

# Cloud Computing

Edoardo Morucci, Tommaso Amarante

2021

# Contents

<b>1 Cloud computing introduction and foundations concepts</b>	<b>2</b>
1.1 Introduction . . . . .	2
1.1.1 Roles . . . . .	3
1.1.2 Cloud Model . . . . .	4
1.1.3 Cloud Computing Infrastructure . . . . .	4
1.2 Benefits of Cloud Computing . . . . .	6
1.2.1 Conventional Computing Paradigm . . . . .	6
1.2.2 Advantages of Cloud Computing . . . . .	7
1.2.3 Other indirect advantages . . . . .	8
1.2.4 Claudification - Use Cases . . . . .	9
1.2.5 Challenges/Risks . . . . .	9
1.3 Cloud Computing Models e NIST Model . . . . .	10
1.3.1 Essential Characteristics . . . . .	11
1.3.2 Service Delivery and Deployment . . . . .	11
1.3.3 NIST Reference Architecture . . . . .	11
1.3.4 Deployments Models . . . . .	13
1.3.5 Service Delivery Models . . . . .	14
<b>2 Virtualization technologies</b>	<b>19</b>
2.0.1 Hypervisor . . . . .	20
2.1 Virtualization types . . . . .	21
2.1.1 Full Virtualization . . . . .	21
2.1.2 Para-Virtualization . . . . .	21
2.1.3 Hardware assisted Virtualization . . . . .	22
2.1.4 Operating System Level Virtualization . . . . .	22
2.2 Emulation . . . . .	23
2.2.1 Virtualization VS Emulation . . . . .	23
2.3 Advantages and Disadvantages of Virtualization . . . . .	24
2.3.1 Advantages . . . . .	24
2.3.2 Disadvantages . . . . .	24
2.4 Recap Computer Architecture . . . . .	24
2.4.1 Multiprogrammazione . . . . .	25
2.4.2 VCPU Execution . . . . .	25
2.5 Full Virtualization . . . . .	30
2.5.1 Hypervisor . . . . .	30

2.5.2	Virtualizing CPU . . . . .	30
2.5.3	Guest OS Execution . . . . .	31
2.5.4	Trap and emulate virtualization model . . . . .	32
2.5.5	Virtualizing physical memory . . . . .	33
2.5.6	Virtual MMU . . . . .	33
2.5.7	Brute Force Method . . . . .	34
2.5.8	Virtualizing I/O devices . . . . .	35
2.5.9	Interrupt management . . . . .	36
2.5.10	Virtualizing interrupts . . . . .	36
2.5.11	Hardware input . . . . .	37
2.5.12	QEMU . . . . .	37
2.6	Hardware assisted virtualization . . . . .	38
2.6.1	Performance penalty . . . . .	38
2.6.2	Hardware Assisted Virtualization . . . . .	38
2.6.3	Virtual machine control structure . . . . .	40
2.6.4	RAM management . . . . .	41
2.6.5	Extended page table . . . . .	42
2.6.6	Hardware passthrough . . . . .	43
2.6.7	I/O mapping . . . . .	44
2.6.8	Interrupts . . . . .	44
2.6.9	VM state . . . . .	44
2.6.10	Interrupt management . . . . .	45
2.6.11	PID update . . . . .	47
2.6.12	DMA devices . . . . .	47
2.6.13	IOMMU . . . . .	48
2.6.14	Nested virtualization . . . . .	49
2.6.15	KVM . . . . .	50
2.7	Para-Virtualization . . . . .	51
2.7.1	Paravirtualization Architecture . . . . .	51
2.7.2	Paravirtualization Implementation . . . . .	51
2.7.3	Paravirtualization Pros and Cons . . . . .	52
2.7.4	XEN . . . . .	52
2.7.5	Virtualization overhead . . . . .	53
2.7.6	Lightweight virtualization . . . . .	54
2.8	Operating system virtualization . . . . .	54
2.9	Containers . . . . .	55
2.9.1	Linux Containers . . . . .	56
2.9.2	Namespaces . . . . .	56
2.9.3	Docker . . . . .	57
2.9.4	Serverless computing . . . . .	57
<b>3</b>	<b>Cloud application structure and architecture</b>	<b>58</b>
3.1	Traditional application . . . . .	58
3.2	Cloudification . . . . .	59
3.3	Cloud application . . . . .	59

3.3.1	Cloud application architecture . . . . .	59
3.3.2	Multi-tier architecture . . . . .	60
3.4	Compute Cluster . . . . .	61
3.4.1	Hight-Availability Clusters . . . . .	61
3.4.2	Load-Balancing clusters . . . . .	62
3.4.3	Compute Intensive Cluster . . . . .	63
3.5	Dynamic adaptation . . . . .	65
3.6	Cloud application design . . . . .	65
3.6.1	Requirements . . . . .	65
3.7	Service-orientes application (SOA) . . . . .	66
3.7.1	SOA architecture . . . . .	66
3.7.2	SOA roles . . . . .	66
3.7.3	SOA broker . . . . .	67
3.7.4	SOA advantages . . . . .	67
3.7.5	SOA scalability . . . . .	67
3.8	Web service . . . . .	68
3.8.1	World Wide Web . . . . .	68
3.8.2	Uniform resource identifier (URI) . . . . .	68
3.8.3	HTTP . . . . .	69
3.8.4	Data encoding . . . . .	69
3.8.5	XML validation . . . . .	70
3.8.6	Web-Service interaction . . . . .	70
3.9	SOAP . . . . .	71
3.9.1	SOAP binding . . . . .	72
3.10	WSDL . . . . .	72
3.10.1	WSDL message architecture . . . . .	73
3.10.2	WSDL type . . . . .	73
3.10.3	WSDL messages . . . . .	74
3.10.4	WSDL porttype . . . . .	74
3.10.5	WSDL binding . . . . .	75
3.10.6	WSDL service . . . . .	75
3.11	UDDI . . . . .	75
3.12	REST . . . . .	76
3.13	JSON . . . . .	77
3.13.1	JSON vs XML . . . . .	77
3.13.2	JSON in WS . . . . .	78
3.14	CRUD . . . . .	78
3.15	Request-Reply communication . . . . .	79
3.15.1	Indirect communication . . . . .	79
3.16	Message oriented communication . . . . .	79
3.16.1	Message broker . . . . .	80
3.16.2	Message queue . . . . .	81
3.16.3	Implementations . . . . .	83
3.17	Data replication . . . . .	84
3.17.1	Requirements . . . . .	84

3.18	System Model . . . . .	85
3.18.1	Group/Multicast Communications . . . . .	86
3.18.2	Primary Backup Replication . . . . .	87
3.18.3	Active replication . . . . .	88
3.18.4	The Gossip architecture . . . . .	89
3.19	Global-scale application . . . . .	93
3.19.1	Load Balancing . . . . .	93
3.19.2	Global Load Balancing with DNS . . . . .	93
3.19.3	Virtual IP addresses . . . . .	94
3.20	Geographically distributed systems . . . . .	96
3.21	CAP theorem . . . . .	96
3.22	BASE semantic . . . . .	97
3.22.1	Inconsistency . . . . .	98
3.22.2	Advantages/Disadvantages . . . . .	98
3.23	Bayou architecture . . . . .	99
3.23.1	Bayou operations . . . . .	99
3.23.2	Primary replica manager . . . . .	99
3.23.3	Conflict detection and resolution . . . . .	99
3.23.4	Performance measurement . . . . .	100
<b>4</b>	<b>Cloud Platform</b>	<b>101</b>
4.1	Cloud computing platform . . . . .	101
4.1.1	Recap: The cloud (for now) . . . . .	101
4.1.2	The cloud infrastructure . . . . .	101
4.1.3	Cloud Platform . . . . .	102
4.2	OpenStack . . . . .	103
4.2.1	OpenStack Software Platform . . . . .	103
4.2.2	OpenStack instances . . . . .	104
4.2.3	OpenStack Architecture . . . . .	105
4.2.4	OpenStack Services . . . . .	106
4.2.5	Information Exchange/Communication . . . . .	107
4.2.6	Keystone . . . . .	107
4.2.7	Nova . . . . .	109
4.2.8	Glance . . . . .	112
4.2.9	Neutron . . . . .	113
4.2.10	OpenVSwitch (OVS) . . . . .	117
4.2.11	Network virtualization . . . . .	119
4.2.12	Virtual LAN . . . . .	120
4.2.13	GRE Tunneling . . . . .	122
4.2.14	VXLAN . . . . .	123
4.2.15	Cinder . . . . .	125
4.2.16	Ceilometer . . . . .	126
4.2.17	Horizon . . . . .	128
4.2.18	Service APIs . . . . .	128
4.2.19	Swift . . . . .	129

4.2.20	Heat . . . . .	129
4.3	OpenStack Service Interactions Summary . . . . .	131
4.3.1	High Availability (HA) . . . . .	132
4.3.2	HA with dedicated node . . . . .	133
4.3.3	HA with simple redundancy . . . . .	134
4.4	Lightweight cloud computing platform for DevOps . . . . .	135
4.4.1	Software development - BACK . . . . .	135
4.4.2	Software development - NOW . . . . .	135
4.4.3	Application development process: DevOps, NoOps . . . . .	136
4.4.4	Service deployment and management . . . . .	137
4.4.5	Service dependencies . . . . .	137
4.4.6	Lihtweight Virtualization . . . . .	138
4.4.7	Kubernetes . . . . .	139
4.4.8	Cluster Architecture . . . . .	141
4.4.9	Application Deployment . . . . .	141
4.4.10	Management and Maintenance . . . . .	142
4.4.11	Container Migration . . . . .	142
4.4.12	Exposing Applications . . . . .	142
4.4.13	Load Balancing . . . . .	143
4.4.14	Failure handling . . . . .	143
4.4.15	Pod design . . . . .	144
<b>5</b>	<b>Cloud Storage and Distributed File System</b>	<b>145</b>
5.1	Block storage and File System . . . . .	145
5.2	Object Storage . . . . .	146
5.3	Single Server Storage Model . . . . .	146
5.3.1	RAID . . . . .	146
5.4	Distributed File System . . . . .	148
5.4.1	NFS . . . . .	148
5.4.2	GlusterFS . . . . .	149
5.5	Cloud Storage . . . . .	151
5.5.1	VM Live Migration . . . . .	151
5.5.2	Strorage Area Network . . . . .	151
5.5.3	Unified Storage Solutions . . . . .	152
5.6	Ceph . . . . .	152
5.6.1	Ceph Services . . . . .	153
5.6.2	Ceph Architecture . . . . .	153
5.6.3	Ceph Object . . . . .	154
5.6.4	CRUSH Algorithm . . . . .	155
5.6.5	Cluster Map . . . . .	155
5.6.6	Data Replication . . . . .	156
5.6.7	Pools . . . . .	157
5.6.8	MON Redundancy . . . . .	157
5.6.9	Placement groups computation . . . . .	158
5.6.10	Recovering and Rebalancing . . . . .	158

5.6.11	Ceph and OpenStack . . . . .	159
5.6.12	Sorage as a Service . . . . .	160

# Chapter 1

## Cloud computing introduction and foundations concepts

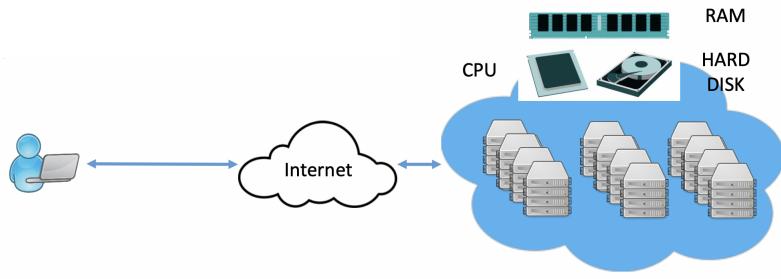
### 1.1 Introduction

Durante gli anni 80 i miglioramenti alla struttura di comunicazione ha portato all'aumento della velocità di trasferimento. Questa miglioria ha permesso lo sviluppo di sistemi distribuiti. Le applicazioni sono programmate per essere suddivise in task che possono essere assegnate a un determinato host, gli host comunicano i risultati fra di loro. L'evoluzione dei sistemi distribuiti sono i cluster. Un cluster è formato da un insieme di pc che sono collegati fra di loro in LAN, questa è l'unica differenza presente con i sistemi distribuiti classici. Il sistema a cluster porta una maggiore efficienza computazionale e sono più affidabili. All'interno di ogni cluster è presente un cluster head, esso gestisce la distribuzione delle task all'interno del suo cluster. Questo rappresenta un single point of failure. La risoluzione a questo problema arriva con il Grid Computing, eliminano il **cluster head** e si adotta un metodo decentralizzato per decidere quale pc dovrà svolgere la task richiesta. Il Grid Computing ha diversi svantaggi:

- Non è possibile lo scaling real time: è necessario cambiare l'architettura per inserire un nuovo nodo
- Non è fault tolerant: se fallisce un nodo fallisce la task
- Se l'hardware è eterogeneo il codice si deve adattare

**Definizione originale di Cloud Computing: è un ambiente distinto dell'IT, sviluppato con lo scopo di fornire risorse IT scalabili e misurabili, accessibili tramite Internet.**

Le risorse fornite dalla Cloud Infrastructure sono usate per implementare i cloud services.



Un cloud service può essere un applicazione o un servizio puro:

- Se applicazione: viene fornita un'interfaccia grafica magari accessibile via web
- Se servizio: viene usato da altre applicazioni tramite API

### 1.1.1 Roles

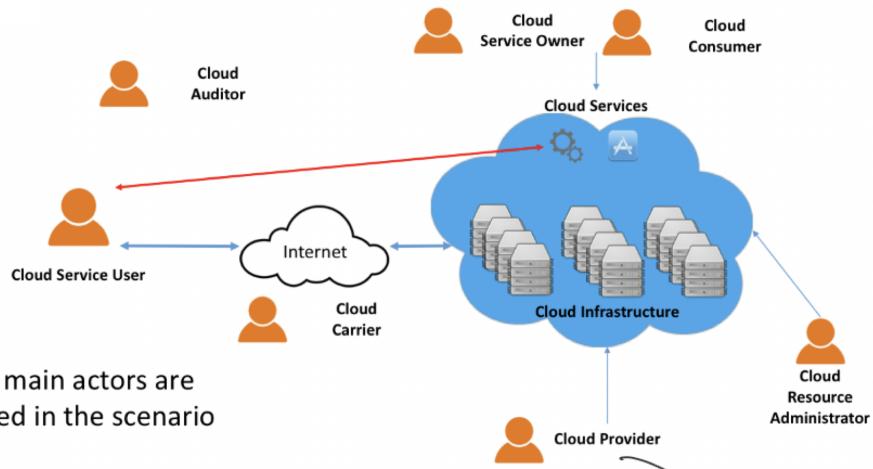


Figure 1.1: Roles in cloud computing.

**Cloud consumer:** organizzazione che hanno un contratto col cloud provider per prendere in prestito e usare IT resources.

**Cloud service owner:** organizzazione che usa la cloud infrastructure per creare un cloud service.

Spesso solo la stessa persona, qualche volta non coincidono. In questo caso il cloud consumer può essere un broker che compra IT resources dal cloud provider per rivenderle e guadagnare. Service owner, consumer e provider spesso coincidono con la stessa organizzazione come ad esempio Google.

**Cloud resource admin:** ha il ruolo di configurare a basso livello le risorse e amministrare, spesso il provider e l'amministratore sono la stessa persona.

**Cloud auditor:** azienda di terze parti che si occupa delle performance e della sicurezza

**Cloud carrier:** provider internet che garantisce la connessione dell’infrastruttura cloud, mette in collegamento l’utente finale con l’infrastruttura. **Cloud service user:** è l’utente finale che accede ai servizi magari tramite applicazione ecc.

### 1.1.2 Cloud Model

Il modello per la consegna delle risorse IT è scalabile e misurabile, questo porta a due conseguenze: la **misurabilità**, ovvero si misurano le risorse utilizzate per adottare un metodo “pay per use”, paghi per quante risorse effettivamente utilizzi e la **scalabilità**, ovvero in un breve tempo le risorse devono essere allocate dinamicamente. Un utility service model si basa quindi su un servizio che misura chiaramente le risorse utilizzate in modo da far pagare il prezzo giusto e eroga le risorse on-demand. Le risorse sono allocate dal cloud provider in un breve periodo di tempo e con il minimo sforzo, magari senza l’intervento dell’uomo. Un software di orchestrazione può decidere quando e se allocare nuove risorse.

Il business model del cloud computing consiste in 2 parti:

- I provider vendono le IT resources (amazon, google ecc)
- Altre compagnie comprano queste risorse per creare applicazioni che poi rivendono ai loro acquirenti (Netflix ad esempio)

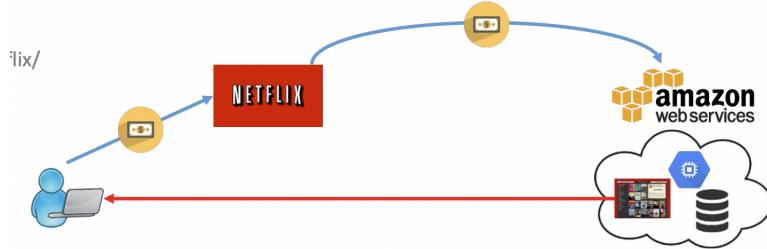


Figure 1.2: Cloud Computing Business Model.

### 1.1.3 Cloud Computing Infrastructure

L’infrastruttura è raccolta in data center dove sono fisicamente installati i server del provider. Le risorse del data center possono essere accedute anche da più utenti in contemporanea, questo crea un problema. In un sistema multi utente (come linux) più utente contemporaneamente possono accedere rispettando le regole imposte dall’amministratore, quando gli utenti cominciano ad essere tanto risulta impossibile per l’amministratore poter regolare tutti. Si passa quindi al concetto di multi-tenancy, un Tenant ha pieno controllo del sistema in cui si trova agendo come se fosse l’amministratore del sistema. Per poter attuare questo metodo è necessario virtualizzare l’hardware in modo da creare più macchine virtuali, a ogni Tenant è affidata una o più VM dove ha l’impressione di aver pieno controllo del sistema in quanto può installare un SO di suo piacimento e tutte le librerie di cui ha bisogno, quello che controlla è in realtà una virtualizzazione del reale hardware del computer host. Per poter virtualizzare l’hardware si utilizza l’Hypervisor, è uno strato di virtualizzazione posto fra l’host e le

macchine virtuali. L'hypervisor si occupa della creazione delle VM e gira sull'host fisico che ospiterà le macchine virtuali.

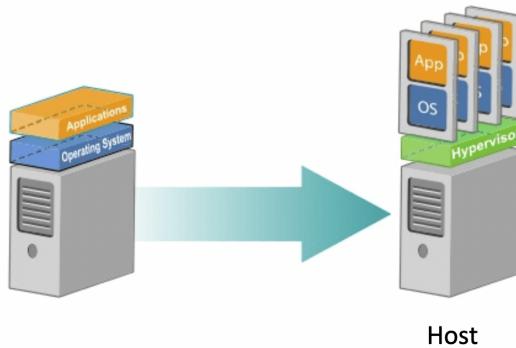
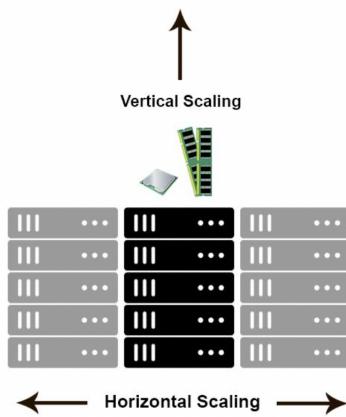


Figure 1.3: Hypervisor.

Si può applicare la virtualizzazione alla cloud infrastructure ottenendo così una serie di server dotati dell'hypervisor per creare delle VM e collegati fra di loro in una LAN, essa può interracciararsi verso l'internet tramite un border router in modo da poter essere raggiunta dai consumer. L'hypervisor garantisce anche l'allocazione dinamica delle risorse permettendo due tipi di scaling diversi:

- **Vertical scaling:** le VM vengono riconfigurate in modo da avere più potenza di calcolo. (Scale Up/down → aumento/diminuisco la capacità).
- **Horizontal scaling:** vengono allocate più o meno VM al servizio in base alla richiesta (scaling UP/DOWN → allocare/deallocare risorse)



l'Horizontal scaling è lo scaling maggiormente utilizzato nel cloud computing al contrario del Vertical scaling, in quanto quest'ultimo richiede dei periodi in cui il servizio non è raggiungibile a causa del fatto che la macchina deve essere aggirionata con nuovi componenti hardware. Non si può chiaramente fare scaling verticale illimitato in quanto la potenza che può essere

virtualizzata è al più la potenza di calcolo fisica dell'host. La virtualizzazione permette anche una misurazione precisa delle risorse utilizzate in modo da calcolare al meglio la quantità di denaro che deve essere richiesta per l'affitto delle risorse.

Horizontal Scaling	Vertical Scaling
less expensive (through commodity hardware components)	more expensive (specialized servers)
IT resources instantly available	IT resources normally instantly available
resource replication and automated scaling	additional setup is normally needed
additional IT resources needed	no additional IT resources needed
not limited by hardware capacity	limited by maximum hardware capacity

Figure 1.4: Comparing vertical and horizontal scaling.

## 1.2 Benefits of Cloud Computing

Il cloud computing ha cambiato in modo radicale il mondo in cui i sistemi informatici vengono sviluppati e costruiti. Ha portato quindi a uno spostamento del paradigma rispetto al modo convenzionale. Rispetto al modo precedente, il cloud computing porta una serie di vantaggi per i business e grazie alle innovazioni tecnologiche è proprio riuscito a cambiare il modo in cui i business vengono organizzati.

### 1.2.1 Conventional Computing Paradigm

Le soluzioni tradizionali prevedono il completo design e installazione dell'infrastruttura all'interno della tua azienda, questo può essere fatto tramite un grande investimento di hardware e persona specializzato. Il personale deve infatti di occuparsi della manutenzione del sistema e deve occuparsi anche del setup iniziale, questo equivale a dire installare l'hardware vero e proprio, installare un sistema operativo e successivamente occuparsi del software necessario.

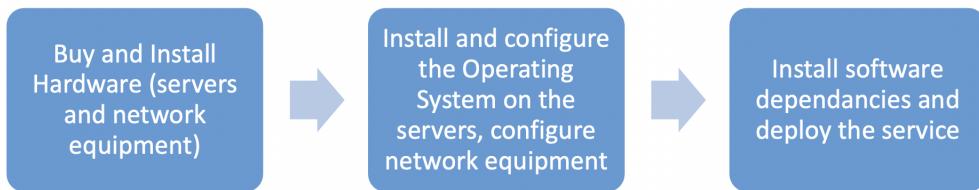


Figure 1.5: Company needs to take care of.

Dopo l'investimento iniziale è presente un costo fisso in quanto è necessario del personale per la manutenzione e per la riparazione dei pezzi che potrebbero rompersi. Quello di cui l'azienda deve occuparsi è, appunto, la riparazione e sostituzione dell'hardware, aggiornare

il sistema operativo per stare al passo con le patch di sicurezza e mantenere funzionanti i software.



Figure 1.6: Maintenance of an application.

Questo paradigma impedisce la scalabilità, vediamo perché:

- Se l'azienda cresce rapidamente, anche in una sola notte in qualche caso, l'hardware dovrebbe scalare per stare al passo con la richiesta, ma questo si può fare solo comprando e installando dell'hardware nuovo, questo porta a costi aggiuntivi.
- Se l'azienda perde mercato e quindi si rimpicciolisce il carico si riduce, lo scale down non è possibile in quanto l'hardware comprato e installato ormai è presente, abbiamo quindi uno spreco di risorse

### 1.2.2 Advantages of Cloud Computing

Analizziamo ora i benefici del cloud computing. Il CC è caratterizzato da un paradigma basato sull'utilizzo, questo può essere estremamente utile dal momento che si paga solo le risorse che si usa, essere vengono allocate in maniera dinamica permettendo quindi un'ottima *scalabilità*, in più con il CC vengono ridotti i *costi iniziali* (quelli per comprare e installare l'hardware) e *quelli di manutenzione futuri*. L'alta scalabilità del sistema può essere garantita grazie a dei software che sono in grado di riconoscere quante risorse necessarie e allocarle o deallocarle di conseguenza. Non ci sono costi fissi in quanto è possibile sviluppare un servizio non avendo comprato nessuno hardware per mettere in piedi l'infrastruttura e non c'è bisogno di persona specializzato che mantenga in funzione tutta l'infrastruttura.

Generalmente i CC provider riescono a fornire risorse hardware a basso costo in quanto essi creano un infrastruttura con le seguenti proprietà:

- **Grande:** vengono comprati in stock componenti hardware per la creazione dell'infrastruttura.
- **Condivisa:** i costi vengono ammortizzati in quanto si sfrutta al massimo l'hardware che viene condiviso fra più utenti.
- **I costi fissi vengono condivisi:** anche il personale per la manutenzione viene condiviso, i costi saranno quindi ulteriormente abbattuti.

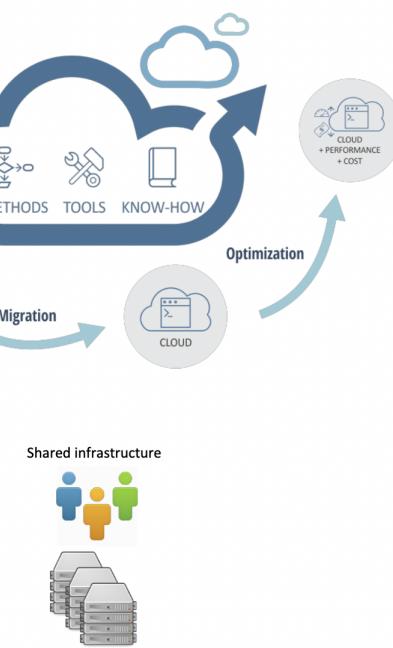


Figure 1.7: Cloud Computing Workflow.

### 1.2.3 Other indirect advantages

Il CC porta anche a dei vantaggi indiretti:

- **Una minima responsabilità di gestione:** avere un'infrastruttura fisica comporta delle responsabilità, anche legali, per la compagnia. Con il CC la responsabilità viene spostata verso il cloud provider, questo comporta anche una riduzione del rischio di gestione in quanto il provider avrà dei team di esperti che possono eccitarsi di tutte le azioni necessarie.
- **Una maggiore qualità del servizio:** se con l'infrastruttura privata l'azienda ha bisogno di una divisione addetta appositamente, con il CC il provider e avrà dalla sua un team di persone super competenti dedicate interamente a fornire la migliore qualità del servizio possibile.
- **Affidabilità:** il CC provider si occuperà di fornire un servizio quanto più affidabile possibile che sia resistente ad esempio alla rottura di qualche pezzo.
- **Disponibilità continua:** il CC provider assicura una disponibilità 24/7.
- **Minima gestione del software:** il CC provider si occupa anche di procurare le licenze e di mantenere funzionante il software, l'azienda quindi non deve più preoccuparsi neanche di questo aspetto.
- **Indipendenza dalla Location:** i servizi di cloud computing sono avvertibili via internet, quindi da ogni parte del mondo e in qualsiasi momento.

- **L'azienda è concentrata solo sul business:** i programmatori, ad esempio, sono concentrati a sviluppare la logica del servizio invece di mantenere l'infrastruttura.

#### 1.2.4 Claudification - Use Cases

Andiamo adesso ad analizzare alcuni esempi di come il CC viene utilizzate dalle aziende:

- **Cludification:** le grandi aziende spostano una parte delle proprie attività sul cloud in modo da ridurre i costi fissi. Ad esempio expedia ha spostato l'80% della propria infrastruttura su AWS riducendo i costi fissi e riducendo anche di molto la latenza.
- **Startup:** con il CC le piccole startup possono quindi trasformare le proprie idee in business in quanto non hanno bisogno di affrontare l'investimento per creare l'infrastruttura.
- **Sporadic HPC (high performance computing):** le aziende che hanno bisogno di performance elevate ma solo per un periodo di tempo limitato possono affittare le risorse invece che creare la propria infrastruttura. Ad esempio Pixar ha bisogno per un breve periodo di tempo di un gran numero di risorse per renderizzare i film.
- **Global service:** con il paradigma tradizionale è impensabile per un'azienda poter costruire un servizio mondiale in quanto, se venisse creato in un singolo luogo si avrebbe sicuramente il collo di bottiglia in quanto le informazioni devono essere trasportate tramite internet altrimenti dovrebbero costruire diversi nodi sparsi per il mondo e ti-disegnare interamente la struttura. Con il CC i provider forniscono la loro infrastruttura suddivisa in data center differenti sparsi per il mondo in modo da coprire efficientemente la maggior parte delle aree geografiche. Un esempio di global service è il motore di ricerca di google che accetta richieste da tutto il mondo gestendo una mole di dati enorme.
- **Data collection:** il CC è un modo ottimale per la raccolta dei dati attraverso sensori i quanto fornisce la scalabilità giusta per gestire una gran quantità di sensori e i relativi dati generati.
- **Content distribution:** i servizi di distribuzione a livello mondiale devono garantire il servizio a un gran numero di utenti nella maniera più veloce possibile. Questo può essere ottenuto grazie al CC in quanto viene fornito supporto al global search e un'ottima scalabilità

#### 1.2.5 Challenges/Risks

L'approccio relativo al cloud porta però degli svantaggi che i provider vedono come sfide da superare e possono essere visti come dei rischi dal punto di vista dell'utente finale, ecco alcuni esempi:

- **Costo della connessione:** per poter accedere al CC è necessaria una connessione a internet che deve essere continua e performante. Questo si traduce in un costo aggiuntivo per l'azienda.

- **Prestazione della connessione:** la prestanza della connessione è un requisito cruciale, se non è abbastanza alta essa può trasformarsi in un collo di bottiglia vanificando gran parte dei vantaggi portati dal cloud computing.
- **Portabilità limitata:** dal momento che non esiste una standardizzazione dei servizi di cloud computing è possibile che un servizio distribuito attraverso l'infrastruttura di un determinato CC provider non funzioni su quella di un provider differente, questo fenomeno alimenta il vendor lock-in. Spiegato, l'utente finale deve spendere una quantità di soldi non indifferente per poter spostare il proprio servizio da un provider ad un altro.
- **Problemi legali:** dal momento che i provider costruiscono la loro infrastruttura in un luogo di convenienza, gli utenti finali possono cadere in problemi legali legati allo storage dei dati. È possibile, ad esempio, che alcune leggi possano regolare alcuni tipi di dati ad empio dati sensibili riguardo la sanità. È necessario che vengano salvati e mantenuti all'interno della nazione dell'utente, questo può creare conflitti.

Un altro problema percettivo che deriva dal CC è quello relativo alla sicurezza dei dati. Il problema sorge dal momento che per salvare o leggere i dati all'interno del cloud è necessario passare da internet e questo potrebbe creare dei problemi di sicurezza. Il problema può essere semplicemente risolto applicando un algoritmo di criptaggio dei dati prima di essere trasmessi tramite internet.

### 1.3 Cloud Computing Models e NIST Model

Il cloud computing non è molto standardizzato, infatti è una tecnologia recentemente introdotta. Ogni cloud provider ha le proprie soluzioni tecnologiche per realizzare i servizi che offre. Abbiamo quindi bisogno di un modello standard, il più famoso ed accettato è il "NIST cloud computing reference architecture" del National Institute of Standards and Technology. Il NIST model definisce gli aspetti base e le caratteristiche del cloud computing:

- Caratteristiche essenziali: set di caratteristiche obbligatorie che ogni sistema basato su cc deve avere.
- Modelli di servizio: servizi offerti dal provider al consumer.
- Modelli di distribuzione: infrastruttura (public, hybrid, ..).

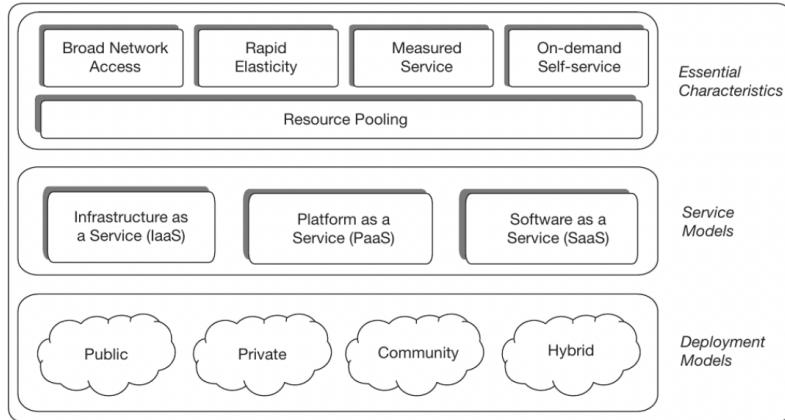


Figure 1.8: Basic aspects of cloud computing.

### 1.3.1 Essential Characteristics

Il modello NIST prevede 5 **caratteristiche essenziali** che ogni infrastruttura cloud deve avere:

- **Broad network access** (l’infrastruttura deve essere accessibile da ovunque)
- **Rapid elasticity** (le risorse devono essere allocate/disallocate molto velocemente)
- **Measured service** (le risorse usate devono essere misurate e il bill deve essere proporzionale all’utilizzo)
- **On demand** (le risorse vengono allocate su richiesta)
- **Resource pooling**: le risorse computazionali sono istallate e disponibili on demand, inoltre sono mantenute in remoto e devono essere abbastanza per soddisfare molti utenti allo stesso tempo.

### 1.3.2 Service Delivery and Deployment

**Modelli di erogazione del servizio:** il modello NIST definisce tre categorie principali di servizio in base al tipo di servizio offerto ai consumatori e al modo in cui il servizio viene fornito: Infrastructure as a Service (*IaaS*), Platform as a Service (*PaaS*) e Software as a Servizio (*SaaS*) **Implementazione:** il modello NIST definisce quattro tipi di implementazione, in base a come viene distribuita l’infrastruttura: cloud pubblico, cloud privato, cloud comunitario e cloud ibrido

### 1.3.3 NIST Reference Architecture

Parlando dell’architettura, essa è l’insieme di elementi che compongono l’ecosistema cloud. Si concentra sul da cosa è composta e non sul come è implementata, è inoltre molto generale quindi copre tutte le implementazioni cloud disponibili sul mercato. Specifica anche gli attori già trattati nella sottosezione 1.1.1.

Si aggiunge però il **Cloud broker** che è una terza parte che si interpone tra il cloud provider e il cloud consumer, si occupa di gestire e fornire i servizi cloud prendendoli negoziando con differenti provider. I benefici sono di non interagire direttamente con il provider e soprattutto vengono ridotti i costi.

Il modello NIST definisce 4 tipi di **Deployment model**: Public cloud, private cloud, community cloud and hybrid cloud, Di seguito li analizzeremo singolarmente.

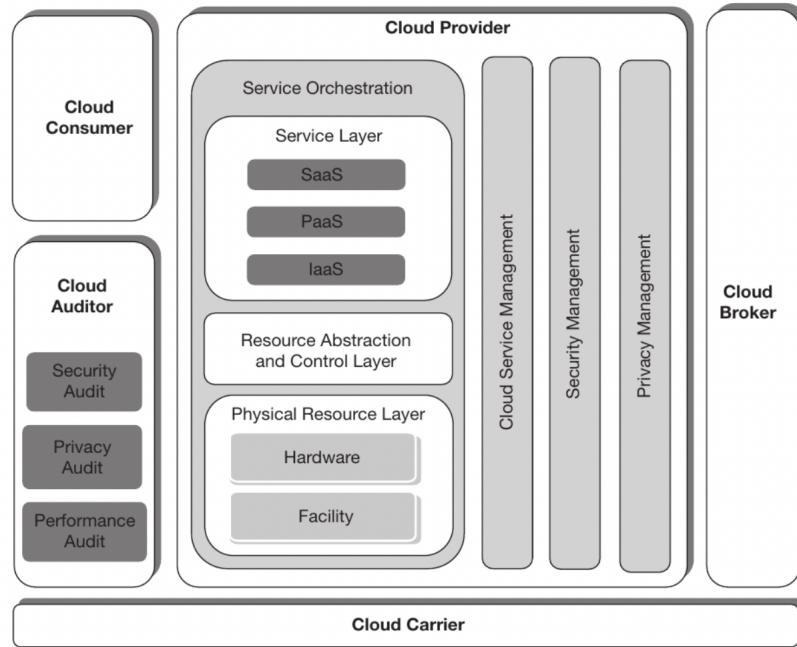


Figure 1.9: NIST Reference Architecture

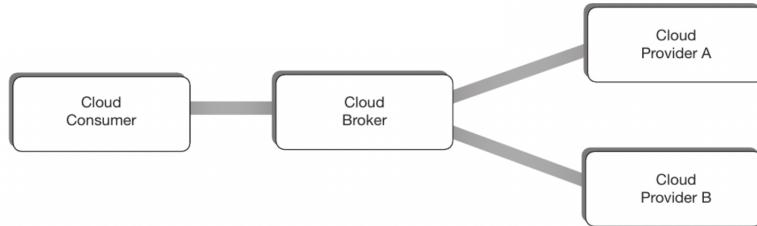


Figure 1.10: Role of Cloud Broker.

### 1.3.4 Deployments Models

L'implementazione di un'infrastruttura cloud (e dei servizi corrispondenti) può essere organizzata in diversi modi, a seconda dei requisiti dell'organizzazione del consumatore. I diversi modelli sono solitamente correlati alla posizione dell'infrastruttura rispetto all'organizzazione e ai suoi confini di accesso.

#### Public cloud

È il più popolare, l'infrastruttura (ed il servizio) è gestito da un'organizzazione esterna che vende servizi e risorse, il consumer accede al servizio da remoto. Public cloud promuove multi-tenancy ad alto livello, le stesse risorse fisiche sono condivise da più consumer indipendenti. Questo è un grande vantaggio per i service provider che con la stessa infrastruttura possono soddisfare un grande numero di clienti. Per i clienti i vantaggi sono nei costi ridotti, utilizzo di tecnologie innovative e persone competenti.

#### Private cloud

Non sono aperti, sono distribuiti e gestiti da una singola organizzazione per un suo uso personale. L'infrastruttura è costruita con tutte le caratteristiche di un cloud computing system, comunque con un accesso ristretto. I Private cloud solitamente risiedono dentro l'azienda proprietaria e la gestione può essere affidata al cliente oppure ad azienda esterna. Solitamente si usa private cloud per clienti che vogliono un controllo specifico sull'intero sistema oppure per requisiti come la gestione dati sensibili.

#### Differences between Public and Private cloud

La differenza principale fra questi due è che il private cloud gestisce prevalentemente relazioni uno-a-uno con l'utente finale, mentre il public cloud gestisce relazioni di tipi uno-a-molti.

<i>Private Cloud</i>	<i>Public Cloud</i>
It can be both of types of on-premises and off-premises.	There cannot be any on-premises public cloud deployment.
On-premises private cloud can be delivered over the private network.	It can only be delivered over public network.
It does not support multi-tenancy feature for unrelated and external tenants.	It demonstrates multi-tenancy capability with its full ability.
The resources are for exclusive use of one consumer (generally an organization).	The resources are shared among multiple consumers.
A private cloud facility is accessible to a restricted number of people.	This facility is accessible to anyone.
This is for organizational use.	It can be used both by organization and the user.

Figure 1.11: Private vs Public.

## Community cloud

l'accesso al cloud è permesso solo a organizzazioni (spesso condividono interessi e obiettivi) e clienti all'interno della community. Viene creato e gestito dai membri della community oppure da una compagnia esterna. Il modello supporta multi-tenancy tra i membri della community, l'obiettivo è di avere i benefici del cloud pubblico con il controllo del privato. I costi sono condivisi tra i membri della community e sono inferiori rispetto al cloud privato, può essere usato anche il pay per use.

## Hybrid cloud

È la combinazione tra private o community cloud e il public cloud. Questo modello aiuta le organizzazioni ad avere i vantaggi sia del private che del public, l'infrastruttura è formata combinando quella del private e public. I vantaggi sono un costo computazionale più basso e alta scalabilità di risorse affiancate da alto livello di controllo e privacy.

Per scegliere il modello appropriato dobbiamo basarci su diversi fattori, principalmente dipende dalle necessità del business (eg data security, low latency) e sulla dimensione dell'azienda cliente. In generale la scelta ricade sul cloud pubblico ma per compagnie che hanno delle particolari necessità sulla privacy e dati sensibili la scelta più corretta può essere private o community. Un altro fattore da non scordare è il budget.

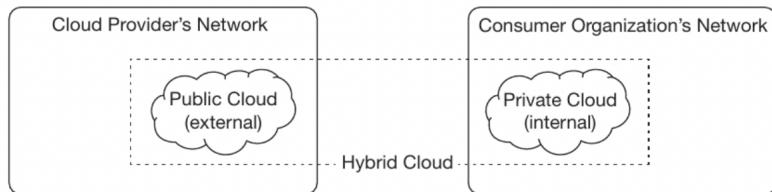


Figure 1.12: Hybrid cloud.

### 1.3.5 Service Delivery Models

Il modello NIST definisce tre categorie principali basate sul tipo di servizio offerto al consumer e nel modo in cui viene rilasciato: **IaaS, PaaS e SaaS**. Essi dipendono dal livello di astrazione dell'infrastruttura hardware: da IaaS a SaaS aumenta il livello di astrazione, la scelta del modello dipende da diversi aspetti che il cliente deve tenere di conto per scegliere quello a lui appropriato. Nella computazione convenzionale si deve: - Creare l'infrastruttura fisica con server, storage e network - Installare e configurare il SO nei server - Installare su ogni server middleware/software per sviluppare applicazioni/servizi - Sviluppare e distribuire il servizio - Gestire/salvare i dati Tutti questi aspetti devono essere gestiti dal consumer. Analizziamo in dettaglio i 3 modelli offerti dal NIST.

#### IAAS

Fornisce risorse hardware virtualizzate al cliente, permette di usare processore virtuale, memoria, storage e rete da remoto. Queste risorse virtuali possono essere usate come risorse reali

per creare VM. IaaS si riferisce anche ad Hardware as a service. I clienti non si devono occupare di gestire e controllare l'infrastruttura, se ne occuperà il service provider, si deve invece occupare di:

- Installare il SO in cima all'hardware virtualizzato.
- Configurare middleware/runtime environment.
- Gestire i dati.

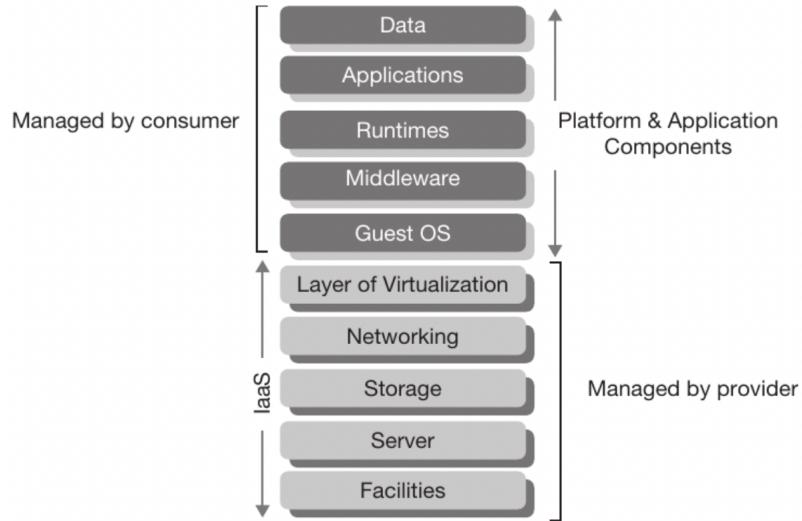


Figure 1.13: IaaS.

IaaS rimuove i costi per costruire e mantenere l'infrastruttura fisica, mantenendo comunque un buon controllo dell'ambiente. La maggior parte dei provider forniscono IaaS, è anche il servizio più utilizzato. Amazon è leader con il 33% del mercato.

## PAAS

Fornisce ogni aspetto dall'infrastruttura fisica all'installazione del SO e del middleware/runtime, il cliente si deve occupare solamente di sviluppare l'applicazione e gestire i dati. Lo svantaggio sta nella mancanza di flessibilità infatti le applicazioni devono essere programmate usando le APIs del sistema, per questa ragione le applicazioni programmate per una piattaforma cloud sono difficili da trasportare su un altro tipo. Gli esempi più popolari di PaaS sono Google app engine, microsoft azure platform, force.com.

