

Sistemi di Applicazioni Cloud

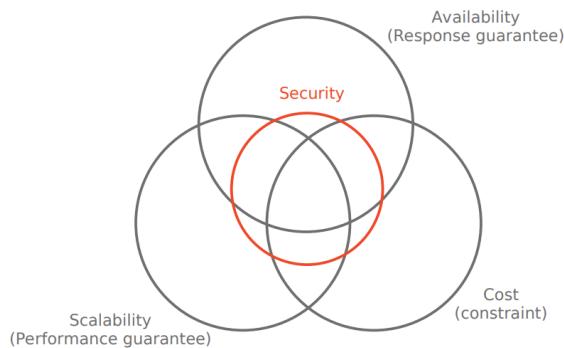
February 15, 2024

1 Principi del Cloud Computing

1.1 Servizi Internet

Le caratteristiche richieste dei servizi Internet odierni sono le seguenti:

- Robustezza e sicurezza: garantiscono il continuo funzionamento e sono resistenti a intrusioni
- Scalabilità: indipendentemente dal carico di lavoro (anche a fronte di attacchi DDoS)
- Usabilità: sono di facile utilizzo
- Costo: (apparentemente) gratuiti
- Infrastruttura: si basano su 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. Ad 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, fruendo di milioni di servizi, messi a disposizione da milioni di providers, costruiti su milioni di server, contenenti zettabyte di dati, connessi tramite petabyte di reti.

Un altro vincolo importante è la costante attenzione per i reclami dei clienti che possono portare alla perdita di fiducia. 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**.

1.1.1 Definizione di servizi

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

Un servizio viene detto funzionante fintanto che continua a fornire delle risposte.

Un concetto importante introdotto nella definizione è la continuità con cui deve essere erogato il servizio. Possiamo quantificarla in due modi:

- Tramite contratto

- SLA: Service Level Agreement in cui ci si accorda sui pagamenti e le penali in caso di mancato adempimento ai requisiti stabiliti
- Reputazione: La mancata continuità nell'erogazione del servizio porta alla perdita di clienti e di fiducia nei confronti del provider

Il compito di un ingegnere cloud è quello di implementare un sistema scalabile che garantisca una certa continuità, attraverso i seguenti passi:

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

Per quanto riguarda invece le linee guida generali a cui attenersi sempre, cosiddette *d'oro*, individuiamo le seguenti:

- Evitare:
 - Colli di bottiglia: singoli punti nell'infrastruttura che limitano la capacità totale dell'applicazione
 - Single Point of Failure: parte del sistema che, qualora fallisse, influenza tutto l'apparato arrestandolo. Benché non sia facile rimuoverli in certi casi, occorre almeno identificarli. Esempi:
 - * Elettricità e continuità di corrente in un data center.

1.2 Cloud Computing

Grid Computing L'architettura **grid computing** unisce diverse unità geograficamente distanti per determinati intervalli di tempo con uno scopo specifico: 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 identici o meno. 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.

L'idea principale è quella di un grande supercomputer distribuito in cui molte organizzazioni contribuiscono mettendo a disposizione le proprie risorse. L'unità di lavoro è il *gridlet*.

Globus Toolkit E' un meccanismo 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.

1.3 Gestione delle Risorse

E' importante cercare di quantificare l'utilizzo di un server per potere soddisfare il carico richiesto. Quest'ultimo, infatti, può variare sensibilmente nel corso di una giornata e non vogliamo incorrere né in perdite di introiti derivanti dalla **mancata erogazione del servizio** né nella **sottoutilizzazione** delle stesse macchine. Un'intuizione vincente fu quella di Bezos che inizio a vendere potenza di calcolo inutilizzata ad altre aziende.

Le politiche per gestire la potenza di calcolo in un'azienda sono diverse, si può scegliere di:

- Soddisfare un workload **minimo**: riduciamo i costi ma siamo poco appetibili sul mercato
- Soddisfare l'utilizzo **medio**: non riusciamo a soddisfare il picco della domanda e potrei comunque avere potenza computazione inutilizzata
- Soddisfare il **picco** della domanda: riesco a guadagnare ottima reputazione tra i clienti, aumentando così gli incassi ma ho molta potenza computazionale inutilizzata

1.4 Alte performance e realizzabilità

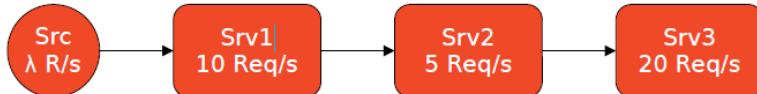
Alte performance si possono ottenere solo:

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

Contariamente a quanto si può pensare sulla base delle regole auree dette prima, evitare SPoF (Single Points of Failure) non garantisce alte performance ma realizzabilità.

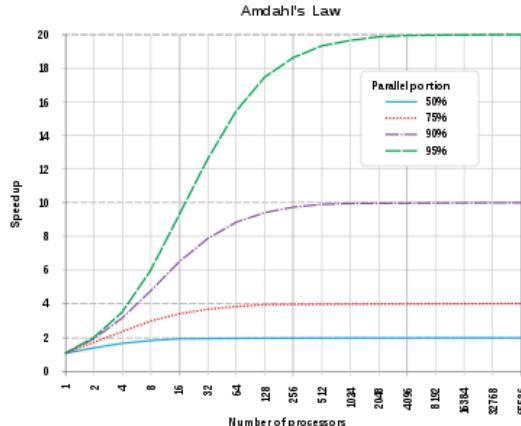
L'obiettivo consiste nel fornire un determinato throughput (rateo di risposte) e un determinato tempo di risposta. I due obiettivi possono essere legati, dove la richiesta di uno prevede la necessità dell'altro.

Osservando una catena di servizi ci basta capire quale unità di elaborazione ha il throughput minore rispetto agli altri per capire quale sarà il collo di bottiglia nella nostra rete.



Per ottenere 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 allinearli al livello degli altri. 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.

1.5 Legge di Amdahl



- T_p : Tempo richiesto per la parte di elaborazione *parallela*
- T_s : Tempo richiesto 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

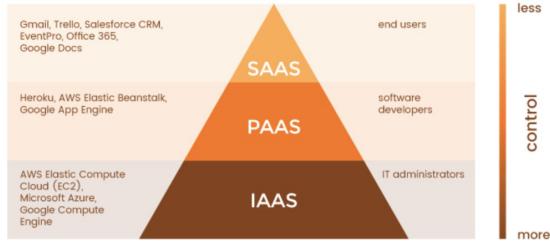
$$S = \frac{T_s + T_p}{T_s + \frac{T_p}{n}}$$

Per $n \rightarrow \infty$:

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

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

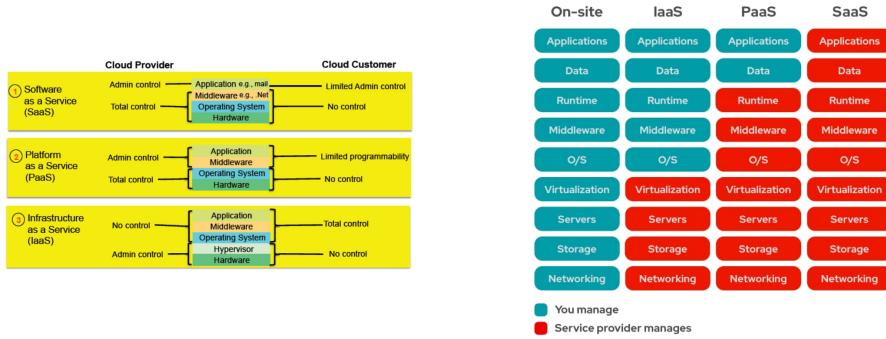
1.6 Paradigmi del Cloud Computing



Le principali caratteristiche del cloud computing sono:

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

1.6.1 Paradigma Service

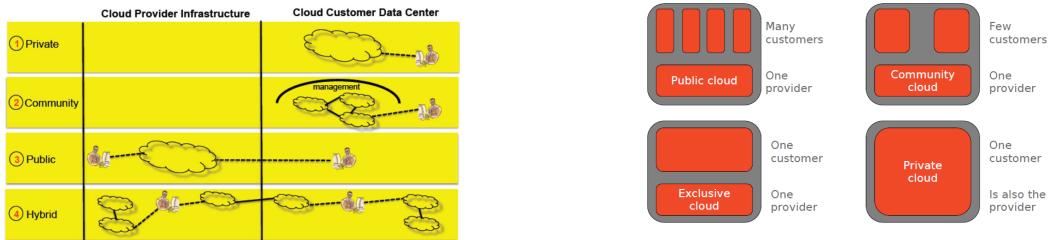


In base al paradigma scelto sono disponibili differenti livelli di controllo:

- Software: è la forma più completa di servizi di cloud computing, che fornisce un'intera applicazione gestita dal provider, tramite un browser web. Gli aggiornamenti del software, le correzioni dei bug e la manutenzione generale del software sono gestiti dal provider e l'utente si connette all'applicazione tramite una dashboard o un'API. Non è necessaria l'installazione del software sui singoli computer e l'accesso di gruppo al programma è più fluido e affidabile.
- Platform: Il provider ospita l'hardware e il software sulla propria infrastruttura e fornisce questa piattaforma all'utente come soluzione integrata, stack di soluzioni o servizio attraverso una connessione Internet. All'ambiente per costruire e distribuire ci pensa il provider.
- Infrastructure: Si tratta di un servizio pay-as-you-go in cui una terza parte fornisce all'utente servizi di infrastruttura, come lo storage e la virtualizzazione, in base alle sue esigenze, tramite un cloud, attraverso Internet. L'utente è responsabile del sistema operativo e di tutti i dati, le applicazioni, il middleware e i runtime, mentre il provider gli fornisce l'accesso e la gestione della rete, dei server, della virtualizzazione e dello storage di cui ha bisogno.

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.

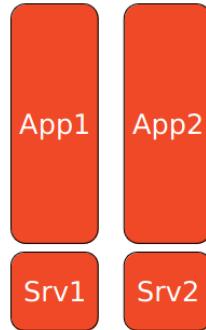
1.6.2 Paradigma del deployment



- **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:** 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).

1.6.3 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 i vari server in un singolo e creare delle macchine virtuali su cui far girare le applicazioni, tuttavia i server rimangono spesso sotto utilizzati.



1.6.4 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

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 impiegati con il minimo costo di gestione e l'interazione del servizio provider.

1.7 Caratteristiche del Cloud

Per capire il passaggio a infrastrutture Cloud nell'utilizzo odierno dell'informatica è importante capire quali sono le motivazioni di ciò e i requisiti che il Cloud ha soddisfatto.

1.7.1 Requisiti

E' assolutamente necessario che il software implementato funzioni bene indipendentemente dal **numero di 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.

1.7.2 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 **consistenza** totale, avendo quindi in ogni punto del software dati aggiornati e pronti all'elaborazione. Lo stesso discorso vale per la comunicazione che non può essere totalizzante e asincrona in ogni momento. Ecco perché per il cloud è opportuno adottare diversi paradigmi a livello software:

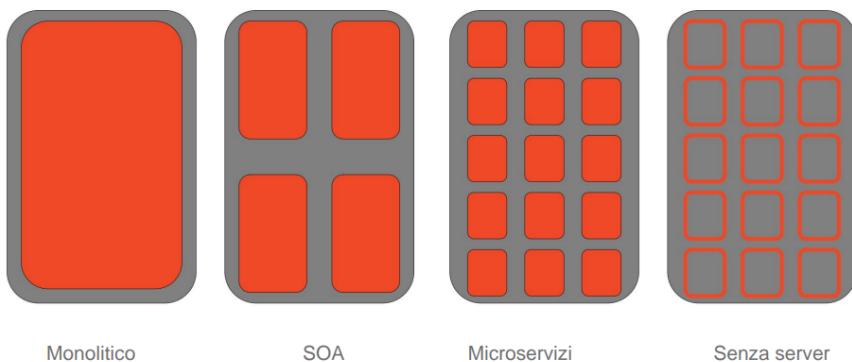
- Parallelismo su larga scala.
- Dati:
 - Lazy consistency: I dati non vengono aggiornati sempre immediatamente.
 - Distribuzione locale o geografica.
 - Caching
- Comunicazione:
 - Asincrona
 - IPC (Inter Process Communication) limitate.

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.

1.8 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 impiegare ma soprattutto da implementare
- Serverless: Architettura senza server, una sorta di microservizi 2.0

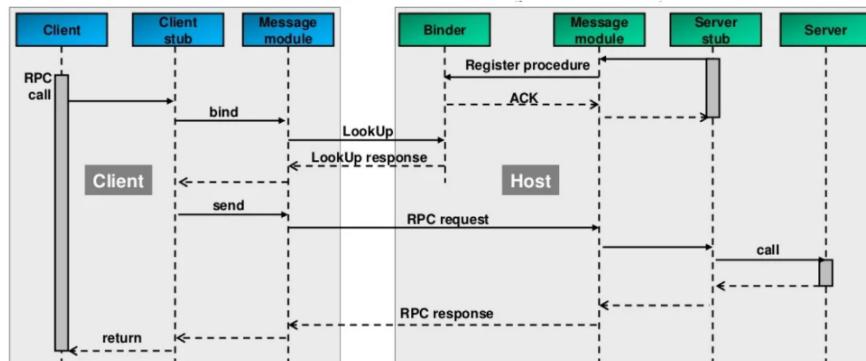
Le architetture ibride prevalgono nella maggior parte delle aziende per la difficoltà di queste a convertirsi ai nuovi paradigni.

1.8.1 Modelli pre-SOA

Prima dell'avvento della Service Oriented Architecture le possibilità erano:

- RPC: Chiamata di procedura remota. Consiste nella creazione di codice per astrarre 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:

- Il client chiama lo **stub** del client. La chiamata è una chiamata di procedura locale, con i parametri inseriti nello stack nel modo normale.
- Lo **stub** del client impacchetta i parametri in un messaggio ed effettua una chiamata di sistema per inviare il messaggio. L'impacchettamento dei parametri è chiamato **marshalling**.
- Il sistema operativo locale del client invia il messaggio dalla macchina client alla macchina server.
- Il sistema operativo locale della macchina server passa i pacchetti in arrivo allo **stub** del server.
- Lo **stub** del server scompatta i parametri dal messaggio. Lo scompattamento dei parametri è chiamato **unmarshalling**.
- Infine, lo **stub** del server chiama la procedura del server. La risposta ripercorre gli stessi passi in senso inverso.

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.

1.8.2 Modelli SOA

Architettura orientata ai servizi con approccio rivolto 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. Un esempio è 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.

1.8.3 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.



Figure 1: Soap è incapsulato in HTTP e definisce una struttura per le risposte HTTP in 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.

1.8.4 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 mo di BUS* condiviso per la connessione tra i componenti permette di interconnettere i vari servizi. 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.

1.8.5 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.

1.9 Architettura REST

1.9.1 Applicazioni Cloud

Ricordiamo che le applicazioni cloud hanno i seguenti requisiti:

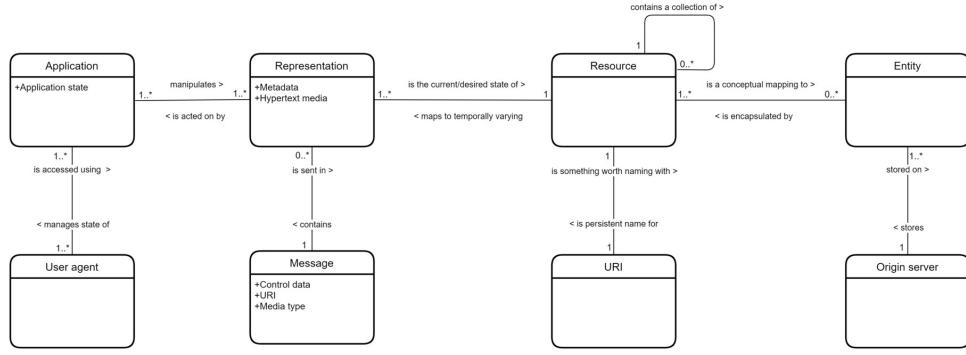
- Resilienza
- Resistenti a failures
- Elastici in modo da scalare da pochi server a decine di migliaia.
- L'architettura deve avere dei servizi *loosely-coupled* e stateless

1.9.2 REST

REpresentation State Transfer, fornisce all'applicazione un architettura senza stati, utilizzando tutti i **verbi** del protocollo HTTP. Il concetto chiave è quello di encapsulare 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 ausiliari per il funzionamento.

I client e i server sono molto leggeri dato che le risorse, identificate da URI, encapsulano 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.
- I risultati possono essere salvati in cache.
- Struttura a strati.
- Un servizio REST può interracciarci con altri servizi.
- Interfaccia uniforme: L'uso di URI (Uniform Resource Identifier), fornisce una rappresentazione delle azioni e dei messaggi autodescrittiva.



