_0 \frac{\rho^k m^m}{m!} \quad k > m$$

✓ BUNDO 30
 ✓ RICORDA: $\lambda = \lambda_0 + \lambda_1 + \dots + \lambda_m$
 ✓ RICORDA: $\mu = \mu_0 + \mu_1 + \dots + \mu_m$
 ✓ RICORDA: $\lambda = m\rho$

dove

$$P_0 = \frac{1}{1 + \sum_{k=1}^{\infty} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}} = \frac{1}{1 + \sum_{k=1}^{m-1} \left(\frac{\lambda}{\mu}\right)^k \frac{1}{k!} + \sum_{k=m}^{\infty} \left(\frac{\lambda}{\mu}\right)^k \frac{1}{m!m^{k-m}}} =$$

$$= \frac{1}{1 + \sum_{k=1}^{m-1} \frac{(m\rho)^k}{k!} + \sum_{k=m}^{\infty} \frac{(m\rho)^k}{m!} \frac{1}{m^{k-m}}} = \frac{1}{\sum_{k=0}^{m-1} \left[\frac{(m\rho)^k}{k!}\right] + \frac{(m\rho)^m}{m!} \frac{1}{1-\rho}}$$

3.8.2 Formula C di Erlang

Un utente che arriva in ingresso al sistema dovrà accodarsi se il numero di utenti nel sistema (k) sarà almeno pari al numero di serventi (m). La probabilità che ciò accada è quindi data da:

$$P\{\text{accodamento}\} = \sum_{k=m}^{\infty} P_k = \sum_{k=m}^{\infty} P_0 \frac{(m\rho)^k}{m!} \frac{1}{m^{k-m}} = P_0 \frac{(m\rho)^m}{m!} \frac{1}{1-\rho}$$

Già CALCOLATA DALLA
 FORMULA PRECEDENTE
 → $\frac{(m\rho)^m}{m!} \frac{1}{1-\rho}$

e sostituendo P_0 si ottiene la **formula C di Erlang**

$$P\{\text{accodamento}\} = C(m, \rho) = \frac{\frac{(m\rho)^m}{m!} \frac{1}{1-\rho}}{\sum_{k=0}^{m-1} \frac{(m\rho)^k}{k!} + \frac{(m\rho)^m}{m!} \frac{1}{1-\rho}}$$

Tale formula è utilizzata per determinare la probabilità di attesa nell'accesso ad un sistema con m serventi.

3.9 Sistema a coda $M/M/m/m/\infty$

Un sistema a coda $M/M/m/m/\infty$ presenta:

- M : processo degli arrivi markoviano;
- M : processo dei serventi markoviano;
- m : numero serventi;
- m : dimensione del sistema, la coda quindi non c'è;
- ∞ : cardinalità popolazione;

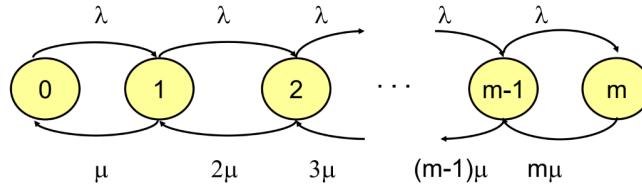


Figura 3.15: Sistema a coda $M/M/m/m/\infty$

Il processo di coda è descrivibile mediante un processo di Markov di nascita e morte con spazio di stato $\{0, \dots, m\}$.

Tale modello si adatta a descrivere un server DHCP dato che il parametro m rappresenta il numero di indirizzi IP disponibili per l'assegnazione. In un server DHCP, questo corrisponde al numero di indirizzi IP configurati per essere distribuiti (il pool di indirizzi). Una volta che tutti gli m indirizzi IP sono assegnati, il sistema non può più accettare nuove richieste finché un indirizzo non viene rilasciato.

3.9.1 Probabilità limite di stato

$$P_k = P_0 \left(\frac{\lambda}{\mu} \right)^k \frac{1}{k!} = P_0 A_o^k \frac{1}{k!} \quad 0 < k \leq m$$

dove

$$P_0 = \frac{1}{\sum_{j=0}^m A_o^j \frac{1}{j!}}$$

e $A_o = \frac{\lambda}{\mu}$ è l'intensità di traffico offerto al sistema.

