

Sistemi di Applicazioni Cloud

Damiano Solerte

1 Principi del Cloud Computing

1.1 Servizi Internet

1.1.1 Caratteristiche comuni

- Robustezza e sicurezza: funzionamento continuo e nessuna vulnerabilità di sicurezza
- Scalabilità: funzionamento indipendentemente dal carico di lavoro (anche a fronte di attacchi DDoS)
- Usabilità
- Costo: apparentemente gratis
- Infrastruttura: grossi datacenter distribuiti

Uno dei problemi principali, che si evolvono con l'evolversi del web e dell'esponenziale crescita dei dati, è la scalabilità. Alcune soluzioni applicabili nel passato oggi non lo sono più. Esempio: oggi so come gestire 10 gigabyte di dati, domani saprò gestirne 10 Exabyte?

Al giorno d'oggi miliardi di persone hanno più di un dispositivo connesso alla rete, consumando milioni di servizi, usufruibili da milioni di providers, costruiti su milioni di server, contenenti zettabyte di dati 10^{21} Byte, connessi tramite petabyte di reti. Un altro vincolo importante è la tolleranza zero per i reclami irrisolti che possono portare alla perdite di un cliente.

C'è un cambio di paradigma, se prima il punto di partenza era lo sviluppo di una tecnologia adesso è lo sviluppo di un **servizio**.

1.1.2 Definizione di servizi

Definizione: La capacità di un sistema di fornire continuativamente una o più risposte in presenza di specifiche richieste al sistema da parte di un utente.

Possiamo definire un servizio funzionante quando continua a fornire delle risposte. Un concetto importante introdotto nella definizione è la **continuità**. Essa può essere definita in molti modi:

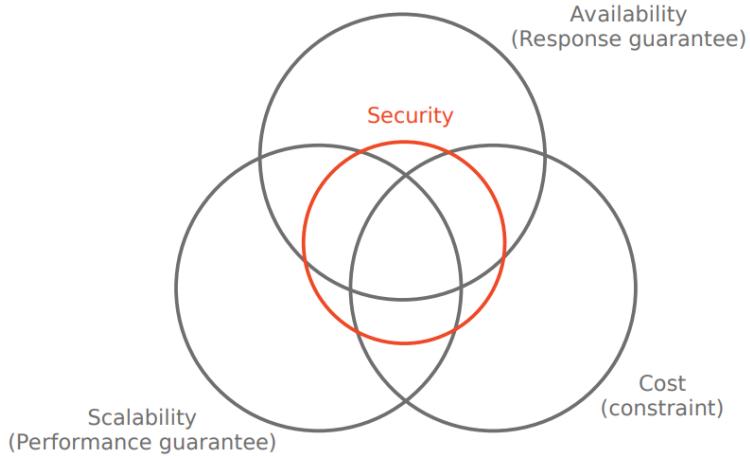
- Tramite contratto
 - SLA: Service Level Agreement in cui ci si accorda sui pagamenti e le penali
- Reputazione: pena la perdita di clienti e di fiducia

Il compito di un ingegnere del cloud è quello di implementare un sistema scalabile e che garantisce una certa continuità, attraverso i seguenti steps:

- Design
- Build
- Test
- Document
- Monitor
- Fix/Update

Le regole d'oro da seguire sono le seguenti:

- Evitare:



- **Colli di bottiglia**
- Single Point of Failure. Cercare almeno di identificarli anche se è molto difficile correggerli
Esempi:
 - * Elettricità e continuità di corrente in un data center.
- Replicazione, a più livelli:
 - Macchine server.
 - Dati.
 - Lavori computazionali.
 - Tasks.

1.2 Cloud Computing

1.2.1 Grid Computing

L’architettura di **grid computing** unisce, anche in cloud, diverse unità geograficamente distanti per intervalli di tempo e scopi specifici: i computer comunicano e agiscono come una singola identità di calcolo. Utilizzare un nodo significa quindi accedere alla potenza computazionale di tutta la rete.

Il grid computing è una rete che può essere sia omogenea che eterogenea, ovvero formata da nodi con lo stesso sistema operativo o diversi. I nodi che compongono l’architettura sono indipendenti: ciascun server o computer mantiene la propria autonomia dalla rete e condivide solo in parte le proprie risorse specifiche. La rete del grid computing è quindi decentralizzata: possono esistere più server autonomi all’interno della griglia. L’idea principale è quella di un’organizzazione virtuale dove molte organizzazioni contribuiscono a mettere a disposizione le risorse attraverso un grande supercomputer. L’unità di lavoro è il **gridlet**.

1.3 Globus Toolkit

E’ un sistema per creare sistemi grid, finalizzato al calcolo scientifico e parallelo. Lo scheduling viene eseguito per **batch jobs**, presentando quindi un’interattività limitata. Assenza di un chiaro modello economico.

1.4 Gestione delle Risorse

E’ importante cercare di capire l’**utilizzo** di un server che può variare nel corso di una giornata, per cercare di non avere delle perdite di introiti derivanti dalla mancata erogazione del servizio oppure di avere delle macchine inutilizzate. Un’intuizione particolare fu quella di Bezos di vendere potenza di calcolo inutilizzata ad altre aziende.

Strategia per gestire la potenza di calcolo in un’azienda:

- Soddisfare un **minimo workload**: si perdono parecchi clienti
- Soddisfare un utilizzo **medio**: non sono capaci di soddisfare il picco della domanda e inoltre potrei avere comunque potenza computazionale inutilizzata
- Soddisfare il **picco** di domanda: ottima per reputazione e incassi ma molta potenza computazionale inutilizzata

1.5 Alte performances e realizzabilità

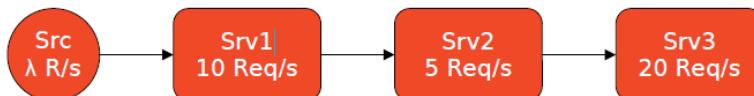
Alte performance si possono ottenere solo:

- Evitando **overload** e **bottlenecks**
- Fornendo **tempi di risposta** accettabili

Evitare invece SPoF (Single Points of Failure) garantisce **realizzabilità**.

L'obiettivo consiste nel fornire un determinato **throughput** (rateo di risposte) e un determinato tempo di risposta. I due obiettivi possono essere legati, dove la richiesta di uno prevede la necessità dell'altro.

Osservando una catena di servizi ci basta capire quale unità di elaborazione ha il throughput minore rispetto agli altri per capire se e quale sarà il collo di bottiglia. Per ottenere quindi un



sistema efficiente che non generi colli di bottiglia bisogna fare in modo che tutti i servizi abbiano lo stesso data rate. Un'opzione è quella di replicare i servizi che abbiano un data rate minore, in modo da allineare tutti i servizi. E' importante considerare che se abbiamo del codice che non può essere ne replicabile ne parallelizzabile, non sarà particolarmente adatto ad una struttura basata sul Cloud, per cui potrebbero facilmente generarsi colli di bottiglia.

1.6 Legge di Amdahl

- T_p : tempo per la parte di elaborazione parallela
- T_s : Tempo per la parte di elaborazione sequenziale
- $p = \frac{T_p}{T_s+T_p}$: quanto incide la parte parallela sulla computazione totale
- $T_n = T_s + \frac{T_p}{n}$
- n : Numero di processori

$$S = \frac{T_s + T_p}{T_s + \frac{T_p}{n}}$$

Per $n \rightarrow \infty$:

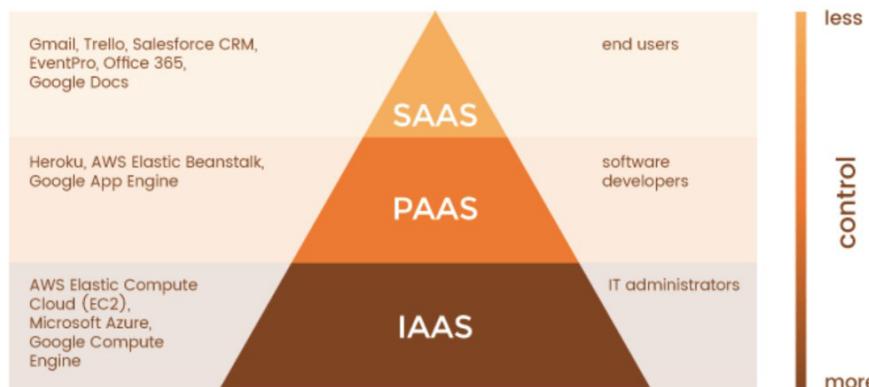
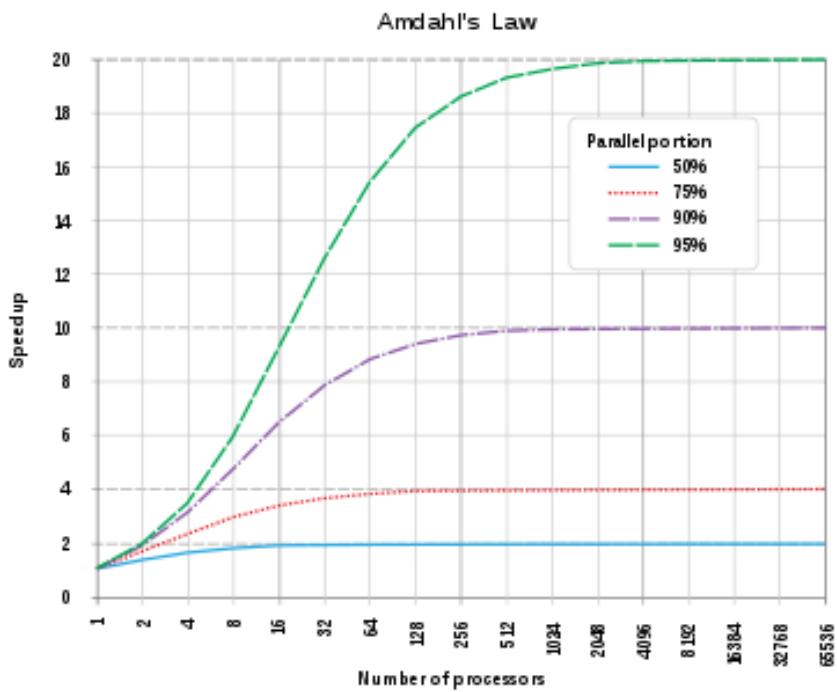
$$S = \frac{S}{1-p}$$

”Il miglioramento delle prestazioni di un sistema che si può ottenere ottimizzando una certa parte del sistema è limitato dalla frazione di tempo in cui tale parte è effettivamente utilizzata”

1.7 Paradigmi del Cloud Computing

Principali caratteristiche del cloud computing:

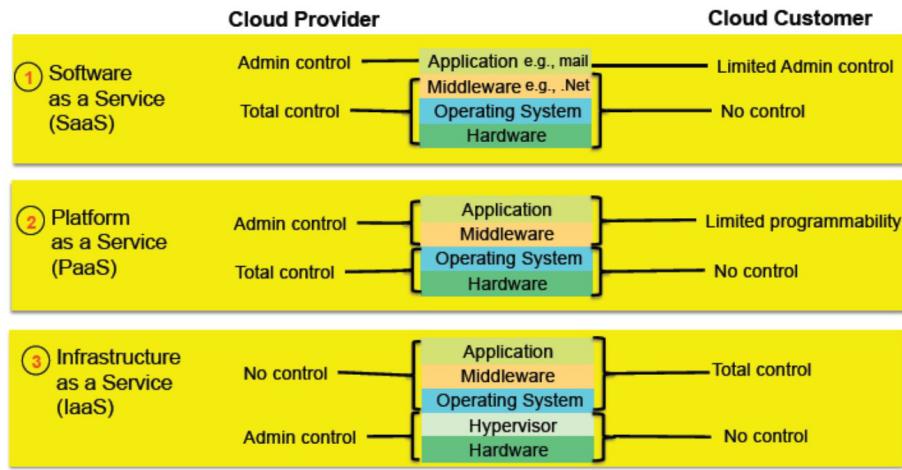
- Service: ogni elemento viene visto e implementato come un servizio.
- Deployment: in base alle caratteristiche dell'infrastruttura e della proprietà.



1.7.1 Paradigma Service

Differenti livelli di controllo:

- Software: In questo caso il provider fornisce all'utente solamente una console controllare il software.
- Platform: L'utente perde un'idea chiara di cosa succede a livello di sistema operativo. A livello applicativo ha anche qualche limitazione, come per esempio versioni di compatibilità. Gran parte del controllo rimane al provider.
- Infrastructure: In questo tipo di struttura l'utente ha la possibilità di gestire il sistema operativo della macchina. Il provider del server dovrà gestire virtualizzazione, servers, storage, e networking. Al giorno d'oggi la differenza tra Software e Platform è sfumata, basti pensare a quanti software forniti come servizi cloud possono essere altamente personalizzati come SAP, Salesforce e Shopify. Il confine è labile anche nell'altra direzione in cui molte piattaforme possono diventare quasi delle infrastrutture.



1.7.2 Paradigma del deployment

- **Public Cloud:** Un cloud pubblico offre i suoi servizi a qualunque utente. Questa struttura deve affrontare il problema della eterogeneità degli utenti, che possono avere bisogni diversi.
- **Community Cloud:** Gli utenti sono tra di loro più simili. In questo contesto il tipo di servizio è cucito su un determinato tipo di utenza. Da un punto di vista strutturale il cloud può essere ospitato sia dal provider che affidarsi ad un terzo (pubblica amministrazione, assistenza sanitaria)
- **Private Cloud:** Un'entità che basa il suo lavoro su tutti i paradigmi del cloud. In questo caso tutto il lavoro svolto dal cloud sarà solo ed esclusivamente legato all'azienda/entità. Può essere gestito sia dall'azienda che da terzi
- **Hybrid:** Mix tra le tipologie precedenti. Ad esempio possono essere più cloud (privati) federati tra di loro che devono svolgere compiti simili, o cloud privati che si appoggiano a cloud pubblici per gestire determinati compiti che possono generare dei picchi di richieste normalmente insoddisfabili. (Amazon).

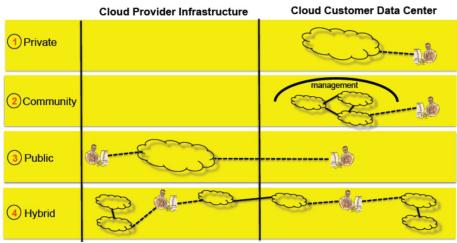


Figure 1: Cloud Provider & Cloud Customer

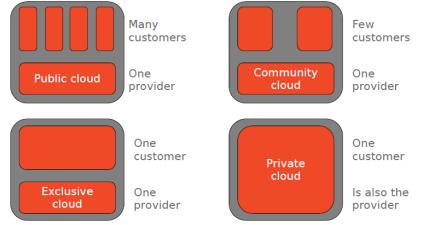


Figure 2: Paradigmi di distribuzione

1.7.3 Strutture del cloud

Vertical Silos

Tipico approccio di deployment di un'applicazione, non è proprio una struttura cloud, ma è stata la prima implementazione simile. Architettura one-to-one, dove ad ogni server corrisponde un'applicazione.

Si può cercare di consolidare il server e creare delle macchine virtuali su cui far girare le applicazioni, tuttavia i server rimangono spesso utilizzati.

1.7.4 Visione del cloud

Il consumatore dal suo punto di vista vede:

- Un set isolato di risorse
- Elastico e scalabile
- Sicuro
- Affidabile
- Con costi abbordabili: pay-per-use

NIST

Definizione: Il cloud computing è un modello per l'accesso immediato a un set di risorse di computing condivise e configurabili (rete, server, storage, applicazioni e servizi) che possono essere rapidamente forniti e deployed con il minimo costo di gestione e l'interazione del servizio provider.

1.8 Caratteristiche del Cloud

Per capire il passaggio a infrastrutture Cloud nell'utilizzo odierno dell'informatica è importante capire quali sono le varie motivazioni e i requisiti soddisfatti

1.8.1 Requisiti

E' assolutamente necessario che il software implementato funzioni bene indipendentemente dagli utenti che lo stanno utilizzando, garantendo prestazioni, affidabilità e robustezza. Per soddisfare questi requisiti si può ricorrere ad una maggiore potenza di calcolo o alla diminuzione dei costi di trasmissione, ma queste sono vecchie soluzioni. Le nuove soluzioni hanno l'obiettivo comune di doversi adattare rapidamente ai cambiamenti e al bisogno di elasticità, in quanto l'utilizzo di un sistema cloud può essere molto variabile. L'**elasticità** consiste nel sapere trovare un equilibrio tra i costi di infrastruttura e i tempi di esecuzione.

1.8.2 Software Cloud

Per ottenere tale elasticità è opportuno evitare errori come colli di bottiglia, SPOF e comportamenti sincroni. E' sempre bene usare pratiche di replicazione e orchestrazione autonoma. A livello di programmazione non è possibile applicare concetti classici come appunto quello della sincronizzazione. Per esempio i dati stessi non posso godere di una totale consistenza, avendo quindi in ogni punto del software dati aggiornati e pronti all'elaborazione. Lo stesso discorso vale per la comunicazione che non può essere totale e asincrona in ogni momento. Ecco perché per il cloud è opportuno adottare diversi paradigmi a livello software:

- Parallelismo su larga scala.
- Dati:
 - Lazy consistency: I dati non vengono aggiornati sempre immediatamente.
 - Distribuzione locale o geografica.
 - Caching.
- Comunicazione:
 - Asincrona.
 - IPC (Inter Process Communication) limitate.

Vi sono varie funzioni logiche da implementare all'interno del cloud:

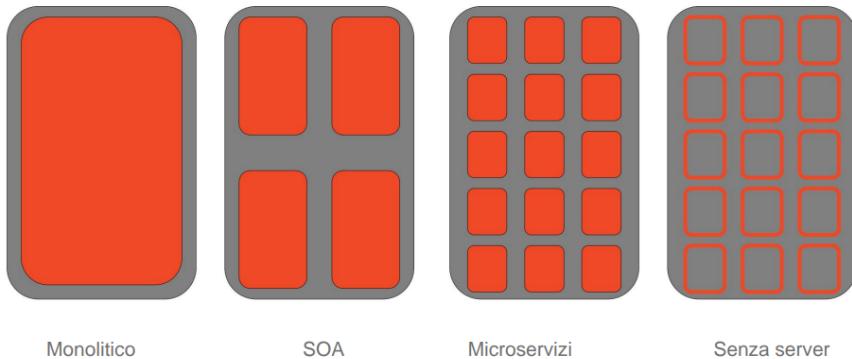
- Database (Model).
- Presentazione (View).
- Aziendale (Controller)

La replicazione può avvenire sia in modo **verticale** che **orizzontale**, ma anche in maniera ibrida.

1.9 Architetture

Per quanto riguarda invece l'architettura di un servizio gli approcci possibili sono i seguenti:

- Monolitico: Un singolo software, difficile da replicare e mantenere, quindi costoso.
- SOA: Service Oriented Architecture. Architettura basata su protocolli SOA e su XML
- Microservizi: Consiste nel suddividere l'architettura in tanti piccoli servizi facili da replicare e riutilizzare ma soprattutto da implementare
- Serverless: Architettura senza server, una sorta di microservizi 2.0



Le architetture ibride prevalgono nella maggior parte delle aziende per la difficoltà a convertirsi ai nuovi paradigmi.

1.9.1 Modelli pre-SOA

Approcci esistenti ad oggi:

- RPC: Chiamata di procedura remota. Viene utilizzato come input per creare del sottocodice. Consiste nella creazione di codice per wrappare un'invocazione remota, nascondendo gli aspetti dell'interazione remota. Viene usato il linguaggio IDL
- RPC orientato agli oggetti.
- Middleware orientato ai messaggi: Approccio client-server con messaggi sincroni o asincroni con code di messaggi. Tipici di applicazioni *fire-and-forget* come le notifiche.

Dal punto di vista semantico, una chiamata di funzione locale e la RPC sono la stessa cosa poiché grazie al codice **stub** la complessità è nascosta. Il codice stub è una porzione di codice che simula il comportamento di un altro software o della parte di codice ancora da sviluppare in via temporanea.

1. The client calls the **client stub**. The call is a local procedure call, with parameters pushed on to the stack in the normal way.
2. The client stub packs the parameters into a message and makes a system call to send the message. Packing the parameters is called **marshalling**.
3. The client's local operating system sends the message from the client machine to the server machine.
4. The local operating system on the server machine passes the incoming packets to the server stub.
5. The **server stub** unpacks the parameters from the message. Unpacking the parameters is called **unmarshalling**.
6. Finally, the server stub calls the server procedure. The reply traces the same steps in the reverse direction.

Dal punto di vista prestazionale la differenza è enorme e ci permette di evitare ritardi di rete o confusione nei parametri, dato che dobbiamo stabilirli in anticipo. Esistono stub sia a livello frontend sia a livello backend.

OO-RPC

E' possibile anche ricreare il concetto di RPC nella programmazione orientata agli oggetti, come per esempio in JAVA dove viene definita un'interfaccia che definisce i metodi che vogliamo esporre. La classe che implementa l'interfaccia è l'analogo dell'IDL.

1.9.2 Modelli SOA

Architettura orientata ai servizi con approccio orientato al business, è basata su componenti software creati con l'obiettivo di evitare software monolitici e nascondere l'architettura centrale sottostante. In questo modo servizi della stessa azienda possono essere riutilizzati o integrati con altri di altre aziende. Si possono avere così tre scenari:

- SOA **intra**-aziendali.
- SOA **inter**-aziendali con rapporti B2B già consolidati: Un esempio può essere EXPEDIA che cerca e confronta varie offerte di altre aziende (compagnie aeree).
- SOA **inter**-aziendali con servizi offerti liberamente.

Gli elementi che caratterizzano i SOA sono:

- Servizio: fatto da moduli, deve avere una descrizione e un interfaccia di accesso.
- Tecnologie abilitanti.
- Governance e politiche SOA.
- Metriche SOA.
- Modello organizzativo e comportamentale.

1.9.3 SOAP

Service Oriented Architecture Protocol. Nasce nell'ambito dei servizi web come tecnologia per la messaggistica di base.

Un **servizio Web** è un sistema software identificato da un URI, le cui interfacce e collegamenti pubblici sono definiti e descritti utilizzando XML. La sua definizione può essere vista da altri sistemi software. Questi sistemi possono quindi interagire con il servizio Web nel modo stabilito dalla sua definizione, utilizzando messaggi basati su XML trasmessi dai protocolli Internet. Il protocollo SOAP è basato sui protocolli HTTP o HTTPS e i dati sono scritti attraverso il linguaggio XML. Tuttavia SOAP presenta dei limiti: la lentezza di XML rappresenta la sua debolezza maggiore. Un approccio migliore è utilizzare JSON che è particolarmente adatto alle architetture REST.



Figure 3: Soap è incapsulato in HTTP e definisce una struttura per le risposte HTTP in XML

1.9.4 Enterprise Service Bus

L’architettura SOA prevede che i componenti delle applicazioni possano essere lanciati su diverse macchine e debbano essere tutti capaci di comunicare fra loro. Tuttavia all’aumentare del numero di applicazioni, la struttura diventa difficile da mantenere.

L’uso di un **middleware** a modo di BUS condiviso per la connessione tra i componenti permette di replicare lo stesso componente per la connessione tra customer e producer. Questo componente si chiama **ESB** (Enterprise Service Bus) e ha le seguenti funzioni

- Instrada i messaggi verso la destinazione.
- Gestisce differenti modelli di comunicazioni (code asincrone, messaggi in base agli eventi, servizi vari).
- Servizi di intermediazione (broker).
- Connettore e ponte per diversi protocolli.
- Supporta la policy e la qualità del servizio (QoS).
- Monitor, Logging, Sicurezza.

1.9.5 Microservizi

L’architettura a microservizi cerca di mappare l’applicazione in vari elementi molto piccoli, quasi atomici. Ogni microservizio ha un suo processo, un suo stack tecnologico diverso (diverso linguaggio di programmazione). All’interno dell’applicazione deve quindi essere implementato un sistema di comunicazione tra i vari servizi.

SOA vs Microservizi

Entrambe le strutture supportano complesse applicazioni modellate in maniera più piccola e più maneggevole con parti indipendenti. Alcuni libri considerano i due approcci identici. I microservizi comunicano molto bene attraverso le API e non richiedono un ESB. Ognuno può essere aggiornato e spento singolarmente (se non usato al momento). La separazione logica tra i microservizi permette la loro esecuzione all’interno di containers.

Benefici

- Migliore testing: I servizi sono piccoli e facili da testare
- Deployment: Possono essere caricati indipendentemente
- Organizzazione: Rende più facile l'organizzazione del lavoro tra team e ogni team può organizzare e gestire il proprio microservizio.

Difetti

- Non tutte le applicazioni sono orientate a microservizi.
- Non è così diffuso in scenari tradizionali.
- Un'applicazione deve implementare la comunicazione tra servizi.
- Overhead: se non implementati a dovere si rischia di mandare in crash il sistema.

1.10 Architettura REST

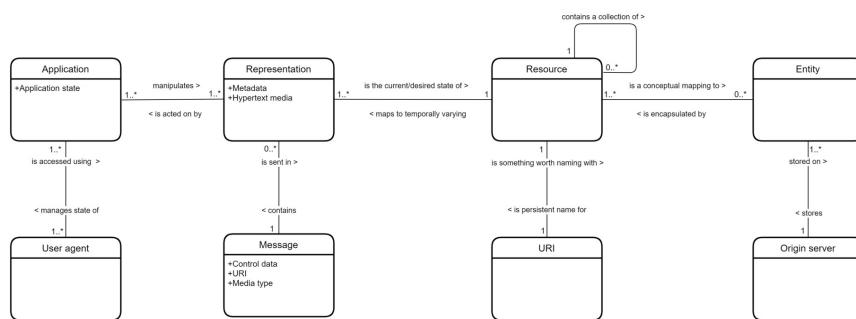
1.10.1 Applicazioni Cloud

Ricordiamo che le applicazioni cloud hanno i seguenti requisiti:

- Resilienza
- Resistenti a failures
- Elastici in modo da scalare da pochi server a decine di migliaia.
- L'architettura deve avere dei servizi *loosely-coupled* e **stateless**

1.10.2 REST

REpresentation State Transfer, fornisce all'applicazione un architettura senza stati, utilizzando tutta le espressioni del protocollo HTTP. Il concetto chiave è quello di incapsulare una risorsa all'interno di un'entità (URI). Le risorse forniscono dati attraverso una rappresentazione (come ad esempio una struttura JSON). Le azioni sulle varie entità sono mappate su messaggi (cioè i vari metodi del protocollo HTTP). Non necessita di molteplici servizi per il funzionamento.



I client e i server sono molto leggeri dato che le risorse, identificate da URI, incapsulano delle entità e per invocarle non c'è bisogno di una SOAP. In questo caso il contenuto delle risorse viene visualizzato con una struttura chiave valore, stile JSON. **Vincoli**

- Modello client server.
- Operazione stateless, una caratteristica molto vantaggiosa.
- Risultati possono essere salvati in cache.
- Struttura a strati.
- Un servizio REST può interraccarsi con altri servizi.
- Interfaccia uniforme: Uso di URI (Uniform Resource Identifier), fornisce una rappresentazione delle azioni e dei messaggi autodescrittiva.

1XX Informational	
100	Continue
101	Switching Protocols
102	Processing

3XX Redirection	
300	Mutlic Choice
301	Moved Permanently
302	Found
303	See Other
304	Not Modified
305	Use Proxy
307	Temporary Redirect
308	Permanent Redirect

4XX Client Error	
400	Bad Request
401	Unauthorized
402	Payment Required
403	Forbidden
404	Not Found
405	Method Not Allowed
406	Not Acceptable
407	Proxy Authentication Required
408	Request Timeout

499	
Conflict	
410	Gone
411	Length Required
412	Precondition Failed
413	Payload Too Large
414	Request-URI Too Long
415	Unsupported Media Type
416	Requested Range Not Satisfiable
417	Expectation Failed
418	I'm a teapot
421	Misdirected Request
423	Unprocessable Entity
424	Locked
426	Failed Dependency
428	Upgrade Required
429	Precondition Required
431	Too Many Requests
441	Request Header Fields Too Large
451	Unavailable For Legal Reasons
499	Client Closed Request

Figure 4: Information - Success - Redirectional

Figure 5: Client Error

Non ci sono esplicativi limiti nei messaggi. Le tipiche azioni seguono il paradigma CRUD:

- CRUD paradigm:
 - Create.
 - Retrieve (Read).
 - Update.
 - Delete.

Le risorse sono accessibile attraverso URL usando i seguenti metodi HTTP:

- GET.
- POST.
- PUT.
- DELETE.
- HEAD: Controlla se una risorsa esiste o è cambiata
- OPTIONS: Quali sono i metodi supportati per una risorsa.

Le differenze tra PUT e POST sostanzialmente risiedono nell'intenzione di voler creare (POST) o aggiornare (PUT) una risorsa. Ecco i vari codici di risposta delle chiamate REST:

5XX Server Error	
500	Internal Server Error
501	Not Implemented
502	Bad Gateway
503	Service Unavailable
504	Gateway Timeout
505	HTTP Version Not Supported
506	Variant Also Negotiates
507	Insufficient Storage
508	Loop Detected
510	Not Extended
511	Network Authentication Required
599	Network Connect Timeout Error

VERB	Standard Return Codes
GET	200, 401, 403, 404
POST	200 (should be 201), 201, 401, 403, 404, 422
PUT	200, 401, 403, 404, 422
PATCH	200, 401, 403, 404, 422
DELETE	204, 401, 403, 404, 422

```
HTTP/1.1 401 Unauthorized
Content-Type: application/json
{
  "errorCode": "401"
  "message": "Unauthorized. You are not
logged in."
}
```

Figure 6: Error

E' possibile definire un TTL, un time to live per stabilire un tempo massimo di esistenza della nostra cache. Una richiesta HTTP può essere cached in modo da non dover nuovamente essere richiesta al server. Un Etag mantiene i metadati sulla risorsa e sull'ultima modifica ad essa fatta tramite un MD5 digest della rappresentazione della risorsa.

1.10.3 RESTful API

Standard *de facto* per la costruzione di sistemi distribuiti. I servizi REST non hanno supporto diretto per generare un client da un IDL mentre i sistemi SOAP possono generare codice STUBS da WSDL (Web Service Description Language), un linguaggio formale in formato XML utilizzato per la creazione di "documenti" per la descrizione di servizi web. Deve essere quindi utilizzato un **description language**, un esempio è OpenAPI, una specifica implementazione per IDL machine-readable usata per descrivere, produrre e visualizzare RESTful web services.

Un concetto fondamentale delle chiamate REST è il controllo dei media, ovvero del tipo di contenuto disponibile che il client può chiedere o che il server può fornire. L'idea di base è che una rappresentazione di un oggetto dovrebbe dire al cliente cosa può fare con l'oggetto o le azioni correlate che potrebbe intraprendere. Tali descrizioni sono chiamate "controlli ipermediali". Il paradigma **Hypermedia as the Engine of Application State** prevede che l'utente non abbia conoscenza a priori di come interagire con un'applicazione e abbia solo una conoscenza generica degli hypermedia. Le public REST APIs devono essere:

- Semplici e stateless: Non essere legate a processi come i servizi Web.
- Bullet proof: Devono contemplare una gestione degli errori/controllo degli input
- Progettate dall'esterno verso l'interno in modo che i lenti cambiamenti interni non vengano avvertiti dall'esterno
- Auto descrittive: Devono usare **documentazione** standard per web API (esempio Swagger)
- Riutilizzabili: Devono essere configurabili non codificate

1.11 Architetture guidate agli eventi (EDA)

Supporta la generazione e la gestione della notificazione degli eventi. Sono delle architetture flessibili e reattive che riescono ad adattarsi ai cambiamenti in tempo reale. Gli elementi che generano notifiche non devono necessariamente conoscere i componenti del ricevitore. Il tempo di risposta non è deterministico.

Le notifiche di eventi annunciano una **modifica** nello stato del sistema, possono essere innescate da:

- Fonti esterne: Input utente, condizioni ambientali.
- Notifiche interne: Invio di dati per la pipeline workchain etc.

Un classico esempio applicativo di questa architettura è il sistema Pub/Sub, ma anche:

- **AMQP**: Advanced Message Queuing Protocol
- **MQTT**: Message Queue Telemetry Transport L'architettura prevede un sensore, che per esempio rileva dei dati inerenti a qualche monitoraggio, che comunica ad una istanza di un broker responsabile dell'inoltro dei messaggi ai client destinatari. I messaggi scambiati hanno una struttura chiave-valore.
- **Apache Kafka**: Si passa dalle entità contenute nei database ad una sequenza di eventi non ordinate descritta nei logs. Gli eventi sono molto più adatti a scalare delle entità. Ogni log è associato ad un topic e può essere replicato o partizionato, suddividendolo in sub-log. Kafka supporta una suddivisione in base agli eventi. Kafka è focalizzato sulla scalabilità e la tolleranza a *failure*. Tolleranza che però viene pagata in termine di costi, data la sua replicazione.

1.11.1 EDA Scalability

I punti chiave per la scalabilità delle architetture ad eventi sono la possibilità di inviare messaggi in modo **asincrono** tra **coppie che si creano liberamente** e implementando una **eventuale consistenza** invece che una forte, dove i dati vengono aggiornati non in maniera istantanea ma solo in determinati momenti.

- Comunicazioni asincrone: Se un'applicazione si satura a causa di un carico eccessivo, l'applicazione asincrona può rallentare e gli eventi si accoderanno.

- Accoppiamento libero: A causa della struttura Pub/Sub, non sappiamo nulla riguardo i Subscriber che decidono di iscriversi ad un determinato argomento per ricevere le notifiche.
- **Eventual Consistency:** Utilizza una cache per memorizzare i dati aggiornati, riduce il carico del sistema e mantiene comunque i dati in un buon stato per tutti i nodi dell'architettura. Il livello di consistenza che si vuole mantenere nel nostro database dipende come sempre dal tipo di applicazione che stiamo creando. Per un social network, un aggiornamento non istantaneo non è un problema, per un sistema di stoccaggio di scorie nucleari forse sì.

Comunicazione Asincrona

Le code in questo caso, vengono utilizzate come dei buffer. I consumatori processano gli eventi come meglio possono e se una applicazione dovesse saturare per il carico eccessivo, rallenterà un po' ma non andrà mai in down.

Perdita dell'accoppiamento

Gli eventi del producer hanno una struttura definita dal producer stesso e vengono pubblicati in un dato istante. Il producer non sa nulla dei subscriber e i consumatori possono aggiungersi in qualunque momento.

Eventual Consistency

Il sistema di caching riduce il carico del sistema e può mantenere il buono stato di tutto il sistema. Il flow dei messaggi di notifica tra i sistemi cooperanti permette infatti, anche se non in maniera istantanea, di mantenere allineate le diverse basi dati. L'eventual consistency, se da un lato risulta più debole sia della consistenza del teorema CAP che della consistenza delle transazioni ACID, dall'altro, a regime e con un certo grado di imperfezione, tende a raggiungerle entrambe. Infatti la ricezione dei messaggi dai sistemi cooperanti permette sia di replicare in locale i dati aggiornati di pertinenza dei sistemi remoti, sia di adeguare i dati di pertinenza del sistema locale al fine di soddisfare i vincoli di integrità complessivi del sistema distribuito.

Atom Feed

Atom è un sostituto basato su HTTP per RSS (Really Simple Syndication) progettato per pubblicare cose come newsfeed, ma a causa della sua implementazione basata su HTTP è di natura molto RESTful. Quando si utilizza Atom in sostituzione di un ESB, invece di iscriversi agli argomenti, un consumatore esegue invece il polling del feed Atom del componente. Quindi elabora i messaggi alla propria velocità e restituisce altri messaggi quando è pronto. Poiché in genere non ci sono troppi abbonati al feed Atom di un singolo componente, il carico risultante non è significativo e ciascun consumatore controlla la propria capacità. Inoltre, se un consumatore si interrompe, non vi è alcuna preoccupazione che abbia perso i messaggi perché continua semplicemente da dove era stato interrotto. Infine, i feed Atom forniscono una linea cronologica del componente.

- I subscriber sono pochi
- Se l'intervallo di poll è breve
- Se la perdita di messaggi non è critica

In generale Atom è più semplice di altri approcci

1.12 Comunicazione e Coordinamento

Dopo aver visto varie implementazioni di strutture cloud è importante cercare di capire come poterli gestire e **orchestrare**.

- Orchestrazione: Un'autorità centralizzata controlla l'esecuzione dei servizi dei nodi, proprio come un direttore d'orchestra fa con i musicisti.
- Coreografia: Ogni componente conosce il proprio set di azioni prestabilito che deve compiere in relazione agli altri elementi con cui interagisce. Alla base di questo design vi è l'**Event Stream**, un bus su cui i componenti pubblicano e ricevono gli uni dagli altri. L'analogia è con i ballerini in cui ognuno sa cosa fare e come interagire con gli altri senza che un coordinatore centrale. Implicitamente decentralizzato.

In entrambi i casi i servizi possono essere incapsulati in REST API, cambia solo il modo in cui vengono gestite le richieste.

Messaggi

Esempio: Il microservizio A manda dei messaggi al topic e si mette in coda:

- A: Servizio a monte

- Consumatori: Servizio a valle
- Comunicazione asincrona
- Il sistema di messaggistica può garantire la consegna e lo store dei messaggi, con conferma della ricevuta.

1.12.1 Resilienza

La resilienza è un obiettivo fondamentale dei sistemi distribuiti per evitare il verificarsi di problemi come **errori a cascata** o chiamate remote fallite, bloccate o senza risposta. Quando qualcosa va storto bisogna gestire il tutto arrecando meno danno possibile. Quando abbiamo servizi connessi tra di loro, un problema su uno di essi può ripercuotersi su tutti gli altri connessi.

Per evitare questi errori è importante mantenere i nostri servizi il più possibile stateless in modo tale da non perdere informazioni in caso di interruzione del servizio. Se viceversa si ha un servizio con stato, la migrazione è molto più complicata e in quel caso si perdono informazioni, ad esempio un utente in un e-commerce perderebbe tutti i prodotti inseriti in un carrello. Un'altra caratteristica importante è quella di scrivere delle chiamate REST robuste e capaci di gestire errori e ritardi.

Circuit Breakers

Se un servizio diventa troppo lento, viene marcato dal breaker come "bad" e restituirà un errore anziché continuare il polling. Bisognerà poi capire quando sarà tornato in buono stato riprovando a testarlo.