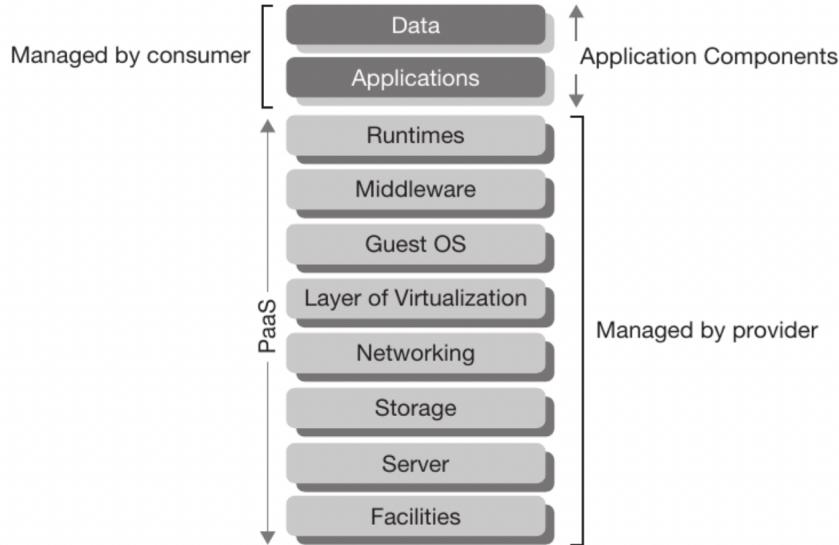


Figure 1.14: Paas.

## SAAS

Fornisce applicazioni come servizio, molte applicazioni sono sviluppate dal cloud provider e offerte all'utente tramite interfaccia web. In questo modello il cloud provider gestisce ogni aspetto del sistema, dall'hardware allo sviluppo software. I clienti non si devono occupare di niente, nemmeno della gestione della licenza software.

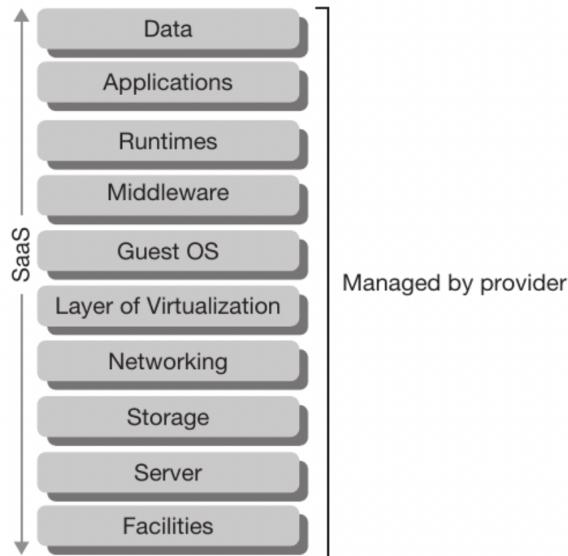


Figure 1.15: Saas.

Vediamo alcuni esempi dell'utilizzo di questi 3 metodi. Paas/SaaS possono essere implementate in cima ad un'infrastruttura IaaS, un IaaS può essere usata per creare una Paas: le VM

vengono create e sopra ci viene messo il software della piattaforma. L'infrastruttura Paas può essere usata per creare un sistema Saas, la piattaforma Paas è usata per creare il software che fornisce la Saas.

## Other Services

È importante menzionare altri servizi di cloud:

- Storage as a service: Solitamente fornito assieme a Iaas ma è anche usato come servizio indipendente.
- Database as a service: Solitamente fornito assieme a Paas ma può essere fornito indipendentemente. Il DBaaS offre una piattaforma unica on demand e self service.
- Backup as a service: è ovviamente sempre importante fare backup, viene infatti fornito come servizio, è utile ed a buon prezzo.
- Desktop as a service: viene fornito un desktop controllabile da remoto.

## Model Examples

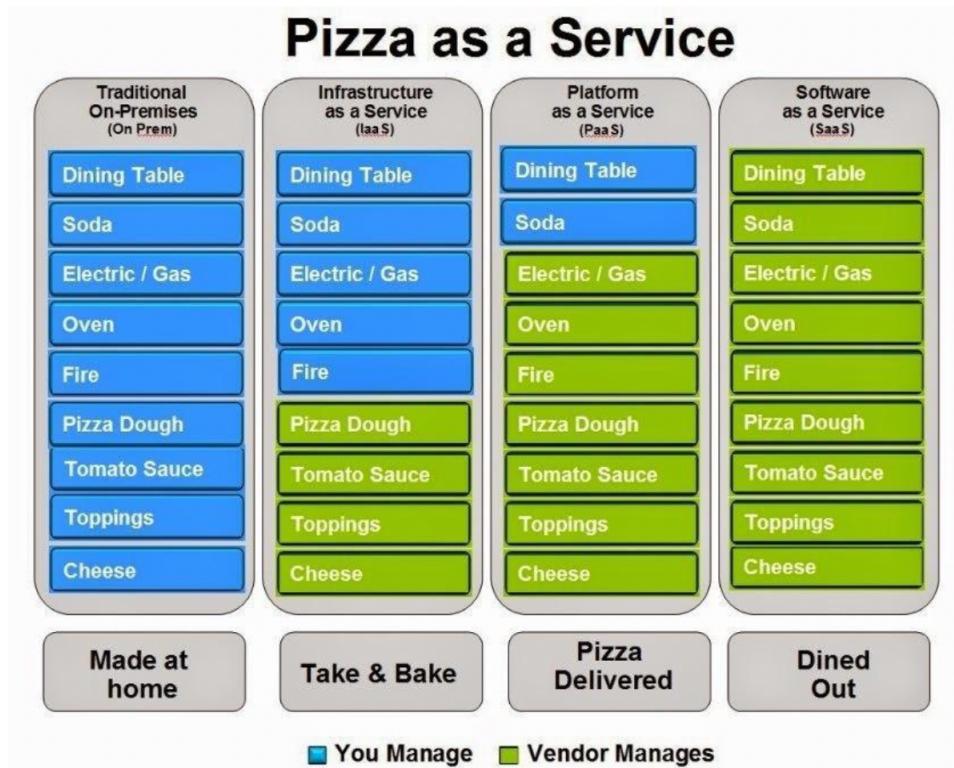


Figure 1.16: Pizza as a service.

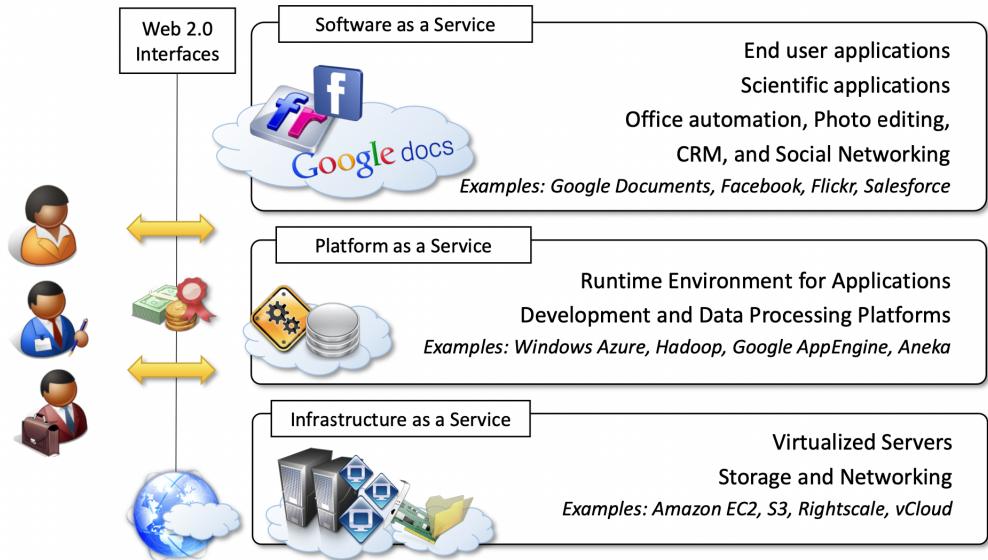


Figure 1.17: Model Examples.

# Chapter 2

## Virtualization technologies

È possibile virtualizzare qualsiasi tipo di risorsa IT: dai componenti base come CPU, RAM etc fino ad altri elementi o periferiche. È necessario sottolineare che perché una risorsa fondamentale venga virtualizzare, la risorsa sia effettivamente presente. Non si può chiaramente virtualizzare una CPU se non si dispone di una CPU fisica, questo ragionamento non si adatta per i pezzi non fondamentali come scheda di rete o le periferiche in quanto possono essere semplicemente emulate tramite software. Lo **strato di virtualizzazione** permette anche di virtualizzare uno specifico componente con caratteristiche diverse da quello fisico, ad esempio posso virtualizzare un processore a 32 bit da uno fisico a 64 bit.

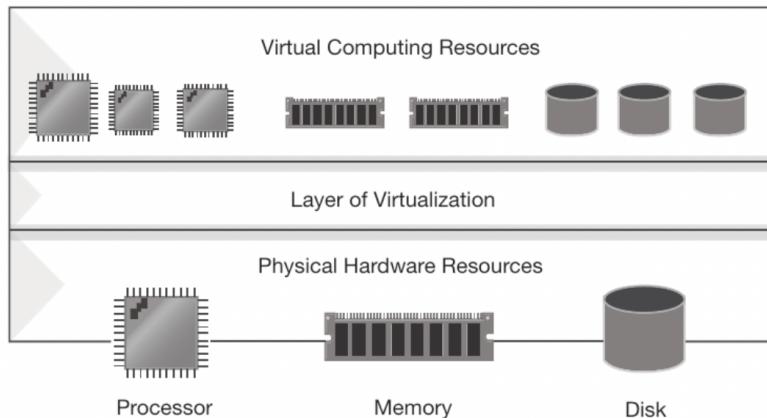


Figure 2.1: Resource virtualization.

Con il termine **virtualizzazione di macchine/server** si intende il concetto di creare una Virtual machine o un virtual server da una macchina fisica. Per Host System si intende la macchina fisica mentre per Guest System si intendono tutte le virtual machine create. Si parla al plurale perché, a differenza del paradigma classico dove la relazione fra computer fisico e SO era di uno a uno, attraverso la virtualizzazione si ottiene una relazione di uno a molti, da una macchina fisica si possono virtualizzare più di una VM. Ogni VM è isolata dalle altre e accede indirettamente all'hardware della macchina ospitante e fa funzionare le applicazioni attraverso il proprio sistema operativo.

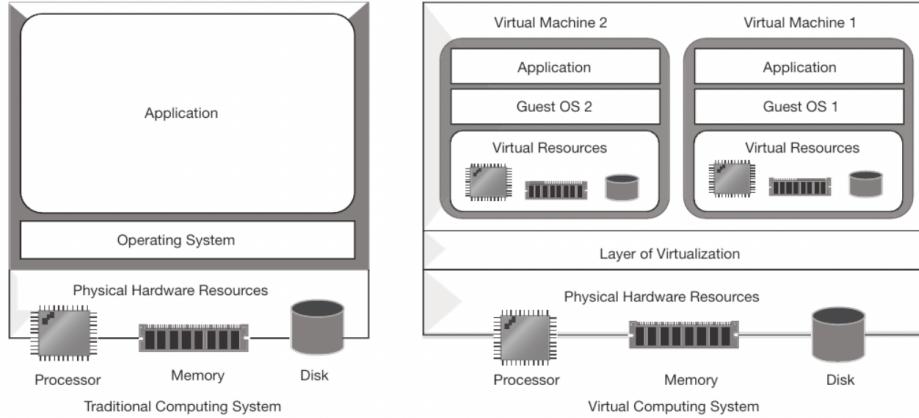


Figure 2.2: Machine/server Virtualization.

### 2.0.1 Hypervisor

Lo Strato di Virtualizzazione è chiamato **Hypervisor** o **Virtual Machine Monitor**. È l'addetto a fornire l'astrazione delle risorse alle macchine virtuali create sopra un host, esso è anche il responsabile della creazione delle Vm stesse. Ci sono due tipologie di Hypervisor:

- **Hosted Approach (tipo 2)**: un sistema operativo è installato sulla macchina host, il programma di Hypervisor gira sull' OS dell'host e crea le macchine virtuali. Si basa sulle funzionalità fornite dal OS installato sull'host. Un esempio è VirtualBox.
- **Bare Metal Approach (tipo 1)**: in questo approccio l'hypervisor lavora direttamente sull'hardware dell'host, facendo ciò è chiaro che devono essere implementati ad esempio i driver che prima erano messi a disposizione dell'OS. Questo approccio è solitamente usato dai cloud provider.

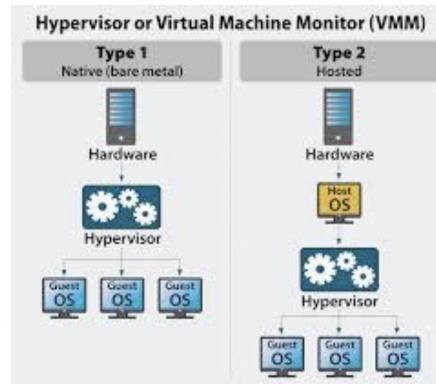


Figure 2.3: Two approaches of hypervisor.

Analizziamo i vantaggi e gli svantaggi delle due tipologie.

**Hosted approach:**

- *Vantaggi*: i driver per interfacciarsi con l'hardware fisico sono forniti dall'OS dell'host, è più facile l'installazione e la configurazione dell'hypervisor ed è compatibile con le sulle VM sono installabili la maggior parte dei sistemi operativi in quanto possono essere usati senza modifiche.
- *Svantaggi*: l'hypervisor non ha diretto accesso all'hardware, quindi ogni richiesta proveniente dalle VM deve passare dall'OS dell'host degradando le performance.

### Bare Metal Approach:

- *Vantaggi*: non essendo presente l'OS nell'host si ha un significativo aumento di prestazioni, inoltre per grandi applicazioni questo approccio è preferibile in quanto vengono forniti mezzi per la gestione delle risorse e della sicurezza.
- *Svantaggi*: si devono aggiungere manualmente i driver e di conseguenza non si possono usare nelle VM sistemi operativi stock ma devono essere appositamente modificati.

## 2.1 Virtualization types

Esistono diversi tipi di virtualizzazione e in questo paragrafo andremo a vederne alcuni nel dettaglio.

### 2.1.1 Full Virtualization

Viene chiamata anche *native virtualization* e in questa tipologia l'hypervisor simula o emula totalmente l'hardware sottostante. Il sistema ospite non si rende conto di essere in uno spazio virtualizzato, assume quindi di essere nel proprio hardware fisico. *La virtualizzazione nativa permette l'utilizzo dei sistemi operativi originali senza modifiche da utilizzare come OS nelle virtual machine*. È compito dell'hypervisor gestire le richieste OS-to-hardware durante l'esecuzione delle macchine ospiti(ad esempio, guest OS to physical hardware). In più il sistema operativo nella VM rimane completamente isolato dall'hardware sottostante.

### 2.1.2 Para-Virtualization

In questa tipologia una porzione della gestione della virtualizzazione viene spostata dall'hypervisor all'OS ospite, per questa motivazione le versioni normali dei sistemi operativi non possono essere usati in quanto devono essere modificati al fine di ampliare le azioni da poter svolgere, quest'azione di modifica si chiama porting. Analizziamo di seguito i pro e i contro di questo tipo di virtualizzazione:

- **Vantaggi**: il para-virtualization permette alle chiamate di funzioni dell'OS ospite di comunicare direttamente con l'hypervisor (senza quindi il bisogno di una traduzione in binario ad esempio), in questo modo diminuisce l'overhead sull'hypervisor e le performance generali aumentano drasticamente. Come secondo vantaggio i sistemi operativi ospiti non sono relegati ai driver delle periferiche forniti dal virtualization layer, questo perché l'hypervisor non contiene nessun driver in quanto essi sono devono essere implementati dal guest OS.

- **Svantaggi:** non possono essere utilizzate le versioni normali dei sistemi operativi, ma essi vanno modificati. Se per i sistemi open source non sussistono problemi, per quelli proprietari dobbiamo affidarci alle versioni compatibili rilasciate dall'azienda.

### 2.1.3 Hardware assisted Virtualization

Questo è un tipo particolare di full virtualization, dal 2006 i produttori di hardware hanno cominciato a produrre componenti capaci di supportare la virtualizzazione nativamente tramite hardware. Questo significa che le chiamate che partono dal sistema operativo ospite possono essere direttamente prese in carico dalla CPU senza che esse siano trasferite all'hypervisor. Non c'è quindi bisogno di para-virtualization o di traduzione binaria. Vengono quindi incrementate le performance generali della full virtualization a patto che vengano usate delle componenti compatibili con questa tecnologia.

### 2.1.4 Operating System Level Virtualization

Questo tipo di approccio è totalmente differente dalle tipologie di virtualizzazione discusse precedentemente. In questo tipo di tecnica di virtualizzazione non viene utilizzato nessun tipo di hypervisor e le VM sono create dal kernel del sistema operativo della macchina fisica. *Il kernel è quindi condiviso fra tutti i virtual server che girano sulla macchina fisica.*

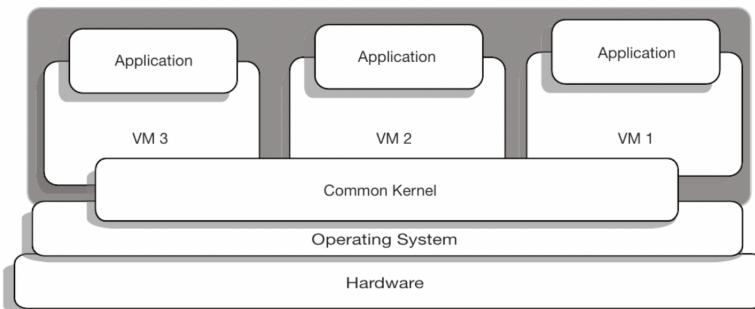


Figure 2.4: Operating system level virtualization.

Il kernel e l'OS della macchina host si occupano di creare molti virtual server logicamente distinti.

Analizziamo ora i vantaggi e le limitazioni:

- **Vantaggi:** questo approccio è più leggero in termini di peso computazionale e grazie a questa caratteristica è possibile per l'host fisico di poter reggere un numero di virtual server maggiore
- **Limitazioni:** tutte le VM hanno lo stesso sistema operativo e questo potrebbe portare a problemi di sicurezza

È importante menzionare la presenza anche di altri tipi di virtualizzazione oltre a quelle riguardanti alle macchine o alla virtualizzazione dei server. Si parla di network e storage virtualization:

- **Network virtualization:** è il processo di combinazione di network resources e network functionalities per creare un network virtuale, grazie a questo si possono far comunicare le VM. Spesso vengono utilizzate le infrastrutture fisiche per lo scambio dei dati.
- **Storage virtualization:** anche lo spazio di archiviazione può essere virtualizzato, ogni macchina può salvare i dati in dischi virtuali, i dati saranno quindi salvati in qualche disco fisico.

## 2.2 Emulation

L'emulazione è il processo grazie al quale si fa sì che un sistema ne imiti un altro. Questo significa che l'architettura di un sistema grazie all'emulazione può supportare le istruzioni di un'altra architettura. I software di emulazione hanno bisogno di convertire i dati in binario in modo che sia possibile eseguire i dati su un altro sistema, esistono due tipi per implementare l'emulazione:

- **Binary translation:** viene fatta una conversione totale in binario, la conversione può essere ricompilata per far sì che funzioni su un altro sistema. La traduzione può essere statica o dinamica.
- **Interpretation:** ogni istruzione è interpretata dall'emulatore quando l'operazione viene incontrata, questo metodo è più facile da implementare ma chiaramente più lento rispetto alla binary translation.

### 2.2.1 Virtualization VS Emulation

Analizziamo adesso le differenze fra emulazione e virtualizzazione: Con l'emulazione è possibile creare una simulazione completa dell'hardware in software, questo crea un ambiente per l'esecuzione di un sistema operativo. In questo ambiente posso girare sistemi operativi ospiti su un host che ha una tipologia diversa di architettura, grazie a questo metodo è possibile utilizzare OS creati per un'architettura su un'architettura diversa. Quest'ultima tipologia si chiama emulation-based virtualization che è differente dalla normale emulazione. Nella virtualizzazione canonica i set di istruzioni usate dal sistema virtuale e il sistema hardware sono gli stessi, quindi il codice della VM può essere eseguito direttamente dal sistema fisico. Non è necessaria la traduzione dell'informazione e, togliendo questo strato, le performance sono significativamente maggiori. Genericamente la virtualizzazione è molto più rapida rispetto all'emulazione.

## 2.3 Advantages and Disadvantages of Virtualization

Analizziamo i vantaggi e gli svantaggi della virtualizzazione partendo dai vantaggi.

### 2.3.1 Advantages

- Utilizzando un numero relativamente alto di VM si sfrutta al massimo l'hardware a disposizione, questo processo si chiama **server consolidation**.
- Può essere raggiunto un miglior utilizzo delle risorse in quanto la virtualizzazione aiuta a **diminuire i costi** delle componenti hardware e quelli dell'infrastruttura generale.
- Interponendo uno strato di virtualizzazione fra l'hardware e il software si **semplifica l'amministrazione del sistema** in quanto i due ambienti sono ben divisi fra di loro.
- Con la virtualizzazione viene **semplificata anche l'installazione del sistema** in quanto per creare una nuova VM si può clonare una VM già esistente saltando quindi tutta la parte della configurazione.
- Con la virtualizzazione abbiamo anche **tolleranza ai guasti e tempo pari a zero di manutenzione**. Questo è possibile perché le VM possono essere migrate su un hardware differente e possono anche essere sottoposte a backup.
- La **sicurezza è aumentata** in quanto ogni singola VM è isolata dalle altre.

### 2.3.2 Disadvantages

- Ogni singola macchina host rappresenta un **single point of failure** per le VM che girano su quella macchina host.
- C'è un **decremento nelle performance** in quanto una VM non potrà mai raggiungere le performance della macchina host dal momento in cui la VM non può accedere in maniera diretta all'hardware.

## 2.4 Recap Computer Architecture

Cominciamo con l'elencare i requisiti della virtualizzazione.

- **Equivalenza**: un SO in esecuzione su una VM sotto l'hypervisor dovrebbe avere lo stesso comportamento di quando è in esecuzione direttamente su una macchina.
- **Controllo delle risorse**: l'hypervisor deve essere in completo controllo delle risorse fisiche e il SO in esecuzione nella VM deve avere controllo delle risorse virtualizzate.
- **Efficienza**: la maggior parte delle istruzioni macchina devono essere eseguite senza l'intervento dell'hypervisor per avere più efficienza.

### 2.4.1 Multiprogrammazione

I requisiti appena elencati sono simili a quelli per la multiprogrammazione. Tramite la multiprogrammazione *emuliamo una macchina con molti processori (e altre periferiche) rispetto a quello/i nell'hardware reale*. Ogni processo viene eseguito sul proprio **processore virtuale (VCPUs)**, ogni VCPU è riservato per un processo.

Un **requisito** è che il processo deve avere l'impressione del completo controllo sul processore, e lui è l'unico processo in esecuzione su di esso.

### 2.4.2 VCPU Execution

Il SO implementa la multiprogrammazione creando una rappresentazione virtuale del processore che contiene una copia dello *stato del processore*, cioè i suoi registri. Quindi una macchina con un solo processore fisico emula un solo processore virtuale alla volta, questo si può fare caricando ogni volta un i registri del VCPU che si trovano nel processo virtuale.

Cosa sappiamo riguardo a:

- Cambio di contesto:
  - Viene salvato lo stato dell'host CPU nella rappresentazione del VCPU.
  - Viene scelto un nuovo VCPU da mandare in esecuzione e viene caricato il suo stato nei registri dell'host CPU.
  - Viene eseguita un'istruzione speciale per saltare all'indirizzo salvato in ret e poi ritorna in user mode.

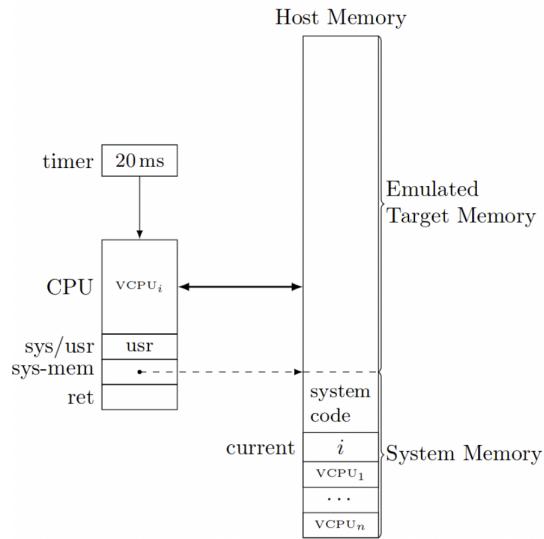


Figure 2.5: Example of memory in multiprogrammed environment.

- Necessità di supporto hardware:

- Interruzioni

- Protezione della memoria
  - \* Usr/sys registry: flag che specifica se la CPU è in user o system mode
  - \* Sys-mem registry: contiene l'indirizzo di partenza della memoria protetta
  - \* Ret register: contiene l'indirizzo di memoria quando si ritorna da un'interruzione
- Privilegi: user e sistema
- Memoria virtuale: meccanismo introdotto per permettere ai processi di astrarre la memoria realmente disponibile su un sistema usando un spazio di indirizzi virtuali. Ogni processo avrà il proprio spazio virtuale, potrà utilizzare indirizzi continui quando in realtà possono non esserlo, inoltre il processo avrà l'impressione di avere l'accesso a tutta la memoria fisica.
  - MMU: i differenti spazi di indirizzi virtuali devono essere mappati sulla memoria fisica che è condivisa tra tutti i processi. La traduzione dell'indirizzo virtuale a fisico viene fatto dalla MMU, dal registro PTBR si accede alla tabella delle pagine e tramite numero della pagina (bit più significativi dell'indirizzo virtuale) si trova una parte dell'indirizzo fisico al quale si aggiunge l'offset. Possono anche esistere più tabelle delle pagine dove in ognuna troviamo una parte di traduzione dell'indirizzo.

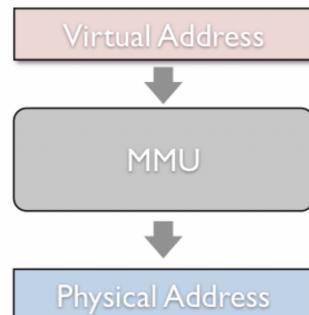


Figure 2.6: Address translation.

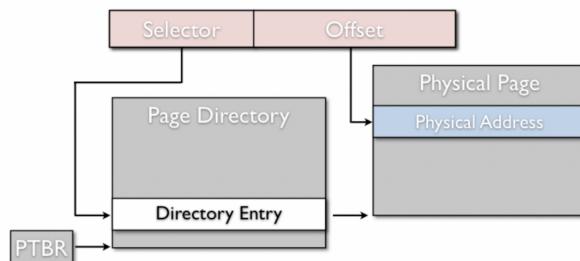


Figure 2.7: Page Table.

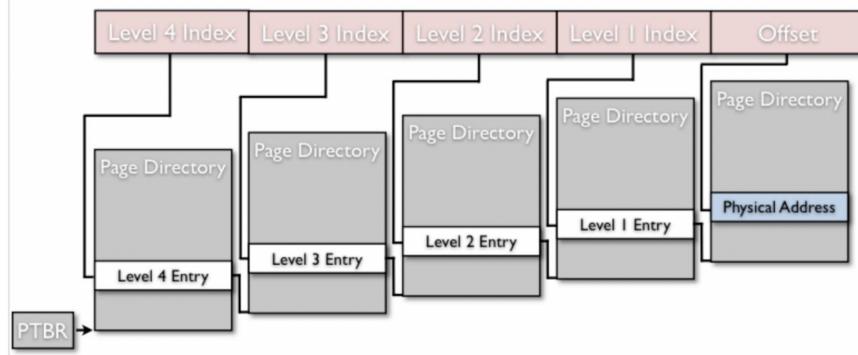


Figure 2.8: Multi-level Page Table.

- Segmentazione: dato che lo spazio virtuale è più esteso della mem fisica dobbiamo usare anche utilizzare lo swap. Se una pagina non è presente in mem fisica verrà generato un page fault, allora la pagina presente nello swap verrà messa in mem fisica.

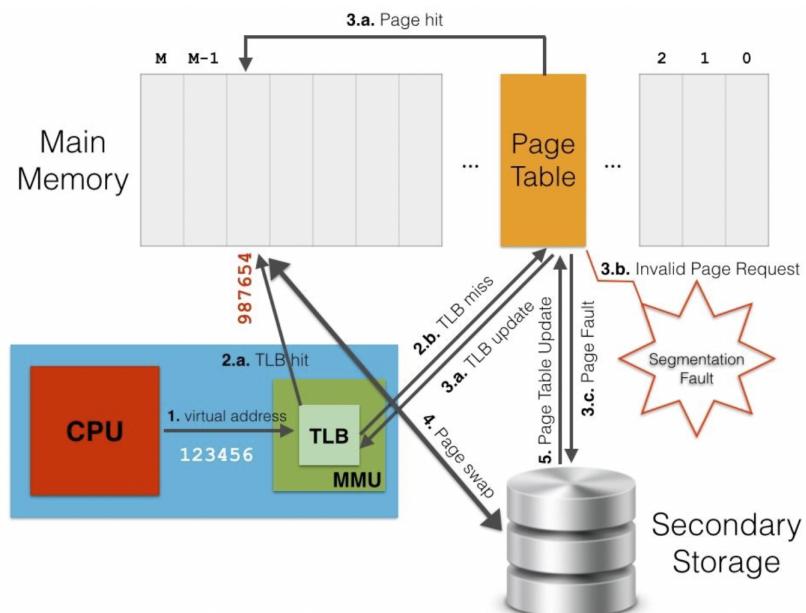


Figure 2.9: Segmentation.

- Interruzioni/eccezioni: sono usate per notificare al sistema eventi che necessitano attenzione durante l'esecuzione di un programma (eg page fault). Loro alternano la normale esecuzione di un programma passando da user mode al kernel mode e andando ad seguire le funzioni rispettive all'interruzione/eccezione definite nel kernel.

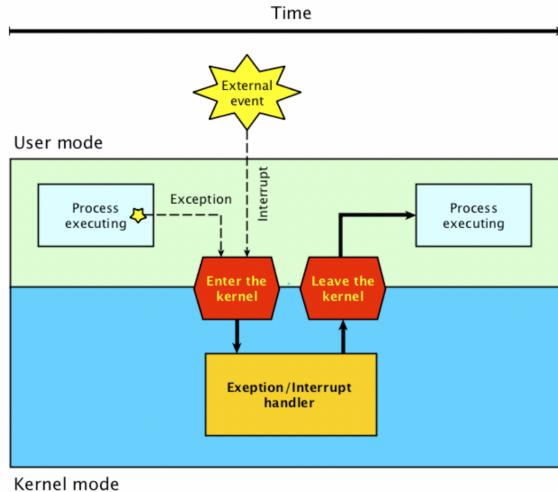


Figure 2.10: Interrupt/Exceptions.

- Eccezioni: sono interne ed asincrone, vengono usate per gestire errori interni a programmi (divisione per zero, indirizzo errato, page fault, ...).

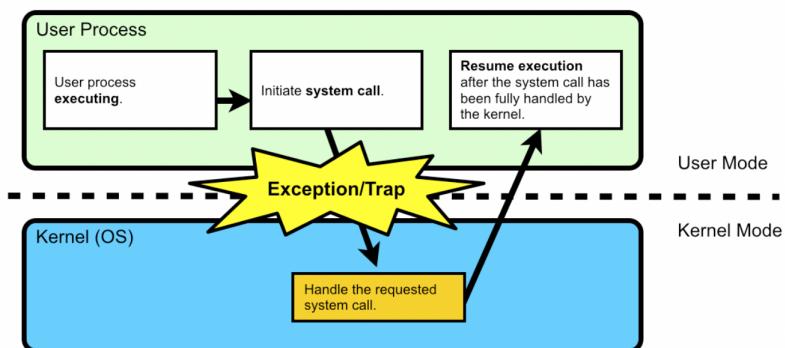


Figure 2.11: Exception.

- Interruzioni: usate per notificare alla CPU eventi esterni, ad esempio la pressione di un tasto sulla tastiera. Nella tabella dei descrittori delle interruzioni (IDT) sono presenti delle entry che correlano per ogni interruzione o eccezione i relativi handler.

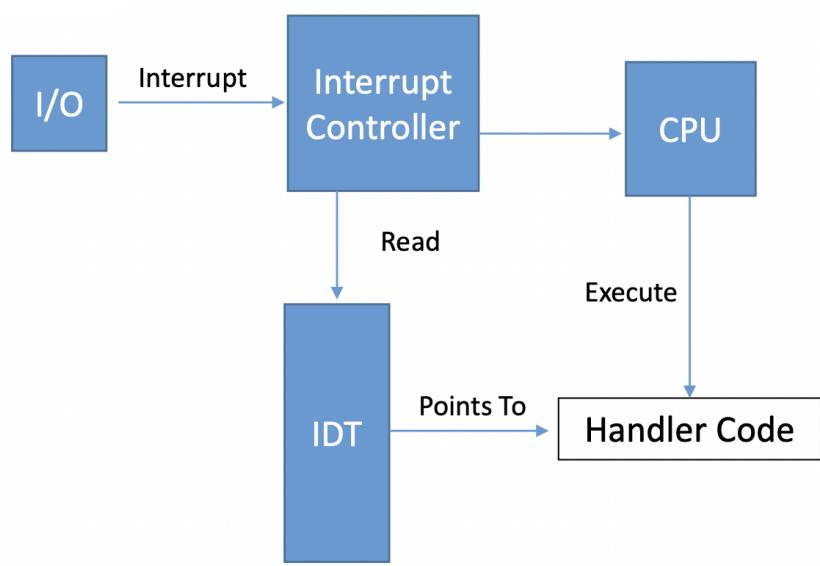


Figure 2.12: Interrupt workflow.

## 2.5 Full Virtualization

La multiprogrammazione permette ad ogni processo di avere l'impressione di avere il completo controllo di CPU e RAM, in modo simile l'hypervisor o il virtual machine monitor (VMM) devono dare alla VM l'impressione di avere il completo controllo sulle risorse fisiche (CPU, RAM, I/O) per ottenere la virtualizzazione a livello sistema (quello che viene fatto con la multiprogrammazione). Il ruolo della VMM è simili al ruolo del kernel del SO, le sue funzionalità sono però più estese del solo supporto alla multiprogrammazione, infatti deve nascondere l'ambiente virtuale alla VM.

### 2.5.1 Hypervisor

Usiamo l'**hypervisor** per creare una rappresentazione virtuale del sistema riutilizzando i meccanismi hardware già disponibili. Se il guest e l'host hanno la stessa architettura, per la maggior parte del tempo, il codice del SO/software in esecuzione sulla VM viene direttamente eseguito sull'hardware senza la necessità di traduzione binaria (per l'emulazione) che introduce overhead.

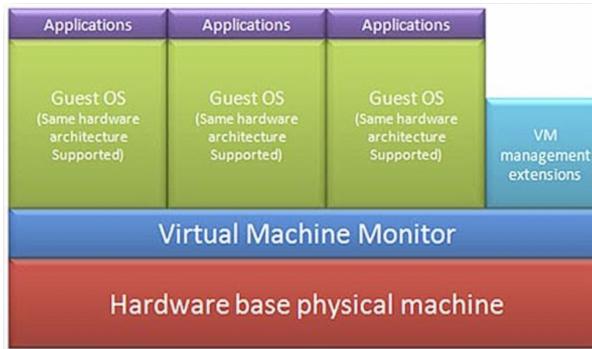


Figure 2.13: Hypervisor.

### 2.5.2 Virtualizing CPU

La VMM adotta la stessa tecnica usata per creare VCPU nell'ambiente di multiprog:

- VMM code viene eseguito nello spazio sistema/kernel, il guest SO code viene eseguito nello spazio user.
- La VMM carica lo stato del VCPU nel processore dell'host e poi fa girare il codice finché la CPU non trova un'istruzione che non può essere eseguita direttamente.
- Quando un'istruzione deve essere emulata è necessario fare un cambio di contesto per concedere il processore al VMM che emula l'istruzione target e successivamente viene ricaricato lo stato del VCPU nel host CPU così che può continuare la propria esecuzione.
- Se ci sono più VM in esecuzione nello stesso host, la VMM potrebbe implementare un timer per il cambio di contesto in modo da garantire equità tra le varie VMM in esecuzione.

- In questo caso la VMM seleziona un'altra VM da mettere in esecuzione caricando lo stato della sua VCPU nell'host CPU.

### 2.5.3 Guest OS Execution

Abbiamo detto che la VMM ha un ruolo simile al kernel del SO nella multiprog, ma differisce per diversi aspetti:

- Il codice della guest VM vorrebbe avere il completo controllo dell'hardware dell'host e eseguire istruzioni non solo a livello utente ma anche a livello di sistema.
- Perciò la VMM deve emulare l'intero processo, sia le funzionalità dell'utente sia quelle del sistema usate per la multiprog, con lo scopo di supportare il guest OS.
- Il codice in esecuzione nel guest OS dovrebbe avere accesso a registri e istruzioni privilegiate.

Nell'ambiente multiprogram ogni volta che un'istruzione privilegiata viene eseguita da un processo user viene ucciso dal kernel, non vogliamo che la VMM uccida la VM ma che gestisca l'eccezione.

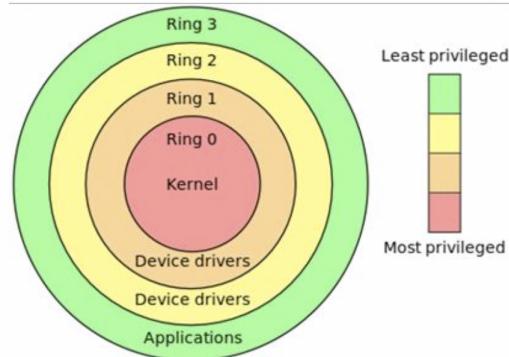
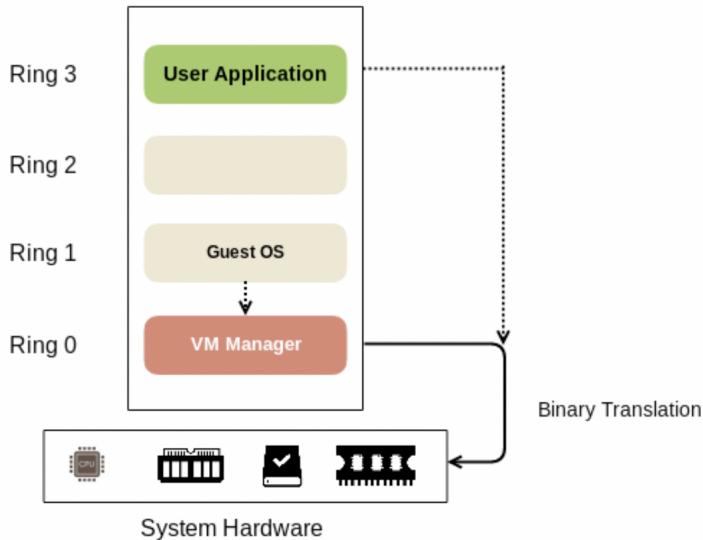


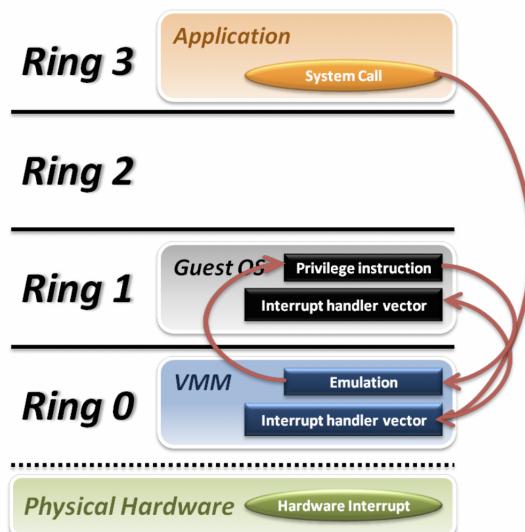
Figure 2.14: Privilege rings.

## 2.5.4 Trap and emulate virtualization model

Il sistema ospite (SO e codice applicazioni) viene eseguito nell'user-space: Ring 1 (SO) e Ring 3 (User App). La VMM *sfrutta le eccezioni/trap* per fare un cambio di contesto dalla VM alla VMM. Siccome il guest OS viene eseguito nello userspace, *ogni volta che un'istruzione privilegiata viene eseguita un'eccezione viene lanciata*. Per esempio un'istruzione che accede ai device di I/O, un'istruzione relativa ad un'interruzione o un'istruzione che coinvolge la MMU.



La VMM (il codice trap) utilizza quindi la traduzione binaria per eseguire le istruzioni privilegiate del SO, emulandone il comportamento. Dopo l'esecuzione di un'istruzione privilegiata il controllo viene ridato al codice del guest OS. **Se il set delle istruzioni privilegiate sono limitate le performance non sono impattate significativamente.**



## 2.5.5 Virtualizing physical memory

La memoria fisica della VM viene implementata utilizzando una parte della memoria dell'host, una parte della memoria dell'host sarà invece riservata alla VMM. La VM deve avere accesso solo alla parte di memoria che gli è stata assegnata e deve pensare che l'indirizzo di partenza sia 0. Per fare tutto ciò usiamo la MMU dell'host, *essa conosce il range di indirizzi assegnati ad ogni VM*. Se c'è un pagefault la VMM si occupa di trovare la pagina e caricarla in memoria. Ci sono comunque delle operazioni che richiedono il cambio di contesto, come ad esempio la modifica della tabella delle pagine.

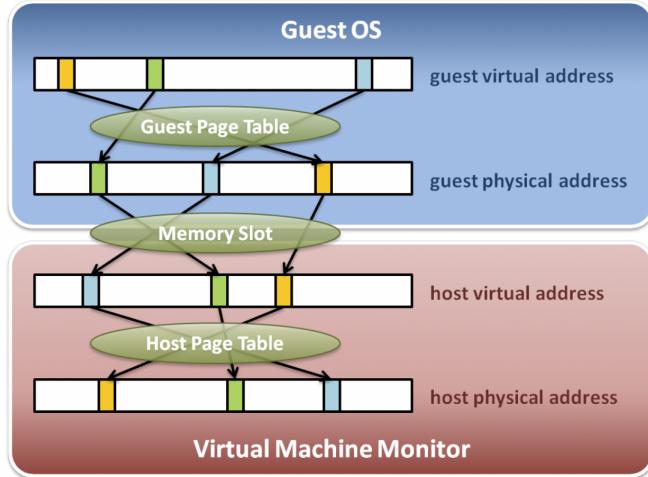


Figure 2.15: Virtualizing Physical Memory.

## 2.5.6 Virtual MMU

La MMU del guest traduce un indirizzo virtuale (V) in fisico (F), questo però è un indirizzo fisico per la VM e non per l'host (F'). Questa traduzione è implementata tramite una funzione G. Dato che la memoria della VM occupa una parte della mem fisica dell'host, l'MMU dell'host tramite la funzione H può tradurre F in F' trovando l'effettiva traduzione in indirizzo fisico dell'host. Per fare tutto ciò ogni VM necessita di una **Virtual MMU** che traduce direttamente l'indirizzo virtuale V in indirizzo fisico dell'host F' sfruttando la funzione  $G \circ H$ .

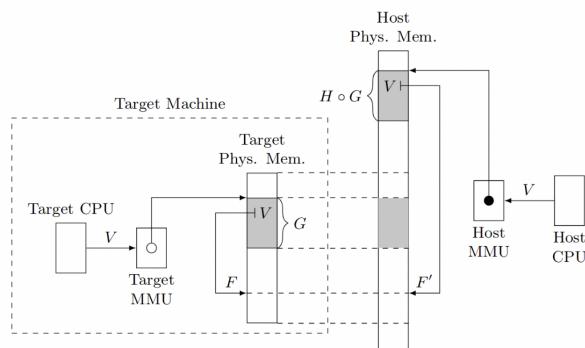


Figure 2.16: Virtual MMU.