Quindi

$$P_k = \frac{A_o^k \frac{1}{k!}}{\sum_{j=0}^m A_o^j \frac{1}{j!}}$$

$$P_0 = \frac{1}{1 + \sum_{k=1}^{\infty} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}}$$

$$\frac{1}{1 + \frac{\lambda_0}{\mu_1} + \frac{\lambda_0 \lambda_1}{\mu_1 \mu_2} + \frac{\lambda_0 \lambda_1 \lambda_2}{\mu_1 \mu_2 \mu_3} + \dots} \rightarrow \frac{\lambda_0}{\mu_1} \frac{\lambda_1}{\mu_2} \frac{\lambda_2}{\mu_3} \dots$$

3.9.2 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:

$$\Pi_p = S_p \frac{P\{r.s.o. | \text{sistema Bloccato}\}}{P\{r.s.o.\}} = S_p$$

Ovvero la probabilità di sistema bloccato (S_p) è uguale alla probabilità di rifiuto (Π_p). Si ha un rifiuto quando lo stato è $k = m$:

$$\Pi_p = S_p = \frac{A_o^m \frac{1}{m!}}{\sum_{j=0}^m A_o^j \frac{1}{j!}}$$

che si chiama **formula B di Erlang**.

La formula B di Erlang può essere calcolata in modo ricorsivo:

$$E_{1,m}(A_o) = \frac{A_o^m \frac{1}{m!}}{\sum_{j=0}^m A_o^j \frac{1}{j!}} = \frac{A_o E_{1,m-1}(A_o)}{m + A_o E_{1,m-1}(A_o)}$$

con il primo termine pari a:

$$E_{1,1}(A_o) = \frac{A_o}{1 + A_o}$$

E' VALIDA PER SVARSI DISTURZ. DEL TIEMPO DI SERVIZIO ! Δ
(DEVONO ESSERE INI)

3.9.3 Parametri prestazionali

L'intensità media di traffico o lavoro smaltito A_s , rappresenta il numero medio di serventi contemporaneamente occupati, dipende da A_o e dal numero di serventi m :

$$A_{s_{\text{med}}} = \sum_{k=1}^m k P_k = A_o (1 - E_{1,m}(A_o))$$

↓
 probabilità di un
servente
occupato

↑
 traffico
offerto
al sistema

↓
 probabilità di accettazione
di un servente

→
 valore medio dei
numero di serventi
che si trovano sotto
di un certo dato

L'intensità media di traffico rifiutato è quindi data da:

$$A_p = A_o - A_s = A_o E_{1,m}(A_o)$$

Il coefficiente di utilizzazione del servente è dato da:

$$\rho = \frac{A_s}{m} = \frac{A_o}{m} (1 - E_{1,m}(A_o))$$

3.9.4 Dimensionamento del sistema

Dimensionare un sistema significa determinare il numero di serventi m necessari affinché la probabilità di rifiuto sia minore di Π_{max} con un traffico offerto stimato A_o . Per fare ciò è sufficiente trovare il più piccolo valore di m tale per cui

$$E_{1,m}(A_o) \leq \Pi_{max}$$

Tale valore può essere facilmente determinato per tentativi a partire da $m = 1$.

È possibile stimare A_o a partire dal traffico smaltito A_s e il numero di serventi m :

$$A_o (1 - E_{1,m}(A_o)) = A_s$$

Noto A_o è possibile calcolare la probabilità di rifiuto:

$$\Pi_p = E_{1,m}(A_o)$$

3.10 Sistema a coda $M/M/1/K$

Il sistema $M/M/1/K$ è un sistema orientato al ritardo con una coda finita e presenta:

- M : processo degli arrivi markoviano;
- M : processo dei serventi markoviano;
- 1: numero serventi;
- K : dimensione del sistema, la coda ha dimensione $K - 1$;
- ∞ : cardinalità popolazione;

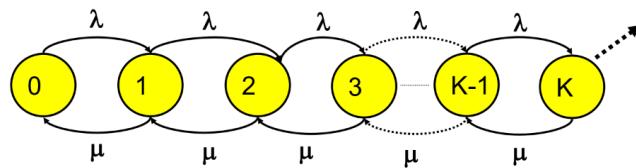


Figura 3.16: Sistema $M/M/1/K$

3.10.1 Probabilità limite

Chiamando $A_o = \frac{\lambda}{\mu}$ si ha:

$$P_k = P_0 \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}$$

83

$\frac{\lambda_0}{\mu_1} \cdot \frac{\lambda_1}{\mu_2} \cdot \frac{\lambda_2}{\mu_3} \cdot \frac{\lambda_3}{\mu_4} \cdot \frac{\lambda_4}{\mu_5} \cdots \rightarrow \frac{\lambda_0 \lambda_1 \lambda_2 \lambda_3 \lambda_4}{\mu_1 \mu_2 \mu_3 \mu_4 \mu_5} = \left(\frac{\lambda}{\mu} \right)^{k-1}$
 $\lambda = \lambda$
 $\mu = \mu$