2 Data Management

Data Replication La scalabilità di un sistema si può ottenere in due modi:

- Scale-Up: Tramite il miglioramento delle risorse hardware del server. Presenta degli ovvi limiti fisici.
- Scale-Out: Replicare il singolo server su più server. Scalando orizzontalmente si ha una replicazione a livello server e a livello cache, per cui bisogna stabilire come quest'ultima venga aggiornata.
- Replicazione:
 - Piena: Tutte le risorse vengono replicate.
 - Parziale: Soltanto una parte viene replicata.
- Policy sulla consistency:
 - Strong: I contenuti sono sempre aggiornati.
 - Weak: I contenuti non sono aggiornati immediatamente.
- Ragioni: Migliorare le performances e aumentare la disponibilità dei dati.

2.1 Data Synchronization

Se non abbiamo nessuna consistenza si hanno **performance migliori**, una migliore scalabilità e assenza di overhead dovuto alla sincronizzazione. Se invece abbiamo consistenza forte gli aggiornamenti sono immediati e non c'è rischio di perdere dei dati, è quindi un paradigma adatto a servizi che necessitano di **alta disponibilità**. Il compromesso tra i due la consistenza debole. Abbiamo due differenti tipi di copie:

- Primarie: Copie autoritarie.
- Master: Copie in lettura/scrittura. Le altre copie sono solo in lettura

Nel caso di consistenza forte tutte le copie sono primarie.

Concurrency control system

One-Copy System: L'idea alla base è che tutte le copie fisiche dei dati si devono comportare come un singolo elemento logico. La sequenza di transazioni distribuite deve essere uguale all'esecuzione serializzata delle stesse sullo stesso elemento logico. Il che significa che quando leggiamo un dato questo deve essere il risultato della scrittura più recente dalla precedente transazione nell'equivalente ordine seriale.

Ad esempio, se io prelevo contemporaneamente dallo stesso conto 20 euro in America e 20 euro in Europa e il mio bilancio iniziale era di 50 euro, vorrei che il bilancio finale fosse di 10 euro. Poiché posso ipotizzare che entrambe le filiali abbiano la propria copia del database dei conti correnti, il bilancio potrebbe non essere sempre 10 euro, ma 30 euro.

Il sistema one-copy prevede che le transizioni vengano effettuate sulla stessa copia logica, risolvendo quindi il problema. La coerenza delle copie è garantita da un sistema di controllo che assicura il corretto funzionamento, dopo l'avvenuta sincronizzazione.

Two-phase commit

E' un'altra strategia di sincronizzazione che ha bisogno di un coordinatore che prenda la decisione finale e interagisca con ogni partecipante. Può esserci anche un approccio gerarchico in cui il coordinatore delega i nodi primari di interpellare i nodi finali. Si basa su due fasi:

- **Preparazione:** in cui il coordinatore prepara tutti i partecipanti ad eseguire la transizione e attende che rispondano con l'esito dell'operazione.
- **Scrittura:** in cui, sulla base dei voti dei partecipanti, il coordinatore decide se continuare nella transazione o interromperla e avvisa i partecipanti dell'esito della decisione. I partecipanti seguiranno le indicazioni o faranno rollback a prima del commit.

Creata per la sincronizzazione di database distribuiti, fornisce un forte supporto di coerenza ma è relativamente costoso. Inoltre il rischio di una rollback aumenta il numero di repliche e il numero di messaggi scambiati.

Gossip Protocol

Un sistema di replicazione dati in base a scambi di informazione con i vicini. L'approccio alla consistenza è dato dal fatto che ci sono almeno **N** repliche di ogni elemento, dove N può essere configurato e $N > 1$ indica alta disponibilità dell'elemento. Un esempio è Cassandra: quando un nodo riceve una query, identifica tutti i nodi che possiedono quella replica attraverso una DHT (Distributed Hash Table). Invia quindi un aggiornamento della replica ai nodi trovati. In questo caso abbiamo una consistenza *lazy* perché prima o poi la propagazione delle modifiche si estenderà a tutti i nodi.

Primario e secondario

Il server secondario può rispondere nel caso di failure del server primario, garantendo quindi **availability**. Può essere anche utilizzato per fornire **performance** aggiuntive. Le operazioni di **lettura** possono essere fatte anche solo sui server secondari. Le operazioni di scritture devono essere inviate al server primario che sarà successivamente sincronizzato con il server secondario, sincronizzazione che avviene periodicamente, siano essi pochi minuti o giorni. Tuttavia la copia nel server secondario viene utilizzata solo se la copia primaria non è disponibile.

2.1.1 Consistenza forte e debole

Le coppie di repliche primarie-primarie mantengono una consistenza forte venendo aggiornate immediatamente, mentre una coppia primaria-secondaria ha una consistenza debole.

Nei server secondari si può avere una perenne consistenza debole ma alla fine i dati convergeranno ad un'unica versione e le copie saranno coerenti. Per fare ciò vi è uno scambio periodico di dati con timestamp, un algoritmo inventato da Lamport per ordinare i messaggi. Vi sono due approcci per regolare la consistenza nel nostro database:

- **BASE**

Basic Availability: La replica dei dati riduce il rischio dell'indisponibilità e il partizionamento dei server. Il sistema risulta così sempre consultabile.

Soft-State: I dati possono essere incoerenti e i servizi devono tenerne conto.

Eventual Consistency: Prima o poi nel futuro i dati saranno coerenti ma non c'è garanzia di quando questo accada.

- **ACID**: Atomicity, Consistency, Isolation, Durability. Tipico approccio dei database relazionali.

- I processi devono essere transazionali, totali o nulli.
- Dati sempre coerenti dopo una transazione.
- Database isolato e indipendente dalle transazioni
- Database robusto alla possibilità di perdita di cambiamenti. Ogni volta che si esegue una transazione i cambiamenti vengono salvati prima di essere effettivamente scritti nei registri di log.

2.2 Replicazione Geografica

Risulta impossibile distribuire server primari in ognuno dei 60000 Autonomous Systems e aumentare il numero di server primari presenta un problema di gestione degli stessi. Per questo si utilizzano un numero sempre crescente di server secondari, utili anche al miglioramento delle performance. **Cache** Il caching dei dati avviene su più livelli: hardware, OS, software ma anche a livello browser, server e **intermediari**. L'obiettivo unico del caching è quello di **replicare** parte del contenuto originale in posizioni il più possibile vicine all'utilizzatore. I principi su cui si basa il caching sono due:

- Località **spaziale**: se un utente accede a un dato è molto probabile che i futuri accessi siano vicini a quello stesso dato.
- Località **temporale**: se un utente accede a un dato è molto probabile che accederà allo stesso dato a breve e non in un futuro remoto

L'efficacia del caching si basa su tre metriche: cache **hit** se trovo il dato in cache, cache **miss** se non lo trovo e cache **hit rate** ovvero il numero di hit in un intervallo di tempo rispetto al numero totale di richieste in quello stesso intervallo.

Vi sono vari meccanismi di memorizzazione nella cache:

- Pull: Il contenuto viene caricato dal server primario quando viene richiesto dall'utente e non ne esiste una copia valida nel server secondario. Principalmente utilizzato da ISP e proxy servers.
- Push: I contenuti che vengono richiesti con maggiore probabilità sono pre-caricati sul server secondario. Utilizzato da reverse proxy, proxy che recupera i contenuti per conto di un client da uno o più server, e Content Delivery Network (CDN).

2.2.1 Replicazioni dei dati

- **Server Proxy:** Quando un browser richiede una risorsa questa prima viene ricercata nella cache locale e, in caso di esito negativo viene interrogato il server proxy. Quando invece un proxy server riceve una richiesta di una risorsa questa viene prima ricercata nella propria cache locale e in caso negativo viene richiesta al server e salvata nella cache, prima di essere restituita al browser. La validità dei dati all'interno della cache si **deduce** dal tempo di creazione e dall'ultima modifica:

- Volatile: Vecchio dato ma modificato recentemente.
- Static: Vecchio dato e modificato tempo fa.

Se si ha un nuovo dato è difficile da stabilire ma si presuppone volatile. Il popolamento della cache è guidato dalle richieste del client. Il proxy server non è molto utilizzato in quanto non gestisce contenuti generati dinamicamente e inoltre non supporta le connessioni HTTPS.

- **Reverse Proxy:** Un server virtuale posto di fronte al Web server in modo che possa immagazzinare risorse generate dinamicamente recenti. Può essere replicato e diffuso geograficamente.
- Terze parti: ISP o CDN.

Sia Forward che Reverse proxy sono in grado di replicare i contenuti popolari, riducono i costi del server di origine, riducono il RTT per i client. Solo il Reverse proxy può suddividere il carico intelligentemente tra i server di origine e modificare dinamicamente i contenuti per conto dei server di origine. Di contro, il Forward proxy riduce i costi legati all'ISP per il client.

Da un punto di vista di business possono esserci vari scenari, il fornitore di contenuti:

- Fornisce e gestisce tutto, dalla creazione di contenuti al master, fino a primary e secondary servers.
- Si ferma ai primary servers, delegando i secondary servers.
- Genera contenuti e fornisce il master, delegando il resto a una o più parti
- Genera solo contenuti, delegando tutto a più parti o ad una sola CDN, la quale si occupa dal master fino al secondary.

2.2.2 Cache WEB

Vi possono essere più server proxy che cooperano tra di loro per lo scambio di dati e aumentare l'**hit rate**. Se, infatti, un proxy fa hit miss allora può esserci un altro proxy nelle vicinanze che quella risorsa l'ha in cache.

- Schema gerarchico: Cooperazione verticale, su più livelli. Dato un miss locale la richiesta viene inoltrata a un nodo nel livello più alto della gerarchia. Ogni livello aumenta però la latenza. Livelli più alti contengono **molti** più dati per aumentare l'hit rate. Risolve il problema dei **compulsory miss**.

- Schema piatto: Cooperazione orizzontale tra pari. I due approcci tipici sono legati alla rappresentazione dell'informazione:

Query-based: Quando arriva una richiesta per una risorsa che non ho in cache, la indirizzo ai miei vicini. Il global miss si avrà solo quando tutti i nodi avranno risposto ed ha la stessa velocità del nodo più lento.

Informed-based: I vicini si scambiano periodicamente informazioni sui contenuti in cache redigendo un **indice della cache**. Quest'ultimo può diventare enorme e quindi difficile da scambiare. Per ovviare a questo problema si può usare una rappresentazione compatta:

- Cache Digest: Utilizza un **bloom filter** per rappresentare l'indice della cache a discapito della precisione. Si usa un vettore binario e più funzioni di hash. L'elemento che vogliamo inserire (ad esempio l'URL di una risorsa) verrà fatto passare attraverso le funzioni di hash e l'output restituito da ognuna di esse sarà l'indice della posizione del vettore da porre a 1. Poiché diversi elementi possono essere *hashati* alla stessa posizione, trovare un elemento all'interno del Bloom filter non ne garantisce l'esistenza ma fornisce solo un'indicazione probabilistica. Aumentando le funzioni di hash possiamo aumentare anche la bontà della probabilità restituita.
- Partitioning: Computa la funzione di hash sull'URL per trovare l'elenco dei nodi responsabili di quella risorsa. Molto efficiente ma se un nodo va offline ritorna il problema della propagazione delle informazioni aggiornate.

- Schemi ibridi

2.2.3 CDN

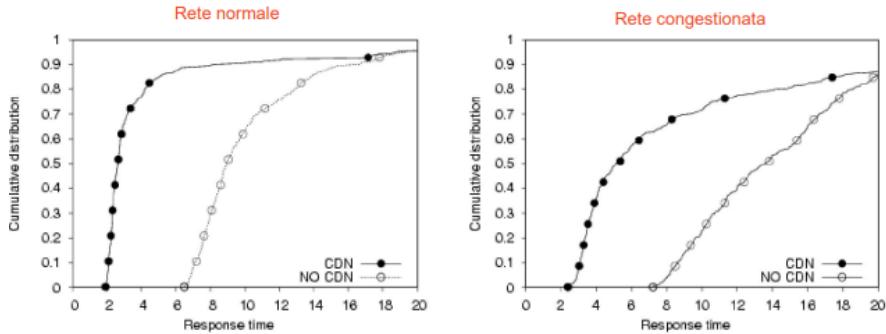
A differenza della memorizzazione cache tradizionale attraverso la cache selettiva riusciamo a selezione solo determinate risorse rilevanti che possono interessare maggiormente rispetto ad altre. Questo tipo di dati che vengono memorizzati possono essere venduti, esempi di questi contenuti possono essere i vari servizi di streaming a pagamento, come video on demand o musica a pagamento, ma anche software pay-per-use.

Da qui nascono le Content Delivery Network (CDN), un gruppo di server distribuiti geograficamente che collaborano per garantire la rapida trasmissione di contenuti Internet **selezionati** dai content provider. La memorizzazione della cache avviene in maniera **cooperativa** e **selettiva**. Usano prevalentemente un paradigma di pushing più che di pulling per via della natura selettiva delle CDN. La replicazione è molto semplice perché è il provider a fornire una nuova versione della risorsa ed è la CDN a sapere in quali server era stata pushata la vecchia. Principalmente vengono utilizzate per contenuti Web statici.

Una CDN è organizzata su due livelli:

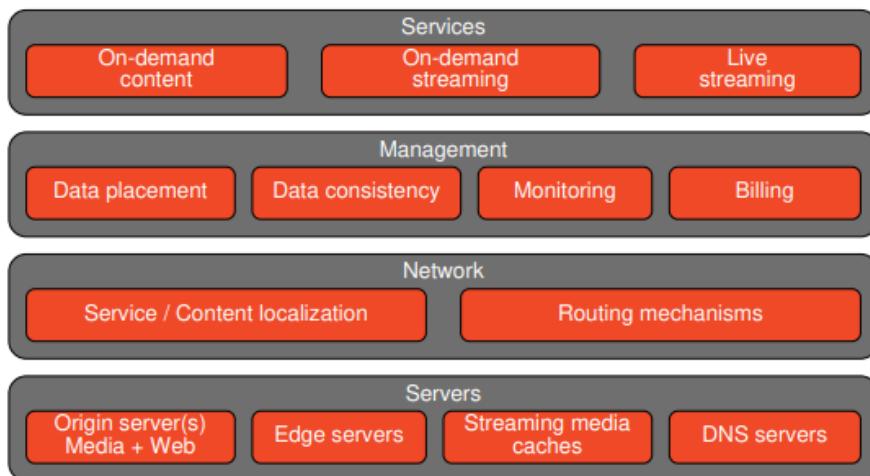
- Core: queste copie non servono le richieste di utenti, ma coordinano la replicazione di contenuti verso gli edge server. Possono dare indicazioni su come deve avvenire la replicazione.
- Edge Server: Sono i server che effettivamente rispondono con la risorsa all'utente. Vi sono alcuni edge-server “primari”, che sono in contatto con l'origin server che regolano la replicazione nel proprio cluster.

Come si può notare dall'immagine sottostante, una CDN riduce drasticamente il tempo di risposta e inoltre ne limita anche la **varianza**. Nonostante le prestazioni migliorino con le CDN, la risoluzione del DNS è più complessa e il tempo per la risoluzione è più alto.



L'adozione della CDN migliora i tempi di risposta

Componenti CDN La CDN usa delle tecniche di routing con algoritmi molto sofisticati per



garantire che il client richiedente la risorsa venga instradato alla copia più vicina e più veloce.
Meccanismi per la gestione dei contenuti

- **URL Rewriting:** La mappatura dei contenuti può essere basata sulla riscrittura delle pagine HTML. L'utente andrà a richiedere la pagina all'origin server che la CDN intercetterà e sovrascriverà gli embedded object con quelli presenti nella propria cache.
- **DNS Outsourcing:** Tramite questa tecnica è possibile evitare del tutto l'origin server grazie ai server DNS delle CDN. Tramite un meccanismo di risoluzione DNS intelligente è possibile, infatti, decidere se risolvere con l'IP dell'origin server o quello del server della CDN. Così facendo si può cachare la pagina e gli embedded object.

Le soluzioni possono comunque essere mixate ad esempio tramite URL rewriting all'origin server e al primary CDN server mentre si potrebbe utilizzare il DNS outsourcing per bilanciare il carico tra origin, primary e secondary servers.

4 Modelli di costo

Esistono vari modelli di business per il cloud computing:

- Servizi: Vendere le proprie competenze.
- Prodotti: Creare e vendere i propri prodotti (Software).
- Vendita: Vendere prodotti di terze parti.

4.1 Servizi

Necessita di un basso costo di setup iniziale, ma bisogna conoscere bene il modello di business. Il successo è determinato dall'introduzione di qualcosa di differente al mercato esistente da un punto di vista di qualità, competenze o prezzi. Il business richiede un dispendio di risorse umane ad alta intensità, non è scalabile come un prodotto. E' importante saper gestire le relazioni con i clienti, in quanto il lavoro spesso tende a compiacere quest'ultimo e i vincoli da lui imposti, che possono anche limitare la creatività del prodotto. Il mercato del cloud è basato sulla fiducia e la reputazione dato che i clienti spesso non hanno le competenze per valutare il servizio e si basano sulle reputazioni.

4.2 Prodotto

Ci si aspetta di ricavare grandi quantità di denaro in tempi ricorrenti, tuttavia la realtà è ben diversa in quanto i software e i prototipi non sono dei veri e propri prodotti ma bisogna cercare di indovinare il bisogno futuro di più clienti. Spesso si lavora su qualcosa senza garanzia di ritorno, che può essere molto grande come anche il rischio di fallimento. Costruire un nuovo prodotto può essere dispendioso sia da un punto di vista economica che psicologica. Da un punto di vista finanziario si avranno degli alti e bassi, infatti la fatturazione è basata sul tempo. Possiamo dividere due categorie di costi:

- CAPEX: Ammortizzati durante gli anni. Ci danno una riduzione del flusso di cassa disponibile
- OPEX: Fondi per gestire le attività quotidiane, si esauriscono entro l'anno di acquisto.

4.2.1 Metriche di costi

TCO

Costo totale di proprietà, tutti i costi diretti e indiretti (CAPEX e OPEX) di gestione di un bene durante la sua vita utile. Rappresenta non solo l'investimento iniziale ma anche tutti i costi associati all'utilizzo della risorsa, può durare dai 3 ai 7 anni. Per calcolare il tco, bisogna valutare le spese di hardware e software, il supporto gestionale e tecnico, tempo di formazione, eventuali viaggio e contratti di supporto, implementazione e costi di comunicazione. Per quanto riguarda i costi indiretti posso riguardare sia l'utente finale, sul sostegno e la formazione, ma anche il tempo di inattività. **Vantaggi TCO**

Direct Costs	Year 1	Year 2	Year 3	Total (\$)	% of Total Cost
Hardware					
Servers - 1	\$ 3,500.00	\$ -	\$ -	\$ 3,500.00	2.7%
Client computers - 10 @ \$2,450	\$ 24,500.00	\$ -	\$ -	\$ 24,500.00	18.9%
Peripherals - 3 @ \$1,200	\$ 3,600.00	\$ -	\$ -	\$ 3,600.00	2.8%
Network installation	\$ 4,600.00	\$ -	\$ -	\$ 4,600.00	3.5%
Maintenance fees	\$ -	\$ 3,930.00	\$ 3,930.00	\$ 7,860.00	6.1%
...	\$ -	\$ -	\$ -	\$ -	0.0%
Total Hardware Costs	\$ 36,200.00	\$ 3,930.00	\$ 3,930.00	\$ 44,060.00	33.9%
Software					
License - 10 users	\$ 15,000.00	\$ -	\$ -	\$ 15,000.00	11.6%
Maintenance fees	\$ -	\$ 2,700.00	\$ 2,700.00	\$ 5,400.00	4.2%
...	\$ -	\$ -	\$ -	\$ -	0.0%
Total Software Costs	\$ 15,000.00	\$ 2,700.00	\$ 2,700.00	\$ 20,400.00	15.7%
...	\$ -	\$ -	\$ -	\$ -	0.0%
Total Communication Fees	\$ -	\$ -	\$ -	\$ -	0.0%
Total Direct Costs	\$ 51,200.00	\$ 6,630.00	\$ 6,630.00	\$ 64,460.00	49.6%

Rappresenta non solo l'investimento iniziale ma considera anche tutti gli altri costi associati. Assicura un'analisi completa a lungo tempo.

Svantaggi TCO

Non considera i vantaggi delle varie opzioni non considerati, non da nessuna tempistica e comporta un costo di acquisizione elevato. Inoltre può essere difficile quantificare tutto. Per un'azienda considerare un TCO significare seguire una strategia sul minimo costo piuttosto che sul massimo rendimento.

ROI :

Il **ritorno sull'investimento** ha lo scopo di confrontare i costi di un progetto con il valore dei suoi risultati, viene espresso in percentuale di un periodo:

$$ROI = \frac{UtileNetto}{Investimento} * 100$$

Grazie a questo modello siamo in grado di analizzare la tempistica e l'entità degli investimenti, oltre che la considerazione di più scenari. Gli errori più comuni si verificano quando si sottovalutano gli investimenti o non si tiene conto del tempo impiegato dai dipendenti o del costo del capitale nel tempo. Esempio:

Costi totali: 114.000 €

Guadagni totali: 180.000 € / anno

ROI: 66.000 / 114.000 € * 100 = 57.8

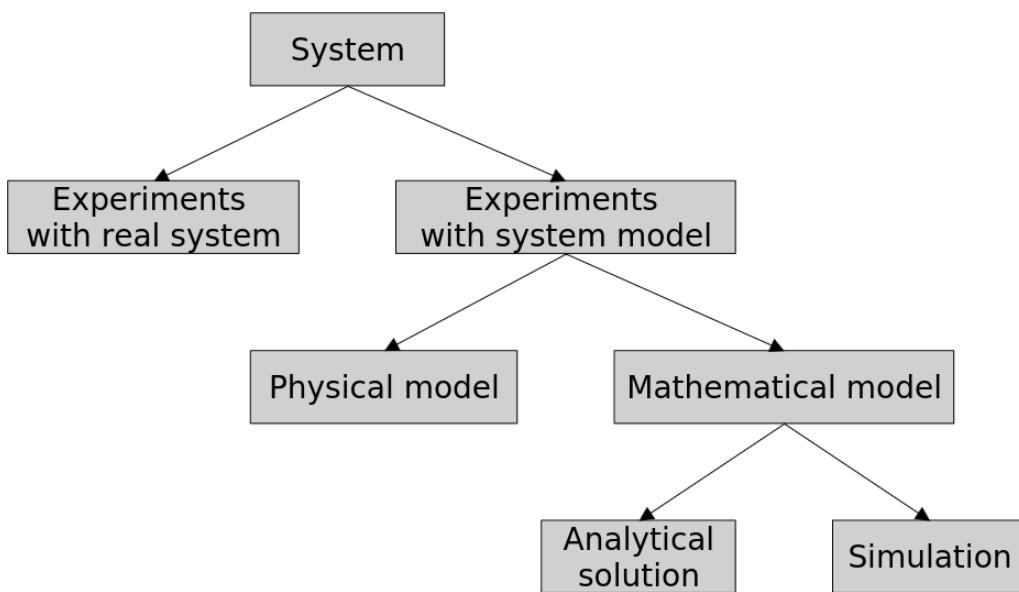
Performance evaluation & simulation

Vi sono due principali modi per fare performance evaluation: tramite modelli matematici e tramite simulazione.

I primi sono in grado di rappresentare approssimativamente un sistema con le sue caratteristiche. Generalmente vengono utilizzati per fare una prima stima della complessità di un problema.

I secondi, invece, permettono di simulare fedelmente o meno un sistema più complesso, poiché rappresentare un sistema di questo genere con modelli matematici inizia a diventare molto difficile.

Approcci alla performance evaluation



- esperimenti con sistemi reali: vengono fatti quando il sistema è molto semplice e contenuto. In questi casi è più conveniente di altri approcci;
- esperimenti con modelli del sistema stesso:
 - modelli fisici: sono rappresentazioni “in scala” del sistema stesso. Vengono generalmente rappresentati tramite scenari virtualizzati. Generalmente sono più costosi degli altri tipi di modelli, inoltre rappresentare degli scenari “what if” è più difficile (cosa succede se il tempo di risposta è 100ms al posto che 50?).
 - modelli matematici: vengono utilizzati quando anche il modello fisico in scala è difficile da produrre. Per scenari “what if” è molto versatile, è molto facile trovare numeri anche approssimativi di scenari con carichi molto sbilanciati o molto alti, che magari sono difficili

da rappresentare fisicamente. In linea del tutto teorica questi modelli matematici sono i più accurati, ma necessitano di trovare il livello di dettaglio giusto.

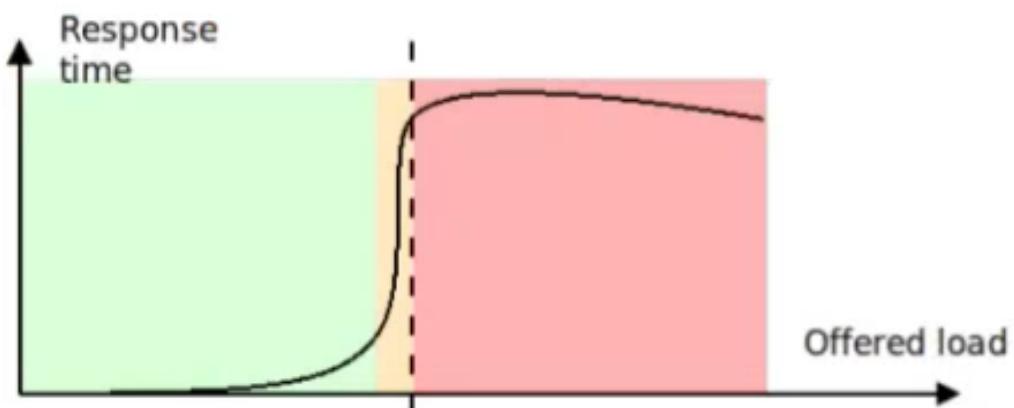
- **soluzioni analitiche**: modelli matematici per rappresentare il sistema tramite determinati costrutti matematici (generalmente variabili aleatorie); Uscire dai casi base del modello, tramite soluzioni analitiche, è molto più complesso, però, quindi meno versatile;
- **simulazione**: tramite software è possibile simulare il comportamento di un sistema. È la via di mezzo fra modello matematico e modello reale, quindi un buon compromesso. È inoltre molto più versatile del modello matematico, ma ha comunque degli intervalli di approssimazione di cui bisogna tenere conto.

I parametri prestazionale di maggior interesse sono:

- tempo di risposta: intervallo fra accettazione della richiesta e fine della risposta;
- throughput: volume di richieste soddisfatto nell'unità di tempo;
- error rate: frazione di richieste non servite correttamente;

Tutti questi parametri dipendono direttamente dal **carico corrente del sistema**.

Le classiche curve di carico di un server sono le seguenti:

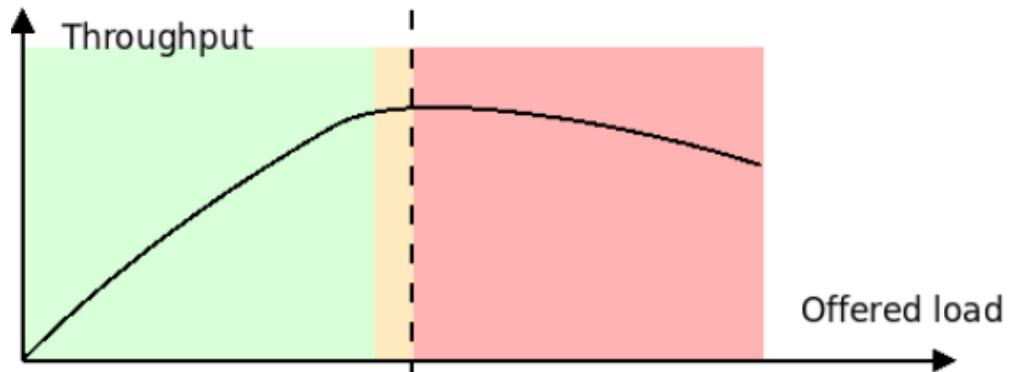


Il tempo di risposta varia in modo particolare: rimane molto basso finché l'offered load (ovvero le richieste per unità di tempo in arrivo al sistema) rimane nella capacità del sistema. Quando si arriva a saturare il sistema si entra nella cosiddetta **kneeling zone**, oltre la quale una piccola variazione di traffico fa variare di diversi ordini di grandezza il tempo di risposta.

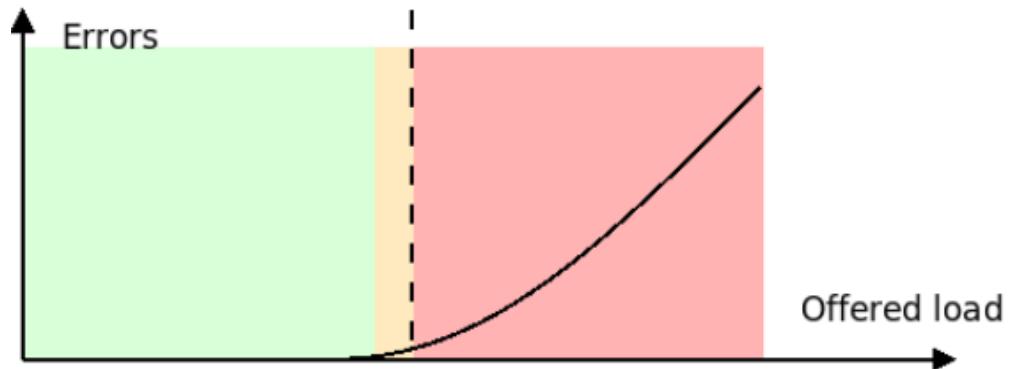
Ciò avviene non perché il sistema ci mette più tempo a processare le richieste (perché *non può* processare più richieste di quanto sia la sua capacità), ma perché il traffico in arrivo che non può essere processato sta venendo **accodato** in determinate strutture dati del sistema.

Più le code sono piene, più la probabilità di

droppare traffico aumenta. In questo caso, il sistema viene portato al *thrashing* (il sistema sta passando più tempo a gestire i context switch fra memoria virtuale o meno che a fare lavoro utile).



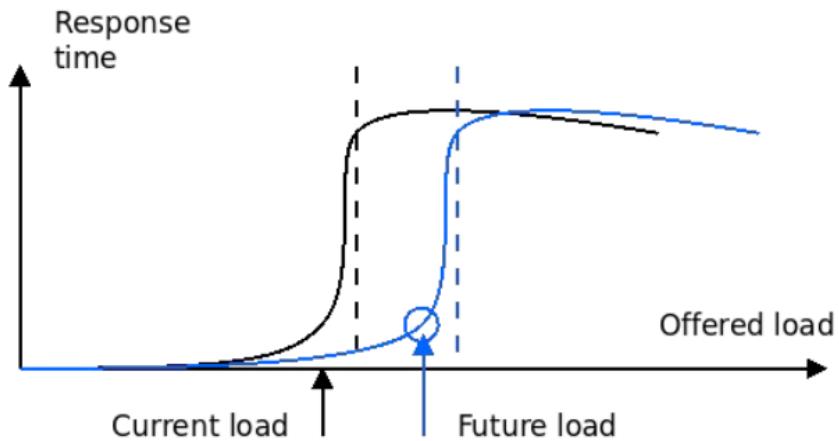
Il tempo di risposta aumenta, quindi, perché il throughput **satura**. All'arrivo nella zona di kneeling, il numero di richieste processate al secondo **stalla (per poi diminuire)**: ciò significa che il sistema non è in grado di processare **nuove** richieste nell'unità di tempo. Poiché le code si riempiono, poi, il throughput diminuisce, per via del fatto che le nuove richieste vengono **droppate**.



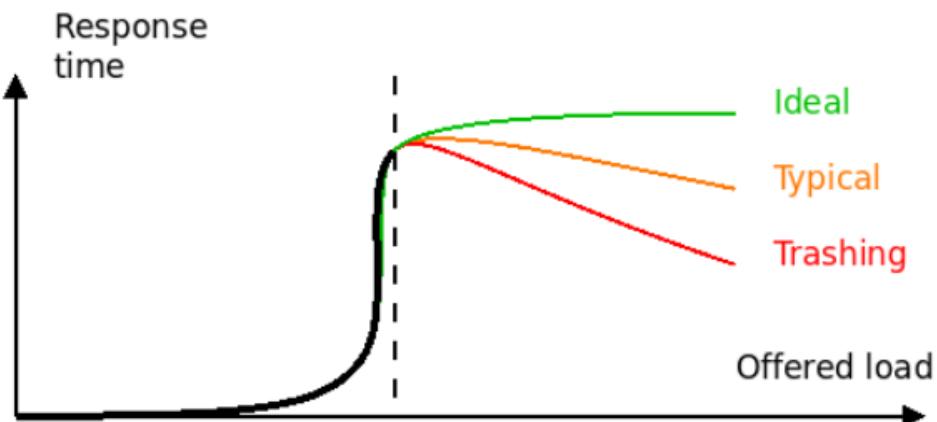
Ovviamente, all'aumentare dei problemi di questo genere, aumentano anche gli errori.

L'obiettivo è quello di avere un sistema che si trovi in un intorno della fine della zona di sottoutilizzazione (quella verde) \Rightarrow in questo modo il sistema non è né sottoutilizzato, né sovrautilizzato, ma sta venendo utilizzato quanto meglio per le sue capacità.

L'ampiezza di questo intorno della knee zone dipende anche dal **capacity planning** che viene fatto, ovvero con che "lungimiranza", rispetto alla quantità di traffico, viene costruito il sistema: quanto traffico **in più** è in grado di gestire il nostro sistema? Cosa succede se il traffico sale del 20%? E del 30%?



Graficamente, il capacity planning può essere rappresentato come lo “spostamento in avanti” del limite dato dalla kneeling zone.



Testing di sistemi reali

Un test su un prototipo viene effettuato, generalmente, come validazione dei calcoli teorici fatti a priori.

Esistono vari tipi di test che si possono fare su un sistema fisico:

- functional testing: sostanzialmente unit/e2e testing. Ovviamente non consente di catturare scenari particolari di malfunzionamento funzionale (race condition, ad esempio), o di workload (lentezza, memory leak, ...);
- activity testing: si testa l'integrazione del sistema. Si testano race condition, memory leak, velocità del sistema e quant'altro sotto condizioni normali (workload testing).

- endurance testing: si testa il sistema sotto il carico **massimo** che può andare a sottostare (alla kneeling zone);
- stress testing: si va oltre la kneeling zone e si controlla dove sta il limite di quest'ultima;
- benchmarking: si testa il sistema sotto carichi standard ⇒ sulle slide c'è tutto il discorso del falsamento dei benchmark.

Benchmark standard

Esistono due enti che producono benchmark: SPEC e TPC. Contengono modelli di workload e strumenti per eseguire raccolta dati in maniera “standard”.

SPEC, offre benchmark più “cloud-friendly”, fornendo misure riguardanti l'elasticità del sistema (tempo di scaling). TPC offre TPC-W per applicazioni cloud.

Oltre a SPEC, vi sono quelli di Yahoo YCSB (testing di database nosql, usato per testare cassandra), mentre Google ha creato PerfKit per benchmarking di applicazioni cloudnative.

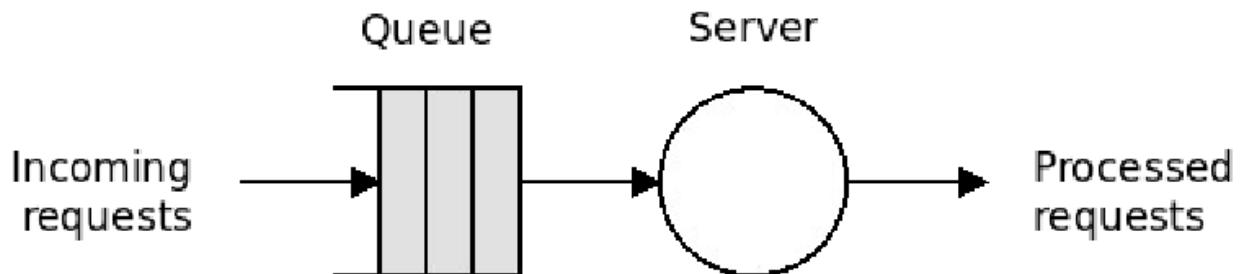
⇒ slide ok

Modelli matematici

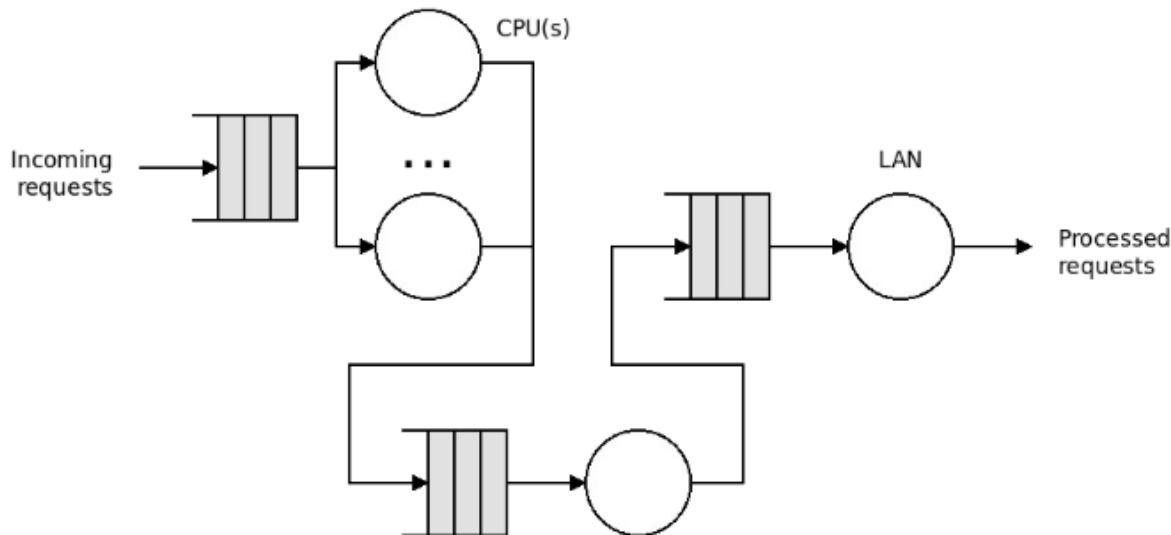
Immaginiamo un sistema composto da più parti: network, vari nodi frontend, vari nodi backend e ciascuna di queste parti siano caratterizzate da determinati tempi di risposta, come quello della CPU, quello dello storage e quant'altro.

Se prendiamo una qualsiasi di queste risorse, è possibile modellarla in funzione delle sue variabili rilevanti. Ciascuna di queste risorse, se modellata tramite la

teoria delle code, può essere rappresentata tramite una **coda** di processi in attesa di essere processati e un **servitore** che offre un determinato servizio con una determinata latenza.