## 2.5.7 Brute Force Method

La tabella delle pagine è modificata con l'aggiunta di un livello (chiamato shadow page table) che implementa la funzione H ( $F \rightarrow F'$ ), ad es. la traduzione dall'indirizzo fisico del guest all'indirizzo fisico dell'host.

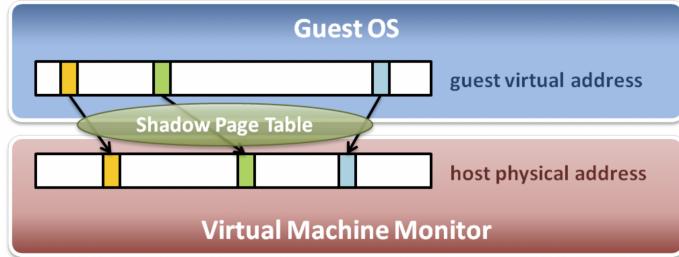


Figure 2.17: From the guest physical address to the host physical address.

### Shadow tables

Le tabelle shadow sono protette in scrittura, ed un eventuale accesso da parte della VM causerà una VM exit. Questo metodo causa però un certo overhead per via dei cambi di contesto per una traduzione continua degli indirizzi.

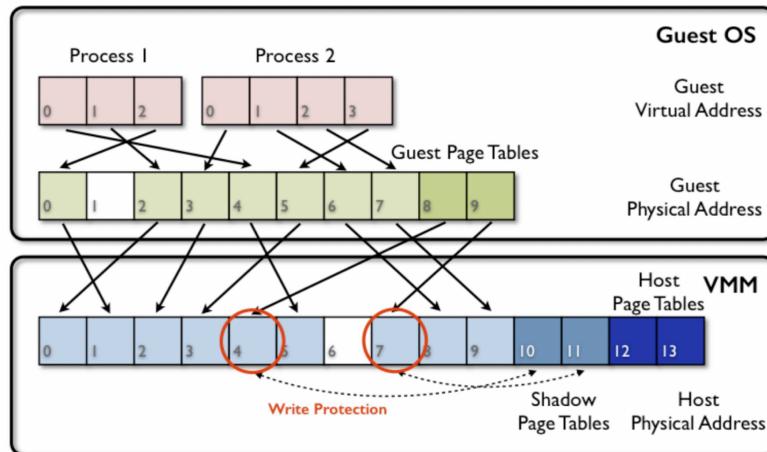


Figure 2.18: Shadow Tables.

### Brute force methods

Per introdurre questo nuovo livello e tenerlo aggiornato, la VMM deve riconoscere (con la trap) tutte le possibili azioni collegate alla MMU e alla tabella delle pagine, nello specifico cambiare il PTBR per cambiare tutta la page table oppure cambiare alcune entry della page table presente. Ogni volta che vengono intercettate queste azioni la VMM prende il controllo aggiornando la shadow table e in caso di cambio del PTBR viene aggiunta una nuova shadow table.

## 2.5.8 Virtualizing I/O devices

Le istruzioni di I/O solitamente sono privilegiate. In ambiente multiprogr i processi accedono allo spazio I/O tramite un'astrazione implementata dal kernel. Dato che una VM non può accedere direttamente alla periferica, la VMM deve dare l'impressione alla VM che essa sia in controllo dell'hardware. Per fare ciò è necessario che la VMM **emuli** l'hardware creandone una rappresentazione virtuale con le strutture dati necessarie.

Le *rappresentazioni virtuali (Device model)* delle periferiche sono accedute dalla VM, la VMM accede/aggiorna le rappresentazioni virtuali ogni volta che c'è una lettura/scrittura. Essendo il VMM l'unico controllore delle periferiche del sistema fisico, esso può tradurre le istruzioni di I/O dirette alle periferiche virtualizzate in istruzioni di I/O dirette alle periferiche fisiche o può emularle direttamente.

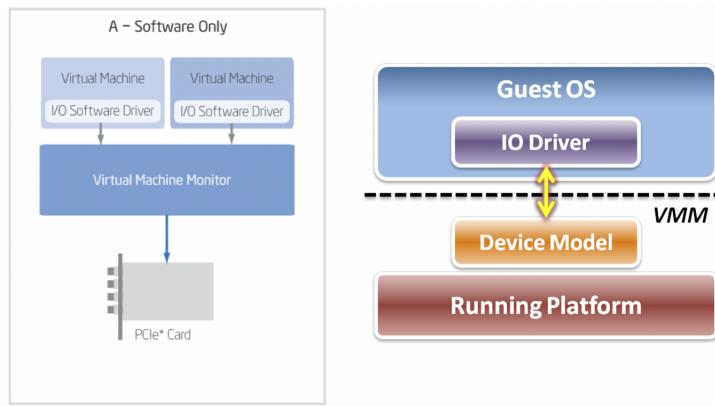


Figure 2.19: Virtualizing I/O Devices.

### Device models implementation

I device model dipendono dal tipo di hypervisor:

- Tipo 1: sono implementati come parte del VMM.
- Tipo 2: sono programmi che girano su user space come servizio apparte.

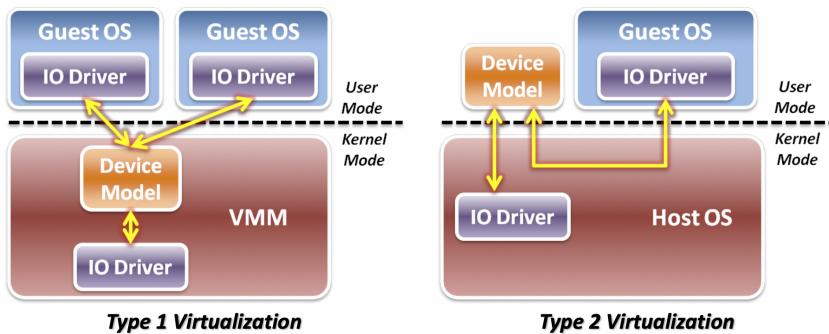


Figure 2.20: Device model depends on hypervisor type.

### 2.5.9 Interrupt management

La VMM si occupa anche delle interruzioni che arrivano dai device fisici, deve gestirle in modo trasparente. La VMM avrà una tabella di descrittori delle interruzioni (inaccessibile dal guest) che la CPU dell'host usa per gestire le interruzioni. Ogni guest VM ha la propria tabella dei descr di interruzioni gestita dal SO guest. La VMM si occupa di gestire le interruzioni che sono generate dai device virtuali emulati dalla VMM per il guest OS.

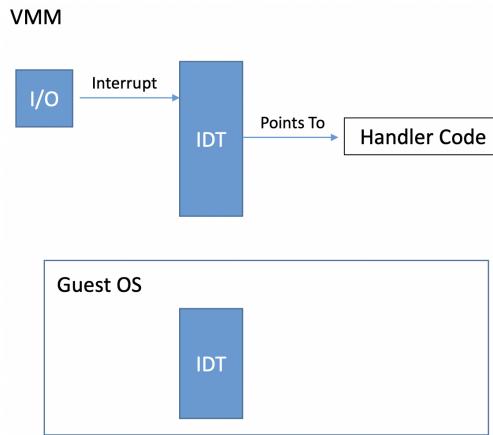


Figure 2.21: Interrupt Management.

### 2.5.10 Virtualizing interrupts

Quando la VMM vuole emulare la ricezione di una interruzione in una VM, deve guardare nella tabella delle interruzioni del guest e fare le operazioni necessarie per eseguire il codice dell'handler nel guest OS:

- Salvare lo stato attuale
- Cambiare l'instruction pointer al valore della prima istruzione dell'handler
- Quando la VMM ridà il controllo alla guest VM verrà eseguito il codice dell'interruzione

La VMM deve considerare che la guest CPU può disabilitare l'interruzioni, in questo caso deve aspettare finchè non sono di nuovo attive prima di emulare la ricezione di un'interruzione.

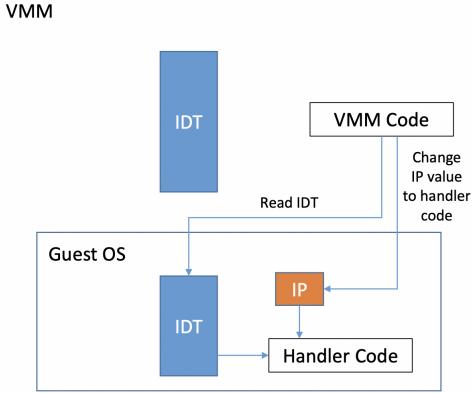


Figure 2.22: Virtualizing Interrupt.

### 2.5.11 Hardware input

VMM è l'unico ad avere accesso all'hardware reale, le interruzioni provenienti dai device sono gestite dalla VMM. Le interruzioni hardware mandano in esecuzione un'interruzione alla VMM (specificata all'interno del Interrupt handler vector) anche se la VM è in esecuzione, queste interruzioni vengono inviate alla VM emulandole via software.

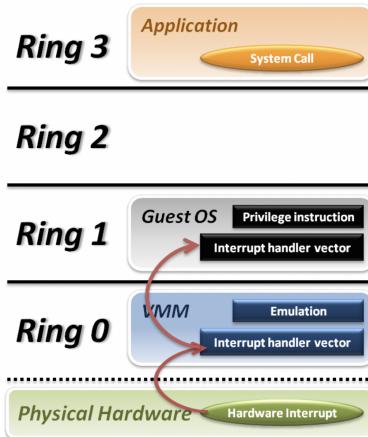


Figure 2.23: Hardware Interrupt.

### 2.5.12 QEMU

Quick Emulator è un software open source VMM che emula l'hardware della macchina, supporta un ampio set di emulazioni hardware. Può eseguire un sistema operativo guest con un set di istruzioni diverso da quello dell'host attraverso la traduzione binaria o eseguire una VM con lo stesso set. In quest'ultimo caso QEMU ha un acceleratore per accelerare l'emulazione al fine di eseguire alcuni del codice del sistema operativo guest come codice in modalità utente. È ampiamente adottato in molti altri progetti open source utilizzando la virtualizzazione.

## 2.6 Hardware assisted virtualization

### 2.6.1 Performance penalty

Le tecniche di virtualizzazione consistono nel far girare l'OS ospite in uno spazio dedicato all'utente. Ogni volta che l'OS ospite vuole "spostarsi" nello spazio dedicato al kernel avviene un cambio di contesto per far girare il VMM (Hypervisor) che ha il compito di imitare l'esecuzione di istruzioni privilegiate o di altro tipo(ad esempio quello di I/O). Questo metodo è usato ad esempio da VirtualBox per far funzionare VMs in ogni sistema host. Anche se le operazioni del kernel dell'OS ospite sono limitate nel tempo, esse rappresentano una grossa penalità in termini di performance, questo perché viene introdotto un notevole overhead per quanto riguarda la traduzione binaria e l'emulazione.

### 2.6.2 Hardware Assisted Virtualization

Per risolvere questo problema e conseguentemente per migliorare le performance, dal 2006, Intel e AMD hanno inserito delle peculiarità nelle loro CPU per supportare in modo efficiente l'implementazione del VMM-Hypervisor. Queste funzioni specifici che sono volte alla riduzione del overhead dovuto alla virtualizzazione.

#### Root and non-root modes

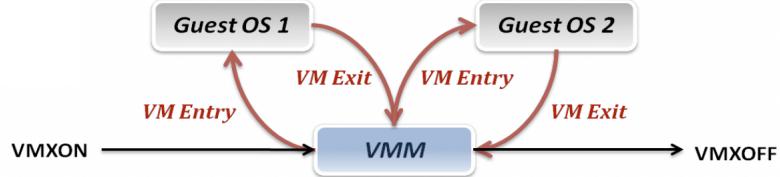
Concentratosi sulle funzioni introdotte nelle CPU Intel, la tecnologia VMX introduce 2 *nuove modalità operative*: **root** e **non-root**. Essendo queste due modalità ortogonali a quelle già esistenti nelle vecchie CPU Intel, il risultato saranno quindi 4 possibili combinazioni che sono di seguito elencate in ordine di privilegio:

- Root/system
- Root/user
- Non-root/system
- Non-root/user

La **root mode** è pensata per il VMM che gira sulla macchina host, mentre la **non-root mode** è stata pensata per il sistema operativo ospite che gira all'interno della VM.

Lo scopo principale di queste due modalità è quello di inserire delle limitazioni controllate tramite hardware alle azioni che possono essere eseguire dall'OS ospite. Infatti quando un OS ospite prova a eseguire un'istruzione che violerebbe l'isolamento della VM o che richiederebbe un'emulazione software, l'hardware può intercettarla (trap) o passare il comando al VMM.

Vengono quindi aggiunte due nuove funzioni assembler per la gestione VMLAUNCH e VMRESUME che sono permesso solo in root/system. La VM può anche ritornare alla root mode facendo tornare in esecuzione il VMM per diverse ragioni, chiamate **VM Exits**.



Queste due nuove modalità ci aiutano a aumentare le performance in quanto riducono la necessità di emulare il software. Facciamo un esempio pratico con INT e IRET che normalmente permettono di cambiare contesto da utente a sistema e viceversa. Senza il supporto hardware la loro esecuzione è imitata dal VMM che si fa carico di modificare lo stato della VM. Con il supporto hardware queste operazioni possono essere eseguite senza il supporto del VMM e quindi riducendo l'overhead. Vediamo come funziona: quando viene fatta una chiamata a una INT il sistema operativo esegue un cambio di contesto da non-root/user a non-root/system e viceversa per la IRET che cambia da non-root/system a non-root/user. Ecco di seguito un esempio della differenza dell'esecuzione di una INT, a sinistra senza supporto hardware e a destra con il supporto.

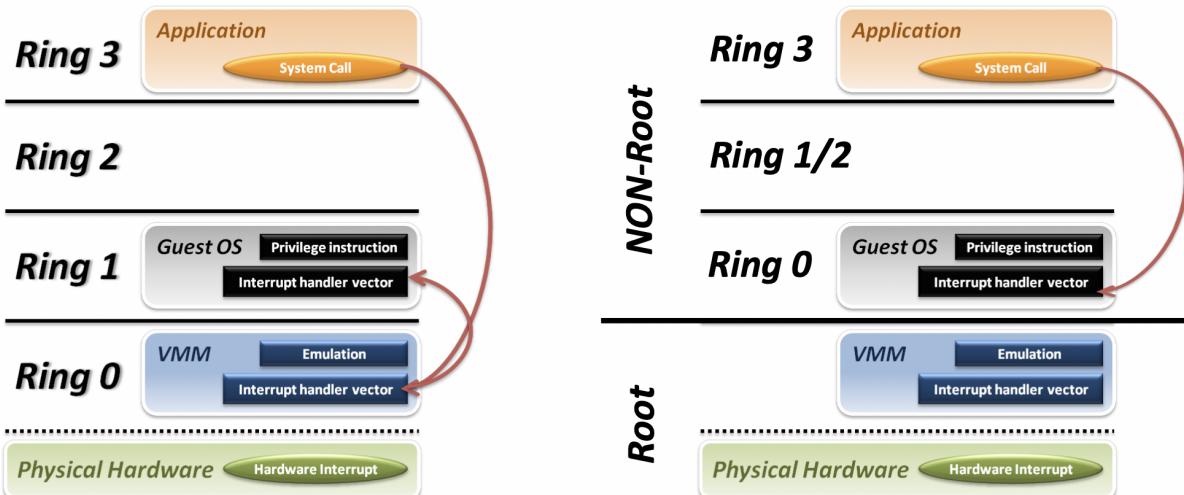


Figure 2.24: Example: INT execution on the VM.

Abbiamo analizzato l'utilità della doppia modalità nella zona di non-root, andiamo adesso ad analizzare l'utilità della doppia modalità nella zona root. Grazie a questa dicotomia è possibile implementare l'architettura in modo che il VMM sia parte di un OS standard che gira su una macchina host. Questo è importante perché le applicazioni devono girare a root/user mentre il kernel e le operazioni del VMM vengono eseguite a root/system. Ecco un esempio:

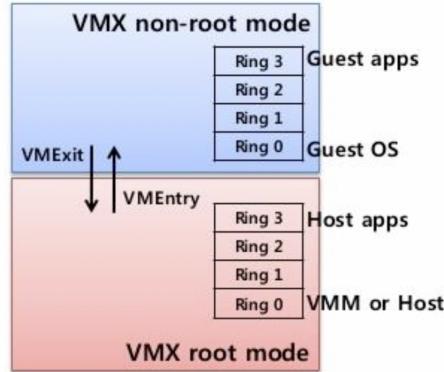


Figure 2.25: Root and non-root mode.

### 2.6.3 Virtual machine control structure

La tecnologia VMX di Intel ha introdotto anche una **virtual machine control structure** (VMCS), essa è una struttura di controllo che contiene tutte le informazioni associate allo stato di una VM. Questa estensione aggiunge una serie di istruzioni grazie alle quali i dati salvati nella struttura possono essere manipolati che sono usati quando viene fatto un cambio di contesto per mandare in esecuzione il VVM o per rimettere in esecuzione la VM.

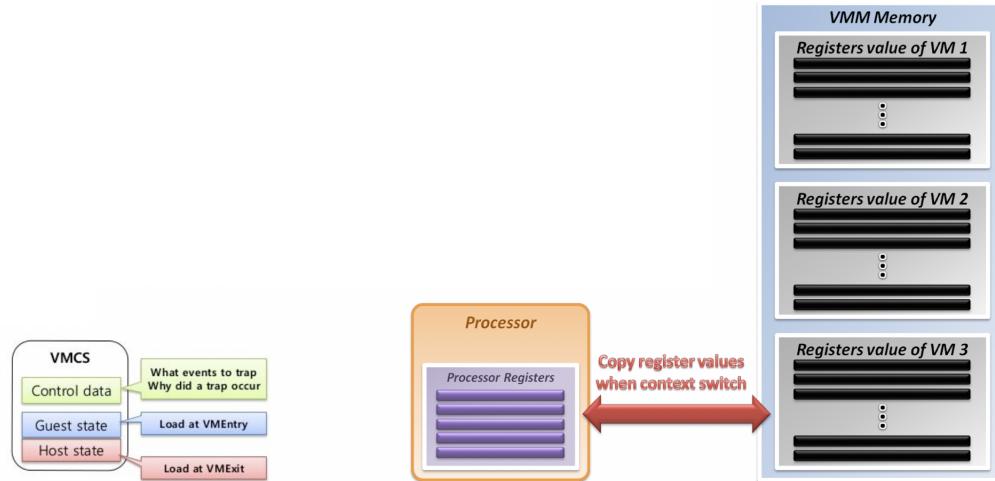


Figure 2.26: VMCS.

La struttura dati ha diversi campi e di seguito ne elenchiamo le categorie con una breve descrizione del contenuto che poi verrà spiegata più avanti

- **Guest State:** salva lo stato del processore virtuale associato alla virtual machine. Durante la VM launch il lo stato viene caricato nel processore, mentre lo stato viene salvato al seguito di una VM exit.
- **Host state:** salva lo stato del processore fisico prima della VM launch, può contenere lo stato del VMM prima della VM launch. Lo stato viene ricaricato durante una VM exit.

- **VM execution control:** questo campo specifica quali operazioni sono permesse e quali non sono permesse durante la non-root mode, queste operazioni possono causare una VM exit.
- **VM exit control:** contiene diversi campi e flag per specificare qualche comportamento opzionale nella transizione da non-root a root.
- **VM enter control:** come il campo precedente ma durante la transazione da root a non-root
- **VM exit reason:** contiene le informazioni che hanno causato l'ultima VM exit.

### **Guest/Host states**

Insieme alle altre informazioni il guest state contiene il puntatore alla prima operazione che deve essere eseguita dopo una VM launch, nel host state viene salvato il valore da inserire nel IP per riprendere l'esecuzione del VMM dopo una VM exit. Nel VM exit reason viene salvato il codice della ragione della VM exit, grazie a questo codice il VMM può avere informazioni riguardo il comportamento della VM

### **VM execution control**

In questo campo sono presenti vari flag in particolare quelli riguardanti la **gestione delle interruzioni e dell'I/O**. I flag sono utilizzati in particolare per il peripheral passthrough. I flag sono:

- Uno che determina quello che deve succedere quando la CPU riceve un'interruzione esterna mentre sta lavorando in non-root mode. *Grazie a questo flag il VMM può specificare se la VM può gestire l'interruzione o deve essere chiamata una VM exit.*
- Un flag che determina *se qualche istruzione critica deve causare un VM exit o no.*
- Un altro *flag specifica se un'operazione di I/O sono permesse all'interno di una VM o devono causare una VM exit.*

### **VM enter control**

Questo campo è composto da altri che possono essere usati dal VMM per generare delle false interruzioni esterne o per iniettare eccezioni e fault. A questo scopo *il VMM scrive il vettore delle interruzioni desiderate all'interno del VM enter control, durante la VMM enter il processore svolgerà tutte le azioni di risposta alle interruzioni*, in particolare: salva lo stato in guest stack e cerca nella guest interrupt descriptor table l'indirizzo della routine che gestisce l'interruzione.

## **2.6.4 RAM management**

Lo scopo dell'hypervisor è quello di far credere ad ogni VM di avere una memoria contigua tutta per lei, attraverso l'hypervisor però ogni segmento della memoria è salvato in memoria fisica il spazi di locazione che magari non sono contigui.

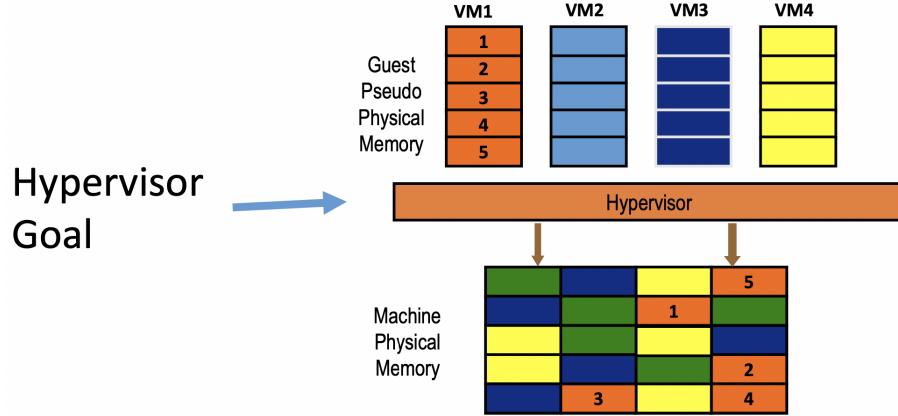


Figure 2.27: RAM Management.

### 2.6.5 Extended page table

Sia Intel che AMD hanno aggiunto delle estensioni ai loro componenti in modo da semplificare la **virtualizzazione della memoria virtuale dell'ospite**. In particolare hanno aggiunto il meccanismo dell'**extended page table** che è visibile all'host ma non alla VM. L'hardware che implementa la MMU sull'host è esteso per avere due puntatori PTBR, uno che punta alla page table della VM ospite e implementa il G mapping, il secondo punta alla page table creata dal VMM e implementa l'H mapping. L'hardware esteso è responsabile di applicare la composizione del G mapping e dell'H mapping sequenzialmente in modo da trovare l'indirizzo fisico.

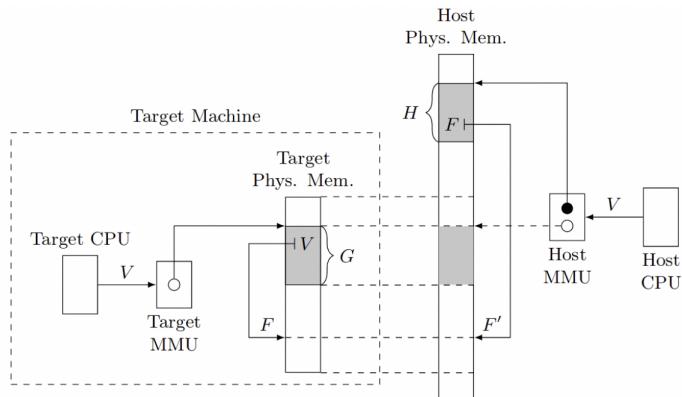


Figure 2.28: Extended Page Tables.

Si parte quindi dall'ospite e tramite la sua MMU si trova in V l'indirizzo di F (sempre dentro la memoria della VM). Con F si accede alla MMU dell'host e all'indirizzo F viene trovato F' cioè l'indirizzo fisico finale. Tramite questa estensione non è più necessario invocare una VM exit ogni volta che la VM modifica una page table. Questa estensione, tuttavia, è pagata con un incremento consistente in costo computazionale in quanto si aggiungono molti accessi in memoria per tradurre un indirizzo. Per mitigare questo costo computazionale, recentemente

le MMU sono state dotate di cache in modo da limitare gli accessi in memoria per gli indirizzi tradotti recentemente.

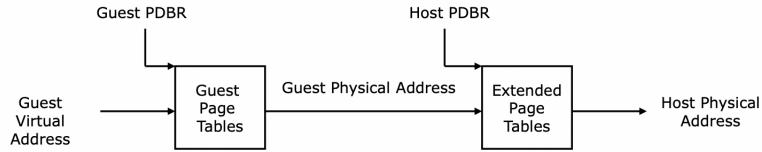


Figure 2.29: Extended Page Tables.

### 2.6.6 Hardware passthrough

L'emulazione dell'I/O è la causa maggiore di problemi di performance in VM perché il VMM deve partecipare in ogni interazione tra VM e periferiche di I/O (spesso emulate), spesso risultando in una serie di VM exit. Una soluzione a questo problema è quella di dare accesso diretto e esclusivo a una periferica. Questo approccio è chiamato **passthrough** e consiste, appunto, nel mappare una periferica direttamente a una VM ospite senza passare attraverso la traduzione. Questo approccio elimina direttamente il problema della chiamata alla VM exit, però la risorsa hardware non è virtualizzata e non può essere condivisa con altre VM. In più la virtual machine si trova a doversi interfacciare direttamente con una componente hardware senza che ci sia uno strato che fa da intermediario. *Per poter utilizzare questo approccio, tuttavia, è necessario che un supporto esplicito da parte dell'hardware*, in particolare sono necessarie due funzionalità principali:

- **Mappaggio diretto** dello **spazio di memoria** di I/O(fisico) nella memoria fisica virtuale della VM, in modo che l'OS ospite possa scrivere direttamente nei registri della periferica.
- **Assegnamento diretto** delle **interruzioni** collegate alla periferica, in questo modo la VM può gestire tutte le interruzioni provenienti dalla periferica mentre il VMM gestisce tutte le altre.

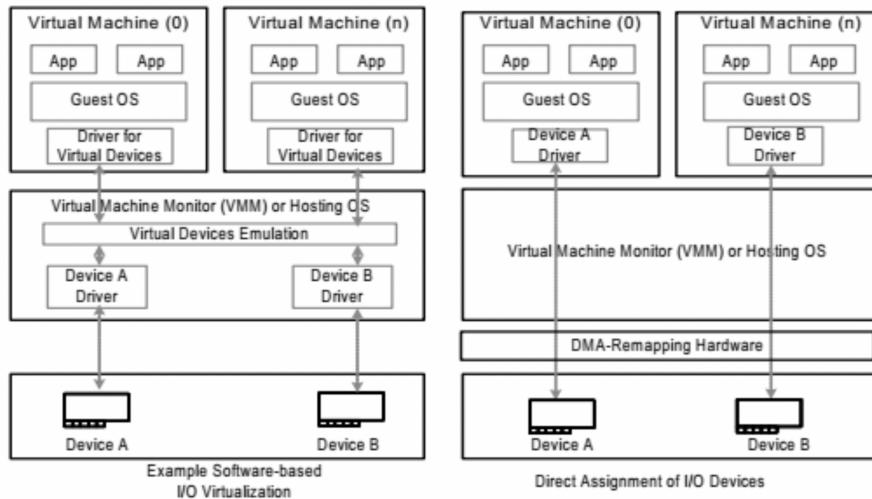


Figure 2.30: Passed-through device.

## 2.6.7 I/O mapping

Per leggere/scrivere da/su i registri della periferica designata per il passthrough vogliamo che la VM sia abilitata a eseguire azioni di lettura o di scrittura nello spazio di I/O **senza l'intervento del VMM**, mentre per le altre periferiche vogliamo che il sistema le intercetti e conceda il controllo al VMM per la loro esecuzione. Per risolvere questo problema con le CPU Inter, quindi tramite la tecnologia VMX, viene inclusa una **I/O bitmap**. Essa contiene un bit per ogni possibile indirizzo di I/O che supporta hardware passthrough. Quando una periferica è settata in passthrough per una determinata VM la I/O bitmap viene modificata in modo da settare tutti i bit corrisponde di allo spazio di I/O della periferica in modo da garantire l'accesso diretto.

Per capire bene il funzionamento facciamo un esempio: in non-root mode la CPU prima controlla la I/O bitmap quando viene eseguita una lettura o una scrittura nello spazio di I/O. Se i bit sono settati allora l'azione viene portata a termine, in caso contrario viene concesso il controllo al VMM tramite una VM exit.

## 2.6.8 Interrupts

La tecnologia Intel VMX permette due modi di gestire le interruzioni:

- L'interruzione causa una VM exit e quindi è gestita dal VMM
- L'interruzione viene gestita direttamente dalla VM senza generare una VM exit.

Il secondo metodo è quello utilizzato assieme all'hardware passthrough, l'interruzione è gestita direttamente dalla VM che manda in esecuzione il codice di gestione apposito. Con questo approccio l'hardware è capace di cercare l'indirizzo dell'handler direttamente all'interno dell' Interruption Vector Table(IVT) dell'OS ospite. L'IVT dell'OS ospite è gestita in maniera autonoma senza l'intervento del VMM. La VM, però, può ricevere un'interruzione della periferica **anche quando la VM non sta girando**, per questo motivo il sistema deve essere in grado di gestire le interruzioni provenienti dalla periferica abilitata al passthrough anche quando la VM non è attiva.

## 2.6.9 VM state

Riallacciandoci al discorso del precedente paragrafo è importante analizzare gli stati in cui una VM può trovarsi.

- **Running state:** una VM si trova in questo stato quando la sua CPU sta effettivamente utilizzando la Host CPU.
- **Ready state:** quando la CPU della VM è stata stoppata in quanto la CPU della macchina Host è usata dal VMM o da qualche altra VM
- **Halted state:** una VM entra in questo stato quando viene eseguita un'istruzione di *HLT* da parte di un software ospite all'interno della VM

Una VM può cambiare stato tra Running e Ready a causa di una VM exit o a causa di politiche di schedulazione del VMM, mentre può uscire dallo stato di Halted solo a seguito di un'interruzione ricevuta dalla VM CPU.

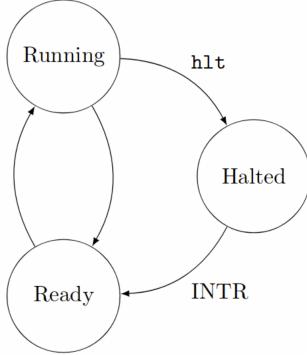


Figure 2.31: VM States.

### 2.6.10 Interrupt management

Quando è presente nel sistema una periferica abilitata al passthrough le interruzioni devono essere gestite in base allo stato in cui la VM si trova, più precisamente:

- **VM running:** l'interruzione è gestita direttamente dal processore, che attraverso la IVT dell'OS ospite mette in esecuzione l'handler corrispondente, tutto questo senza che il VMM interferisca.
- **VM ready:** in questo caso l'interruzione deve essere salvata e dovrà essere gestita solo quando la VM CPU sarà nuovamente assegnata all'HOST CPU. Ricordiamo che la Host CPU al momento sta eseguendo il VMM oppure una VM diversa.
- **VM halted:** l'interruzione deve essere salvata e la VM deve essere sposta nello stato VM ready.

Per salvare le richieste di interruzione si utilizza il **posted interrupt mechanism** presente sulle CPU più recenti. Il meccanismo permette l'assenza di intervento del VMM. Questo meccanismo utilizza altri due elementi **per ogni VM**:

- Interrupt Remapping Table (**IRT**)
- Posted Interrupt Descriptor (**PID**)

#### Interrupt remapping table

La IRT permette di ridirezionare da parte VMM ogni interruzione hardware in maniera dinamica. La IRT è composta da un entrata per ogni possibile richiesta di interruzione, ogni entrata mappa la richiesta al vettore di una VM o al PID (se sono abilitate le posted interrupts). Quindi ogni volta che viene scatenata un'interruzione tramite l'IRT si decide se l'interruzione deve essere gestita dall'Interrupt Vector o dal PID. Vediamo adesso come è strutturato il PID.

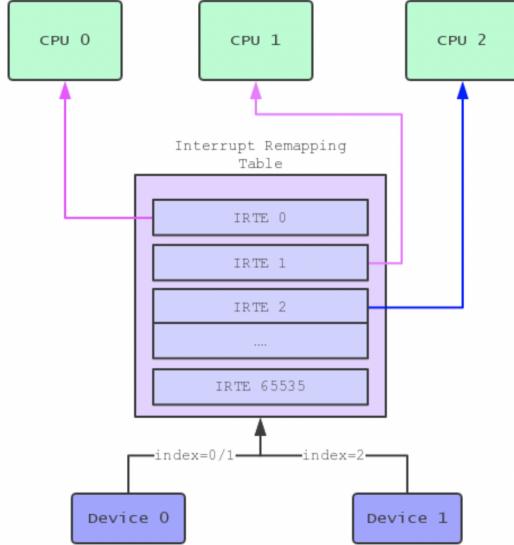


Figure 2.32: Interrupt Remapping Table.

### Posted interrupt descriptor

Il PID contiene informazioni riguardanti lo stato della VM e salva le informazioni sull'interruzione scatenata per una notifica e una gestione asincrona della stessa. Contiene tutti i campi necessari per implementare il posted interruption mechanism, più nello specifico sono presenti i seguenti campi:

- **Posted interrupt request (PIR)**: una struttura nella quale sono poste le interruzioni.
- **Suppress Notification (SN)**: un flag che determina se il controllore dopo aver postato l'interruzione deve notificare la CPU (SN=0) o no (SN=1).
- **Notification Vector (NV)**: un vettore che punta al vero vettore delle interruzioni per notificare alla VCPU della VM.

### Interrupt delivery

Quando il controllore delle interruzioni ha bisogno di mandare un'interruzione alla VM e il meccanismo delle posted interrupt è abilitato, l'entrata corrispondente nella IRT punta ad una istanza di un PID. In questo caso la IRT esegue le seguenti operazioni:

- Setta i bit necessari nella PIR
- Se SN = 1 (VM è nello stato pronto) allora non fa nulla. Altrimenti:
  - Se la VM è in esecuzione, la IRT interrompe il processore usando il vettore NV
  - Se la VM è halted si scatena il risveglio della VM

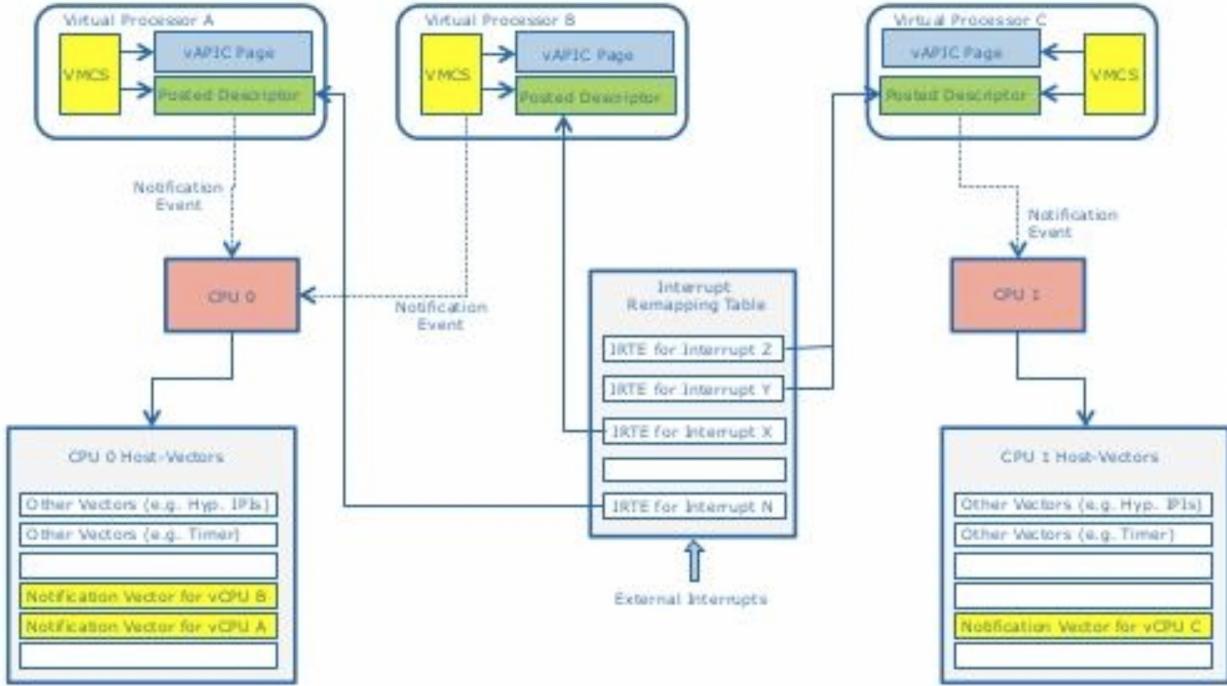


Figure 2.33: Interrupt Delivery.

### 2.6.11 PID update

Quando la VM cambia stato, il VMM deve aggiornare lo stato del PID coerentemente:

- Quando la VM va in **Running** state, VMM setta SN = 0 e setta NV ad un vettore chiamato Active Notification Vector che punta alla IDT (Interrupt Descriptor Table) della VM.
- Quando la VM va in **Ready** state, il VMM setta SN = 1.
- Quando la VM va in **Halted** state, il VMM setta SN = 0 e setta NV ad un vettore chiamato Wakeup Notification Vector che scatena il risveglio della VM.

*Quando il VMM cambia lo stato di una VM in Running state, il VMM deve anche controllare il PIR e, se ci sono dei bit settati, deve anche cambiare il NV come detto sopra. In questo modo il processore può processare le interruzioni che erano state inviate quando la VM non era in Running state.*

### 2.6.12 DMA devices

I dispositivi DMA sono dispositivi di I/O che hanno bisogno di leggere direttamente dalla RAM, la CPU da le istruzioni di lettura o scrittura al device per leggere o scrivere solo in un piccolo sotto insieme di registri, a seguito di ciò la CPU preleva i dati dalla RAM. I device DMA hanno accesso diretto alla memoria fisica.

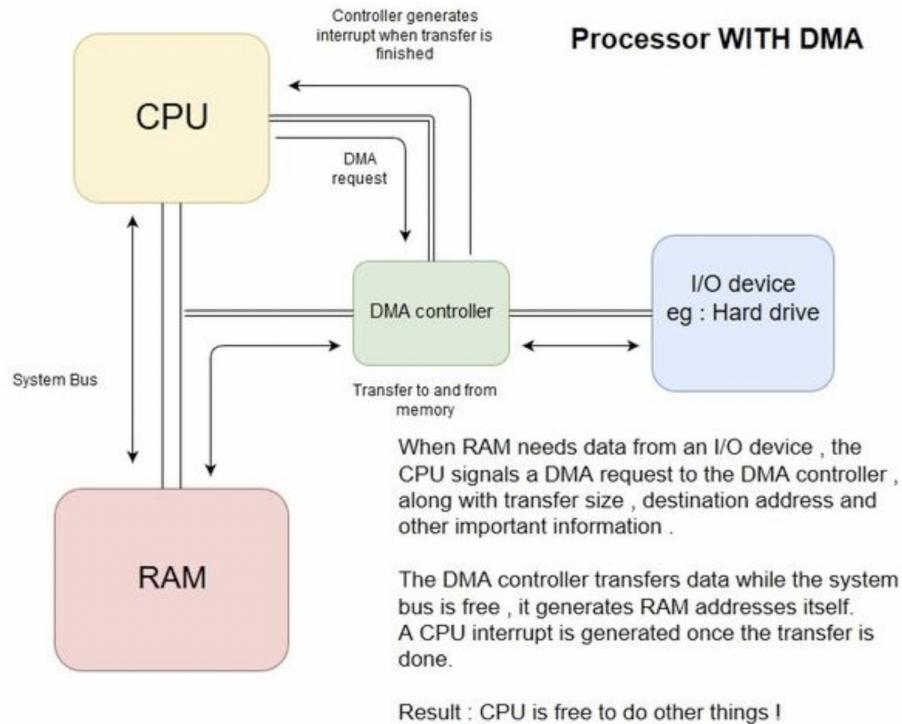


Figure 2.34: Processor with DMA.

### 2.6.13 IOMMU

Per poter tradurre indirizzi virtuali in indirizzi fisici, i device DMA utilizzano una MMU specifica per l'I/O. La IOMMU è responsabile della traduzione di indirizzi virtuali in indirizzi fisici per far in modo che il device di I/O possa leggere o scrivere da o su la RAM. Questa speciale MMU è stata introdotta per la prima volta nei sistemi multiprogrammati e poi è stata utilizzata per la memoria virtuale.

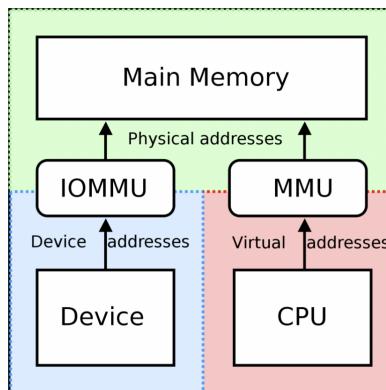


Figure 2.35: IOMMU.

Oltre che per la virtualizzazione, la IOMMU può anche causare interruzioni di page fault se le pagine target non sono presenti nella RAM.

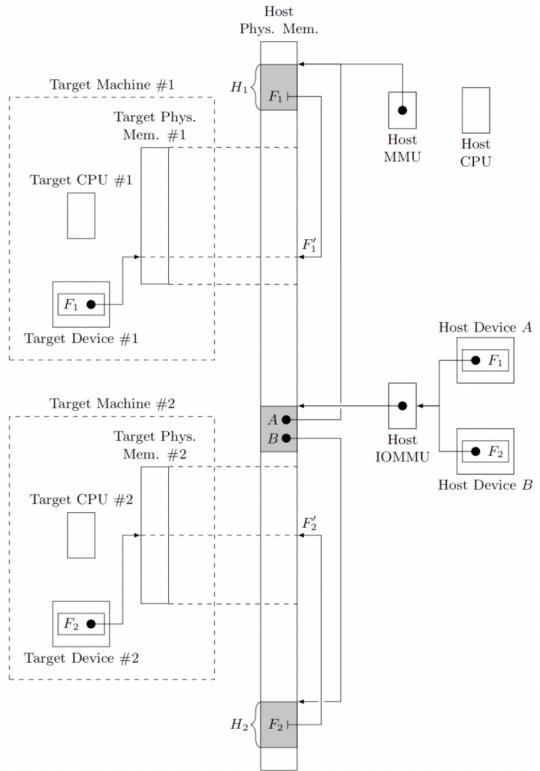


Figure 2.36: IOMMU and Virtualization.

### 2.6.14 Nested virtualization

La virtualizzazione annidata consiste nel far funzionare un hypervisor all'interno di una VM che a sua volta gira grazie ad un ulteriore host hypervisor. Questo tipo di virtulizzazione è usato per la maggior parte per fare testing anche se, al giorno d'oggi, può essere un'alternativa per la produzione in quanto vengono abbattuti i costi per aggiornare l'infrastruttura. Può infatti permette un cloud consumer di metter su una sua infrastruttura IaaS sopra un set di VM che girano su una infrastruttura IaaS già presente.