dove

$$\begin{aligned}
 P_0 &= \frac{1}{1 + \left(\sum_{j=1}^{\infty} A_o^j - \sum_{i=K+1}^{\infty} A_o^i \right)} = \frac{1}{1 - \frac{A_o^{K+1}}{1 - A_o}} = \frac{1 - A_o}{1 - A_o^{K+1}} \\
 &= \frac{1 - \frac{A_o}{1 - A_o} - \frac{A_o^{K+1}}{1 - A_o}}{1 + \left(\frac{A_o}{1 - A_o} - \frac{A_o^{K+1}}{1 - A_o} \right)} = \frac{1 - A_o}{1 - A_o^{K+1}}
 \end{aligned}$$

$$P_0 = \frac{1}{1 + \sum_{k=1}^{\infty} \prod_{i=0}^{k-1} \frac{\lambda_i}{\mu_{i+1}}}$$

e quindi

$$P_k = \begin{cases} \frac{1 - A_o}{1 - A_o^{K+1}} A_o^k & 0 \leq k \leq K \\ 0 & k > K \end{cases}$$

3.11 Proprietà PASTA

Theorem 3.11.1. In un sistema a coda che soddisfa l'ipotesi LAA (Lack of Anticipation: futuri tempi di interarrivo e i tempi di servizio dei clienti arrivati precedentemente sono indipendenti), se il processo degli arrivi è di Poisson allora:

$$a_n = p_n$$

dove $p_n = \lim_{t \rightarrow \infty} P\{N(t) = n\}$ e $a_n = \lim_{t \rightarrow \infty} P\{N(t^-) = n \mid \text{arrival at } t\}$

Dimostrazione. Si chiama $A(t, t + \delta)$ l'evento in cui c'è un arrivo in $[t, t + \delta]$.

Noto che un utente arriva al tempo t , la probabilità $a_n(t)$ di trovare il sistema nello stato n è data da:

$$a_n(t) = P\{N(t^-) = n \mid \text{arrival at } t\} = \lim_{\delta \rightarrow 0} P\{N(t^-) = n \mid A(t, t + \delta)\}$$

$N(t^-)$ è determinato dai tempi di arrivo $< t$ e dai corrispondenti tempi di servizio. Per cui se il processo degli arrivi è di Poisson $A(t, t + \delta)$ è indipendente dagli arrivi $< t$ e se vale la LAA $A(t, t + \delta)$ è indipendente dai tempi di servizio degli utenti arrivati $< t$. È quindi possibile dire che $P\{N(t^-) = n \mid A(t, t + \delta)\} = P\{N(t^-) = n\} P\{A(t, t + \delta)\}$.

Si ottiene

$$a_n(t) = \lim_{\delta \rightarrow 0} P\{N(t^-) = n \mid A(t, t + \delta)\} = \lim_{\delta \rightarrow 0} \frac{P\{N(t^-) = n\} P\{A(t, t + \delta)\}}{P\{A(t, t + \delta)\}} = P\{N(t^-) = n\}$$

In definitiva è possibile dire che

$$a_n = \lim_{t \rightarrow \infty} a_n(t) = \lim_{t \rightarrow \infty} P\{N(t^-) = n\} = p_n$$

□

La probabilità di stato stazionario dopo una partenza:

$$d_n = \lim_{t \rightarrow \infty} P\{X(t^+) = n \mid \text{departure at } t\}$$

coincide con a_n , ovvero $a_n = d_n$. In condizioni stazionarie, il sistema appare stocasticamente identico agli utenti che arrivano e che lasciano il sistema.

3.11.1 Proprietà dei processi di Poisson

L'aggregazione (somma) di N processi di Poisson di parametro λ_i , è un processo di Poisson di parametro $\lambda_{TOT} = \sum_i \lambda_i$.

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 $\lambda_i = \lambda p_i$.

3.12 Teorema di Burke

Date due code in tandem si vuole trovare la distribuzione dei tempi di interarrivo alla seconda coda noto che la prima è una $M/M/1$.

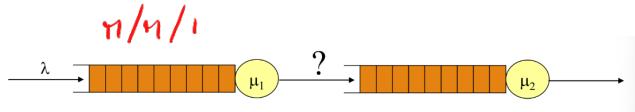


Figura 3.17: Tandem di code

La distribuzione dei tempi di interarrivo della seconda coda è uguale alla distribuzione dei tempi di interpartenza della prima coda.

Si indica con $D(t)$ la distribuzione dei tempi di interpartenza e $B(t)$ la distribuzione dei tempi di servizio.

Quando un cliente parte da una coda può verificarsi uno solo dei seguenti eventi:

- ① un altro cliente è presente sulla coda 1 e sarà subito servito;
 - ② la coda 1 è vuota, bisognerà quindi 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;

Nel primo caso si ha $D(s)|_{\text{coda non vuota}} = B(s)$, nel secondo caso si ha invece $D(s)|_{\text{coda vuota}} = B(s) \frac{\lambda}{s+\lambda}$.
Essendo il servente di tipo esponenziale si ha inoltre che $B(s) = \frac{\mu_1}{s+\mu_1}$.