Benché non ci siano esplicativi limiti nella configurazione dei messaggi, Le azioni tipiche seguono il paradigma CRUD:

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

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

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

Ecco i vari codici di risposta delle chiamate REST:

1XX Informational	
100	Continue
101	Switching Protocols
102	Processing

2XX Success	
200	OK
201	Created
202	Accepted
203	Non-authorative Information
204	No Content
205	Reset Content
206	Partial Content
207	Multi-Status
208	Already Reported
226	IM Used

3XX Redirectional	
300	Multiple Choices
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 Client Error	
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
422	Unprocessable Entity
423	Locked
424	Failed Dependency
426	Upgrade Required
428	Precondition Required
429	Too Many Requests
431	Request Header Fields Too Large
444	Connection Closed Without Response
451	Unavailable For Legal Reasons
499	Client Closed Request

Figure 2: Information - Success - Redirectional

Figure 3: Client Error

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."
}
```

Una richiesta HTTP può essere *cached* in modo da non dover nuovamente essere richiesta al server. E', inoltre, possibile definire un TTL, *time to live*, per stabilire il tempo massimo di esistenza all'interno della cache.

Un *Etag* mantiene i metadati sulla risorsa e sull'ultima modifica ad essa fatta tramite un MD5 digest della rappresentazione della risorsa, tramite cui viene verificato se la risorsa cached è aggiornata o meno.

1.9.3 RESTful API

Rappresenta lo standard *de facto* per la costruzione di sistemi distribuiti.

I servizi REST non hanno supporto diretto per generare un client da un IDL (termine generico per indicare un linguaggio che consente a un programma di comunicare con un altro scritto in una lingua sconosciuta) 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.

Occorre quindi un description language, cioè una specifica implementazione per IDL machine-readable usata per descrivere, produrre e visualizzare RESTful web services. Un esempio è OpenAPI.

Un concetto fondamentale delle chiamate REST è il controllo dei media, ovvero del tipo di contenuto disponibile che il client può chiedere e 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 ipermediiali**". 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

1.10 Architetture guidate agli eventi (EDA)

Alternativa all'architettura REST, si basa sul concetto di evento che guida le interazioni o i risultati. Questo evento può assumere una varietà di forme e la natura di come questi eventi vengono

comunicati all'utente finale è spesso l'elemento più significativo che definisce l'implementazione dell'EDA. 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 software del ricevente. Il tempo di risposta non è deterministico proprio per la natura di questa architettura.

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, ad esempio, un sensore che rilevando dei dati inerenti a qualche monitoraggio comunica ad un intermediario, il 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, infatti, 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 è focalizzato sulla scalabilità e la tolleranza a *failure*. Tolleranza che però viene pagata in termine di costi, data la sua replicazione.

1.10.1 EDA Scalability

I punti chiave per la scalabilità delle architetture ad eventi sono la possibilità di inviare messaggi in modo **asincrono** tra le *loosely couples* che si creano (publisher-subscriber) e la **consistenza debole**, dove i dati vengono aggiornati non in maniera istantanea ma solo in determinati momenti.

- Comunicazioni asincrone: Le code 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.
- Accoppiamento libero: A causa della struttura Pub/Sub, i publisher non sanno 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 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.

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.

1.11 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. Questo non è un componente fisico vero e proprio come l'ESB. L'analogia è con i ballerini in cui ognuno sa cosa fare in base a come si muove il ballerino prima e dopo di lui. Questo modello di coordinamento è implicitamente **decentralizzato**.

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

1.11.1 Resilienza

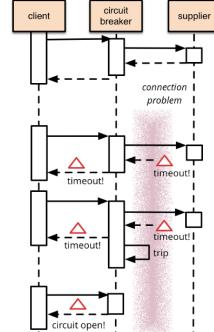
La resilienza è un obiettivo fondamentale dei sistemi distribuiti per evitare il verificarsi di problemi come errori a cascata o chiamate remote fallite, bloccate o senza risposta. Quando qualcosa va storto bisogna gestire il tutto arrecando meno danno possibile. Quando abbiamo servizi connessi tra di loro, un problema su uno di essi può, infatti, 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 I Circuit Breakers sono un modello di progettazione utilizzato nello sviluppo del software. Vengono utilizzati per rilevare i guasti e prevedono una logica per evitare che questi si ripetano costantemente.

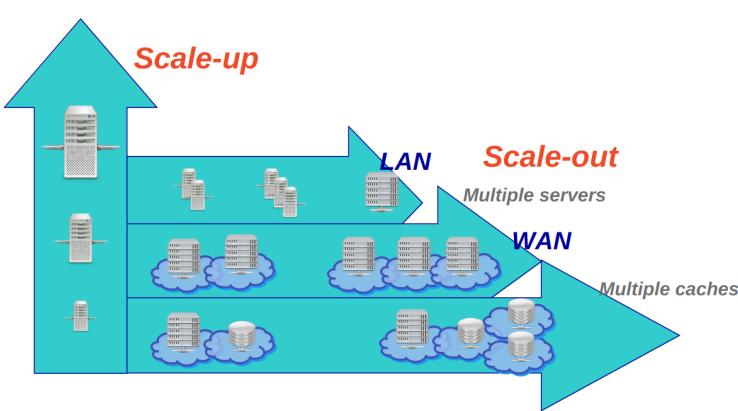
Il circuit breaker funge da intermediario per le operazioni che potrebbero fallire. Questo deve monitorare il numero di fallimenti recenti che si sono verificati e utilizzare queste informazioni per decidere se consentire all'operazione di procedere o se restituire immediatamente un'eccezione, marcando il servizio come **bad**.



2 Data Management

Data Replication La scalabilità si può ottenere tramite:

- Scale-Up: Tramite il miglioramento delle risorse hardware del server. Presenta degli ovvi limiti fisici, risolvibili solo tramite lo scale out.
- Scale-Out: Lo scaling orizzontale prende l'infrastruttura esistente e la replica per lavorare in parallelo. Questo ha l'effetto di aumentare la capacità dell'infrastruttura in modo approssimativamente lineare. Scalando orizzontalmente si ha una replicazione a livello di storage, per cui bisogna stabilire come quest'ultimo 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.

2.1 Data Synchronization

Se non abbiamo nessuna consistenza si hanno performance migliori, una migliore scalabilità e assenza di overhead dovuto alla sincronizzazione ma ovviamente non è adatto a nessuno scenario in cui vi sia replicazione. Se invece abbiamo consistenza forte gli aggiornamenti sono immediati e non c'è rischio di perdere dei dati: è un paradigma adatto a servizi che necessitano di alta disponibilità. Il compromesso che nasce tra i due è la consistenza debole.

Possiamo avere due differenti tipi di copie:

- Primarie: Copie autoritarie.
- Master: Copie in lettura/scrittura.

Tutte le altre copie che non siano primarie o master sono in sola lettura. Nel caso di consistenza **forte** tutte le copie sono primarie.

2.1.1 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.

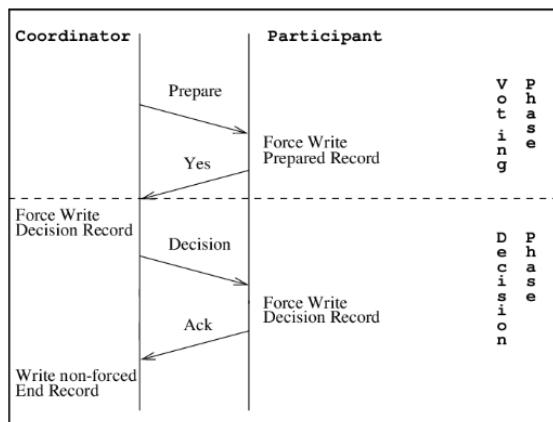
Ad esempio, se io prelevo contemporaneamente dallo stesso conto 20 euro in America e 20 euro in Europa e il mio bilancio iniziale era di 50 euro, vorrei che il bilancio finale fosse di 10 euro. Poiché

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 transazione e attende che questi rispondano con l'esito dell'operazione. Tutti i nodi, quindi, aggiornano il proprio log interno (struttura dati usata per tracciare gli update ricevuti), ma non aggiornano ancora gli indici: il cambiamento non è ancora visibile all'esterno
- 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 facendo o meno 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 basato su scambi di informazioni con i vicini. La disponibilità è dato dal fatto che ci sono almeno N nodi che replicano un 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 replicano l'elemento interessato 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.

Quando avviene una scrittura, questa può arrivare su qualsiasi nodo che a sua volta la propagherà ai suoi vicini. I vicini risponderanno con un ACK quando avranno fatto la scrittura e, una volta inviato l'ACK, questi procederanno a spargere la scrittura sui propri vicini, e così via. Colui che inizia il procedimento può decidere un **quorum** di nodi che devono aver ritornato l'ACK per stabilire che la scrittura sia andata a buon fine, generalmente $(N/2) + 1$, dove N è il numero di vicini dell'iniziatore.

Primario e secondario In questo sistema il server secondario può rispondere solo nel caso di failure del server primario, garantendo quindi availability. Alternativamente può essere utilizzato per fornire performance aggiuntive. Le operazioni di scritture devono essere inviate al server primario che procederà successivamente a sincronizzare i server secondari, con una periodicità variabile (da pochi minuti a diversi giorni). Tuttavia la copia nel server secondario viene utilizzata solo se la copia primaria non è disponibile.

2.1.2 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 prima o poi 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 riuscire a gestirne la consistenza. 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 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 nelle immediate vicinanze di 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 fornito dal server primario a quello secondario quando quest'ultimo viene interpellato da un utente, e non ne esiste una copia valida in cache. Principalmente utilizzato da ISP e proxy servers.
- Push: I contenuti che vengono richiesti con maggiore probabilità sono attivamente inviati al 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: Dal punto di vista del proxy server, quando questo riceve una richiesta per 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: Dato vecchio ma modificato recentemente.
- Static: Dato vecchio e modificato tempo fa.

Se si ha un nuovo dato è difficile da stabilire ma si presuppone sia volatile.

Il popolamento della cache è guidato dalle richieste del client.

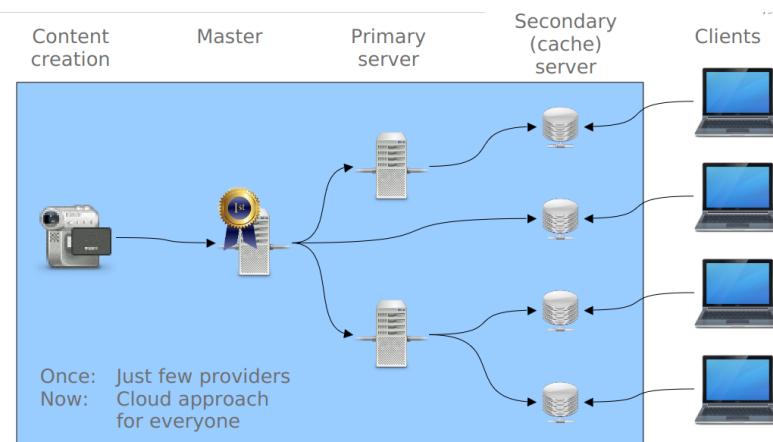
Il proxy server non è molto utilizzato in quanto non gestisce contenuti generati dinamicamente e inoltre non supporta le connessioni HTTPS *end-to-end*.

- Reverse Proxy: Un server virtuale posto di fronte al Web server in modo che possa immagazzinare risorse recenti generate dinamicamente. 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, in quanto più vicino al server primario. 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 per 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. Esistono diversi approcci ai proxy server cooperativi:

- 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*, cioè quando accediamo ad un dato per la prima volta e quindi questo non è presente in cache

- 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

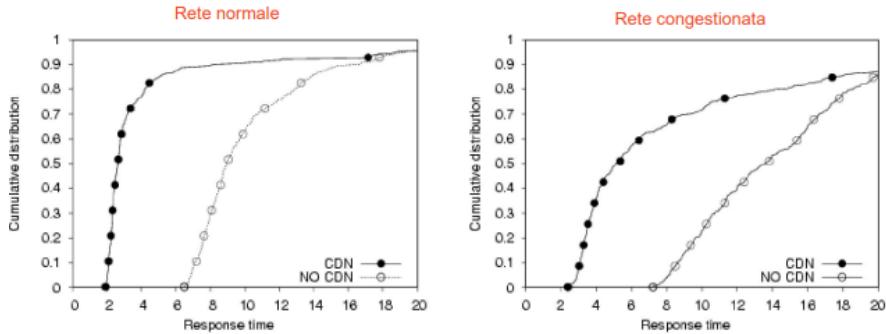
A differenza della memorizzazione cache tradizionale attraverso la cache selettiva, nelle CDN vengono selezionate solo determinate risorse rilevanti che possono interessare maggiormente rispetto ad altre. Questo approccio, ad esempio, può essere utilizzato per contenuti premium, per i servizi di streaming 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. Il *caching* avviene in maniera cooperativa e selettiva. Usano prevalentemente un paradigma di pushing più che di pulling perché sono i provider dei contenuti a decidere quali di questi vogliono vedere replicati. Proprio per questo motivo, la replicazione è molto semplice dato che è il provider a fornire la nuova versione della risorsa ed è la CDN a sapere in quali server era stata replicata 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 interessato. 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 richiede più tempo.



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 nel modo più veloce possibile. 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 sia la pagina che gli embedded object.

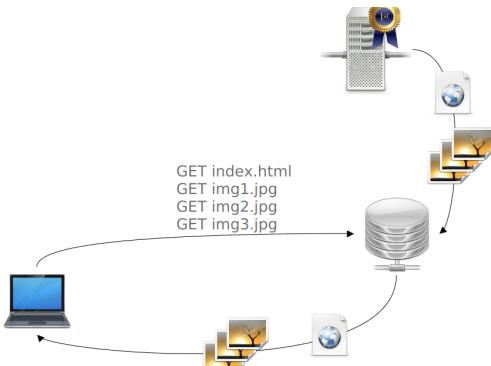


Figure 4: URL Rewriting

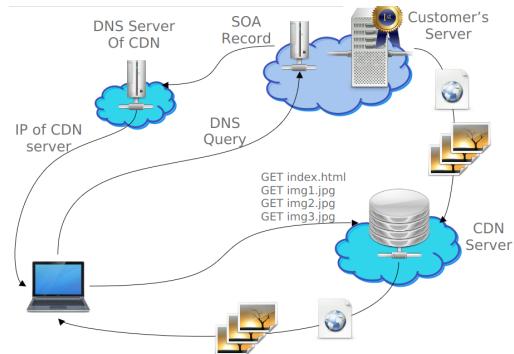
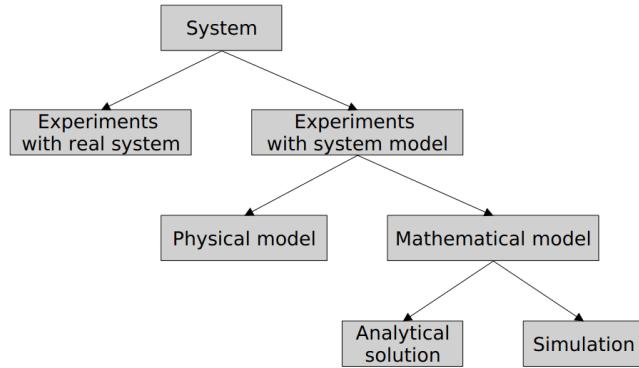


Figure 5: DNS Outsourcing

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.

3 Valutazione delle performance e simulazione

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.



- Esperimenti con sistemi reali: vengono fatti quando il sistema è molto semplice e contenuto. In questi casi si rivela più conveniente di altri approcci.

Un test su di un sistema reale viene fatto, generalmente, come validazione dei calcoli teorici fatti a priori.

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

- Functional Testing: verifica il corretto funzionamento del programma con determinati risultati attesi. Ovviamente non consente di catturare scenari particolari di malfunzionamento funzionale (race condition, ad esempio), o di workload (lentezza, memory leak etc.)
- Activity Testing: si testa il funzionamento sotto normali condizioni di lavoro (workload testing)
- Endurance Testing: si testa il sistema sotto il carico massimo che può andare a sottostare (alla kneeling zone). L'obiettivo è quello di avere dei dati per fare capacity planning in vista di carichi futuri o meno
- Stress Testing: si va oltre la kneeling zone e si verifica dove sia il suo limite
- Benchmark: si testa il sistema sotto carichi il più possibile standardizzati. Possiamo avere: **toy** benchmarks, piccoli algoritmi poco rappresentativi; **programmi reali** che possono comunque non essere disponibili per ogni piattaforma; benchmark **sintetici** basati sulle operazioni comuni nei software (ma comunque proni ad abusi); benchmark **di applicazioni reali** di problemi più rappresentativi delle operazioni aritmetiche o compilazione del kernel, in scenari completamente standardizzati.

- Esperimenti con modelli del sistema complesso:

- Modelli fisici: sono rappresentazioni “in scala” del sistema stesso. Vengono tipicamente 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 teorica i modelli matematici sono i più accurati, ma necessitano di trovare il giusto livello di dettaglio

* Soluzioni analitiche: utilizzano 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

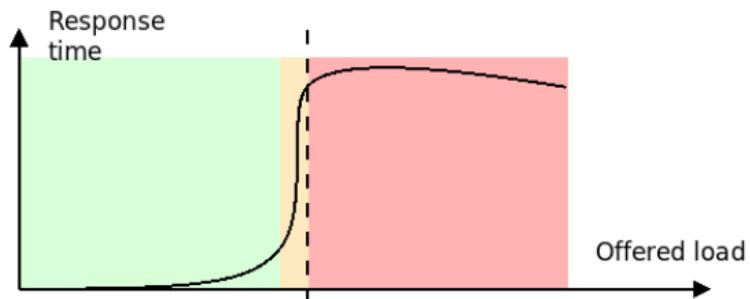
- * Simulazione: tramite software è possibile simulare il comportamento di un sistema. È la via di mezzo fra modello matematico e modello reale, quindi un buon compromesso. Inoltre, è molto più versatile del modello matematico ma ha comunque degli intervalli di approssimazione di cui bisogna tenere conto

I parametri prestazionale di maggior interesse sono:

- Tempo di Risposta: intervallo fra accettazione di una richiesta e fine della risposta
- Throughput: volume di richieste soddisfatte nell'unità di tempo
- Error Rate: frazioni di richieste non servite correttamente

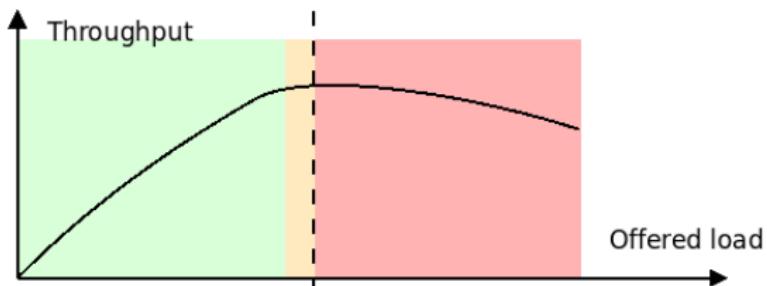
Tutti questi parametri dipendono direttamente dal carico corrente del sistema.

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



Il tempo di risposta varia in un modo preciso: rimane molto basso finché l'offered load (ovvero *richieste* *intervallo*) 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 metta più tempo a processare le richieste bensì perché il traffico in arrivo, impossibile da processare, viene accodato. Più le code sono piene, più aumenta il tempo di risposta. Il limite superiore è garantito dal *timeout* che impedisce alla curva di salire ancora.



Il throughput invece cresce linearmente fino alla **saturazione**, momento nel quale raggiunge la capacità massima. Da quel punto in poi le richieste verranno accodate e il sistema passerà più tempo a gestire i context switch (*trashing*) che ad eseguire lavoro utile. Il throughput quindi cala.



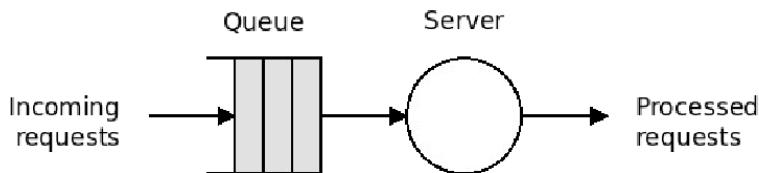
Conseguenza naturale dell'aumentare dei problemi di questo genere è l'aumento anche degli errori. L'obiettivo è quello di avere un sistema che si trovi in un intorno della fine della zona di sottoutilizzazione (quella verde): in questa zona il sistema non è né sottoutilizzato, né sovrautilizzato

ma è al massimo della propria capacità tollerabile. L'ampiezza di questo intorno della kneeing 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%?

3.0.1 Modelli Basati su Sistemi a Coda

In questi modelli si considerano i sistemi a coda (con accodamento di richieste). Non serve che il modello ritorni un risultato preciso, è sufficiente conoscere almeno l'*ordine di grandezza per sapere se è necessario scalare o meno*.