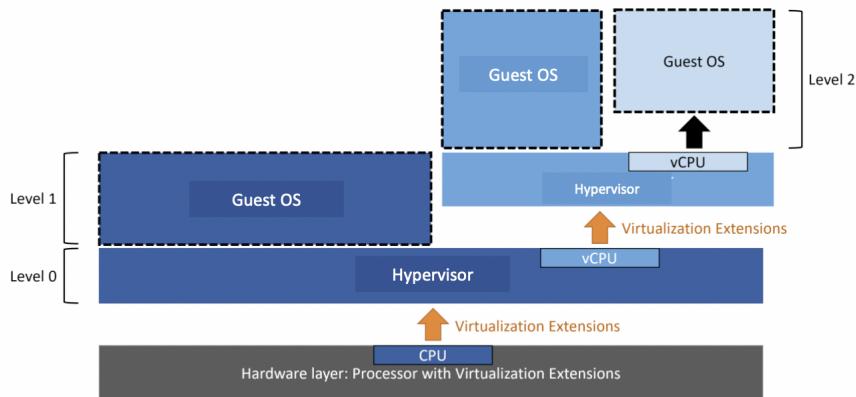


Figure 2.37: Nested Virtualization.

Questo tipo di virtualizzazione ha però bisogno del supporto specifico da parte dell'hardware, infatti AMD e Intel possono supportare **solo un livello di virtualizzazione**. Molteplici livelli di virtualizzazione possono essere compresi in uno singolo, ad esempio le VM che girano grazie al Guest Hypervisor girano come le altre VM che girano grazie all'Host Hypervisor. *L'hardware che accede il Guest Hypervisor è quello che viene emulato dall'Host Hypervisor.*

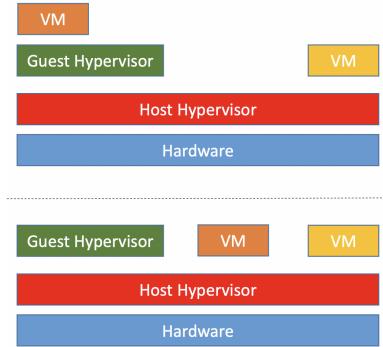


Figure 2.38: Nested Virtualization.

## 2.6.15 KVM

KVM(Kernel-Based virtualization) è un'infrastruttura di virtualizzazione del Kernel di Linux aggiunta per la prima volta nel 2007. KVM sfrutta la virtualizzazione dell'hardware per far girare le VM all'interno di un ambiente già virtualizzato in maniera quasi identica all'hardware fisico. KVM non gira come un programma normale all'interno di Linux ma utilizza il kernel per girare. Si comporta come un driver per la gestione delle istruzioni emanate dall'hardware.

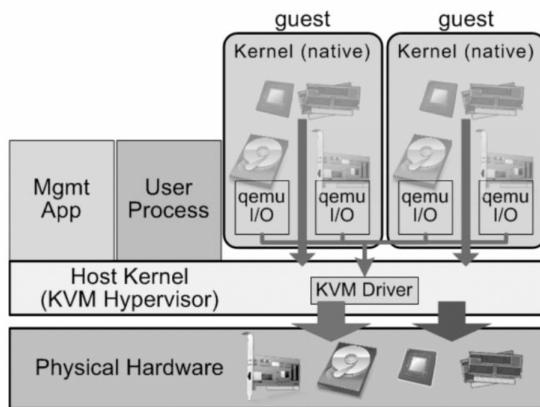


Figure 2.39: KVM.

## 2.7 Para-Virtualization

La full virtualization richiede la virtualizzazione dell'intera architettura, l'hardware virtuale è esposto alla VM, nella quale un SO viene **eseguito senza modifiche**. Le istruzioni del guest OS che non possono essere eseguite direttamente, sono emulate dalla VMM. Senza il supporto hardware, per la virtualizzazione si ha un gran numero di cambi di contesto (tra VM e VMM). In aggiunta, molte istruzioni devono essere emulate.

Un approccio alternativo (prima che venisse introdotto il supporto hardware) è la **paravirtualizzazione**. In un sistema paravirtualizzato, abbiamo ancora la VMM, ma l'hypervisor viene implementato partendo dall'assunzione che il guest OS può essere modificato e può essere fatto consapevole del fatto che è in esecuzione dentro una VM. Le modifiche al guest OS sono solitamente limitate al kernel (le applicazioni che girano all'interno della VM sono ancora inconsapevoli).

### 2.7.1 Paravirtualization Architecture

Lo scopo è di introdurre modifiche al guest OS in modo da prevenire istruzioni che non possono essere eseguite direttamente (ie IO instruction) oppure funzionalità in modo da non richiedere l'emulazione (ie memory management). Queste istruzioni o funzioni sono sostituite con delle chiamate a specifiche API messe a disposizione dalla VMM tramite un'interfaccia chiamata Virtualization Layer.

La VMM è responsabile per l'implementazione delle istruzioni/funzioni necessarie e per la creazione della rappresentazione virtuale dell'hardware, eventualmente anche la gestione del real hardware.

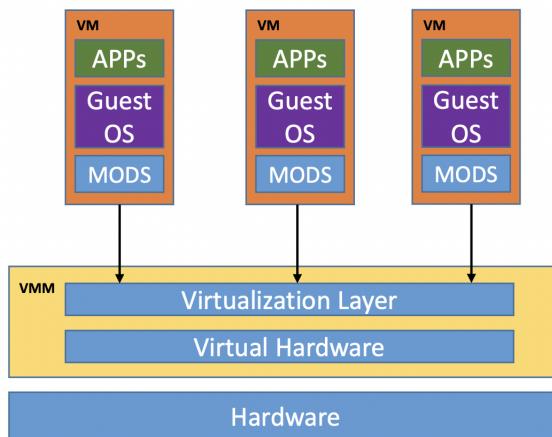


Figure 2.40: Paravirtualization Architecture

### 2.7.2 Paravirtualization Implementation

L'emulazione è evitata modificando il guest OS e invocando le **Hypercalls**, cioè le funzioni fornite dalla VMM che sostituiscono le istr/funz non virtualizzabili. Il codice delle hypercalls e la VMM sono eseguite in system mode, quindi possono eseguire istruzioni privilegiate e interagire direttamente con l'hardware.

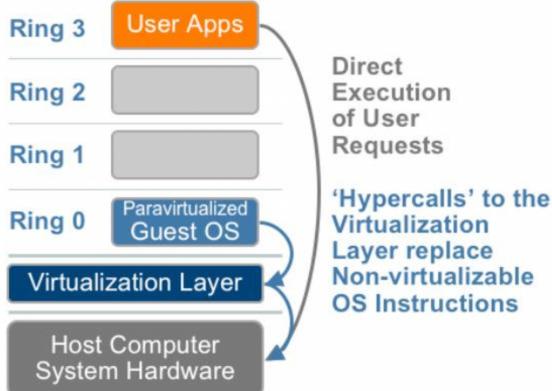


Figure 2.41: Rings Structure

### 2.7.3 Paravirtualization Pros and Cons

Analizziamo in questo paragrafo i vantaggi e gli svantaggi della paravirtualization.

- Vantaggi:
  - Incrementa significamente le performance della virtualizzazione, l'emulazione viene eliminata
  - Non richiede supporto hardware specifico
- Svantaggi:
  - Supporta solo un SO modificato
  - Alcune risorse dell'OS più chiuse non vengono supportate

I sistemi paravirtualizzati sono diventati molto popolari all'inizio per via del considerevole aumento delle performance. Con l'avvento del supporto hardware per la virtualizzazione non hanno più avuto un significante vantaggio sulla full virtualization, quindi ad oggi non sono così popolari. I sistemi paravirtualizzati sono stati convertiti a full virtualization.

### 2.7.4 XEN

E' il più popolare hypervisor basato sulla paravirtualizzazione. Rilasciato nel 2003, introdotto per ridurre l'overhead della full virtualization, i sistemi x86 non erano facili da virtualizzare per lo standard hypervisor trap-and-emulate.

Xen si basa su un set di modifiche al kernel Linux, così da far girare il SO linux più efficientemente tramite la paravirtualizzazione.

Come detto sopra l'avvento del supporto hardware ha reso la paravirtualiz meno conveniente, per questo oggi xen supporta sia la para che la full virtualization tramite supporto hardware. Usando questo metodo ogni SO (non solo quelli modificati) possono essere eseguiti su Xen.

## XEN architecture

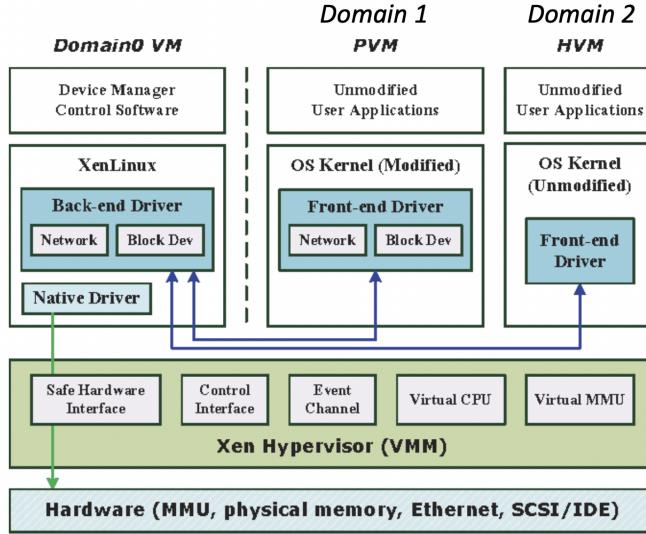


Figure 2.42: XEN Structure

Xen è un piccolo kernel isolato dalla macchina e garantisce il diretto controllo sull'hardware. Sopra a Xen, a livello di privilegio più basso, abbiamo i domini. Teoricamente tanti domini quanti sono necessari, all'interno di ognuno di essi abbiamo un intero SO con le sue applicazioni. I domini sono isolati tra loro e sono usati per implementare le macchine virtuali. Un dominio speciale, Dom 0, ha accesso alle Xen API per creare e distruggere gli altri domini. Dentro il Dom 0 può essere installato un qualsiasi SO Linux e possono essere sviluppati degli strumenti ad hoc per gestire gli altri domini. I domini possono avere accesso diretto a device di I/O, oppure possono usare full o para virtualization.

### 2.7.5 Virtualization overhead

Un caso d'uso per la virtualizzazione è la condivisione sicura di risorse hardware tra differenti applicazioni che girano sullo stesso sistema. L'**isolamento** delle applicazioni dovrebbe essere fornito dal SO, anche se recentemente i SO sono diventati meno efficaci nell'assicurare questo isolamento, di conseguenza le configurazioni tipiche garantiscono l'isolamento facendo girare le applicazioni su macchine virtuali differenti.

L'overhead della virtualizzazione non è trascurabile, sia in termini di memoria che di tempo di processazione:

- Della VMM
- Del SO dove le applicazioni vengono eseguite

Spesso accade che differenti VM condividono lo stesso guest OS, di conseguenza abbiamo più istanze dello stesso SO eseguito sullo stesso hardware.

### 2.7.6 Lightweight virtualization

La paravirtualizzazione è ancora caratterizzata dall'overhead introdotto dalla VMM. In aggiunta, in caso di multiple istanze dello stesso SO dentro le VM e il tutto sullo stesso hardware, ci saranno quindi multiple istanze del kernel, di processi del SO e librerie che gireranno sulla stessa macchina. Recentemente è stata introdotta la virtualizzazione leggera (lightweight virtualization) con lo scopo di **creare un ambiente leggero di esecuzione** per servizi e applicazioni che non richiedono la virtualizzazione dell'intero sistema garantendo sempre gli stessi vantaggi della virtualizzazione:

- Isolamento
- Installazione dinamica
- Ambiente autonomo

## 2.8 Operating system virtualization

Recentemente è stato proposto una differente virtualizzazione leggera: Operating system virtualization or shared kernel approaches. Lo scopo è avere una approccio leggero che minimizza l'overhead delle molteplici VM, con lo stesso SO, in esecuzione sullo stesso sistema.

Con la virtualizzazione del SO l'hypervisor viene rimosso: i server virtuali sono abilitati dal kernel del sistema operativo della macchina fisica. Il kernel del SO è condiviso tra tutti i server virtuali in esecuzione sul kernel. Da quando il kernel è condiviso tra tutte le VM esse condiviscono lo stesso SO, che deve implementare istanze di spazio utente logicamente distinte. Un esempio è FreeBSD o **Linux containers**. Analizziamo ora i pro e i contro di questo tipo di virtualizzazione.

- Vantaggi:
  - La virtualizzazione del SO è più leggera in termini di overhead finché tutti i server virtuali condividono la stessa istanza del kernel.
  - Un singolo sistema con le stesse risorse può supportare più VM rispetto alla full virtualization.
- Svantaggi:
  - Tutte le VM devono condividere lo stesso kernel.
  - Non tutti i sistemi operativi offrono la possibilità della virtualizzazione.

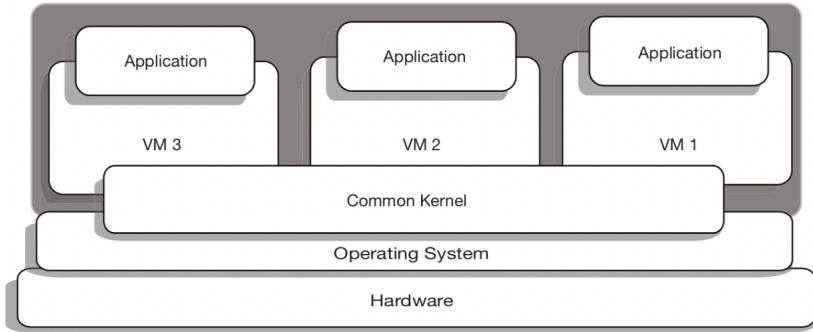


Figure 2.43: Operating system virtualization

## 2.9 Containers

I container sono un modo per isolare un set di processi e far sì che credono che siano i soli in esecuzione sulla macchina.

La macchina che vedono i containers ha un subset di risorse attualmente disponibili sull'hardware fisico (eg, meno memoria, meno spazio su disco, meno cpu, meno banda). I containers non sono macchine virtuali: i processi in esecuzione all'interno di un container sono normali processi in esecuzione sul kernel dell'host. Di conseguenza non c'è un guest kernel in esecuzione all'interno del container. Non si può far girare un SO arbitrario all'interno di un container. Poiché il kernel è condiviso con l'host, deve essere lo stesso dell'host OS.

Il più importante vantaggio rispetto alle VM sono le performance: **non ci sono perdite di performance tra un'applicazione in esecuzione all'interno del container o in esecuzione sull'host.**

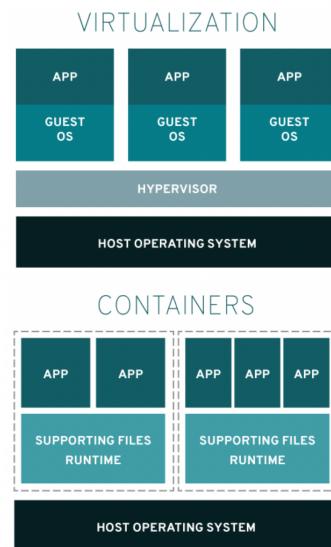


Figure 2.44: Virtualization vs containers

### 2.9.1 Linux Containers

I **containers linux** sono i più famosi e utilizzati per implementare la virtualizzazione del SO. Sono implementati usando due caratteristiche del kernel: **namespace** e **gruppi di controllo** (control groups). Queste caratteristiche assicurano che ogni container ha uno spazio isolato per far girare le proprie applicazioni ed inoltre ha accesso solamente alle risorse fornite (file system, device, altri processes, ...).

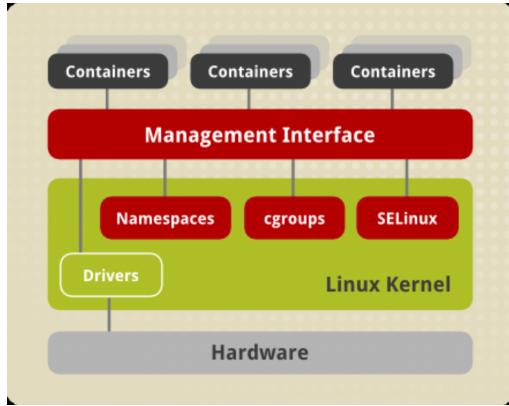


Figure 2.45: Linux Containers

### 2.9.2 Namespaces

I *namespace* forniscono un modo per nascondere le risorse da processi desiderati. Sono un'estensione di una vecchia funzione di Unix chiamata `chroot()`, una chiamata di sistema che permette a un processo di specificare una porzione del file system in cui l'applicazione è confinata. Venne definita per confinare processi malfidati ad una porzione di file system così da fargli accedere solo a file di cui hanno bisogno.

Questa funzione era stata fatta per il filesystem, i namespace sono stati introdotti per nascondere o creare altre copie di altre risorse del sistema. Differenti namespace sono definiti, eg network namespace per nascondere le interfacce network o per creare interfacce virtuali, pid namespace per nascondere processi.

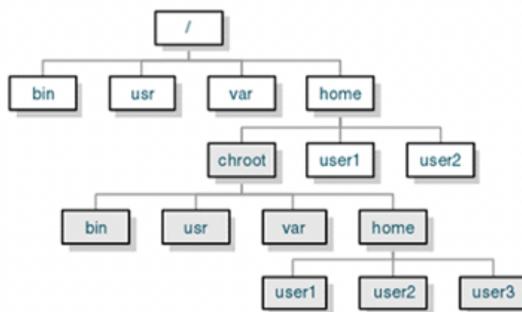


Figure 2.46: Namespace example.

### 2.9.3 Docker

I container oltre a fornire virtualizzazione leggera sono ad oggi utilizzati per la **software distribution**. Un esempio è Docker, un sistema che sfrutta la virtualizzazione del SO per mandare software packages dentro i container. I containers sono perfetti per la distribuzione del software perché forniscono un ambiente isolato nel quale software, library and configuration file possono essere inclusi. Un docker package dopo l'installazione è pronto per essere messo in esecuzione, non richiede l'installazione di librerie aggiuntive o dipendenze.

Docker permette di istanziare containers su un sistema basato su un'immagine scaricata da una repository centrale, il docker registry. L'host esegue un Docker engine tramite il quale si può scaricare, dal docker registry, ed eseguire un container. Un bundle di software può essere uploadato dai dev sul registry, mentre il Docker engine locale fornisce un'interfaccia all'utente per scegliere e installare i software packages.

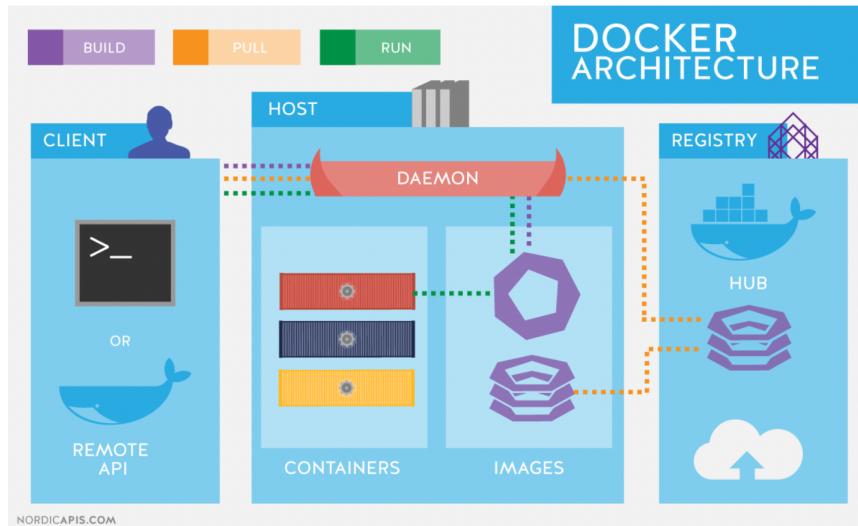


Figure 2.47: Docker architecture

### 2.9.4 Serverless computing

I containers permettono un nuovo tipo di modello di cloud computing: serverless computing o microservizi. L'utente del cloud non crea un'intera VM per fornire i propri servizi (o software) sul cloud, ma crea e fornisce containers con il suo software in esecuzione all'interno di un ambiente con dipendenze e librerie pre installate e pronte all'uso. I cloud providers vendono servizi basati su questi modelli, nuove piattaforme cloud sono create per supportare.

# Chapter 3

## Cloud application structure and architecture

Per quello che abbiamo visto fino ad ora il cloud dal punto di vista dell'infrastruttura ci permette di creare delle VM, ognuna delle quali è dotata di uno o più VCPU, della propria RAM e un local Hard-drive. Su questo hard-drive è installato l'OS ospite, eseguito dalla VM. Ogni VM è dotata anche di una scheda di rete virtuale grazie alla quale può comunicare con le altre VM o tramite una LAN o tramite internet. Le VM possono anche avere l'accesso al cloud storage, uno storage fisico o virtuale condiviso fra tutte le VM, è generalmente accesso tramite la rete LAN.

### 3.1 Traditional application

Le applicazioni sviluppate tramite l'approccio classico hanno un'architettura molto semplice formata da un core server dove gira il servizio che implementa la logica dell'applicazione, le informazioni sono salvate su uno storage fisico. L'altra parte dell'applicazione solitamente è un client leggero che prevede solitamente una GUI. Il client e il server solitamente dialogano tramite una rete locale o tramite internet.

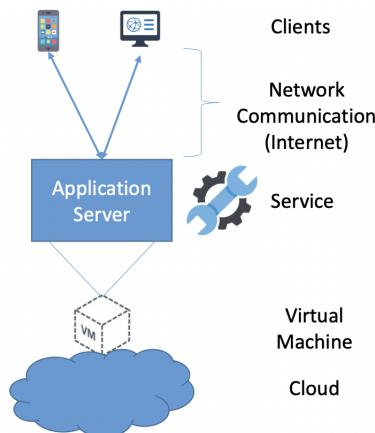


Figure 3.1: Traditional application's architecture.

## 3.2 Cloudification

Tramite il processo di cloudification si spostano le applicazioni, così come sono, e vengono spostate su infrastruttura di tipo cloud. L'architettura dell'applicazione non cambia e neanche il metodo di comunicazione fra client e server, l'unica differenza è che il sistema operativo è installato su una macchina virtuale e non più su un server interamente dedicato al servizio.

Il processo di claudification non crea applicazioni native per il cloud, si limita solamente a spostare quelle già esistenti per permettere l'utilizzo dei benefici di una cloud infrastructure, prevalentemente la riduzione dei costi.

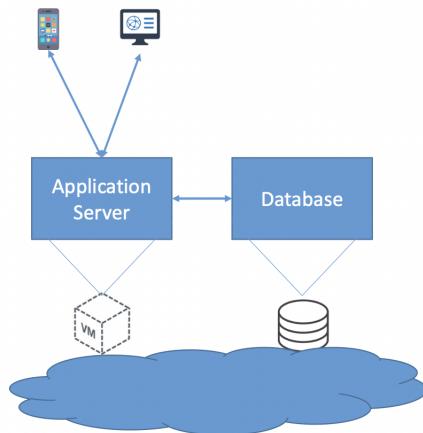


Figure 3.2: Traditional application's migration.

## 3.3 Cloud application

Preservare l'approccio tradizionale non permettere il pieno sfruttamento delle possibilità di una cloud infrastructure, proprio per questo motivo è importante che le applicazioni vengano ridisegnate, adattando il modello in base ai requisiti e ai casi d'uso dell'applicazione stessa. Analizziamo adesso le architetture tipiche di un'applicazione nativa per il cloud.

### 3.3.1 Cloud application architecture

Solitamente le applicazioni sono sviluppate secondo un sistema multi-tier: le differenti funzionalità dell'applicazione vengono suddivise su varie VM, ognuna delle quali si occupa di una funzionalità specifica. Ci sono diversi modi per lanciare un'applicazione, il modo più semplice di tutti è il two-tier architecture, in questo approccio vengono divise le funzionalità che si occupano dei dati da quelle che si occupano dei servizi dell'applicazione. Il risultato sarà un primo tier, l'application server tier, che si occupa di ricevere e gestire le richieste dei client, e un secondo tier, il database tier, nel quale vengono salvati i dati. Le applicazioni interagiscono con il database tier per ricavare i dati necessari e gestire correttamente le richieste.

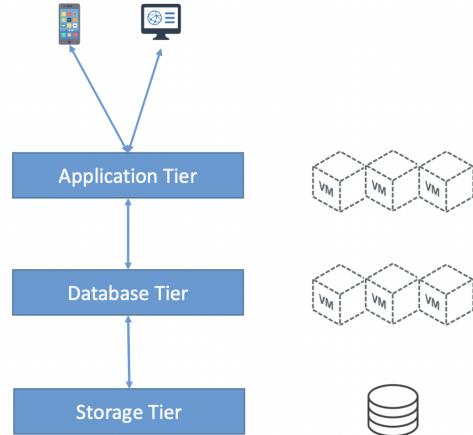


Figure 3.3: Classic cloud application architecture.

### 3.3.2 Multi-tier architecture

Se necessario possono essere utilizzati anche più tier, ad esempio potremmo voler spostare la funzionalità di storage puro dal database tier creando uno storage tier che solitamente implementa un cloud storage (potrebbe non essere implementato tramite una VM). L'application tier riceve le richieste da parte del client, le processa e, se è necessario ottenere qualche dato contatta il database tier che, interagendo con lo storage tier, restituisce i dati e a catena viene gestita la richiesta fino a rispondere al client. La risposta viene fornita al client tramite una chain of interactions che avviene tra i server dei differenti tier. I serve nei core tiers comunicano tramite interazioni di tipo client-server e sono connessi fra di loro tramite una LAN e usano un protocollo applicazione che può essere implementato da un software di middleware in modo da facilitare lo sviluppo.

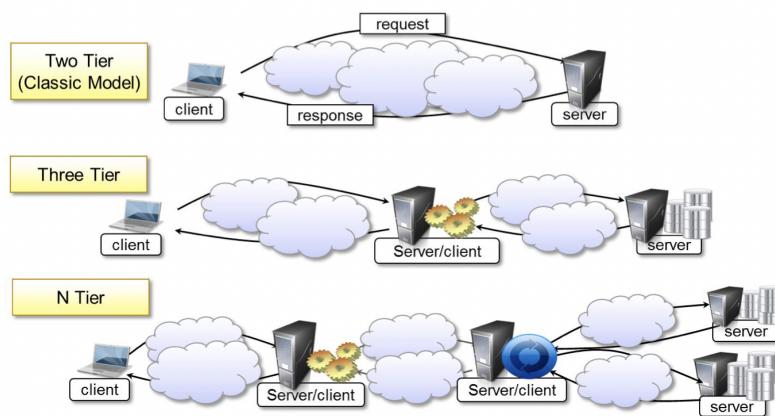


Figure 3.4: Multi-tier cloud application architecture.

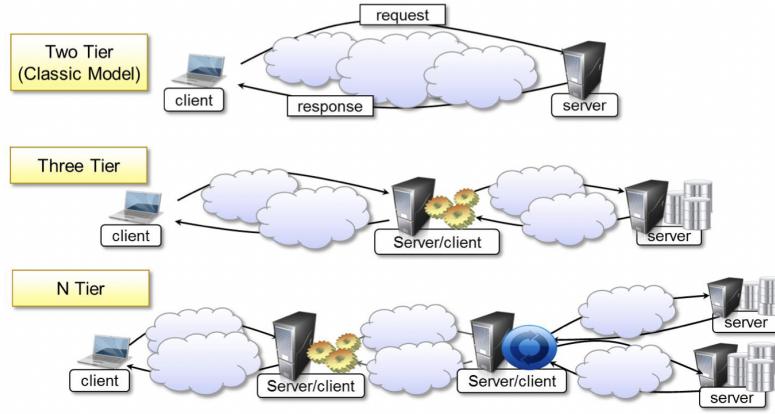


Figure 3.5: Chain of interaction.

## 3.4 Compute Cluster

Generalmente un gruppo di VM appartenenti tutte allo stesso tier sono organizzate in gruppi chiamati Compute Cluster, solitamente le VM sono organizzate per eseguire un calcolo complessivo o su un job singolo e grande oppure su un set di operazioni. I nodi del cluster non devono eseguire troppe operazioni di I/O tranne quelle che servono quando c'è bisogno dello scambio dei dati fra i nodi all'interno del cluster. Viene sfruttato software del middleware per gestire la comunicazione tra le VM, c'è da sottolineare anche che l'architettura del singolo cluster dipende requisiti dell'applicazione e conseguentemente dai casi d'uso.

### 3.4.1 Hight-Availability Clusters

I cluster possono essere di tipo **Hight-Availability**, cioè sono cluster strutturati per essere fault tolerant. Sono molto utili per i servizi che hanno bisogno di essere reperibili nella quasi totalità del tempo. Per ottenere la robustezza, il cluster deve introdurre delle ridondanze, più ridondanze sono presenti e più è robusto il cluster. Per ottenere questo è necessario che, all'interno dello stesso tier, siano presenti più VM capaci di svolgere la stessa operazione o di fornire lo stesso servizio. Analizziamo ora l'architettura di un HA cluster.

#### HA architecture

All'interno di un tier viene ottenuta la ridondanza indicando una VM come master del tier e le altre come slave, le funzioni di un tier vengono comunicate ad un altro tier o al client tramite il master. Le VM slave sono configurate per comportarsi da backup e sono pronte a prendere il posto del master nel momento in cui il master smettesse di funzionare. Per selezionare la macchina slave che può prendere il posto del master si usa una procedura di elezione o un assegnamento statico. Dobbiamo essere sicuri che la macchina slave fornisca il servizio nello stesso modo del master, viene quindi utilizzato un meccanismo di replica dei dati, della configurazione e delle opzioni da affidare alla macchina slave nel momento nel quale diventa il master.

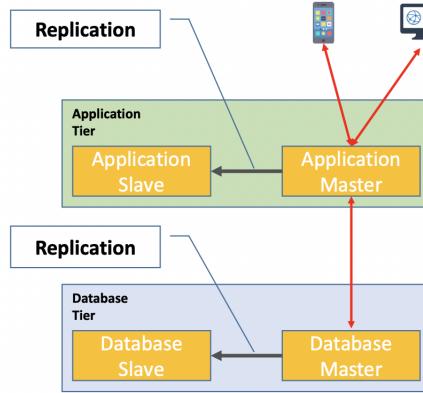


Figure 3.6: Hight-Availability Clusters structure.

### 3.4.2 Load-Balancing clusters

Questo tipo di cluster sono studiati in modo da permettere un utilizzo maggiore di risorse tramite il bilanciamento delle richieste su tutti i nodi partecipanti. Lo scopo è quello di assicurare la scalabilità, per questo tutti i nodi devono condividere il carico di lavoro, infatti le richieste che arrivano dagli utenti sono distribuite tra tutti i nodi del cluster per bilanciarne il carico. Tutto questo porta quindi a un bilanciamento del carico di lavoro che teoricamente porta ad un utilizzo maggiore delle risorse e ad un incremento delle performance. Analizziamo quindi l'architettura utilizzata per questo tipo di cluster.

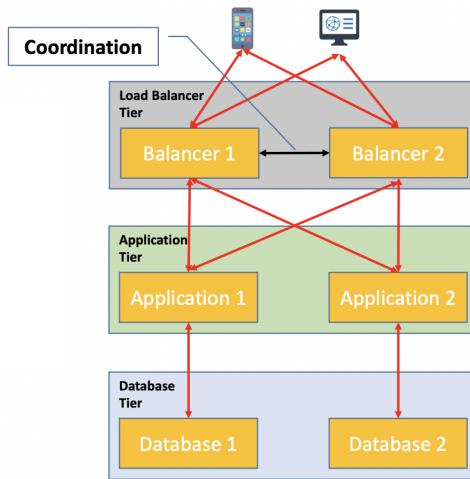


Figure 3.7: Load-Balancing clusters structure.

### LB architecture

Il bilanciamento del carico può essere ottenuto inserendo come primo un ulteriore tier, chiamato appunto load balancer tier. Questo tier è responsabile di ricevere le richieste dai client e di inoltrarle ad una VM appartente all'applicazione tier, i client possono mandare la richiesta a una qualsiasi delle VM presenti nel primo tier. L'istanza che riceve la richiesta la inoltra ad

una VM del tier sottostante, generalmente viene inoltrato alla VM con meno carico. La VM nell'application tier inoltra la richiesta ad una VM presente nel database tier a lui assegnata. Le richieste generali sono quindi bilanciate.

Per utilizzare questo tipo di approccio è però necessario che i dati siano sincronizzati fra tutte le VM presenti in ogni tier. Per ottenere questo ci sono due soluzioni: la prima è quella di utilizzare la replicazioni dei dati all'interno del tier, tuttavia questa prima soluzione porta a un costante overhead dovuto alla replicazione dei dati. La seconda soluzione risolve questo problema, viene aggiunto uno storage tier comune al quale tutti possono accedere in modo che i dati siano tutti nello stesso tier, sarà chiaramente necessario gestire gli accessi ai dati in modo da mantenerli consistenti.

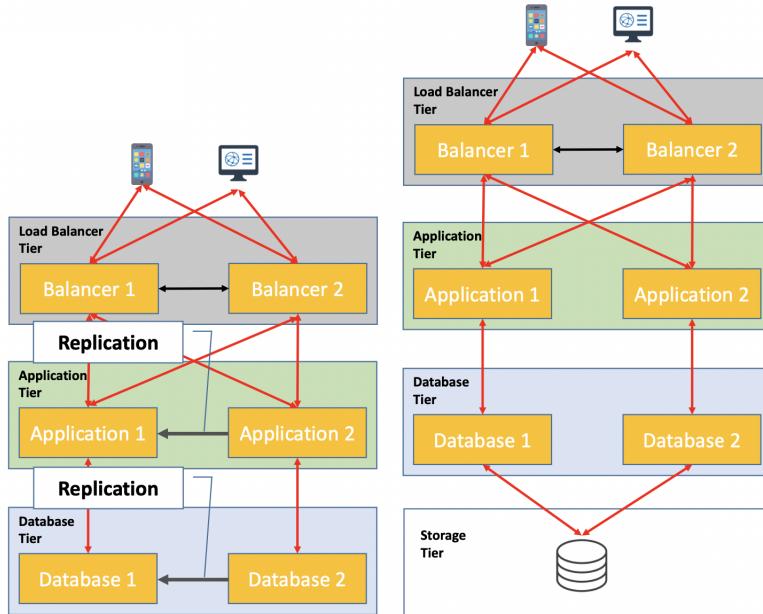


Figure 3.8: Differences with data layer.

### 3.4.3 Compute Intensive Cluster

I *Compute Intensive Cluster* sono cluster studiati per l'analisi di grandi quantità di dati, per questo motivo queste analisi non possono essere gestite su una singola macchina fisica o virtuale perché richiederebbe troppo tempo a prescindere dalla potenza della macchina. Per questo i CI cluster utilizzano il metodo del “dividi et impera”:

- La task di analisi viene suddivisa in una serie di sub-task più semplici da eseguire (jobs)
- I dati vengono divisi in chunks e assegnati ognuno a un job
- Un job e un chunk sono assegnate a un server (workers) per eseguire l'analisi
- Il worker esegue l'analisi e restituisce il suo risultato a un collector
- Il collector unisce tutti i risultati parziali per ottenere il risultato finale

Un esempio di questo approccio sono gli algoritmi di ricerca web.

## CI architecture

Per poter garantire la scalabilità a un gran numero di richieste è necessario il Load Balancer Tier. Un'istanza dell'Application tier suddivide la richiesta iniziale creando i jobs e divide i dati in chunk. I dati sono contenuti in uno Storage tier dal momento che è impossibile salvare i dati nelle VM vista la quantità. I job vengono assegnati ad una schiera di worker ad un'infrastruttura distribuita.

Per limitare l'overhead dovuto alla comunicazione l'infrastruttura di worker può avere accesso allo storage tier, in questo modo worker diversi possono accedere a parti diverse della memoria. Utilizzando questo metodo l'application tier non deve più trasferire i dati interamente ma si limita a fornire ad ogni worker il puntatore alla sezione di dati che deve analizzare.

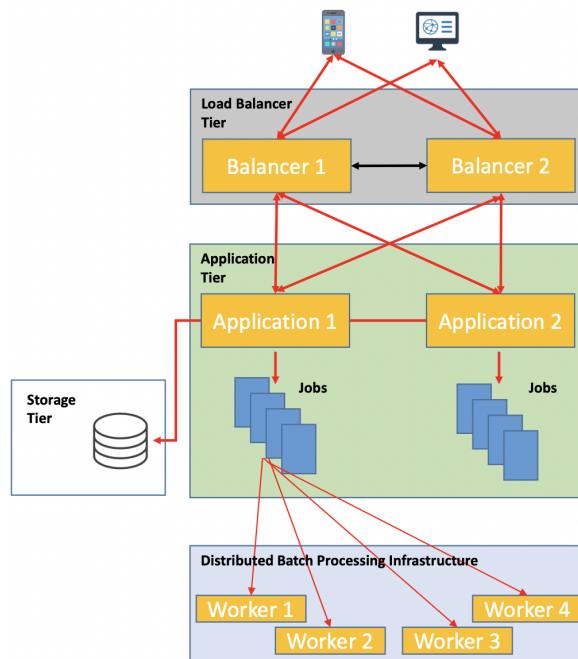


Figure 3.9: Compute Intensive Cluster architecture.

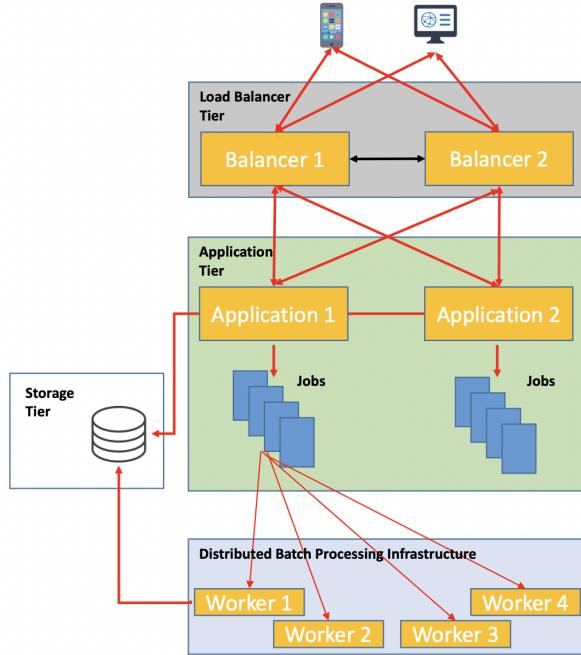


Figure 3.10: Alternative configuration.

### 3.5 Dynamic adaptation

Uno dei vantaggi principali del cloud è la scalabilità, la migliore è quella orizzontale dove basta aggiungere nodi per incrementare la potenza di calcolo. Sia nell'architettura **Compute Intensive** che in quella **Load-Balancing** la scalabilità orizzontale è pronta all'uso grazie allo scaling tier. Quando il traffico aumenta possono essere allocate più VM e inserite nei tier per ridurre il carico di quelle già esistenti, al contrario, quando il traffico diminuisce le VM possono essere facilmente deallocate. Nell'infrastruttura **High Availability** la scalabilità orizzontale può essere sfruttata per aumentare la ridondanza in quanto più ridondanza c'è e più il servizio è robusto, questo tipo di scalabilità può anche essere sfruttato per ribilanciare un tier in caso di rottura di un nodo.

### 3.6 Cloud application design

Le applicazioni cloud sono sistemi distribuiti, fatti da differenti componenti che interagiscono tra loro e eventualmente offrono un'applicazione all'utente.

L'implementazione e integrazione di componenti software differenti per l'ambiente cloud sono task che possono richiedere tempo e denaro e devono seguire una rigorosa metodologia.

#### 3.6.1 Requirements

Un software, parte di un'applicazione cloud, dovrebbe essere sviluppato con i seguenti requisiti:

- **Interoperabilità:** i componenti software devono interagire tra di loro, per garantire l'interoperabilità e l'integrazione essi devono avere un'interfaccia standard con cui altri software/componenti applicazione possono facilmente interagire.
- **Scalabilità:** una delle principali caratteristiche è la scalabilità orizzontale, l'applicazione dovrebbe essere sviluppata per essere pronta ad essere integrata con un numero crescente di componenti, dinamicamente.
- **Composability:** visto che l'ambiente cloud è dinamico e possono avvenire dei cambiamenti rapidi (es fail di un componente, creazione/distruzione VM), il software dovrebbe poter essere usato con componenti diversi senza, o minimi, cambiamenti.

## 3.7 Service-orientate application (SOA)

Emersi recentemente, non è uno strumento o un framework ma un approccio modulare per lo sviluppo di oggetti software. Quest'ultimi si presentano come servizi autonomi che implementano delle specifiche funzionalità. Le applicazioni SOA sono costruite come una raccolta di servizi di componenti software esistenti. Le applicazioni sono altamente modulari invece che un singolo, e grosso, programma. I componenti software possono essere sviluppati usando differenti linguaggi di programmazione e che vengono eseguiti su host eterogenei ed in modo distribuito.

### 3.7.1 SOA architecture

I servizi all'interno delle applicazioni comunicano tramite scambio di messaggi. Ogni messaggio ha uno specifico schema che definisce il formato e lo scambio dell'informazione. Ogni messaggio è gestito internamente dal servizio, in modo nascosto.

Un altro approccio, come per esempio il Remote Method Invocation (RMI), limita la flessibilità e l'interoperabilità perché è basato su dipendenze tecnologiche tra i vari componenti software (appunto l'uso del rmi).

I messaggi che seguono lo schema SOA eliminano le dipendenze; le applicazioni possono essere sviluppate in diversi linguaggi in quanto è necessario solo implementare lo schema dei messaggi. Quest'ultimo non definisce solo il servizio (cosa fa e che informazioni fornisce) ma descrive anche il pattern per lo scambio di messaggi che interagiscono con il servizio.

### 3.7.2 SOA roles

Nell'interazione SOA ci sono due ruoli:

- Consumer service: manda una richiesta di servizio al service provider tramite scambio di messaggi.
- Provider service: ritorna una risposta al consumer contenente il risultato.

Un servizio può sempre essere un consumer, un provider, oppure entrambi, interpretando un ruolo o l'altro a seconda della situazione.

### 3.7.3 SOA broker

In generale i service provider e i consumer inizialmente non interagiscono direttamente tra loro, il consumer dovrà sapere l'indirizzo e il message schema del provider. Solitamente nel mezzo c'è un modulo software chiamato broker. Il broker è responsabile per la creazione di un catalogo dei servizi (o registro) nel quale c'è una lista con i servizi disponibili. Il client service può ottenere un provide service solamente se l'interfaccia è pubblicata dal broker. Dopo che il consumer ha ottenuto i dati può contattare direttamente il provider. L'indirizzo e il descrittore del broker sono conosciuti; quando un consumer ha già contattato il service provider non c'è bisogno che lo rifaccia la volta successiva perché ha già i dati salvati.

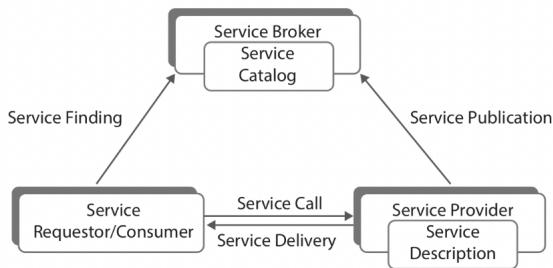


Figure 3.11: How your message is retrieved.

### 3.7.4 SOA advantages

L'implementazione dei servizi rimane nascosta, rendendo così possibile fissare l'implementazione senza intaccare altri servizi fintanto che la dichiarazione di servizio/interfaccia rimane intatta. Questa separazione permette loose coupling (accoppiamento sciolto) che permette di plug e unplug servizi specifici da un'applicazione. Questa caratteristica rende il sistema flessibile ai cambiamenti e all'espansione, ie: nuove caratteristiche possono essere facilmente aggiunte in breve tempo. SOA assicura anche la compasability (componibilità), in quanto un sistema può essere usato come combinazione di servizi esistenti.

I messaggi si basano sul paradigma della comunicazione, viene assicurato che la comunicazione sia fata in modo stateless: un modulo tratta ogni transazione separatamente e indipendentemente dalle altre.