Più risorse possono essere concatenate per rappresentare un sistema completo (un nodo, ovvero una macchina fisica/virtuale, ad esempio):



Per descrivere un singolo
nodo (single-core) del nostro sistema, si utilizza, generalmente:

- una coda
- carico di lavoro, identificato da:
 - λ_i : tasso di arrivo (in richieste al secondo) all'i-esima risorsa;
 - W_i : tempo di attesa medio nella coda, anch'esso modellato tramite una variabile aleatoria;
 - S_i : tempo di servizio della nostra risorsa i-esima ($1/\mu$), generalmente rappresentata da una densità di probabilità con un determinato valore medio. Se S_i è il tempo di servizio medio, avrò un tasso di processing pari a μ . Se, per processare una richiesta, ci si mette 1/10 di secondo, allora, in media, si serviranno 10 richieste/secondo (μ). μ è quindi il carico "massimo" che il sistema può sopportare.

Altre metriche rilevanti sono:

- R_i : tempo di risposta medio alla coda i-esima, dato da $S_i + W_i$;
- X_i : throughput all'i-esima coda, che sarà $\max(\mu, \lambda_i)$. Ad un dato istante, $X_i = \lambda_i * S_i$;
- X_0 : throughput di tutto il sistema;
- N_{iw} : all'i-esima risorsa vi sarà un numero di richieste in coda. N_{iw} è quindi la dimensione della coda dell'i-esima risorsa;
- N_{is} : numero di richieste che stanno venendo servite in un dato momento (in base a quanti servitori ha una risorsa, i.e. CPU);
- N_i : numero di richieste all'i-esima risorsa, ovvero $N_{is} + N_{iw}$.

Legge dell'utilizzazione e Legge di Little

L'utilizzazione è la **probabilità di avere il sistema “busy” ad un dato istante**, ovvero che sta servendo e che ha qualcosa in coda.

- B_i the time the serve is busy in observation period τ
- C_i completed requests in period τ
- Utilization $U_i = B_i/\tau$
 - But $X_i = C_i/\tau$
 - Hence $1/\tau = X_i/C_i$
 - $U_i = B_i * X_i/C_i = (B_i/C_i) * X_i = S_i * X_i$
- In a steady state: $X_i = \lambda_i * S_i$
 - Often used notation: $\rho = \lambda/\mu$

L'utilizzazione è scrivibile come:

$\rho = \lambda/\mu$, tasso di arrivo su tassi di uscita. Rappresenta qual è la percentuale di tempo in cui il sistema è occupato!

Legge di Little

La Legge di Little stima quante richieste vi sono all'interno del sistema. Questa legge non fa alcuna assunzione, pertanto è applicabile a tutti i contesti (black box).

I dati che servono per usare questo teorema sono:

- X : throughput del sistema
- R : tempo speso nel sistema
- viene ricavato N , il numero di richieste
- $N = X * R$ oppure $N = \lambda * W$

Ovviamente, la Legge di Little è declinabile in tutti gli aspetti visti prima:

- $N_{iw} = X_i * W_i$

- $N_{is} = X_i * S_i$
- $N_i = X_i * R_i$

Open/Closed loop models

È possibile modellare la popolazione di utenti in due principali modi:

- closed loop: la popolazione di utenti è finita. Un utente entra, fa una richiesta, entra in think time, e rifà un'altra richiesta e quant'altro;
- open loop: la popolazione di utenti è potenzialmente infinita, si catturano solo le richieste che arrivano.

In sistemi open loop, la formulazione della Legge di Little è quella già vista, mentre nei casi **closed**, la formulazione cambia, aggiungendo il tempo di *think Z* e la grandezza della popolazione M:

$$R + Z = M/X$$

Teoria delle code

La teoria delle code descrive, tramite equazioni, un sistema simile a quelli già descritti.

Tutti gli eventi del sistema, come l'arrivo dei nuovi job, il suo processing e altro, vengono descritti tramite

variabili aleatorie, pertanto tramite processi stocastici (distribuzioni di probabilità).

Lo scenario classico è quello di un sistema ad anello aperto con due processi, entrambi rappresentati con una variabile aleatoria. Sulla base di queste variabili è possibile categorizzare i sistemi in determinate classi mediante la sigla:

{Processo di arrivo}/{Processo di servizio}/{Numero servitori}/{Lunghezza coda}

I due processi possono essere:

- M \Rightarrow poissoniano, il processo è quindi "senza memoria". La probabilità, quindi, di passare in uno stato, ovvero il numero di job dentro il sistema, è indipendente dallo stato attuale.
- D \Rightarrow deterministico, il tempo di attesa di quel processo è quindi **fisso** e non cambia mai;
- E_k \Rightarrow erlang, la somma di **k variabili gaussiane** di parametro λ ;
- G \Rightarrow generico, sistemi in cui la variabile è una generica variabile aleatoria.

M/M/1 \Rightarrow arrivo e processo poissoniani, con numero di servitori = 1.

Questo vuol dire che l'arrivo e il processo dell'evento non è deterministico e **senza memoria**.

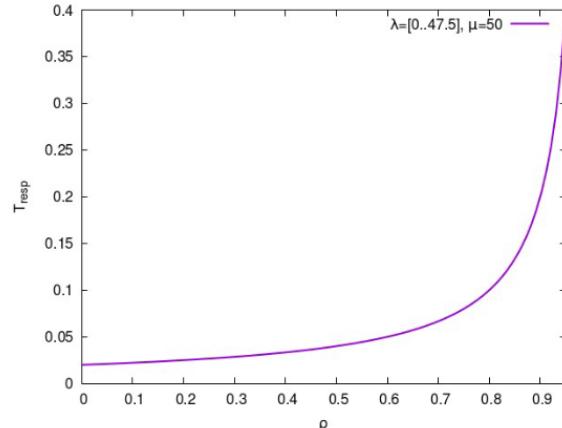
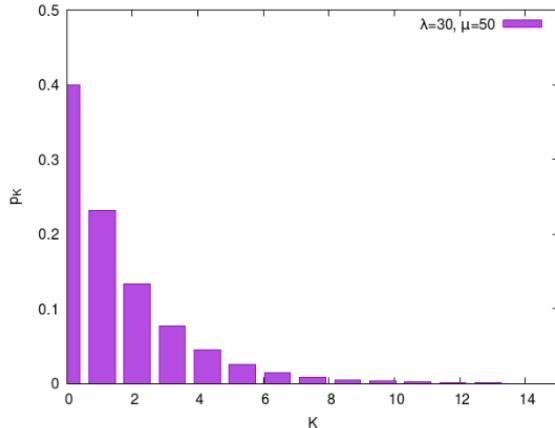
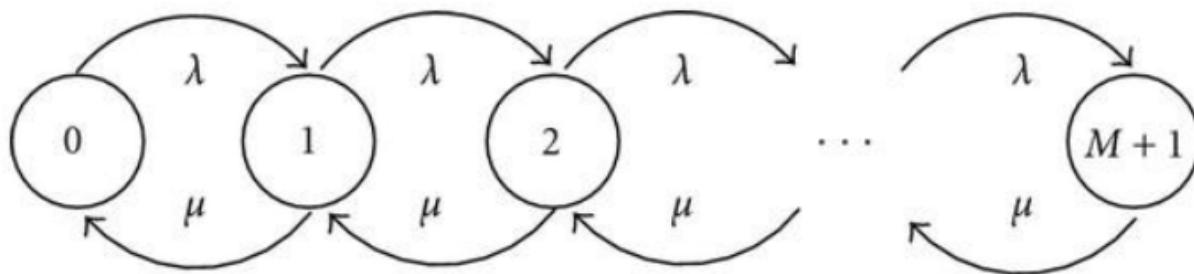
Processi poissoniani

I processi poissoniani sono molto semplici da modellare. Prendendo come esempio il nostro servitore con coda, questo può:

- avere un tempo di interarrivo λ , che rappresenta anche la **probabilità con la quale il sistema passa da uno stato ad un altro**. Con "stato" si identifica il **numero di richieste nel sistema**, quindi la probabilità di arrivo di un pacchetto rappresenta la probabilità che il numero di richieste presenti nel sistema passi da N a $N + 1$;
- avere un tempo di servizio μ , che rappresenta con che probabilità si passa da N a $N - 1$, ovvero la probabilità di **uscita**, appunto, dal sistema. Notare come sia **μ che λ sono due variabili poissoniane!**
- avere un'utilizzazione, quindi, data da $\rho = \lambda / \mu$;

La probabilità di essere allo stato k , quindi, è:

$$P(k) = (1 - \rho) * (\rho)^k$$



Il **tempo di risposta** Tresp è quindi identificato da:

Tresp = $1 / (\mu - \lambda)$, pertanto è una **curva esponenziale** con una determinata **varianza e media**. Questo perché, ovviamente, se aumenta la capacità di processare (μ), il tempo di risposta **cala**, ma se aumenta il tasso di arrivo (λ) il tempo **sale**!

$T_{\text{resp}} = (1 / \mu) * (1 / 1 - \rho) \Rightarrow$ Quando $\mu = \lambda$, le code sono **sature**, pertanto il tempo di risposta diverge all'infinito. Questo torna, poiché il tempo di risposta non è altro che $S + W$ (tempo di servizio + tempo di attesa), quindi:

- $S = 1 / \mu$
- $W = S * (\rho / 1 - \rho) = 1 / \mu * (\rho / 1 - \rho)$

PASTA theorem (Poisson Arrivals See Time Averages)

Describe i sistemi M/G/1: il servizio è quindi una distribuzione qualsiasi.

Esprime quindi, lo stato che ci si aspetta di trovare in un sistema a code che vede un processo di arrivo **poissoniano** e una qualsiasi distribuzione generica.

Un processo di arrivo poissoniano, quindi, esprime un arrivo in un qualsiasi istante di tempo, quindi randomico, espresso dunque da una variabile aleatoria **gaussiana**.

Sapendo che:

- $Cv = \sigma / \mu$ (**μ in questo caso è la media di una variabile aleatoria**)
- $Cv(\text{poisson}) = 1$

e data la formula di **Pollaczek–Khinchine**:

$$\begin{cases} E[W] = \frac{\lambda E[S^2]}{2(1 - \rho)} &= \frac{1 + C_v^2}{2} \cdot \frac{\rho}{1 - \rho} \cdot E[S] \\ E[T] = E[S] + \frac{\lambda E[S^2]}{2(1 - \rho)} &= \left(1 + \frac{1 + C_v^2}{2} \cdot \frac{\rho}{1 - \rho}\right) \cdot E[S] \end{cases}$$

- $E[W]$: average waiting time
- $E[T]$: average response time
- $E[S]$: average service time ($1/\mu$)

Il numero **medio** di utenti in un sistema è:

$$N = \rho + \frac{\rho^2 + \lambda^2 Var(S)}{2(1 - \rho)}$$

per una distribuzione M/G/1.

Note:

- una distribuzione M/M/1 è un caso particolare di M/G/1. Sapendo che Cv(poisson) = 1, ottengo la definizione tempo di attesa medio poissoniano, ovviamente:

$$E[W] = \frac{1 + C_v^2}{2} * \frac{\rho}{1 - \rho} * E[S] = \frac{1 + 1}{2} * \frac{\rho}{1 - \rho} * E[S] = \frac{\rho}{1 - \rho} * \frac{1}{\mu}$$

- stessa cosa per il tempo di risposta M/M/1:

$$E[T] = \left(1 + \frac{1 + C_v^2}{2} * \frac{\rho}{1 - \rho}\right) * E[S] = \left(1 + \frac{\rho}{1 - \rho}\right) * \frac{1}{\mu} = \frac{1}{1 - \rho} * \frac{1}{\mu} = \frac{1}{\mu - \lambda} = T_{resp}$$

Avendo delle distribuzioni molto varianti per le distribuzioni di servizio (G), ciò che succede è che il tempo di attesa in coda **aumenta** rispetto al modello poissoniano! Quindi, usando modelli troppo semplici (quello poissoniano), si va a sottostimare l'impatto che ha la varianza nel nostro modello.

Approssimazione di Allen-Cuneen

Per i sistemi G/G/N vale l'approssimazione di Allen-Cuneen.

$$W_M = \frac{P_{cb,N}}{\mu N(1 - \rho)} \left(\frac{C_S^2 + C_D^2}{2} \right), \quad P_{cb,N} \approx \begin{cases} (\rho^N + \rho)/2 & \text{if } \rho \geq 0.7 \\ \rho^{\frac{N+1}{2}} & \text{otherwise.} \end{cases}$$

- W_M Average waiting time
- $P_{cb,N}$ Probability that all servers are busy
- C_S, C_D , Coefficient of variation
 - arrival (C_S)
 - Service (C_D)
 - Coefficient of variation = Std. Dev / Average

Questa formula esprime il comportamento di un sistema nel quale né il processo degli arrivi né quello di servizio è vincolato ad una determinata distribuzione di probabilità. Si noti come il quadrato del coefficiente di variazione viene considerato sia in arrivo che in servizio, ovviamente, un po' come nel modello P-K.

“A dirty trick” - spiegazione delle slide

Dobbiamo trovare λ^* tale per cui il tempo di risposta viene dimezzato, sapendo di avere un processore che è un 10% più veloce (in realtà) della competizione.

- We need to find the right load λ^*
 - Goal: **2x gain on response time**
- Real advantage
 - Throughout speedup is **K**
 - Competitor processor: 50 req/s
 - Our processor: 55 req/s $\rightarrow K=1.1$
- Assume a simple setup:
 - Poisson request arrival
 - Poisson service

Si suppone di essere in M/M/1.

- $\mu_1 = 50$
- $\mu_2 = 55$
- voglio fare in modo che $T_{resp1} = 2T_{resp2}$. Quanto vale lambda?

Sapendo che $T_{resp} = 1 / \mu - \lambda$, e sapendo quanto valgono i due μ , posso eguagliare i termini:

$$\begin{aligned}
 & \left(\frac{\lambda}{\mu} + \frac{1}{1-p} \right) \bar{\mu} = \left(\frac{1}{1-p} \right) \mu \\
 & = \frac{1}{1-p} \cdot \frac{1}{\mu} \\
 T_{R_1} & = 2 T_{R_2} \rightarrow \frac{1}{\mu_1 - \lambda} = 2 \frac{1}{\mu_2 - \lambda} \\
 \frac{1}{50-\lambda} & = 2 \frac{1}{55-\lambda} \rightarrow \frac{1}{50-\lambda} - \frac{2}{55-\lambda} = \frac{55-\lambda - 2(50-\lambda)}{(50-\lambda)(55-\lambda)} \\
 \cancel{-10} & \rightarrow \cancel{55-\lambda} = 100 + 2\lambda = 100 + 2\lambda \\
 \cancel{100-2\lambda} & \rightarrow \cancel{55} = -\cancel{\lambda} \Rightarrow \lambda = 45 \\
 & \lambda = 45
 \end{aligned}$$

Altro esercizio (sovraffollato) - spiegazione

- We have an incoming load $\lambda > \mu$
- We want to scale-up our system $\rightarrow K$ servers
- Assumption:
 - Requests can be distributed uniformly over servers
- How many servers do I need to avoid overload?
- Assumption
 - I have a maximum response time T_{SLA}
- How many servers do I need to meet the SLA?
- Volunteers welcome

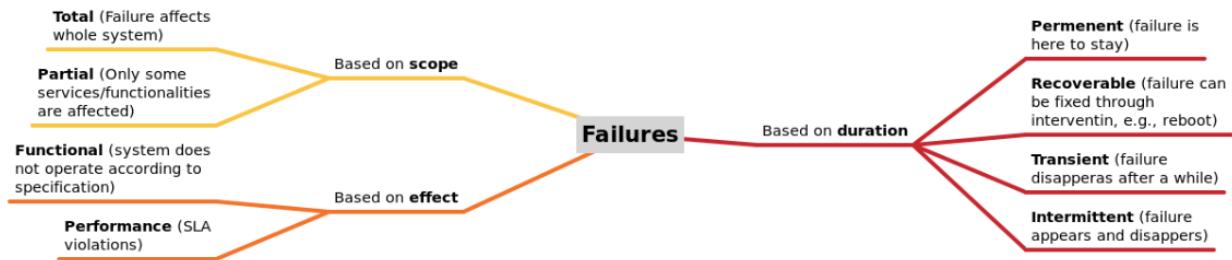
Supponendo, quindi, che il sistema sia in sovraccarico ($\lambda > \mu$), e sapendo che distribuisco uniformemente su K server, posso dire che il tempo di risposta diventa:

$$T_{resp} = \frac{1}{\mu - \frac{\lambda}{k}}$$

A questo punto, non mi resta che, una volta impostato TSLA, usare la formula con K incrementale, finché non ho trovato che:

$$T_{resp} \leq T_{SLA}$$

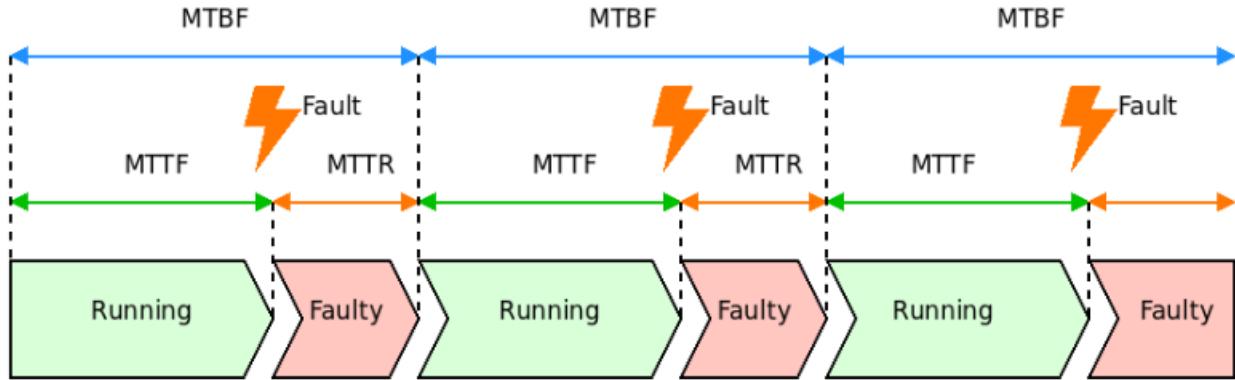
Failures



Ci sono failure di tipo di verso. Le failure possono essere **funzionali**, nelle quali il sistema non funziona effettivamente più, oppure **prestazionali**, nelle quali il sistema funziona correttamente, ma non soddisfa i requisiti di SLA.

Alcune metriche che si possono utilizzare per rappresentare il **tasso** di failure in un sistema sono:

- MTBF: mean time between faults;
- MTTF: mean time to fault;
- MTTR: mean time to repair;
- $MTBF = MTTF + MTTR$



⇒ scopo del gioco: la frazione rossa deve essere quanto meno grande possibile.

Su queste metriche è possibile definire il concetto di **availability**: $A = \text{MTTF} / \text{MTBF}$, ovvero la probabilità che **ad un istante t**, il sistema stia funzionando. È una misura istantanea.

Sulla metrica di availability, si definiscono varie

classi di availability:

Availability Class	Availability	Unavailable [min/y]	System Type
1	90.0%	52560 (36d)	Unmanaged
2	99.0%	5256 (3.6d)	Managed
3	99.9%	526 (8.8h)	Well-managed
4	99.99%	52.6	Fault-tolerant
5	99.999%	5.3	Highly Available
6	99.9999%	0.53	Very Highly Available
7	99.99999%	0.053	Ultra Available

Oltre all'availability, abbiamo la **reliability R**, che sostanzialmente è lo stesso concetto della availability, spalmato su un periodo T di tempo. Se il periodo T è sufficientemente grande, allora $A = R$.

Fault tolerance

La fault tolerance si basa su due principali concetti:

- error detection: senza error detection, non si possono scoprire gli errori durante il funzionamento di un sistema. Vi sono due modi per fare error detection:
 - concurrent detection: viene fatto durante il funzionamento del sistema;
 - preemptive detection: viene fatta quando il servizio offerto dal sistema viene sospeso. Ad esempio, se devo testare un hardware faulty, devo per forza spegnere l'hardware e installare le varie sonde che verificano lo stato dell'hardware, prima di procedere.

- recovery: come si rientra dai fallimenti all'occorrere di una failure. Prevede:
 - error handling: correzione dell'errore in maniera quanto più efficace e **veloce** possibile. Sostanzialmente è il MTTR, quindi, generalmente, non è un processo istantaneo:
 - rollback: si ritorna all'ultimo "safe state";
 - rollforward
 - compensation: si utilizza la ridondanza per gestire l'errore. Ad esempio, un load balancer principale in heartbeat con un altro load balancer di backup, che viene messo online automaticamente quando il load balancer principale va offline;
 - fault handling: prevenire che gli errori ricapitino:
 - diagnosis
 - isolation
 - reconfiguration
 - re-initialization/fix

Più sistemi, con diversi tassi di reliability r_i e availability a_i , possono essere collegati **in serie o in parallelo**.

- nei sistemi **in serie**, il fallimento di un componente prevede il fallimento di tutto il sistema, pertanto il tasso di reliability del sistema è:

$$R = \prod r_i$$

- nei **in parallelo**, il fallimento di tutto il sistema avviene solo se tutti i sottosistemi falliscono, pertanto:

$$R = 1 - \prod(1 - r_i)$$

Esercizi chiesti in sede d'esame:

Given subsystem with identical reliability

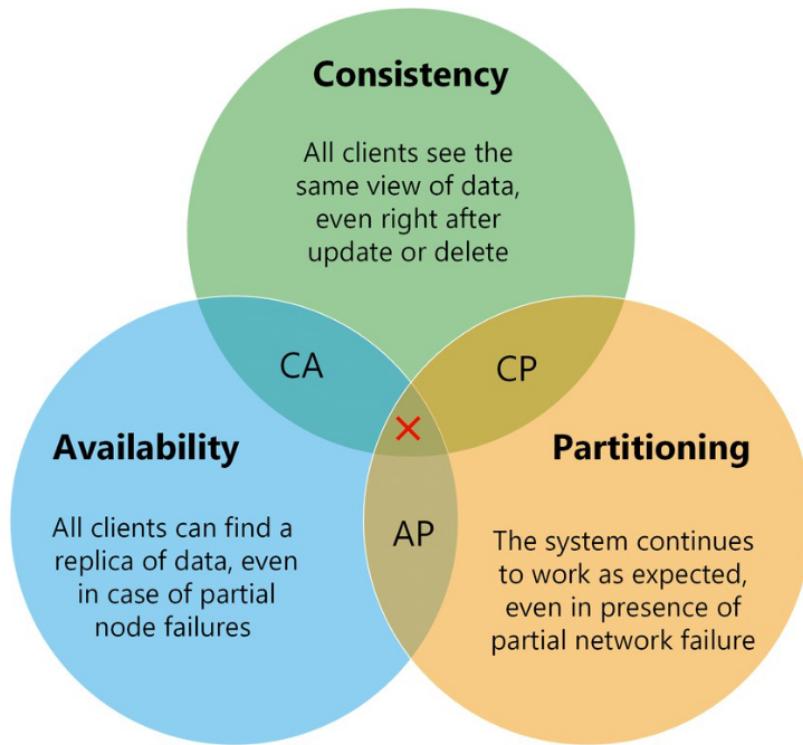
In a parallel system

- How many subsystem do I need to guarantee a reliability of X?

In a serial system

- What is the maximum number of subsystem to guarantee a reliability of X?

CAP theorem



Il CAP theorem definisce tre proprietà di un sistema:

- consistency
- availability

- partitioning

Ed enuncia che un sistema potrà avere **solo due di questi tre proprietà.**

Simulation

Quando i modelli matematici iniziano a diventare molto complessi, è utile avere un modello di simulazione che permetta di avere una forma chiusa più semplice da realizzare e **meno difficile (e approssimata)** da risolvere!

Attraverso la simulazione è possibile gestire vari scenari del sistema in maniera molto veloce (basta cambiare i parametri di simulazione) con diversi livelli di dettaglio, per catturare il comportamento del sistema in modo quanto più preciso possibile.

Vi sono due tipi di simulazione:

- statica: una simulazione è statica quando non viene considerato il fattore del tempo nella simulazione;
- dinamica: vi è una rappresentazione dello scorrere del tempo;

Che possono essere:

- stocastiche: i processi del sistema vengono modellati tramite variabili aleatorie;
- deterministiche: i processi hanno comportamenti ben definiti;

I modelli possono essere:

- discreti: rappresentati da **eventi** che accadono all'interno del sistema. Il tempo viene quindi frazionato in unità di tempo e poi gli eventi andranno ad occupare "slot di tempo";
- continui: il tempo non può essere frazionato in slot.

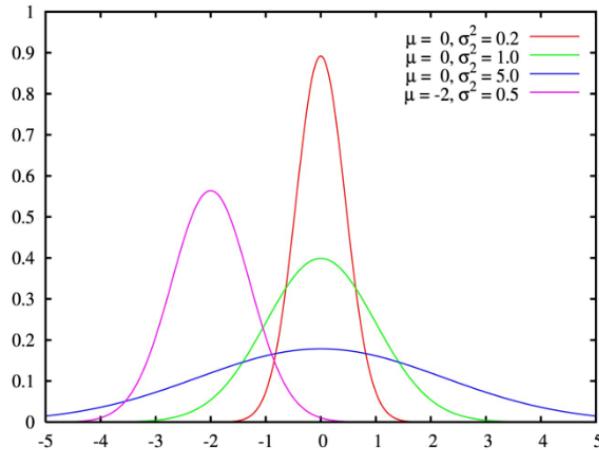
⇒ le slide sono autoesplicative;

Richiami di statistica

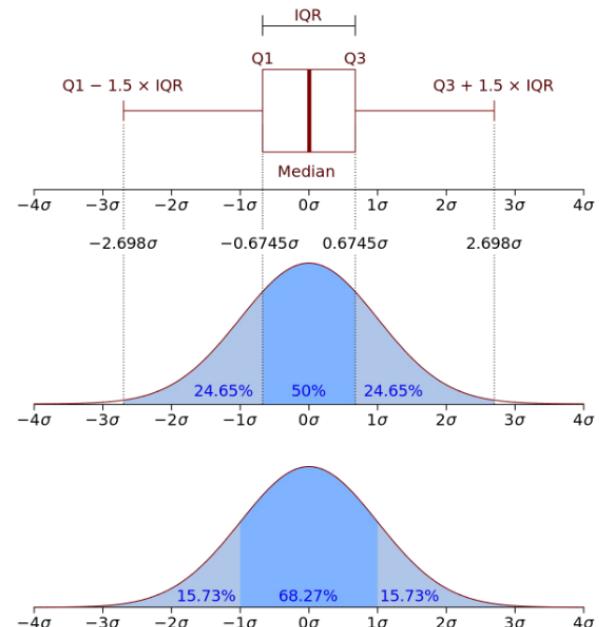
- **Experiment:** a sample of a random variable X
- **Sample space S:** the set of possible outcomes (Co-dominion of random variable)
 - Each sample belongs to S
 - Example:
 - Coin toss → $S=\{H, T\}$
 - 1d6 roll → $S=\{1, 2, \dots, 6\}$
 - S can be a set of finite values, an interval or \mathbb{R}
- **Probability density:**
 - $P(x) = P(X=x)$
 - The sum of probabilities is 1
 - $\sum_{x \in S} p(x)=1$
 - $\int p(x) dx = 1$
- **Average value** $\mu=E(X)$
 - $E(X) = \sum x p(x)$
 - Relevant properties:
 - $E(cX) = cE(X)$
 - $E(\sum_i c_i X_i) = \sum_i c_i E(X_i)$
- **Median** value
 - Value x: $P(X \leq x) = 0.5$
- **Quantiles** are defined in a similar way
- **Variance** $\sigma^2=E[(X-\mu)^2]=E(X^2)-\mu^2$
 - $E[(X-E(X))^2]=E[X^2 -2X E(X) +E(X)^2]=$
 $=E(X^2) -2E(X)E(X) + E(X)^2=E(X^2) -2E(X)^2 + E(X)^2$
- **Standard Deviation** $\sigma= \sqrt{\sigma^2}$

- Variables can be:
 - Independent
 - Correlated
- Independent variables can be studied separately
- For correlated variables the probability distribution are intertwined
 - $P(X=x)=f(P(Y=y))$
- In simulation variables are often correlated
- To obtain statistically valid results we need to achieve variable independence
- In simulation we repeat experiments:
 - Each experiments starts with a different seed for RNG
 - Experiments are statistically independent
- From central limit theorem
 - $F(X)=P(X < x)$
 - $F(X) \rightarrow G(X)$ for $N \rightarrow \infty$
 - $G(X)$ = Gaussian function
- Law of large numbers
 - N = number of observations
 - $N \rightarrow \infty$ means that sample average converges to $E(X)$
- teorema del limite centrale: data una collezione di variabili statisticamente indipendenti, queste andranno a disporsi come una distribuzione gaussiana. Con abbastanza ripetizioni di un esperimento, è possibile modellare queste ripetizioni tramite una gaussiana!

- Meaning of **variance**:
- **Low variance**:
→ High and narrow peak
- **High variance**:
→ Low and wide peak



- Analysis of a **Gaussian distribution**
- Median=Average
- **Confidence**:
 - $\mu \pm \sigma \rightarrow 68\% \text{ confidence}$
 - $\mu \pm 3\sigma \rightarrow 99\% \text{ confidence}$
- Need to estimate μ and σ from repeated experiments



Generalmente, gli input dei simulatori vengono modellati tramite **variabili aleatorie!**

L'implementazione di queste variabili aleatorie possono essere fatte mediante o partendo da delle tracce, quindi avendo un dataset di dati "storici", oppure mediante un modello matematico, più o meno approssimato. Generalmente il modello matematico viene interpolato da dataset "storici".

Le tipiche curve che descrivono gli input ai modelli sono **gaussiana e poissoniana**.

- n.b: una normale è definita da scarto quadratico medio e valor medio.

Il valor medio è dato da:

$$\mathbb{E}[X] = \sum_{i=1}^{\infty} x_i p_i.$$

Caso finito

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x f(x) dx.$$

Caso continuo

ed è sostanzialmente il prodotto tra il valore di ciascun campione e la **probabilità** del verificarsi di tale campione. Rappresenta il centro della campana della nostra distribuzione.

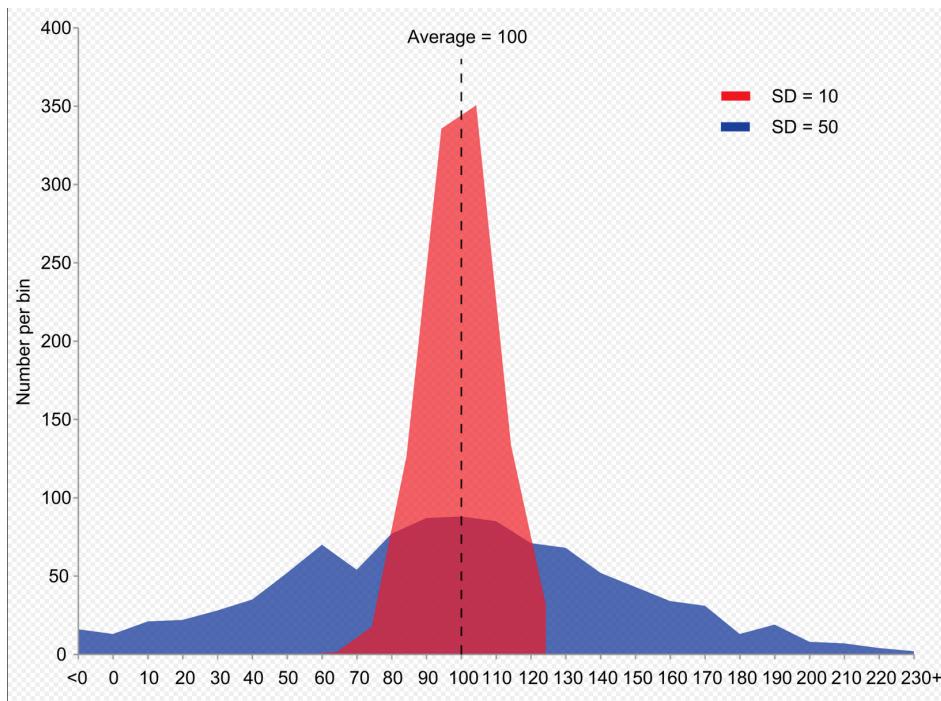
Lo scarto quadratico medio (deviazione standard) è dato da:

$$\sigma_X = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \mu_X)^2},$$

e rappresenta quanto “varia” il valore medio. Se elevato al quadrato, abbiamo la varianza, che rappresenta “l’apertura” della nostra campana.

La campana viene tracciata tramite la funzione di densità di probabilità:

$$f(x) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{1}{2} \left(\frac{x-\mu}{\sigma}\right)^2}$$



Due gaussiane con varianza diversa, ma media uguale.

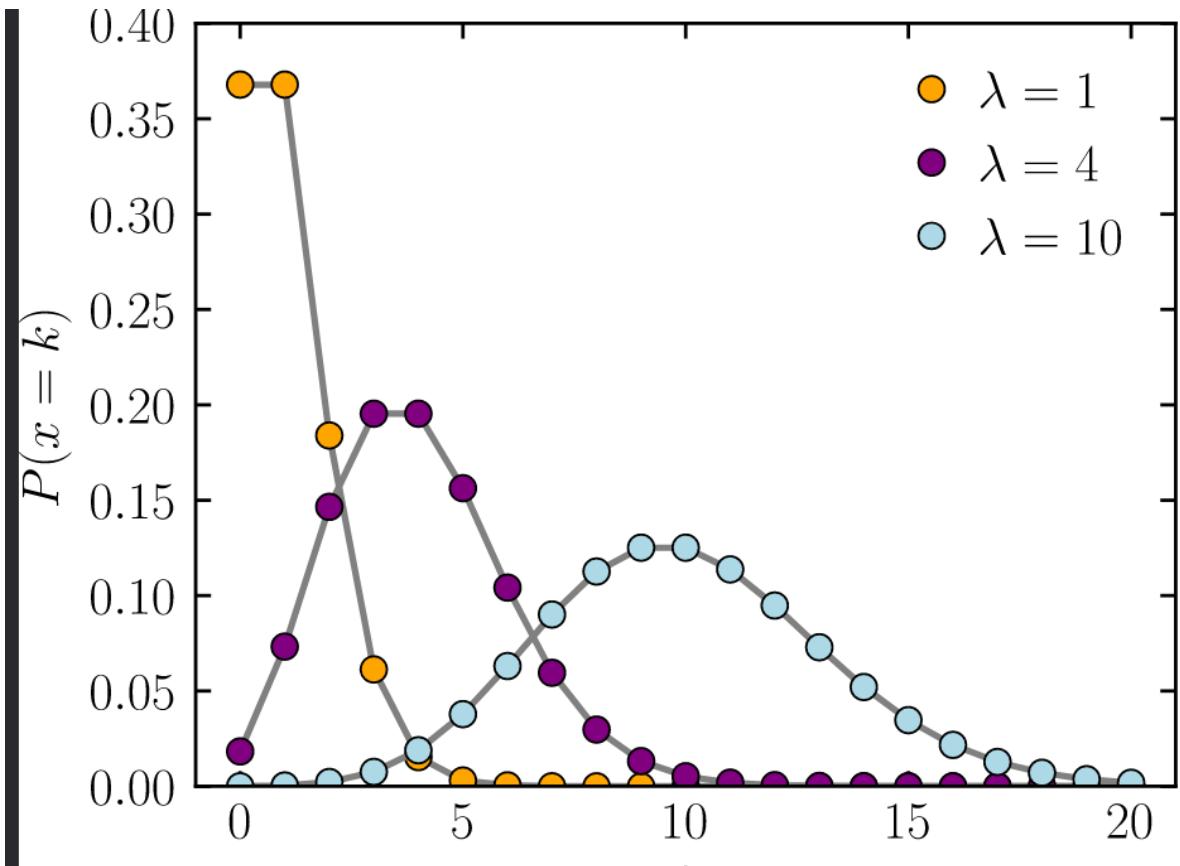
- n.b: poisson:

In teoria delle probabilità la distribuzione di Poisson (o poissoniana) è una distribuzione di probabilità discreta che esprime le probabilità per il numero di eventi che si verificano successivamente ed indipendentemente in un dato intervallo di tempo, sapendo che mediamente se ne verifica un numero λ . Ad esempio, si utilizza una distribuzione di Poisson per misurare il numero di chiamate ricevute in un call-center in un determinato arco temporale, come una mattinata lavorativa. Questa distribuzione è anche nota come legge degli eventi rari.

Media e varianza coincidono a λ .

La distribuzione di una v.a. poissoniana è:

$$P(x) = \frac{\lambda^x}{x!} e^{-\lambda}$$



- funzione di densità di probabilità = funzione di probabilità “continua”, ovvero quella funzione che associa ad ogni x la probabilità $P(X = x)$, dove X è la variabile aleatoria.

Virtualizzazione

La virtualizzazione è, ovviamente, una delle enabling technologies del cloud computing, che permette di adibire hardware, che magari viene sottoutilizzato, per altre operazioni, per evitare lo “spreco” di risorse (**server consolidation**) ⇒ consolida le risorse su un'unica macchina.

Un altro scenario molto importante è quello della **business continuity**: la virtualizzazione permette di risolvere il problema riguardante i malfunzionamenti delle macchine fisiche. Poiché è possibile migrare le macchine virtuali su altri nodi, qualora uno di questi abbia problemi,

Inoltre, è possibile partizionare ambienti di applicazioni per scenari di testing (produzione, staging e quant'altro) in modo molto semplice (**software development and testing**) tramite lo **snapshot** di macchine virtuali.

Concetti base

La virtualizzazione permette di creare una versione **virtuale** di una determinata risorsa: CPU, GPU, RAM, storage o risorse network.

Questa tecnologia, generalmente, viene realizzata tramite un sistema che permette di garantire il provisioning e il deprovisioning (anche automatico) delle suddette risorse, gestendo il tutto in maniera molto flessibile.