Immaginiamo un sistema composto da più parti: network, vari nodi frontend, vari nodi backend etc., ciascuna di queste parti è caratterizzata da determinati tempi di risposta, come quello della CPU, quello dello storage etc. 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 come una **coda di processi** in attesa di essere processati e un servitore che li serve 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 del nostro sistema si utilizzano:

- Una coda
- Il carico di lavoro. Questo è caratterizzato da:
 - λ_i : Tasso di arrivo all'i-esima risorsa
 - W_i : Tempo di attesa medio nella coda all'i-esima risorsa
 - S_i : Tempo di servizio dell'i-esima risorsa. Reciproco del tasso di servizio μ . Se, per processare una richiesta, ci si mette 1/10 di secondo, allora, in media, si serviranno 10 richieste/secondo (μ)

Altre metriche rilevanti sono:

- $R_i = S_i + W_i$: tempo di risposta medio
- $X_i = \max(\lambda, \mu)$: throughput medio
- N_{iw} : numero di richieste in coda (*waiting*) all'i-esimo nodo
- N_{is} : numero di richieste che stanno venendo servite all'i-esimo nodo
- $N_i = N_{iw} + N_{is}$: numero di richieste totali all'i-esimo nodo

Legge dell'utilizzazione di Little L'utilizzazione è la probabilità di avere il sistema “busy” ad un dato istante, ovvero impegnato a servire e/o con qualcosa in attesa di essere servito è:

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

L'utilizzazione è quindi scrivibile come il tasso di arrivo sul tasso di uscita, ovvero la percentuale di tempo in cui il sistema è occupato.

La legge di Little stima quante richieste vi sono

Notazione dei Sistemi a Coda I sistemi a coda vengono definiti mediante la seguente sintassi:

$$\{\text{Processo di Arrivo}\}/\{\text{Processo di Servizio}\}/\text{Numero di Servitori}/\text{Lunghezza della Coda}$$

Se la lunghezza della coda non viene specificata si assume che abbia lunghezza infinita. I Processi possono essere classificati con delle lettere, ciascuna corrispondente a una *distribuzione di probabilità*:

- M : Processo di **Poisson** (M sta per *Memory Less*, caratterizzato dall'assenza memoria). Viene solitamente utilizzata per richieste non correlate tra loro, come ad esempio le sessioni utente.
- D : Processo **Deterministico**
- G: Processo **Generico** (solitamente Gaussiana o Log Normale, una Gaussiana con sopra un Logaritmico)

Un processo Poissoniano è un processo nel quale gli arrivi non dipendono dallo stato, ovvero la probabilità di un arrivo è indipendente dalle altre. Hanno due caratteristiche fondamentali:

- Un tempo di inter-arrivo λ che rappresenta 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 $N + 1$
- Un tempo di servizio μ , che rappresenta con che probabilità si passa da N a $N - 1$, ovvero la probabilità di uscita dal sistema.
- Per la legge di Little, l'utilizzazione è quindi pari a $\rho = \frac{\lambda}{\mu}$

Il **tempo di risposta** è uguale a:

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

Graficamente è rappresentabile come una curva esponenziale con una determinata varianza e media. Questo perché se aumenta la capacità di processare (μ), a parità di λ , il tempo di risposta cala ma se aumenta il tasso di arrivo (λ), a parità di μ , il tempo sale.

M/M/1 Il tempo di risposta aumenta normalmente con l'aumento dell'utilizzazione (e quindi del carico) del sistema. Il *tempo di risposta* T_R è dato da: $T_R = \frac{1}{\mu - \lambda}$.

M/G/1 In questi sistemi, con distribuzione poisson-iana di arrivo e **generica** di servizio, viene utilizzato il teorema PASTA (Poisson Arrival See Time Average). Vengono introdotti diversi parametri come il tempo di attesa medio $E[W]$, il tempo di risposta medio $E[T]$ e il tempo medio di servizio $E[S] = \frac{1}{\mu}$. In particolare, analizzando la formula del tempo di attesa medio, questo è dato da:

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

Mentre il tempo medio di risposta è dato da:

$$E[T] = 1 + \frac{1 + C_v^2}{2} \frac{\rho}{1 - \rho}$$

Dove C_v^2 è il coefficiente di variazione, ovvero il **rappporto tra la deviazione standard e il valore medio**, il quale **descrive la variazione del tempo di servizio**. Per Poisson il valore di questo coefficiente è 1: trattandosi di sistemi senza memoria, questi tengono conto solo dello stato attuale del sistema. Per questo motivo possiamo affermare che il caso **M/M/1** è un caso particolare del più generale **M/G/1** con C_v uguale ad 1.

G/G/N Sistema Generico, senza restrizioni sul tipo di distribuzione dei processi di arrivo o di servizio. Anche il numero di servitori viene indicato con un numero n di servitori generici. Un sistema di questo tipo può essere usato per calcolare il tempo di risposta di un sistema generico.

$$W_M = \frac{P_{cb,N}}{\mu N(1 - \rho)} \frac{C_S^2 + C_D^2}{2}$$

$$P_{cb}, N \approx \begin{cases} (\rho^N + \rho) & \text{se } \rho \geq 0.7 \\ \rho^{\frac{N+1}{2}} & \text{otherwise} \end{cases}$$

Segue l'approssimazione di Allen-Cuneen:

- W_m = Tempo medio di attesa
- P_{cb}, N = Probabilità che tutti i server siano impegnati.
- C_s, C_d Coefficiente di variazione(deviazione standard / media):
 - C_s : Arrivo
 - C_d : Servizio.

3.0.2 Metriche di Failure

MTBF: Mean Time Between Failures

MTTF: Mean Time To Failures

MTTR: Mean Time To Repair

$$MTBF = MTTF + MTTR$$

La disponibilità A (Availability) è data da:

$$A = \frac{MTTF}{MTBF} = 1 - \frac{MTTR}{MTBF}$$

Questa indica la probabilità che ad un istante t il sistema stia funzionando correttamente. Oltre alla availability, che è una misura istantanea, abbiamo anche la **reliability**, che indica l'availability su un periodo di tempo sufficientemente lungo. Sulla base di questo fattore è possibile calcolare delle classi di disponibilità.

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 il ripetersi degli errori
 - * Diagnosis
 - * Isolation
 - * Reconfiguration
 - * Re-initialization

Considerando invece N sottosistemi, ciascuno con una probabilità di failure r_i la **probabilità che fallisca l'intero sistema** R dipende dalla tipologia di interazione del sistema:

- Seriale: basta che un solo sottosistema fallisca $\rightarrow R = \prod r_i$
- Parallelo: tutti i sottosistemi devono fallire $\rightarrow R = 1 - \prod(1 - r_i)$

3.0.3 CAP Teorem

Il CAP Teorem afferma che nella *progettazione di un datacenter* è possibile scegliere solo **due caratteristiche** tra:

- Consistency: ogni client vede lo stesso dato anche dopo una modifica o una cancellazione
- Availability: ogni client può trovare una replica del dato anche in caso di fallimenti parziali dei nodi
- Partition Tolerance: il sistema continua a funzionare nonostante malfunzionamenti parziali

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 diverso da ciò già presente nel mercato 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 i futuri bisogni dei clienti. Spesso si lavora su qualcosa senza alcuna garanzia di ritorno, incorrendo anche nel fallimento nei casi peggiori. Costruire un nuovo prodotto può essere dispendioso sia da un punto di vista economico che psicologico. Da un punto di vista finanziario si avranno degli alti e bassi, infatti la fatturazione è basata sul tempo.

Possiamo individuare 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

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%

Costo totale di proprietà, tutti i costi diretti, CAPEX, e indiretti, OPEX, di gestione di un bene durante la sua vita utile.

Le spese in conto capitale (CAPEX) sono spese per l'acquisto di beni o servizi significativi che verranno utilizzati per migliorare le prestazioni dell'azienda in futuro. Le spese in conto capitale riguardano tipicamente attività fisse come immobili, impianti e attrezzature (PP&E). Questi costi vengono ammortizzati nel corso degli anni.

Le spese operative (OPEX) sono i costi che un'azienda sostiene per la gestione delle sue attività

quotidiane. In quanto tali, non si applicano ai costi relativi alla produzione di beni e servizi. Tipicamente riguardano gli acquisti per i consumi entro l'anno di attività.

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.

Vantaggi TCO 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 considerate, non da nessuna tempistica e comporta un costo di acquisizione elevato. Inoltre può essere difficile quantificare tutto. Per un'azienda considerare un TCO significa seguire una strategia basata sul costo minimo 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 a considerare 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: $66.000 / 114.000 * 100 = 57.8$

5 Virtualization

Vi sono diversi motivi per utilizzare la virtualizzazione nel cloud:

- Ridurre l'utilizzo dell'hardware consolidando i server
 - Indipendenza Hardware.
 - Migliore scalabilità.
 - Gestione semplificata.
- Continuità
 - Backup
 - Ripristino di emergenza
 - Alta disponibilità
 - Failover hardware (multipathing)
- Maggiore efficienza nella gestione del ciclo di vita del software (sviluppo e test)
 - Ambiente di test
 - Provisioning delle applicazioni
 - Switch semplificato tra le versioni del software

Definizione La virtualizzazione è:

Astrazione delle componenti hardware degli elaboratori col fine di fornirli al software sotto forma di risorsa virtuale.

Possibili oggetti di virtualizzazione:

- Risorse hardware (es, CPU, GPU, memoria)
- Risorse di archiviazione
- Risorse di rete

La virtualizzazione è una tecnologia abilitante per il Cloud, come mostrato nel modello IaaS dove si usano VMs per astrarre problemi hardware.

La virtualizzazione può avvenire a diversi livelli:

- Emulazione →Circuiti
- Virtualizzazione →Hardware
- Containerizzazione →Sistema operativo

5.1 Emulazione

Emulazione di componenti hardware che tipicamente hanno un'architettura differente da quella dell'host, per cui occorre tradurre le operazioni macchina generate dal processore nell'architettura di destinazione. Presenta un'ottima flessibilità nonostante il grande overhead.

Esempi:

- BOCH
- VICE (Commodore 64 emulator)

5.2 Virtualizzazione

Terminologia:

- Hypervisor: software per l'esecuzione di VM
- Virtual Machine (VM)
- Guest: Una VM
- Host: il calcolatore in cui viene fatto girare l'hypervisor

Ogni Guest vede le proprie risorse hardware, le quali sono state virtualizzate a partire dall'hardware presente sull'host, senza accorgersi della virtualizzazione. Più VM possono esistere sullo stesso Host e possono lanciare sistemi operativi diversi e combinazioni di hardware differenti.

Alcuni esempi:

- VMWare
- VirtualBox
- Xen
- KVM

Containerizzazione Più ambienti separati:

- Gruppo di processi per ogni ambiente (es. `cgroups`)
- Separazione tra gruppi di processi (es. `namespace`)

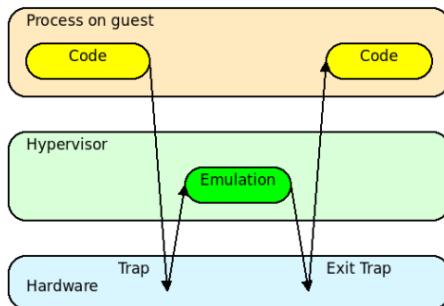
Tutti i gruppi di processi condividono lo stesso kernel, tipicamente Linux. Sono presenti software aggiuntivi per la gestione: delle immagini del file-system, della rete o del ciclo di vita dei container. Esempi:

- Docker
- Kubernetes (Google)

5.3 Approccio alla virtualizzazione

Grazie alle definizioni di macchina virtuale, memoria virtuale e hypervisor, possiamo definire alcuni concetti che forniscono i requisiti minimi hardware per supportare la virtualizzazione.

Un primo approccio storicamente utilizzato è *"Trap ed execute"*: il codice gira sul guest fino a quando non viene invocata una istruzione **sensibile** che genererà una trap. Le trap sono gestite dall'hypervisor che si occupa di emulare il comportamento corretto e far riprendere l'esecuzione.



Come si può notare, non tutte le istruzioni sono uguali, infatti quelle che possono accedere o modificare lo stato della macchina vengono considerate critiche in quanto possono:

- Interrompere l'esecuzione.
- Accedere al registro degli stati.
- Accedere alla memoria tramite indirizzo fisico.

Le istruzioni sensibili devono quindi essere gestite solo dall'hypervisor tramite l'uso di "trap" a livello hardware.

Queste operazioni di trap sono onerose e rendevano lente le piattaforme virtuali, ecco perchè la virtualizzazione ha cominciato a perdere d'interesse. Le tecnologie abilitanti per il cloud computing hanno fatto sì che la virtualizzazione prenda più piede tra le tecnologie emergenti degli ultimi anni. Il nuovo approccio consiste nell'utilizzare software appositamente ottimizzato per la virtualizzazione e l'introduzione di supporto hardware compatibile con quest'ultima e notevolmente più potente.

5.4 Principi di virtualizzazione

Grazie al paper di Popek e Goldberg del 1974 si stabilirono delle basi teoriche sui concetti chiave e requisiti della virtualizzazione, oltre che alla definizione dell'approccio trap and execute.

Definizione VM:

Una macchina virtuale è considerata un duplicato efficiente e isolato della macchina reale.

In quanto software un VM Monitor ha tre caratteristiche essenziali. In primo luogo, il VMM fornisce un ambiente per i programmi che è essenzialmente identico al computer originale; secondo, i programmi eseguiti in questo ambiente mostrano nel peggiore dei casi solo lievi diminuzioni di velocità; e infine, il VMM ha il controllo completo delle risorse di sistema.

Identifichiamo quindi tre requisiti fondamentali:

- Equivalence/Fidelity
- Resource control/Safety
- Efficiency/Performance

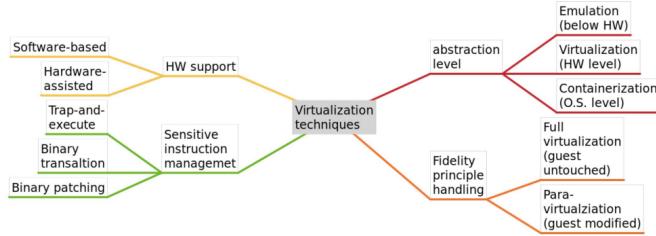
E, come già accennato prima, una differente classificazione delle istruzioni:

- Istruzioni **sicure**: Eseguibili dal guest senza problemi
- Istruzioni **privilegiate**: Supponendo che la CPU abbia due modalità di esecuzione, supervisor (*ring 0*) e standard (*ring 3*), queste istruzioni sono eseguibili solo in modalità supervisor.
- Istruzioni **sensibili**: Sono quelle istruzioni che l'hypervisor deve trappare per emulare il normale comportamento di un OS dando l'illusione al guest di possedere le risorse hardware. Un'istruzione sensibile è, quindi, un'istruzione che osserva o modifica lo stato privilegiato della macchina, ovvero qualsiasi stato che può essere utilizzato per modificare il livello di privilegio del processore corrente.

Se tutte le istruzioni sensibili sono anche privilegiate, la virtualizzazione diventa più facile e i trap vengono generati automaticamente quando un'istruzione sensibile viene eseguita dall'hypervisor. Se invece sono presenti istruzioni sensibili che non sono privilegiate potremmo avere dei problemi, in quanto l'hypervisor non può fare affidamento su trappole per acquisire e modificare il proprio comportamento e diventa più complessa l'implementazione.

5.5 Tecniche di virtualizzazione

Abbiamo visto precedentemente che le tecniche di virtualizzazione si basano sul livello al quale vogliamo eseguire l'astrazione (emulazione, virtualizzazione, container), attraverso approcci basati sull'hardware o sull'approccio per gestire le istruzioni sensibili.



5.5.1 Emulazione

Un interprete emula il comportamento dell'hardware riproducendo l'architettura desiderata. Il sistema operativo nativo viene eseguito sull'hardware emulato riproducendo le caratteristiche principali del chip.

- Vantaggi:
 - Separazione completa dall'ambiente emulato.
 - Può eseguire software pensato per hardware diverso.
 - Elevata flessibilità.
- Svantaggi:
 - Enorme overhead
 - Non è per nulla vicino alla velocità nativa a meno che non si emuli un hardware più vecchio.

5.5.2 Traduzione binaria

A differenza dell'approccio basato sulle trap, che funziona a *tempo di esecuzione*, si ha un cambio di paradigma passando ad un approccio proattivo, intervenendo sul codice **prima** che questo venga eseguito. Grazie alla compilazione anticipata, introdotta per dare supporto all'emulazione, questo metodo viene usato anche adesso nei sistemi di virtualizzazione.

I suoi punti cardine sono:

- La traduzione di istruzioni sensibili ma non privilegiate.
- Può funzionare anche in fase di esecuzione.
- Può memorizzare nella cache il codice tradotto.

Si articola in due fasi:

- Scansione del codice: Viene analizzata solo una porzione del codice, concentrandosi a livello kernel, lasciando in sicurezza il codice *userspace* perché si presuppone che quest'ultimo non utilizzi funzioni sensibili.
- Modifica del codice: Viene modificato il codice originale sostituendo le istruzioni sensibili in codice sicuro. Vengono **mantenute** entrambe le copie.

Le unità di codice considerate sono sequenze di istruzioni dette **basic block**. I blocchi possono essere tradotti una volta, riscrivendo gli indirizzi di salto presenti nel codice originale, e salvati nella cache per un eventuale riutilizzo.