### 3.7.5 SOA scalability

I servizi SOA sono senza stato e disaccoppiati, perfettamente adattabili per la scalabilità dell'infrastruttura cloud. Il broker in particolare può aiutare a gestire il software che vengono creati o distrutti su richiesta e di associarli con i consumatori. Quando una nuova istanza di un servizio è creata (per via dell'incremento di richieste) viene registrato al broker che la rende disponibile per il consumer. Viceversa, quando un'istanza è distrutta viene deregistrata e rimossa in modo trasparente.

## 3.8 Web service

Il web è diventato un metodo per connettere le persone in remoto tramite applicazioni. Il termine Web Service (WS) è spesso riferito ad un'applicazione modulare autonoma, progettata per essere utilizzata e accessibile da altre applicazioni software sul web. I Web Service sono le più comuni istanze dell'implementazione SOA. Il gruppo dietro lo standard WWW, cioè i W3C, definiscono un web service come un sistema software sviluppato per fornire interazioni interoperabili tra due macchine in rete.

### 3.8.1 World Wide Web

Il web fornisce già un meccanismo per:

- Identificare risorse.
- Trasferire dati.
- Un linguaggio di markup per codificare informazioni.

I WS sono la naturale evoluzione e l'adozione del paradigma SOA sul Web.

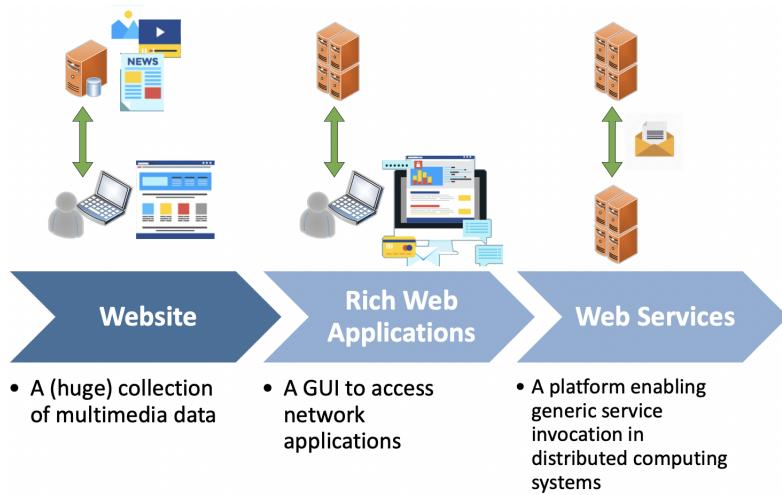


Figure 3.12: World wide web structure.

### 3.8.2 Uniform resource identifier (URI)

Un URI "è una stringa compatta di caratteri che identificano una risorsa astratta o fisica".

#### URI syntax [RFC 1630]

- `<uri> ::= <scheme> ":" <scheme-specific-part>`
- valid schemes are managed by IANA
  - <http://www.iana.org/assignments/uri-schemes.html>

A URI can be

- A name: Uniform Resource Name (**URN**)
  - `<urn> ::= "urn:" <nid> ":" <nss>`
  - `urn:isbn:0-395-36341-1`
- An address: Uniform Resource Locator (**URL**)
  - `"http://" "/" <host> [":" <port>] [<abs_path> ["?" <query>]]`
  - <http://www.unipi.it/>

Figure 3.13: URI example.

### 3.8.3 HTTP

HTTP è un protocollo di livello applicazione per sistemi distribuiti, questo protocollo serve per scambiarsi dei messaggi come illustrato nella seguente immagine.

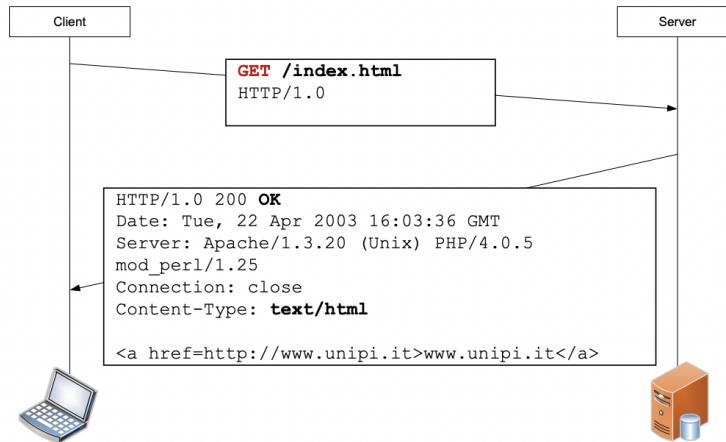


Figure 3.14: HTTP protocol.

### 3.8.4 Data encoding

Il Web ha adottato HTML, la perfetta soluzione per dati visuali, non è ottima però per dati codificati di applicazioni. Per questa ragione nel 2000 è stato introdotto eXtensible Markup Language (XML). XML è un linguaggio di markup di codifica basato su teso che definisce un insieme di regole per la codifica di qualsiasi tipo di documento in un formato che sia leggibile da persona e leggibile da macchina.

L'XML specifica la sintassi, cioè come le informazioni devono essere scritte all'interno del documento. Specifiche grammaticali, ad es. che tipo di informazioni possono essere scritte su documento, possono essere disponibili ma non sono obbligatorie. Il linguaggio è estensibile in quanto il set di tag non è pre-specificato e può essere facilmente archiviato o trasmesso via internet.

### 3.8.5 XML validation

La sintassi XML definisce se un documento è in un formato corretto o meno. Con lo scopo di verificare se un documento è valido o no, è necessario che contenga un riferimento ad un Document Type Definition (DTD) o ad uno schema XML (la versione più recente).

DTD/XML Schema è uno schema o una grammatica del documento, ad es. specifica che tipo di informazioni possono essere incluse nel documento in relazione ad elementi e attributi. Un documento XML conforme alle regole di un documento DTD/XML è valido, altrimenti non lo è.

### 3.8.6 Web-Service interaction

Un WS ha un'interfaccia descritta in un formato eseguibile dalla macchina, il Web Service Description Language (WSDL). Il WSDL è utilizzato per interagire con il broker, un modulo che fornisce un registro globale per la pubblicità e la scoperta di servizi web. Tramite il WSDL ogni fornitore di servizi si registra al broker, mentre il consumatore di servizi utilizza il WSDL per cercare un servizio specifico. Il consumatore e il fornitore di servizi interagiscono utilizzando il Simple Object Access Protocol (SOAP) che fornisce un formato di messaggistica standard.

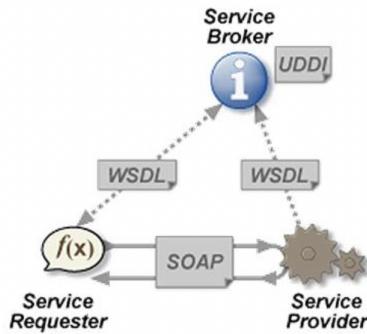


Figure 3.15: Interaction structure.

## 3.9 SOAP

Fornisce una struttura standard per la trasmissione di documenti XML su vari protocolli interni. Un SOAP Message è costituito da un elemento root chiamato envelope che contiene un header e un body. L'header contiene le informazioni aggiuntive e di sicurezza. Il body include il payload, che include a sua volta informazioni di richiesta/risposta, e la sezione Fault che include errori ed eccezioni, se presenti. Di seguito sono illustrati i processi di scambio di messaggi, più precisamente la richiesta e la risposta.

### SOAP MESSAGE EXAMPLE REQUEST

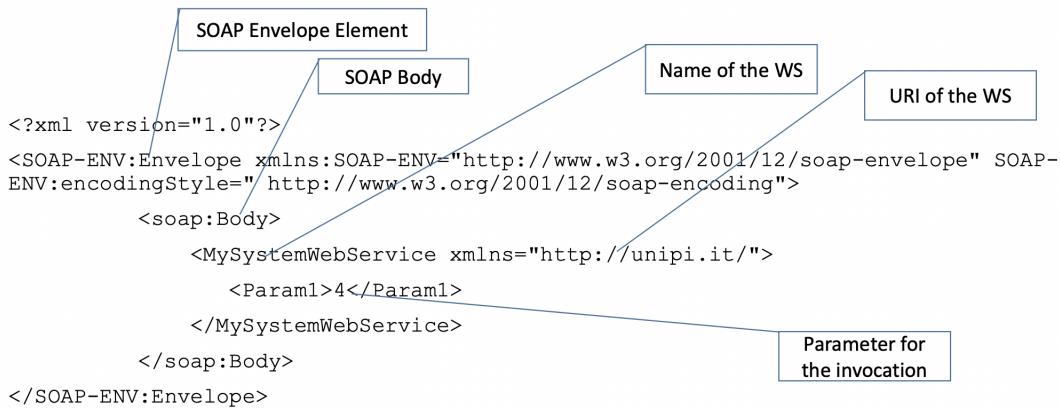


Figure 3.16

### SOAP MESSAGE EXAMPLE RESPONSE

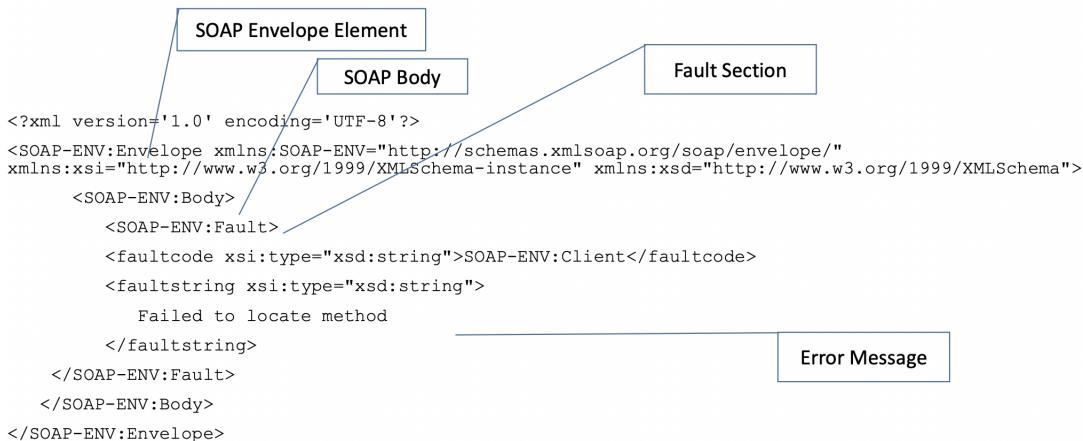


Figure 3.17

### 3.9.1 SOAP binding

Solitamente SOAP è associato ad HTTP, significa che i messaggi SOAP sono trasmessi sopra HTTP. SOAP può usare GET o POST. Con GET la risposta è un messaggio SOAP, la richiesta è opzionale, con POST sia richiesta che risposta sono SOAP.

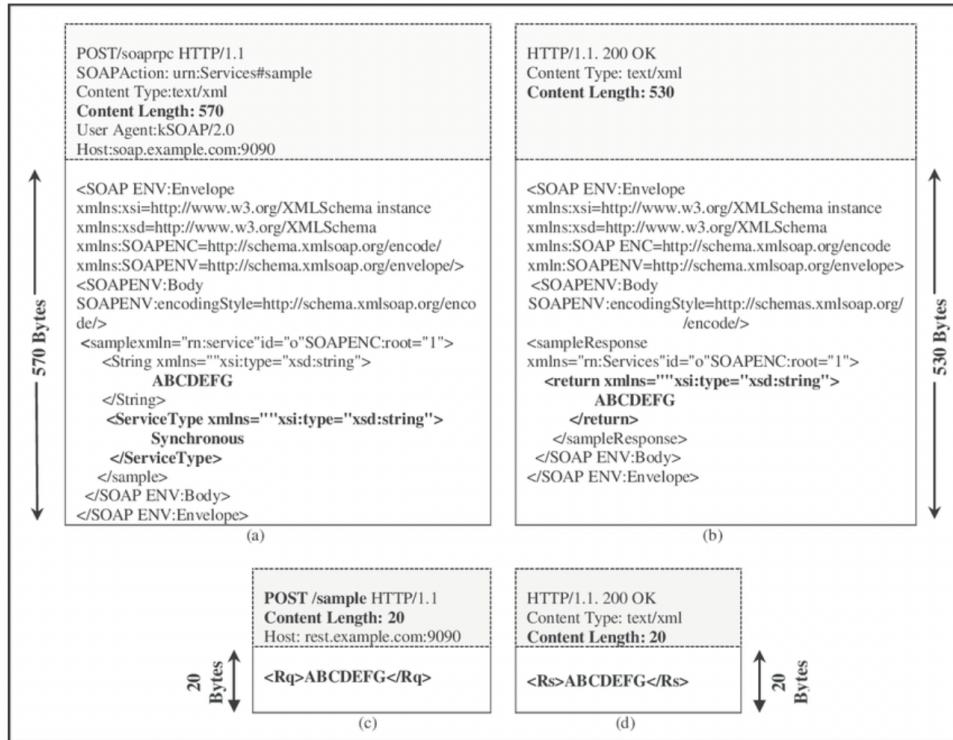


Figure 3.18: How message is transmitted

## 3.10 WSDL

Describe l'interfaccia e l'insieme delle operazioni supportate da un WS in un formato standard. Standardizza inoltre l'insieme di parametri di entrata e di uscita delle operazioni di servizio ma anche il protocollo di servizio, es. il senso in cui i messaggi saranno trasferiti a/dal servizio. Tramite WSDL, i clienti possono automaticamente capire:

- Cosa il servizio può fare.
- Dove si trova il servizio.
- Come può essere invocato.

Ogni volta che un cliente vuole invocare un WS sconosciuto, richiede e analizza il documento WSDL, solo successivamente invoca il servizio tramite SOAP.

### 3.10.1 WSDL message architecture

Un messaggio WSDL è un documento XML ed è composto da due sezioni:

- Il Service Interface Definition, che definisce in modo astratto il servizio. Comprende: il tipo di dati (types), i messaggi scambiati tra il richiedente e il provider (messaggi) e le operazioni che possono essere fatte (portType).
- Il Service Implementation Definition, che descrive in modo specifico come il servizio viene invocato. Comprende: come trasferire i messaggi (binding) e come invocare il servizio (service).

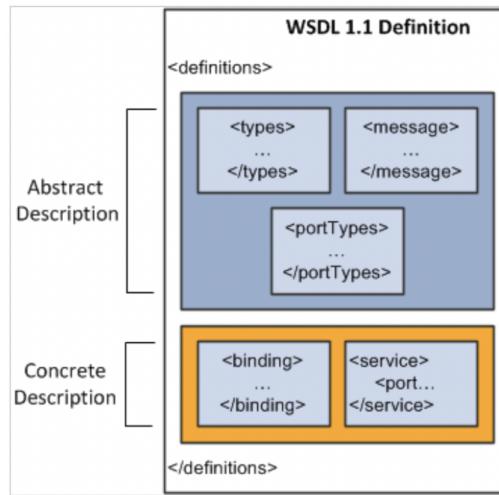


Figure 3.19: WSDL message architecture

### 3.10.2 WSDL type

Il tipo è utilizzato per definire tipi di dati complessi. Questi tipi sono quindi usati nei messaggi.

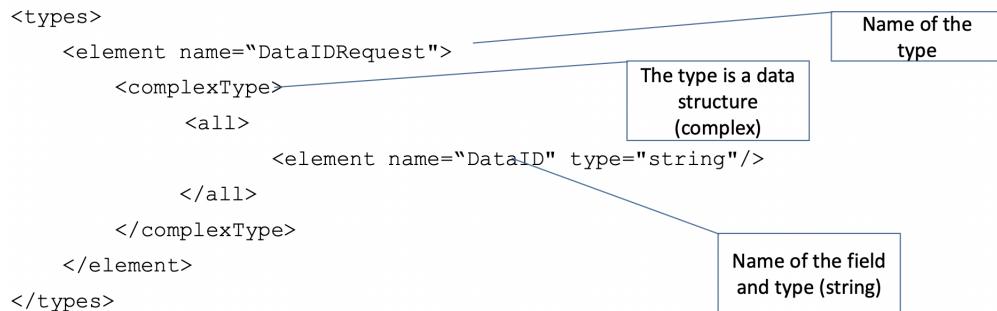


Figure 3.20: WSDL type

### 3.10.3 WSDL messages

I messaggi vengono scambiati tra client e WS, possono definire sia un messaggio di input che output.

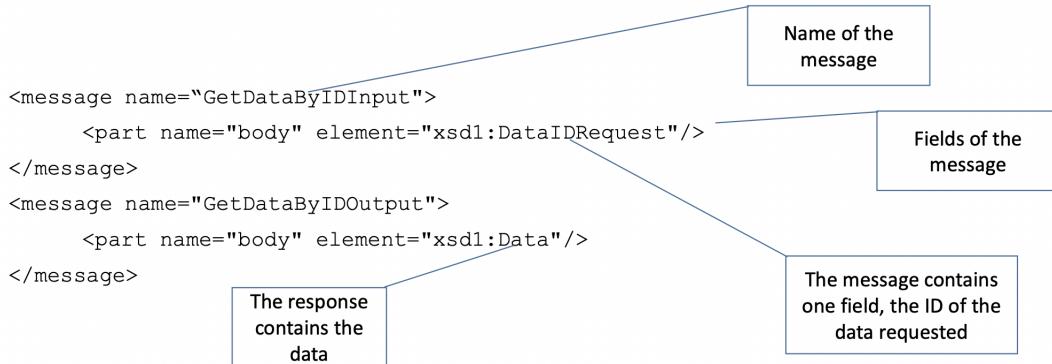


Figure 3.21: WSDL messages

### 3.10.4 WSDL porttype

PortType viene usato per legare messaggi di input e output in un'operazione logica.

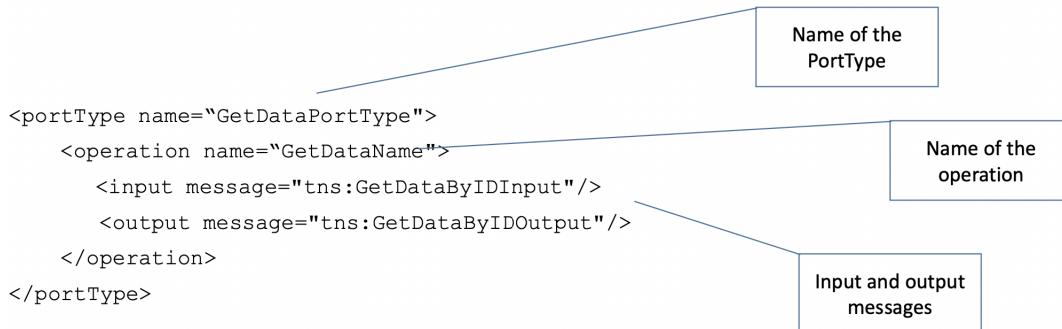


Figure 3.22: WSDL PortType

### 3.10.5 WSDL binding

Il binding (legame) viene utilizzato per specificare come vengono scambiati i messaggi associati ad un'operazione porttype.

```
<binding name="GetDataBinding" type="tns:GetDataPortType">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="GetDataName">
            <soap:operation soapAction="http://unipi.it/getdataname"/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
    </binding>
```

The diagram shows the WSDL binding code with two callout boxes. One points to the 'name' attribute of the binding element, labeled 'The name of the PortType'. Another points to the 'soapAction' attribute of the operation element, labeled 'The SOAP Action parameters'.

Figure 3.23: WSDL Binding

### 3.10.6 WSDL service

I servizi vengono usati per specificare come l'operazione è praticamente invocata, es. l'URI associato con il servizio.

```
<service name="GetDataService">
    <documentation>GetDataService</documentation>
    <port name="GetData" binding="tns:GetDataBinding">
        <soap:address location="http://unipi.it/getdata"/>
    </port>
</service>
```

The diagram shows the WSDL service code with two callout boxes. One points to the 'binding' attribute of the port element, labeled 'The binding associated'. Another points to the 'location' attribute of the soap:address element, labeled 'The URI for the SOAP invocation'.

Figure 3.24: WSDL Service

## 3.11 UDDI

La specifica Universal Description, Discovery, e Integration (UDDI) definisce un modo per pubblicare e scoprire informazioni sui WS. E' un protocollo basato su XML indipendente dalla piattaforma che permette di creare un registro di sistemi per archiviare informazioni sulle WS. Utilizzando UDDI un business può registrare un WS per un broker, mentre i clienti possono cercare un WS per cercare informazioni WSDL per la sua invocazione.

## 3.12 REST

SOAP è spesso considerato un protocollo complesso, in quanto richiede la creazione della struttura XML, che è obbligatoria per il passaggio di messaggi.

La Representational State Transfer (REST) è stata introdotta come una valida alternativa leggera per la costruzione di WS. Fornisce un modello per la progettazione di sistemi software basati sulla rete utilizzando il modello client/server basato sul protocollo di trasferimento HTTP.

In un sistema RESTful, un client invia una richiesta tramite HTTP utilizzando i metodi HTTP standard (GET, PUT, POST, DELETE), e il server rilascia una risposta che include la rappresentazione della risorsa.

Affidandosi al supporto minimo offerto dall'HTTP, è possibile fornire tutto il necessario per sostituire le funzionalità base e più importanti fornite da SOAP: il metodo di invocazione. Il contenuto dei dati è ancora trasmesso usando XML come parte del contenuto HTTP, ma il markup aggiuntivo richiesto da SOAP viene rimosso.

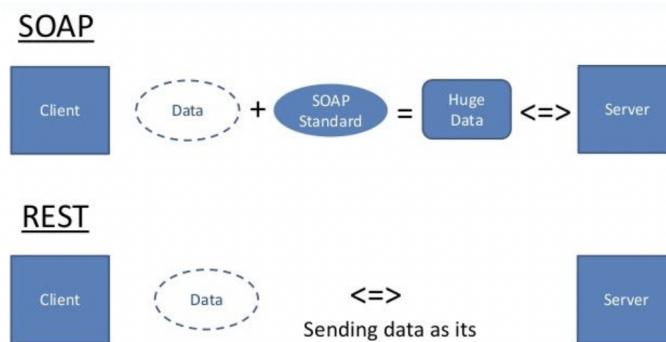


Figure 3.25: SOAP and REST comparison

I metodi GET, PUT, POST, and DELETE costituiscono l'insieme minimo di operazioni per ricevere, aggiungere, modificare e eliminare dati. I parametri da includere nella richiesta, o dati da inviare al servizio, potrebbero essere inclusi come valori di interrogazione nella richiesta (parametri) o come carico utile (dati da caricare). Insieme ad un'organizzazione URI per identificare le risorse, tutte le operazioni atomiche richieste da un WS possono essere implementate. Per questo motivo, REST rappresenta una valida e leggera alternativa al SOAP, che funziona efficacemente in contesti in cui sono assenti aspetti aggiuntivi oltre a quelli gestibili tramite HTTP. L'identificativo globale assegnato a ciascuna risorsa rende uniforme il metodo di accesso delle risorse. Qualsiasi risorsa a un indirizzo conosciuto può essere accessibile da qualsiasi agente applicativo. REST permette molti formati standard come XML, JavaScript Object Notation (JSON) o testo normale, nonché qualsiasi altro formato concordato per lo scambio di dati.



Figure 3.26: Exempre of REST request

## 3.13 JSON

JavaScript Object Notation è un formato standard aperto che usa testo human-readable per trasmettere oggetti di dati. E' stato originariamente definito per le applicazioni web e per lo scambio di dati asincrono (AJAX). E' basato su un sottoinsieme di javascript ed è un formato di testo e lingua indipendente. E' stato progettato per codificare i dati in modo leggero, si basa solo su due strutture: collezione di coppie nome/valore e una lista ordinata di valori.

### 3.13.1 JSON vs XML

JSON	XML
<pre>{   "menu": {     "id": "file",     "value": "File",     "popup": {       "menuitem": [         {"value": "New", "onclick": "CreateNewDoc()"},         {"value": "Open", "onclick": "OpenDoc()"},         {"value": "Close", "onclick": "CloseDoc()"}       ]     }   } }</pre>	<pre> &lt;menu id="file" value="File"&gt;   &lt;popup&gt;     &lt;menuitem value="New" onclick="CreateNewDoc()" /&gt;     &lt;menuitem value="Open" onclick="OpenDoc()" /&gt;     &lt;menuitem value="Close" onclick="CloseDoc()" /&gt;   &lt;/popup&gt; &lt;/menu&gt;</pre>

Figure 3.27

### 3.13.2 JSON in WS

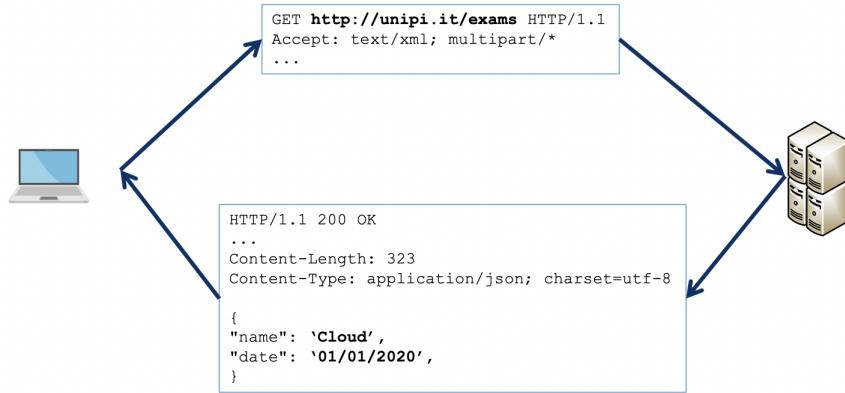


Figure 3.28

## 3.14 CRUD

L'acronimo CRUD si riferisce alle più importanti funzioni che sono implementate in una applicazione con database relazionale: Create, Read, Update e Delete. Ogni lettera dell'acronimo può essere mappata in un'istruzione SQL, tramite l'HTTP. (Con HTTP possiamo fare richieste al db). Questo è tipicamente utilizzato per costruire API RESTful. I principi CRUD sono mappati ai comandi REST per rispettare gli obiettivi dell'architettura RESTful.

Route name	URL	HTTP Verb	Description
Index	/blogs	GET	Display a list of all blogs
New	/blog/new	GET	Show form to make new blogs
Create	/blogs	POST	Add new blog to database, then redirect
Show	/blogs/:id	GET	Show info about one blogs
Edit	/blogs/:id/edit	GET	Show edit form of one blog
Update	/blogs/:id	PUT	Update a particular blog, then redirect
Destroy	/blogs/:id	DELETE	Delete a particular blog, then redirect

Figure 3.29: CRUD Examples

## 3.15 Request-Reply communication

Sia SOAP che REST utilizzano il protocollo Request-Reply per la comunicazione. Questo protocollo implementa una comunicazione sincrona, ossia il client si blocca fino a quando non riceve una richiesta dal server, per questo il client e il server si dicono **time-coupled**. Il protocollo è anche affidabile in quanto la risposta fa da acknowledgement di corretta ricezione della richiesta. Il protocollo utilizza anche una comunicazione diretta. Il client interagisce direttamente con il server senza intermediari (time-coupling), questo porta però all'introduzione di overhead e complessità addizionali. Il paradigma request-response è adatto per la comunicazione fra i client e il frontend dell'applicazione cloud perché il paradigma è semplice e affidabile e l'implementazione del client permette di tollerare space e time coupling. Per quanto riguarda la comunicazione fra frontend e backend è differente in quanto nel backend ci aspettiamo di avere un numero consistente di VM per garantire la scalabilità tramite il load-balancing e la disponibilità del servizio tramite la data replication. Dato che supportare la dinamicità è un problema primario, time e space coupling non sono adatti a garantire la scalabilità nella comunicazione sia all'interno del backend sia fra il backend e il frontend.

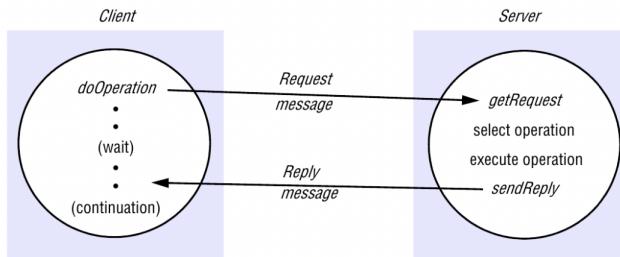


Figure 3.30: Communication Example

### 3.15.1 Indirect communication

Viene utilizzato un paradigma di comunicazione indiretta che garantisce:

- **Space uncoupling**: il mittente non ha bisogno di sapere e non conosce l'identità del destinatario e vice versa, i mittenti o i destinatari possono essere rimpiazzati, migrati o replicati al tempo si esecuzione. Questo viene raggiunto aggiungendo un intermediario senza direct coupling fra mittente e destinatario.
- **Time uncoupling**: il mittente e il destinatario hanno vita differente. In un ambiente volatile mittente e destinatario non hanno bisogno di vivere nello stesso momento per comunicare. Questo è assicurato dall'intermediario che adotta un paradigma di comunicazione asincrona: il mittente manda il messaggio all'intermediario senza però bloccarsi, l'Intermediario si occuperà di consegnare il messaggio al destinatario.

## 3.16 Message oriented communication

Molti paradigmi di comunicazione basati su messaggi indiretti sono stati definiti per le applicazioni cloud, tuttavia tutti si basano sul fatto che sender e receiver non aprono un canale di

comunicazione fra di loro, come accade ad esempio in UDP/TCP, ma inseriscono il messaggio in un Bus. Il bus è implementato come un intermediario che riceve i messaggi e li inoltra ai destinatari, andremo a descrivere due tipologie principali di intermediari **broker** e **message queue**.

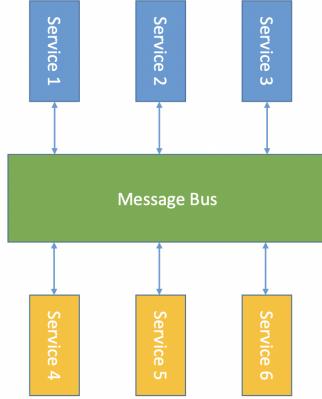


Figure 3.31: Message oriented communication

### 3.16.1 Message broker

Un sistema Publish-Subscribe nel quale i mittenti (chiamati produttori o publisher) pubblicano dati strutturati, tipicamente un evento, mentre i destinatari (chiamati consumer o subscriber) esprimono una preferenza verso uno specifico tipo di dato attraverso un’iscrizione. Al centro del sistema c’è il broker che è responsabile di trasferire i dati dai sender ai receiver quando ad esempio viene trovata corrispondenza fra un’iscrizione e un tipo di dato pubblicato. Tutto questo risulta in una comunicazione una-a-molti.

È importante sottolineare l’esistenza di diversi tipi di iscrizione, vediamone tre:

- **Channel-based:** i publisher pubblicano i dati in un determinato canale e i destinatari si iscrivono ad un canale per ricevere i dati
- **Topic-based:** ogni dato ha un numero di campi, uno di questi indica il topic. I destinatari si iscriveranno quindi ad un determinato topic di interesse. Il metodo è identico a quello basato sui canali con la differenza che in questo il topic viene espresso direttamente in quanto campo dei dati, mentre nei canali è espresso indirettamente.
- **Type-based:** le iscrizioni vengono definite in termini di tipo di eventi e le corrispondenze vengono definite in termini di tipo o sottotipo del filtro usato.

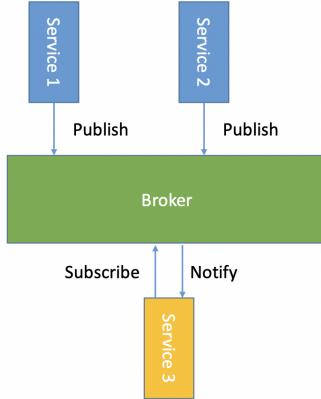


Figure 3.32: Message Broker Structure

### 3.16.2 Message queue

Questo approccio fornisce una comunicazione point-to-point per lo scambio dei dati fra produttore e consumatore, più in dettagli, il produttore inserisce i dati in una coda, i dati verranno rimossi dalla cosa da parte del destinatario. Le code vengono gestite dal Message Queuing, i produttori possono scegliere una determinata coda dove inserire il messaggio e anche i destinatari possono scegliere una cosa dalla quale prendere un messaggio. Il set di operazioni offerte dal Message Queuing System è semplice:

- Send: usato dal produttore per inserire il messaggio in una coda.
- Blocking Receive: usato dal consumatore per bloccarsi fino all'arrivo di un determinato messaggio.
- Non-blocking receiver (polling operation): usato dal consumatore per controllare lo stato di una cosa, ritorna il messaggio se presente o un'indicazione di “non available” altrimenti.
- Notify: usato dal Message Queuing System per notificare un consumer quando il messaggio è disponibile nella coda associata.

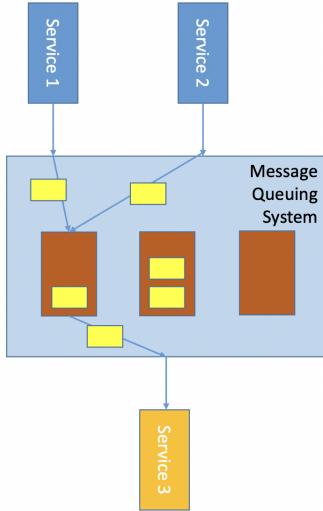


Figure 3.33: Message Queue Structure

Analizziamo ora i tipi di messaggi. Ogni messaggio consiste di:

- Una destinazione, univoca che identifica una coda
- Metadata associati con il messaggio: includono campi come la priorità del messaggio o come esso deve essere mandato
- Corpo del messaggio

Il corpo del messaggio è generalmente opaco e non toccato dal Message Queuing System, la selezione dei messaggi viene invece fatta attraverso i metadata. Il metodo di coda è generalmente FIFO, alcune code però possono supportare il fattore di priorità, andando quindi a consegnare per primi i messaggi ad alta priorità. I consumatori possono prendere un messaggio in base alle sue proprietà. È importante dire che il sistema di code è persistente, i messaggi rimangono salvati nelle cose per un tempo indefinito fino a che non vengono prelevati da un consumatore, i messaggi salvati nelle cose vengono anche salvati sul disco per aumentare la sicurezza del sistema.

I messaggi in arrivo possono anche subire un processo di trasformazione, questo serve per far sì che i messaggi possano essere letti dalla maggior parte dei receiver. La trasformazione può anche banalmente essere un cambio di endianess passando da big a little o vice versa.

## Scalability

Le code dei messaggi sono ottime per la comunicazione nel backend dell'applicazione in quanto può mantenere la dinamicità che caratterizza l'ambiente. Le code possono gestire un numero di VM ampio, ad esempio le VM che implementano la stessa funzionalità all'interno di un cluster possono ricevere i dati dalla stessa coda. Dal momento che il destinatario non deve necessariamente esistere al momento della creazione, si può gestire una situazione in cui le VM vengono allocate e deallocate in maniera dinamica. Per questo motivo i produttori di dati non devono aspettare la risposta di avvenuta consegna come avvenuta nel paradigma request-response.

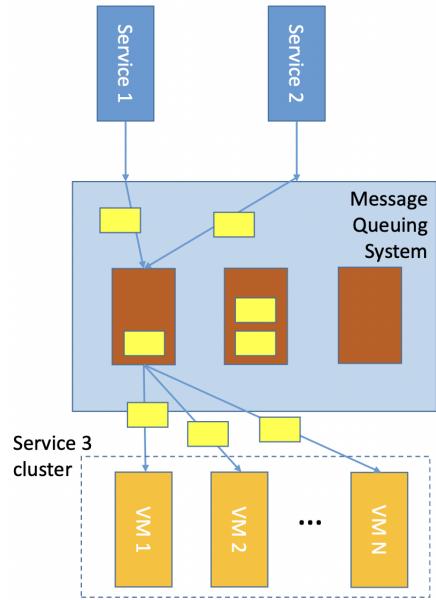


Figure 3.34: Message Queue Scalability

### 3.16.3 Implementations

Per quanto riguarda l'implementazione dei Message Broker o dei Message Queuing Systems può essere centralizzata o distribuita. Analizziamo i due scenari:

- In un'implementazione **centralizzata** il broker o il message queuing system sono implementati su una singola macchina che potrebbe essere una VM: questa implementazione è la più facile anche se pecca di scalabilità e di resilienza in quanto è presente un single point of failure e un possibile bottleneck che impatta sulle performance generali.
- In un'implementazione **distribuita** il broker o il message queuing system è implementato su una sistema di macchine: è più complessa dal momento che sono necessari la replicazione e il coordinamento dei dati, questa implementazione garantisce però la resilienza e la scalabilità.

L' Advanced Message Queuing Protocol (AMQP) è uno dei protocolli maggiormente usati per l'implementazione del Message Queuing System.

## 3.17 Data replication

Nelle applicazioni cloud le macchine all'interno dello stesso tier svolgono le stesse funzioni in modo da assicurare un incremento di performance (scalabilità), availability e tolleranza ai guasti. Per assicurarsi che ogni VM operi sugli stessi dati si sfrutta la replicazione dei dati, generalmente viene implementata tramite lo scambio di messaggi fra le VM dello stesso tier. Tuttavia, quando i dati da scambiarsi sono troppi, si preferisce utilizzare un livello di storage condiviso tramite il quale ci assicuriamo che tutte le VM lavorino sullo stesso dataset, anche in questo caso gli scambi di messaggi fra le VM sono ridotto solo ai messaggi necessari al coordinamento.

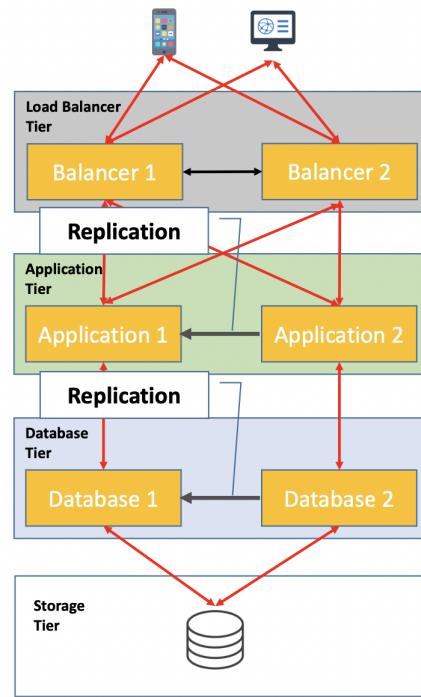


Figure 3.35: Data Replication with single data layer

### 3.17.1 Requirements

La replicazione dei dati fra le VM devono rispettare i seguenti requisiti:

- *Replication transparency*: la replicazione dei dati deve essere trasparente all'utente, nel senso che un utente può accedere a varie copie dei dati senza esserne a conoscenza.
- *Consistency*: i dati devono essere consistenti fra tutte le VM.
- *Resiliency*: il sistema dovrebbe essere resistente alla rottura di una VM, anche se le interruzioni di corrente nella comunicazioni non possono avvenire. Questa assunzione può essere presa per vera se tutte le VM sono connesse fra di loro tramite una rete LAN.

### 3.18 System Model

I dati nel sistema sono una collezione di oggetti, ad esempio un file, una classe o un set di record. Ogni oggetto logico è implementato come una collezione di copie fisiche, chiamate repliche. Ogni replica è salvata in un differente replica manager, un componente a parte che salva istanze di una replica (una VM che compone il cluster di livello, nel nostro caso). Per assicurare la consistenza dei dati le repliche sono State machine, questo significa che oltre alla replica dei dati anche dello stato addizionale viene replicato. Ogni manager delle repliche accetta degli aggiornamenti dei dati in maniera riparabile nel caso in cui avvenissero dei malfunzionamenti.

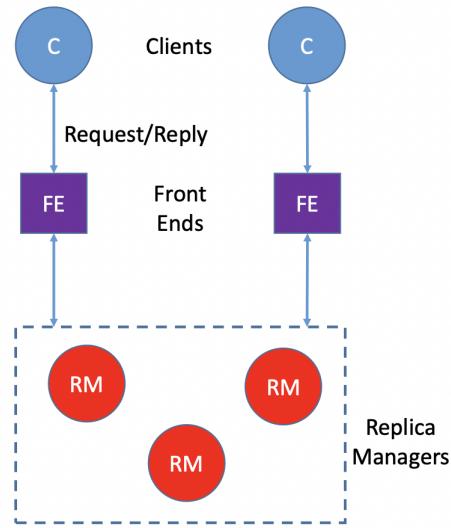


Figure 3.36: System Model example

#### Request Handling

Una richiesta può essere una semplice query per ricevere dei dati o può essere un aggiornamento dei dati per che modifica lo stato dell'oggetto. In generale le seguenti fasi vengono implementate per la gestione di una richiesta:

- *Request*: la richiesta è ricevuta dal modulo frontend che lo inoltra ad uno o più replica manager. A seconda dell'implementazione il frontend comunica con un singolo replica manager che poi si occuperà di comunicare con gli altri oppure può comunicare direttamente con tutti i replica manager.
- *Coordination*: i replica manager si coordinano per preparare l'esecuzione della richiesta. Si possono coordinare per decidere se eseguire la richiesta o se deve essere rimandata per mantenere la consistenza.
- *Execution*: i replica managers eseguono la richiesta. La richiesta è portata a termine per tentativi in modo che la modifica possa essere annullata successivamente in caso di necessità.

- *Agreement*: i replica manager cercano il consenso sugli effetti della richiesta se necessario, se viene raggiunto l'accordo l'esecuzione della richiesta è sottomessa e gli eventuali cambiamenti vengono applicati.
- *Response*: Uno o più replica manager rispondono al frontend. In alcuni sistemi solo un replica manager risponde al frontend, in altri casi un insieme di replica manager rispondono insieme.

### 3.18.1 Group/Multicast Communications

Sono disponibili diversi tipi di multicast con proprietà differenti:

- *Reliable/atomic multicast*: il messaggio viene consegnato o a tutti i membri che ne hanno diritto o a nessun membro. Vengono gestite le perdite di comunicazione e vengono eliminati i duplicati, non è garantito l'ordine di arrivo dei messaggi.
- *Total/casual order multicast*: ogni membro deve ricevere tutti gli update nello stesso ordine. Ad esempio se A e B sono due azioni e A viene prima di B allora ogni membro deve ricevere prima A e solo dopo B. Tuttavia non è garantito l'ordine dei membri che ricevono gli update, nel senso che non è garantito che prima tutti ricevano A e solo dopo tutti ricevano B.
- *View-synchronous communication*: tutti i membri hanno la stessa situazione per tutto il tempo. Cioè prima ricevono tutti A e poi ricevono tutti B.

Partendo dalla prima e arrivando alla terza *si hanno più garanzie ma aumenta la complessità*.

### 3.18.2 Primary Backup Replication

Questo tipo di modello è utilizzato per garantire la tolleranza ai guasti. In ogni momento solo un replica manager è designato come primario mentre gli altri sono designati come slave. Il frontend comunica solo con il replica manager primario, il quale si occuperà di gestire la richiesta e mandare una copia dei dati aggiornati alle macchine di backup. Nel caso in cui la macchina primaria sia inutilizzabile, una delle macchine di backup viene scelta come nuova macchina principale. Vediamo ora come viene gestita una richiesta ricevuta dalla macchina primaria.

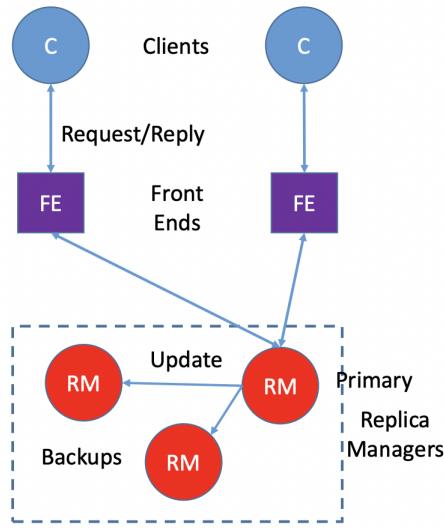


Figure 3.37: Primary Backup Replication

#### Request handling

- *Request*: il frontend emette una richiesta che contiene un ID univoco che serve da identificativo della richiesta stessa.
- *Coordination*: la macchina primaria gestisce le richieste automaticamente nell'ordine secondo il quale vengono ricevute. Viene controllato l'ID della richiesta, se è già stato eseguito viene mandata la medesima risposta. Non è necessaria nessuna coordinazione fra le repliche.
- *Execution*: la macchina primaria esegue la richiesta e salva la risposta.
- *Agreement*: se la richiesta è un update, la macchina primaria manda lo stato aggiornato, la risposta e l'ID unico a tutte le altre macchine di backup, le quali risponderanno con un acknowledgment. Per questo tipo di comunicazione è sufficiente L'atomic multicast.
- *Response*: il primario risponde al frontend che si occupa di inoltrare la risposta al client.