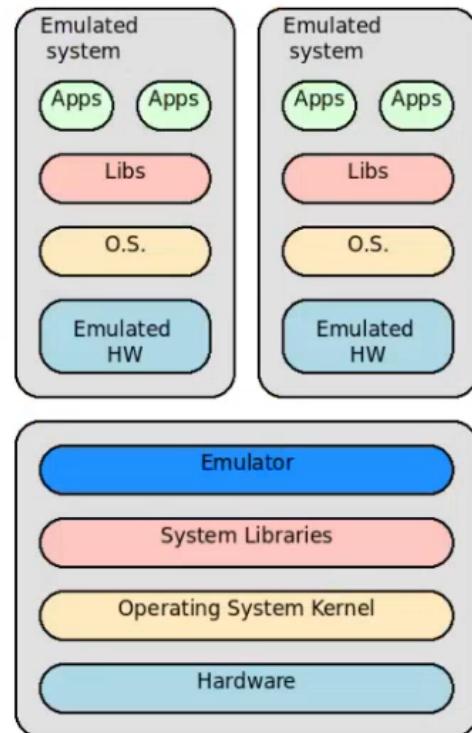
La virtualizzazione può avvenire su diversi livelli:

- emulazione: viene simulata **l'architettura** di un determinato processore! Per far girare contesti emulati è necessario **tradurre le operazioni macchina che vengono generate dal processore**;
- virtualizzazione: viene virtualizzato l'hardware che viene assegnato alla risorsa da virtualizzare (macchine virtuali). L'hardware che viene virtualizzato è quello dell'architettura sulla quale l'hypervisor sta girando, ma ad ogni macchina virtuale può essere assegnato hardware diverso (sulla stessa macchina);
- containerizzazione: viene creato un layer di virtualizzazione al di sopra del **kernel del sistema operativo**. Quindi, tutti i container che girano su una macchina

condivideranno lo stesso **kernel**, ma vivranno in uno spazio di processi isolato dagli altri.

Emulazione

- Recreation of hardware components
- Emulated hardware
 - Can be **different** form the hardware where the emulator runs
 - Huge **overhead**
 - Highest **flexibility**

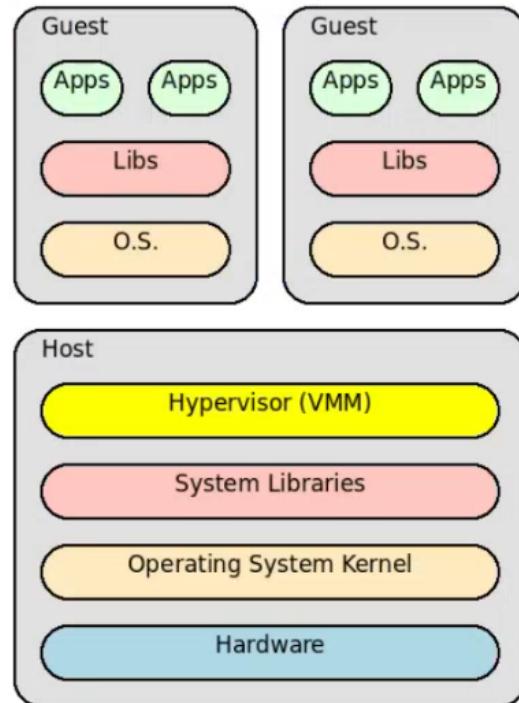


Notare come i **guest** hanno emulato anche l'hardware (processore, genericamente).

Virtualizzazione

Con la virtualizzazione l'host si estende fino all'hypervisor, il componente dello stack che si occuperà di creare le macchine virtuali (guest). Pertanto tutte le macchine virtuali vedranno come set di istruzioni quelle fornite dall'hardware dell'host, ma potranno avere sistemi operativi diversi, poiché il layer di virtualizzazione si estende soltanto fino all'hardware stesso.

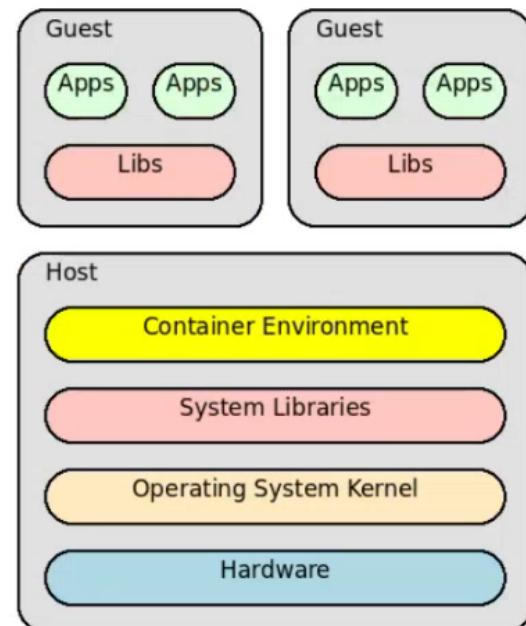
- Terminology:
 - **Hypervisor** → software for running VMs
 - **Virtual Machine (VM)**
 - **Guest** → a VM
 - **Host** → the node where the hypervisor runs
- VM has same CPU as the Host
 - Multiple VM on same Host
 - Different VMs can run **different Operating Systems**



Contanerizzazione

Con la containerizzazione non vi è più un hypervisor, ma vi è un software backend che regola l'esistenza dei container (i guest).

- Multiple separated environments
 - **Group** of processes for each environment (e.g., cgroups)
 - **Separation** between process groups
- All process groups share the **same kernel**
- Typically implemented over Linux Kernel
- Additional software for **management**



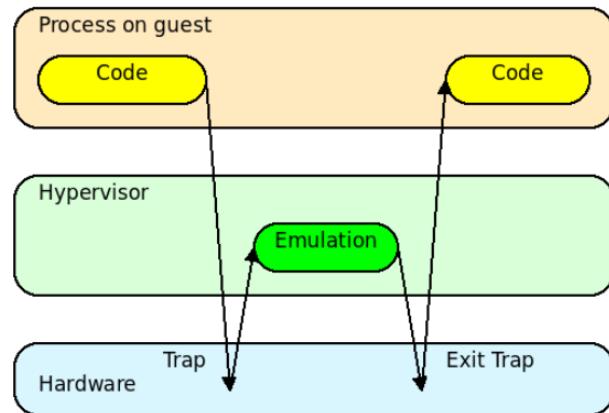
I concetti della virtualizzazione

I concetti succitati hanno in comune il fatto che determinati layer comprendenti determinate risorse vengono **astratti** (virtualizzati) e questi layer offriranno un'interfaccia sulla quale poi i guest andranno a funzionare, usando la stessa interfaccia astratta (**solo** quella): hiding e encapsulation delle risorse virtualizzate (Paper & Goldberg paper)

Trap and execute

Un modello un po' arcaico di virtualizzazione è quello **trap and execute**:

- Code runs on guest
 - Until a **sensitive** instruction is reached
- Sensitive instructions generate **traps**
- Traps are redirected to hypervisor
 - Hypervisor emulates correct behavior
 - Returns correct values
- Execution of guest resumes



Le funzioni “sensibili” sono quelle che modificano lo stato della macchina (accesso a un’istruzione privilegiata) vengono eseguite dall’hypervisor e non dalla macchina stessa. Quando vi è un’istruzione potenzialmente non safe in un contesto virtualizzato, l’hypervisor sospende l’esecuzione della macchina virtuale, lo esegue in modo safe, e restituisce il controllo alla macchina virtuale.

Un esempio di istruzione “trappable” è l’accesso alla RAM: poiché il codice gira sul guest, quando si fa una richiesta di accesso alla RAM, l’indirizzo generato è valido soltanto all’interno del guest! Pertanto, l’accesso genererebbe un’eccezione a livello di sistema operativo (poiché, alla fine, la RAM acceduta è sempre quella “vera”). Per

questo genere di operazioni, infatti, entra in gioco l'hypervisor, che traduce appositamente l'indirizzo generato dal guest, in uno valido per l'host!

Per questo genere di modello, serve **supporto hardware!** È necessario che l'hardware sia in grado di discernere fra indirizzi “virtualizzati” (non *virtuali*) e non. Inoltre c’è bisogno che l'hardware sia in grado di riconoscere gli eventi **trap** generati dalle istruzioni “sensibili”.

L'eventuale overhead di questo metodo, però, è abbastanza evidente, pertanto, in generale, l'interesse nella virtualizzazione (negli anni ‘80) si perde.

Inoltre l'hardware continua a diventare sempre meno costoso, pertanto al mercato conviene di più comprare nuovo hardware, piuttosto che virtualizzare (per via della lentezza dell'approccio alla virtualizzazione).

Anni ‘90 e la rinascita della virtualizzazione

In questo periodo iniziano a nascere veramente tantissimi sistemi operativi, pertanto ritorna l'interesse per la virtualizzazione, proprio per questo motivo. Un altro driving factor alla rinascita della virtualizzazione è stato proprio il **cloud computing**.

Il problema del trap and execute, però, è sempre presente, perché la virtualizzazione non è stata considerata per l'ultimo decennio.

Nascono nuovi sistemi di virtualizzazione, nascono estensioni per le architetture che supportano la virtualizzazione (per la gestione delle istruzioni **sensitive**).

Principi della virtualizzazione

Il paper di Popek & Goldberg **definisce una macchina virtuale come** il duplicato di una macchina fisica, ma isolata.

I tre requisiti principali per la virtualizzazione sono:

- **equivalence/fidelity:** il software eseguito sulla macchina virtuale è indistinguibile da quello che gira su una macchina fisica. Il software gira su una macchina virtuale, quindi, senza **modifiche**;

- The virtualized environment is “essentially identical” to the original one

By an “essentially identical” environment, the first characteristic, is meant the following. Any program run under the VMM should exhibit an effect identical with that demonstrated if the program had been run on the original machine directly, with the possible exception of differences caused by the availability of system resources and differences caused by timing dependencies.

- **resource control/safety:** una macchina virtuale accede alle risorse “reali” solo sotto controllo dell’hypervisor! L’hypervisor implementerà questo comportamento mediante l’uso di preemption delle risorse, evitando quindi di starvare la risorsa richiesta;

[resource control], labels as resources the usual items such as memory, peripherals, and the like, although not necessarily processor activity.

The VMM is said to have complete control of these resources if (1) it is not possible for a program running under it in the created environment to access any resource not explicitly allocated to it, and (2) it is possible under certain circumstances for the VMM to regain control of resources already allocated.

- **efficiency/performance:** è necessario che la macchina virtuale giri con delle prestazioni simili a quelle della macchina fisica. L’overhead è accettabile, ma in determinati intervalli. È ciò che separa in modo molto netto la **virtualizzazione** dall’**emulazione**. Per realizzare questa cosa, è necessario che le funzioni **sensitive** siano quanto meno possibili, per evitare il “salto” da guest ad hypervisor. Le architetture moderne fanno questo genere di ottimizzazione in modo massivo.

- Performance degradation must be low
 - Hardware execution is the most efficient method
 - Most instructions must be executed directly by hardware

It demands that a statistically dominant subset of the virtual processor's instructions be executed directly by the real processor, with no software intervention by the VMM. This statement rules out traditional emulators and complete software interpreters (simulators) from the virtual machine umbrella.

Vi sono quindi diverse categorie di istruzioni:

- safe: eseguibili tranquillamente dai guest senza alcun “salto”;
- privileged: eseguibili **solo** dall'hypervisor. Generano quindi una **trap** se non eseguite in modalità hyperV;
- sensitive: eseguibili **con l'intervento** dell'hypervisor. Sono tutte quelle istruzioni che cambiano il proprio comportamento in base allo stato corrente del sistema (memory management, accesso alla memoria dipende da che tipo di paginazione...). Sono anche quelle istruzioni che modificano la configurazione appena citata! È quindi necessario che **tutte le istruzioni sensitive siano anche privileged!**

N.B: vi è quindi un nuovo livello di privilegio (user, kernel, **hypervisor**).

Secondo P&G, la virtualizzazione funziona senza problemi quando tutte le istruzioni **sensitive** sono anche **privileged**.

È importante che questo accada, perché qualora esistessero funzioni sensitive che **non sono privileged**, allora tali funzioni non solleverebbero **trap**, causando evidenti problemi di sicurezza! In altre parole, voglio che tutte le funzioni sensitive **sollevino trap**, perché devono essere gestite dall'hypervisor, per far avvenire la virtualizzazione in modo corretto!

The terms are usually used in the context of *hardware virtualization*: virtual machines. Sensitive instructions are those that the *hypervisor* or *virtual machine monitor* (VMM) wants to trap and emulate to give an unmodified OS the illusion it owns its hardware resources, i.e. to successfully virtualize and run an OS.

Meanwhile, privileged instructions just refers to the set of instructions that your ISA defines as privileged. That is, these instructions must be executed by a process running in ring 0. (Notice

this notion has nothing to do with userspace or kernel mode *per se*, instead it has to do with the ring level your process is running in. It so happens that almost all the time, we run userspace processes in 3 and the kernel in ring 0).

Ideally, we want the set of sensitive instructions to equal that of instructions, **this allows us to trap and emulate using the hardware**. That used to not be the case though, so hardware e.g Intel VT-x were created to address this problem. Almost modern CPUs have support for hardware virtualization, partially by the VMM to trap and emulate all sensitive instructions.

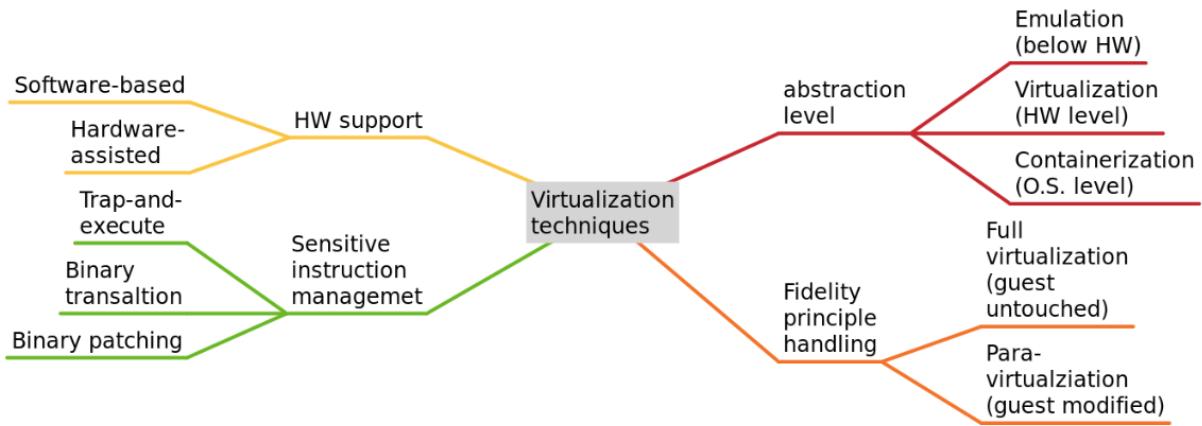
Until recently, the x86 architecture has not permitted classical trap-and-emulate virtualization. Virtual Machine Monitors for x86, such as VMware ® Workstation and Virtual PC, have instead used binary translation of the guest kernel code. However, both Intel and AMD have now introduced architectural extensions to support classical virtualization. We compare an existing software VMM with a new VMM designed for the emerging hardware support. Surprisingly, the hardware VMM often suffers lower performance than the pure software VMM. To determine why, we study architecture-level events such as page table updates, context switches and I/O, and find their costs vastly different among native, software VMM and hardware VMM execution. We find that the hardware support fails to provide an unambiguous performance advantage for two primary reasons: first, it offers no support for MMU virtualization; second, it fails to co-exist with existing software techniques for MMU virtualization. We look ahead to emerging techniques for addressing this MMU virtualization problem in the context of hardware-assisted virtualization.

Approcci alla virtualizzazione

In questo paragrafo approfondiamo i diversi approcci alla virtualizzazione:

- basati sul concetto di fedeltà di P&G: paravirtualizzazione (da vedere dopo) o virtualizzazione completa;
- basati sul bisogno di supporto hardware: software virtualization o hardware-assisted virt;

- basati sul bisogno di gestire le istruzioni sensitive: trap and execute, binary translation e binary patching ⇒ tutti questi modi realizzano la **virtualizzazione completa**.



Trap and execute

Abbiamo visto come questo approccio sia il più vecchio e quello con più overhead. Inoltre necessita di supporto hardware. Inoltre non si possono annidare più macchine virtuali, perché un solo hypervisor può “occupare” l’hardware.

Binary translation

Le istruzioni sensitive, tipicamente, sono nel **kernel**. Le applicazioni userspace fanno richieste al kernel per accedere alle risorse hardware e non hanno codice critico al loro interno. Posso quindi spostare la virtualizzazione **solo sul kernel**. Virtualizzando **solo il kernel**, il principio di efficienza è molto più semplice da soddisfare.

Nel codice dei binari, quindi, si cercano di identificare i frammenti in cui si fa l’accesso al kernel per determinate operazioni per tradurli se vanno ad effettuare accesso a componenti del kernel che vanno virtualizzati (perché magari fanno accesso alla RAM o altro).

In questo modo, quindi, solo quando queste istruzioni critiche (accesso al kernel) vengono eseguite, l’hypervisor si “sveglia” e va a tradurre/virtualizzare le richieste fatte dall’applicazione.

La binary translation fa uso del cosiddetto **basic block**, ovvero un’unità di codice che **dovrà essere tradotto**. Tramite questa tecnica vengono analizzati tutti i basic block che

vengono tradotti a tempo di compilazione. Una volta fatto ciò, poi, i blocchi tradotti verranno cacheati per eventuali riutilizzi.

- Initially introduced as a fast emulation technique
- Similar to JVM techniques
 - Just in Time compiling
 - Ahead of Time compiling
- Now used also in virtualization
- Focus:
 - Translation of sensitive but non privileged instructions
 - Can operate at runtime
 - Can cache translated code

Pertanto cosa succede: il kernel viene compilato sostituendo tutti i basic block con codice “safe” da un punto di vista della virtualizzazione. Pertanto, tutte le applicazioni che gireranno sulla macchina virtuale accederanno ad un kernel **safe**, le quali istruzioni sensitive verranno gestite dall’hypervisor!

- Two tasks
- **Scanning code** searching for sensitive instructions
 - Feasible because only a fraction of code is relevant
 - Focus only on kernel-level code
 - User space code is safe
- **Modification of code** to rewrite sensitive instructions with safe code
 - Make guest hypervisor-safe before execution

The biggest single difference between emulation and virtualization is that with virtualization, the guest operating system and applications are run natively; they can directly use the system processor without having to go through the same kind of translation layer as is used in emulated environments. Though most instructions that a processor performs—arithmetic, numerical comparisons, moving memory around—are safe to use in this way, most processors also have a small set of special instructions that require more care. For example, most processors have instructions for configuring processor modes, for setting up virtual memory, and so on.

Instruction sets that are designed from the ground up to support virtualization allow these special, sensitive instructions to be "trapped." Whenever the guest operating system tries to perform one of these privileged operations, the processor will "trap" the instruction and hand over control to the host operating system or hypervisor, so that it can do the required operation and then return control back to the guest. However, most real-world instruction sets, including x86, were not designed with virtualization in mind. As a result, there are privileged instructions that do not have any corresponding trap facility.

Binary translation addresses this problem directly. Instead of depending on the processor itself to detect the privileged instructions and pass control over to the hypervisor or host, virtualization software inspects the instruction stream in software. Whenever the virtualization software detects a problem instruction, it rewrites it on-the-fly, typically replacing it with a kind of manual trap, that will hand over control to the hypervisor at the appropriate moment. Essentially, this allows the hypervisor to emulate a small set of important instructions, while leaving the majority untouched.

In the early days of x86 virtualization, binary translation was the only option, as a result of x86 including untrappable privileged instructions. With the introduction of Intel's VT-x and AMD's AMD-V, the need for binary translation was greatly reduced; these extensions allow the processor to know that it's executing a guest operating system, and it can properly restrict the privileged instructions accordingly. Binary translation can still be desirable as a possible performance optimization, but it's no longer necessary to ensure correct functionality.

- Example code translation
 - From “A Comparison of Software and Hardware Techniques for x86 Virtualization”
 - Keith Adams and Ole Agesen
 - ASPLOS 2006
- Simple C code

```
int isPrime(int a) {
    for (int i = 2; i < a; i++) {
        if (a % i == 0) return 0;
    }
    return 1;
}
```

-
- ASM code from compiler (plus annotation)
 - Basic blocs are marked in colors

```
isPrime:    mov    %ecx, %edi ; %ecx = %edi (a)
            mov    %esi, $2   ; i = 2
            cmp    %esi, %ecx ; is i >= a?
            jge   prime   ; jump if yes
nexti:     mov    %eax, %ecx ; set %eax = a
            cdq
            idiv  %esi   ; a % i
            test  %edx, %edx ; is remainder zero?
            jz    notPrime ; jump if yes
            inc   %esi   ; i++
            cmp    %esi, %ecx ; is i >= a?
            jl    nexti   ; jump if no
prime:     mov    %eax, $1   ; return value in %eax
            ret
notPrime:  xor   %eax, %eax ; %eax = 0
            ret
```

I basic block vengono poi classificati secondo varie categorie:

- Rules for translation
- **IDENT**
 - Code can be translated identically
- **Direct Control Flow**
 - Mapping of addresses into translated code for jmp, call, ret instructions
- **Indirect Control flow**
 - When address is not available at translation time
 - Need to compute addresses on the fly
- **Privileged Instructions**
 - Need to rewrite them accordingly

La binary translation permette di ridurre particolarmente l'overhead **durante l'esecuzione**, perché la "fatica" viene fatta prima dell'esecuzione del programma (in molti casi), a compilazione, riducendo il tempo di overhead. ⇒ Abilita la **nested virtualization!**

VMWare è stato il primo esempio di questo genere di approccio.

VMware translates the binary code that the kernel of a guest OS wants to execute on the fly and stores the adapted x86 code in a Translator Cache (TC). User applications will not be touched by VMware's Binary Translator (BT) as it knows/assumes that user code is safe. User mode applications are executed directly as if they were running natively.

Il **binary patching** è una variante della binary translation che **rimpiazza il codice originale** (la binary translation tiene entrambe le versioni). Viene usata per ridurre il memory footprint della macchina virtuale quando possibile.

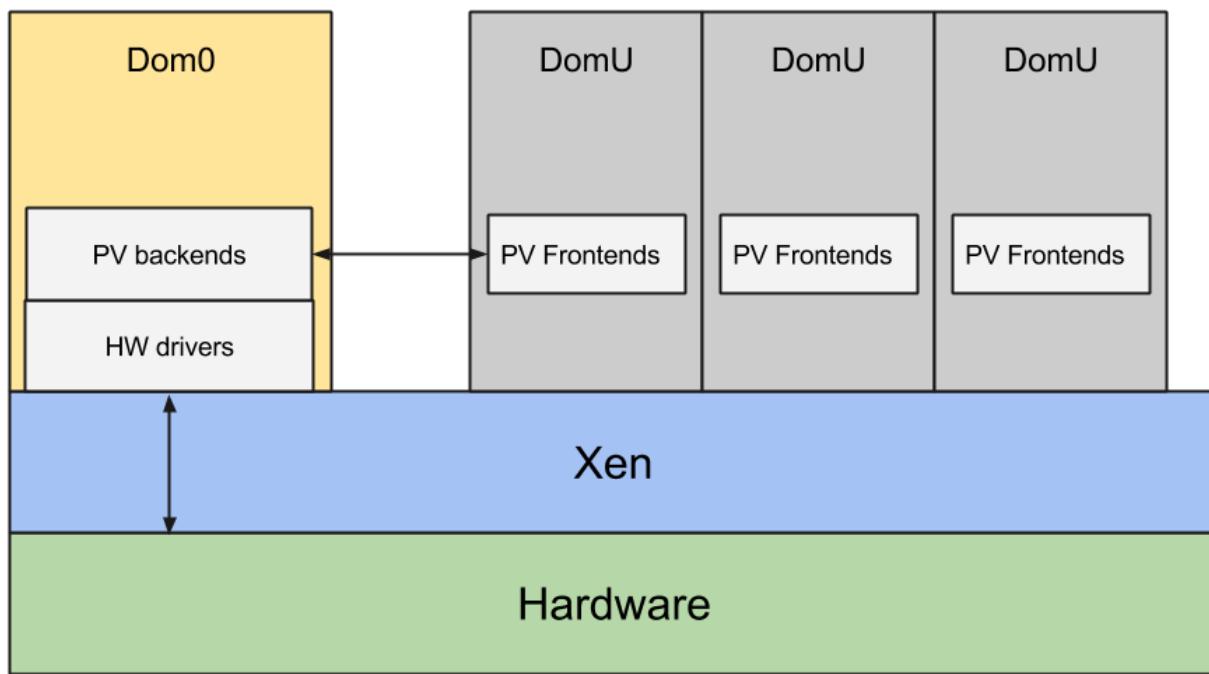
Paravirtualizzazione

Con la binary translation, le istruzioni vengono tradotte **quando vengono eseguite**. Spostandoci all'estremo di questa filosofia, si può andare a creare un sistema operativo della macchina **guest** (che prima veniva eseguito a ring-0) che, quando deve fare

richieste “pericolose” (i.e. sensitive) farà delle cosiddette *hypercall*, richiamando l’hypervisor. In questo modo, il kernel del sistema operativo che viene virtualizzato non è più al più basso livello possibile, pertanto non vi è più bisogno di supporto hardware!

- Offloading virtualization requirements on **guest**
- Guest must be safe
 - Code on guest MUST not execute sensitive instructions
 - Access to APIs to interact with the hypervisor
 - Hypercalls for:
 - Interrupts
 - Memory paging
- Like binary translation not at runtime
 - Translation at compile time

In altre parole, prendendo come esempio Xen, tutte le macchine faranno riferimento ad una macchina “madre”, **dom0**, che si occuperà di gestire le richieste di tutti i guest (**domN**):



Ovviamente, questo approccio alla paravirtualizzazione necessita di supporto da parte dei sistemi operativi. Sempre facendo riferimento a Xen, tutte le distro linux vanno benissimo, perché dom0 è basata su linux in ogni caso, mentre altri casi closed-source necessitano supporto da parte del vendor.

Con la virtualizzazione, la traduzione delle operazioni sul filesystem può essere particolarmente complessa, poiché è necessario mappare le operazioni del filesystem stesso su un file (quello che rappresenta la memoria di massa della macchina virtuale) che ha una geometria completamente diversa! Stessa cosa per altre periferiche di I/O, come schede di rete (che devono essere virtualizzate e poi mappate alle operazioni "vere" sulla scheda di rete "vera") o GPU! ⇒ Emulare le periferiche ha tanto overhead!

Tramite la paravirtualizzazione è possibile caricare un device driver che è consci di essere all'interno di un ambiente virtualizzato. In questo modo è possibile creare una "periferica finta" che non fa nient'altro che scrivere su un **buffer di memoria** che viene letto da dom0 (tramite hypercall) che comunicherà questa cosa alla **periferica "vera"**!

Inoltre, la gestione della memoria viene fatta in maniera più ottimizzata: dom0 (o equivalenti) genera un processo che cerca di occupare quanta più RAM libera possibile per renderla disponibile all'hypervisor per i propri guest! In questo modo non c'è bisogno di mappare la RAM "vera" con quella del guest stesso (come avviene in ambienti virtualizzati "canonici").

Gestione dell'hardware

N.B: questi concetti sono spesso implementati anche in hypervisor che “canonicamente sono virtualizzatori completi” (e non paravirtualizzatori come Xen), infatti anche **kvm** o **hyper-v** implementano queste cose.

- emulazione: come nella virtualizzazione “canonica”, le chiamate vengono emulate. Massimizza la **sicurezza**;
- accesso diretto all'hardware: quando determinati dispositivi (GPU) lo supportano, è possibile mapparli 1-1 sulla macchina virtuale. In questo modo il guest può accedere direttamente al dispositivo senza passare dall'hypervisor. Chiaramente il limite è che questa periferica non è virtualizzata, esponendola ad eventuali rischi. Massimizza le **performance**;
- split driver: il device driver viene diviso in due parti, per cercare un common ground fra i due approcci succitati.

Split driver

Il driver viene diviso in due parti: backend e frontend. Il primo gira nel dom0, ed è in grado di accedere all'hardware del dispositivo, mentre l'altro non è altro che un buffer che viene copiato sul backend dall'hypervisor.

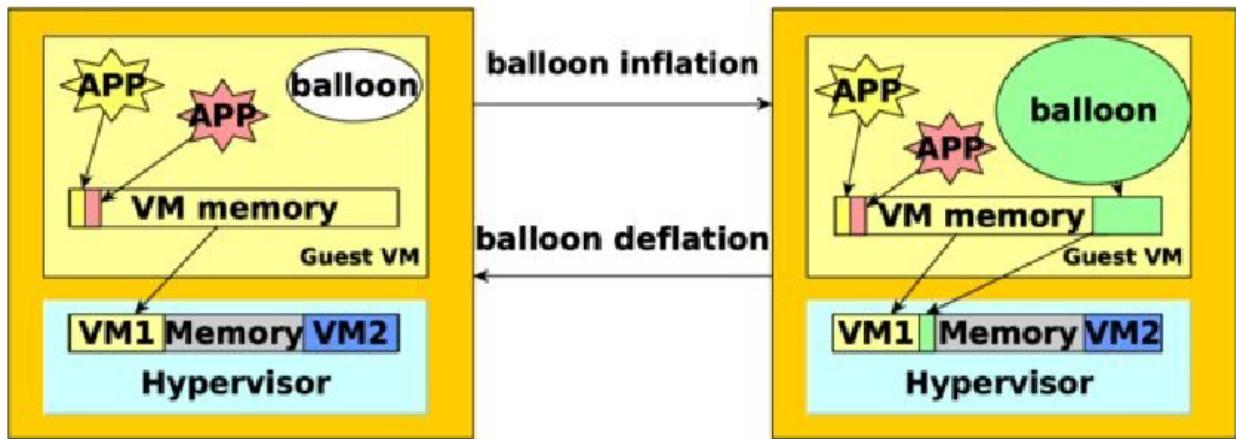
Quando il kernel avrà bisogno di mandare un frame al dispositivo, farà una hypercall all'hypervisor che si occuperà di incaricare il backend per fare effettivamente l'accesso al dispositivo. Poiché l'hypervisor è in mezzo a questa comunicazione, la coesistenza di più macchine virtuali che accedono alla stessa risorsa (GPU) è possibile (a differenza dell'accesso diretto)!

Memory ballooning

Questo metodo viene utilizzato per gestire la RAM in maniera efficiente. Spesso, le macchine virtuali non utilizzano tutta la RAM a loro assegnata, pertanto questo processo “ballooner” occupa le pagine di memoria correntemente non utilizzate.

A questo punto, quando una VM richiederà più RAM, l'hypervisor potrà interrogare il balooner per richiedere la memoria da assegnare alla VM. In questo modo, inoltre, l'hypervisor potrà essere meno complesso a livello di codice, poiché queste operazioni vengono delegate a questo processo.

Anche una live migration di una macchina virtuale potrà essere molto più efficiente da implementare, poiché le pagine non utilizzate dalla macchina virtuale sono ancora assegnate al balooner e non alla macchina virtuale, risparmiando notevolmente tempo!



Hardware extensions

Per soddisfare i requisiti di P&G, le architetture vengono estese per avere un ulteriore livello (quello hypervisor) di operatività. Ci sono quindi istruzioni in più che gli hypervisor possono utilizzare per funzionare. Queste funzioni non sono solo di interesse della CPU, ma anche per varie periferiche I/O, oltre che alla RAM!

Per quanto riguarda Intel, sono disponibili tre estensioni: CPU, IOMMU (I/O e RAM) e network.

Vengono aggiunte quindi 10 nuove istruzioni adibite a questo; inoltre, vengono aggiunte altre

due modalità di operazione (all'interno della modalità hypervisor), VMX root e VMX non-root, usato dalle macchine **guest**.

In realtà quello che avviene è che il numero di modalità raddoppiano: vi sono 4 rings per VMX non-root e altrettanti per VMX root, per emulare “completamente” tutto lo stack. Questo significa che, ad esempio, il kernel di una macchina guest girerà al **ring 0 VMX non-root**, perché è una macchina guest, mentre il kernel dell'hypervisor girerà a ring 0 VMX root. Per switchare modalità, ovviamente, vi saranno determinate traps.

A livello hardware, vengono create delle strutture dati (VM control structures) che permettono di gestire le macchine virtuali un po' come dei processi (praticamente i PCB però con le informazioni delle macchine virtuali).

Le istruzioni che vengono aggiunte da queste estensioni permettono, quindi, di interagire con queste strutture dati che permettono di gestire le macchine virtuali a livello hardware.

- VMLAUNCH
- VMRESUME —> queste due fanno entrare in VM non-root mode
- VMEXIT —> torna in VM-root mode
- VMCALL (hypervisor call)
- **VMCS Virtual Machine Control Structure**
 - VMPTRLD → load pointer from VMCS
 - VMPTRST → store pointer in VMCS
 - VMREAD → read field form VMCS
 - VMWRITE → write field in VMCS
 - VMCLEAR → clears VMCS
- Management of extension
 - VMXON → enables ring -1
 - VMXOFF → disables ring -1

Network and I/O extensions

In questo caso, il problema è quello di andare ad accelerare le operazioni di I/O con supporto hardware. Ad esempio, avendo una scheda di rete prestante, il sistema operativo ha accesso **diretto** alla periferica (DMA!). È possibile fare la stessa cosa in un contesto virtualizzato? Sì, infatti è possibile rimappare la gestione della memoria da parte dell'hypervisor in modo tale che le macchine guest possano utilizzare il supporto hardware per questo genere di cose.

Additional extension for networking (Intel only)

- VM Device queues (**VMDQ**)
- Management of **per-VM transmission buffers**

Memory virtualization

In contesto virtualizzato, il modello paginato inizia ad avere dei problemi, serve un livello di indirezione ulteriore.

Idealmente, nella gestione della memoria vi saranno gli identificatori di pagina che, assieme ad un offset, mi daranno la cella di memoria giusta. In un contesto virtualizzato occorre una page table su più livelli: uno per la macchina virtuale, che sta sull'host, e uno per i

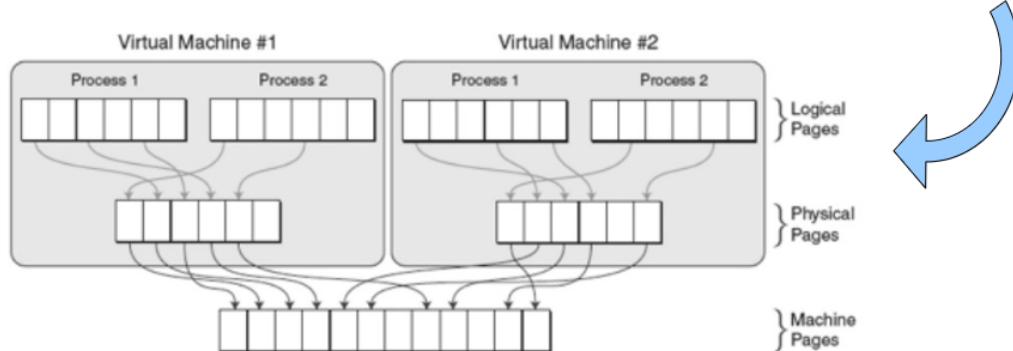
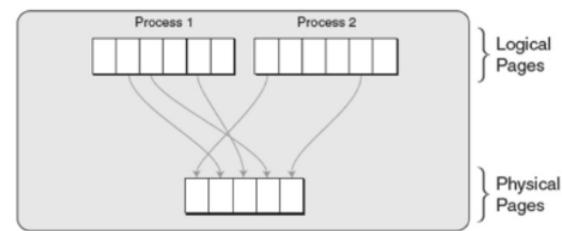
processi all'interno della macchina virtuale. Vengono dunque sviluppati, dai vendor delle CPU, supporti hardware per il TLB e per la MMU.

Virtualization adds another **level of indirectness**

Physical pages are **per-guest**

- Set of physical pages for each guest

Physical pages are mapped on host memory



Bisogna che quindi questo doppio livello abbia delle eventuali protezioni. Il kernel di una macchina virtuale deve essere capace di leggere **solo fino alle pagine logiche "interne" alla macchina stessa** ⇒ **Shadow pages**.

Il problema delle shadow pages è che, in questo modo, per accedere effettivamente alla RAM bisogna attraversare

due livelli di indirizzamento. Il guest non sa che gli indirizzi che sta accedendo non esistono, perché quando vengono acceduti subentra l'hypervisor, che si preoccuperà di fare la traduzione di quell'indirizzo ad un indirizzo virtuale "vero". Tale indirizzo virtuale "vero" dovrà poi essere tradotto **di nuovo** per accedere direttamente alla RAM!

Per realizzare questo meccanismo, VMX/AMD-V fa uso del registro

CR3, che non è altro che un puntatore al base page register per la guest correntemente in esecuzione.

Il problema è che ogni volta che si va a fare uno switch fra due macchine virtuale, l'overhead causato da queste operazioni è molto alto, pertanto ci sono altri approcci.

Xen, ad esempio, introduce una visione diversa della memoria virtuale. Infatti, viene generata un'unica struttura di pagine che viene condivisa fra più macchine. Quando si accede alla page table, non lo si fa tramite un kernel "originale", ma tramite un kernel "paravirtualizzato", che sa di essere in una macchina virtuale. La struttura dati è **unica**, quindi, e ogni para-vm sa quali sono le pagine che possiede! La differenza fra i due meccanismi è che il primo viene usato per fare in modo che **nulla** cambi all'interno del guest e che la VM esegua senza nessuna modifica, come se fosse una macchina "vera" (principio di fedeltà), mentre la paravirtualizzazione, come già detto, rilassa questo vincolo, a scapito di una modifica del kernel.

L'approccio moderno vede l'utilizzo di **nested page tables**. In particolare, le pagine TLB vengono **annidate** (in hardware!) e, pertanto, il lookup (da guest a RAM "vera") è **molto più veloce** (Intel EPT o AMD RVI).

L'hypervisor potrà mettere in atto anche politiche di **page sharing**, pertanto macchine che hanno pagine di memoria **uguali** possono **condividerle!**

Shadow page tables are used by the hypervisor to keep track of the state in which the guest "thinks" its page tables should be. The guest can't be allowed access to the hardware page tables because then it would essentially have control of the machine. So, the hypervisor keeps the "real" mappings (guest virtual -> host physical) in the hardware

when the relevant guest is executing, and keeps a representation of the page tables that the guest thinks it's using "in the shadows," or at least that's how I like to think about it.

Notice that this avoids the GVA->GPA translation step.

As far as page faults go, nothing changes from the *hardware's* point of view (remember, the hypervisor makes it so the page tables used by the hardware contain GVA->HPA mappings), a page fault will simply generate an exception and redirect to the appropriate exception handler. However, when a page fault occurs while a VM is running, this exception can be "forwarded" to the hypervisor, which can then handle it appropriately.

The hypervisor must build up these shadow page tables as it sees page faults generated by the guest. When the guest writes a mapping into one of its page tables, the hypervisor won't know right away, so the shadow page tables won't instantly "be in sync" with what the guest intends.