```

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

isPrime:    mov    %ecx, %edi ; %ecx = %edi (a)
            mov    %esi, $2      ; i = 2
            cmp    %esi, %ecx ; is i >= a?
            jge   prime       ; jump if yes
            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
            mov    %eax, %ecx ; set %eax = a
            nexti:    mov    %eax, %ecx ; set %eax = a
            cdq
            idiv   %esi          ; a % i
            test   %edx, %edx ; is remainder zero?
            jz    prime        ; jump if yes
            notPrime:    mov    %eax, %esi ; a = %eax
            ret
            prime:    mov    %eax, %esi ; a = %eax
            xor    %eax, %eax ; %eax = 0
            ret

```

Figure 6: Codice in C

Figure 7: Codice del compilatore tradotto

Regole per la traduzione:

- IDENT: il codice può essere tradotto in modo identico
- Flusso di controllo diretto: mappatura degli indirizzi in codice tradotto per istruzioni jmp, call, ret
- Flusso di controllo indiretto: quando l'indirizzo non è disponibile al momento della traduzione bisogna calcolare gli indirizzi al volo.
- Istruzioni privilegiate: bisogna riscriverle accuratamente

La traduzione binaria riduce l'overhead durante l'esecuzione in quanto il grosso del compito viene svolto prima dell'esecuzione del programma, in fase di compilazione.

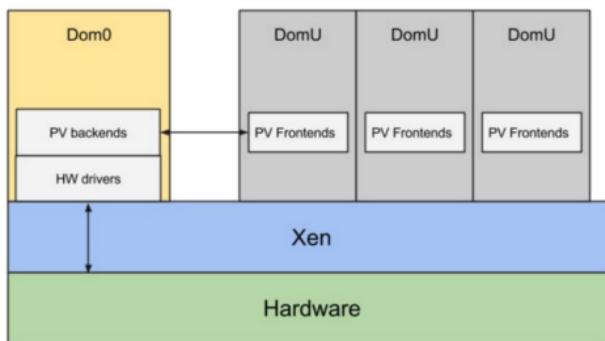
5.5.3 Patch Binaria

E' molto simile alla traduzione binaria ma la differenza sta nella gestione del codice tradotto. In questo caso il codice tradotto va a sostituire il codice originale. Il controllo per la memorizzazione della cache è, però, più complesso. Un esempio di questa tecnica di virtualizzazione è Virtual Box

5.5.4 Paravirtualizzazione

Il software di paravirtualizzazione agisce direttamente sull'hardware in modo da gestire la condivisione delle risorse destinate alle varie virtual machine. Il codice del guest quando deve eseguire istruzioni sensibili utilizza delle API per interagire con l'hypervisor per interrupt o paginazione della memoria, dette **hypercalls**. Il guest deve quindi essere a conoscenza del fatto che opera in ambiente virtualizzato.

Come nel caso della traduzione binaria, la traduzione avviene in fase di compilazione. L'esempio più comune è il progetto Xen.



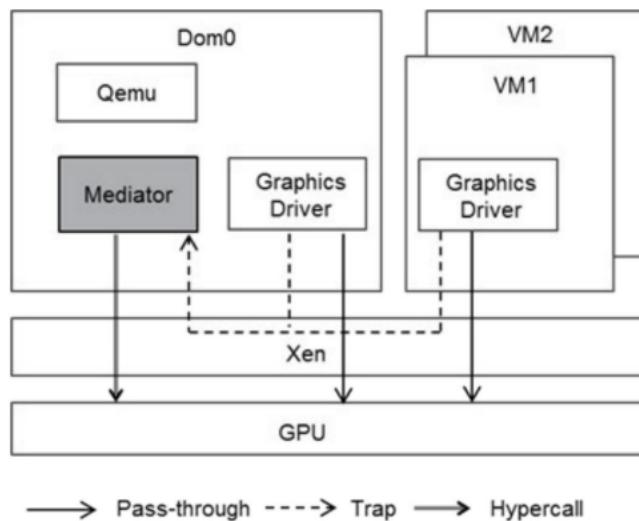
I limiti sono:

- Modificare il guest: Occorre il supporto da parte del sistema operativo.

- Per rappresentare una valida alternativa, la paravirtualizzazione deve essere integrata con altre opzioni qualora non sia possibile modificare il guest.

Xen è un kernel molto piccolo che viene caricato per primo sulla macchina host e ottiene il controllo diretto dell'hardware. Sopra Xen, a un livello di privilegio inferiore, abbiamo i cosiddetti **domini**. Ci possono essere tanti domini quanti vogliamo, e all'interno di ogni dominio si può eseguire un intero sistema operativo con le proprie applicazioni. I domini sono isolati l'uno dall'altro e sono utilizzati per implementare le macchine virtuali guest. Un dominio speciale, Dom 0, ha accesso all'API di Xen per creare e distruggere altri domini tramite l'utilizzo di **hypercalls**.

I dispositivi paravirtuali sono suddivisi in un **front-end** e un **back-end**, ogni dispositivo viene eseguito in un dominio diverso e scambia richieste di I/O al di fuori del proprio. La configurazione tipica prevede che il back-end venga eseguito in un dominio che ha accesso diretto ai dispositivi hardware (Dom0).



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 così da facilitare il lavoro dell'hypervisor quando dovrà assegnarla. In questo modo non c'è bisogno di tradurre la RAM vera con quella del guest stesso (come avviene in ambienti virtualizzati canonici).

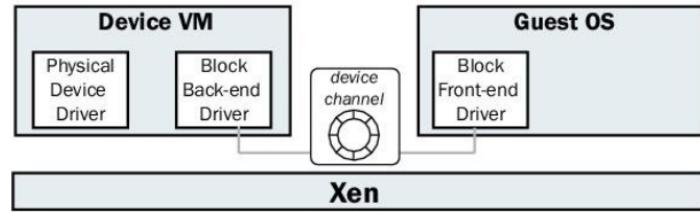
Gestione dell'hardware Sono concetti implementati sia in hypervisor di virtualizzatori complessi sia in hypervisor di paravirtualizzatori.

- Emulazione dell'hardware: Massimizza la sicurezza. Utilizza la tecnica trap and execute. Si può avere un enorme overhead portando a un basso throughput e pessime performance (specialmente per quanto riguarda la GPU).
- Accesso diretto all'hardware: Per i dispositivi che lo supportano è possibile l'**assegnazione diretta** alla macchina virtuale. Massimizza le performance a discapito della sicurezza. Un ruolo importante dell'hypervisor è quello di escludere l'accesso simultaneo allo stesso hardware.
- Split driver: Un compromesso tra i due. In questo caso si hanno due parti:
 - Back-end che può accedere all'hardware e gestisce l'accesso simultaneo. Si occupa di fare l'accesso all'hardware.
 - Front-end che è in esecuzione sul guest, non è altro che una coda di richieste da cui il backend preleverà.

Back-end e Front-end sono consapevoli l'uno dell'altro e scambiando dati in modo efficiente usando hypercall e code di eventi.

Split Device Drivers in Xen

Physical driver runs in an isolated VM, connected over a shared memory device channel to a guest VM accessing the device.

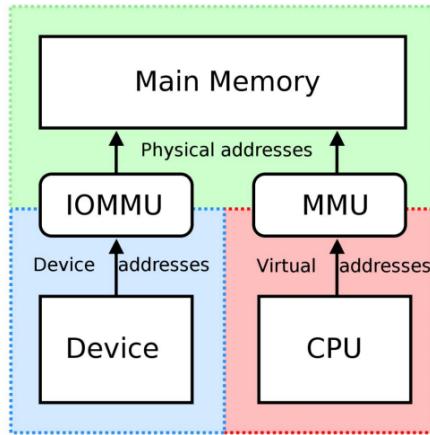


5.6 Extensions

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 viene usato un processo “*ballooner*” che le pagine di memoria correntemente non utilizzate, ”gonfiandosi”. Quando una VM richiederà più RAM, l’hypervisor potrà interrogare il processo ballooner per richiedere la memoria da assegnare alla VM, facendolo ”sgonfiare”. Grazie alla delega di queste operazioni, l’hypervisor può essere meno complesso in termini di codice.

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 ballooner e non alla macchina virtuale, risparmiando notevolmente tempo.

Estensioni Hardware Per soddisfare i requisiti di P&G, le architetture vengono estese per avere un ulteriore livello destinato all’hypervisor di operatività (*ring -1*). 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 mentre il kernel dell’hypervisor girerà a *ring 0* VMX root. Per commutare modalità verranno utilizzate apposite traps.

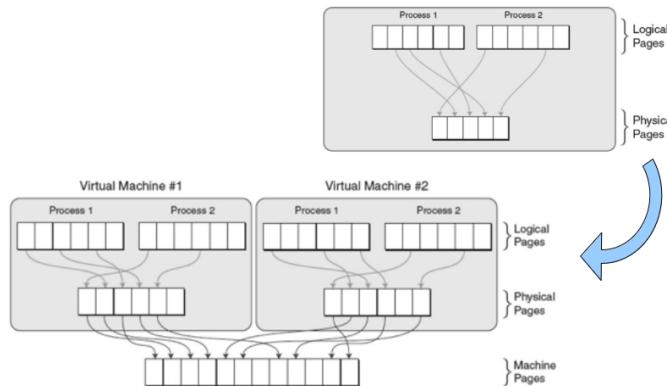
Vengono anche create delle strutture dati (VM control structures) che permettono di gestire le macchine virtuali come se fossero dei processi (come i *Process Control Blocks*). Le istruzioni aggiuntive permettono di interagire con tali strutture dati, consentendo di gestire le macchine virtuali a livello hardware.

Estensioni Network and I/O 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.

5.7 Virtualizzazione della memoria

Nel contesto virtualizzato, il modello paginato inizia ad avere dei problemi, serve un livello di indirezione ulteriore. Tipicamente è possibile risalire alle pagine fisiche, partendo da quelle logiche, semplicemente consultando la *page table*. In un contesto virtualizzato occorre, però, una page table su più livelli: uno per la macchina virtuale, che sta sull'host, e uno per ogni macchina virtuale e i suoi spazi di indirizzamento.

La virtualizzazione aggiunge quindi anche un altro livello indiretto: per ogni guest sono presenti pagine considerate dal guest fisiche ma che fisiche non sono perché l'hardware è virtualizzato. Vengono dunque sviluppati, dai vendor delle CPU, supporti hardware per TLB *innestata* e la MMU.



Tuttavia, è necessario che a partire dalla page table guest non sia possibile risalire fino a quella fisica dell'host, altrimenti il guest avrebbe essenzialmente il controllo della macchina. Vengono in aiuto le cosiddette **shadow page tables**, tabelle costruite dall'hypervisor, in cui si tiene traccia della rappresentazione delle page table del guest. Viene utilizzato il registro CR3, un puntatore al base page register per il guest correntemente in esecuzione, per gestire il cambio di page table. La necessità di mappare le operazioni sulle pagine guest alle pagine dell'host può però portare ad un sovraccarico significativo nonostante sia trasparente per i guest.

Un approccio alternativo prevede l'uso della paravirtualizzazione: guest e VMM condividono le tabelle ma il guest ha accesso in sola **lettura** alle pagine e la modifica avviene per mezzo di hypercall. Questo approccio particolarmente efficiente è utilizzato in Xen.

L'approccio attuale prevede invece l'utilizzo delle pagine nidificate, affidandosi al supporto hardware. Vi è un TLB per le pagine dei guest e uno aggiuntivo per mappare l'indirizzo fisico del guest su quello dell'host.

5.8 Virtualizzazione dell'archiviazione

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. Abbiamo due possibili approcci:

- VMDK: Consiste nella descrizione di un'immagine del disco fisica rappresentante la geometria e i settori del disco. La struttura del file è **sparsa**, cioè i blocchi vuoti non occupano effettivamente spazio ma se ne tiene traccia tramite metadati, così da ottimizzare lo spazio su disco. L'altra rappresentazione è quella **flat**, cioè l'allocazione contigua del file.

I file VMDK permettono di creare degli **snapshot** dei filesystem: una volta che ho, ad esempio, installato il sistema operativo è possibile catturare un'istantanea dello stesso. Viene quindi creato un nuovo layer sul quale verranno scritte le modifiche successive. Per realizzare questo meccanismo vengono usati i *delta links*, che contengono le modifiche fatte sullo snapshot (**base disk**).

Diverse copie della stessa VM potranno partire dalla stessa immagine di base per creare diverse versioni con le proprie modifiche apportate mediante i delta links.

- QCOW: Evolve VMDK, implementando compressione, crittografia. Aggiunge supporto a sparse files anche se il sistema operativo host non lo supporta.
- File Docker a più livelli, come squashFS.

5.9 Gestione delle risorse

La prima risorsa da dover gestire accuratamente è la memoria che deve essere sempre allocata e restituita all'hypervisor quando non viene utilizzata dal guest.

Per quanto invece riguarda la CPU, per evitare che le macchine virtuali la saturino è necessario un intervento dell'hypervisor che fissa un limite agli utilizzi delle varie VM. Lo *stealing* della CPU si verifica quando i processi sono pronti per essere eseguiti dalla CPU virtuale ma rimangono in attesa che l'hypervisor assegna loro una CPU fisica. Ciò accade perché l'hypervisor sta servendo un'altra macchina virtuale. Se $steal > 0$ l'hypervisor deve applicare il limite. Ci sono due cause principali di *steal* della CPU:

- I processi richiedono più della CPU a loro allocata
- Il server fisico è sovraccaricato dalle macchine virtuali

Sfortunatamente, è difficile capire in quale di questi due casi rientra una situazione solo guardando lo *stealing time*.

L'ultimo problema da gestire è l'**interferenza** tra le varie macchine virtuali, ovvero la contesa sulle risorse (CPU, Rete) fisiche. Ad esempio ogni volta che avviene un context switch tra VM la cache si svuota per accogliere i contenuti del nuovo guest. E' difficile ottenere un perfetto isolamento delle prestazioni tra varie macchine virtuali anche se di norma il comportamento di una macchina virtuale non dovrebbe infierire su altre macchine virtuali.

Una conseguenza di questo fenomeno è il fatto che le interferenze possano essere usate come *side-channel* per attacchi informatici: due macchine virtuali apparentemente non comunicanti tra di loro, possono in realtà interagire facendo appositamente variare l'utilizzo di risorse. Questa variazione può essere codificata per comunicare oppure ottenere informazioni dalle altre VM.

5.10 Migrazioni di macchine virtuali

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

- Migrazione offline: stop su nodo 1 e 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 implementare per un hypervisor, ma non necessariamente quella ideale in scenari di high availability.

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 con layer multipli 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 inizierà a copiare le pagine, partendo da quelle meno “hot” (perché, ovviamente, sono quelle che saranno sporcate più facilmente, quindi avrà meno senso copiarle). Tutte queste operazioni sono fatte con la macchina di origine accesa, pertanto è possibile che durante il procedimento le pagine in copia vengano sporcate, richiedendo più round di copia. Oltre allo spostamento delle pagine, viene spostato anche il contenuto del disco
- Migration: la VM di origine viene arrestata e vengono copiate le pagine “hot” per rigenerare interamente la macchina a destinazione. Viene trasferito anche lo stato dei registri della macchina di origine. Infine, la macchina di origine viene fermata, mentre quella di destinazione accesa. In questo ultimo step, la disruption sarà nell’ordine dei millisecondi

6 Containers

I container permettono la distribuzione di software auto-contenuto (con librerie, configurazioni e file necessari al funzionamento), garantiscono la compatibilità con il software pre-esistente (anche condividendo le librerie quando è utile) e la coesistenza di setup differenti grazie agli spazi isolati e modulari. Il tutto impacchettato in modo che sia facile da condividere e installare.

Le componenti di un software sono:

- Processi: un processo è composto da:
 - Codice statico: Elenco delle istruzioni del programma
 - Stato dinamico: Registri, memoria, allocazione delle risorse di sistema.
- Codice statico: programmi, librerie.
- Pacchetti software.

I programmi utilizzano repository di codice aggiuntivi per le loro attività come librerie di funzioni comuni (libc, stdlib, GUI etc.). Esistono due tipi di linking:

- Statico: In fase di compilazioni dove i simboli vengono risolti dal linker.
 - file .o: Oggetto creato dal compilatore.
 - file .a: Librerie.
- Il file di output è collegato staticamente, non è necessario interagire con le librerie di sistema e interagisce solo con le chiamate di sistema.
- Dinamico: Si sommano i tempi di compilazione con il tempo di binding a runtime. I simboli anche qui sono risolti dal linker:
 - .o files: Oggetto creato dal compilatore.
 - .a files: stubs di librerie
 - .so files: oggetti condivisi, possono essere condivisi da processi multipli, in modo da risparmiare spazio nel disco e nella memoria.
 - ld.so: linker dinamico.

Linking statico	Linking dinamico
File eseguibili di grandi dimensioni	Piccoli file eseguibili
Interagisce solo con il kernel	Interagisce anche con gli oggetti condivisi
Nessun'altra dipendenza durante l'installazione	Dipende dalla disponibilità delle librerie corrette
Necessità di ricompilare per aggiornare le librerie	Può aggiornare le librerie senza toccare l'eseguibile
Esempio: Distribuzioni Linux	Standard nella maggior parte dei sistemi

Table 1: Differenze fra linking statico e dinamico

I pacchetti contengono il software per l'installazione (.dev, .rpm, .apk), che contiene meta-informationi sulle dipendenze, i possibili problemi possono essere:

- Grafo delle dipendenze complesso
- Problema di aggiornamenti e conflitti sulle dipendenze

Se vogliamo confrontare i container all'utilizzo della macchina virtuale:

	Container	Virtual Machine
Astrazione	Livello sistema operativo	Hardware
Eseguibilità	Stesso sistema operativo	Può eseguire diversi sistemi operativi
Utilizzo memoria	Ridotto	Ampio ed elevato
Tempo di avvio	Ridotto	Elevato (avvio sistema operativo)

6.1 Principi di containerizzazione

Il concetto di base nasce con l'introduzione del comando `chroot` che permette ad un processo e ad i suoi *child processes* di lavorare con una root directory diversa da quella del sistema. Rappresenta il primo esempio di software isolation.

Successivamente, verrà introdotto il concetto di `jail`, funzionalità che permette l'isolamento a livello network, oltre che a quello di filesystem. 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.

Tuttavia `chroot` e `jail` forniscono una sicurezza limitata ed è possibile eluderne i vincoli. Infatti, è possibile usare i path relativi per muoversi al di fuori alla `chroot` jail. Inoltre, `chroot` permette soltanto di isolare un processo a livello di filesystem, non a livello di risorse.

Lo sviluppo della containerizzazione va di pari passo con lo sviluppo della virtualizzazione, infatti i due fenomeni si muovono quasi parallelamente nel tempo, perché 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.

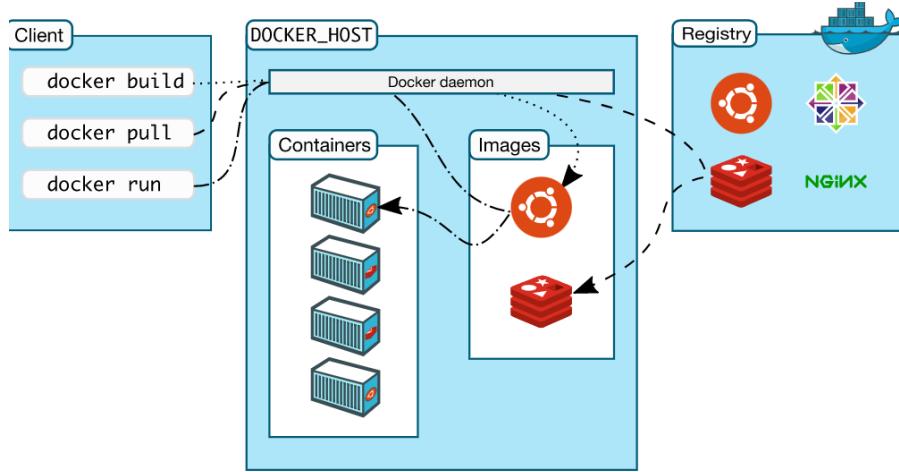
Namespace e CGroups I `namespace` Linux sono meccanismi che permettono di isolare un processo su molteplici livelli: network, fs, processlist etc. Ogni namespace, infatti, fa riferimento a qualche area di interesse del sistema operativo ed è isolato rispetto a tutti gli altri: quando un processo si associa ad un namespace, vedrà solamente le risorse associate a quest'ultimo. I `cgroup`, invece, sono meccanismi del kernel che consentono di gestire l'allocazione delle risorse ai processi.

- `namespace` → limitano la visibilità del processo
- `cgroup` → limitano l'utilizzo delle risorse dell'host per il processo

I vantaggi dei container sono i seguenti:

- Portabilità: I software dei container sono eseguibili e tutte le librerie sono autonome. Il container può essere eseguito su diversi sistemi
- Agilità: Supporto essenziale per lo sviluppo in DevOps, che garantisce un lavoro più fluido, aggiungendo solo configurazioni e codice specifico inerente al servizio.
- Velocità: I container sono un tipo di virtualizzazione molto leggero quindi hanno un tempo di avvio più rapido e incidono meno sul sistema, che si traduce in costi minori.
- Isolamento: Un container rimane perfettamente isolato e non inficia ne su altri container né sull'host. Per isolamento si intende anche un isolamento delle prestazioni, in quanto l'allocazione di memoria e CPU è limitata per il container.
- Efficienza: Relativa alla velocità e anche al rischio ridotto di sovraccarico, rispetto alla macchina virtuale che condivideva CPU e memoria. Inoltre non c'è bisogno di alcun driver para virtualizzato, il che rende più rapido l'accesso alle risorse.
- Facilità di gestione: Il container gestisce il networking (DNS,...), il bilanciamento del carico, assegnazione delle risorse, *mounting* dell'archiviazione locale e basata sul cloud.
- Sicurezza: Il codice dannoso nel container non può propagarsi (isolamento).

6.2 Docker



6.2.1 Concetti

Gli elementi che compongono Docker sono i seguenti:

- Container Management a cura di **Containerd**
- Container Engine a cura di **Docker**
- Data Management a cura dei file systems
- Orchestration a cura di **Docker-compose**, **Docker-swarm**, **Kubernetes**

Containerd Si occupa dell’astrazione e della esecuzione dei container a basso livello. Le sue funzioni principali, esposte tramite API, sono:

- **Esecuzione** dei container (usando **runc**)
- Gestione delle **immagini** dei container
- Push/Pull delle immagini nel **registro** dei container.

Containerd è a sua volta un’astrazione di **runc**. I molteplici strati su cui è costruito Docker sono fatti per semplificare la portabilità sia tra container runtimes (ad esempio Kubernetes usa runtime differenti) sia tra diversi sistemi operativi (grazie al fatto che containerd è stato portato per molteplici OS).

Docker è un **container engine** usato per:

- **Assemblare** container
- **Gestire** container
- **Eseguire** applicazioni container-izzate

Fornisce quindi i benefici dei container per uno sviluppo DevOps friendly e scalabile. Ha un’architettura client-server:

- Un demone gestire i container (**dockerd**)
- Il client invoca i servizi del demone (**docker**) tramite REST API o UNIX socket

Docker fornisce un repository delle immagini dei container mediante il **registro**, utilizzato per salvare e accedere alle immagini container.

Docker gestisce oggetti che possono essere **immagini** o **container**. Le immagini sono template read-only, contengono istruzioni per creare un container e possono anche essere derivate da altre

immagini.

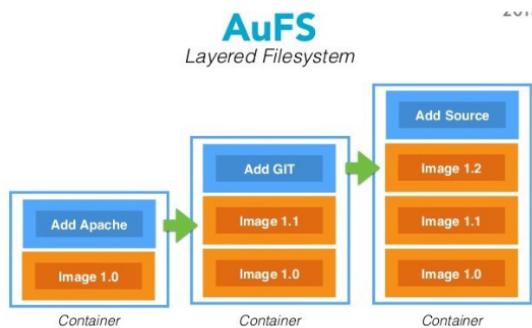
Esempio: Apache httpd server, è basato su un immagine Linux che aggiunge file per lanciare Apache httpd.

I container sono istanze eseguibili di un immagine, supportano operazioni di creazione, start/stop, cancellazione. L'interazione è basata su comandi docker o comandi a riga. I container sono definiti da file di configurazione e da immagini.

6.2.2 Container file-system

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

I container invece sono oggetti di lettura e scrittura ma solo il livello **superiore** è scrivibile in quanto basato sull'approccio 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 file modificati, e solo quelli, vengono copiati nel layer read-write succitato.



I container possono condividere la stessa immagine, con la differenza del livello superiore. Il sistema Copy-On-Write aumenta l'efficienza diminuendo il tempo di avvio e risparmiando spazio su disco.

Sono disponibili anche altri approcci:

- File system di unione: AUFS, sovrapposizioni
- Snapshot: BTRFS, ZFS.
- Dispositivi a blocchi Copy-On-Write.

	Unione di file system	Snapshot dei file system	File system di copia su scrittura
Approvvigionamento	Molto veloce Molto a buon mercato	Veloce A buon mercato	Veloce A buon mercato
Modifica di file di piccole dimensioni	Molto veloce Molto a buon mercato	Veloce A buon mercato	Veloce Caro
Modifica di file di grandi dimensioni	Lento (prima volta) Inefficiente (copia)	Veloce A buon mercato	Veloce A buon mercato
Diffidare	Molto veloce	Molto veloce	Lento
Utilizzo della memoria	Efficiente	Efficiente	Inefficiente (tieni tutto nella RAM)
Svantaggi	Non tutti i FS nella linea principale Problemi AuFS	ZFS non nella linea principale Btrfs non è carino	Utilizzo inefficiente del disco

Ogni livello del file system è una directory con un nome di una stringa esadecimale, la directory del livello più basso possiede:

- File di collegamento

- Directory diff con il contenuto del file system.

6.3 Docker Swarm

Servizio di modello dichiarativo, descrive come deve essere fatto il deployment e come sono strutturati i vari containers. Attraverso un meccanismo di scaling automatico si definiscono i livelli di replicazione per ogni servizio. La riconciliazione dello stato desiderato serve per analizzare man mano lo stato corrente.

Multi-host networking Riusciamo a distribuire le nostre connessioni su più nodi attraverso un overlay network, un contenitore di interconnessioni mappato su un deployment distribuito, la configurazione della rete è automatica.

Service Discovery DNS server integrato in swarm, per cui ogni container può comunicare con un altro container non su indirizzo IP ma attraverso un nome precedentemente definito.

Load Balancing Attraverso il meccanismo interno di gestione della rete da parte di Docker riusciamo a gestire il carico in maniera bilanciato, andando a specificare come devono essere mappati i nostri nodi. Questo può servirci per capire come gestire l'Availability del nostro sistema, andando a istituire diverse policy.

Sicurezza Docker Swarm è sicuro di default, la comunicazione criptata tra i vari nodi viene istituita tramite identificazione. E' possibile istituire certificati di autenticazione locali o esterni.

Aggiornamenti Docker Swarm ci permette di fare degli aggiornamenti continui in maniera textbf{incrementale}, non c'è bisogno di aggiornare ogni cosa alla volta. Questo ci può permettere di evitare problemi di traffico su :

- Hosts
- Volumi
- Immagini di registro

6.3.1 Concetti di Docker Swarm

Uno swarm è una rete di nodi, dove all'interno vi è un host (fisico o virtuale), che può essere:

- Manager: Gestione dei membri e delle delegazioni. Riceve le descrizioni per il deployment, distribuisce l'unità di lavoro (task) su più nodi. Possono anche agire come Worker node.
- Worker: Manda in esecuzione i servizi swarm. Riceve ed esegue i task, gli agenti su questi nodi ricevono e monitorano i task
- Entrambi.

Un nodo è un'istanza di sistema docker, tipicamente si ha un nodo per macchina, ma molti possono coesistere più macchine su un nodo.

6.4 Kubernetes

Kubernetes è un software formato da più componenti software distribuite in nodi master e nodi worker che formano il cluster kubernetes.

Le componenti che si occupano di controllare l'esecuzione dei container applicativi sono raggruppate nel control plane. Il data plane raggruppa invece le componenti software coinvolte nelle funzionalità che gestiscono il carico di lavoro del cluster.

Il controllo del sistema avviene specificando un desired state (stato desiderato). Ogni partecipante

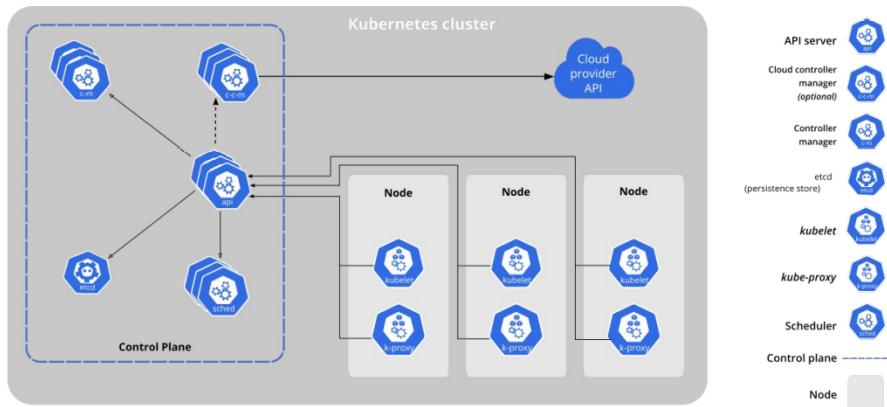
si attiva per contribuire a mutare il sistema verso il desired state definito nel master.

Kubernetes ha funzionalità auto-riparanti: rileva e gestisce il guasto di un container, che può essere:

- Container che non rispondono
- Controlli definiti dall'utente

6.4.1 Componenti

- Nodi: Un nodo è un server fisico oppure un server virtuale. Essi si coordinano per l'esecuzione dei carichi di lavoro, ovvero delle applicazioni containerizzate (container).
- Pods: Il pod è la risorsa che descrive l'unità elementare eseguibile su un nodo del cluster. Un pod raggruppa dei container che condividono le risorse e che vengono eseguiti sullo stesso nodo. Il pod si occupa di astrarre rete e storage al fine di poter essere spostato e replicato facilmente sui nodi del cluster, permettendo una forte scalabilità orizzontale
- Control Plane: L'attore centrale di un cluster in quanto a lui fanno riferimento tutti gli altri nodi per coordinarsi nell'esecuzione dei container.
- Service: Definisce come esporre dei pod su una rete interna o esterna.



Pod Unita di carico di lavoro di base in Kubernetes, esistono più tipi di pod:

- Pod contenitore singolo: caso più comune, viene eseguito un mapping 1 a 1 con i containers
- Pod contenitori multipli: un singolo pod contiene più applicazioni complesse. Stretto accoppiamento dei contenitori

Template Pod:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: hello
spec:
  template:
    # This is the pod template
    spec:
      containers:
        - name: hello
          image: busybox
          command: ['sh', '-c', 'echo "Hello, Kube!" && sleep 3600']
      restartPolicy: OnFailure
```