### 3.18.3 Active replication

Questo approccio differente garantisce alta disponibilità. Le replica manager sono organizzate in gruppi, il frontend inoltra le sue richieste ad ogni replica manager del gruppo, le quali possono processare la richiesta e rispondere in maniera identica. Se una replica manager si rompe, non c'è impatto sulle performance perché risponderanno le altre.

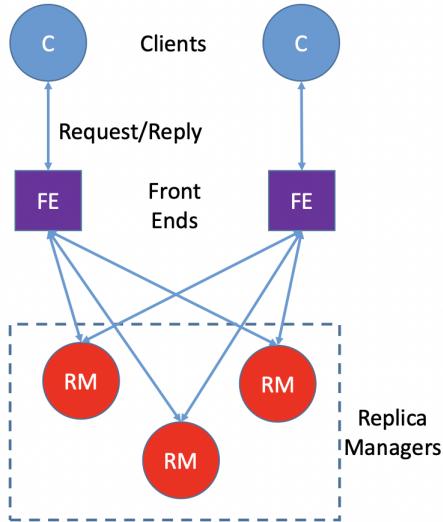


Figure 3.38: Active replication

#### Request handling

Vediamo come viene gestita una richiesta.

- *Request:* il frontend assegna un identificatore unico alla richiesta e fa il multicast alle altre repliche manager del gruppo.
- *Coordination:* Non c'è bisogno di coordinazione in questa fase, il multicast si prende cura di mandare tutte le richieste ai replica manager. Bisogna utilizzare il total order multicast per assicurare che le richieste siano ricevute nello stesso ordine da tutti i replica manager.
- *Execution:* Ogni replica manager esegue la richiesta. Poiché le richeiste sono consegnate in total order, anche la processazione verrà fatta nello stesso ordine.
- *Agreement:* Grazie al total order multicast non c'è bisogno di accordo tra le replica manager.
- *Response:* Ogni replica manager manda la propria risposta al frontend. Considerando che l'algoritmo multicast è il total order, il frontend può inoltrare la prima richiesta che gli arriva.

### 3.18.4 The Gossip architecture

Gli approcci attivi/passivi si concentrano sulla disponibilità, nell'approccio passivo le repliche backup non gestiscono le richieste degli users, di conseguenza tutto il carico è sulla primary replica. Nell'approccio attivo invece le repliche fanno richieste in modo ridondante. Un differente approccio fornisce allo stesso tempo alta availability e scalabilità, concentrandosi sul fornire al cliente l'accesso al servizio per quanto più tempo possibile. In particolare:

- I replica manager non inviano updates tempestivamente appena ci sono per poi dare il controllo al client, vengono invece inoltrati quando possibile ed il controllo è dato indietro al client appena possibile.
- Differenti replica manager sono utilizzati in parallelo per la gestione delle richieste provenienti dal frontend, per ottimizzare l'utilizzo delle repliche.

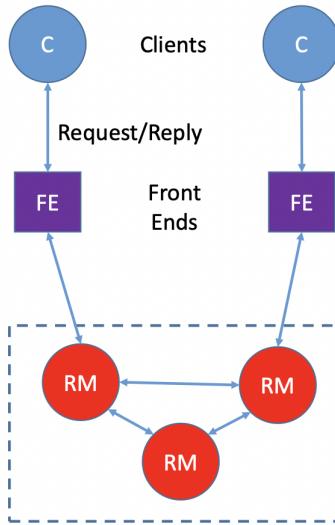


Figure 3.39: Gossip architecture

Parliamo di gossip architecture. In questo approccio i replica manager si scambiano messaggi di gossip periodicamente in modo da scambiarsi gli aggiornamenti che hanno ricevuto dai client. Ogni client è assegnato ad una specifica istanza del frontend in modo da garantire il bilanciamento, l'assegnamento tuttavia può essere cambiato se un'istanza di FE dovesse crashare. Vengono forniti due tipi fondamentali di operazioni:

- *Query*: sono operazioni di sola lettura
- *Update*: questo tipo di operazione modifica l'oggetto, l'operazione tuttavia non legge lo stato.

Il sistema di gossip porta due importanti garanzie:

- Ogni client ottiene un servizio consistente nel tempo: i replica manager rispondono al client in modo coerente con le modifiche fatte da lui fino a quel momento.

- La consistenza fra le repliche può essere dilazionata nel tempo grazie alla frequenza dei messaggi di Gossip: ad esempio un client riceve un informazione vecchia ma consistenza come risposta ad un query. È garantita la consistenza sequenziale.

## Request handling

Analizziamo come viene gestita una richiesta:

- *Request*: il frontend manda la richiesta solo un replica manager alla volta
- *Coordination*: il replica manager che riceve la richiesta non la esegue fino a quanto non la può processare seguendo i vincoli di ordinamento. Ad esempio il replica manager che riceve la richiesta deve attendere degli aggiornamenti provenienti ad altre repliche tramite messaggio di gossip. Non è necessaria altra coordinazione tra i replica manager
- *Execution*: il replica manager esegue la richiesta
- *Agreement*: i replica manager aggiornano i propri dati attraverso lo scambio di messaggi di gossip che contengono gli aggiornamenti più recenti che hanno ricevuto. I messaggi sono scambiati in maniera lazy fashion, ad esempio i messaggi vengono scambiati dopo che sono stati collezionati un certo numero di aggiornamenti o dopo che è passato un certo periodo di tempo
- *Response*: se la richiesta è un query il replica manager risponde dopo l'esecuzione, se invece è un update riceve subito dopo averlo ricevuto.

Per mantenere ordinate le richiesta si ricorre all'utilizzo dei **timestamp**, per questo motivo ogni istanza del front end deve avere il clock sincronizzato con le altre. In particolare ogni istanza del FE mantiene un vettore con i timestamp riguardo alla versione degli ultimi dati ai quali è stato fatto l'accesso. Il timestamp è mandato al replica manager attraverso il campo prev insieme alla query, il replica manager risponde con il nuovo timestamp attraverso il campo new insieme alla risposta. Ogni timestamp ritornato dai replica manager viene unito al timestamp precedente.

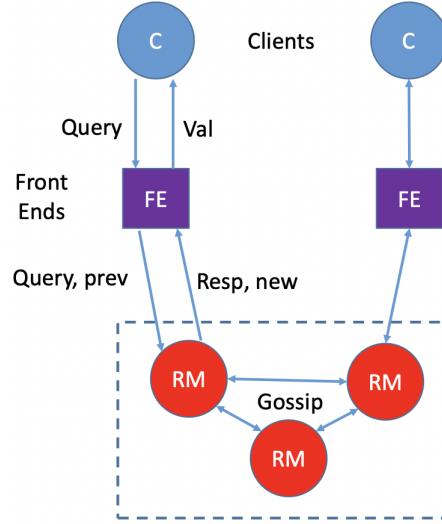


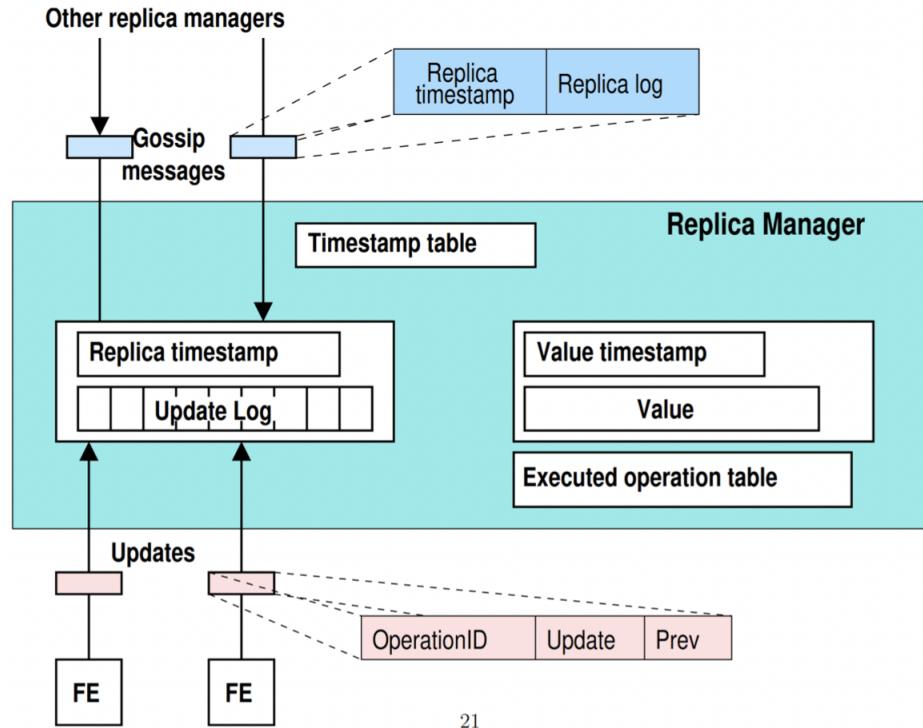
Figure 3.40: Time Stamp message order

### Processing query

Analizziamo adesso come vengono processate le query. Il timestamp che viene inviato con la query rappresenta l'ultima versione che il frontend ha letto o che ha modificato tramite un update. Per questo motivo ogni volta che la query viene ricevuta, il replica manager deve rispondere con un valore che sia almeno recente quanto il timestamp inviato dal FE insieme alla query. Quando il timestamp dell'ultima versione presente sulla replica è più vecchio di quello inviato dal FE allora la richiesta viene ritardata e messa in una coda di richieste in sospeso per aspettare l'aggiornamento mancante. Quando la query può essere eseguita la replica invia al frontend il timestamp pari all'ultima versione presente nella replica stessa.

### Processing update

Passando alla processazione delle richieste di update. Quando un replica manager riceva una richiesta di update da un FE, esso prima analizza l'ID della richiesta per controllare se sia già stata effettuata in quanto le richieste uguali hanno sempre lo stesso ID. Se la richiesta non è duplicata il replica manager crea un unico timestamp incrementando il valore di prev ricevuto dal FE e dopo il replica manager manda indietro al FE la risposta con il timestamp aggiornato. Aggiunge anche una nuova entrata nel file di log locale delle operazioni di update in sospeso. Se la solita condizione sul timestamp viene soddisfatta l'operazione è eseguita altrimenti viene ritardata fino al momento che sia soddisfatta, la condizione viene verificata tutte le volte che il replica manager riceve un nuovo messaggio di gossip.



21

Figure 3.41: Replica Manager Structure

## 3.19 Global-scale application

Consideriamo un'applicazione cloud che deve essere accessibile da centinaia di milioni di utenti da tutto il mondo e nello stesso momento, per fare un esempio pratico il *Google Web Search engine* soddisfa 5.4 miliardi di richieste al giorno da utenti in tutto il mondo. Con una mole di dati di questo tipo è chiara la necessità di un cluster di VM in modo che si bilanci il carico fra le VM stesse e che quindi si raggiunga la possibilità di scalare. Tuttavia la divisione in cluster potrebbe non essere sufficiente perché il limite di richieste massimo che può essere gestito è definito dai limiti fisici dell'infrastruttura che trasmette le richieste dai client alle VM, in altre parole il *Load Balancer Tier* può anche essere molto grande ma il vero bottleneck è situato nei collegamenti fisici che portano i dati alle VM dell'*Application Tier*. Un modo per mitigare questo bottleneck è quello di suddividere in più *data center* sparsi per il mondo le VM appartenenti all'*Application Tier* in modo da avere un bilanciamento del carico naturale in quanto il traffico è diviso per area geografica, in più anche l'infrastruttura fisica è separata.

### 3.19.1 Load Balancing

Il *Load Balancing* solitamente è eseguito a due livelli:

- **Global level:** a questo livello ci si occupa di inoltrare le richieste fra i vari datacenter a livello globale. Si utilizzano diverse politiche a seconda dell'applicazione presa in esame, ad esempio la richiesta proveniente da un servizio che offre una ricerca può essere inoltrata al datacenter più vicino in termini di RTT per garantire una latenza minima. Un servizio di upload di video, invece, può decidere di inoltrare la richiesta al datacenter meno utilizzato così da garantire il massimo throughput a discapito della latenza.
- **Local level:** a questo livello si decide quale VM debba prendere in carico la gestione della richiesta all'interno dello stesso datacenter. Solitamente a questo livello viene usata una politica che punta a bilanciare il carico per ottenere la scalabilità.

### 3.19.2 Global Load Balancing with DNS

Prima che un client possa mandare una richiesta è necessario che esso conosca l'IP address della destinazione, per ottenerlo deve prima rivolgersi al DNS. Il GLB DNS sfrutta quindi questa fase per introdurre un primo livello per la distribuzione della richiesta. La soluzione è quella di configurare i record del DNS in modo che restituiscano *AAAA records* (record con molteplici A) nella risposta in modo che il client scelga arbitrariamente l'ip di destinazione, tuttavia questo metodo ha delle limitazioni. Esse sono rappresentate dal fatto che *garantiscono poco controllo sul comportamento del client* in quanto molto spesso i client scelgono gli IP in maniera casuale non potendo sapere quali rappresentano indirizzi più vicini. Per sopperire al fatto che i client non utilizzano la distanza come paramentro della scelta dell'IP viene utilizzato il *Localized DNS* in modo che le risposte siano determinate in base alla disposizione geografica dei client. Questo metodo è il semplice e efficiente anche se offre un controllo a grana grossolana in quanto un record DNS non può essere aggiornato in tempo reale perché per cambiare un'informazione di un record possono anche essere necessarie delle ore. Un controllo più specifico può essere offerto dai *Virtual IPs*.

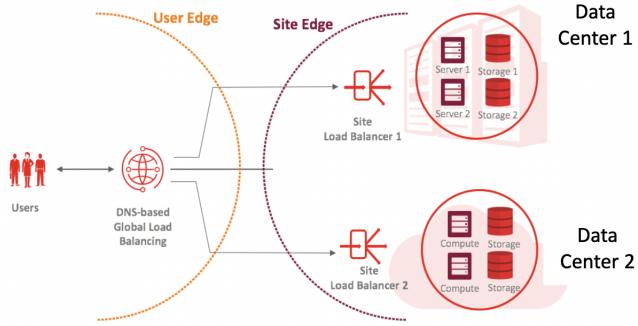


Figure 3.42: Global Load Balancing with DNS

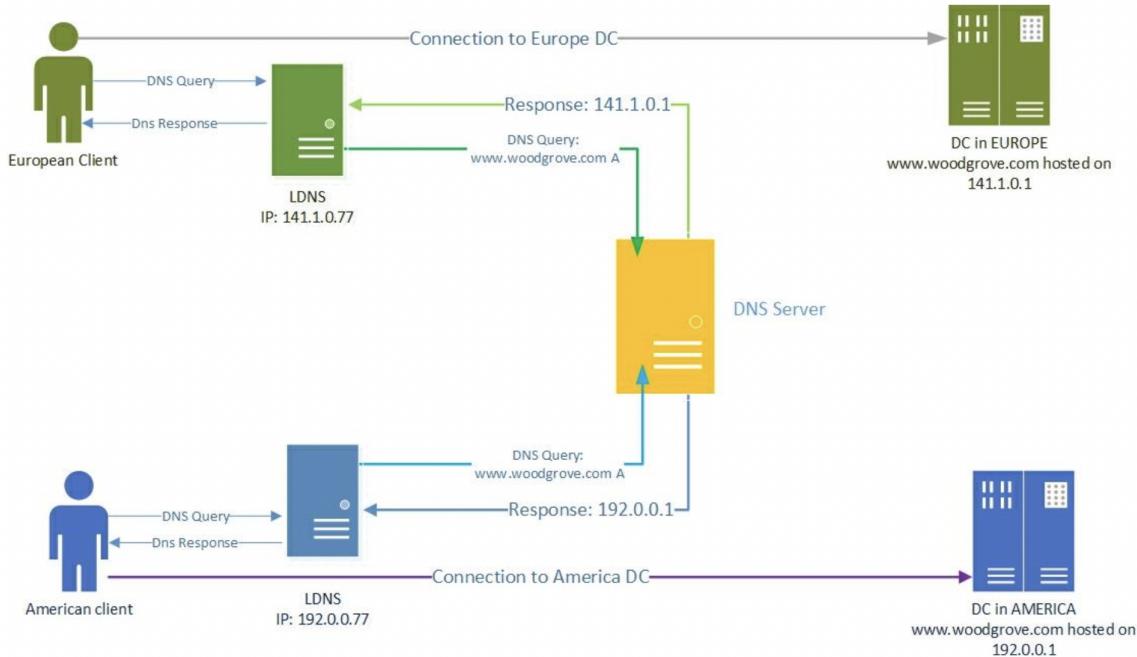


Figure 3.43: General structure

### 3.19.3 Virtual IP addresses

Un *Virtual IP address* (VIP) è un indirizzo IP che però non è assegnato ad un host o a una scheda di rete specifica, esso viene condiviso fra più device differenti. Dal punto di vista dell'utente il VIP rimane un semplice indirizzo IP andando quindi a mascherare l'implementazione vera e propria. Per utilizzare questo metodo si possono, ad esempio, assegnare ad un'applicazione una serie di VIP e ogni VIP è poi assegnato ad esempio ad una regione (prendendo in esame un'applicazione come un web search engine). Dietro al singolo VIP possono esserci molteplici datacenter. Il regional DNS è configurato con i VIP assegnati a una determinata regione per un servizio specifico. Per ogni regione è presente uno o più *Global Load Balancer*, essi sono installati sull'infrastruttura dell'ISP o in dei nodi specifici dove diversi IPS si interconnettono. Ad ogni Global Load Balancer viene assegnato un VIP fra quelli assegnati alla regione.

## Packet Encapsulation

Il *Global Load Balancer* è responsabile di ricevere le richieste e di inoltrarle al datacenter più conveniente in quel momento, la selezione del datacenter può quindi variare nel tempo. Questo processo viene portato a termine tramite il **packet encapsulation**, in altre parole il pacchetto prima di essere spedito al datacenter viene incapsulato in un altro pacchetto. A destinazione il pacchetto viene decapsulato e processato come se fosse stato ricevuto direttamente da quel datacenter, questa operazione viene svolta solitamente dal *datacenter local load-balancer*. Per incapsulare i pacchetti si usa il *Generic Routing Encapsulation (GRE)*, solitamente si utilizza per instaurare un tunnel fra il global load balancer e il datacenter. Il GRE permette anche di incapsulare un pacchetto IP dentro un altro pacchetto IP, il pacchetto più esterno contiene l'indirizzo IP del load-balancer del datacenter di destinazione.

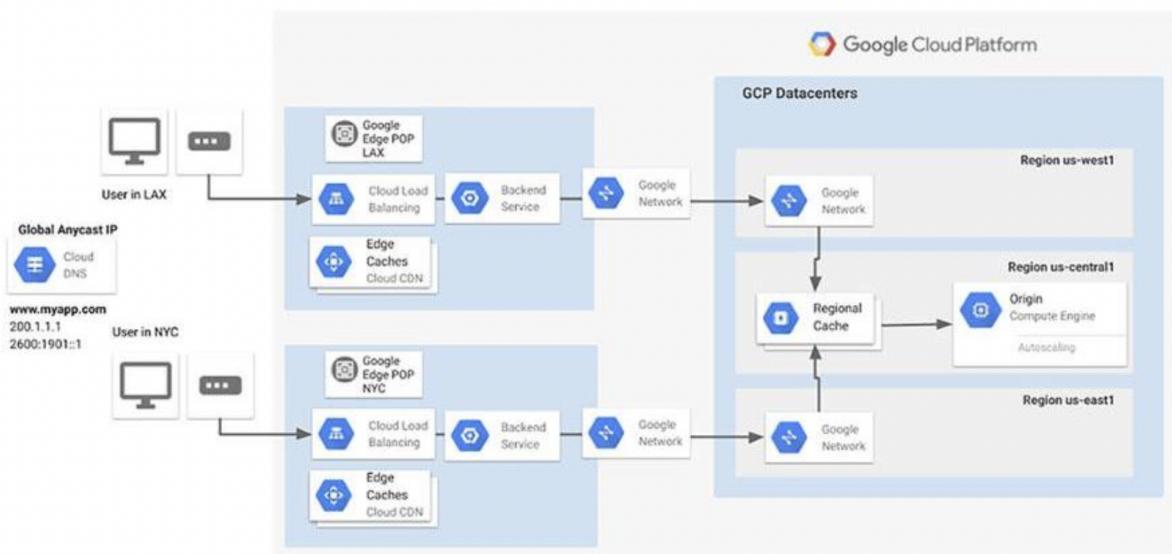


Figure 3.44: Load Balancing with Virtual IP

## 3.20 Geographically distributed systems

I sistemi distribuiti a livello globale necessitano di una visione consistente sullo stato del sistema e dei dati fra tutti i siti. La *replicazione dei dati* e la *sincronizzazione di essi* fra siti differenti che sono situati in diverse parti del mondo che sono collegati tramite *link WAN* dal momento che:

- La banda è limitata: questo fattore può aumentare in modo significativo il tempo impiegato per la sincronizzazione dei dati.
- Poca affidabilità nella comunicazione: questo fattore può porta a frequenti *network partition*.

A causa di questi motivi le *strategie di replicazione dei dati tradizionali* non possono essere adeguate in quanto esse si basano su una connessione affidabile e su una bassa latenza per lo scambio dei dati. In particolare le **Network partition** non sono considerate in questo tipo di strategie. Un nodo può disconnettersi per un periodo di tempo durante il quale non può erogare il servizio, quando torna online esso deve sincronizzare i dati con gli altri nodi. In un sistema distribuito a livello globale le **network partition** devono essere tollerate in quanto sono inevitabili. Una regione o un datacenter può disconnettersi dalle altre per un periodo di tempo durante il quale gli altri **devono** continuare a erogare il servizio.

Non si possono usare transazioni di tipo ACID in quanto si concentrano sulla consistenza dei dati e solo in maniera secondaria alla disponibilità, per assicurarla si fa uso della replicazione dei dati, che però viene eseguita in maniera rigorosa per mantenere la consistenza dei dati. Le tecniche di replicazione tradizionali si basano sul fatto che le partizioni siano rare in quanto le VM partizionate vengono considerate non funzionanti. Questo approccio non è compatibili con le applicazioni distribuite a livello globale, nelle quali le partizioni sono frequenti e gli elementi partizionati devono continuare ad assicurare il servizio.

## 3.21 CAP theorem

Un sistema distribuito può avere le seguenti caratteristiche:

- Consistency: tutti i nodi devono vedere gli stessi dati nello stesso momento.
- Availability: il fallimento di un nodo non deve compromettere il funzionamento degli altri.
- Partition-tolerance: il sistema continua a funzionare nonostante una partizione della rete.

Secondo il **Teorema CAP** un sistema distribuito può soddisfare al più due delle caratteristiche elencate contemporaneamente. Questo teorema porta inevitabilmente a dover individuare dei compromessi nel design del sistema distribuito. Seguendo il teorema sono possibili 3 tipo di sistemi distribuiti:

- CP: Consistency - Partition tolerance

- AP: Availability - Partition tolerance
- CA: Consistency - Availability

Il teorema assume che non è raggiungibile un perfetto availability insieme alla presenza di partizioni. È importante sottolineare che la terza scelta che non si persegue non viene ignorata del tutto, ad esempio i sistemi *AP* talasciano la consistenza a favore della disponibilità ma non sono incostistenti.

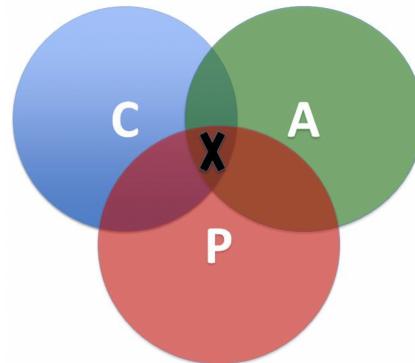


Figure 3.45: CAP Theorem

## 3.22 BASE semantic

Questa semantica è stata definita per i sistemi che devono gestire un volume di dati molto elevato, per i quali la semantica ACID non è perseguitabile. Questa nuova semantica prende il nome di **BASE**: *Basically Available, Soft state and Eventual consistency*. Il concetto di **eventual consistency** è alla base di questa nuova semantica: lo stato del sistema e i suoi dati possono rimanere inconsistenti per un determinato periodo di tempo, però è garantito che si raggiunga la consistenza dopo un periodo di transizione. Questo concetto è l'opposto della *strict consistency* secondo la quale, dopo un update, tutti devono essere in grado di leggere il dato aggiornato. L'approccio con l'eventual consistency può essere applicato ad applicazioni che non hanno nei requisiti quello di avere i dati consistenti durante il periodo di transizione come ad esempio un social network, lo stesso non vale per applicazione e servizio come quelli di una banca dove i dati devono essere sempre consistenti.

### Query/Update

La differenza principale fra le due semantiche è il fatto che gli update sono festiti in maniera differente, vediamo come:

- I sistemi ACID trasmettono gli update in maniera sincrona, questo porta al fatto che l'utente che invia l'update ad aspettare che tutte le macchine lo abbiano ricevuto.
- I sistemi BASE replicano i dati in maniera asincrona in background. La richiesta viene gestita immediatamente e i dati vengono aggiornati in locale, gli update vengono propagati in maniera differita seguendo un protocollo chiamato **anti-entropy protocol**.

Le query invece vengono gestite semplicemente rispondendo con la copia dei dati locali.

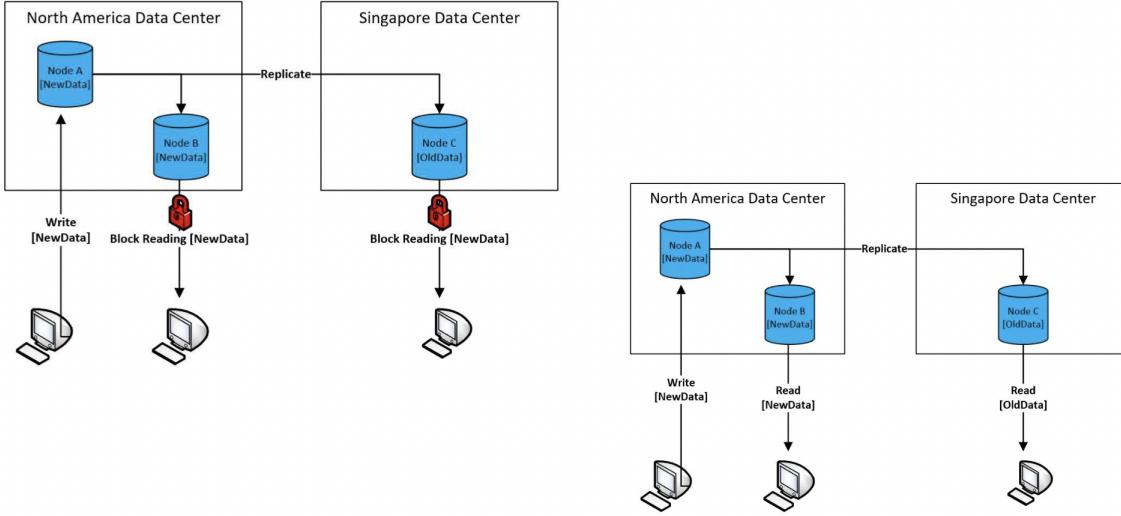


Figure 3.46: ACID vs BASE

### 3.22.1 Inconsistency

A causa della propagazione differita degli update è possibile incorrere in delle inconsistenze, ad esempio una query può restituire dei dati inconsistenti se uno specifico update non ha ancora raggiunto il nodo che deve gestire la query. Oltre alle inconsistenze possono avvenire anche dei conflitti se due richieste di update provengono da due nodi. Quando sorgono dei conflitti è necessario che intervenga un meccanismo di risoluzione dei conflitti, ad esempio si può banalmente scegliere di mantenere l'update con il timestamp più recente scartando tutti gli altri.

### 3.22.2 Advantages/Disadvantages

Questo tipo di semantica permette di gestire la partizione della rete in quanto se avviene la partizione le restanti regioni o datacenter possono continuare ad operare senza interruzione, quando la partizione incriminata torna ad essere operativa l'*anti-entropy protocol* distribuirà i dati per raggiungere l'*eventual consistency*. In più la risposta ha un delay minimo dovuto al fatto che i dati non si devono sincronizzare fra tutte le macchine. Tuttavia questo approccio non è sostenibile per i sistemi che non tollerano i conflitti.

## 3.23 Bayou architecture

L'architettura Bayou è un'architettura *eventual consistent* che fornisce la replicazione dei dati per fornire high availability con però **scarsa garanzia dell'arrivo sequenziale dei dati**. Ogni replica manager scambia i dati con un anti-entropy protocol per fornire updated in gruppo, l'architettura fornisce anche una politica di risoluzione dei conflitti di tipo *domain-specific*.

### 3.23.1 Bayou operations

Parlando delle operazioni, quando viene ricevuto un update, esso viene applicato e marcato come **tentativo**, fino a che l'update è un *tentativo* il sistema può eliminare e riapplicare l'update per raggiungere uno stato consistente. Quando l'update diventa *committed* rimane necessariamente applicato. In ogni momento il sistema organizza gli update in quelli *committed* in ordine di arrivo seguiti dagli update *tentativo* generalmente ordinati in base al timestamp di ricezione.

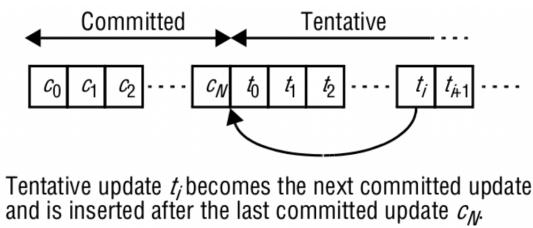


Figure 3.47: Bayou operations

### 3.23.2 Primary replica manager

L'architettura Bayou adotta uno schema *primary commit*, in altre parole solo un replica manager è selezionato come primario ed è responsabile di rendere gli update *committed*. La lista di questi update è scambiata tramire l'anti-entropy protocollo, e, oltre a decidere quando un update diventa *committed*, il primary replica manager si comporta come gli altri.

Il protocollo di commit è una politica che stabilisce in quale ordine gli update sono applicati e può essere pensata in maniera specifica per applicazione. Se il primary replica manager diventa non disponibile non si crea nessun tipo di problema in quanto gli update vengono accettati come tentativi e le query restituiscono dati basandosi sullo stato corrente del sistema, quando il primary torna disponibile allora procede a rendere permanenti gli update.

### 3.23.3 Conflict detection and resolution

Quando un update è applicato, sia come tentativo che come permanente, possono sorgere dei conflitti, l'architettura Bayou adotta due meccanismi per trovare e risolvere i conflitti:

- Dependency check: vengono forniti dagli sviluppatori delle query e quello che ci si aspetta di ricevere per capire se determinati prerequisiti sono soddisfatti per l'esecuzione dell'update. Prima di applicare un update viene eseguita la query e se il risultato è diverso da quello fornito dagli sviluppatori allora siamo davanti a un conflitto.

- merge procedure: è una procedura che viene eseguita quando si trova un conflitto, la procedura è ovviamente fornita dagli sviluppatori.

Entrambi i tipi sono deterministicici in modo che i conflitti possano essere risolti nello stesso modo da tutte le macchine. Come risultato si ha che nel replica manager è presente la storia della risoluzione dei conflitti e quella dell'esecuzione.

### 3.23.4 Performance measurement

I sistemi eventually consistent sono molto usati anche non forniscono molta sicurezza. Un paramentro per misurare l'efficenza di una sistema è il tempo:

- la finestra di incosistenza.
- il tempo necessario affinchè un'informazione sia visibile nel sistema.

Entrambi sono dipendenti dall'anti-entropy protocol e più in particolare dall'anti-entropy rate, cioè dalla frequenza alla quale il protocollo invia gli aggiornamenti. Dato un certo enti-entropy rate il tempo atteso per la consistenza può essere calcolato per ottenere un certo *Probabilistically Bounded Staleness* (PBS), molti sistemi permettono di modificare alcuni parametri per ottenere un certo PBS.

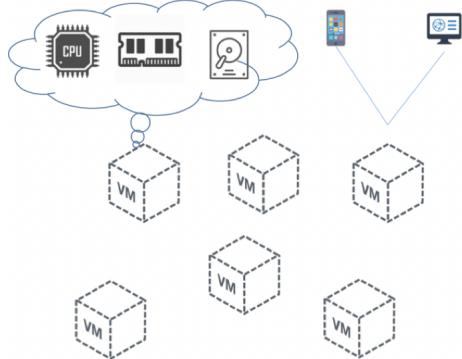
# Chapter 4

## Cloud Platform

### 4.1 Cloud computing platform

#### 4.1.1 Recap: The cloud (for now)

Ad ora il cloud è un'infrastruttura che offre la possibilità di creare VM. Tale infrastruttura è progettata, implementata e mantenuta dal cloud provider che vende risorse virtualizzate per ottenere del profitto. In cima alle VM vengono create applicazioni dal cloud consumer, questi servizi sono offerti all'utente finale con lo scopo di profitto o meno. Sappiamo già come creare una VM su una singola macchina usando le tecniche di virtualizzazione. Com'è fatta una infrastruttura di cloud Computing? Come possiamo creare un'infrastruttura che ci permette di creare e controllare un grande numero di VM su larga scala?



#### 4.1.2 The cloud infrastructure

Recenti tecniche di virtualizzazione permettono di creare delle VM in cima ad hardware non modificato, un potente server può supportare la creazione e l'esecuzione di multiple VM allo stesso tempo e sullo stesso hardware. Come abbiamo detto l'hardware non è modificato cioè non richiede nessuna modifica specifica. Un gruppo di server è interconnesso tramite una LAN ad alta velocità, compresa nella infrastruttura cloud. Questi server sono connessi ad Internet tramite un router quindi ogni VM può comunicare con l'utente e viceversa. Essi si troveranno ovviamente in un data Centre, ogni server avrà una hypervisor installato su di esso.

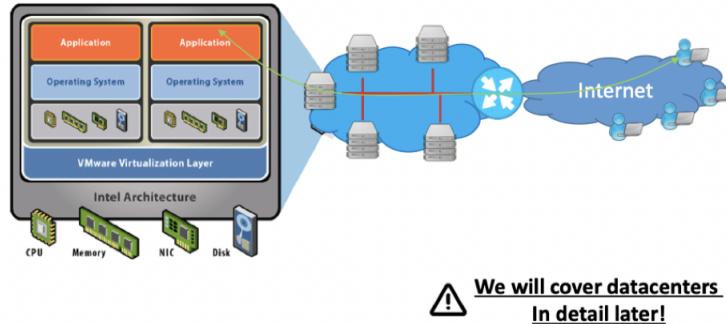


Figure 4.1: Cloud Infrastructure.

#### 4.1.3 Cloud Platform

Un Pool di server, ognuno che esegue un hypervisor, creano un cloud Platform. Esso è un sistema distribuito (può essere considerato un tipo speciale di applicazioni cloud) che controlla l'hypervisor di tutti i server dell'infrastruttura, cioè si occupa di gestire la creazione e la distruzione di VM dell'infrastruttura, ad esempio ad ogni momento controlla quale VM deve essere in esecuzione, su quale server e con quali proprietà. Per permettere ciò la piattaforma espone un'interfaccia di controllo. Questa interfaccia può essere usata da un umano per controllare la generazione delle VM oppure da un software per automatizzare il controllo. La cloud platform permette di implementare il modello cloud IaaS, in cima all'infrastruttura possono essere implementati modelli più complessi (PaaS e SaaS).

#### General Architecture

In generale l'architettura è semplice: un insieme di server sono connessi tramite una LAN ad alta velocità. Ogni server ha il proprio sistema operativo installato sopra l'hardware fisico (bare metal), sopra il sistema operativo viene installato l'hypervisor per virtualizzare le risorse fisiche offerte dal server. Inoltre all'hypervisor viene installato anche un software di cloud Platform per controllare le risorse locali virtualizzate dall'hypervisor, questo software viene spesso riferito come l'agent della cloud platform. Con lo scopo di implementare un largo sistema di gestione e controllo delle funzionalità del cloud Platform viene installato su un server una particolare istanza del cloud platform software, il cloud platform controller. Questo software può essere installato su un server che non ha l'hypervisor oppure potrebbe anche essere installato su una VM, è responsabilità del software controllare l'intera piattaforma comunicando con gli agent.

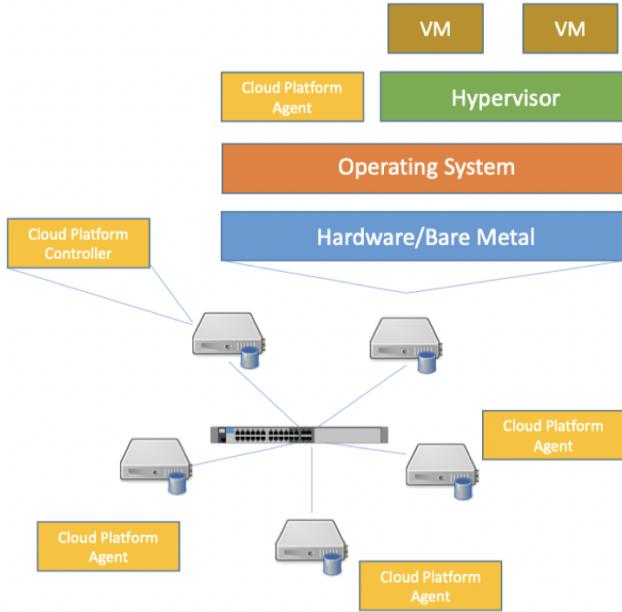


Figure 4.2: Cloud Architecture.

L’agent controlla l’hypervisor per creare e distruggere le VM, il controller coordina e gestisce tutti gli agent, indicando quale VM deve essere creata o distrutta.

### Cloud platform implementations

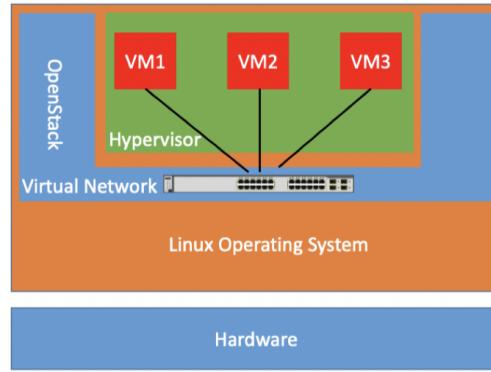
Sono presenti cloud Platform pubbliche o private, alcune sono commerciali ed altre open source. Ad oggi non è presente uno standard per l’architettura di cloud platform, anche se sono molto simili. In questo corso analizzeremmo l’architettura e le funzionalità di Open Stack.

## 4.2 OpenStack

Open stack è un software platform open source per Iaas Cloud Computing. Openstack è supportata e gestita da una fondazione che comprende oltre 500 compagnie (Cisco, Oracle, Ericsson, IBM, ecc). E’ la soluzione primaria adottata per il cloud computing privato.

### 4.2.1 OpenStack Software Platform

Il platform software viene installato su ogni server e gira sopra il SO dell’host. La piattaforma può essere eseguita solo su Linux OS, ma supporta molte tecnologie di hypervisor. Ogni server che ha installato il software OpenStack viene chiamato OpenStack Node. La piattaforma gestisce la connessione delle VM alle virtual network, che possono essere usate per comunicare con altre VM o con reti esterne. La piattaforma gestisce anche lo storage, questo permette la creazione di virtual hard drive che possono essere connessi alle VM.



#### 4.2.2 OpenStack instances

Single OpenStack instance: formata dai nodi che eseguono il software di OpenStack, ogni nodo contribuisce al computing, memory and storage per creare le VM. In una singola istanza, almeno un nodo viene configurato come controller che deve coordinare le funzioni e gestire le risorse disponibili all'istanza. Gli altri nodi vengono configurati come compute, offrono risorse computazionali e di archiviazione per eseguire VM. I nodi vengono connessi tramite una rete reale ad alta velocità che viene usata come:

- Infrastruttura sopra alla quale vengono create VM
- Rete di comunicazione usata dai componenti di openstack per comunicare, coordinare e gestire.

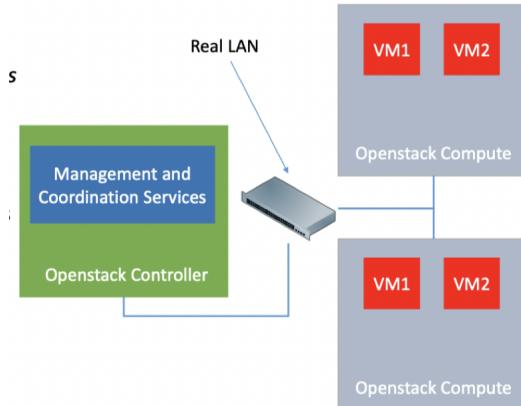
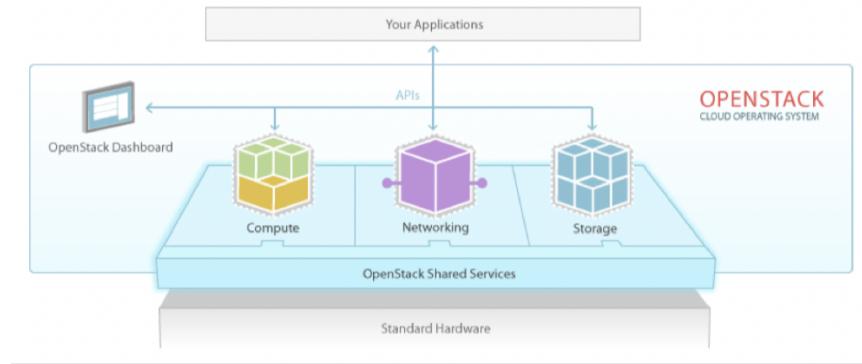


Figure 4.3: OpenStack instance.

### 4.2.3 OpenStack Architecture

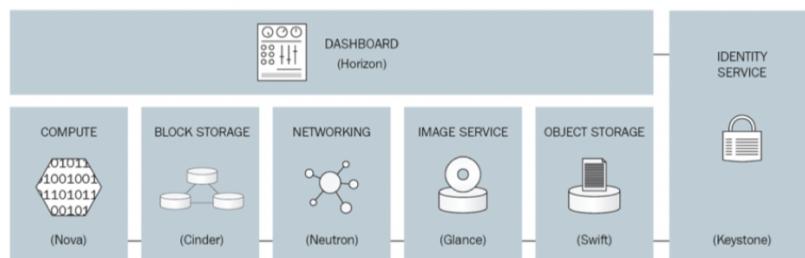
La piattaforma espone una dashboard web per permettere agli utenti e agli amministratori di gestire le risorse virtuali, viene offerta anche un'interfaccia per gestire da linea di comando. Inoltre è presente un'interfaccia basata su REST API che possono essere usati per programmi esterni.



Il software OpenStack è modulare:

- Le funzionalità della piattaforma sono raggruppate in servizi, per esempio i servizi network per gestire la creazione di reti virtuali, storage service, compute service.
- Ogni servizio viene implementato come un modulo separato
- Ogni modulo potrebbe lavorare separatamente dalla piattaforma e la progettazione e il mantenimento come progetti separati

Ogni servizio espone un set di REST API, tramite questa interfaccia è possibile ricevere richieste dagli utenti o amministratori tramite web/linea di comando e da altri servizi. La coordinazione delle operazioni e lo stato dei servizi vengono invece gestiti tramite scambio di messaggi utilizzando un sistema di coda.



#### 4.2.4 OpenStack Services

L'unico servizio obbligatorio in ogni installazione è il Core Service, gli altri servizi sono opzionali quindi possono essere installati solo se le funzionalità fornite necessitano di una specifica istanza OpenStack.