Dato che la probabilità di avere il sistema non vuoto è pari a $\rho = \frac{\lambda}{\mu}$ si ottiene che

$$D(s) = (1 - \rho)D(s)|_{\text{coda vuota}} + \rho D(s)|_{\text{coda non vuota}}$$

TEMPO DI PARTENZA
SISTEMA VUOTO B(s) · SERVITO
SERVITO SISTEMA CON UNO SCELTO
SERVITO SISTEMA CON DUE O PIÙ SCELTI

e sostituendo i valori precedenti si ottiene

$$D(s) = (1 - \rho) \frac{\mu_1}{s + \mu_1} \frac{\lambda}{s + \lambda} + \rho \frac{\mu_1}{s + \mu_1}$$

che risulta uguale a

$$D(s) = \frac{\lambda}{s + \lambda} = A(s)$$

e antitrasformando si ottiene

$$D(t) = A(t) = \lambda e^{-\lambda t}$$

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.

3.13 Teorema di Jackson

Il teorema di Jackson generalizza il teorema di Burke per le reti di code aperte.

Si consideri una rete di code formata da K nodi (sistemi a coda) che soddisfano le seguenti tre condizioni:

- ogni nodo contiene 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 $1 - \sum_j p_{kj}$;

Il tasso degli arrivi al generico nodo k è:

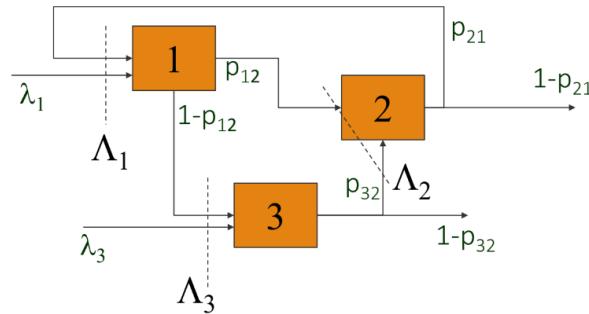
$$\Lambda_k = \lambda_k + \sum_{j=1}^K p_{jk} \Lambda_j$$

DA ESTERNO
DA ALTRI nodi

Indicando con $p(n_1, \dots, n_k)$ la probabilità congiunta stazionaria di stato nei nodi, se

$$\Lambda_l < c_k \mu_k \quad \forall k \quad \frac{\Lambda_l}{c_k \mu_k} < 1 \quad \text{per stazarietà}$$

allora $p(n_1, \dots, n_k) = p_1(n_1) \dots p_k(n_k)$. *Sono indipendenti*



$$\begin{cases} \Lambda_1 = \lambda_1 + p_{21}\Lambda_2 \\ \Lambda_2 = p_{12}\Lambda_1 + p_{32}\Lambda_3 \\ \Lambda_3 = \lambda_3 + (1-p_{12})\Lambda_1 \end{cases} \rightarrow \Lambda_1, \Lambda_2, \Lambda_3$$

Figura 3.18: Rete di code

3.14 Ipotesi di indipendenza di Kleinrock

L'ipotesi di indipendenza di Kleinrock è un concetto utilizzato per semplificare l'analisi delle reti di code, specialmente in contesti in cui i sistemi hanno una struttura complessa.

L'ipotesi di indipendenza afferma che:

- Gli arrivi ai nodi di una rete di code sono indipendenti.
- Le code operano in modo indipendente l'una dall'altra.

Questa ipotesi consente di trattare il comportamento di ogni nodo nella rete separatamente, anziché analizzare l'intero sistema come un'entità interdipendente. Più precisamente, si assume che gli arrivi ai nodi successivi (cioè, dopo che un pacchetto lascia un nodo) si comportino come un processo stocastico indipendente, spesso modellato come un processo di Poisson.

L'ipotesi di indipendenza semplifica notevolmente i calcoli, rendendo possibile:

- Applicare modelli matematici noti per sistemi a singola coda (come M/M/1 o M/M/c) a reti di code più complesse.
- Analizzare reti di comunicazione, come internet o reti di computer, senza dover considerare le correlazioni tra i nodi.

L'ipotesi di Kleinrock non è sempre rigorosamente valida, ma si applica in modo approssimativo in molti casi pratici. Essa funziona bene quando:

- Il traffico è sufficientemente disperso: il carico su ciascun nodo non è troppo elevato, evitando effetti di saturazione.
- Gli arrivi sono distribuiti in modo casuale e non presentano una forte correlazione temporale.
- Le reti non sono altamente congestionate: in caso di alta congestione, le dipendenze tra i nodi diventano più pronunciate.