Se viene modificato un modello pod, i pods vengono ricreati da un nuovo modello. Il modello Pod può specificare i volumi, simili ai volumi Docker. Ogni pod ha un IP e i vari contenitori all'interno dello stesso pod possono comunicare utilizzando interfaccia di loopback.

Control plane La gestione delle applicazioni viene fatta attraverso chiamate API, le configurazioni vengono salvate tramite una struttura chiave valore.

Scheduler Tiene traccia dei pods, si concentra sui nuovi pod che non vengono assegnati a dei nodi, mentre gestisce l'assegnazione dei task ai vari nodi. Implementa funzioni per il bin-packing, bilanciamenti e gestioni delle ridondanze.

Controller Manager Gestisce i processi del controller, la sua implementazione di riferimento è quella del kube-controller-manager, è un controller specifico per il cloud che ha accesso alle API del provider. Vi sono poi altri controller, per vari aspetti dell'architettura:

- Nodi.
- Replicazione.
- Endpoints.
- Service account e token.
- Instradamento.
- Servizi: bilanciamento del carico del cloud.

7 Serverless

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

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.

Se facciamo un confronto con le architetture IaaS dove si ha un controllo completo sull'infrastruttura, l'approccio senza server è un ridimensionamento automatizzato monitorato dal provider. Il servizio viene arrestato quando non utilizzato. L'utente deve quindi preoccuparsi solo dell'aggiornamento del proprio software.

	Serverless	IaaS
Scalabilità	Automatizzata	Gestita dall'utente
Monitoring	Gestito dal provider	Gestito dall'utente
Costi	Sulla capacità consumata	Sulla capacità allocata
Sicurezza	Gestita dal provider	A carico dell'utente

I principali vantaggi sono:

- Gestione: Nessuna preoccupazione per la gestione e la governance dell'infrastruttura.
- Costi: Test, risoluzione dei problemi, controllo degli accessi esternalizzati al fornitore di servizi cloud.
- Scalabilità e disponibilità: Il ridimensionamento orizzontale è automatico ed elastico. La tolleranza ai guasti è un problema del fornitore di servizi cloud
- Riduzione al minimo della latenza: Le funzioni serverless in genere non operano da un singolo server di origine ma da molteplici server distribuiti in modo che l'utente possa reindirizzato a quello più vicino
- Riduzione della complessità del software: Le funzioni devono essere semplici per poter essere deployate serverless e ciò aiuta a risolvere i bug e rilasciare aggiornamenti
- Miglioramento della produttività del software: Separazione di sviluppatori software da amministratori dell'infrastruttura

Vi sono due categorie di servizi serverless:

- Backend as a Service → Google Firestore
- Function as a Service → Google Functions

7.1 Backend as a Service

Tipicamente il backend fornisce un servizio ad altre applicazioni sotto forma di API messe a disposizione per gli sviluppatori. Alcuni esempi possono essere Cloud Databases, servizi di autenticazione e autorizzazione, notifiche push etc.

7.2 Function as a Service

Se il software può essere suddiviso in funzioni ancora più piccole dei microservizi che esse servono allora queste possono essere deployate su server esterni in modo da separare lo stato dei server, e quindi affrancandosi dal bisogno di gestire l'infrastruttura.

Così facendo abbiamo una logica a grana ancora più fine in cui varie funzioni compongono un microservizio e, a loro volta, diversi microservizi costituiscono l'applicazione monolitica.

Le caratteristiche principali delle Functions sono:

- Stateless: Inerentemente parallele, sono facilmente scalabili
- Effimere: Quando non utilizzate vengono dismesse. Non c'è persistenza
- Event-driven: Determinate condizioni di attivazione scatenano un'azione che viene eseguita

- Completamente gestite dal cloud provider
- Modello di costi per uso (ad esempio la RAM viene misurata in quantità occupata al secondo)
- Business Logic: Ci si concentra solo sulle funzionalità e non sull'infrastruttura

Gli utilizzi principali sono in quegli scenari in cui lo scaling automatico può fare la differenza, dove cioè ci sono grosse differenze di traffico tra il picco e il minimo oppure dove ci sono prolungati periodi di idle e workload incerti e variabili.

Gli svantaggi sono legati a doppio filo ai vantaggi che porta questa tecnologia:

- Performance: I server hanno bisogno di un certo tempo per essere avviati, una volta che sono stati interrotti per mancato traffico. Questo può portare ad un aumento di latenza che risulta critico in determinati contesti (es. trading).
- Risorse limitate: Il provider potrebbe imporre dei limiti sulle risorse disponibili rappresentando un problema in scenari di High Processing Computing. Inoltre non tutte le funzioni sono compatibili con tutti i provider serverless.
- Minor controllo: L'infrastruttura è completamente gestita dal provider
- Monitoring e Debugging: Le metriche fornite dal provider potrebbero non essere sufficienti e il debug non praticabile
- Sicurezza: Problema della monocultura, cioè quando viene trovata una vulnerabilità nell'infrastruttura del provider, questa intacca tutti i clienti che ne fanno uso
- Vendor Lock-in: Non c'è standardizzazione tra i provider

7.3 OpenWhisk

OpenWhisk vuole evitare proprio il vendor lock-in, infatti è interamente open source (Apache license). Il modello di programmazione prevede la definizione di un trigger al quale è associata una regola che, qualora si verifichi, attiva un'azione.

Ogni azione ha delle caratteristiche abbastanza comuni:

- Le azioni devono essere **idempotenti**: $T(T(x)) = T(x)$
- Devono essere **parallelizzabili**
- Possono essere **eseguite più volte** rispetto allo stesso evento. Quindi l'ordine di esecuzione non è lo stesso dell'ordine di invocazione
- Ogni operazione deve essere **indipendente** dall'azione (non vi è uno stato interno)
- Le azioni **non sono atomiche**

Le azioni hanno un input e un output, hanno una struttura a dizionario (chiave-valore), in cui la chiave è una stringa e il valore è un JSON. Ogni azione è definita da:

- Namespace
- Package name
- Action name

Le azioni possono operare con doppia semantica:

- Richiesta/Risposta: Bloccante. Ritorna i dati solo quando l'azione è generata
- Fire and Forget: L'azione una volta terminata genera un evento (una sorta di callback) per comunicarlo

Quando lavoriamo in Fire and Forget è importante l'elemento dell'Activation Record: dove per ogni invocazione dell'azione viene memorizzato in un log:

- Activation ID
- Nome della funzione (con namespace)
- Timestamps (Inizio e fine)
- Logs
- Valore di risposta

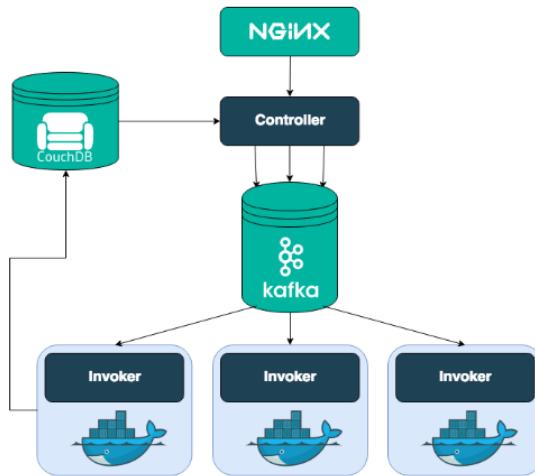
WEB actions Azioni per generare applicazioni web. Possono operare ricevendo richieste HTTP, settando headers status code e gestire i cookies. Restituiscono un body HTML. La risposta è sempre un JSON object.

Gestione delle risorse Openwhisk supporta limiti per l'utilizzo delle risorse nelle funzioni. Possiamo imporre dei limiti relativi a:

- Tempo di esecuzione
- Utilizzo della memoria
- Output size
- Numero delle esecuzioni ricorrenti
- Activation rate

I trigger sono un canale nominativo di una classe di eventi che operano su una sorgente dati. Le regole definiscono un vincolo tra i trigger e le azioni che verranno eseguite.
Model Process disponibili:

- NGINX: Gestisce le richieste in arrivo
- Kafka: Coda di messaggi
- Docker: Vengono eseguite delle funzioni
- CouchDB: Log e storage di metadati



7.4 OpenFaaS

Usato negli scenari di produzione.