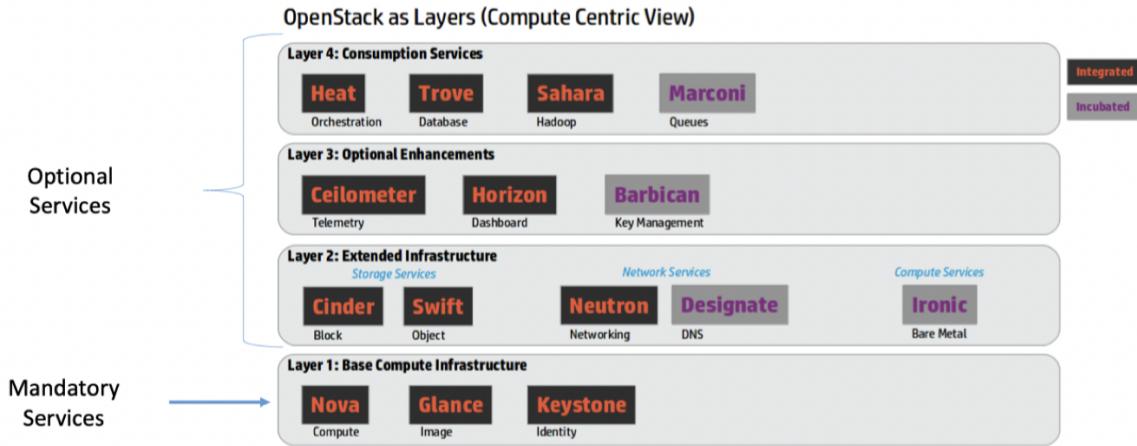


Figure 4.4: OpenStack Services.

I servizi sono installati sul controller oppure sui compute a seconda delle loro funzionalità, alcuni servizi devono essere installati su entrambi con differenti configurazioni. Tutti i servizi del controller sfruttano servizi di supporto come ad esempio il Database (es MySQL) per lo stato e la configurazione oppure il Message Broker (es Rabbit MQ) per lo scambio di messaggi tra moduli differenti.

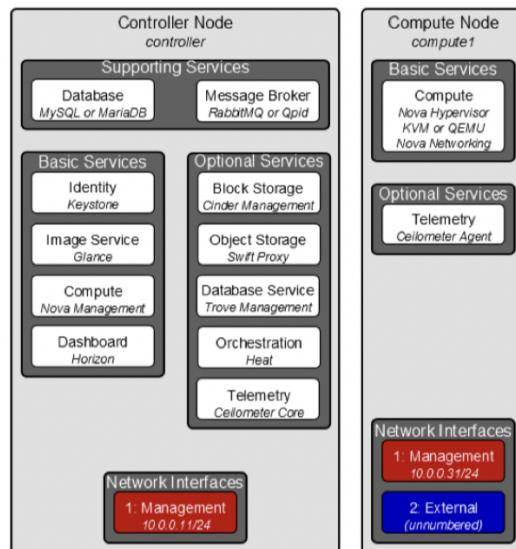


Figure 4.5: Services division by controller and compute.

#### 4.2.5 Information Exchange/Communication

Dentro la piattaforma, servizi e sotto servizi interagiscono tra di loro in tre modi differenti:

- Message Broker: lo scambio di messaggi è diretto ed avviene tramite il message broker
- REST API: un servizio invoca l'interfaccia REST API nello stesso modo fatto per un programma esterno
- Database: due moduli scambiano informazioni tramite il db, richieste/risposte sono fatte tramite una connessione TCP/IP.

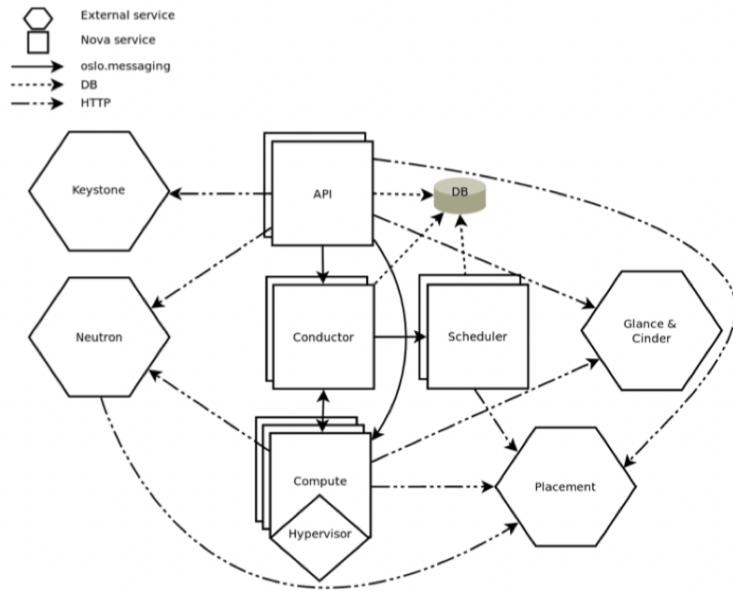


Figure 4.6: Example of Microservice-based application.

#### 4.2.6 Keystone

Keystone è il **componente di gestione dell'identità**, viene utilizzato da Openstack per l'autenticazione e l'autorizzazione ad alto livello. Solitamente viene installato sul nodo Controller.

Garantisce la sicurezza garantendo/negando l'accesso agli oggetti (ad es. Macchine Virtuali o Reti Virtuali) a diversi **Utenti**. Gli oggetti sono raggruppati in progetti, le autorizzazioni possono essere concesse per progetto assegnato ad un utente.

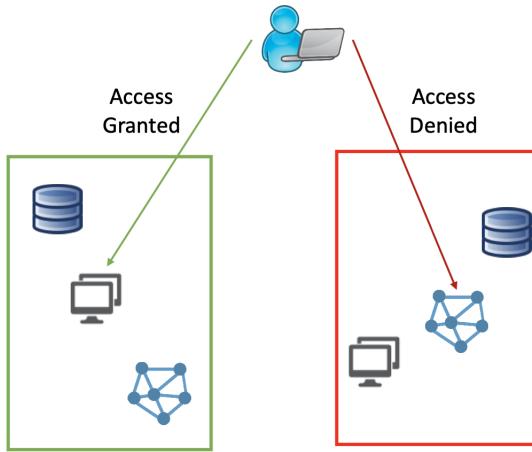


Figure 4.7: Example of Microservice-based application.

Keystone implementa un'autorizzazione basata su token. Un utente interagisce prima con keystone utilizzando un'autorizzazione basata su user/pass, se il token viene correttamente ricevuto verrà usato dallo user per accedere ai servizi di OpenStack, ogni servizio dovrà validare il token.

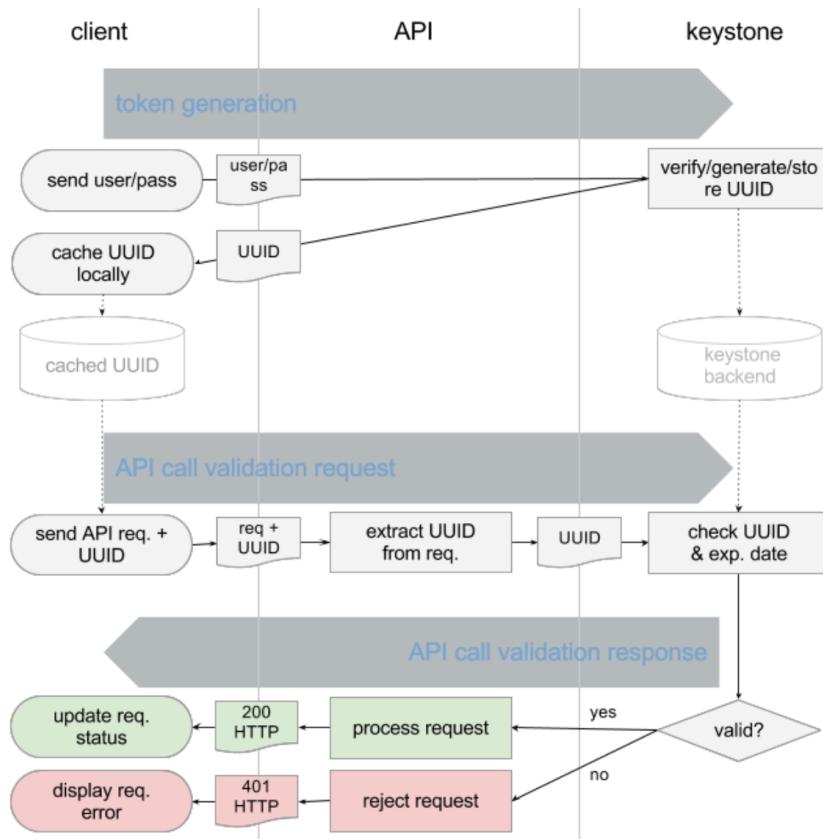


Figure 4.8: Example of Microservice-based application.

#### 4.2.7 Nova

Nova è il componente di **gestione le VM**, ad es. il servizio di compute. E' responsabile della *creazione* e *gestione* di VM, non implementa una nuova tecnologia di virtualizzazione, ma si interfaccia con gli hypervisor esistenti per gestire le VM.

Si possono usare diverse tecnologie di virtualizzazione, dalle tecnologie open-source (ad es. KVM, Xen) a quelle commerciali (ad es. Vmware ESX). Include due diversi moduli, il *controller nova* installato sul nodo del controller e il *nova agent* installato sui nodi di calcolo.

#### Nova Subcomponents

Il modulo Nova è installato sul nodo **controller** e sul nodo di **compute**. Sul primo il componente nova è responsabile della gestione delle richieste degli utenti e delle risorse disponibili sui compute node. In particolare, raccoglie le richieste di creazione/distruzione di VM dall'interfaccia, valuta le azioni corrispondenti da eseguire e invia i comandi al compute node. Sul compute node, il componente è responsabile della ricezione di istruzioni dal modulo di controllo e di tradurlo ai comandi corrispondenti per l'hypervisor.

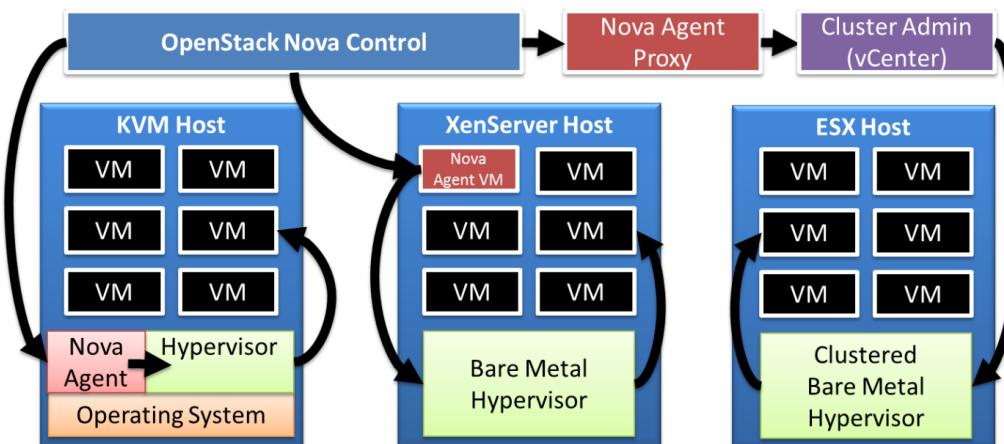


Figure 4.9: Nova subcomponents in controller and compute node.

#### Controller Subcomponents

Il modulo Nova installato sul nodo **controller** è composto da:

- **Nova-API:** mostra l'interfaccia esterna REST utilizzata dagli utenti e da altri servizi.
- **Nova-Conductor:** gestisce tutte le operazioni di controllo
- **Nova-Scheduler:** valutare il posizionamento corretto di una nuova VM in base allo stato dei nodi di elaborazione e alle risorse disponibili
- **Nova-Network:** implementa servizi di rete di base per VM o interfacce con altri servizi di rete se installati

- **Nova-Consoleauth e Nova-Novncproxy**: implementa un servizio di console attraverso il quale gli utenti possono connettersi alle VM

Tutti i componenti interagiscono attraverso la coda di messaggistica.

## Compute Subcomponents

Nel **compute node** il modulo nova è composto solo dal **compute service**. Il *compute service* riceve comandi dal controller (Conductor service) e istanzia/termina le istanze delle VM interagendo con l'hypervisor. I driver per diversi hypervisor sono mantenuti per interfacciare il compute service a diversi hypervisor:

- Vmware Esxi, un hypervisor commerciale di Vmware
- Xen, Libvirt, Qemu, Docker

Ogni driver espone un'interfaccia comune utilizzata dall'agente nova-compute per interagire con ogni hypervisor.

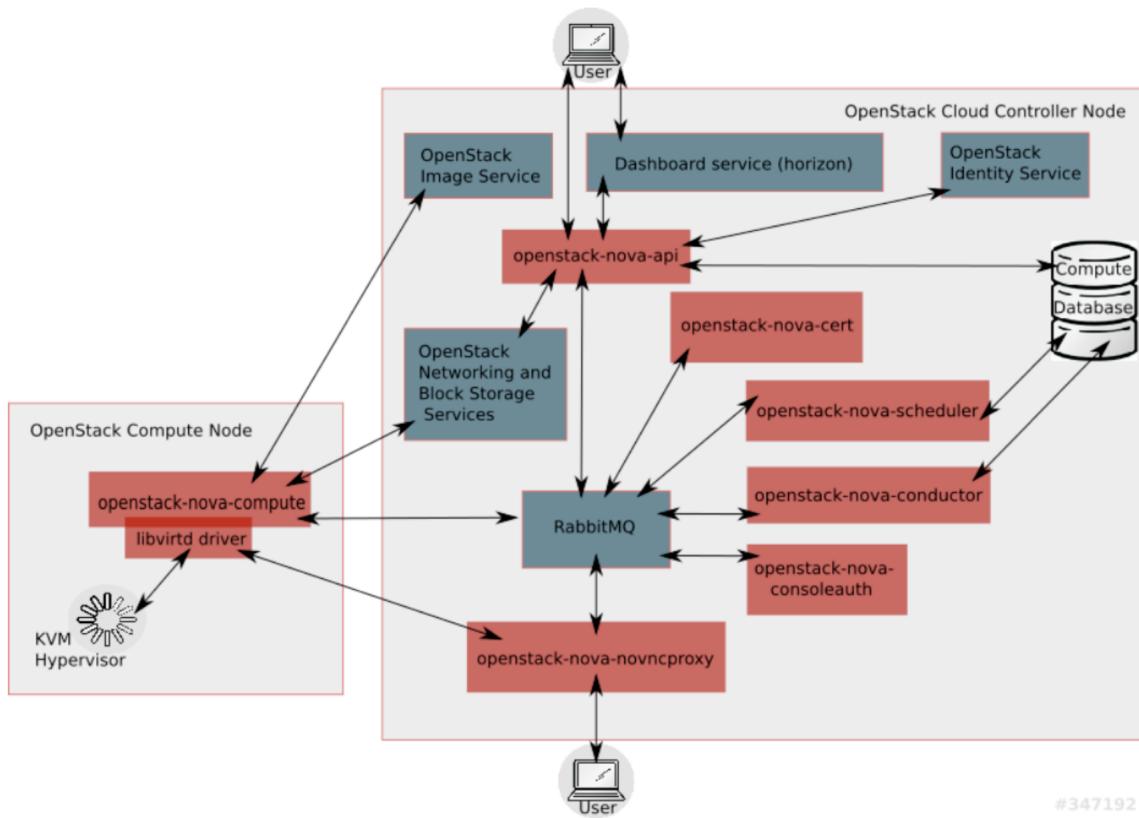


Figure 4.10: Compute subcomponents.

#347192

## Nova - VM Creation Workflow

Quando una richiesta di creazione di una nuova VM viene emessa dall'utente, i componenti nova eseguono le seguenti operazioni:

1. Nova-API riceve la richiesta che viene inoltrata a Nova-Conductor
2. Nova-Conductor inoltra la richiesta a Nova-Scheduler, il che suggerisce il corretto posizionamento per la nuova VM (che il nodo di elaborazione ha risorse disponibili)
3. Se il Nova-Scheduler consente la creazione della VM (sono disponibili risorse sufficienti nel sistema) Nova-Conductor inoltra la richiesta al servizio Nova-Network per istanziare la corretta configurazione di rete
4. Poiché la rete è configurata correttamente, la richiesta viene inoltrata al servizio Nova-Compute del nodo selezionato da Nova-Scheduler
5. Il Nova-Compute locale interagisce con l'hypervisor per creare la VM

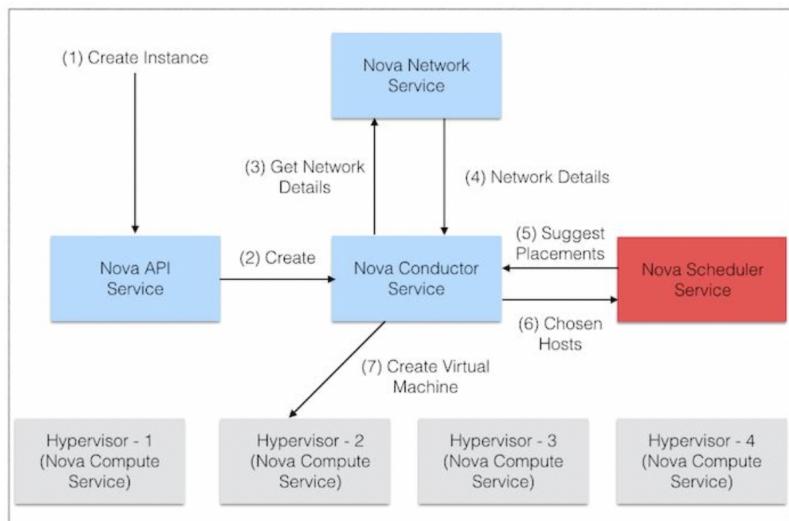


Figure 4.11: VM Creation workflow.

## Nova scheduling strategy and overcommitment

La politica di scheduling può essere personalizzata per soddisfare esigenze diverse. Lo scheduler esegue una fase di filtraggio iniziale in cui vengono rimossi gli host non adatti (non hanno una caratteristica specifica come un passedthrough-device o sono fuori dalle risorse), quindi va ad ordinare gli host. La politica può essere configurata per includere un certo grado di overcommitment per RAM e CPU, ad es. più VCPU allocate che CPU disponibili in un host (la CPU andrà più lenta), più RAM di quella disponibile nel sistema viene allocata alle VM (l'host farà swap).

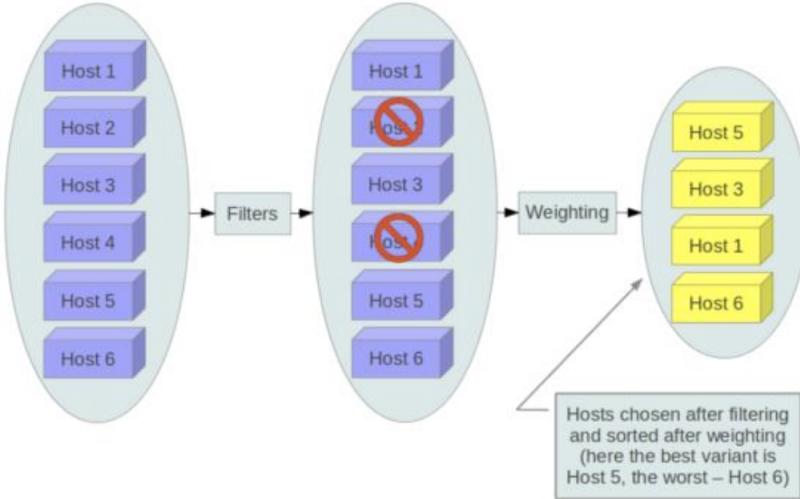


Figure 4.12: Nova Scheduler.

#### 4.2.8 Glance

Glance è il servizio di **gestione delle immagini**, ogni VM è *istanziata da un'immagine* che include un *sistema operativo specifico pre-installato e pre-configurato*. Le immagini possono essere personalizzate, ad es. un'immagine del server web può avere pre-installato un pacchetto web server, Glance gestisce la collezione di *VM templates*. Sottocomponenti di Glance: **glance** (per la gestione delle immagini) e **glance storage** (per la gestione dello storage).

**Glance** è responsabile dell'esposizione di una API REST per la gestione delle immagini (ad es. caricamento di un'immagine, cancellazione di un'immagine, ecc.) e memorizza l'elenco delle immagini disponibili e delle sue caratteristiche sul DB.

**Glance storage**, invece, è responsabile della memorizzazione delle immagini. Supporta diverse opzioni di archiviazione tramite diversi driver, tra cui:

- Il file system locale del nodo su cui è installato il servizio
- Un file system cloud, ad es. un file system distribuito (tra diversi host) o un cloud storage (li tratteremo più avanti nei dettagli!).

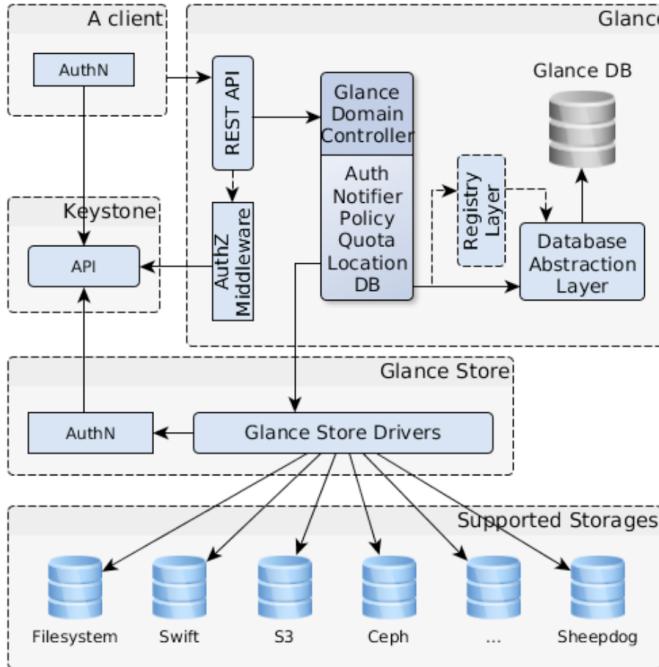
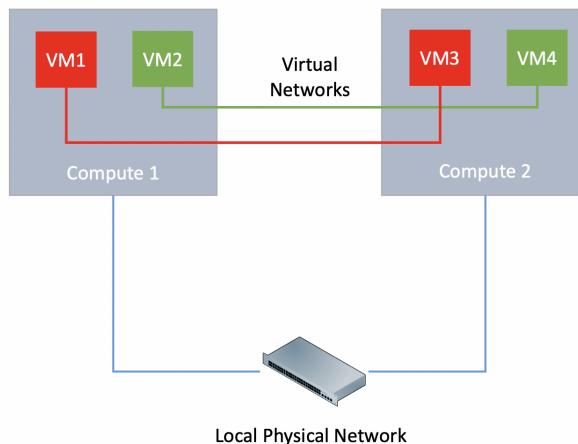


Figure 4.13: Glance Architecture.

#### 4.2.9 Neutron

Neutron è il componente di gestione della rete, le VM richiedono una rete virtuale per comunicare tra loro. Diverse reti virtuali sono istanziate per diverse VM al fine di garantire l'isolamento tra di loro, ad es. VM create da diversi tenants non possono comunicare.

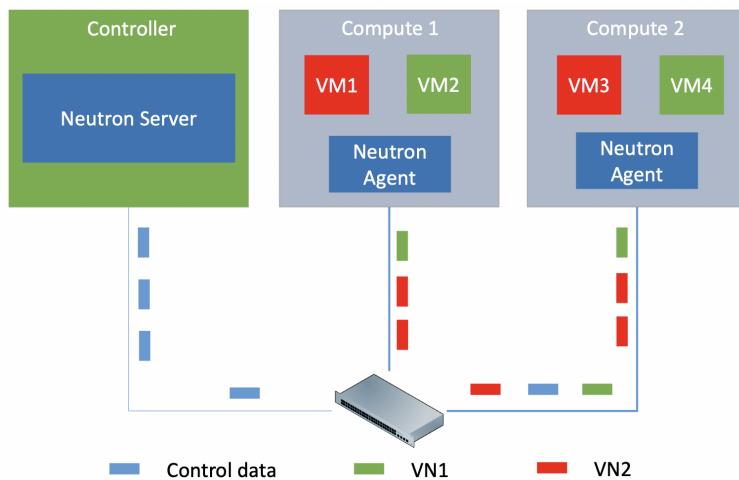
Neutron è il servizio responsabile della gestione dell'infrastruttura di rete virtuale, permette la creazione di Reti Virtuali tra VM che possono funzionare su diversi compute node di Openstack. La rete fisica locale che interconnette i nodi di calcolo viene sfruttata come infrastruttura per abilitare le reti virtuali su diversi compute node.



## Subcomponents

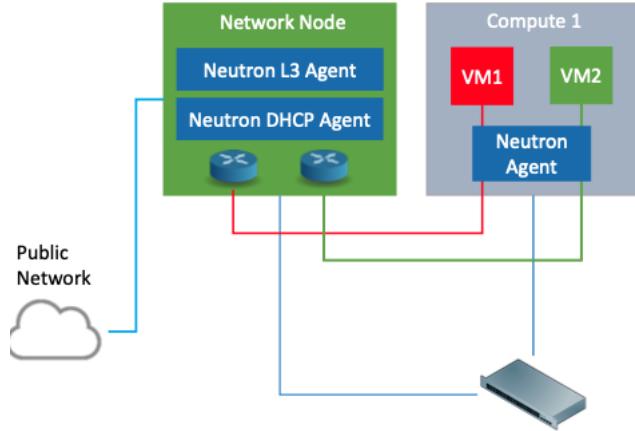
I componenti di neutron sono: neutron-server e neutron-agent.

- **Neutron-Server:** è installato sul controller ed è responsabile della gestione delle Reti Virtuali coordinando i *neutron-agent* in esecuzione sui compute node. È inoltre responsabile dell'esposizione delle API REST per la gestione delle reti virtuali agli utenti.
- **Neutron-Agent:** è responsabile della gestione del traffico da/verso le VM in esecuzione sul nodo di elaborazione su cui è installato. In particolare, distribuisce il traffico verso/da le VM locali al fine di emulare diverse reti virtuali che si estendono su diversi compute node. I dati vengono inviati in cima alla rete fisica locale. L'agente deve imporre l'isolamento tra diverse reti virtuali



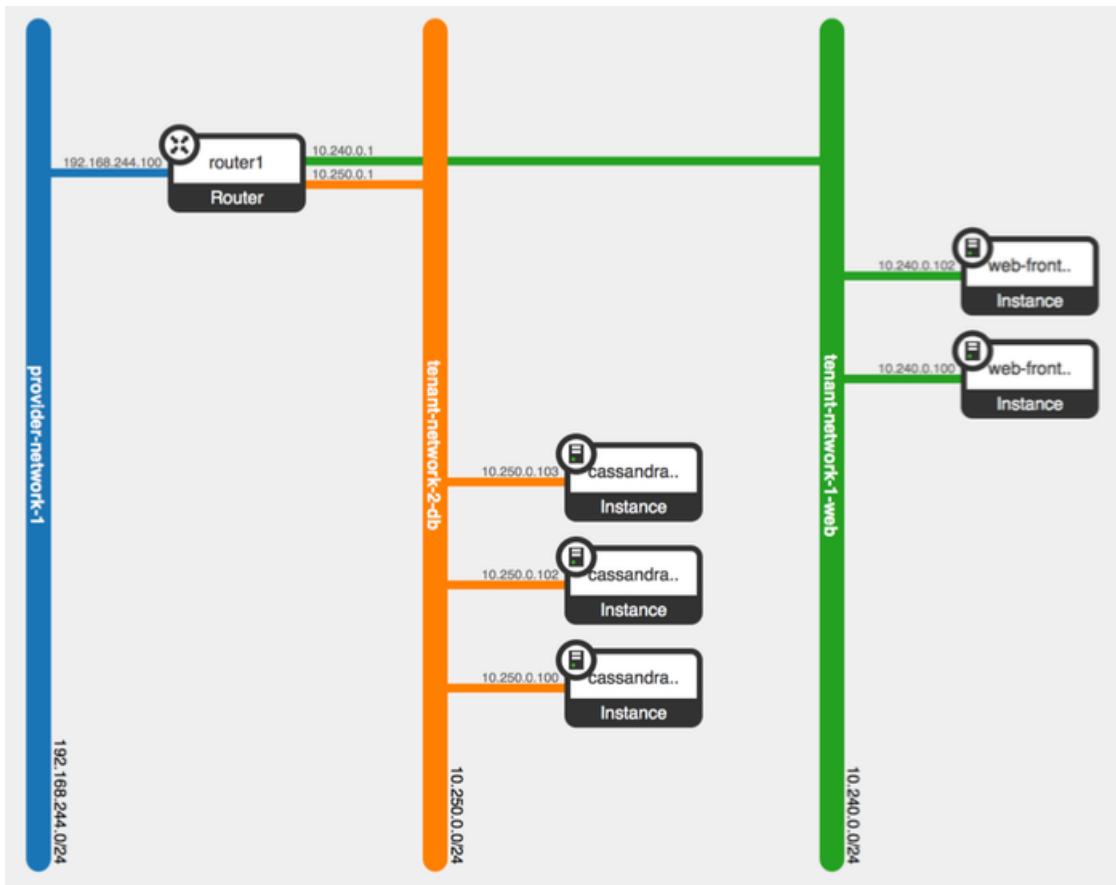
## Network Node

Le reti virtuali sono solitamente reti private che non possono interfacciarsi con l'esterno. Neutron permette alle VM di essere connesse con le reti esterne in modo che le VM stessa possano connettersi a internet e che possano essere anche raggiunte da internet. Per connettere la rete virtuale a internet è necessario che un **Network Node** sia istanziato, solitamente questo nodo coincide con il **Controller Node**, esso è connesso sia ad una rete esterna (rete pubblica) tramite internet sia alla rete locale che connette i nodi di OpenStack. Il Network Node fa girare un particolare agent chiamato **Neutron-L3-Agent** che è il responsabile dell'inoltro del traffico dalla/alla rete pubblica alla/dalla *Rete Virtuale*, questa funzionalità è raggiunta grazie alla creazione di *router virtuali* i quali implementano *funzionalità NAT e protocolli di inoltro del traffico*. Oltre alle componenti appena citate il Network Node ospita anche un secondo agent chiamato **Neutron-DHCP-Agent** che si occupa di gestire l'assegnamento della configurazione della rete alle VM.



## IP Assignment

Alle VM possono essere assegnati IP pubblici e il *Router Virtuale di frontiera* sarà il responsabile di implementare la *Network Address Translation*. La piattaforma permette anche di poter assegnare gli IP in maniera dinamica chiamati *floating IP addresses*.

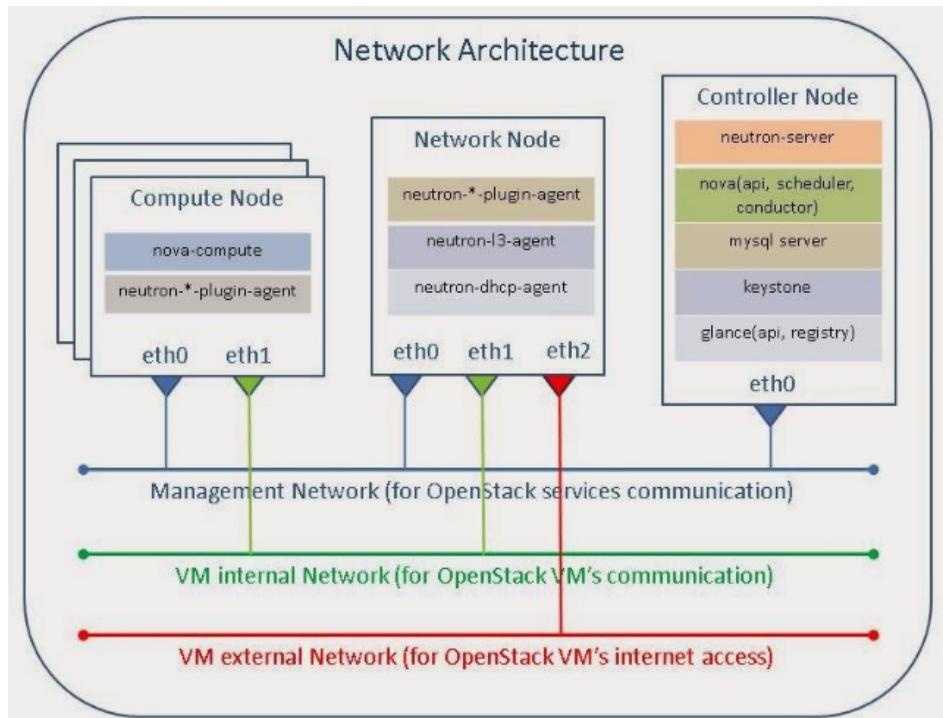


## Physical Network Infrastructure

L'infrastruttura della rete di OpenStack è generalmente configurata con 3 reti fisiche diverse:

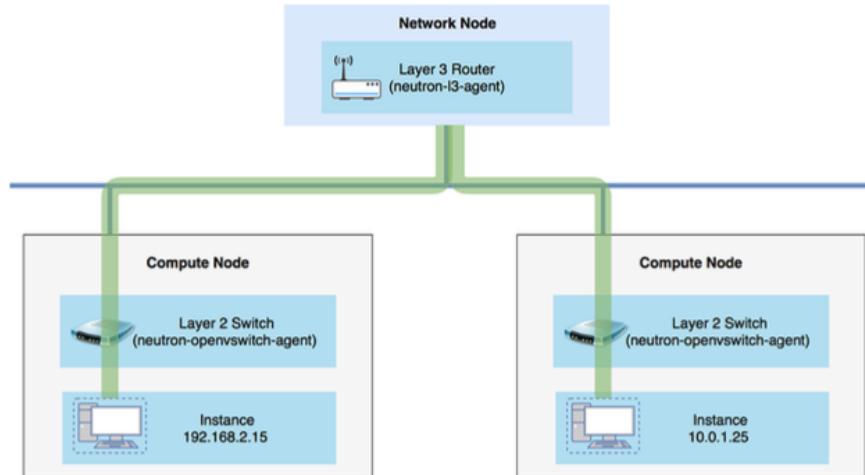
- *Management Network* usata per la comunicazione dei servizi di OpenStack.
- *VM internal network* usata per l'invio della comunicazione interna fra le differenti VM.
- *External network* usata per l'accesso a internet.

La *Management Network* e la *VM internal network* possono essere implementate sulla stessa LAN fisica, nel senso che il traffico proveniente dalle due reti virtuali può essere trasmesso sulla stessa LAN fisica. Solo il *Network Node* deve essere connesso all'*external network*, il controller node può essere connesso solo alla management network. In questo caso il controller node e il network node sono due entità distinte.

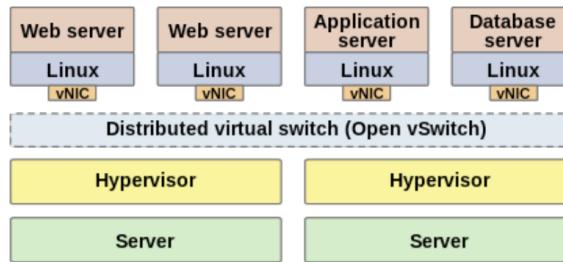


#### 4.2.10 OpenVSwitch (OVS)

Il *Neutron Agent* in esecuzione su ogni compute node è altamente configurabile, sono possibili differenti configurazioni che sfruttano differenti tecnologie per processare e inoltrare il traffico. Una delle ultime versioni sfrutta OpenVSwitch per implementare le funzionalità del network locale per processare e inoltrare il traffico tra i compute node e network node. OpenVSwitch permette una rapida riconfigurazione del comportamento di ogni agent seguendo direttive dal nova server.



**OpenVSwitch (OVS)** è un’implementazione open-source di uno switch virtuale di rete distribuito. E’ specificatamente progettato per fornire uno switching stack per ambienti di virtualizzazione hardware, può essere eseguito direttamente all’interno (o congiuntamente con) l’hypervisor per gestire il traffico da/verso le VM. Supporta l’implementazione di funzionalità di filtraggio, routing, switching e manipolazione dei pacchetti. OVS adotta un approccio centralizzato (SDN): ogni nodo ha la propria istanza OVS, controllata da un controller. Seguendo le direttive del controllore, tutte le istanza OVS implementano uno switch virtuale distribuito.

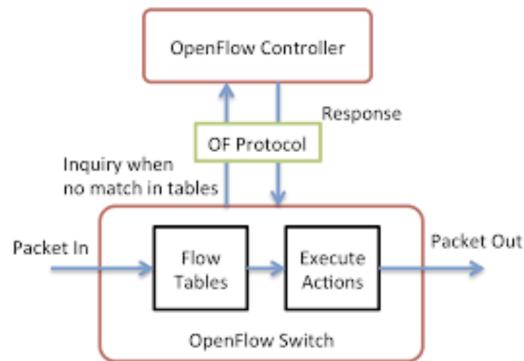


#### Software Defined Networking (SDN)

OVS adotta il **paradigma SDN**: le regole di rete sono configurate dinamicamente in ogni istanza, le direttive provengono dal controller centralizzato. Ogni nodo OVS ha una tabella (tabella di flusso) che descrive le caratteristiche dei pacchetti in entrata (ad es. IP sorgente, MAC des-

tinatario, etc.) e la corrispondente azione da eseguire (ad es. inoltra il pacchetto, modifica il pacchetto, etc.).

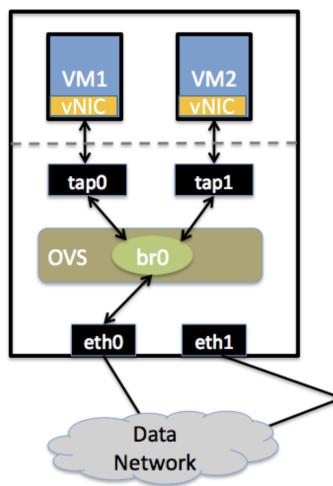
Quando un pacchetto viene ricevuto, la tabella viene acceduta, se il pacchetto fa match con una entrata della tabella allora viene fatta l'azione corrispondente. Se non c'è match con nessuna entry OVS manda un messaggio di richiesta al controller che risponderà con una nuova regola da aggiungere nella tabella locale oppure direttamente con un'azione da eseguire una sola volta. Il protocollo che implementa la comunicazione tra le istanze OVS e il controller si chiama **OpenFlow**.



## VM - OVS communication

Ogni VM ha una Virtual NIC, il cui hardware viene emulato dall'hypervisor. Per connettersi con l'istanza OVS del compute node, il vNIC è collegato ad un altro NIC virtuale creato nel sistema operativo host, ad es. tap0, tap1.

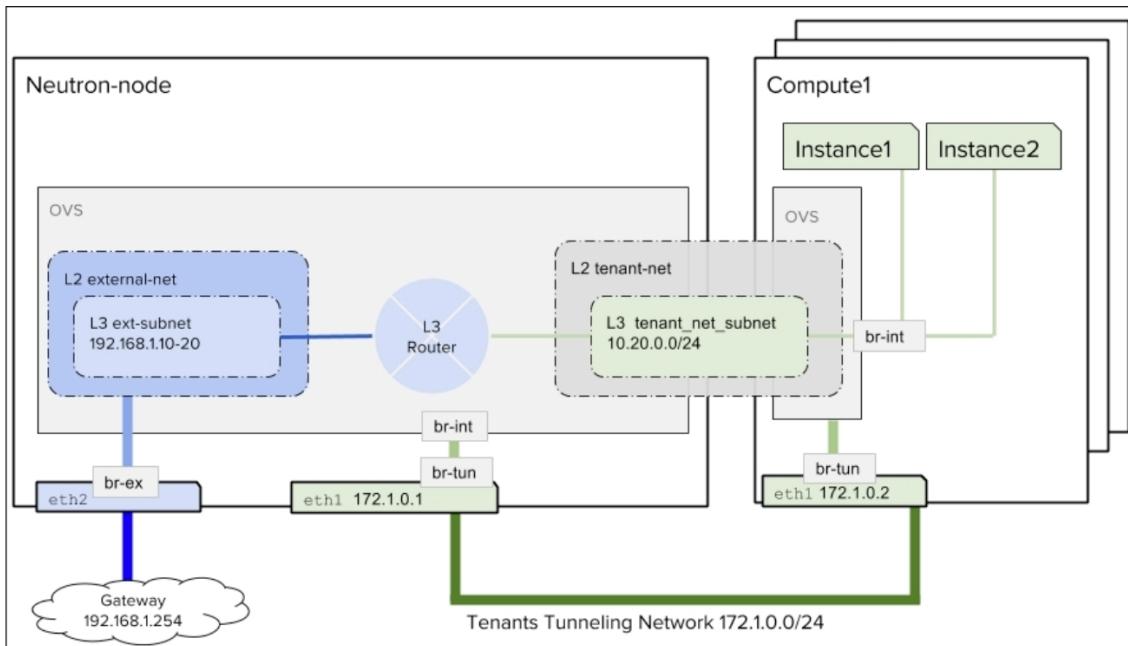
Un **tap** è un'interfaccia di rete virtuale che può essere collegata con un software, l'hypervisor in questo caso, per ricevere/inviare traffico di rete (frame di dati di livello 2) dal software. Il tap viene usato per ricevere/inviare traffico dati da/alla VM. Il traffico delle interfacce tap o dalle interfacce fisiche è gestito da OVS per implementare funzionalità di routing e forwarding.



## Network Node

Le istanze OVS possono essere programmate dal Neutron Server per inviare i dati su diverse VM in esecuzione sullo stesso compute node o su diversi compute node. Quando i dati sono indirizzati verso una rete esterna, invece, vengono inviati al network node.

Per implementare le funzionalità di instradamento L3 e l'invio dei dati, un'istanza OVS è installata anche sul nodo di rete. Tramite l'istanza OVS in esecuzione sul nodo di rete, vengono implementate funzionalità di routing virtuale L3 per instradare il traffico da/verso la rete esterna e le VM (o in generale i Virtual Networks locali).



### 4.2.11 Network virtualization

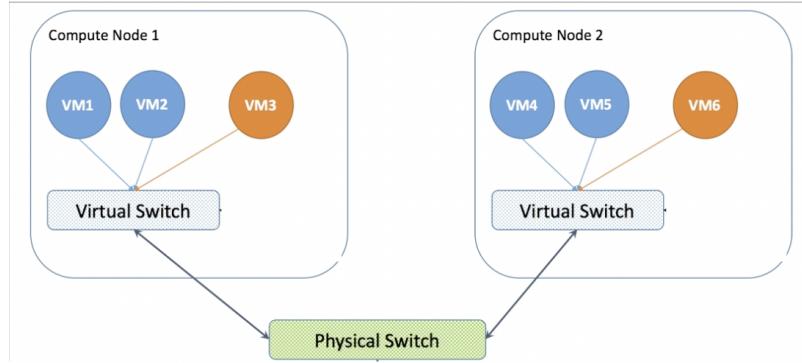
La virtualizzazione della rete è il processo tramite quale vengono create le Virtual Network (VNs) per le VMs sulla stessa infrastruttura fisica. Le VN sono reti Layer2, nello specifico reti Ethernet.

L'implementazione di Virtual Network richiede:

- Inoltro dati di layer 2 tra VM appartenenti allo stesso VN
- Isolamento tra VN, ad es. due VM appartenenti a VN differenti non possono comunicare, a meno che non ci sia un router che si connette al livello 3 i due VN.

In OpenStack, la virtualizzazione della rete richiede che il traffico venga inoltrato tra differenti compute node e da/verso il network node secondo l'esistenza di VN, ad es. se due compute node ospitano due differenti VMs appartenenti allo stesso VN il servizio di rete dovrebbe essere configurato per inoltrare il traffico tra di loro.

A questo scopo ogni pacchetto deve essere contrassegnato con l'ID della VN a cui appartiene in modo che possa essere configurato per inoltrare il traffico tra di loro. Sono adottate diverse tecnologie di rete esistenti: packet tagging o tunneling.



#### 4.2.12 Virtual LAN

VLAN è una delle più utilizzate per la virtualizzazione della rete. Lo standard di rete Ethenet IEEE 802.1Q è stato definito specificatamente per permettere la creazione di Virtual LANs sopra una normale rete Ethernet. A questo scopo, lo standard aggiunge un nuovo campo che è l'ID della VLAN (VID) a cui appartiene.

Uno switch (virtuale o fisico) che supporta IEEE 802.1Q è responsabile di aggiungere il campo VID quando un pacchetto viene inviato sulla rete, di rimuoverlo quando arriva a destinazione ed infine per la distribuzione secondo il VID. Ad esempio, i frame broadcast devono essere distribuiti solo agli host appartenenti allo stesso VID, per garantire l'isolamento.