So the hypervisor will build up the shadow page tables in, e.g., the following way:

- Guest writes a mapping for VA `0xdeadbeef` into its page tables (a location in memory), but remember, this mapping isn't being used by the hardware.
- Guest accesses `0xdeadbeef`, which causes a page fault because the real page tables haven't been updated to add the mapping
- Page fault is forwarded to hypervisor
- Hypervisor looks at guest page tables and notices they're different from shadow page tables, says "hey, I haven't created a real mapping for `0xdeadbeef` yet"
- So it updates its shadow page tables and creates a corresponding `0xdeadbeef` >HPA mapping for the hardware to use.

The previous case is called a *shadow page fault* because it is caused solely by the introduction of memory virtualization. So the handling of the page fault will stop at the hypervisor and the guest OS will have no idea that it even occurred. Note that the guest can also generate genuine page faults because of mappings it hasn't tried to

create yet, and the hypervisor will forward these back up into the guest. Also realize that this entire process implies that *every*

page fault that occurs while the guest is executing must cause an exit to the VMM so the shadow page tables can be kept fresh. This is expensive, and one of the reasons why hardware support was introduced for memory virtualization.

Storage virtualization

Un grosso problema delle macchine virtuali è la **rappresentazione dei loro dischi**. Generalmente, lo spazio di indirizzamento di un disco è un **blocco di dati** identificato da una terna di dati: settore, distanza della testina dal centro, piatto. La rappresentazione di un disco fisico deve essere **mappata** in un qualche modo su un **file!**

VMDK & QCOW

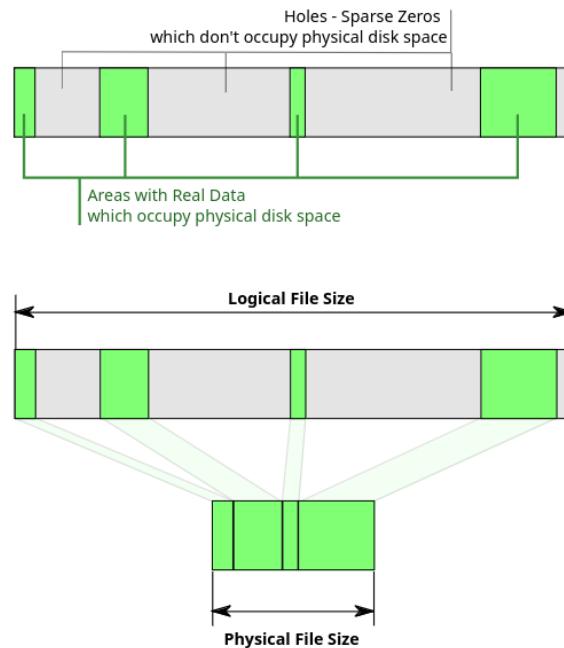
VMDK file

Description of physical disk image

- Disk geometry description
- Disk sectors

File structure

- Sparse file
- Flat file



È possibile utilizzare **sparse files**, che vengono usati per ottimizzare lo spazio su disco (ottimi su SSD, perché non usano la testina che deve andare in giro a fare casini) o **flat**

file, che permettono di memorizzare un file **contiguamente**, ma scrivere “dati veri” solo quando viene effettivamente memorizzato qualcosa.

I file VMDK permettono di creare degli **snapshot** dei filesystem: una volta che ho, ad esempio, installato il sistema operativo, è possibile fare uno snapshot e avere tale layer “congelato”.

A questo punto, viene creato un nuovo layer sul quale poi la VM andrà effettivamente a scrivere. Per realizzare questo meccanismo vengono usati i **delta links**, che conterranno le modifiche fatte sullo snapshot (base disk).



Questa cosa è straordinariamente potente! Tante copie della VM potranno prendere la stessa base image per avere, ad esempio, lo stesso sistema operativo, ma ciascuno di essi avrà i delta link con i cambiamenti fatti per ciascuna macchina virtuale!

QCOW è la versione “rivisitata” dei file VMDK:

Introduced ad support file for QEMU emulator

3 versions of file

Support for:

- Disk size growth → can implement features like sparse files even if host file system has no support for them
- Compression
- Encryption (since v2) → based on AES

Hypervisor resource management

Sebbene esistano molteplici meccanismi che permettono di astrarre la macchina host dalla macchina guest, è necessario che l'hypervisor sia in grado, anche, di gestire le risorse che lui stesso alloca sull'host.

CPU Capping

L'hypervisor è solito allocare **core virtuali**: un core virtuale corrisponde ad un core fisico, ma è ovvio che più macchine virtuali possono andare ad utilizzare più core di quelli che sono fisicamente disponibili, poiché generalmente una VM usa meno risorse di quelle ad essa assegnate.

In ogni caso, qualora una VM utilizzi troppa CPU, l'hypervisor potrà limitare la loro performance, specialmente se, da un punto di vista di cloud provider, queste non sono le più "redditizie" (le istanze più "cheap"). Questo meccanismo è chiamato CPU capping.

Le macchine che vengono **cappate** subiscono il cosiddetto "CPU steal", ovvero il cloud provider "ruba" tempo di CPU alla VM che sta andando oltre i limiti ad essa imposti. Questa metrica mi dice la percentuale di tempo in cui la macchina è fuori dai limiti di CPU imposti e che l'hypervisor sta rubando quel tempo di CPU (allocandolo ad altre macchine, volendo). Se `steal > 0`, allora l'hypervisor sta togliendo tempo di CPU per darlo ad altre macchine virtuali.

È probabile che macchine più grandi e performanti non siano afflitte (o lo siano meno) da questo problema.

Modelli di interferenza fra macchine virtuali

In certi casi è possibile che le macchine virtuali “interferiscano” tra di loro: l'esempio classico è l'azzeramento delle cache ogni volta che si fa un context switch fra macchine virtuali (cache interference). Questi comportamenti violano il principio di isolamento delle macchine virtuali.

L'altro problema di questi fenomeni è il fatto che possano essere usati come **side-channel** per attacchi informatici: due macchine virtuali che apparentemente non sono comunicanti tra di loro, possono in realtà comunicare facendo variare l'utilizzo di risorse. Questa variazione di utilizzo delle risorse può essere interpretata come “codice per comunicare” un qualcosa di malevolo o quant'altro (ricerca qualcosa, sembra interessante).

Side channels

Interference can be used as a **side channel for data leak**

VM1 wants to transfer data to VM2

- E.g., VM1 is compromised and is trying to download data without leaving traces

If VM2 is co-located (i.e., on same host) with VM1

- VM1 can start bursts of operations to slowdown host
- VM2 can monitor its performance to detect slowdowns

Interference is used as a **low-bandwidth** but **undetectable** data channel

Markov model

Il modello aleatorio utilizzato per modellare questo genere di problemi è lo stochastic markov model: con una data probabilità è possibile entrare in uno dei due “stati” del modello - fast o slow.

In ciascuno dei due stati, il sistema avrà una diversa capacità di processing: se si trova nello stato slow, sarà **mu_slow**, altrimenti **mu_fast**.

Migrazione di macchine virtuali

Può succedere che sia necessario passare una macchina da un nodo all'altro. Gli scenari possibili sono:

- migrazione “offline”: stop su nodo 1 & restart su nodo 2;
- live migration: migrazione senza downtime da nodo 1 a nodo 2.

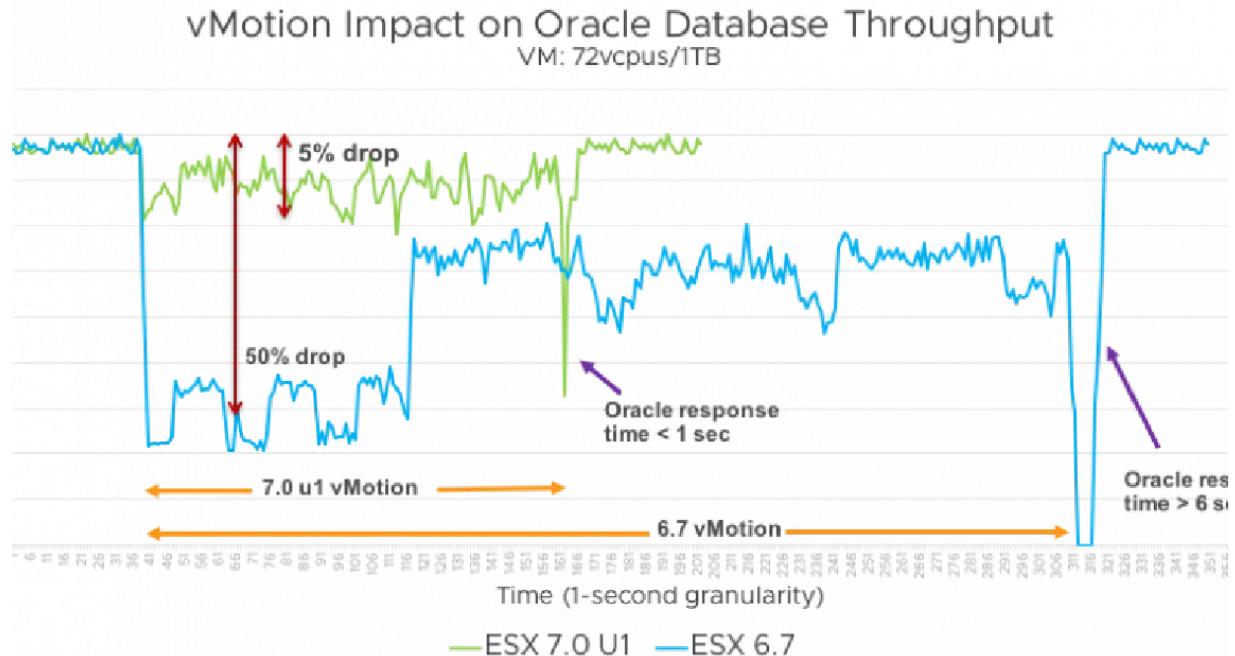
La migrazione offline è il tipo di migrazione più facile da capire e implementare per un hypervisor, ma non necessariamente la cosa più ideale in scenari di high availability, poiché il downtime di una macchina virtuale è comparabile all'interno downtime annuale (in 99.999% SLA).

Live migration

La live migration è più complessa e viene generalmente implementata tramite il processo di **pre-copy**, che ha quattro fasi: preparation, memory copy, migration, resume;

- preparation: si fa uno snapshot del disco e si fanno diversi layer per facilitarne il trasferimento;
- memory copy: dato il principio di località, una macchina virtuale avrà una lista di “hot pages” che starà accedendo. Generalmente, l'hypervisor durante questo step ha già creato la macchina di “destinazione” e, ora, starà copiando le pagine da RAM a RAM, partendo da quelle meno “hot” (perché, ovviamente, sono quelle che saranno sporcate più facilmente, quindi avrà meno senso copiarle). La cosa interessante è che tutte queste operazioni sono fatte con la macchina di origine **accesa**, pertanto è possibile che le pagine in copia vengano “sporcate”, pertanto sono necessari più round di copia. Oltre allo spostamento delle pagine, viene spostato anche il contenuto del disco (eventualmente solo determinati layer, se l'immagine di base è condivisa!);
- migration: la VM di origine viene effettivamente fermata e vengono copiate anche le pagine “hot” per rigenerare interamente la macchina a destinazione. Viene trasferito anche lo stato dei registri della macchina di origine e, infine, la macchina di origine viene fermata, mentre quella di destinazione viene accesa. In questo ultimo step, la disruption sarà nell'ordine dei **millisecondi**.

Il processo di live migration, però, è tutt'altro che leggero, infatti ha un evidente performance toll sul sistema.



Containerizzazione

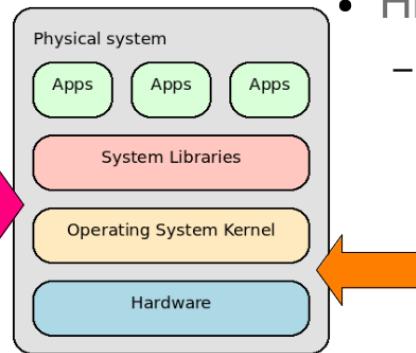
Dal punto di vista software, un container è visibile come un insieme di dipendenze ed eseguibili che formano un'unità *self-contained* che può essere distribuita con estrema facilità. I container sono unità software che possono essere prese individualmente e che funzioneranno sempre allo stesso modo, perché vengono impacchettati a contenere tutte le dipendenze necessarie per funzionare.

Per fare un'analogia, i container possono essere visti come **software units**. Le software units (e.g pacchetti deb o rpm) sono un insieme di **meta-informationi** che vanno a comporre **l'installazione** di un determinato software. Contengono una lista di **dipendenze** da cui dipende il programma che si vuole installare e tutti i passi per installarle.

Il problema di queste software units (non dei container) è che sono molto complicate da gestire da un punto di vista di aggiornamenti.

Poiché i container sono autocontenuti, il problema riguardante la gestione delle dipendenze non si pone, perché sono unità che sono autocontenute e che contengono tutte le dipendenze necessarie, ma che allo stesso tempo sono **isolati da tutti gli altri container** (se non collegati appositamente). Avere diverse dipendenze in progetti, a questo punto, è molto più semplice e replicare ambienti diversi ambienti di sviluppo diventa molto più semplice. Un container, essenzialmente, è un binario staticamente linkato “on steroids”.

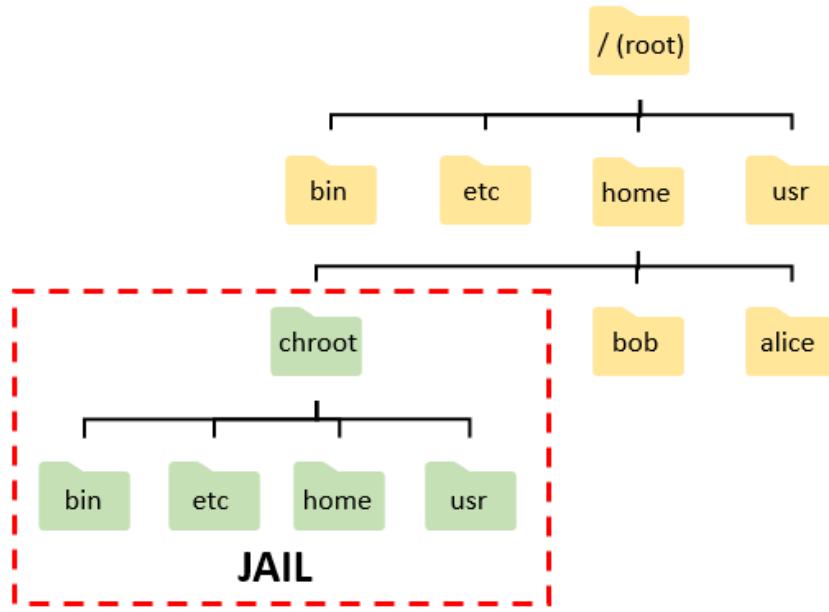
- Container
- Abstraction @ O.S. level
- Run on same O.S.
- Small memory footprint
- Low overhead
- Low startup time
 - `clone()` syscall
- VM
- Abstraction @ Hardware
- Can run different O.S.
- Large memory footprint
- High overhead
- High startup time
 - O.S. boot



Dal punto di vista prestazionale un container è poco più di un **processo**, a livello di “peso”, poiché è una semplice astrazione sopra il kernel dell’host.

Principi di containerizzazione

Negli anni ‘70 viene introdotta la system call `chroot`: tramite questa chiamata, il processo è in grado di avere una **root directory diversa da quella di sistema**. È il primo esempio di **software isolation**.



Successivamente, verrà introdotto il concetto di **jail** (anni 2000): in questo caso, le jail vengono isolate **a livello network**, oltre che a quello di root. Non solo ogni jail avrà un indirizzo diverso, ma, in generale, un intero stack di rete diverso, una diversa process list, un hostname diverso, un diverso set di utenti e di root users, insomma, è un'entità veramente isolata all'interno del sistema operativo.

Fast forward fino al giorno d'oggi: abbiamo Docker, che permette di gestire i container tramite immagini.

Lo sviluppo della containerizzazione va di pari passo con lo sviluppo della virtualizzazione, infatti i due fenomeni si muovono quasi parallelamente nel tempo, perché effettivamente i due approcci rispondono alla stessa domanda: ospitare più applicazioni sulla stessa macchina. Un altro driving factor dello sviluppo dei container è la nascita della filosofia **DevOps**.

Limiti di **chroot**

Le chroot-jail possono essere facilmente evase, però, perché è possibile usare i path relativi per muoversi “sopra” alla chroot jail. Inoltre, chroot permette soltanto di “isolare” un processo a livello di filesystem, non a livello di risorse e altro!

- Example:

```
#include <sys/stat.h>
#include <unistd.h>
int main(void)
{
    mkdir(".out", 0755);
    chroot(".out");
    chdir("../..../..");
    chroot(".");
    return execl("/bin/bash",
                "-i", NULL);
}
```

Una chroot-jail può essere facilmente evasa

Namespace e CGroups

I namespace Linux sono meccanismi che permettono di isolare “veramente” un processo su più livelli: network, fs, processlist,

Ogni namespace, infatti, fa riferimento a qualche area di interesse del sistema operativo ed è isolato rispetto a tutti gli altri: quando un processo si associa ad un namespace, vedrà **soltanto** (per davvero, stavolta) le risorse associate a quest’ultimo.

I **cgroup** d’altro canto, sono meccanismi del kernel che permettono di controllare e **limitare** le risorse per determinati processi (i.e. all’interno di interi namespace, infatti un cgroup può essere associato ad un namespace).

Benefici della containerizzazione

Portabilità

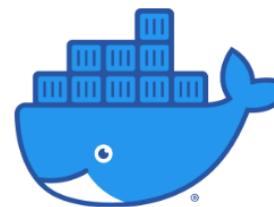
Un’immagine che genera un container non è altro che un set di istruzioni che è sostanzialmente replicabile su qualsiasi sistema in maniera sostanzialmente idempotente (works on my machine!).

Poiché i container sono standard, essi sono anche indipendenti dal backend che viene utilizzato per gestire la containerizzazione (CNI, CNM, ...).

- per i benefici ⇒ slide sufficienti

- Core functions of Containerd
 - **Running** containers (using runc)
 - **Management** of the container **images**
 - **Pushing/pulling** container images on **register**
- Containerd exposes its function using **API**
- Functions not supported by Containerd
 - Building images
 - Mounting volumes
 - Managing network (too complex with VNF/SDN!)
 - Orchestration

- Docker → container engine
- Built on top of Containerd
- Used to:
 - Build containers
 - Manage containers
 - Run containerized applications
- Docker provides the benefits of containers
 - DevOps-friendly development
 - Scaling

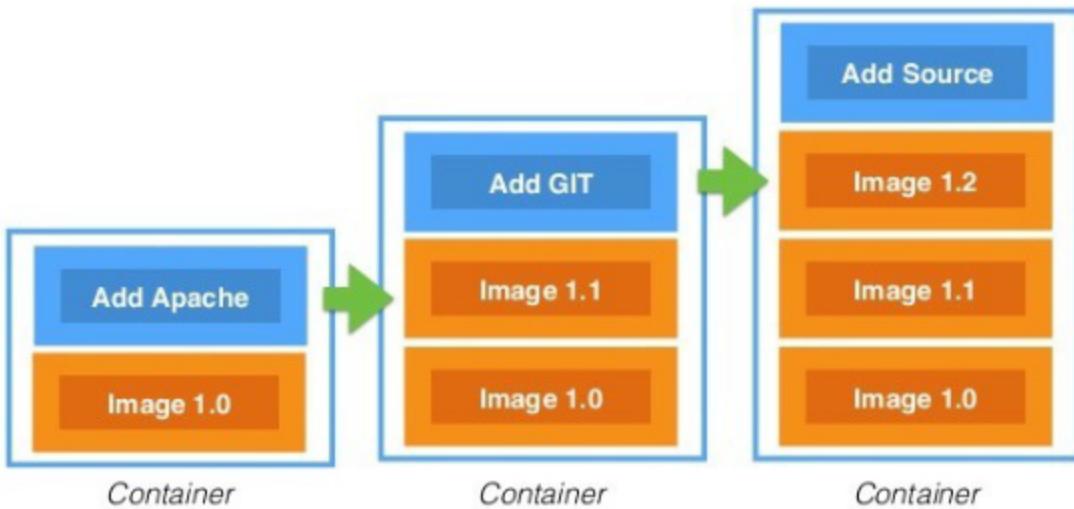


Filesystem dei container

I container lavorano con i cosiddetti *layered filesystem*, molto simile al meccanismo dei file usati per memorizzare i dischi delle macchine virtuali: la differenza principale di questi ultimi è che questo genere di filesystem (quelli per i container) ragionano a livello di **file**, non di blocchi.

I layer vengono impilati tra di loro e ciascuno di essi avrà una lista di file ad esso associati: funziona molto bene con i container, perché generalmente questi vengono costruiti

incrementalmente. Posso creare, dunque, a partire da un'immagine base (con i propri file), immagini custom, impilando i layer uno sopra gli altri.



Inoltre, i layer impilati a creare un'istanza di container eseguibile diventano **readonly**, solo l'ultimo layer diventa read-write.

Images → Read-only

Containers → Read-write

- Only **top layer** is writable
- Based on **Copy-on-Write** approach

Containers can **share** the same image

- Difference is the the top layer

Copy on Write increases **efficiency**

- Faster **startup** time
- Less **disk space** used

Copy-on-Write: i layers delle immagini sono condivisi finché non vengono modificati, nel senso che i layer vengono scaricati e riutilizzati per tutte le immagini che li usano.

Quando un container modifica un layer condiviso (i.e. durante la creazione dell'immagine tramite Dockerfile), i file modificati (e solo quelli!) vengono copiati nel layer read-write succitato.

When you start a container, a thin writable container layer is added on top of the other layers. Any changes the container makes to the filesystem are stored here. Any files the container does not change do not get copied to this writable layer. This means that the writable layer is as small as possible.

When an existing file in a container is modified, the storage driver performs a copy-on-write operation. The specifics steps involved depend on the specific storage driver. For the

`overlay2` driver, the copy-on-write operation follows this rough sequence:

- Search through the image layers for the file to update. The process starts at the newest layer and works down to the base layer one layer at a time. When results are found, they are added to a cache to speed future operations.
- Perform a `copy_up` operation on the first copy of the file that is found, to copy the file to the container's writable layer.
- Any modifications are made to this copy of the file, and the container cannot see the read-only copy of the file that exists in the lower layer.

Serverless

La filosofia serverless vede come componente principale, la possibilità di astrarre completamente la gestione e la manutenzione dei server, delegandola al provider, permettendo a chi sviluppa di dedicarsi completamente allo sviluppo software.

Il provisioning e il deployment di tutto lo stack necessario per eseguire l'applicazione è completamente *managed*: chi sviluppa dovrà preoccuparsi di fornire una **definizione** del software che sta scrivendo, che può essere un'immagine di un container o, a volte, anche meno.

⇒ PaaS, NoOps

Si riescono a tagliare i costi di gestione dell'infrastruttura, ma si **deve pagare il provider**.

- | | |
|---|---|
| <ul style="list-style-type: none">• Serverless• Scaling is automated• Monitoring and analysis managed by provider• Service shutdown when not used• Fine-grained cost model<ul style="list-style-type: none">- Cost on capacity consumed• Security and patching managed by provider<ul style="list-style-type: none">- User must concern only on updating its software | <ul style="list-style-type: none">• IaaS• Complete control of infrastructure• Scaling managed by user• Monitoring and analysis managed by user• Coarse-grained cost model<ul style="list-style-type: none">- Cost model based on capacity allocated• Security, patching, updates managed by user<ul style="list-style-type: none">- Including base software (O.S., servers, ...) |
|---|---|

La scalability, la availability, il monitoring, la sicurezza, il patching e tante altre cose, vengono passate in mano al **provider**, togliendo un sacco di preoccupazioni e di **costi** al team di sviluppo dell'applicazione.

Il costo è basato su quante invocazioni vengono fatte sulla funzione, quanto è la memoria media usata e quanto è il tempo di utilizzo (megasecondi, gigasecondi...).

- Minimize **latency**
 - Serverless functions typically do not operate from a single origin server
 - No single location to which end user's traffic has to be directed to
- Minimize **software intricacy**:
 - Functions must be simple
 - Segregation of codebase into small functions assists in fixing bugs and rolling out updates
- Enhancing the software **productivity**:
 - Separate applications developers tasks from infrastructure and operations administrators

Vi sono due categorie di servizi serverless:

- Backend as a Service: BaaS ⇒ Google Firestore!
- Function as a Service: FaaS ⇒ Google Functions

Function as a Service

Quando il workload è particolarmente elastico, avere una scalabilità più fine, che va oltre la VM, può essere benefico. Infatti, le driving technologies del FaaS sono le stesse che abilitano alla containerizzazione!

Con FaaS, l'esecuzione della singola funzione può essere fatta su un server qualsiasi presente nella *fleet* del provider, proprio perché le funzioni stesse devono essere disegnate per essere quanto più **stateless** possibili!

Le funzioni sono:

- stateless
- ephemeral: non si ha il controllo della loro esecuzione, né della loro **localizzazione**;
- event driven: si possono configurare in base a determinati trigger;
- fully managed

- per use cost model: RAM, ad esempio, viene misurata in Giga-secondi (quanti giga di RAM uso in un dato secondo);
- business logic focused

È importante, per la loro natura, rinforzare l'idea che una funzione FaaS sia **stateless** per evidenti motivi (non ci possono essere risorse condivise "localmente"). Questa caratteristica delle FaaS le rende **estremamente parallele**.

Oltretutto, poiché ephemeral, le FaaS non hanno alcun tipo di persistenza e possono essere **teared down** quando il traffico non è sufficientemente alto!

Use cases

UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA



- Whenever **automatic scaling** can make the difference
 - Large difference between **low** and **high** traffic periods
 - **Idle** periods
 - **Unknown** and highly **variable workload**
- Some examples
- **Data management**
 - Data **ingestion** functions (batch and on-the-fly)
 - **Report** production (batch processes)

Use cases

UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA



- **IoT** scenarios
 - Lots of devices (scalability issues)
 - Sensor data to manage
- Message-driven applications
 - **Bots** (typical example)
 - Real world application: **Slack**

Ovviamente non tutto è rosa e fiori: le FaaS hanno anche importanti drawback! Il controllo dell'esecuzione delle funzioni non è facile da controllare, stessa cosa per il **tempo dell'esecuzione della funzione stessa**, per non parlare del **debugging** di eventuali problemi sulla funzione. Altro problema importante è quello del vendor lock-in.

⇒ warm start e cold start

⇒ no latency critical!

⇒ no HPC! Vengono messi hard limit alla durata dei task e alle risorse che occupano

Da un punto di vista della sicurezza, FaaS è una lama a doppio taglio:

- da un punto di vista, se la funzione è sicura, il resto è sicuro, da un punto di vista di software, perché non viene gestito interamente dall'utente;
- per lo stesso motivo, il vendor potrebbe avere problemi di vulnerabilità, pertanto il problema si sposta semplicemente da un'altra parte;
- problema della monocoltura: poiché l'infrastruttura delle FaaS è comune a **tantissimi progetti** (anche diversi!), dovesse venire fuori una vulnerabilità di tale infrastruttura, tale vulnerabilità affliggerebbe tantissimi deployment cloud! Se tutti usano gli stessi software, tali software avranno molto più impatto se afflitti da vulnerabilità.

Tecnologie abilitanti

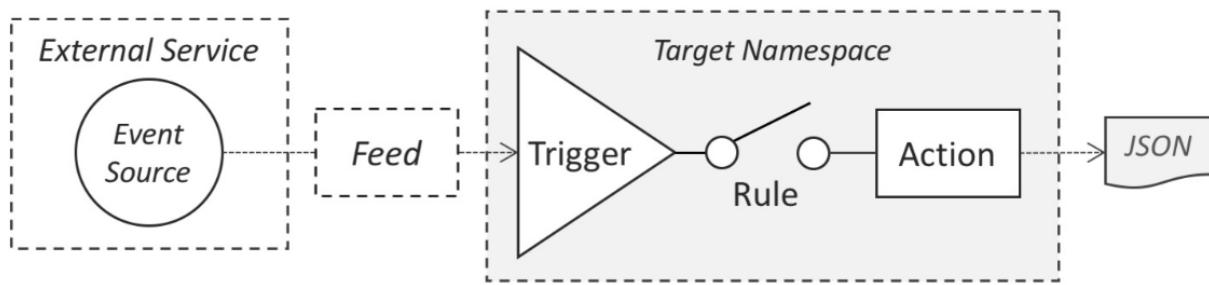
Esistono tantissimi framework che abilitano FaaS. In particolare ci focuseremo su **OpenWhisk** e **OpenFaas**.

OpenWhisk

OpenWhisk promuove principalmente l'evitare il vendor lock-in, infatti è interamente open source (Apache license).

Il programming model di OpenWhisk è chiaramente ben definito e documentato: viene definito un trigger al quale è associata una regola che **attiva** un'azione!

- Event-driven model
- Incoming data → Feed
- Match over condition → **Trigger and rule**
- Executing code → **Action**



Permette di monitorare tantissime sorgenti eventi out-of-the-box e permette anche l'estensione di questo comportamento in maniera da sviluppare plugin custom.

Action

Una action è definita o come uno spezzzone di codice (funzione vera e propria!) o come un container, nel caso voglia lavorare con linguaggi non supportati.

L'azione deve essere **idempotente**, quindi eseguibile in tante istanze in maniera estremamente concorrente e **senza garanzia di ordine** (exec order ≠ calling order). Inoltre le azioni non sono garantite **atomiche**.

Gli input delle azioni sono sostanzialmente dictionary JSON, mentre le azioni vengono definite alla “pacchetto Java”:

- Actions defined by:
 - **Namespace**
 - **Package name**
 - **Action name**
- Example: /**whisk.system**/**samples**/**greeting**

Le azioni possono essere:

- fire and forget: non bloccante, l'azione viene invocata in background e non ritorna niente;
- request-response: bloccante, come una chiamata http, ritorna effettivamente qualcosa.

Ogni azione ritorna un activation record con tutte le informazioni riguardanti all'esecuzione della funzione stessa.

- **Activation record**
 - For every action invocation
 - Critical for fire-and-forget requests
- Stored in OpenWhisk **logs**
- Contains
 - **Activation ID**
 - **Name** (with namespace) of the function
 - **Timestamps** (start and end)
 - **Logs**, annotations to track execution
 - **Response** (including **status**)

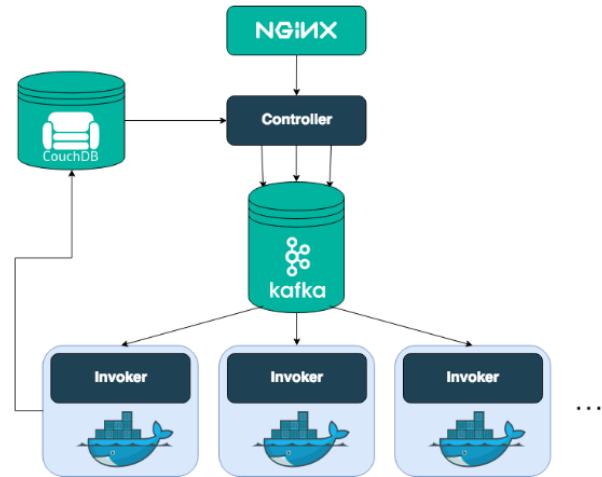
⇒ Particolari azioni sono le **web-based actions** ⇒ **HTTP interactions**

Ogni azione può essere limitata secondo:

- RAM usata
- CPU usata
- output size
- numero massimo di esecuzioni concorrenti
- rate limiting
- code size
- input size

Il modello architettonico è molto semplice:

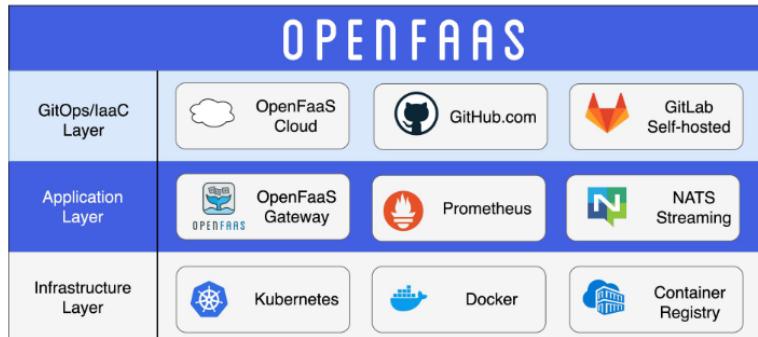
- Flow processing model
- NGINX
 - Manages incoming requests
- Kafka
 - Message queue
- Docker
 - Execution of functions
- CouchDB
 - Meta-data storage
 - Logs



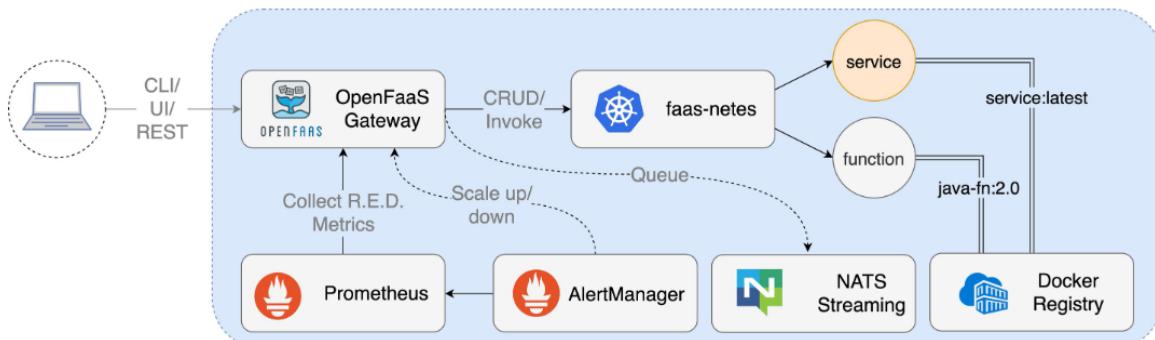
OpenFaaS

OpenFaaS ha un approccio più lowlevel, e prevede anche l'utilizzo di un **daemon**, `faasd`, che è molto più leggero di OpenWhisk.

- Typical technology stack
- GitOps/IaaS Layer
 - Infrastructure as Code
 - High level representation
- Application layer
 - Prometheus → monitoring
 - NATS → message queue
- Infrastructure layer
 - Container management



⇒ NATS: coda di eventi



AWS Lambda e Google functions ⇒ slide

Virtualized storage

Nel mondo cloud-based, anche lo storage viene virtualizzato, ovviamente, viene quindi astratto per abilitare questo genere di operazioni al mondo cloudnative.

Come faccio a memorizzare e processare exabyte di storage? Non esiste **un hard disk da 1PB!** Anche lo storage va reso **available, load balanced e fault tolerant.**

Si possono categorizzare i metodi di virtualizzazione dello storage seguendo due principali caratteristiche:

- granularità della virtualizzazione
- scope della virtualizzazione

Granularità della virtualizzazione

Il tipo di granularità dipende dall'altezza alla quale avviene la virtualizzazione.

Il mapping può essere fatto

sotto il filesystem (virt. a blocchi), col quale viene offerto un dispositivo a blocchi virtuale che si occupa della loro gestione, oppure **sopra il filesystem**, dove viene semplicemente astratta la “*locazione fisica*” del file (e.g. /home/macca/test.txt accessibile via rete, la location viene astratta completamente, monto il filesystem virtuale come cartella nella mia home e bona).

Block-based

L'idea generale è quella di esporre i dischi disponibili (non residenti tutti sulla stessa macchina!) come aggregato di blocchi di disco logici. Su questi blocchi viene costruita la struttura del filesystem! Il filesystem, quindi, può **spennare più hard disk fisici**, pertanto un blocco potrà trovarsi su una macchina, mentre un altro su un'altra.
⇒ SCSI over TCP, SCSI over FC, InfiniBand (DMA remoto), ...

Il filesystem, ovviamente, dovrà essere in grado di gestire operazioni con un filesystem così distribuito!

Identificazione di un blocco

Un blocco viene identificato con due numeri: LUN e LBA. Il primo è il Logical Unit Number (nome del dispositivo, come `sda-b-c-..`). All'interno del disco vi è il Logical

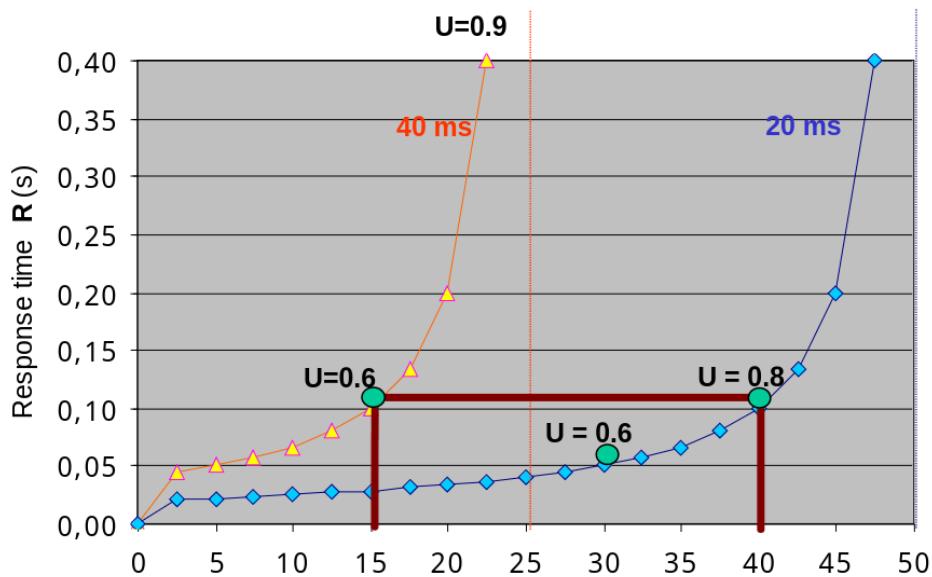
Block Address (LBA), ovvero dove si trova il blocco all'interno di tale LUN. LBA potrà essere, ad esempio, la terna che identifica i settori in un disco rotativo.

Per astrarre anche il tipo di disco utilizzato, si utilizza un altro livello di indirezione, infatti, dagli LBA fisici, vengono ricavati degli LBA **logici**, che verranno utilizzati effettivamente nella comunicazione. Questo grado di indirezione, permette di aggiungere **metadati** agli indicatori di blocco: è possibile aggiungere altri blocchi **fisici** ai quali tale blocco logico è disponibile, per implementare la **ridondanza** in maniera molto efficace!