7.5 AWS Lambda

Piattaforma per computing serverless. Fa parte dell'ecosistema Amazon, può interagire infatti con tutti i servizi AWS:

- DynamDB
- S3

- AWS Step: un modo per andare a definire funzioni complesse, per esempio analizzando dei dati(Ingestion - Istanziamento delle macchine di un cluster spark - Esecuzione dell'app che esegue l'analisi - Alla fine spegnere il cluster). Tutte queste operazioni di automazione di microfunzioni possono essere definite con il linguaggio AWS step. Sostanzialmente si istanzia un file JSON dove ogni passo corrisponde un'invocazione di una funzione lambda come effettuare un'azione o far partire un trigger. Strumento molto utile per l'automazione.

Prezzi

- Richieste: Costo per un 1M
- Costo per un 1GB di ram al secondo
- Free Tire: 1M di rew, 400GBxsec per month)

Esempio:

- Richiste: 3M di invocazioni: $0.2\$ \times (3-1(\text{franchigia})) = 0.4\$$
- Durata: $3M \times 0.5\text{GB} \times 1\text{sec} = 18.3\$$

Layer : Il contesto di esecuzione può essere già noto o una runtime specifica. Gli eventi possono essere:

- Custom: free format, inerenti per una determinata applicazione
- Service Event: La struttura in questo caso è legata a qualcosa che sta succedendo nella piattaforma Amazon. in questo caso sarà schematizzata con determinati tipi di indici di un JSON.

Il runtime decide quanti containers mandare in esecuzione in base alle richieste, anche qui le funzioni sono regolate da trigger. Stesso approccio arriva nelle Google Functions (supporta JS, Python, GO, Java).

7.6 Google Functions

Supportano molti tipi di eventi:

- HTTP requests
- Cloud Storage
- Pub/Sub system
- Cloud Firestore
- Firebase

Triggers Possono essere invocati tramite righe di comando o console web. Ogni evento in questo caso ha la sua struttura dati. Il modello di costo è analogo ai sistemi precedenti quindi basato sempre sul tipo di runtime e al tempo di esecuzione. Per cui la qualità e l'efficienza del codice influisce molto sul costo dell'applicazione.

8 Virtualized Storage

La virtualizzazione dello storage è necessaria per incrementare la capacità di memoria: poiché non esiste nessun disco che abbia le capacità fisiche di soddisfare le ingenti richieste dati dei datacenter, si uniscono molteplici dischi. Così facendo anche il throughput aumenta dato che molte unità possono operare in parallelo e suddividersi il carico. Infine, la virtualizzazione dello storage aumenta la disponibilità dei dati e riduce i costi.

Possiamo classificare i metodi di virtualizzazione secondo due caratteristiche:

- Granularità
- Obiettivo

8.1 Granularità della virtualizzazione

Il tipo di granularità dipende dal livello al quale avviene la virtualizzazione. Il mapping può essere fatto al di sotto del filesystem (virt. a blocchi), col quale viene offerto un dispositivo a blocchi virtuale che si occupa della loro gestione, oppure al di sopra di esso (virt. file based), dove viene semplicemente astratta la locazione fisica del file.

8.2 Block Based

Lo storage viene visto come un **pool di blocchi logici** su cui viene costruito il file system. Così facendo non è necessario che i dischi siano residenti sulla macchina ma possono essere distribuiti e remoti. Il file system spazierà quindi su diversi dischi in locazioni differenti. Occorre il supporto di quest'ultimo nella gestione di filesystems distribuiti. Tipicamente si lavora con strumenti come SCSI, un linguaggio comandi per i dischi, incapsulati tramite protocolli di rete:

- TCP/IP
- FCP (Protocolli in canali fibra)
- InfiniBand, attraverso il remote DMA support

Un blocco viene identificato tramite la coppia

- LUN: Logic Unit Number.
- LBA: Logical Block Address.

Il primo rappresenta il nome del dispositivo (es. `sda`) mentre il secondo indica 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. Occorre però tenere traccia delle **associazioni logico-fisiche**, da qui l'introduzione dei **metadati**: tabelle che contengono le associazioni LBA fisico - LBA logico. Grazie ai metadati è possibile implementare agilmente la **ridondanza** mappando lo stesso blocco logico in più blocchi fisici.

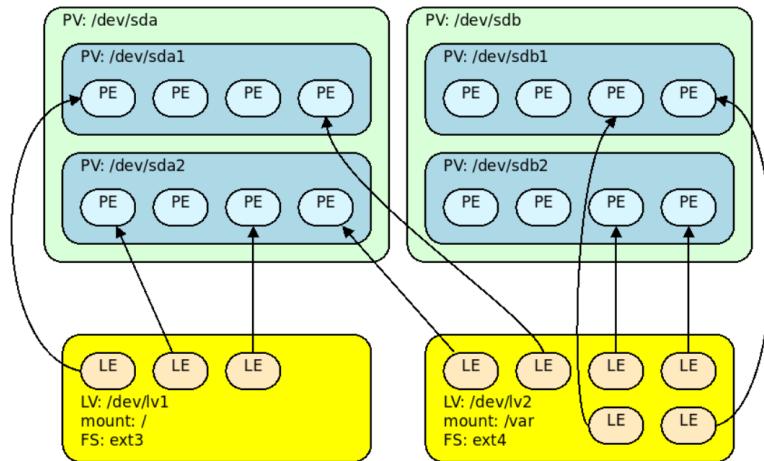
Un'altra utilità dei metadati è quella di realizzare il *load balancing* dell'utilizzo dei dischi fisici: se ho un file contiguo, posso mappare i blocchi logici contigui su **dischi fisici diversi**. In questo modo, il blocco logico contiguo viene letto, sostanzialmente, in parallelo, perché presente su dispositivi fisici diversi e quindi accessibili **parallelamente**.

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 e tenuti al sicuro.

Attraverso un sistema RAID possiamo avere delle performance maggiori, in quanto il throughput raddoppia, per cui il tempo di risposta si dimezza.

LVM Mediante la tecnica LVM, Logical Volume Management, usata in sistemi linux per mappare eventi logici in estensioni fisiche (blocchi su dischi), si costruisce un file system con vari LE (Logical Extents) collegati a dei PE (Physical Extents) distribuiti su dischi diversi, per bilanciare meglio il carico. I suoi concetti chiave sono:

- PV: Volume fisico.
- PE: Estensione fisica.
- LV: Volume logico.
- LE: Estensione logica.



E' possibile avere due approcci per accedere ai metadati:

- Accesso **trasparente**: La richiesta viene fatta per il Logical Extent e sarà compito del Software Defined Storage tradurla in PE
- Accesso ai metadati tramite **API**: I file system interrogano i metadati per un dato LE e sarà compito loro effettuare la traduzione

Se il primo approccio è meno oneroso per i file system del secondo, è penalizzato in quanto a scalabilità.

Alcune features facilmente implementabili grazie all'approccio block-based:

- Snapshotting: Attraverso il sistema a blocchi è facile implementare gli **snapshot**. Infatti, se tutti i blocchi di un file vengono marchiati come copy-on-write, le scritture successive produrranno una copia del file. Questa operazione avviene solo sui metadati e non sui blocchi effettivi del disco.
- De-duplicazione dei dati: Evitare copie dello stesso blocco, meccanismo intrinseco dello snapshotting, che possono far risparmiare ingenti quantità di memoria.

8.3 Filesystem-based

Utilizza i file come unità base di memorizzazione e sfrutta file system di rete:

- NFS
- SMB/CIFS

L'accesso alle directory e ai file avviene tramite delle API di rete che mascherano le syscalls come se avvenissero in locale. La ridondanza viene gestita internamente e non presenta la necessità di metadati per mappare LE a PE.

8.3.1 NFS

E' un sistema molto veloce basato su RPC (Remote Procedure Call). Ha visto diverse release e numerose funzionalità sono state aggiunte negli anni.

8.3.2 SMB/CIFS

Un filesystem estremamente più completo, che permette di accedere a delle risorse, con relativi protocolli, di vario genere (Stampanti, Porte, Files).

8.4 Obiettivi virtualizzazione dello storage

Con obiettivo della virtualizzazione si intende la dimensione e la visibilità dello storage che vogliamo avere. Si declina in tre principali infrastrutture:

- DAS: Identifica un disco collegato localmente ad una macchina.
- NAS: Identifica un disco visibile a più macchine.
- SAN: Identifica più dispositivi dedicati allo storage.

8.4.1 DAS

Si tratta di un disco esterno collegato a un host. Le interazioni si basano su protocolli non di rete (ad esempio il disco può essere collegato mediante una periferica USB). Quando il disco viene collegato all'host, viene preso in carico dal sistema operativo attraverso i driver. I volumi logici sono tipicamente visibili localmente ma possono essere esposti anche sulla rete.

8.4.2 NAS

Non vi è bisogno di collegare direttamente il server che deve fruire dello storage allo storage stesso perché la comunicazione 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. I componenti principali sono:

- Memorizzazione: viene effettuata attraverso dischi, SSD o HDD. E' possibile scegliere quale tipo di disco utilizzare anche in base alla metrica UBER.
UBER è una metrica per il tasso di occorrenza di errori di dati, pari al numero di errori di dati per bit letti durante l'intera vita di un SSD. Si calcola:

$$UBER = \frac{\text{number of data errors}}{\text{number of bits read}}$$

Gli SSD hanno un tasso UBER più alto di quello dei dischi, cioè una maggiore probabilità di perdita dei dati di failure.

- Controller: RAID per gestire i dischi collegati in serie. I controller RAID possono essere sia software che hardware. Nel primo caso tipicamente si usano degli standard open e i dischi sopravvivono alla rottura del controller mentre nel primo caso la rottura porterebbe alla perdita dei dati sui dischi.
- Interfacce di rete: costituite da varie schede di rete ad alta velocità capaci di supportare anche connessioni a 10Gbps.

8.4.3 SAN

Questo tipo di reti possono avere archiviazione nell'ordine degli Exabyte. Vi sono essere più livelli:

- Host Layer: Il frontend che accede ai dati, 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 di archiviazione 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.

8.5 Software defined storage

La vera sfida di questa tipologia di archiviazione è la disponibilità e l'affidabilità dei dati, i quali devono essere sempre disponibili e non possono venir persi o danneggiati.

Inoltre è richiesto che siano anche gestibili e scalabili, quindi devono essere recuperati in modo in efficiente e condivisi tra loro, ma a causa della loro dimensione il sistema deve anche essere capace di gestire il gran numero di richieste.

All'interno dello storage pool avrò dischi diversi di vendor differenti con molteplici filesystem, motivo per il quale serve un layer che astragga la complessità e mi permetta di gestire SAN estremamente **eterogenee**. Alla base vi è il concetto di hardware independence.

Le funzionalità chiave per fornire gli attributi richiesti da questo tipo di archiviazione sono:

- Automazione: cioè la gestione semplificata che riduce i costi di manutenzione dell'infrastruttura di storage.
- Interfacce standard: tramite API per gestione, provisioning e manutenzione di dispositivi e servizi di archiviazione.
- Modalità di accesso: interfacce Block, File e Object.
- Scalabilità

Alcuni esempi sono:

8.5.1 Ceph

Ceph è un filesystem che permette SDS in maniera estremamente semplice. Consente di distribuire i dati su più dischi mediante l'algoritmo CRUSH (hash-based load balancing). Da un punto di vista architettonale, 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.

Due concetti fondamentali:

- Meta Data Server distribuito con supporto al bilanciamento del carico per le operazioni di apertura file, creazione e lista directory
- Reliable Autonomic Distributed Object Store che gestisce il filesystem block based per le operazioni effettive di scrittura e lettura

9 Software Defined Networks

9.1 NFV: Network Function Virtualization

NFV 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), e utilizzare macchine con hardware più *off the shelf*. Ovviamente questo concetto non è applicabile a tutti gli scenari (IXP e backbone richiedono hardware dedicato).

Risulta 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.

9.2 SDN: Software defined network

SDN si rivolge 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 etc.

Avendo hardware proprietario, tali configurazioni cambiano per ogni vendor, è quindi necessario uno 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.

9.2.1 Motivazioni

Attraverso la definizione di reti definite dal software, si cerca di soddisfare alcuni requisiti fondamentali:

- Condivisione delle risorse.
- Isolamento delle prestazioni
- Supporto alla ridondanza
- Riduzione dei costi OPEX e CAPEX tramite dispositivi hardware off-the-shelf e non più dedicati, quindi costosi.

L'obiettivo critico è quello di adattarsi a scenari che possono cambiare in maniera molto dinamica, e in caso di problemi il sistema cloud debba gestirli repentinamente in modo automatico.

Alcune aree critiche rimangono:

- Interazione controller: pacchetti inviati con grande latenza.
- Problemi di prestazioni del piano dati: tabelle grandi e predici complessi.
- Scalabilità del controller: può diventare un collo di bottiglia e l'utilizzo di più controller necessita l'implementazione di una coordinazione.
- Ridondanza del controller.

Per capire meglio come funzionano le SDN occorre definire due elementi principali:

- **Data Plane**: Dove avviene la manipolazione dei pacchetti. Nel caso di un router è la parte che si occupa del forwarding dei pacchetti
- **Control Plane**: Dove vengono prese le decisioni per la gestione dei pacchetti. Nel caso di un router è la parte che si occupa dell'instradamento dei pacchetti

Il Software Defined Networking (SDN) è un'architettura per la realizzazione di reti di telecomunicazioni nella quale il piano di controllo della rete e quello di trasporto dei dati sono separati logicamente.

Questa separazione logica permette da un lato la possibilità di gestire via software tutta la rete da un unico controller, garantendo così una maggiore scalabilità e standard di affidabilità e sicurezza della rete più elevati, dall'altro quella di utilizzare indifferentemente apparati prodotti dalle diverse

aziende che non conteranno più al loro interno le funzioni di gestione favorendo, così, la nascita di una rete dinamica e non più legata al larghissimo numero di protocolli differenti attualmente utilizzati.

Il control plane, cioè i controller SDN, comunicano con gli applicativi tramite la **northbound API**, tipicamente REST o BGP, OSPF per sistemi legacy. La comunicazione con il data plane avviene invece mediante la **southbound API** di cui OpenFlow rappresenta lo standard de facto. Ad esempio il controller ascolta le modifiche alle macchine virtuali sulla northbound API e procede ad inoltrare i comandi tramite la southbound API agli apparati di rete.

SDN fornisce alla NFV i vantaggi di una connessione programmabile tra le funzioni di rete virtualizzate; la NFV, invece, mette a disposizione dell'SDN la possibilità di implementare le funzioni di rete tramite software su server COTS (Commercial off-the-shelf). Si ha, così, la possibilità di virtualizzare il controller SDN implementandolo su di un cloud che può essere facilmente migrato in qualsiasi posizione in base alle esigenze della rete.

9.2.2 Switch SDN

Uno switch SDN è un programma software o un dispositivo hardware che inoltra i pacchetti in un ambiente SDN. Deve supportare i protocolli SDN. Il primo e il più noto protocollo SDN è OpenFlow. Pertanto, uno switch SDN viene spesso chiamato anche switch OpenFlow. Esistono anche altri protocolli SDN sviluppati dai fornitori di switch SDN, come OpFlex, NETCONF, BGP (Border Gateway Protocol), XMPP (Extensible Messaging and Presence Protocol), ecc.

Funzionamento Il piano di controllo (routing di alto livello) è disaccoppiato dall'hardware dello switch SDN, ma è implementato nel controller SDN (un'applicazione in esecuzione sul server o da qualche parte), che si trova tra i dispositivi di rete e le applicazioni. Ogni switch SDN nel modello SDN è programmabile dal controller SDN attraverso i protocolli SDN.

Uno switch SDN è composto da **porte** e **tabelle**. I pacchetti arrivano e lasciano lo switch attraverso le porte. Le tabelle sono costituite da righe contenenti un **classificatore** e un insieme di azioni. Quando uno switch SDN riceve un pacchetto che non ha una riga corrispondente nella tabella, comunica con il controller SDN e chiede cosa fare con questo pacchetto. Il controller può scaricare un flusso sullo switch, che include il primo classificatore che meglio corrisponde al pacchetto e le azioni.

Le **azioni** gestiscono il trattamento del pacchetto, che può essere l'inoltro alle porte, l'incapsulamento e l'inoltro al controllore, l'eliminazione del pacchetto o l'invio alla normale pipeline di elaborazione. Una volta che i predicati sono stati scaricati nella tabella degli switch, questi vengono processati in autonomia.

Il predicato del matching può usare molti campi per stabilire la regola: può agire su campi che vanno dal livello più basso dello stack TCP/IP fino ai protocolli di trasporto. Uno switch SDN può, quindi, gestire qualsiasi livello dello stack, rendendolo molto efficace.

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

- Controller to switch: Messaggi di controllo e configurazione
- Messaggi asincroni da switch a controller: Messaggi inviati quando è necessario l'intervento del control plane
- Messaggi simmetrici: Messaggi usati per controllo della salute dei dispositivi

9.3 Vantaggi SDN

Un vantaggio generale dell'uso degli switch SDN è la facilità di controllo e configurazione del flusso. Con gli switch SDN, non è necessario recarsi presso le postazioni dello switch e accedere alla riga di comando per configurarlo. È possibile controllare e programmare in remoto più switch attraverso un unico controller SDN che utilizza il protocollo SDN e fornisce API per gli switch SDN. Inoltre, sarà più facile effettuare il bilanciamento del carico anche a velocità di dati elevate e il traffico potrà essere isolato senza bisogno di VLAN, poiché il controller SDN permette di rifiutare determinate connessioni.

10 Software Defined Data Center

L'applicazione dei principi di virtualizzazione ai data center nasce dall'esigenza di avere un sistema elastico e flessibile.

10.1 Punti di Vista

Ci sono diversi aspetti che un cloud provider deve tenere in considerazione, ciascuno di questi appartenente a punti di vista differenti:

Utente

- Un utente deve avere la percezione di avere una macchina per se
- La scalabilità del sistema messa a disposizione dell'utente deve essere istantanea: in pochi istanti deve essere possibile per l'utente avere a disposizione altre macchine
- Il sistema deve garantire una certa sicurezza: sui data center possono essere messi dati sensibili, pagando si deve essere certi che un team di persone competenti (con privilegi diversi) mantengano la piattaforma sicura
- Il servizio offerto deve essere affidabile: le risorse sempre disponibili (ridondanza dei dati) e il downtime di pochi minuti all'anno

Ingegneristico L'infrastruttura deve essere gestita sotto gli aspetti software e hardware, in particolare in termini di consumo energetico, raffreddamento e sicurezza (personale che ha accesso fisico ai sistemi).