Un'altra utilità di questi metadati è quella di realizzare il load balancing dell'utilizzo dei dischi fisici: se ho un file contiguo, posso mettere i blocchi contigui su dischi fisici diversi. In questo modo, il blocco contiguo viene letto, sostanzialmente, in **parallelo**, perché letto da dispositivi fisici **diversi!**

Ovviamente, questi metadati sono importantissimi, poiché **rappresentano l'informazione** (senza metadati, gli id fisici sono inutili, poiché il fs ragiona per indirizzi **logici!**), pertanto vanno anch'essi **ridondati**.

- Response times vs arrival rate for disks
- Response time is more than halved



Il carico ridotto dalla parallelizzazione dei blocchi riduce di più della metà il response time

I metadati

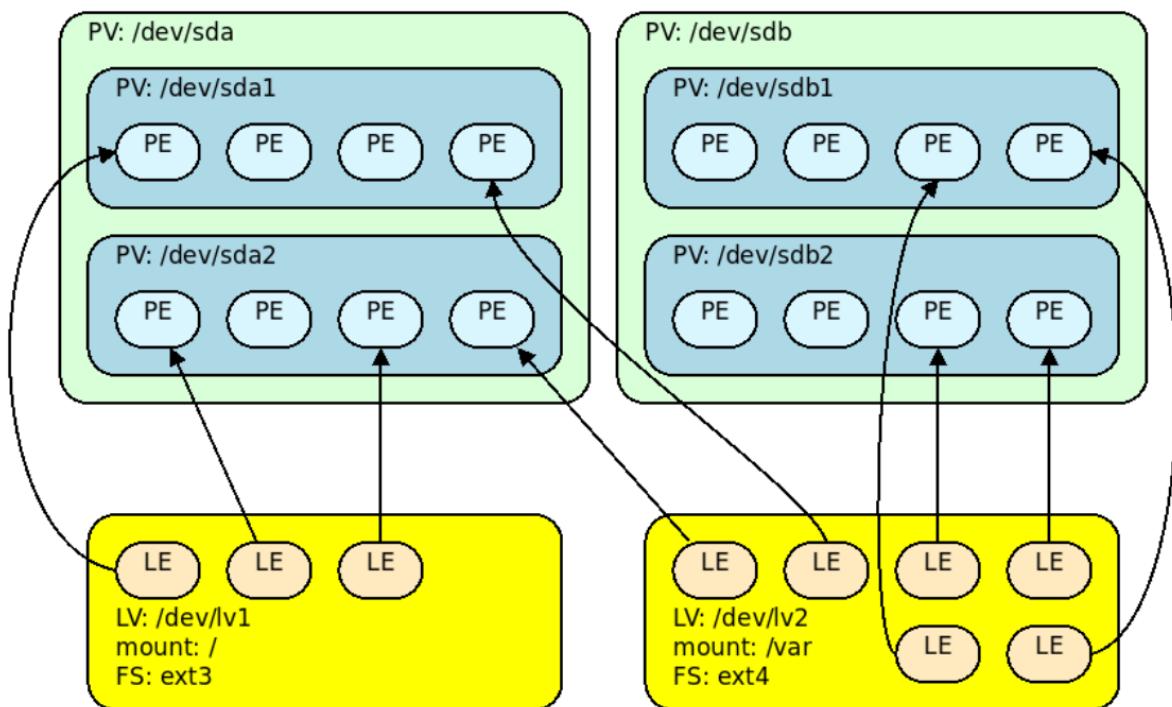
Come già detto, i metadati (logical extents) vengono mappati a uno o più **blocchi fisici** (physical extents).

PV: Physical volume

PE: Physical Extent

LV: logical Volume

LE: Logical Extent



Idealmente, da come si nota nella figura, è possibile utilizzare anche filesystem diversi per accedere ai dati!

L'approccio può essere **trasparente** o a livello di **metadati**. Nel primo si fa una richiesta col nome del file, sostanzialmente un logical extent, che deve essere tradotto in physical extent.

L'altro approccio permette di andare a fare una query direttamente sui registri dei metadati (tramite, ad esempio, una determinata REST API) e recuperare i dati “a mano” dai physical extent.

I metadati abilitano operazioni che sembrano complesse, come lo snapshotting:

- Mark all blocks of a file as copy-on-write
- Subsequent writes will create a copy
- Operation only **on meta-data**

Le operazioni, infatti, vengono fatte **soltanto sui metadati, non sui blocchi effettivi del disco!**

Useful for VM images

- Snapshot of VM for checkpoint and restore
- Multiple copies of same VM with same base image
- Same mechanism based on CoW

File based (NFS e SMB/CIFS)

Il filesystem viene **esteso su tutta la rete**. Viene quindi utilizzato un **network filesystem**, che è in grado di dare una visione “unita” di più macchine.

Dal punto di vista di un client, l'accesso ad una cartella remota viene fatta come se fosse locale, pertanto le chiamate di sistema sono sempre quelle!

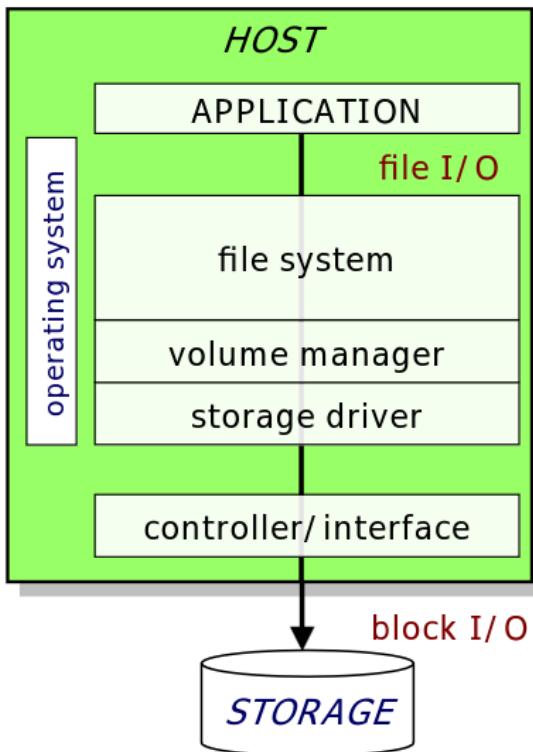
- Block-based is still highly used in cloud
 - More popular than filesystem-based approach
- Several reasons
 - Easy to manage
 - Highly scalable
 - More flexible (actual FS is implemented on top)
- Main reason
 - Extremely efficient with **scattered files**
 - Highly used in VM images

Storage infrastructure scope

Con “scope” dello storage si intende la “larghezza” e la “visibilità” dello storage stesso.
Si declina in tre principali infrastrutture:

- Direct Attached Storage (DAS): identifica un **disco** attaccato localmente ad una macchina;
- Network Attached Storage (NAS): identifica un **disco visibile a più macchine** (ma sempre un disco solo è! Single point of failure quindi);
- Storage Area Network (SAN): vi sono più dispositivi dedicati allo storage, un sottinsieme del datacenter è dedicato allo storage.

DAS



Non vi sono protocolli di rete in gioco, semplicemente vi è la gestione di un dispositivo **fisico**.

L'interazione, quindi, viene gestita interamente dal sistema operativo dell'host.

Questi blocchi non possono essere **virtualizzati** e sono generalmente visti localmente, anche se possono essere eventualmente esposti network-wide.

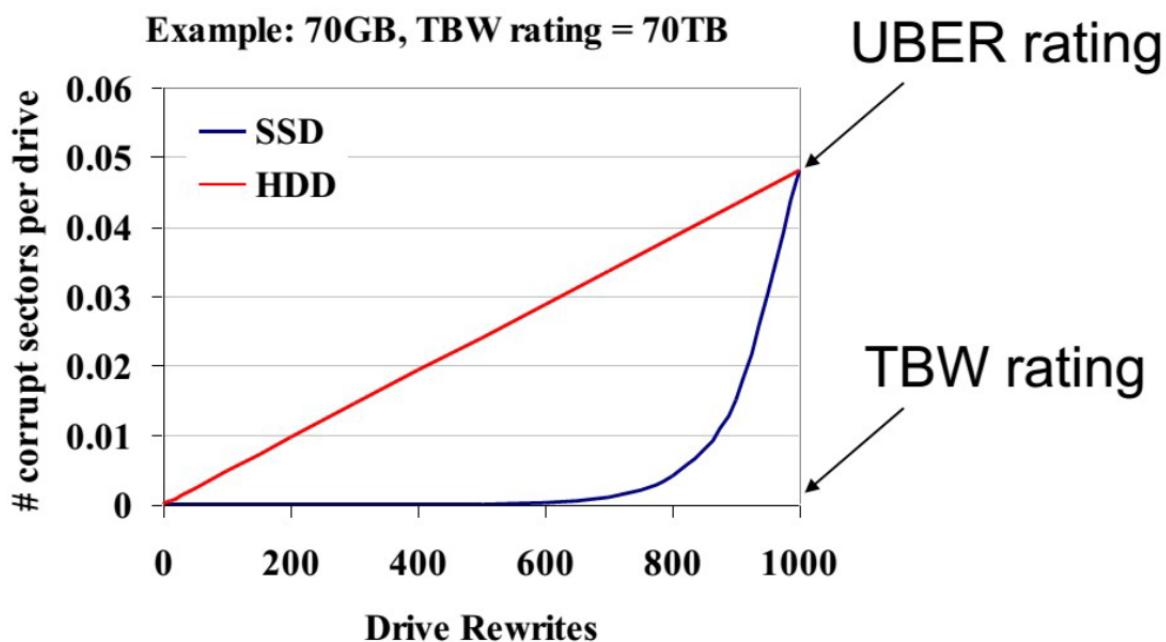
NAS

Non vi è bisogno di collegare direttamente il server che deve fruire dello storage allo storage stesso. Di mezzo vi è un'interazione fra storage e server che avviene mediante lo stack TCP/IP. Il controller del disco non è più il server che deve fruire dello storage, ma sarà il sistema operativo presente sul NAS stesso, che esporrà delle interfacce per accedervi.

L'accesso remoto può essere fatto sia in modo **block-based** che in modo **filesystem-based**.

⇒ poiché possono essere usati diversi tipi di dischi, è possibile usare SSD as cache e disks as large slow data (o dati sequenziali).

- Some metric to consider when choosing storage
- UBER (Unrecoverable error ratio)
 - Occurrence of data errors, equal to the number of data errors per bits read
- Endurance rating (TBW rating)
 - Number of TB that may be written to the SSD while still meeting the requirements.

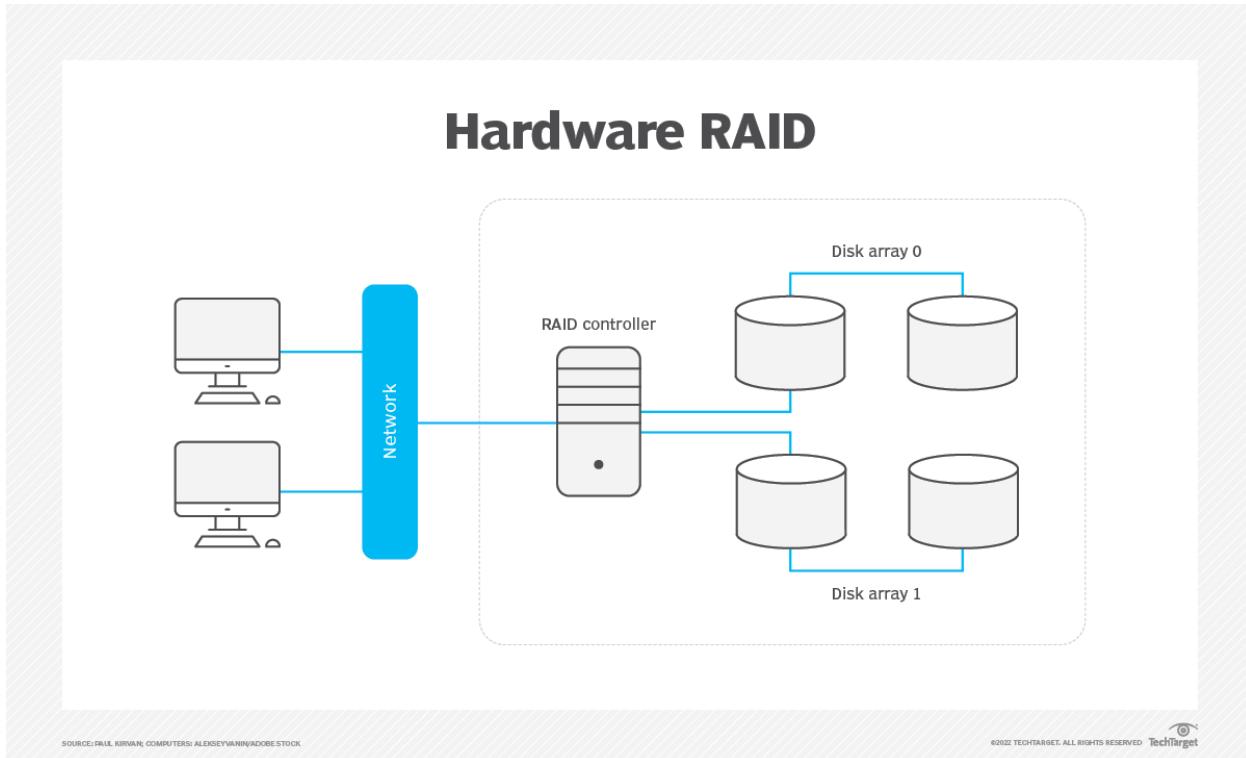


⇒ Attenzione alla scelta dei dischi: gli HDD hanno un tasso di failure sostanzialmente proporzionale alla loro età, quindi sono abbastanza “prevedibili”. Gli SSD, d’altro canto, no. Dopo un certo numero di rewrite, le prestazioni crollano vertiginosamente. Per questo, quando si fa un array RAID di SSD, è buona norma comprare gli SSD da lotti diversi, perché SSD con la stessa età **si rompono tutti allo stesso momento**.

I controller RAID di questi NAS possono essere sia software che hardware.

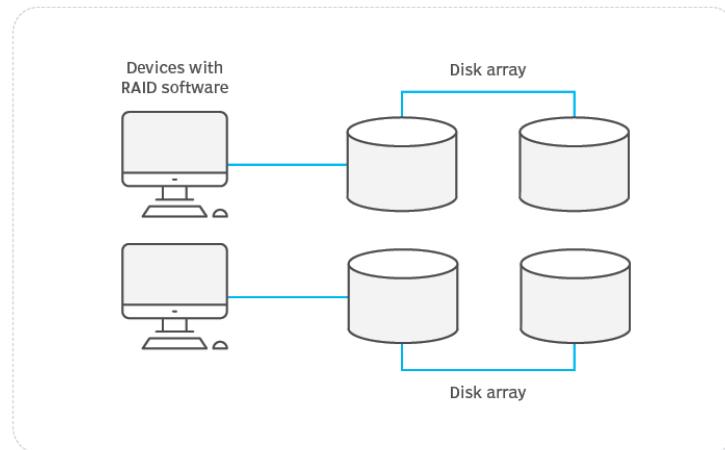
- hardware (**accelerated**) raid: il vendo del NAS ha implementato RAID (che è **sempre implementato via software**) nel firmware del controller, spesso in maniera

proprietaria, pertanto se si rompe il controller si perdono tutti i dati riguardanti a quel controller;



- “software raid”: il raid è implementato tramite standard generalmente open e non proprietari, pertanto anche se un disco esplode o se un controller (i.e. un PC che si interfaccia coi dischi) esplode, il problema non si pone.

Software RAID

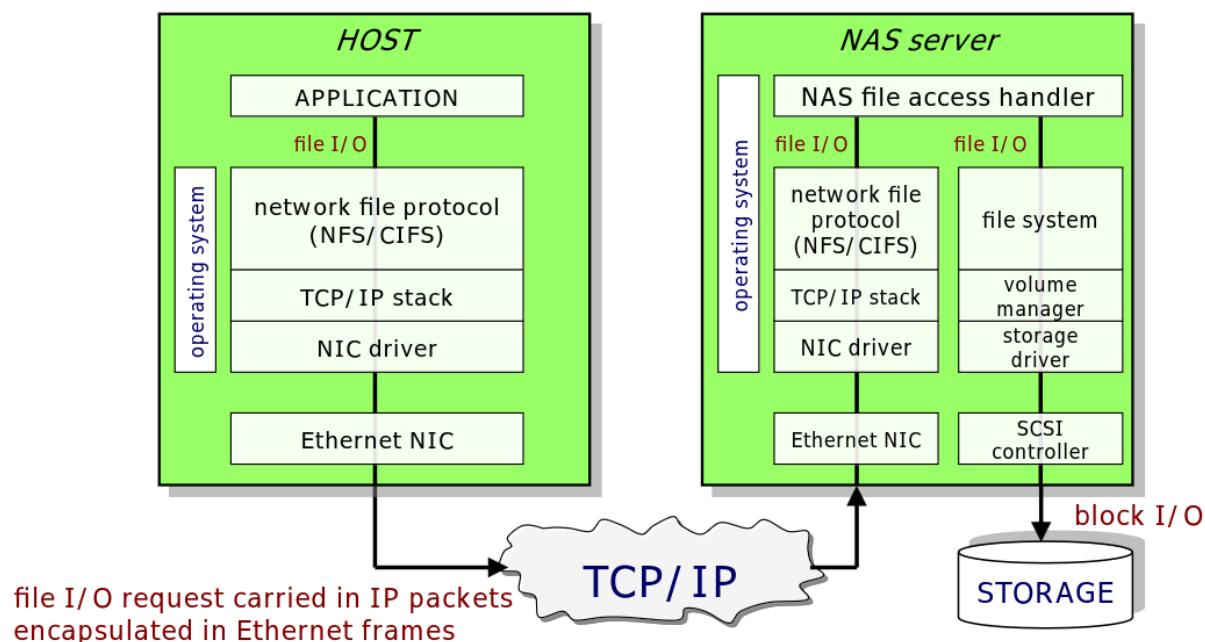


SOURCE: PAUL KIRKIN; COMPUTERS: ALEKSEYVANIN/ADOBESTOCK

©2022 TECHTARGET. ALL RIGHTS RESERVED TechTarget

NAS – The big picture

UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA



Generalmente un NAS include più schede di rete per ridondanza e per load-balancing.

SAN

Lo storage gestito da questo genere di sistemi sta nell'ordine dei PB, con interfacce di rete capaci di gestire centinaia di Gpbs ⇒ infatti gira tutto su fibra ottica.

Vi sono vari layer:

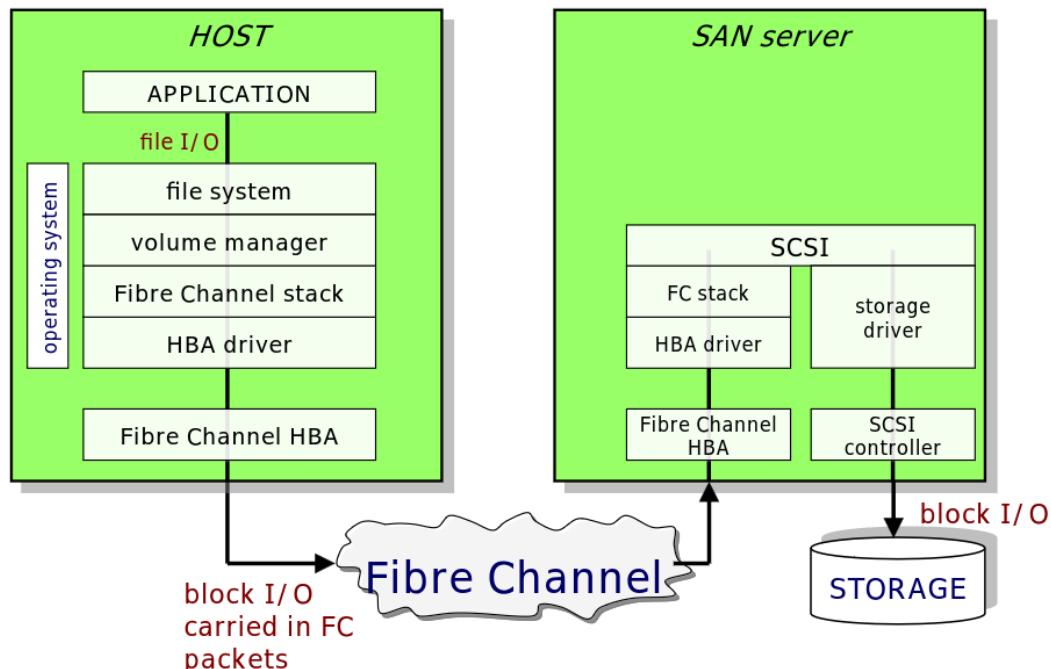
- host layer: il frontend che **accede** ai dati, che generalmente viene **replicato** (i.e. una rest api che accede ai dati) e che rappresenta l'endpoint per accedere alla SAN;
- fabric layer: la fibra ottica che mette in collegamento gli switch e i router;
- storage layer: dischi che sono tipicamente pensati per esporre i dati in maniera block-based e in modo estremamente flessibile e reliable ⇒ RAID e JBOD.

L'accesso è tipicamente derivante da protocolli simil-SCSI.

SAN – The big picture

UNIMORE

UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA



Software defined storage

Il termine utilizzato per virtualizzare (o meglio, **astrarre**) lo storage è **software defined storage**. Con questo approccio è possibile nascondere la complessità dell'accesso ai dischi tramite API che astraggono il tutto in modo **automatizzabile e programmatico**.

Questo vuol dire che all'interno della mia storage pool avrò **dischi diversi, filesystem diversi, vendor diversi**, insomma, mi serve un **layer che astrappa tutto questo genere di complessità e mi permetta di gestire SAN estremamente eterogenee**. ⇒ **Hardware independence!**

⇒ Software RAID! Storage pooling! SLA support...

Un sistema SDS deve essere in grado di supportare quanto più il concetto di “plug-n-play” dell’hardware.

Le modalità di accesso ai file vengono anch’esse virtualizzate:

- block
- file
- **object**

Ceph

Ceph è un filesystem che abilita SDS in maniera estremamente semplice.

Permette di distribuire i dati su più dischi mediante l’algoritmo **CRUSH** (hash-based load balancing).

Da un punto di vista architetturale, Ceph riutilizza il concetto di **metadati** (come in block-based) e introduce il concetto di **journaling**, quindi tutte le operazioni sul filesystem (distribuito!) sono **atomiche**.

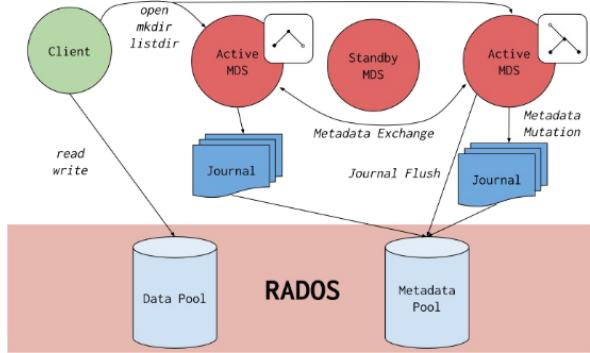
- Architectural overview

- **MDS**

- Meta Data Server
- Support for load balancing

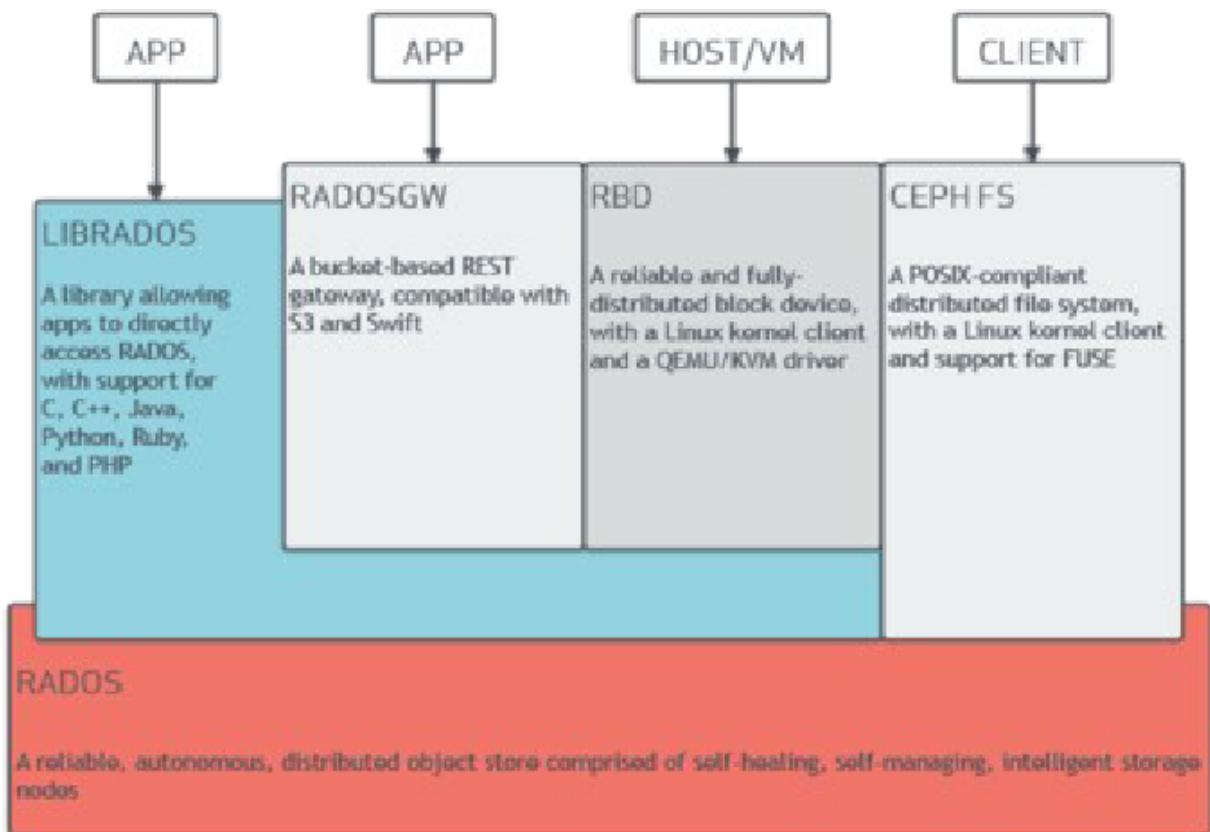
- **RADOS**

- Reliable Autonomic Distributed Object Store
- RBD: RADOS Block Device
- RGW: Radow Gateway



Le API per accedere ai file sono:

- S3-like
- RADOSGW (sempre object)
- RBD (block storage vm-friendly)
- CEPH FS (filesystem-based per i client Linux che vogliono navigare il tutto)
- FUSE ⇒ usermode filesystem, ottimo



SDN

Due trend abbastanza diffusi vanno diffusi sono quelli della NFV (Network Function Virtualization) e la SDN (Software Defined Network).

VFN serve ai grandi ISP per ridurre i costi OPEX e CAPEX, utilizzando hardware più “generico”, ma comunque dedicato, usando sistemi operativi standard per realizzare le funzioni di routing e switching che servono.

L'obiettivo, quindi, è quello di abbandonare i router e gli switch “dedicati” (hardware verticali), ma si utilizzano macchine con hardware più “off the shelf”. Ovviamente questo concetto non è applicabile a tutti gli scenari (IXP e backbone richiedono hardware dedicato).

Tutto ciò diventa molto interessante quando si ha uno scenario di virtualizzazione: dal momento in cui le macchine virtuali devono parlare tra di loro, posso **virtualizzare un router che le colleghi**. A questo punto, anche i dispositivi di rete diventano **virtuali**.

SDN nasce più verso il mondo cloud-oriented, dove l'accento è più sul problema della gestibilità. Quando si lavora con un datacenter vi sono migliaia di macchine fisiche che gestiscono milioni di macchine virtuali che hanno necessità di automatizzazione: regole NAT per garantire l'accesso alle macchine virtuali, regole di indirizzamento che cambiano se le VM vanno su o giù,

Avendo hardware “proprietario”, la configurazione cambia per vendor, è necessario, anche qui, qualcosa di standard (per rispettare lo SLA, molto importante).

Servono quindi dei dispositivi di rete che siano **cloud-enabled**, all'interno del datacenter: la logica di switching e di routing viene svincolata da hardware proprietario per essere quanto più flessibile e accessibile (si fa tutto con hardware e macchine custom anche qui).

Motivazioni

Le motivazioni principali sono le stesse che hanno portato allo sviluppo del cloud: disponibilità, prestazioni alte, ridondanza e quant'altro.

Garantire questi elementi a livello cloud richiede che gli switch e i router siano

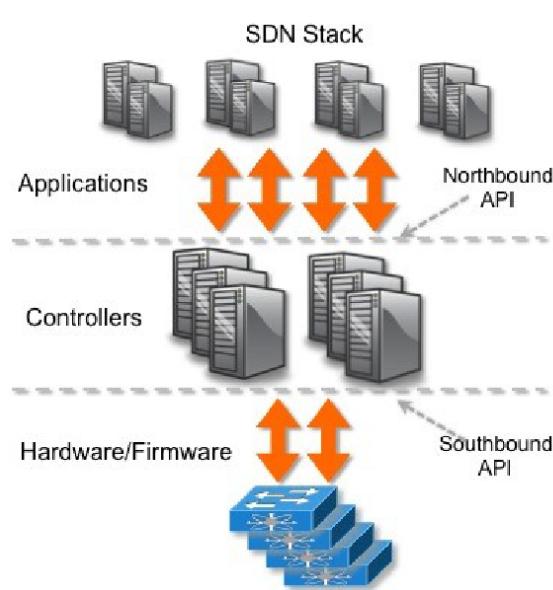
“programmabili”, siano dunque

software-defined. Ad esempio, dovrebbe essere possibile che i traffico venga switchato in base al carico corrente della rete, cosa che router e switch “classici” non possono fare (perché non sono così estendibili, dato che sono molto verticali sul problema di switching e routing). Serve una gestione programmabile della cosa! Un altro esempio: dato il mio network setup, le mie VLAN e tabelle di routing, mi serve un meccanismo che permetta di instradare i pacchetti opportunamente anche quando lo scenario cambia drasticamente, ad esempio quando un’intera zona di VM viene migrata ⇒ anche qui servono sistemi che si autoriprogrammino!

SDN

Per SDN, vengono definiti due elementi principali:

- data plane: ciò che manipola i dati, che manipola i pacchetti. Nel caso di un router, la parte di data plane è il forwarding dei pacchetti e quant’altro;
- control plane: ciò che manipola **la configurazione, l’intelligenza degli apparati**.



Gli apparati sono essenzialmente “brainless”, non prendono decisioni, ma eseguono delle regole che vengono stabilite dal **control plane**. Il data plane e il control plane sono due insiemi di dispositivi che sono anche **fisicamente separati**.

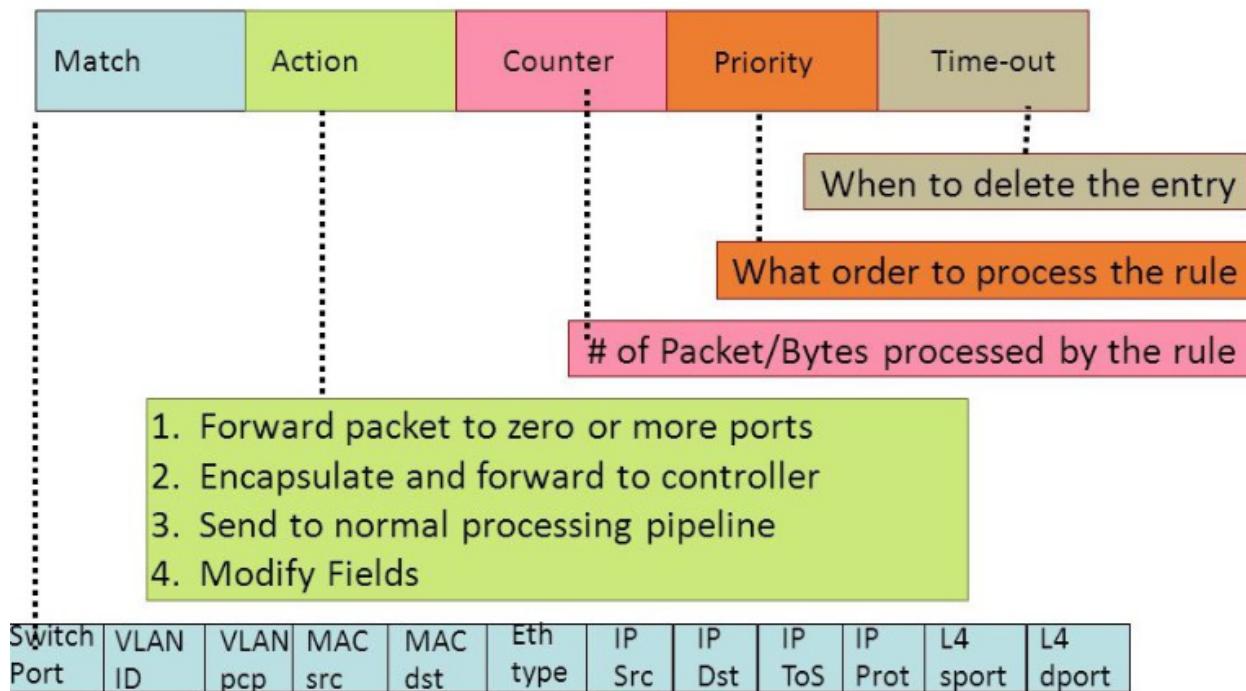
Il control plane, quindi, si interfaccia (tramite la cosiddetta **northbound api**), ad esempio, con l’hypervisor per “ascoltare” modifiche sulle macchine virtuali (ad esempio) e, di conseguenza, notificherà queste modifiche sul data plane (tramite la **southbound api**).

Praticamente lo si può pensare come kubernetes a livello network: ci sono i nodi e le istanze di kubelet che vengono configurate dall’api server in base a cose che l’api

server vede (vanno giù dei nodi: ok, cambia tutte le regole sui nodi per dire che quel servizio non è più disponibile, etc etc).

OpenFlow

Con OpenFlow, le regole sono fatte come in questa immagine:



- match: la condizione per matchare la regola. Il predicato di matching può usare tantissime informazioni:

Relevant fields (H2N, Network, Transport)

- Input port
- VLAN ID and priority (PCP)
- MAC source address
- MAC dest. address
- Ethernet type field
- IP source address
- IP destination address
- IP protocol
- IP ToS
- TCP/UDP source port
- TCP/UDP dest. port

Notare come i predicati spannino **più livelli dello stack!** Uno switch SDN può gestire automaticamente **qualsiasi livello dello stack**, rendendolo molto generico.

- action
- counter: contatore che stabilisce varie statistiche di utilizzo della regola (quanto traffico gira su questa porta? etc...)
- priority
- TTL

Il data plane, poiché non ha intelligenza ed “esegue solo ordini”, fa operazioni che sono molto facili da ottimizzare in-hardware.

Soutbound API

I messaggi che possono essere inviati tramite la southbound API sono:

- controller —> switch
- async messages da switch —> controller
- symmetric messages

I primi sono classici messaggi di “controllo” e configurazione

Features	Request the capabilities of a switch. Switch responds with a features reply that specifies its capabilities.
Configuration	Set and query configuration parameters. Switch responds with parameter settings.
Modify-State	Add, delete, and modify flow/group entries and set switch port properties.
Read-State	Collect information from switch, such as current configuration, statistics, and capabilities.
Packet-out	Direct packet to a specified port on the switch.
Barrier	Barrier request/reply messages are used by the controller to ensure message dependencies have been met or to receive notifications for completed operations.
Role-Request	Set or query role of the OpenFlow channel. Useful when switch connects to multiple controllers.
Asynchronous-Configuration	Set filter on asynchronous messages or query that filter. Useful when switch connects to multiple controllers.

I messaggi asincroni, invece, vengono mandati quando è richiesto il controllo del control plane

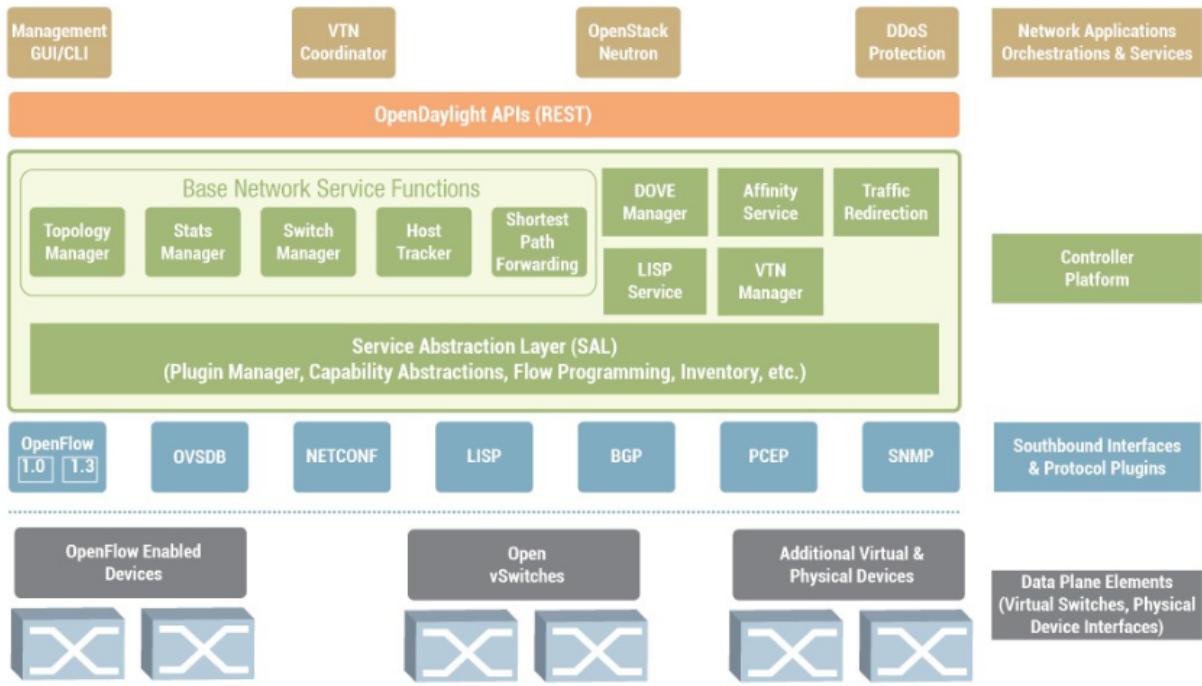
Packet-in	Transfer packet to controller.
Flow-Removed	Inform the controller about the removal of a flow entry from a flow table.
Port-Status	Inform the controller of a change on a port.
Error	Notify controller of error or problem condition.

I symmetric messages sono usati per healthcheck e simili:

Hello	Exchanged between the switch and controller upon connection startup.
Echo	Echo request/reply messages can be sent from either the switch or the controller, and they must return an echo reply.
Experimenter	For additional functions.