- Software: È necessario un sistema di monitoraggio per conoscere le macchine accese e le risorse utilizzate, in modo da sapere quanto far pagare i servizi.
- Hardware: I nodi di calcolo (host) vengono raggruppati in racks che possono essere raggruppati in pods, dei container di racks (generalmente spostati usando dei camion) che presentano tre cavi: per la rete, energia e per il raffreddamento. I pods possono essere visti anche come dei mini data center, e rappresentano un'alternativa alla struttura gerarchica classica del data center tradizionale.

10.1.1 Data Center Cloud

Oggigiorno molti data center stanno migrando verso la struttura cloud. Ci si aspetta che un data center cloud rispetti determinati criteri e caratteristiche:

- Conveniente
- Scalabile
- Realizzabile
- Sicuro
- Elastico
- Internet-based

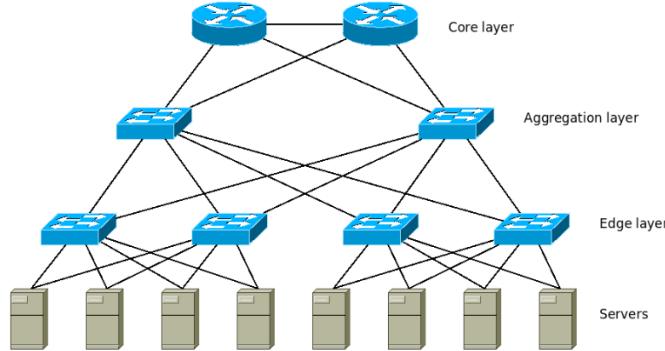
10.2 Infrastruttura

Il data center viene organizzato in **blade systems**, composti da un certo numero (solitamente 4) di blade (l'unità di computazione vera e propria) inserite all'interno di uno **chassis**. Se nella blade troviamo componenti come RAM, CPU e hardware di rete, nello chassis si trova l'alimentazione, il sistema di raffreddamento e quello di monitoraggio. I blade systems vengono organizzati in **racks**, delle sorte di armadi che presentano degli switch utilizzati per l'interconnessione di più blade systems tra loro e il resto del datacenter. Nei racks sono presenti due tipi di switch:

- top-of-rack: per la connessione degli elementi appartenenti allo stesso rack
- end-of-row: per la connessione di racks sulla stessa fila

Le blades montano sistemi di storage di piccole dimensioni NVRAM o SSD (generalmente intorno ai 500Gb, 1Tb) utilizzati per il caching dei dati e per far girare l'hypervisor, mentre l'unità di storage vera e propria dei data center consiste in una Storage Area Network (SAN), uno storage persistente che mantiene le copie delle immagini delle VM.

10.2.1 Data Center Network



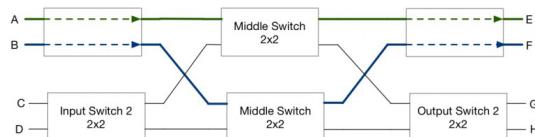
Struttura La rete di un datacenter presenta una topologia ad albero nella quale è possibile individuare diversi livelli (dal basso all'alto):

- Servers: si tratta dei **blade systems**, rappresentano le foglie dell'albero e sono interconnessi tra loro mediante dei collegamenti da 1-10Gbps.
- Edge Layer: anche chiamato Access Layer, è composto da **switch top-of-rack** ed eventualmente da quelli di tipo end-of-row. In questo layer i blade systems vengono connessi e vengono implementate le funzioni di isolamento del traffico e di *gestione dei guasti*.
- Aggregation Layer: su questo livello vengono interconnesse **zone separate** dei datacenter (interconnessione di pods). Il layer implementa funzionalità di QoS per i flussi dati (è possibile anche rallentarne alcuni in casi di sovraccarico). Traffic shaping in caso di overload.
- Core Layer: in questo layer si lavora esclusivamente a livello di rete. Gestisce tutte le **connessioni con l'esterno** e l'instradamento in caso di failures (quando alcuni link non sono disponibili), inoltre interagisce con la SAN. Il livello di disponibilità delle risorse è estremamente importante, serve infatti un'ottima politica di gestione in caso di disponibilità critica.

Limiti della Topologia ad Albero Il principale problema di questa topologia sta nei **layer superiori** (aggregation e core), che possono diventare dei colli di bottiglia o essere soggetti a failures. La soluzione consiste nell'utilizzo della replicazione e nella **riprogettazione della topologia**. Con una nuova topologia di tipo Fat Tree vengono messi a disposizione più link nella **parte superiore della rete**, ottenendo una banda maggiore, e quindi maggiore scalabilità e **tolleranza ai guasti**. La configurazione di rete utilizzata per ottenere questi risultati si chiama CLOS Network.

CLOS Network Si considera il datacenter come un grafo, dove i nodi sono i server/switch e gli archi tra i nodi le connessioni. L'obiettivo di una rete di questo tipo è quello di minimizzare il numero di percorsi bloccati (in uso), minimizzando il numero di connessioni. Prima dell'introduzione della rete Clos, il numero di punti di incrocio doveva essere uguale al numero di ingressi moltiplicato per il numero di uscite. Questo approccio, noto come *n-squared* o n^2 , poteva portare rapidamente a un numero enorme di punti percorsi, con conseguenti costi significativi.

Nelle reti Clos gli switch sono stati organizzati in un'architettura a tre stadi che comprende uno



stadio intermedio inserito tra gli stadi di ingresso (input) e di uscita (output). In questo modo, ciascun output degli switch di input si collega ad uno switch intermedio, i cui output a loro volta, si collegano ad uno switch di uscita. Il risultato è una topologia **non bloccante** che richiede meno punti di incrocio rispetto alle reti di switch più convenzionali dell'epoca. Grazie alle reti Clos è possibile **ri-organizzare** la rete per trovare un percorso che prima sembrava inaccessibile o bloccato.

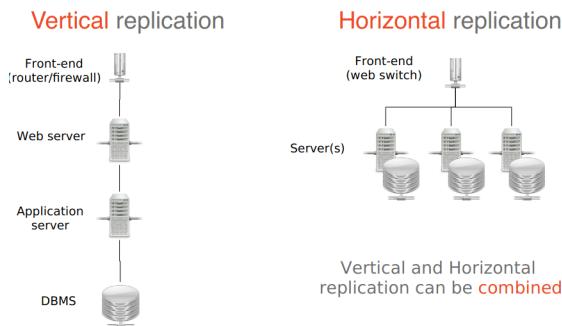
Le reti Clos sono un caso particolare delle più generali reti Benes.

Google Data Center - Scenario La crescita del traffico per i data center Google segue la legge di Moore, raddoppiando ogni 12/15 mesi. Questo aumento ha come conseguenza un *aumento del numero di elementi di rete da interconnettere*. Nel 2004 il rapporto tra la banda gestibile dal dispositivo e quella entrante era di 10:1. È in scenari simili che l'adozione di una CLOS Network fa la differenza.

10.2.2 Scalabilità

La scalabilità rappresenta il problema principale di un datacenter. Quando si parla di scalabilità questa può essere verticale, si scala facendo un upgrade degli apparati hardware/software oppure orizzontale, basata sulla replicazione. Tra i due la *scelta migliore*, nonché la più economica è quella della replicazione orizzontale, che permette anche di fare affidamento sulla generazione precedente, con un 10% in meno di performance su dispositivi che costano la metà. Scelto il tipo di scalabilità rimane da decidere *come replicare*. Anche per la replicazione sono disponibili due metodi:

- Replicazione **verticale**: dove un servizio complesso viene suddiviso in sottoservizi ciascuno messo su un dispositivo diverso.
- Replicazione **orizzontale**: che consiste nella replicazione di oggetti identici



Architettura di un Web Cluster In generale, l'architettura tipica di un Web Cluster fa uso di entrambi i metodi di replicazione e si divide su due livelli:

- Front-end: costituito dal web switch *visibile all'esterno come un unico indirizzo IP pubblico*, detto **Virtual IP**. Il ruolo del web switch nell'architettura è quello di direttore d'orchestra. Può essere implementato a tutti i livelli (hardware, kernel-level, applicativo). Deve avere delle funzionalità per il bilanciamento del traffico, e per questo si hanno due opzioni:
 - Switch di Livello 4 (Trasporto): vengono utilizzate delle **binding tables** (strutture dati) per tenere traccia delle connessioni TCP. Il 3-way-handshake viene mediato dallo switch e poi portato avanti dal backend server a cui è diretto il pacchetto.
 - Switch di Livello 7 (Applicativo): si tratta di nodi più intelligenti che lavorano in termini richiesta, e possono tenere traccia delle **sессии utente**. Il 3-way-handshake viene gestito interamente dallo switch, che deve conoscere la richiesta per sapere dove inoltrarla.

Non c'è una soluzione che sia migliore dell'altra. Possono essere adottate entrambe in base al contenuto del traffico.

Esempio: Instagram. La maggior parte del contenuto del contenuto è **statico** (immagini), pertanto si possono usare switch di livello 4. Per tutte quelle funzionalità che utilizzano delle API per, ad esempio, consigliare post, chi seguire, è possibile utilizzare switch di livello 7.

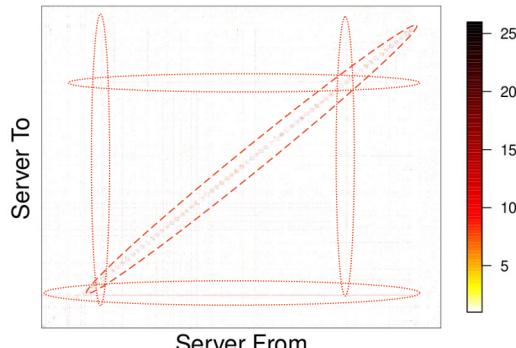
- Backend: insieme di server configurati con **IP privati** che utilizzano il front-end per le comunicazioni con l'esterno

Limits I limiti della scalabilità sono strettamente collegati alla consistenza dei dati: le repliche devono essere mantenute aggiornate. I colli di bottiglia possono verificarsi sia negli switch che nella rete dei datacenter. Inoltre, è opportuno considerare la crescita dei datacenter anche in relazione al consumo energetico che richiedono.

10.2.3 Traffico

Quando si parla di traffico di un datacenter si deve far riferimento a piccoli gruppi di cluster interconnessi. Si possono identificare dei **pattern** nella comunicazione tra i nodi di un cluster. Infatti, i nodi di un rack tipicamente comunicano con i nodi dello stesso rack, motivo per il quale vanno messi vicini anche se non necessariamente sulla stessa macchina fisica, poiché potrebbero entrare in competizione per le risorse causando un **overload** e di conseguenza un calo di prestazioni. D'altro canto non possono neanche essere lontani per evitare di spostare grandi **quantità di traffico**.

Vi è anche un altro pattern riconoscibile: alcuni nodi sono particolarmente propensi a ricevere da tutti gli altri nodi (e collateralmente ad inviare). Questi nodi devono essere messi in una posizione **baricentrica** all'interno del datacenter.



Inoltre, è opportuno spostare tutte quelle operazioni onerose che occupano traffico non utile, come i backup, durante le ore notturne, quando è meno probabile che il server debba soddisfare le richieste degli utenti.

10.2.4 Gestione del Data Center

Il posizionamento e la migrazione delle macchine virtuali possono essere risolutivi per diversi problemi:

- Bilanciamento del carico
- Consumo energetico: **consolidare** server idle può aiutare a risparmiare energia
- Utilizzo della rete: le VM possono cambiare i loro pattern di carico di lavoro, analizzando i dataset differenti e le differenti interazioni tra i pattern. Non è necessario re-deployare la VM se si preserva la località della rete.
- Gestione del calore: bisogna evitare i cosiddetti **hotspot**, ovvero le concentrazioni di server ad alta utilizzazione e cioè ad alta produzione di calore. Le macchine negli hotspot lavoreranno a maggiore temperatura e quindi saranno sottoposte a maggiore usura. Inoltre se alcune macchine non vengono utilizzate è possibile spegnere l'impianto di condizionamento per tale area di modo da risparmiare notevole energia.

Reti Esterne Vi sono altri problematiche che vanno oltre quelle del datacenter e riguardano l'instradamento di traffico fra più datacenter per applicazioni distribuite. I possibili punti di congestione sono:

- *last mile*: dall'uscita della rete al client finale
- *first mile*: dal datacenter all'ingresso della rete
- *peering point*: punti di scambio fra AS

La congestione non è sempre un problema *vicino al datacenter*. Spesso può essere causata dall'ultimo miglio (lato client) ma in questo caso il cloud provider non può agire per ovvi motivi. Dove può agire è sul *first mile* e sul *peering*, anche se tipicamente gli AS non hanno problemi di banda. Possibili soluzioni alla congestione possono essere:

- Aumento della connettività del data center

- Replicazione dei service point
- **Gestione del data center e delle reti che li interconnettono**

L'infrastruttura di Google Cloud si sviluppa su tre layer concentrici:

- Al centro ha i suoi **datacenter** distribuiti nelle località dove Google stessa offre i suoi servizi
- Nell'anello successivo sono presenti i **Point of Presence** che si occupano di instadare i pacchetti preferibilmente all'interno della propria rete (*cold potato policy*). Così facendo è possibile ottimizzare il traffico e ricorrere agli ISP solo per l'ultimo miglio.
- Nell'ultimo anello invece sono localizzati i migliaia di **edge nodes** che formano la CDN di Google. Questi sono i più numerosi e i più replicati e si occupano di fornire i contenuti statici. Questi nodi sono generalmente posizionati all'interno della AS di cui fa parte l'utente finale per minimizzare gli hop necessari.

10.3 Software Defined Data Center

In un data center tradizionale c'è una netta distinzione tra il livello di computazione (*gestione delle VM*) e quello di comunicazione (rete). In un Software Defined Data Center (SDDC) questi due livelli si uniscono formando un unico livello. Non c'è più una visione di una singola VM ma di reti virtuali, composte da Macchine Virtuali (VM) e Virtual Router (VR). Le funzionalità di rete e i suoi apparati vengono implementati via software e via hardware (general purpose).

10.3.1 Inibitori SDDC

Ci sono diversi aspetti elementi da considerare per sfruttare appieno i vantaggi offerti da un SDDC:

- Gestione della rete statica: occorre rimuovere le VLAN. Ogni piccolo cambiamento nelle VLAN comporta una riconfigurazione, un dettaglio non trascurabile in uno scenario dinamico dovuto alle continue migrazioni delle macchine. Poiché le VLAN sono complesse e possono introdurre errori, bisogna mapparle in modo intelligente sul layout fisico del datacenter (le macchine parlando con un *sottoinsieme di nodi*).
- VLAN: Rete locale divisa in *più reti locali logicamente non comunicanti tra loro* ma che *condividono la stessa infrastruttura fisica*.
- Frammentazione delle risorse: a causa dell'unificazione del livello computazionale con quello di comunicazione bisogna garantire un corretto load balancing tra tutte le macchine in gioco. Allo stesso tempo però è opportuno che alcune macchine siano localmente vicine andando contro al principio di distribuzione del carico che le vorrebbe distribuite su più nodi.
- Peer 2 Peer poor connectivity: bisogna evitare il problema dell'**oversubscription**: ogni volta che si dispone di uno switch di accesso in cui la larghezza di banda totale possibile combinata di tutte le connessioni client supera la larghezza di banda upstream dallo switch di accesso al core, si è in presenza di un oversubscription. Inoltre, se la connettività host to network è particolarmente efficace e poco costosa, quella IP-based dei SDD è più costosa e tipicamente più lenta. Occorre avere quindi una connettività nell'ordine dei gigabit anche localmente.

10.3.2 Soluzioni

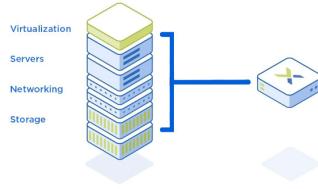
Vengono proposti diversi approcci per venire in contro ai requisiti di un SDDC.

Iperconvergenza È un tentativo di condensare in un livello di intelligenza unico tutte le funzioni di gestione del datacenter. L'Iperconvergenza è ampiamente utilizzata in contesti cloud e consiste in un sistema hardware verticalizzato, al cui interno integra gli elementi di computazione, storage e networking.

Sono a tutti gli effetti dei **piccoli datacenter** già testati in modo da non risultare colli di bottiglia. Sistemi di questo tipo si auto configurano, comportando una riduzione dei costi CAPEX e OPEX e un deployment efficiente.

Sono possibili diversi tipi di architettura di iperconvergenza:

- Sistemi stack integrati: Server+Storage+Network+Application software già presente al proprio interno



- Integrated infrastructure systems: Server+Storage+Network+Virtualization se si vuole un'infrastruttura di computazione generica. Rappresentano i blocchi sui cui viene costruito il cloud
- In combinazione con il *fabric computing*

Riduzione della Topologia di Rete - Fabric Computing Consiste nella virtualizzazione delle risorse di rete in modo da avere una visione semplificata. Gli elementi di switching multipli vengono condensati su un unico elemento di switching. L'idea è quella di avere una rete *mesh* che interconnette tra loro dispositivi di rete e di storage virtualizzando entrambi. Tipicamente viene realizzato mediante le Software Defined Networks. Riducendo tutto ad una mesh la comunicazione tra macchine è particolarmente veloce in quanto vicine e non occorre più preoccuparsi della topologia della rete. Inoltre, è possibile scalare la rete e riconfigurarla efficacemente in base alle necessità.

Topi ed Elefanti Sfrutta le caratteristiche del traffico di rete **non uniforme** nei data center. La terminologia "topi" (breve) ed "elefanti" (lungo) per la durata del flusso fornisce una metafora per il traffico web. Ci sono relativamente pochi elefanti e un gran numero di topi. Può accadere quindi che vi siano link sovraccaricati e altri sottoutilizzati. Si utilizzano piccoli set di link veloci riconfigurabili dinamicamente in base alla richiesta. In questo modo flussi più grandi verranno dirottati su determinati link mentre flussi più piccoli su altri. Tipicamente vengono utilizzati collegamenti **ottici** con switch ottico-meccanici.

10.4 Progettazione di un Data Center

10.4.1 Posizione

Il posizionamento di un data center è un fattore importante da tenere in considerazione. In generale è bene scegliere zone economiche, non soggette a disastri naturali (in America le zone vicino a Las Vegas sono sicure, ma non economiche), lontane da impianti chimici o nucleari e che non siano obiettivi militari.

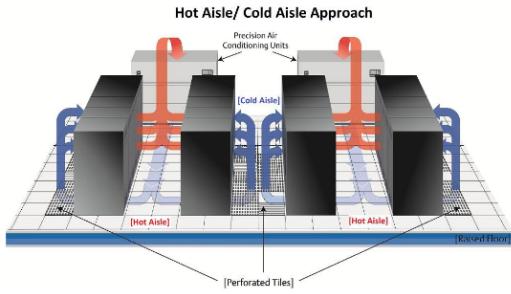
Quando si sceglie dove costruire un data center è importante guardare i costi delle risorse come l'energia, le tasse e il costo delle risorse umane. Anche questo mercato viene mosso dalle tendenze che fanno lievitare i prezzi. Anche fattori come il clima e la logistica (autostrade, trasporto) sono da considerare. Inoltre, la vicinanza all'utilizzatore è un aspetto da tenere in conto, in quanto contribuisce a un miglioramento del tempo di risposta dei servizi.