Control plane

Il control plane è la parte responsabile della gestione della configurazione dell'infrastruttura. Nella pratica, viene spesso utilizzato **OpenDaylight**.



I controller hanno una visione globale del sistema: questo comporta una forte complessità nel raccogliere lo stato da sorgenti diverse, pertanto si tende a frammentare la rete in vari pezzi, con uno strato controller per frammento.

Datacenter e Software Defined Datacenter

I datacenter “classici” non sono più funzionali. La logica cloud è intimamente legata alla “software-definedness” e i “vecchi” datacenter non sono così elastici e adattabili. I datacenter cloudnative devono essere in grado di rispettare tutti i concetti di elasticità, disponibilità, virtualizzazione,

Tipicamente un datacenter è composto da:

- blade servers
- racks: armadi (raggruppati in **pod**)
- SAN

⇒ le slide fino a 30 bastano

Datacenter network

Da non sottovalutare è la **struttura logica del datacenter** e come questa venga mappata sulla struttura fisica.

Tipicamente, un datacenter è un insieme di virtual machine che va a collaborare per raggiungere una serie di obiettivi.

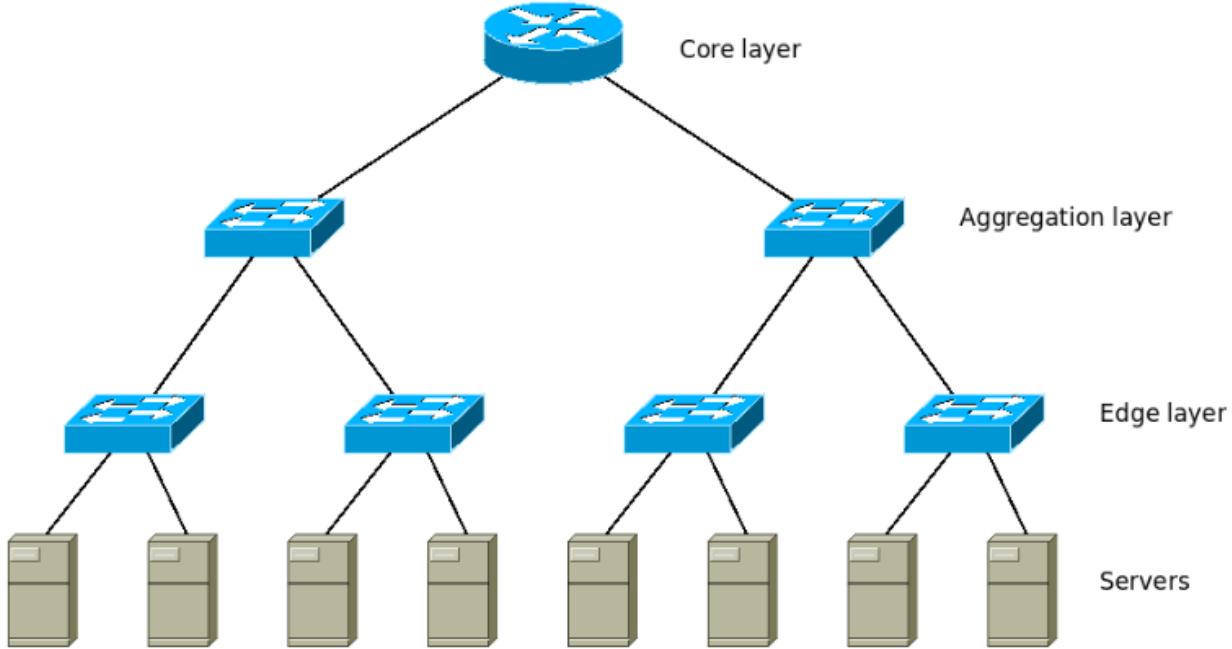
Top Of Rack: ogni rack ha uno switch dedicato;

End Of Rack: tutti i rack di una aisle si connettono ad uno switch “alla fine” della aisle stessa.

-
- Some **simplifying** assumptions
 - Limited **heterogeneity** in network technologies
 - Unified **management**
 - Uniform policies
 - Controlled traffic sources
 - Limited presence of **malicious** traffic
 - Topology **design**
 - Not uncontrolled growth
 - Compare with Internet scenarios
 - Totally different challenges

Dal punto di vista della rete, tipicamente l'approccio è gerarchico, ovviamente (ad albero).

Vi sono determinati
livelli all'interno della rete:



Il primo è quello dei **server**, che necessitano principalmente di connettività a livello 2 con velocità 1-10Gbps.

Il livello edge è quello che interconnecte i blade server tra di loro (all'interno di un rack [ToR], oppure all'interno di una aisle di server [EoR]). Generalmente, questo livello viene chiamato **access level**, che implementa policy di **isolamento**, **ACL**, **QoS**, e **fault management**. Viaggia a 1-10Gbps. I primi due livelli sono, quindi, “locali” al rack.

L'aggregation layer mette in comunicazione tra di loro i **pod** nei quali sono racchiusi i **rack** di determinate “isole”. È importante che vi siano policy di QoS e traffic shaping, oltre alla fondamentale **ridondanza** di questo livello. Una failure a questo livello mina la stabilità dell'intero datacenter.

Il core layer è la **backbone** del datacenter, che interconnecte le varie sale del datacenter stesso. Tipicamente il core layer comprende anche il border router del datacenter. Poiché tutto il traffico in uscita passa per questo layer, la velocità dei link è a multipli di 10Gbps.

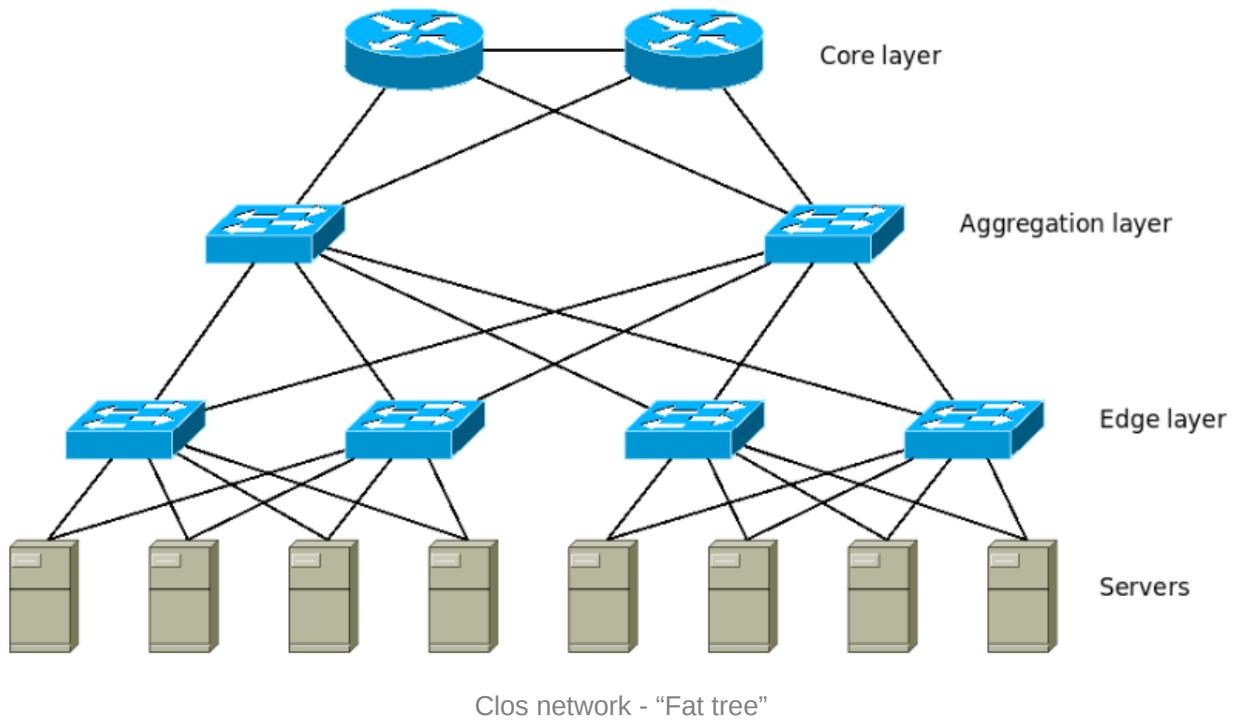
Le zone SAN e di outbound non rientrano in questa architettura perché vengono gestite separatamente.

Ovviamente non è una topologia ad “albero puro”, perché:

- Limits of tree topology
- Single Points of Failure
 - Core layer is not replicated
 - A failure can isolate the data center
- Bottlenecks
 - Critical for sparse interactions
 - Aggregation and core layer may be overloaded
- Solutions
 - Replication
 - Topology re-design

Tipicamente la gerarchia è chiamata **fat-tree**: i piani alti dell'albero sono ridondati (contengono “più root”).

Clos network



Le fat tree network vengono implementate mediante il modello di rete **Clos**, che garantisce un livello di ridondanza per **ciascun livello** della rete. Per causare una partizione, bisogna che vengano persi **due elementi di switching specifici** che fanno da “backup” l’uno dell’altro, riducendo così la probabilità di failure.

Una cosa particolare è che le clos network sono implementate anche **internamente agli switch**, per realizzare i collegamenti tra le porte.

Negli switch “classici”, non clos quindi, vi è un elemento di interconnessione per ogni coppia di porte (quindi tutte le porte sono collegate l’una con l’altra), creando una **matrice completa** ($8 \text{ porte} = 64 \text{ switch}$, ...), pertanto switch di questo tipo sono costosissimi per un alto numero di porte.

Una soluzione a questo problema è implementare le **clos network all’interno degli switch**: si usano più livelli di ridondanza, ma non $O(n^2)$! Sostanzialmente, al posto che un collegamento tra porte (per ridondanza) **fisico**, vi è un collegamento “logico”, dato da “più porte collegate tra di loro”.

L’intercollegamento fra porte può quindi diventare un “percorso”, qualora le porte non siano fisicamente collegate tra di loro.

Datacenter challenges

Il problema principale di un datacenter è quello della **scalabilità (verticale o orizzontale)**.

Web cluster architecture

Come si realizza un sistema che espone un **singolo IP (routable)**, ma che, behind the scenes, è composto da molteplici server?

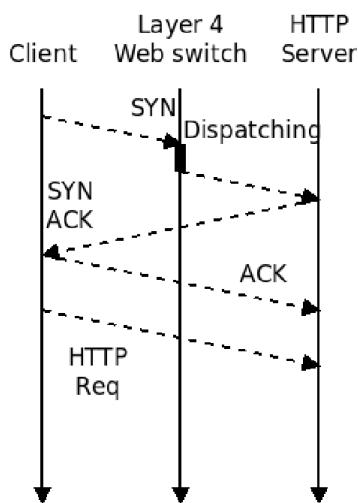
- Seen from outside
 - One hostname (www.unimore.com)
 - One IP address (130.186.5.242)
- IP address
 - Routable address
 - Called **Virtual IP** address
 - Only public IP for site

Il virtual IP può essere un indirizzo in heartbeat oppure può essere mappato su più server tramite BGP.

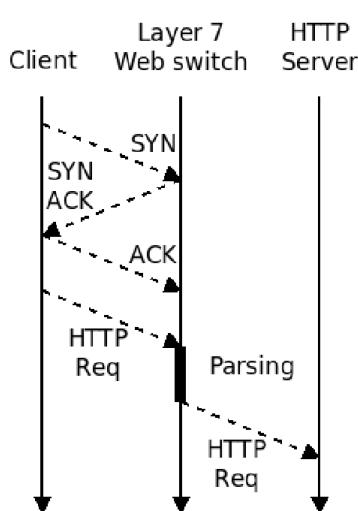
Pertanto il web switch viene esposto al pubblico con quel virtual IP, mentre il backend, ovvero i server “sotto” allo web switch, avranno tutti indirizzi privati.

Vi sono due approcci per costruire lo web switch:

- L4: non si lavora con applicazioni di livello applicativo, ma si arriva solo al trasporto (IP:porta). Si riesce a tracciare tranquillamente la connessione, ma non il suo contenuto;
- L7: uno switch di L7 può avere informazioni applicative per fare un balancing più fine.



Uno switch di livello 4 è molto più veloce di quello di livello 7, questo poiché si interfaccia **direttamente con la destinazione** (il server di backend a cui è diretto il pacchetto). Quindi è possibile forwardare i pacchetti SYN direttamente alla destinazione, facendo semplicemente da proxy per il SYN.

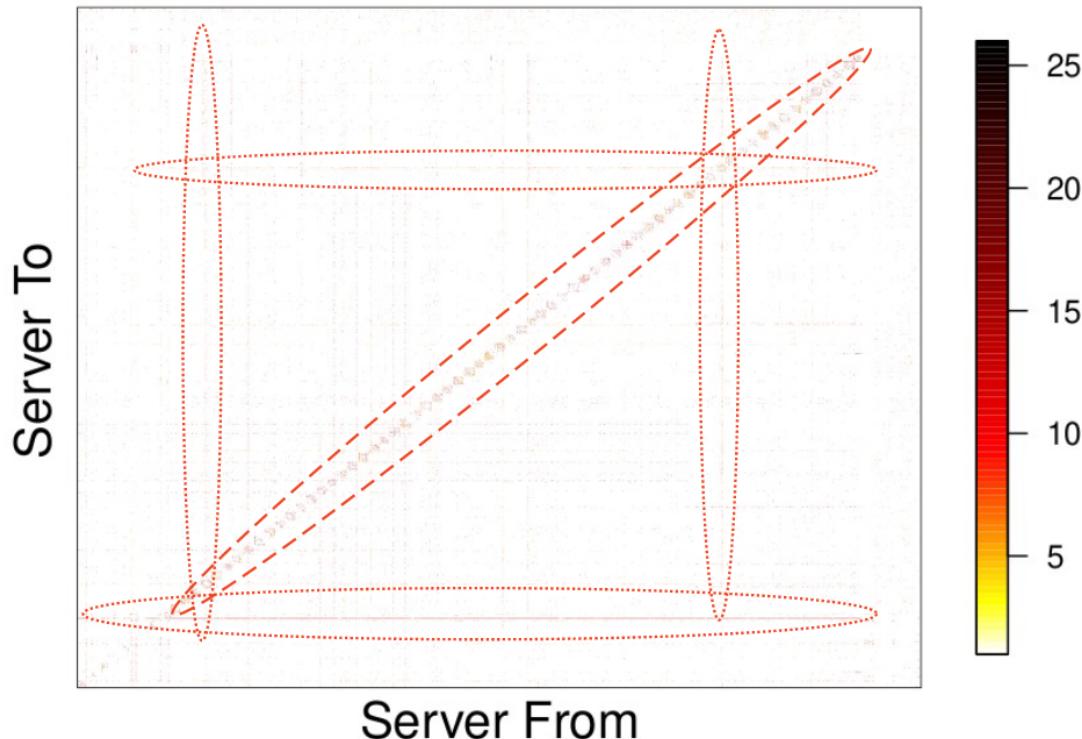


Uno switch di L7, invece, deve salire tutti i livelli dello stack, non può semplicemente forwardare il pacchetto di SYN (per parsare la richiesta mi serve **la connessione che me la manda - la ho solo dopo l'handshake**).

- Layer 4 Web switch
- TCP connection
- Content-blind dispatching
- No session support
- Layer 7 Web switch
- HTTP connection
- Content-aware dispatching
- Can support sessions

Ovviamente, un approccio “single datacenter” ha i suoi limiti, pertanto bisogna supportare **global load balancing** ⇒ **BGP cloud load balancing**.

Traffico all'interno di un datacenter



I punti sulla diagonale indicano che un nodo in un determinato rack, va a comunicare con un nodo nello stesso rack: la comunicazione tra nodi (VM) non ha pattern casuali, ma ha una forte località. Alcune macchine sono fortemente propense a comunicare tra di loro.

Vi sono altri pattern particolari:

- alcuni nodi parlano con **tutti gli altri** (righe verticali): sono i nodi che si occupano di servizi di coordinamento (storage, dns, routing, ...). Questi sono nodi che devono essere in una posizione **baricentrica** all'interno del datacenter.

Lo scambio di dati, quindi, è:

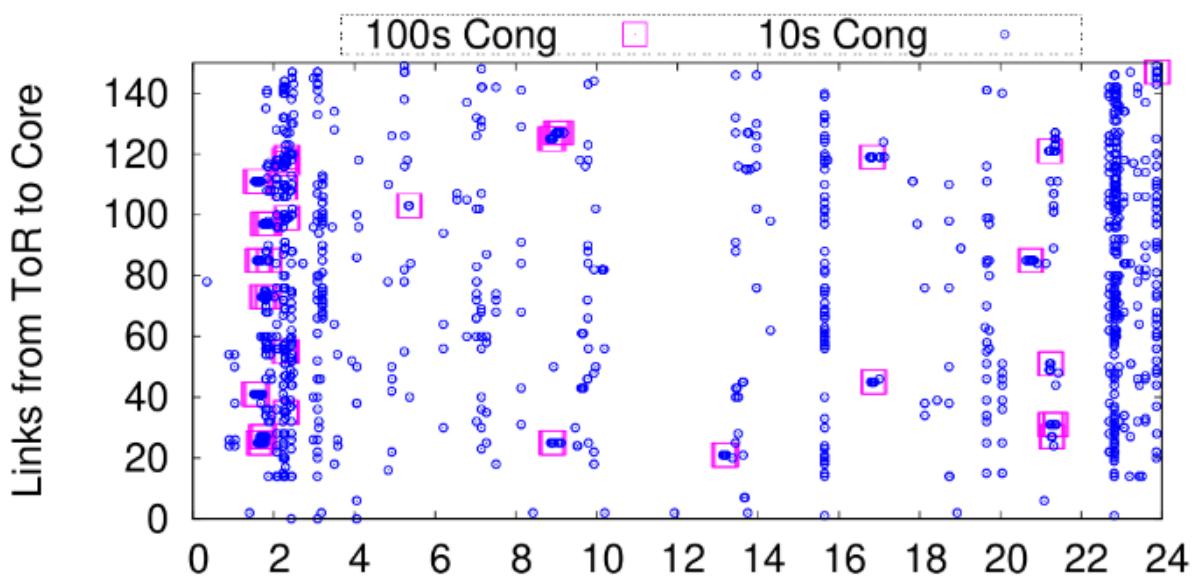
- piccoli frammenti nello stesso rack (mice)
- grossi scambi di dati all'esterno del rack (elephant)

Queste due classi di traffico hanno ottimizzazioni completamente diverse: per trasferimenti enormi, la latenza non è una misura molto importante (latenza alta per gigabyte di dati va benissimo), ma lo è per flussi più piccoli!

⇒ flussi piccoli: banda bassa e poca latenza

⇒ flussi alti: banda alta e latenza anche moderata/alta

Il momento critico dei workload è quello nel quale partono i backup delle macchine virtuali: è il momento di massima congestione (la notte, generalmente).



La cosa principale da ricordare è che nel datacenter vi sono **bolle di macchine tra di loro affini**: il traffico è estremamente locale.

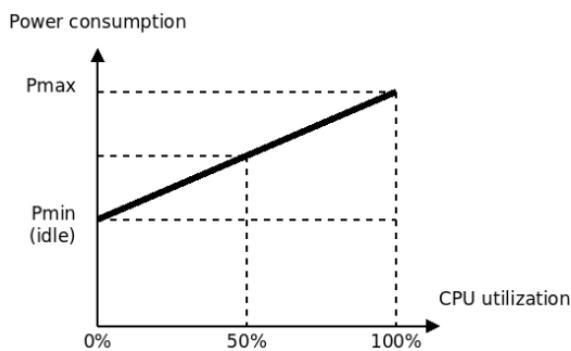
Per questo ultimo motivo, è necessario mappare i deployment di macchine virtuali "affini" in maniera abbastanza

locale, ovvero in rack/macchine "vicine" tra di loro.

Macchine virtuali che sono fortemente affini dovrebbero, in realtà, trovarsi o sullo stesso nodo fisico o su nodi a distanza di 1-hop. Se questo non è il caso, livelli di rete più alti sono afflitti (perché devo comunicare fra reti diverse), causando congestione (perché le due macchine, poiché affini, comunicano spesso).

Oltre a queste considerazioni, bisogna sempre garantire **load balancing, consumi ottimizzati, utilizzazione della rete e riscaldamento!**

- **Power consumption**
 - Server consolidation
- Server power model
 - $P = P_{\min} + U \times (P_{\max} - P_{\min})$
 - $U \rightarrow$ Utilization

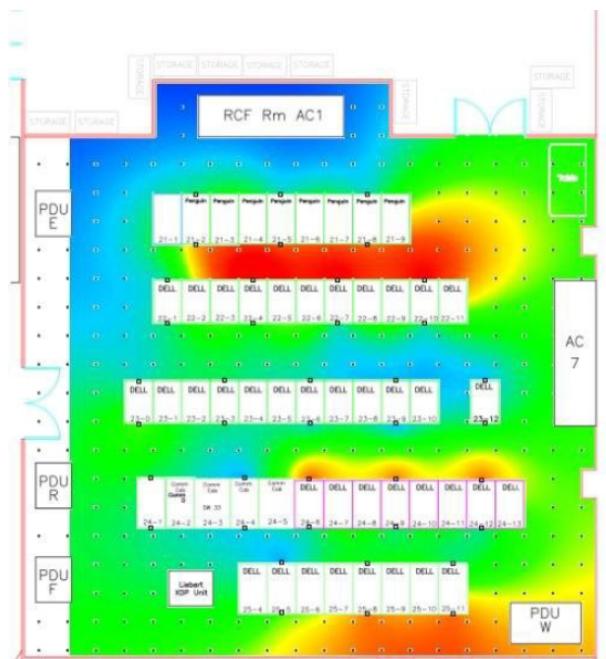


- Example
 - $P_{\min} = 0.5 \text{ KW}$, $P_{\max} = 1 \text{ KW}$
- 3 servers, half utilized
 - $U = \{0.5, 0.5, 0.5\}$
 - $P = 3(0.5 + 0.5 \times 0.5) = 2.25 \text{ KW}$
- Server consolidation
 - $U = \{0.75, 0.75, \text{OFF}\}$
 - $P = 2(0.5 + 0.5 \times 0.75) = 1.75 \text{ KW}$
 - Save 0.5 KW
(idle power of 1 server)

Server consolidation: spengo un server, migrando le macchine sui due. Consolidare i server può aiutare a risparmiare energia (3 server accesi con più “idle” consumano di più di due server accesi più utilizzati).

Per quanto riguarda il management del calore del datacenter, ciò che si vuole evitare sono i cosiddetti **hotspot**. Le macchine negli hotspot avranno usura maggiore, poiché lavorano a temperatura più alta di altre macchine!

- Heat management
- Identification of hot spots
- Concentration of servers with high utilization
- Mitigation of hot spot
 - Load distribution
- Switch-down data center areas
 - Can save cooling power



Nota bene: se alcune macchine non vengono utilizzate, è possibile **spegnere l'impianto di condizionamento per quelle macchine, risparmiando moltissimo (per 1KW di energia di un rack, 600W sono consumati dai condizionatori!).**

Oltre al singolo datacenter

Vi sono altri problemi che vanno oltre ai problemi “fisici” del datacenter, che sono sostanzialmente riguardanti all’instradamento di traffico fra più datacenter per applicazioni distribuite.

I possibili punti di congestione sono:

- last mile: rete del client che istanzia la connessione
- first mile: rete del datacenter in ingresso
- peering point: punti scambio fra AS

Gli AS, generalmente, non hanno problemi di banda, internamente, poiché composto da milioni di dispositivi, molto “localizzati” (vicini) tra di loro. I costi di località sono molto inferiori rispetto a quelli riguardanti i cavi usati nei peering point (transoceanici e quant’altro).

Da un punto di vista di ottimizzazione infrastrutturale, però, i cloud provider possono agire su problemi sui punti di first mile e peering. Ovviamente non può agire sulla last mile (non può portare la fibra in casa di tutti i clienti).

- Traffic growth
 - → congestion problems **exacerbated**
- Possible solutions
- Increasing **data center connectivity**
 - Address last mile problem
- **Replicating** service points
 - Address last mile and peering point problems
- Taking care of **data centers and network**
 - Management of world-wide AS
 - Integration with self-owned Content/Services Delivery Network

Google infra

L'infrastruttura di Google si sviluppa su tre layer concentrici (a partire dal centro):

- datacenters
- edge points
- edge nodes

I datacenter sono le varie **località** nelle quali Google offre i propri servizi.

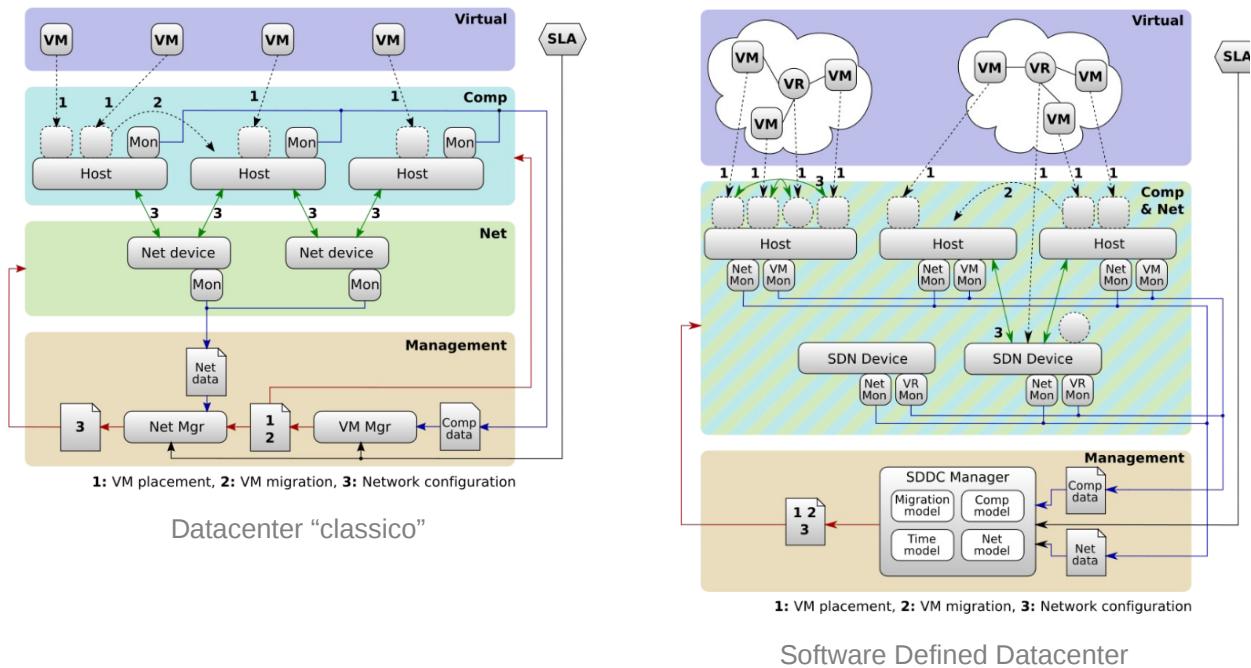


I point of presence (PoP) si occupano di routing “interno alla rete di Google”. L'instradamento viene fatto utilizzando quanto più possibile l'infrastruttura proprietaria, ottimizzata per il traffico di Google stesso, lasciando di gestire il traffico il meno possibile dagli ISP. Questo approccio permette a Google di appoggiarsi su reti “terze” solo quando deve arrivare “in casa” all'utente (quando arriva all'ISP).



Gli **edge nodes**, invece, formano la CDN di Google (la Google Global Cache). Tutti i contenuti statici vengono spinti all'estremo della rete. Sono i più numerosi e i più replicati di tutti. Questi nodi sono posizionati, generalmente, **internamente all'AS di cui fa parte l'utente finale**, riducendo di tantissimo la distanza della richiesta (pochi hop).

Software defined Datacenter



Quando si abbraccia la visione software-defined, il focus non è più sulle **singole** macchine virtuali, ma su un **cluster di macchine virtuali affini** (virtual networks). Gli elementi di switching non sono più **strettamente fisici**, ma possono essere **macchine virtuali che si occupano di questo** (Virtual Router).

Il layer “virtual”, quindi, si occupa della configurazione della rete e di come, ad esempio, le macchine virtuali vengono mappate sulla infrastruttura fisica (comp & net). Come si può ben vedere, la parte di calcolo e quella di networking si **fonde**, realizzando clustering di macchine virtuali simili in **virtual networks**. La configurazione di VM e reti virtuali avviene quindi allo stesso momento.

Problemi allo switch verso SD Datacenter

Static network management

L'utilizzo **massivo** delle VLAN può rendere lo shift molto difficile, poiché, tradizionalmente, la rete ragiona su host “fisici”. Bisogna utilizzare politiche di switching, quindi, VM-aware!

Resource fragmentation

È importante, visto che ora è tutto “fuso” in un solo magma, garantire load balancing efficiente fra tutte le macchine in gioco. Questa cosa, allo stesso tempo, però, potrebbe richiedere il dispatching su **nodi diversi** di macchine virtuali che richiedono di essere “localmente vicine”.

Poor server-to-server connectivity

I server devono avere una connettività **nell'ordine dei gigabit** anche **localmente**.

- **Poor server-to-server connectivity**
- Oversubscription
 - Not all links 100% utilized
 - Bandwidth in switching element can be less than sum of incoming link capacity
- Oversubscription rates
 - 10 for H2N-only links
 - 80 for (expensive) IP-based links
- Additional risk of congestion in data center core network

Evitare il problema di oversubscription: troppe richieste su un link solo, traffico non ben bilanciato.

Tecnologie abilitanti

- Integration of data center functions
 - Hyperconvergence
- Reducing network topology levels
 - Fabric computing
- Leveraging network traffic characteristics
 - Mices and elephants
- fabric computing: schiaccia l'albero della topologia della rete, cercando di connettere "bene" tra di loro tutti i nodi (bene = ridondanza e con velocità)

Hyperconvergence

L'idea alla base dell'iperconvergenza è quella secondo la quale si cerca di utilizzare hardware che ha **integrate** le funzioni del **datacenter software defined** (storage VM-aware, router VM-aware, etc..).

- Several types of hyperconverged infrastructures
- Integrated stack systems
 - Sever+Storage+Network+Application software
 - Built-in application appliance
- Integrated infrastructure systems
 - Sever+Storage+Network+Virtualization
 - General computing infrastructure
 - Cloud building block
- Possible scaling and combination with **fabric computing**

Esempio: software-defined rack

- VCE
 - Now part of Dell
- Joint venture
 - Cisco (network and computing)
 - EMC (Storage)
 - VMWare
- VBlock / VxBlock architecture
 - Modular building blocks



- Challenges addressed by hyperconvergence
 - Data-driven applications
 - Workload fast changes
 - → scalability and flexibility demands
- Benefits of hyperconvergence
 - **Agility** (low provisioning/deployment time)
 - Low cost per system (**CAPEX** reduction)
 - Low operating cost (**OPEX** reduction)
 - Simplified **management** (uniform architecture)

Fabric computing

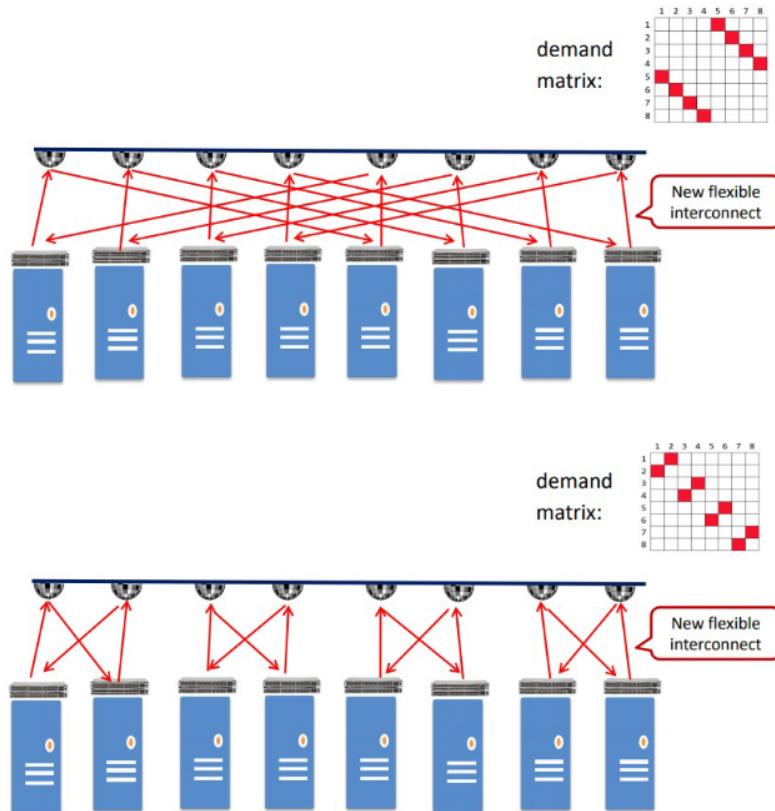
La struttura gerarchica di una rete “classica” viene “schiacciata” per rendere la rete più semplice e più interconnessa. Questo, viene realizzato mediante un approccio di **software defined networks**, centralizzando la gestione della rete in un control plane, avendo un data plane possibilmente **virtuale**.

Appiattendo la rete, viene gratis la comunicazione “locale” fra le macchine “affini” (perché tutti gli elementi sono “vicini”), grazie al **mesh** di dispositivi che si è realizzato con questo approccio.

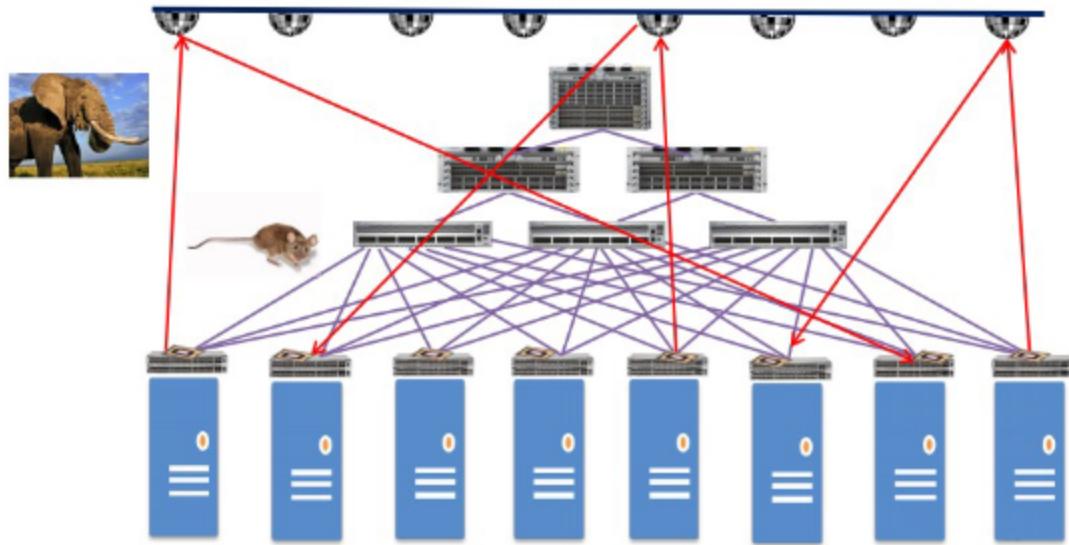
Mices and elephants

Traffico con caratteristiche diverse, richiede **diversa gestione**. Se il workload non è uniforme (e non controllato) è possibile trovare **overload su determinati link** e **underload su altri**: il traffico finale non è **bilanciato**.

La soluzione sta nel utilizzare **piccoli gruppi di link veloci** riconfigurabili dinamicamente (in base alla banda richiesta). In questo modo è possibile configurare i link in modo da richiedere, on demand, che tipo di struttura di rete si vuole utilizzare, in base alla richiesta.



È possibile, in questo modo, mappare i flow di dati “elephant” su determinati link, mentre i flow “mice”, su altri:



Gestione di un datacenter

Non ci sono solo i problemi “diretti” di un datacenter (networking, VM, ...), ma anche problemi più “nascosti”, che sono comunque molto importanti (legali, tasse, posizione geografica, ...).

⇒ vanno bene slide

Datacenter management

Tipicamente la gestione di un datacenter è legata ad un determinato obiettivo di ottimizzazione, che però è composto, generalmente, da più “sotto-obiettivi” che sono in contrasto tra loro. Un esempio classico è quello secondo il quale il datacenter debba servire quanto più traffico possibile, usando **meno energia possibile** (le due cose sono in contrasto!). In generale, quindi, un datacenter ha un determinato SLA da raggiungere minimizzando i costi. È quindi molto importante stabilire con correttezza i **key performance indicator** (KPI) sui quali si vanno ad eseguire le nostre analisi, che possono apparire come **variabili** della funzione di ottimizzazione o come **vincoli**.

KPI

Tipicamente, le metriche prese in considerazione sono:

- metriche riguardo il consumo energetico;
- metriche riguardo lo SLA
- metriche riguardo il failure rate
- metriche riguardo l'utilizzazione delle risorse e la loro gestione termica
- metriche riguardo la reliability/availability

Consumo energetico

Ci sono alcuni modelli specifici per descrivere il consumo energetico a livello di CPU e dischi, in base a diversi workload. Se non ci sono informazioni chiare riguardanti il workload, modellare il consumo energetico di un sistema diventa molto complesso, per via del fatto che il numero di parametri **esplode**.

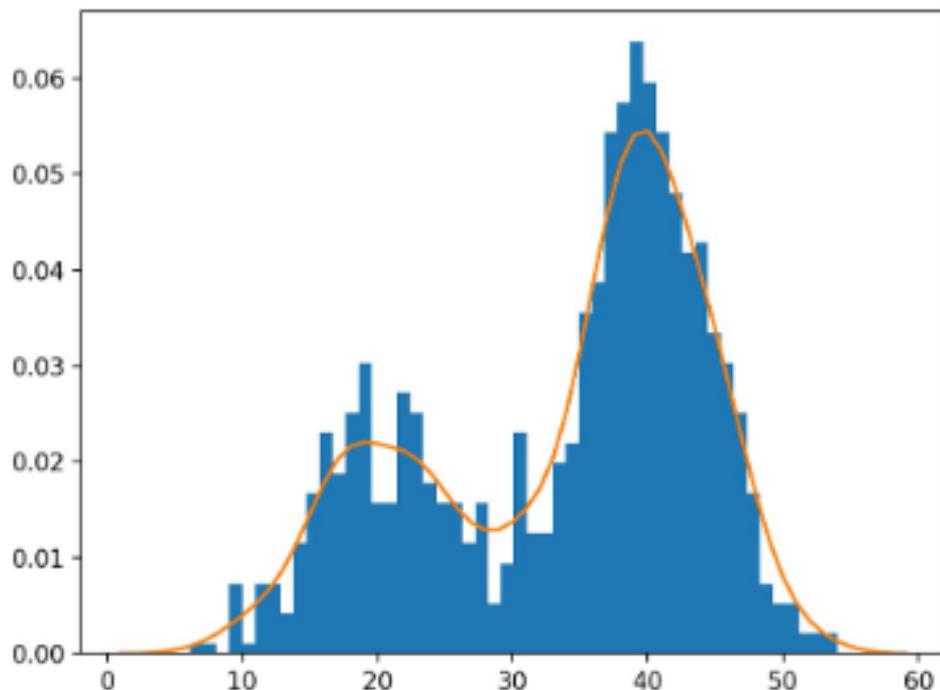
Può essere utile, in prima battuta, sapere qual è il mix energeitico che si ha durante la giornata: durante il giorno è possibile avere un'abbondanza di energia da fonti rinnovabili, mentre la sera **no**.