10.4.2 Sicurezza dei Data Center

Il fattore umano è la maggiore causa di incidenti. Occorre fare un analisi del comportamento dei dipendenti, prima dell'assunzione e durante il periodo lavorativo. Inoltre il personale richiede una certa formazione, ed è necessaria una separazione in ruoli. L'accesso fisico alle strutture deve essere vietato: i data center richiedono entrate controllate con sistemi di videosorveglianza e guardie di sicurezza. Qualora dovesse essere concesso l'accesso questo deve avvenire mediante dei badge o dei sistemi di autenticazione con dati biometrici. Ciascun dipendente deve disporre dei privilegi di accesso minimi per poter svolgere il proprio lavoro.

10.4.3 Raffreddamento

Il raffreddamento rappresenta il costo maggiore per la manutenzione di un data center. La tecnologia più utilizzata per quest'aspetto consiste nell'utilizzo delle **Computer Room Air Conditioner** (CRAC), ovvero di stanze con condizionatori enormi. Questi includono un sistema di raffreddamento e di filtraggio della polvere (che rende meno efficiente la dissipazione del calore). Il flusso dell'aria viene gestito in modo da avere dell'aria fredda che circola al di sotto del pavimento e che



fuoriesce attraverso delle piastrelle perforate.

I server vengono messi tutti rivolti nella stessa direzione e si cerca di creare un sistema di corridoi alternati di aria fredda e di aria calda. Per evitare la dissipazione del calore dai corridoi caldi possono anche essere utilizzati delle tettoie o tende poste sopra ai racks. Il riscaldamento è un aspetto molto importante perché ha un ***impatto importante sulla vita dei componenti***. Data la loro importanza anche i sistemi di raffreddamento devono essere ridondanti. Inoltre, deve esserci un protocollo di emergenza nei casi in cui uno o più condizionatori si spengono.

Google e Facebook: Per i due colossi, il raffreddamento rappresenta il costo energetico più impattante nei loro data center. Hanno cominciato a costruire data center in zone con un clima freddo, in modo da poter utilizzare l'aria esterna e risparmiare sui costi di raffreddamento.

10.4.4 Energia

L'enorme richiesta di energia dei data center non è dovuta solo alla potenza computazionale ma anche al raffreddamento. L'energia richiesta dai DC è una fetta consistente del consumo globale. Sono necessari generatori di corrente nell'ordine delle decine di MW con gruppi di continuità a gasolio per i cali di corrente, sistemi di raffreddamento ridondanti etc.

11 Gestione dei Data Center

La gestione del data center è un problema complesso per la quantità di bisogni in gioco, specie alcuni in conflitto tra loro. In generale, si deve cercare di *ridurre i costi massimizzando il profitto*, il tutto cercando di garantire la SLA (*Service Level Agreement*, requisiti di throughput e tempo di risposta che devono essere rispettati dal Service Provider nei confronti dei propri clienti). Il problema della gestione può essere visto come un **problema di ottimizzazione**, nel quale bisogna tenere conto di diverse metriche di performance:

- Key Performance Indicator (KPI): le metriche di performance del sistema
- Vincoli: solitamente requisiti aggiuntivi del sistema (es. tempi di servizio)

11.1 Metriche di Performance

Per descrivere le performance di un data center si possono considerare diversi aspetti:

- Consumo Energetico
- SLA
- Failure Rate
- Utilizzo delle Risorse
- Affidabilità / Disponibilità

11.1.1 Consumo Energetico

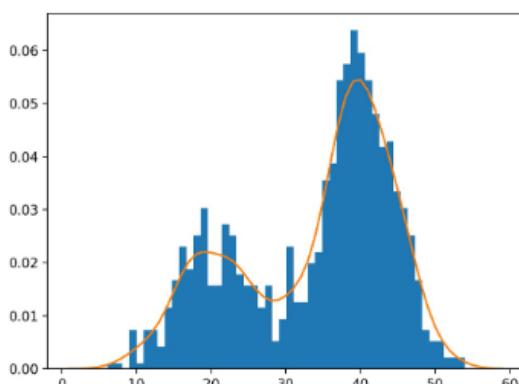
Per considerare il consumo energetico di cui ha bisogno un data center è possibile pensare ai singole componenti del consumo: computazione, networking, storage, e cooling. La nuova frontiera tende verso i **green data center**: siccome i data center sono strutture estremamente affamate di risorse, sarebbe ideale limitare il consumo di brown energy (non rinnovabile), promuovendo il consumo di energia green, quindi generata da fonti rinnovabili.

11.1.2 Service Level Agreement (SLA)

Il SLA fa riferimento ai requisiti contrattuali che il sistema deve rispettare nei confronti del cliente. I requisiti si esprimono solitamente sotto forma di **tempi di risposta, throughput e disponibilità** del sistema.

Per rappresentare il tempo di risposta del sistema è necessario prendere in considerazione un intervallo di tempo sensato: una finestra di 24h non avrebbe senso se il periodo per gran parte del tempo durante la giornata è inattivo, per questo viene solitamente calcolato in una finestra di pochi minuti. Il tempo di risposta inoltre non può essere preso dai log, che tengono conto anche delle *tempistiche lato client*.

Trattandosi di una variabile aleatoria, rappresentata come valore medio o 90-percentile, è necessario utilizzare una funzione cumulativa di probabilità (integrale della densità di probabilità): ponendo sull'asse delle X i valori (istanti di tempo) e sull'asse delle Y la 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. Il SLA talvolta può essere garantito all'interno di certe soglie di carico. All'aumentare del carico del sistema aumenta la probabilità di accodamento delle richieste.

11.1.3 Utilizzo delle Risorse

Utilizzo della CPU La variazione del carico ha un notevole impatto anche sul funzionamento della CPU e in particolar modo sulla sua temperatura. L'improvvisa accensione e spegnimento di un server, può causare uno **shock termico**, che va a accorciare la vita dei componenti.

Il principale parametro per stimare la riduzione di vita dei componenti è il fattore di accelerazione AF che dipende dal gradiente di temperatura e dal numero di cicli di potenza.

$$AF = \left(\frac{f_{STD}}{f_{SM}}\right)^{-m} \left(\frac{\delta_{STD}}{\delta_{SM}}\right)^{-n} \left(e^{\frac{E_a}{K}(1/T_{STD}^{max}-1/T_{SM}^{max})}\right)$$

L'utilizzo della CPU è un parametro da tenere sotto controllo per le performance del sistema, ed è possibile farlo guardando *gli istanti di tempo che questa trascorre in stato IDLE*, ma anche considerandone il carico, considerando il numero di processi in attesa di esecuzione che può scegliere lo scheduler.

La probabilità di un server di essere occupato si esprime come $\rho = \frac{\lambda}{\mu}$. Dove λ è frequenza di arrivo delle richieste (tasso di arrivo), mentre μ è il frequenza con cui le richieste vengono soddisfatte (tasso di servizio).

Utilizzo della Rete È possibile studiarne l'utilizzo in maniera analoga a quanto visto con la CPU, guardando l'**utilizzo dei canali di rete**, facendo però attenzione al fatto che la banda teorica dei dispositivi non può mai essere raggiunta. Tipicamente, infatti, oltre ad un certo tasso di utilizzazione aumentare il throughput provoca solamente più collisioni, con esiti deleteri per il throughput.

Le due metriche per misurare l'utilizzazione sono il **throughput** (dati trasmessi nel periodo di tempo) e la **bandwidth** (massimo dei dati che è possibile trasmettere).

11.2 Modelli di Performance

11.2.1 Modello Energetico

Con un modello energetico è possibile misurare il *consumo energetico* in un sistema. Nel caso più semplice si considera una **dipendenza lineare** tra l'utilizzazione della CPU e la dissipazione di calore.

Concentrandosi sul consumo energetico della CPU si può adottare il modello **Dynamic Voltage Frequency Scaling** (DVFS), dove l'energia spesa varia con un fattore cubico rispetto alla frequenza del processore. Ulteriori modelli di questo tipo possono basarsi sull'utilizzo del disco o sul trasferimento dei dati (utilizzo della rete).

Se gli apparati di rete dedicati hanno costi fissi in termini di consumo energetico non si può dire lo stesso per le reti software defined, dove il traffico incide profondamente sul consumo energetico.

11.2.2 Modelli Basati sull'Affidabilità o Disponibilità

L'affidabilità e la disponibilità del sistema sono solitamente usati come vincoli di un modello. Bisogna tenere conto di come sono collegati i sistemi: se in serie o in parallelo, del loro comportamento in caso di failure e del loro livello di replicazione. Inoltre, è necessario distinguere se si utilizza un singolo data center o molteplici data center, scelta che comporta problemi differenti.

Il problema di gestione di più datacenter si articola su due layer:

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

Nel caso di più datacenter bisogna capire come **distribuire il carico** efficacemente su tutti i DC, definire dei **predittori** del workload futuro, far fronte alla **località delle richieste** e gestire i molteplici costi delle VM nei DC. Nel secondo caso invece l'attenzione è sul singolo DC e su come **scalarlo**, reagire agli **errori di predizione** e ai **burst di traffico**, **allocare** le VM necessarie e infine gestire la **sincronizzazione** tra i vari DC.

11.2.3 Modelli Economici

Se un Provider deve pensare a **massimizzare** il ricavo, un cliente dal canto suo deve **minimizzare** la spesa. Un provider tipicamente deve chiedersi come agire nel caso di alto traffico (se prendere in prestito macchine da altri provider, se rivedere il tipo di VM che usa), nel caso di sotto utilizzo dei server (se vendere potenza computazionale con macchine Spot e quanto sia conveniente rispetto al solo tenerle spente). Il problema principale è, però, quello di definire i **modelli di prezzo** delle varie VM per massimizzarne il ritorno.

Queste macchine sono:

- Istanze di VM Dedicate: macchine economiche, il cui utilizzo si paga anticipo per essere utilizzate mesi o anni, sempre disponibili
- Istanze di VM On Demand: costi alti ($3 \times$ quelle dedicate), il cui utilizzo si paga per un breve periodo (ore), sempre disponibili
- Spot VM: macchine generalmente vendute all'asta con l'intento di fare un profitto (anche se minimo), anche queste pagate per un breve termine, possono essere indisponibili. Il tempo in cui la macchina non è disponibile viene rimborsato

11.3 Variabili Decisionali

11.3.1 Posizionamento delle VM

Il Problema del posizionamento delle VM può essere visto come un *bin packing problem* mono-, considerando solo l'utilizzazione CPU, o multi-dimensionale, considerando anche l'utilizzazione delle altre risorse (RAM, CPU, ecc...).

Inoltre, è necessario fare analisi su più istanti temporali in base all'utilizzazione delle risorse che vengono assegnate alle VM. Poiché il problema del bin packing esplode all'aumentare del numero di VM, è possibile semplificare non guardando in termini di numero di VM ma di **classi di servizio**.

Categorizzando le macchine in base a determinate alle classi di risorse maggiormente utilizzate si possono ridurre drasticamente le dimensioni del problema a scapito della precisione. Il trade-off è necessario per datacenter che vogliono eseguire predizioni sul traffico in tempi computazionali accettabili.

Si vuole diminuire il numero di server accesi, dove la formula:

$$\min \sum_{s \in S} O_s$$

rappresenta il vettore booleano che indica se il server è acceso o spento, mentre:

$$\sum_{s \in S} I_{m,s} = 1$$

rappresenta un vettore booleano che ha valore 1 se e solo la macchina m è allocata sul server s . Una macchina può trovarsi su un solo server, quindi deve per forza essere 1. Si ha che le risorse utilizzate dal server devono essere minori delle risorse disponibili sul server ($V_{r,s}$) quando questo è acceso (O_s):

$$\forall s, \forall r, \forall t \quad \sum R_i \cdot r(t) \cdot I_{m,s} < V_{r,s} \cdot O_s$$

11.3.2 Migrazione delle VM

La migrazione consiste nello spostamento di una VM da un server a un altro, e si verifica solitamente quando i server vengono sotto-utilizzati ($<= 20\%$): questi vengono spenti e le macchine spostate su altri server più attivi, in modo da poter spegnere i primi risparmiando risorse ed energia (**server consolidation**). Queste variabile vengono identificate con i termini $g_{m,s}^-$ quando una *macchina esce da un server*, $g_{m,s}^+$ quando entra.

11.3.3 Ottimizzazione della Rete

È necessario mappare il modo in cui le macchine comunicano tra loro in maniera ottimale all'interno della rete. Identificando le macchine **affini** sulla base dello **scambio di dati** che avviene fra di

loro è possibile definire quali devono stare vicine le une alle altre (stesso rack o cluster). Gli altri vincoli, sempre presenti, riguardano le risorse sul server per evitare il sovraccarico e la garanzia di disponibilità.

E' possibile concentrarsi anche sul consumo di energia, l'utilizzo della banda evitando colli di bottiglia e minimizzare il numero di hops in modo da rendere la rete quanto più compatta possibile.

11.3.4 Orchestrazione

Dal punto di vista dell'utilizzatore Cloud, il Deployment del sistema deve considerare i vari microservizi, il carico atteso su ciascuno di essi, i requisiti SLA e da decidere quale provider è in grado di far fronte a questi requisiti, sempre con l'obiettivo di **minimizzare** i costi. Il deployment ottimale viene stabilito in base a il numero e il tipo di VM per ciascun servizio, tenendo conto dell'evoluzione del suo carico in base alle richieste e ai vincoli SLA.

11.4 Soluzioni in Ambito Industriale e di Ricerca

11.4.1 Soluzione Basata sull'Utilizzo

Si tratta della "regola del pollice" in ambito cloud: viene utilizzata per la server consolidation, in modo da evitare server sovraccarichi e server sottoutilizzati, viene anche utilizzata per le decisioni di scaling e può essere affiancata da algoritmi di predizione del carico.

Si utilizzano due soglie, indicando con U l'utilizzo del server:

- $U > 80\%$: parte del carico viene spostato su un nuovo server (bilanciamento del carico).
- $U < 20\%$: il carico viene distribuito su altri server (*non sovraccarichi*) e il server viene spento (migrazione).

Per modellare l'andamento dell'utilizzazione in ottica di predizioni, sono spesso utilizzate tecniche di predizioni tramite la **regressione**. I predittori di carico non sono, però, molto utilizzati in ambito industriale, perché non sempre il carico è prevedibile e talvolta è meglio reagire in ritardo piuttosto che reagire male.

EcoCloud

REF Paper. "Ecocloud"

La migrazione può essere visto come un processo stocastico **locale** per stabilire se un server deve scaricarsi o se deve accettare nuove macchine. Ad ogni VM è associata una probabilità di essere oggetto o destinazione di una migrazione.

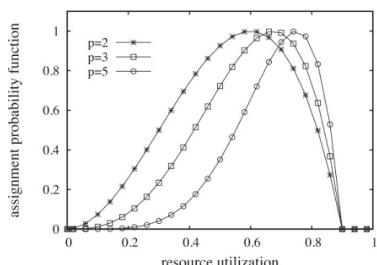


Figure 8: Oggetto di migrazione

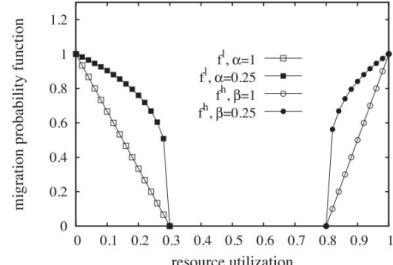


Figure 9: Soggetto a migrazione

Nodi con un'utilizzazione compresa tra il 40% e il 70% saranno destinazione di una migrazione per poter consolidare i server. Quando invece la curva di utilizzo raggiunge il 90% difficilmente tale nodo potrà accoglierne altri senza evitare l'overload e probabilmente subirà una migrazione così da abbassare il carico. Invece quando un nodo ha una bassa utilizzazione è probabile che sarà oggetto di migrazione così da poterlo spegnere. Questo permette la convergenza del sistema ad una condizione ideale dove i nodi hanno un'utilizzazione medio-alta.

Questa visione è stata confrontata con altre *alternative euristiche* come Best Fit Decreasing, che tiene conto di molti parametri e una visione completa del data center, riuscendo comunque a ottenere buoni risultati.

MBFD

REF Paper. “Energy Aware Resource Allocation - Modified Best Fit Decreasing” (MBFD)

Questo algoritmo considera, a differenza del classico best-fit decreasing, macchine virtuali **eterogenee**. Il posizionamento delle VM guarda al **minimizzare il consumo energetico**. La selezione delle macchine da migrare nei server in sovraccarico avviene cercando di minimizzare il numero di **migrazioni**, il numero di **accensioni e spegnimenti**. Su questa base, vengono scelte per la migrazione le macchine più vicine alla soglia, oppure in maniera casuale.

Hierarchical

REF. Paper “Hierarchical”

Si considerano due tipi di problemi: quelli a **lungo termine** e quelli a **breve termine**. Quelli a lungo termine consistono nel distribuire il carico su più data center cloud, quello a breve termine riguarda l’allocazione delle VM. In questo genere di rappresentazione, il tempo di risposta viene modellato tramite **reti di code**.

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 di soluzione a breve termine segue il principio del **receding horizon**, in cui la soluzione ottimale si ottiene considerando l’intera finestra temporale, ma l’algoritmo applica solo le decisioni calcolate per il passo temporale più vicino. Il processo di ottimizzazione viene quindi ripetuto considerando il secondo intervallo di tempo come nuovo punto di partenza.

VMPlanner

REF. Paper “Optimizing Network Utilization - VMPlanner”

Si cerca di posizionare correttamente le VM in modo da ridurre i sovraccarichi della rete, in particolare alleggerendo la parte core della rete. Essendo un *problema NP Hard*, questo viene diviso in sottoproblemi, ciascuno *NP completo*:

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

La soluzione ideale consiste nell’implementazione di un’infrastruttura di rete adattiva, in grado di gestire i flussi di traffico indipendentemente dalla loro dimensione. I nodi all’interno della rete riconoscono il tipo di flusso e ricostruiscono le loro matrici per la comunicazione.

JCDME

REF. Paper “Joint Minimization of the Energy Cost - JCDME”

Approccio che cerca una funzione per minimizzare il **costo energetico** tenendo conto di tanti parametri, a ciascuno dei quali viene associato un peso. La migrazione è un’operazione costosa in termini di consumo energetico, pertanto affinché la decisione abbia senso questo costo deve essere spalmato su più unità di tempo.

Se si ricorre alla migrazione bisogna essere sicuri che in seguito il sistema continuerà a consumare e a richiedere un livello più basso di energia per una quantità di tempo tale che giustifichi la scelta.