In base a questo, e in base al costo della **brown energy** più alto di quello della **green energy**, è possibile schedulare i carichi di lavoro maggiori durante i periodi di abbondanza di green energy.

SLA

Lo SLA è una **composizione di indicatori**, infatti viene generalmente misurato come un'aggregazione di valori.

Se, ad esempio, si vuole garantire uno SLA sul tempo di risposta, si avranno una serie di valori raccolti, che formeranno una **distribuzione di probabilità**:



Molto spesso, specialmente per distribuzioni multimodali, si è soliti utilizzare metriche come il **90-percentile**, che esprimono l'andamento della **maggior parte della misura** per la quale si vuole garantire un determinato SLA.

Riguardo questa distribuzione di valori, è importantissima la **observation window**: è fondamentale che questa finestra sia **rappresentativa** di scenari **reali**.

Cicli termici delle risorse

Si può ottimizzare in base alla temperatura **media** delle CPU e delle apparecchiature. Vi sono alcuni studi che vanno a studiare come metrifica l'acceleration factor, ovvero quanto invecchia un'apparecchiatura in funzione dell'accensione/spegnimento di quest'ultima: questo può servire, perché, qualora si voglia mantenere una determinata

temperatura in un'area del datacenter, bisogna farlo senza **distruggere le CPU** (dato che, per raffreddare una zona, si potrebbe semplicemente pensare di spegnere le CPU più calde e riaccenderle poi).

Utilizzazione

È la classica metrica usata per misurare **CPU** e **RAM** e, di solito, è il rapporto fra numero di unità di tempo di risorsa utilizzata rispetto ad un determinato intervallo.

Oltre all'utilizzazione, vi è anche il **CPU load**, ovvero la dimensione **media** della *runqueue* (non in altre code! Non in *ioqueue* o altre code!) del sistema operativo (quanti processi *ready* possono essere selezionati dallo scheduler). In generale, quando il CPU load eccede il numero di core, si è in una situazione di potenziale sovraccarico.

Dal punto di vista della teoria delle code abbiamo che l'utilizzazione è la probabilità che la CPU sia occupata, quindi λ , mentre il CPU load è μ , ovvero il tasso di servizio.

Pertanto l'utilizzazione è calcolabile, come al solito, come:

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

Un'altra grandezza misurabile è la **network utilization**, ovvero quanto tempo la rete è occupato, rispetto ad un intervallo di tempo. Può essere misurata in base alla **banda del canale** disponibile all'istante oppure rispetto al **current throughput**. La banda utilizzata, però, non è mai al 100%, per via della natura algoritmica sulla quale sono costruite le reti (CSMA/CD): oltre ad un certo tasso di utilizzazione (~80%), salire di utilizzazione provoca semplicemente **più collisioni**, decrementando vertiginosamente la banda offerta.

Modelli di performance

Abbiamo parlato di **cosa** si può modellare, ora parliamo di modelli che si utilizzano nella realtà per rendere misurabili le metriche di cui abbiamo parlato.

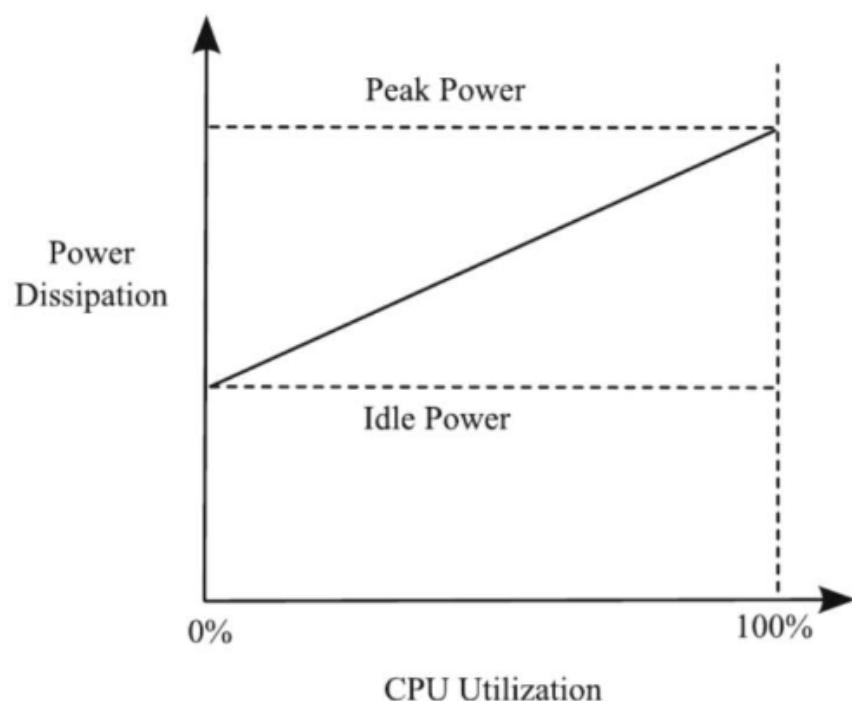
Modelli energetici

L'obiettivo è quello di minimizzare **il consumo energetico** (e di conseguenza anche quella necessaria per **raffreddare** l'apparecchiatura!).

Le parti del datacenter più interessate in questi modelli sono i **server** e le loro **CPU**.

Modello lineare

Il classico modello è quello lineare, nel quale si vede la dissipazione di calore come linearmente correlato all'utilizzazione della CPU.



DVFS (Dynamic Voltage Frequency Scaling)

Concentradosi sul consumo energetico della CPU, è possibile considerare la caratteristica dei processori di **dynamic voltage frequency scaling**: il processore cambia la frequenza del clock (e quindi il suo consumo) in base ai requisiti di **lavoro** richiesti! ⇒ è inutile avere cicli “vuoti” a 3GHz!

⇒ I cambi di stato all'interno della CPU (in termini di correnti) consumano in **funzione della frequenza a cui avvengono** (frequenza alta = più corrente ⇒ più consumo!).

I modelli di riferimento esprimono metriche di consumo energetico in funzione del **cubo** della frequenza!

Altri modelli

- gli apparati di rete “dedicati” hanno costi sostanzialmente **fissi** in termine di consumo energetico: lo stesso switch a 10Mbps o a 1000Mbps consuma sostanzialmente lo stesso quantitativo di energia. Discorso **diversissimo** è quello che si fa per reti **software defined**, dove il traffico incide **tantissimo** sul consumo energetico.
- per i tempi di risposta vi sono varie soluzioni:
 - se si hanno metriche espresse in “valori medi” allora si può fare ricorso alla teorie delle code. Si possono aggregare più sistemi di questo tipo mediante tecniche già viste (più stadi all’interno del sistema con, eventualmente, più processi stocastici);
 - in altri casi, si possono utilizzare altri modelli con profondità di dettaglio molto maggiore;

Availability

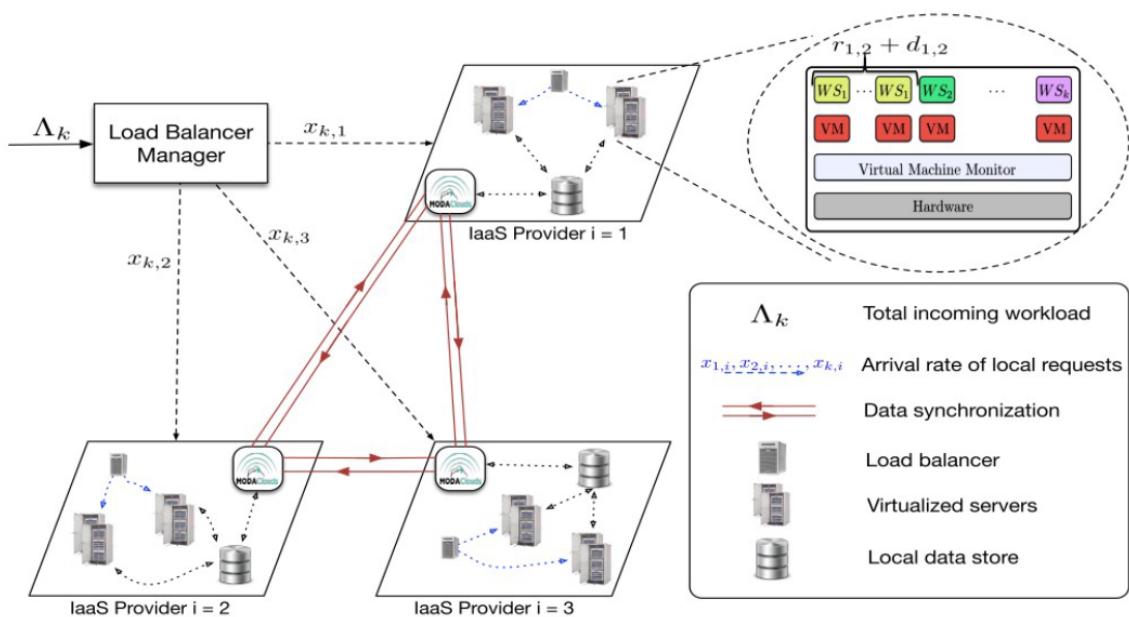
Abbiamo già visto come misurare l’availability di sistemi in serie e in parallelo.

Più datacenter

Il problema di gestione di più datacenter si scandisce su due principali dimensioni:

- come si distribuisce il carico sulla serie di datacenter;
- come il singolo datacenter reagisce al carico.

- Multi DC decisions
 - Top layer
- Workload distribution across data centers
- Coarse-grained time scale
 - Use workload prediction
- Multiple objectives
 - Match request locality
 - Cope with different VM costs in DCs
- Single DC decisions
 - Lower layer
- Data center-level scaling
- Fine-grained time scale
 - Cope with prediction errors
 - Sudden bursts of traffic
- Allocation of VMs
- Manage data synchronization over DCs



Modelli economici

La modellazione economica può essere fatta rispetto a tantissimi scenari diversi:

- se **vendo** servizi dal mio datacenter (cloud provider) \Rightarrow massimizzare la revenue che deriva da tale vendita;
 - Some choices for a cloud provider
 - Given a **high** incoming load
 - Can I **borrow** capacity from another provider
 - What kind of VM can I use (Spot, On demand)?
 - Given **underused** server capacity
 - Can I **sell** computing power with Spot VMs?
 - What **cost** can be worth?
 - Is it better to **turn off** servers?
 - General question (typical game theory problem)
 - How to define **pricing models** of VM types to maximize **revenue**?
- se sono un **utente** di un cloud provider \Rightarrow voglio spendere quanto meno possibile per utilizzare i servizi cloud;
 - Some examples
 - Azure, Amazon, ...
 - Terms may change
 - Cloud services can have different characteristics
 - **Dedicated** VM instances
 - Low cost
 - Long-term commitment (pay in advance for months or years)
 - Always available
 - **On-demand** VM instances
 - High cost
 - Short term commitment (pay per hour)
 - Always available
 - **Spot** VM instances
 - Variable cost (auction model)
 - Short term commitment (pay per hour)
 - Can be unavailable (spare capacity)

- ...

Modellazione dei problemi

In questa sezione si vedranno modelli matematici per problemi **reali** all'interno di un datacenter, con tanto di funzione obiettivo, vincoli, ...

VM placement problem

Il classico problema di un cloud provider è l'allocazione delle macchine virtuali sui server fisici.

- Classical IaaS provider problem
- Given
 - A set of **new VMs** to allocate (with resource requirements)
 - A **residual capacity** of the infrastructure (given current allocation of VMs)
 - A set of **SLA** to satisfy (i.e., avoid overload)
- Decision variables
 - **Where** (on which server) to allocate each new VM?
 - Do we need to power-up **new servers**?

Il modello utilizzato, in prima battuta, generalmente, è quello del **bin packing**: i server sono i **bin** con una data capacità e le VM sono gli elementi da mettere dentro questi bin.
Il bin packing può essere:

- monodimensionale: si considera solo una grandezza per modellare la capacità residua (i.e. CPU utilization);
- multidimensionale: si considerano più grandezze per modellare la capacità residua, come CPU e RAM assieme.

Inoltre bisogna fare l'analisi in base all'utilizzazione **effettiva** delle risorse garantite, oppure in base agli hard limit imposti. L'analisi di questi due scenari varia sensibilmente,

pertanto è necessario fare analisi su più istanti temporali, in base all'utilizzazione delle risorse che vengono date alle macchine virtuali (quale sarà l'utilizzo fra 2 minuti? fra 2 ore?).

Il problema del bin packing è che **esplode** con il numero di VM \Rightarrow Generalmente si **categorizzano** le macchine in base a determinate **classi** di risorse che vengono utilizzate principalmente (RAM-optimized, CPU-optimized, ...). In questo modo si riducono drasticamente le dimensioni del problema, sacrificando un po' la precisione, ma diminuendo allo stesso modo il tempo di esecuzione del calcolo della soluzione del problema, rendendolo eseguibile in tempi sensati per un datacenter che deve fare predizione sul traffico.

VM migration problem

Altro problema è quello della **migrazione** delle VM: si identificano nodi **sottoutilizzati** ($\leq 20\%$) e per quei server si smistano le macchine virtuali sugli altri server. Questi nodi, poi, vengono **spenti**, per risparmiare energia. \Rightarrow **Server consolidation**

Si modella il tempo in **time slot**, e si decide, in questi time slot, come e dove piazzare le macchine virtuali che migrano dai nodi sottoutilizzati.

Network optimization problem

- Focus on **data transfer**
- Several possible approaches
 - Manage **energy** consumption
 - Optimize **bandwidth** utilization (avoid bottlenecks)
 - Minimize number of **hops**
- Alternative view
 - Manage limited number of high capacity links to satisfy communication demands
 - Reconfigurable infrastructure

- Traditional problem model
- Identify groups of **well connected** VMs
 - Define **affinity** between VMs based on data exchange
 - **Clustering** problem
- **Map network-wise affinity over servers**
 - Try to place these VM close together
 - Avoid core network when possible
- Other constraints
 - **Resource** on servers (no overload)
 - **Availability** concerns

⇒ I problemi di network optimization sono generalmente di **clustering**: si raggruppano le macchine virtuali per rendere la rete quanto più compatta possibile, in modo da minimizzare determinate metriche.

Orchestrazione di microservizi

⇒ Kubernetes/OpenStack...

- Cloud user point of view
- Given:
 - A set of micro-services
 - A workload for each micro-service
 - SLA requirements (response time, availability, minimum resources)
- Knowing that a set of providers can
 - Offer different pricing models, can match or not the requirements, ...
 - The same service can be deployed with different options (different VM size, different payment method, ...)

⇒ Poiché l'applicazione viene generalmente espressa tramite un grafo di microservizi che comunicano tra di loro, è importante rappresentare correttamente le dipendenze tra i microservizi.

Strategie di management

- Some strategies to manage cloud data centers
- Derived from industrial practice
 - **Utilization**-based (Amazon, Azure, Google, ...)
- Based on research
 - **Stochastic migration** (**EcoCloud**)
 - **MBFD**
 - **Hierarchical**
 - **VMPlanner**
 - **Adaptive infrastructure**
 - **Energy-Network-Migration Aware** (**JCDME**)

Utilization based strategies

Generalmente, questa strategia è chiamata “regola dell’80-20”: quando l’utilizzazione raggiunge l’80% ⇒ offload del carico, mentre se arriva al 20% ⇒ server consolidation.

In questo modo si evitano server **sovraffollati**, ma si evita allo stesso modo il **“sottocaricamento”** dei server stessi, ottimizzando l’utilizzo delle risorse.

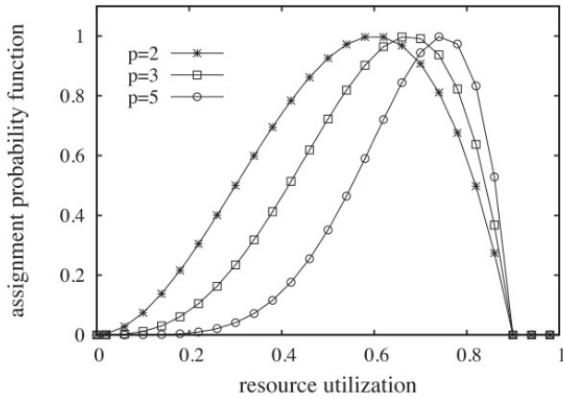
⇒ Generalmente vengono utilizzate per prendere decisioni di *scaling* di un sistema (up o down).

Dal punto di vista della predizione, sono spesso utilizzate tecniche di **regressione**, per modellare l’andamento dell’utilizzazione in ottiche di predizione.

- Why prediction is not widely used in industry
 - Not every workload is **predictable**
 - No predictor guarantee good accuracy for every case
 - **Better to react slowly than to react badly**
 - **Open problem**

EcoCloud

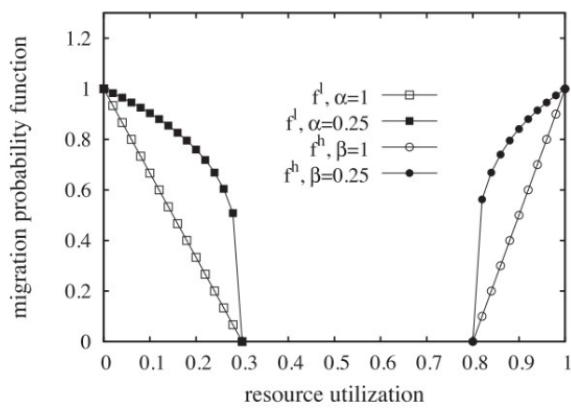
È possibile modellare problemi di migrazione delle VM mediante processi stocastici. L'idea di base di EcoCloud è quella di decidere, mediante un processo stocastico **locale**, se un server deve **scaricarsi** o se deve **accettare nuove macchine**. Il risultato di queste decisioni **locali**, poi, avrà un effetto **globale** per l'intero datacenter \Rightarrow approccio completamente distribuito.



Questa curva esprime la **probabilità di assegnamento di una VM ad un server**.

Notiamo che, in base alla utilizzazione, un server è più o meno papabile come destinazione di una macchina virtuale: se il server è molto carico, la probabilità di accettare macchine diventa 0 (oltre il 90%), ma anche se è **moltissimo scarico**, perché probabilmente sto cercando di spegnere il server. La probabilità cresce nella zona 40-70%.

$$f(x, p, T) = \frac{1}{M_p} x^p (T - x) \quad 0 \leq x \leq T,$$



Quest'altra curva esprime il concetto **duale**: la probabilità di “scarico” di un server.

$$f_{migrate}^l = (1 - x/T_l)^\alpha$$

$$f_{migrate}^h = \left(1 + \frac{x - 1}{1 - T_h}\right)^\beta.$$

Questo modello converge in poco tempo in una situazione in cui il livello di carico è ben bilanciato e il numero di macchine virtuali è vicino a quello ottimo. La cosa forte di questo approccio è che non vi è alcun modello da risolvere \Rightarrow molto semplice da implementare!

MBFD - Modified Best Fit Decreasing

Questo algoritmo prevede, a differenza del classico **best-fit decreasing**, macchine virtuali **eterogenee** (i.e. bin con diversa capacità).

Lo scaling avviene in base all'utilizzo

energetico (si minimizza il consumo energetico), modellato **linearmente**.

Placement of VMs

- Energy consumption minimization
- Linear energy model

$$P(u) = k \cdot P_{\max} + (1 - k) \cdot P_{\max} \cdot u,$$

MBFD algorithm

- **Admission** of new requests (new VMs)
- Add VMs from most CPU-intensive
- Minimize power increase
- **Heterogeneous servers**

Vari scenari:

- minimum potential growth: si usano macchine virtuali “grosse” per minimizzare il numero di migrazioni;

Algorithm 1: Modified Best Fit Decreasing (MBFD)

```

1 Input: hostList, vmList Output: allocation of VMs
2 vmList.sortDecreasingUtilization()
3 foreach vm in vmList do
4   minPower  $\leftarrow$  MAX
5   allocatedHost  $\leftarrow$  NULL
6   foreach host in hostList do
7     if host has enough resource for vm then
8       power  $\leftarrow$  estimatePower(host, vm)
9       if power < minPower then
10         allocatedHost  $\leftarrow$  host
11         minPower  $\leftarrow$  power
12   if allocatedHost  $\neq$  NULL then
13     | allocate vm to allocatedHost
14 return allocation

```

- maximum potential growth: massimizzare il load, per evitare macchine sottoutilizzate;

- **Consolidation** of VMs
 - For overloaded servers
 - **Selection** of VM to migrate
 - **Minimize migrations**
 - Avoid power on/off
- Mathematical model

$$R = \begin{cases} \left\{ S \mid S \in \mathcal{P}(V_j), u_j - \sum_{v \in S} u_a(v) < T_u, \right. \\ \quad \quad \quad \left. |S| \rightarrow \min \right\}, & \text{if } u_j > T_u; \\ V_j, & \text{if } u_j < T_l; \\ \emptyset, & \text{otherwise} \end{cases}$$

- Other definitions to select VM to migrate
- **Highest Potential Growth (HPG)**
 - Select VMs that consume less resources
 - Remaining load as close as possible to threshold
- **Random choice (RC)**
 - Random selection

Algorithm 2: Minimization of Migrations (MM)

```

1 Input: hostList Output: migrationList
2 foreach h in hostList do
3   vmList  $\leftarrow$  h.getVmList()
4   vmList.sortDecreasingUtilization()
5   hUtil  $\leftarrow$  h.getUtil()
6   bestFitUtil  $\leftarrow$  MAX
7   while hUtil  $>$  THRESH_UP do
8     foreach vm in vmList do
9       if vm.getUtil()  $>$  hUtil  $-$  THRESH_UP then
10         t  $\leftarrow$  vm.getUtil()  $-$  hUtil  $+$  THRESH_UP
11         if t  $<$  bestFitUtil then
12           bestFitUtil  $\leftarrow$  t
13           bestFitVm  $\leftarrow$  vm
14         else
15           if bestFitUtil  $=$  MAX then
16             bestFitVm  $\leftarrow$  vm
17             break
18           hUtil  $\leftarrow$  hUtil  $-$  bestFitVm.getUtil()
19           migrationList.add(bestFitVm)
20           vmList.remove(bestFitVm)
21         if hUtil  $<$  THRESH_LOW then
22           migrationList.add(h.getVmList())
23           vmList.remove(h.getVmList())
24 return migrationList

```

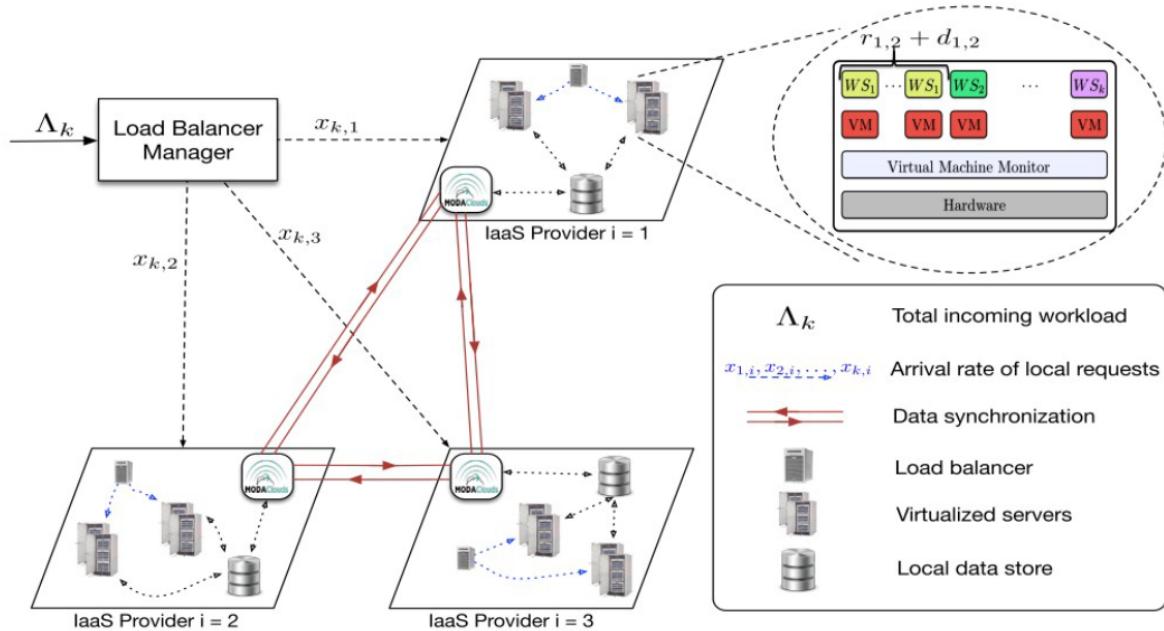
$$R = \begin{cases} \left\{ S \mid S \in \mathcal{P}(V_j), u_j - \sum_{v \in S} u_a(v) < T_u, \right. \\ \quad \quad \quad \left. \sum_{v \in S} \frac{u_a(v)}{u_r(v)} \rightarrow \min \right\}, & \text{if } u_j > T_u; \\ V_j, & \text{if } u_j < T_l; \\ \emptyset, & \text{otherwise} \end{cases}$$

$$R = \begin{cases} \left\{ S \mid S \in \mathcal{P}(V_j), u_j - \sum_{v \in S} u_a(v) < T_u, \right. \\ \quad \quad \quad X \stackrel{d}{=} U(0, |\mathcal{P}(V_j)| - 1) \left. \right\}, & \text{if } u_j > T_u; \\ V_j, & \text{if } u_j < T_l; \\ \emptyset, & \text{otherwise} \end{cases}$$

Hierarchical

L'idea è quella di avere un modello con due sottoproblemi:

- bilanciamento di carico **fra datacenter**;
- gestione del carico **nel datacenter**.



- Long term problem model
- Multiple **applications**
 - Classes of workload
- Different types of VM
 - **On demand**
 - **Reserved**
- Different cost for VM types

Global parameters	
\mathcal{I}	Set of IaaS providers
C_i	VMs instance capacity at provider i
δ_i	Time unit cost (measured in dollars) for <i>on-demand</i> VMs at provider i
ρ_i	Time unit cost (measured in dollars) for <i>reserved</i> VMs of provider i
\mathcal{K}	Set of WS classes
$D_{k,i}$	Queueing delay (measured in s) for processing WS class k requests at provider i
$R_{k,i}$	Average response time (measured in s) for WS class k request at provider i
\bar{R}_k	Average response time threshold (measured in s) for WS class k request
W_i	Maximum number of <i>reserved</i> instances available at provider i
μ_k	Maximum service rate (measured in requests/s) of a capacity 1 VM for executing WS class k requests
Long Term Problem	
T_{long}	Long-term CA time horizon, measured in hours
$\hat{\Lambda}_k$	Prediction of the total exogenous arrival rate (measured in requests/sec) for WS class k for the whole Cloud system
γ_i	Minimum percentage of traffic distributed to each provider i

In questo genere di rappresentazione, il tempo di risposta viene modellato tramite reti di code:

- server \Rightarrow servitore
- delay center \Rightarrow per rappresentare network e i/o

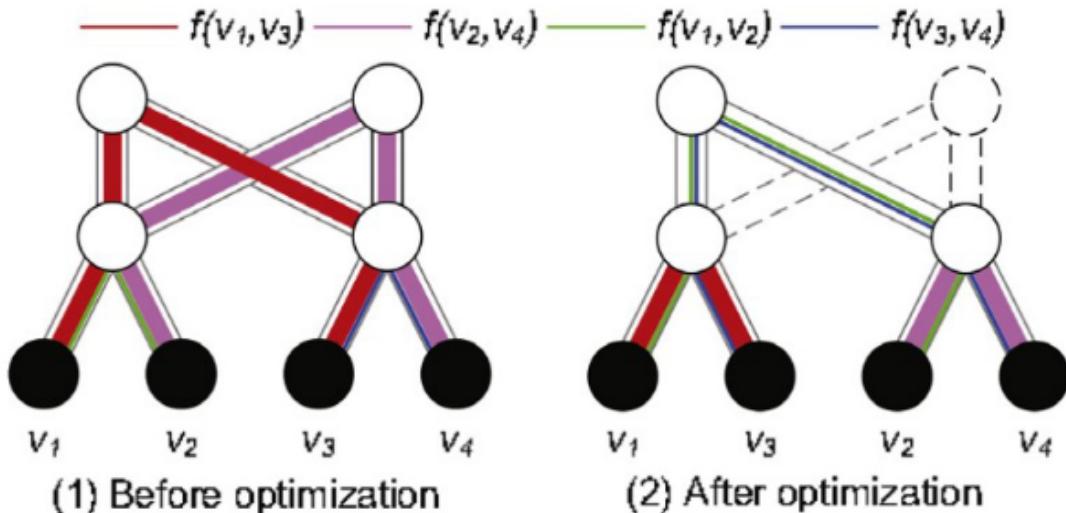
Non si utilizzano **code**, per via del fatto che non viene modellata la congestione del datacenter, ma solo il tempo di risposta.

L'algoritmo **long term** ragiona con una finestra temporale di un'ora, il tempo di riferimento per la frequenza di pagamento delle macchine virtuali (0.056€/hr, ad esempio), utilizzando un predittore del traffico in arrivo.

L'algoritmo **short term** ragiona con una finestra di pochi minuti e utilizza una tecnica chiamata **receding horizon**: i prossimi n passi nel tempo verranno ottimizzati in base alla "distanza" nel tempo che questi hanno. L'algoritmo, quindi, pianifica **costantemente** nel tempo, non si limita a predirre un workload e ad eseguire tale piano di predizione, ma continua ad affinarlo nei passi successivi, anche in quelli già "predetti".

VM planner

Questo algoritmo si focalizza molto sull'utilizzo della rete da parte delle macchine virtuali: si piazzano le macchine virtuali in modo da minimizzare i link utilizzati:



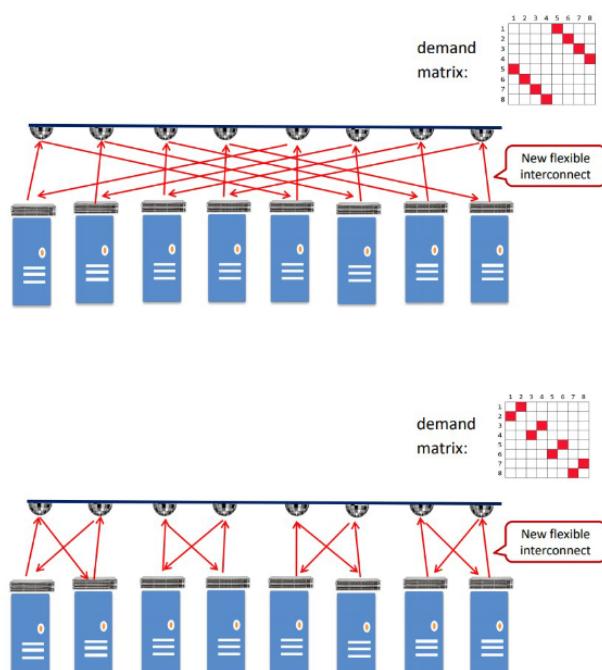
- Le macchine virtuali vengono raggruppate in base a diverse classi di traffico \Rightarrow Balanced Minimum K-cut problems;
- I gruppi di traffico vengono mappati sui vari **rack** all'interno del datacenter \Rightarrow Quadratic assignement Problem;
- Una volta trovato il posizionamento, viene deciso come viene mappato il flusso di traffico sulla geografia ottenuta \Rightarrow Multi-commodity flow Problem.

Adaptive infrastructure

Un approccio completamente diverso viene attuato dal concetto di *adaptive infrastructure*: in un contesto cloud vi sono due classi di traffico principali, **gli elephant e i mice**, con diversi requisiti di latenza (elephant \Rightarrow latenza alta accettabile, mice \Rightarrow latenza alta assolutamente dannosa).

A seconda dei flussi di traffico che avvengono tra le macchine, la rete può essere riconfigurata on demand, in base ad una matrice che viene specificata da un “controller” del traffico:

- Reconfigurable optical switches
- Problems:
 - How to **map data flows** on (limited number of) high performance **links**
 - How to build the binary demand matrix
 - Must **predict** data flows intensity and duration
 - Must interact with **existing infrastructure**



La rete diventa quindi **bipartita**:

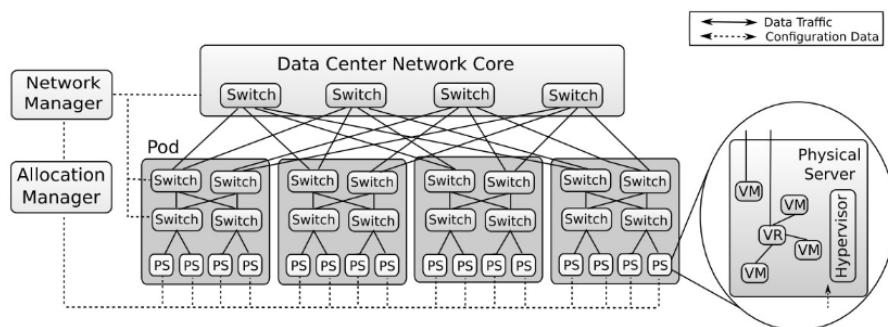
- si utilizza una rete per i flussi di traffico molto grandi e un'altra per i flussi **piccoli!**

- commutazione di circuito \Rightarrow flussi grandi
- commutazione di pacchetto \Rightarrow flussi piccoli

JCDME

Problema di minimizzazione dell'energia in un datacenter che vede come metrica cardine l'amount di migrazioni che avvengono in un determinato intervallo di time slot.

- Reference model:
 - Software-Defined Data Center
 - Presence of VMs and **virtual routers** (VRs)
- **Discrete** time model (use of time-slots for decisions)



- Objective function
 - Multiple components
 - Same metric: Energy
- Energy for computation
 - Based on linear model
- Energy for data transfer
 - Matrix for energy consumption
 - Amount of data exchanged
- Energy for migration
 - Computational overhead
 - Cost for data migration

$$\mathcal{E}_{C_i}(t) = O_i(t)\tau P_i^{MAX} \left[K_{C_i} + (1 - K_{C_i}) \frac{\sum_{j \in \mathcal{N}} x_{i,j}(t)c_j(t)}{c_i^{MAX}} \right]$$

$$\begin{aligned} \mathcal{E}_D(t) = & \sum_{i \in \mathcal{M}} O_i(t)\tau P_i^{NET} + \sum_{j_1 \in \mathcal{N}, j_2 \in \mathcal{N}} \sum_{i_1 \in \mathcal{M}} \\ & \times \sum_{i_2 \in \mathcal{M}} x_{i_1,j_1}(t)x_{i_2,j_2}(t)d_{j_1,j_2}(t)\tau \mathcal{E}_{d_{i_1,i_2}} [J] \end{aligned}$$

$$\begin{aligned} \mathcal{E}_{M_j}(t) = & \sum_{i_1 \in \mathcal{M}} \sum_{i_2 \in \mathcal{M}} g_{i_1,j}^-(t) \cdot g_{i_2,j}^+(t) \\ & \times \left[m_j(t) \cdot \mathcal{E}_{d_{i_1,i_2}} \right. \\ & \left. + (1 - K_{C_{i_1}}) \cdot P_{i_1}^{MAX} \cdot K_{M_{i_1}} \cdot \tau \right. \\ & \left. + (1 - K_{C_{i_2}}) \cdot P_{i_2}^{MAX} \cdot K_{M_{i_2}} \cdot \tau \right] \end{aligned}$$

- Basic optimization model
 - Objective function
 - Use of weight factor $\gamma(t)$
- Constraint for computational overload
- Constraints for max bandwidth
- Constraint for memory
- Decision variables
 - Placement of VMs
 - Migration
 - State of server (on/off)
- Model based on time slots

$$\min \left[\sum_{i \in \mathcal{M}} \mathcal{E}_{C_i}(t) + \mathcal{E}_D(t) + \gamma(t) \sum_{j \in \mathcal{N}} \mathcal{E}_{M_j}(t) \right]$$

subject to:
Eq. (3)-(5)

$$\begin{aligned} \sum_{j \in \mathcal{N}} x_{i,j}(t) \cdot c_j(t) &\leq c_i^{MAX} \cdot O_i(t) \quad \forall i \in \mathcal{M}, \\ \sum_{j_1 \in \mathcal{N}} \sum_{j_2 \in \mathcal{N}} [x_{i,j_1}(t) + x_{i,j_2}(t) - 2x_{i,j_1}(t)x_{i,j_2}(t)]d_{j_1,j_2}(t) &\leq d_i^{MAX} \cdot O_i(t) \quad \forall i \in \mathcal{M}, \\ \sum_{j \in \mathcal{N}} x_{i,j}(t) \cdot m_j(t) &\leq m_i^{MAX} \cdot O_i(t) \quad \forall i \in \mathcal{M}, \\ \sum_{i \in \mathcal{M}} x_{i,j}(t) = 1 &\quad \forall j \in \mathcal{N}, \\ \sum_{i \in \mathcal{M}} g_{i,j}^+(t) = \sum_{i \in \mathcal{M}} g_{i,j}^-(t) &\leq 1 \quad \forall j \in \mathcal{N}, \\ g_{i,j}^-(t) \leq x_{i,j}(t-1) &\quad \forall j \in \mathcal{N}, i \in \mathcal{M}, \\ g_{i,j}^+(t) \leq x_{i,j}(t) &\quad \forall j \in \mathcal{N}, i \in \mathcal{M}, \\ g_{i,j}^-(t) + g_{i,j}^+(t) \leq 1 &\quad \forall j \in \mathcal{N}, i \in \mathcal{M}, \\ x_{i,j}(t) = x_{i,j}(t-1) - g_{i,j}^-(t) + g_{i,j}^+(t) &\quad \forall j \in \mathcal{N}, i \in \mathcal{M}, \end{aligned}$$