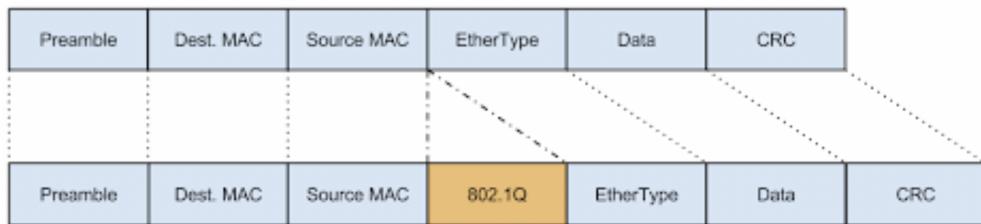


Figure 4.14: Ethernet packet.

#### Neutron VLAN Operations

Neutron può essere configurato usando VLAN per creare VNs. Quando una VN viene creata per connettere due o più VMs, un VID viene assegnato alla VN dal Nova-Server.

Quest'ultimo configura OVS dei compute node coinvolti di conseguenza, al fine di etichettare il traffico proveniente da una VM con il VID assegnato al suo VN e di inoltrare i pacchetti a tutti i compute node coinvolti. Quando un pacchetto viene ricevuto da un compute node, il VID viene rimosso e il pacchetto viene inoltrato a tutte le VMs coinvolte.

In caso in cui la VN è connessa ad internet tramite un Virtual Router, il Neutron-Server configura anche l'OVS in esecuzione sul Network Node per etichettare i pacchetti in arrivo da/diretti al Router assegnato alla VN.

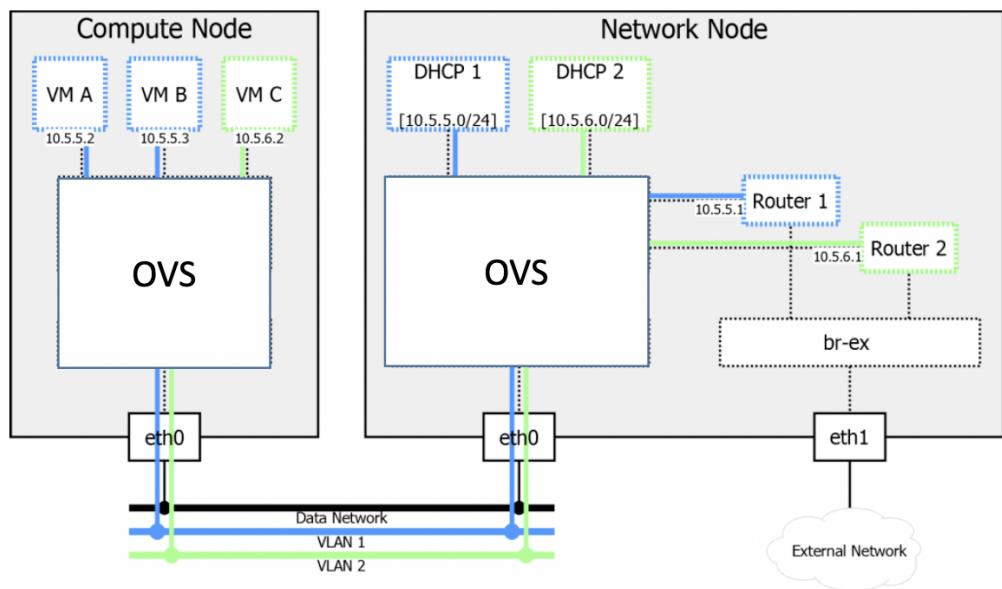


Figure 4.15: VLAN used between compute and network node.

#### 4.2.13 GRE Tunneling

Un altro meccanismo di virtualizzazione adottato è il **Generic Routing Encapsulation (GRE)**, un protocollo di tunneling che permette di incapsulare pacchetti IP dentro altri. Il protocollo viene usato per creare un tunnel tra due host di internet, inoltre aggiunge un campo addizionale **GRE header** per specificare il tipo del protocollo di incapsulamento.

GRE può essere usato come meccanismo per creare Virtual Networks: un GRE tunnel è creato per ogni coppia di compute node (e tra il compute e network node) che eseguono VMs appartenenti ad una specifica rete virtuale. Differenti ID di GRE tunnel sono usati per ogni VN.

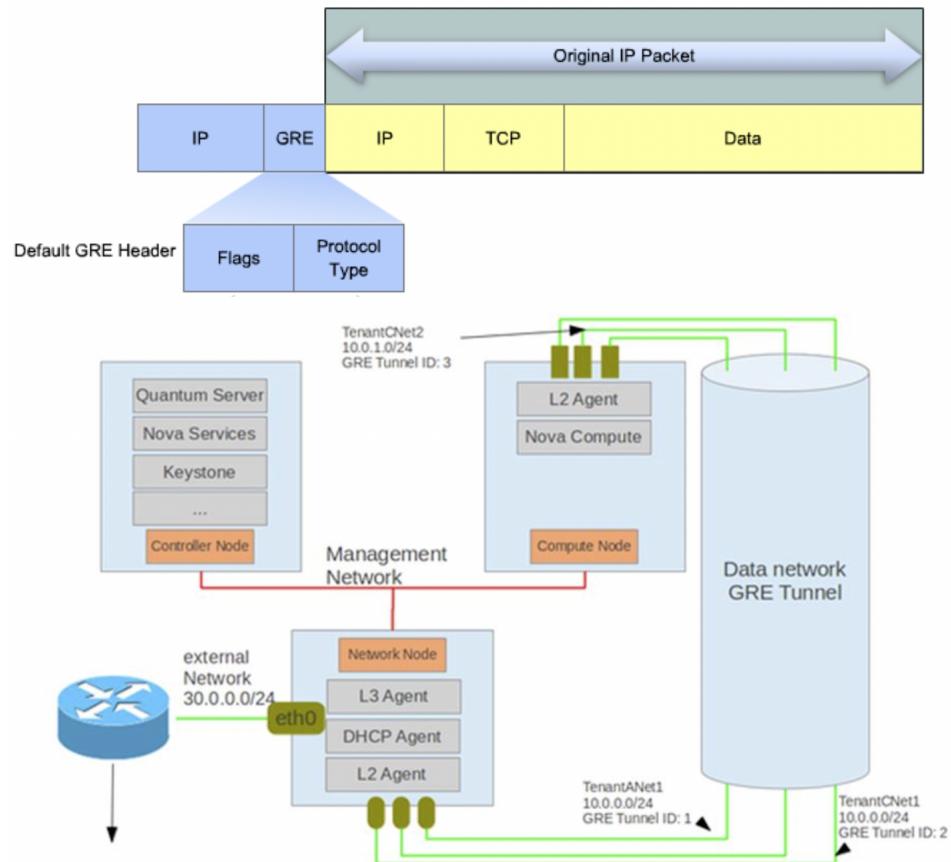


Figure 4.16: GRE header added and GRE Tunnel Example.

## Neutron GRE Operations

GRE può essere usato per creare VNs. Per ogni VN viene istanziato un **GRE ID** dal Nova-Server. Quest'ultimo istruisce le istanze OVS per creare un adattatore di rete virtuale (*gre-1*) sopra l'interfaccia fisica.

Le interfacce GRE sono responsabili di creare i tunnel tra i compute node coinvolti (nodi che hanno almeno una VM che appartiene alla VN, oppure compute node e server di rete). e encapsulare/decapsulare il traffico.

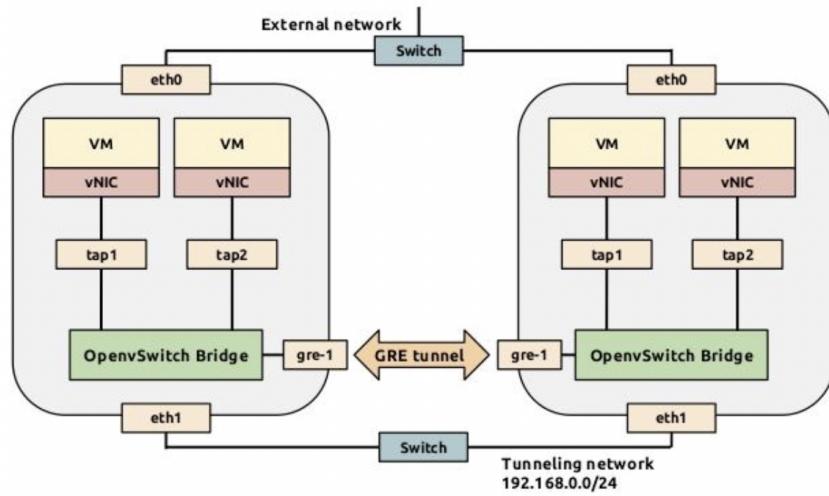


Figure 4.17: GRE tunnel example between two OpenSwitch Bridge.

### 4.2.14 VXLAN

VLAN ha un limite, i suoi identificatori sono lunghi 12 bit, di conseguenza il massimo numero di VLAN è 4094. In grandi implementazioni cloud questo numero, anche se alto, potrebbe essere un problema.

Virtual Extensible LAN (VXLAN) è una tecnica di virtualizzazione di rete che tenta di risolvere problemi di scalabilità associati alle grandi distribuzioni di cloud computing al fine di rimuovere questo limite.

Con VXLAN si possono avere fino a 16 milioni di VN diversi. VXLAN è ancora una metodologia di encapsulamento dei pacchetti tra due endpoint, chiamati endpoint del tunnel virtuale (VTE) in questo caso. I frame sono encapsulati all'interno dei pacchetti UDP, mentre a livello IP sono specificati la sorgente e la destinazione dei VTE. È inclusa un'intestazione VXLAN a segnalare l'ID della LAN virtuale in quanto due VTE possono supportare più VLAN contemporaneamente.

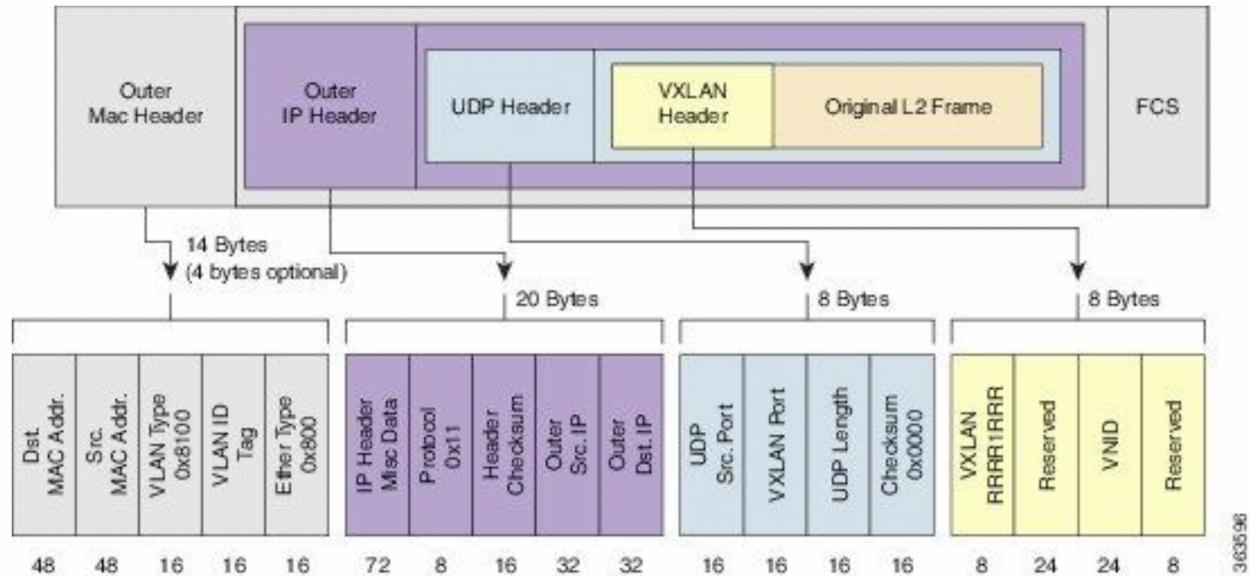


Figure 4.18: VXLAN Packet.

## Neutron VXLAN Operations

I tunnel VXLAN sono sfruttati da Neutron nello stesso modo dei tunnel GRE. Per ogni VN viene istanziato un nuovo ID VXLAN dal Nova Server. Quest'ultimo incarica le istanze OVS di incapsulare i pacchetti in tunnel VXLANs secondo l'istanza VNs.

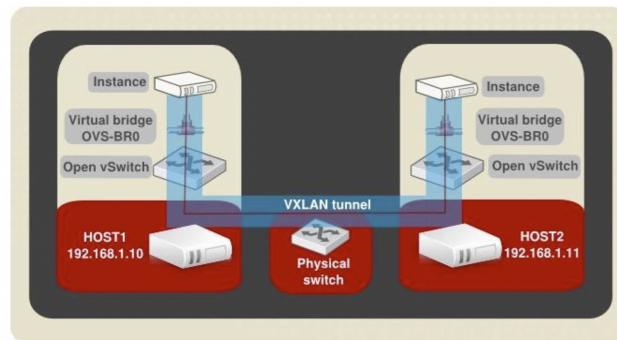


Figure 4.19: VXLAN Packet.

#### 4.2.15 Cinder

Cinder è il componente responsabile della gestione dei **volumi**, ogni VM ha un disco rigido virtuale predefinito che memorizza il sistema operativo. Se una VM richiede storage aggiuntivo, è possibile creare e attaccare volumi aggiuntivi dinamicamente a un'istanza. Tali volumi sono visti come dischi rigidi virtuali dalla VM. Alla fine, un disco rigido virtuale viene memorizzato nel file system come un'immagine (un file o un oggetto).

Cinder è responsabile della gestione di tali immagini e della loro esposizione alle VM. Un disco rigido virtuale è esposto dall'hypervisor, che accede al disco rigido virtuale reale tramite il protocollo iSCSI. **iSCSI** è uno standard di rete di archiviazione basato su protocollo Internet che fornisce accesso a livello blocco allo storage. Cinder memorizza le immagini del volume nel file system locale del nodo su cui è installato il servizio o in un file system cloud.

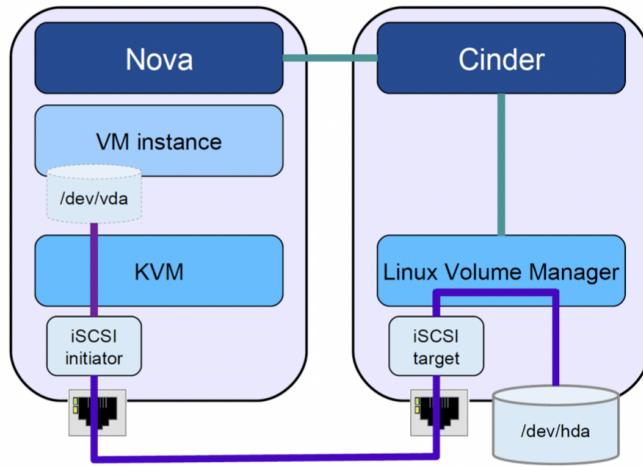


Figure 4.20: VXLAN Packet.

#### Cinder Architecture

Cinder comprende i seguenti componenti:

- **Cinder-API**: che espone un'interfaccia di riposo ai clienti per controllare le operazioni di cenere (ad es. Nova o direttamente l'utente finale).
- **Cinder-Volume**: che è responsabile per la gestione diretta delle richieste dalla interfaccia di riposo.
- **Cinder-Scheduler**: che è responsabile per la selezione dello storage corretto per memorizzare nuove unità (se più archivi sono configurati sul sistema).
- **Cinder-Backup**: che è responsabile per la creazione di backup dei volumi esistenti quando richiesto dagli utenti.

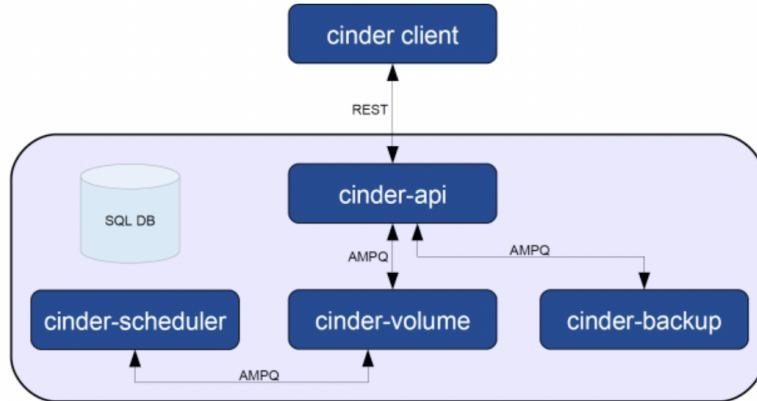


Figure 4.21: Cinder Architecture.

#### 4.2.16 Ceilometer

Ceilometer è la componente di telemetria. Controlla tutti i componenti dell'istanza, **misurando la risorsa utilizzata** da ogni utente. I dati raccolti da Ceilometer possono essere utilizzati a fini di **fatturazione**. Ceilometer raccoglie anche statistiche di telemetria che possono essere utilizzate per **verificare lo stato del sistema**.



Figure 4.22: Ceilometer dashboard.

## Ceilometer - Atchitecture

Ceilometer ha un **Ceilometer-Collector** centralizzato che è responsabile della ricezione di tutti i dati da tutti i componenti Openstack e di memorizzarli in un DB (di solito un DB Nosql come Mongo DB). Al fine di raccogliere dati da tutti i nodi di elaborazione, un **Ceilometer-Agent** è installato su ogni nodo. L'agente è responsabile della raccolta di tutte le notifiche da tutti i componenti locali e li trasmette al collettore. Alla fine il Collector espone una serie di API REST per recuperare i dati dal database.

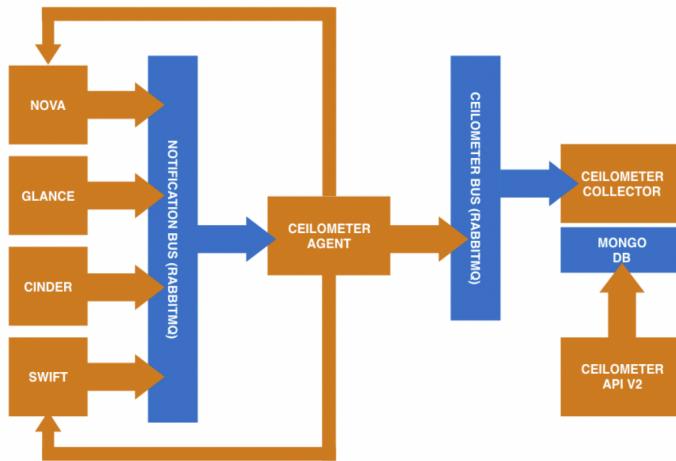


Figure 4.23: Ceilometer architecture.

#### 4.2.17 Horizon

Le funzionalità di Openstack sono esposte agli utenti attraverso un'**interfaccia web**. La dashboard è di solito esposta dal controller e consente la gestione di tutti gli aspetti delle istanze. Un insieme di strumenti a linea di comando sono inclusi anche per la gestione del backend.

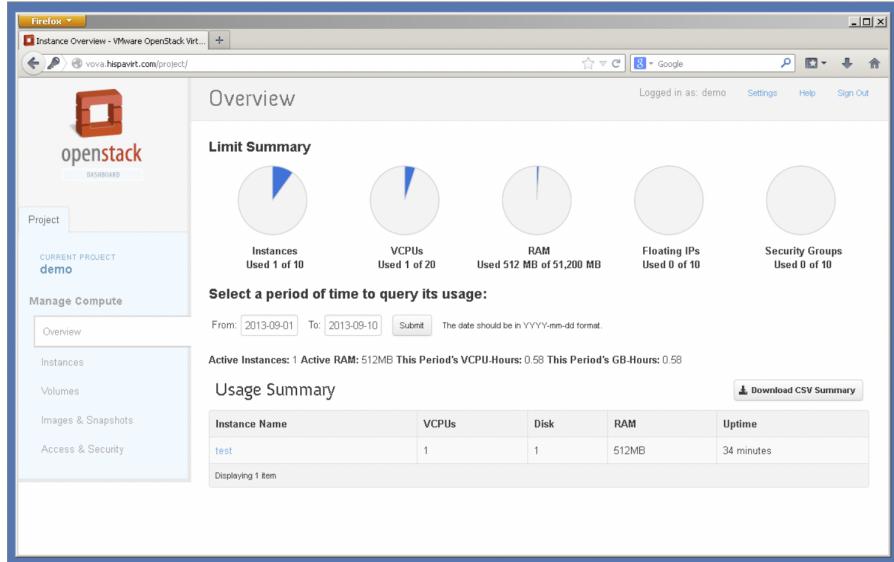
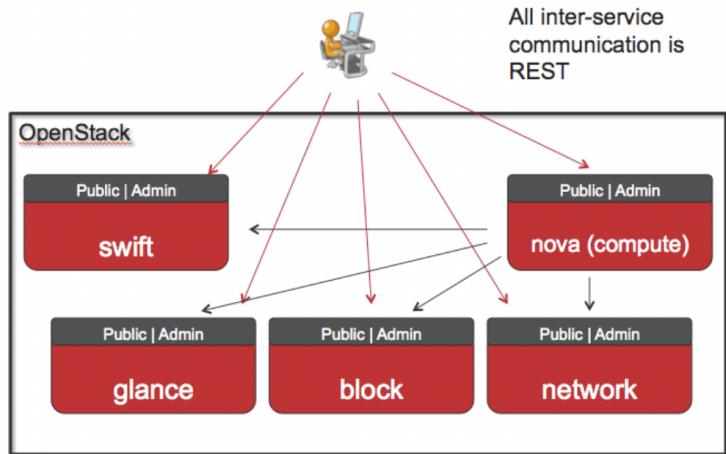


Figure 4.24: Horizon web interface.

#### 4.2.18 Service APIs

Ogni servizio Openstack **espone un insieme di API**. Tutte le API di comunicazione è **REST**. Le API sono esposte da ogni servizio per l'interazione inter-service e per esporre un insieme di funzionalità agli utenti. Le API possono essere sfruttate dagli utenti per incorporare il processo di automazione in applicazioni esterne.

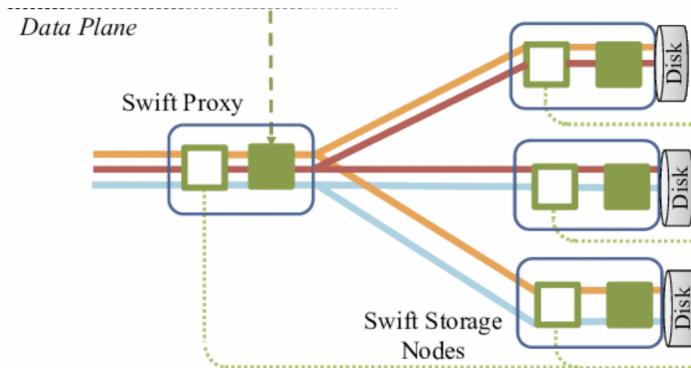


#### 4.2.19 Swift

Swift è il servizio di archiviazione di oggetti disponibile in Openstack, consente agli utenti e ad altri moduli Openstack di memorizzare dati (rappresentati come oggetti) nella piattaforma cloud.

Altri servizi, come ad esempio Cinder o Glance, possono utilizzare Swift per memorizzare i backup dei volumi o le immagini, rispettivamente. Il componente è responsabile della gestione di tutti gli aspetti della memorizzazione dei dati, dalla ricezione e memorizzazione dei dati al loro recupero. I dati vengono solitamente memorizzati in un archivio cloud, per garantire la scalabilità (anche se è consentito l'archiviazione locale nel server in cui è installato il servizio). Swift è composto da due componenti:

- **Swift-Proxy**: che è responsabile di esporre l'interfaccia REST e gestire la richiesta (per memorizzare un nuovo oggetto o per recuperare un oggetto).
- **Swift-Storage-Node**: che è installato su ogni nodo di archiviazione (un nodo che ospita alcuni dei dati) per memorizzare effettivamente i dati su un'unità fisica.



#### 4.2.20 Heat

**Heat** è l'orchestratore di OpenStack, esso può gestire la creazione/distruzione delle VMs oppure può gestire automaticamente le loro impostazioni.

Questa automazione si basa su un set di regole che specificano quando si verificano delle certe condizioni per compiere azioni di conseguenza sulla VM. Tramite Heat, ad esempio, si può creare un servizio di autoscaling che sfrutta lo stato dei dati di Ceilometer per creare/distruuggere VMs.

Heat è composto da:

- **Heat-API**: esponde l'interfaccia all'utente per configurare l'orchestratore.
- **Heat-Engine**: è responsabile della gestione delle richieste dell'utente e della relativa implementazione.

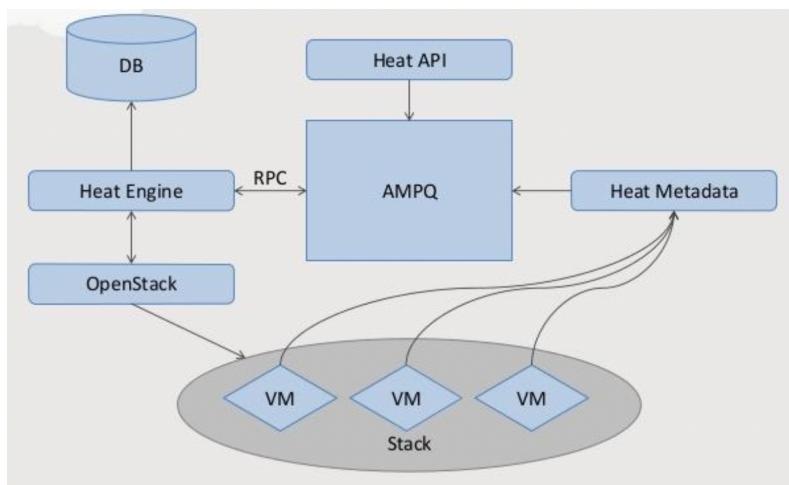
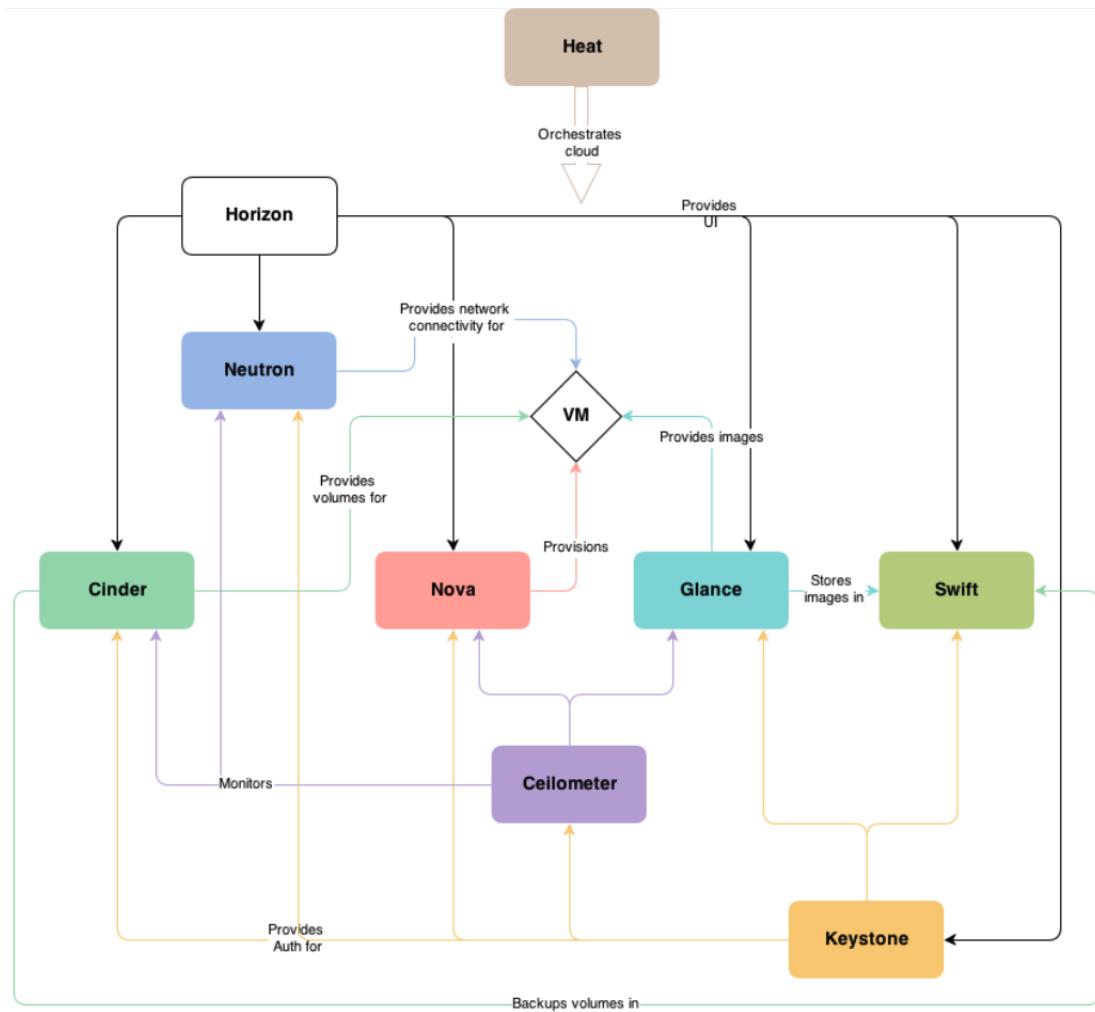


Figure 4.25: Heat.

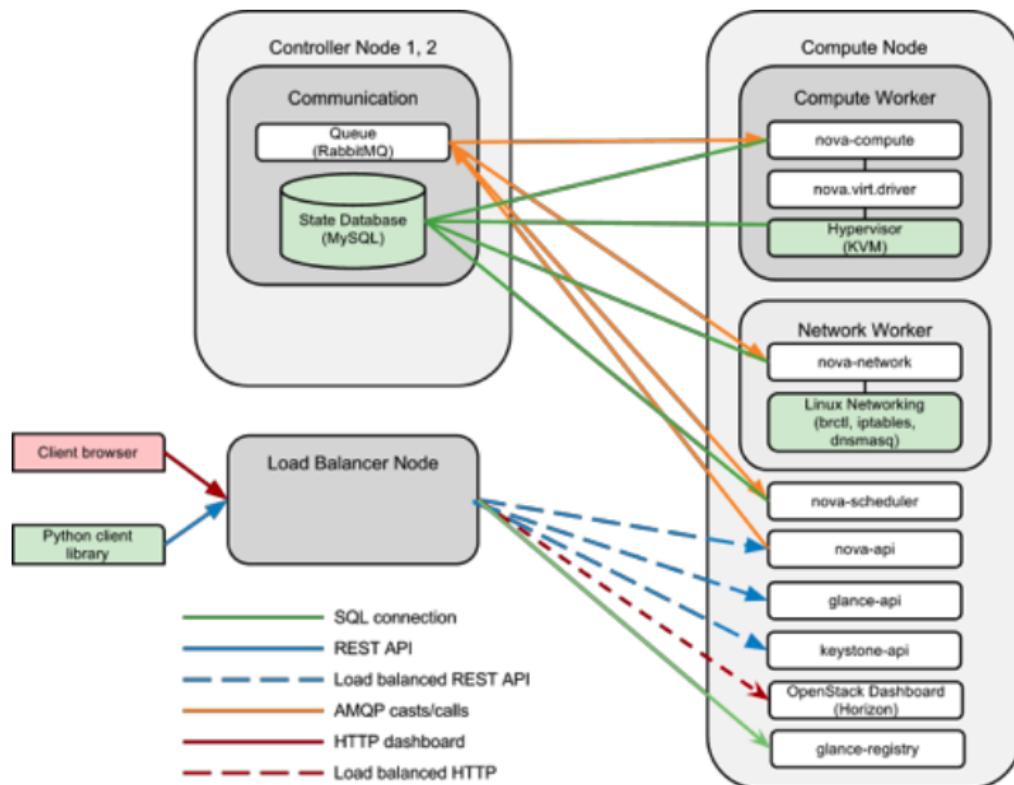
## 4.3 OpenStack Service Interactions Summary



#### 4.3.1 High Availability (HA)

L'architettura attuale include solo un'istanza dei componenti di gestione. OpenStack può essere installato nella cosiddetta configurazione ad alta disponibilità, ad es. più istanze di ogni servizio possono essere distribuite al fine di garantire elevata disponibilità e resilienza. Questo può essere eseguito in modo simile alla distribuzione in modo ridondante delle applicazioni cloud.

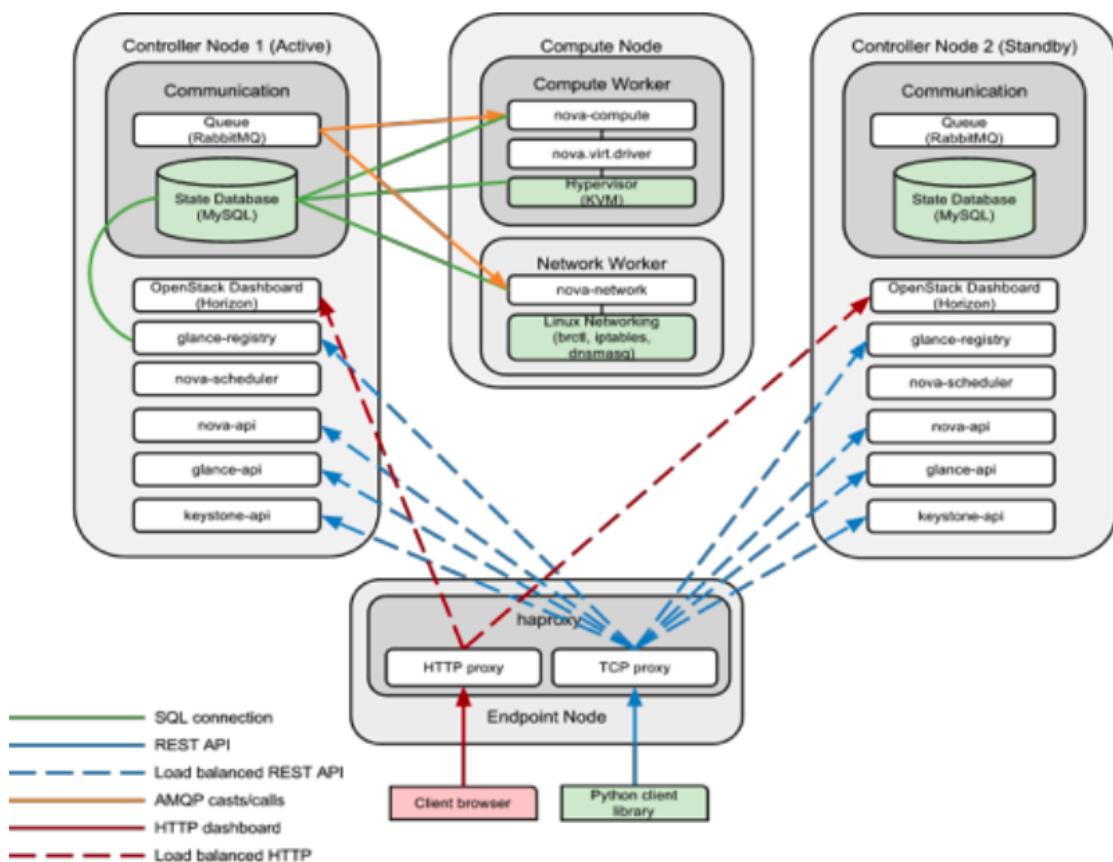
In una configurazione ad alta disponibilità, istanze multiple del controller node sono distribuite con tutti i servizi corrispondenti per controllare lo stesso set di compute node.



#### 4.3.2 HA with dedicated node

Le API OpenStack complessive sono non sono esposte direttamente dal nodo controller ma attraverso un proxy HA.

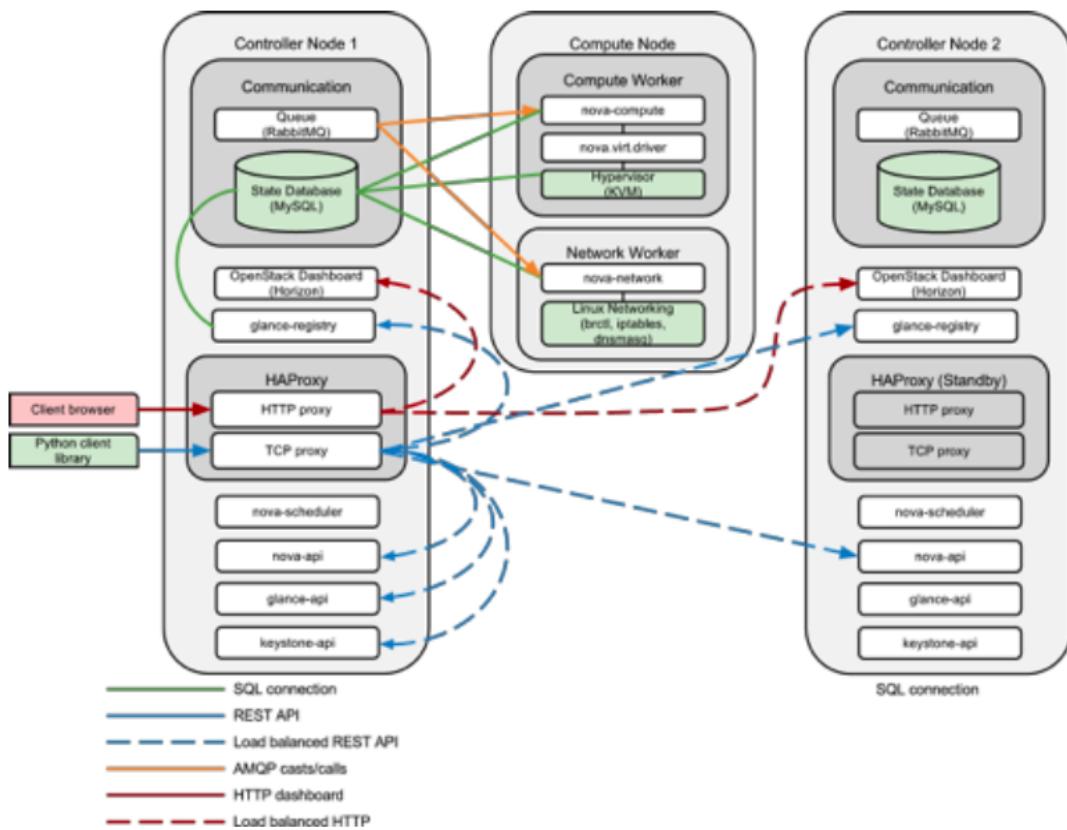
Il **proxy HA** è responsabile della ricezione delle richieste e del dispaccimento di esse ad uno dei controller node. Il proxy deve implementare funzionalità per rilevare quando si verificano guasti al controller node per rimuoverlo e metterne un'altro (dalla lista di quelli attivi) in esecuzione. Lo stato di archiviazione deve essere replicato attraverso il controller node.



### 4.3.3 HA with simple redundancy

Le API OpenStack complessive sono esposte direttamente da un controller, selezionato come master controller. Le funzionalità del proxy HA sono implementate su tutti i controller, uno di questi è segnato come attivo e serve le richieste, mentre gli altri sono contrassegnati come inattivi.

Le istanze proxy HA devono coordinarsi per rilevare i guasti e avviare la riconfigurazione quando necessario. Di nuovo, lo stato di archiviazione deve essere replicato attraverso i nodi del controller.



## 4.4 Lightweight cloud computing platform for DevOps

### 4.4.1 Software development - BACK

Anni fa la maggior parte dei software era un grande monolita, che eseguiva un singolo processo o un piccolo numero di processi. Questo tipo di applicazioni hanno un lento ciclo per essere rilasciate e vengono aggiornate poco frequentemente dato che lo sviluppo è completato e poco pratico: i dev tirano su l'interno sistema che poi viene consegnato all'ops team, il team di amministratore del sistema gestisce invece l'infrastruttura (I server).

Cambiamenti su una parte dell'applicazione richiedono un la ridistribuzione dell'intera app. Nel tempo la mancanza di confini tra i componenti software aumenta la complessità dello sviluppo e mantenimento, che porta ad una deteriorazione della qualità dell'intera applicazione. Le app monolitiche richiedono spesso un piccolo numero di potenti server.

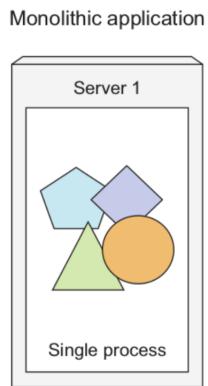


Figure 4.26: Example of monolithic application.

### Legacy System

Questi sistemi sono ancora attivi ed in esecuzione (anche se lentamente vengono rimpiazzati). Se conveniente vengono messi sul cloud, ciò significa che i server fisici vengono rimpiazzati dalle VM.

Questo processo semplifica il mantenimento dell'infrastruttura, il processo di sviluppo e mantenimento del software è però ancora complesso. Quando l'applicazione ha un carico che aumenta vengono aggiunte delle risorse ai server (scalabilità verticale) senza modificare l'architettura originale dell'applicazione. Ci sono però dei limiti e non è sempre fattibile farlo.

### 4.4.2 Software development - NOW

Il cloud computing ha radicalmente cambiato il modo di sviluppare e distribuire le applicazioni. Le grandi monolithic legacy applications sono ora **divise in componenti indipendenti** chiamati servizi o microservizi, che possono essere sviluppate, distribuite, aggiornate e scalate in modo indipendente. Ogni servizio/microservizio viene eseguito come un processo indipendente e comunica con gli altri tramite delle interfacce predefinite.

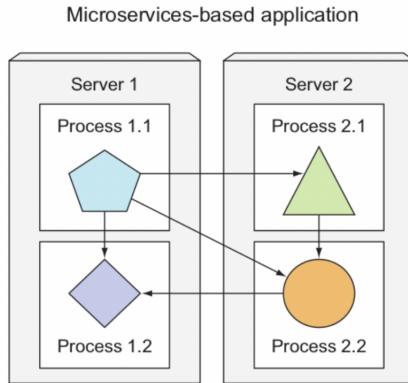


Figure 4.27: Example of Microservice-based application.

### Service independency

Questa struttura permette un lifecycle dell'applicazione più rapido: i componenti possono cambiare rapidamente e facilmente, spesso per necessità dovute ad un rapido cambiamento del business.

I servizi comunicano tramite un protocollo sincrono (es. SOAP, HTTP/RESTful) o in modo asincrono tramite lo scambio di messaggi (es. Coda di messaggi), questi protocolli sono semplici e conosciuti dai dev.

I servizi possono essere sviluppati separatamente, un cambiamento in uno non richiede cambiamenti o la ridistribuzione di altri microservizi ( purchè non cambino le interfacce).

### 4.4.3 Application development process: DevOps, NoOps

Questa nuova architettura ha cambiato anche l'intero processo di sviluppo, in particolare, come le applicazioni sono gestite nella produzione.

Nel passato il lifecycle dell'applicazione era gestito da due team (spesso separati):

- **Development team (dev team):** avevano il compito di progettare e programmare il software dell'app
- **Operations team (ops team):** dovevano prendersi cura di distribuire e mantenere l'app.

*Ora le aziende adottano un approccio differente: lo stesso team che sviluppa l'applicazione prende parte anche nella distribuzione e mantenimento per l'intera vita dell'applicazione. Questo approccio è chiamato DevOps.*

### DevOps

Il vantaggio principale di questo approccio è quello di avere i dev più coinvolti nella produzione dell'applicazione. Questo porta ad avere una migliore comprensione delle necessità dell'utente e dei problemi che possono accadere nella distribuzione dell'app.

Il processo di distribuzione è snello, questo significa che nuove release possono essere fatte più spesso: idealmente ci possono essere dev che distribuiscono la loro applicazione senza

coinvolgere l'ops team (*continuous development*). Questa pratica richiede che i dev abbino una profonda conoscenza dei dettagli dei livelli sottostanti dell'infrastruttura, questo non è sempre fattibili o desiderato.

### NoOps

I dev e gli amministratori di sistema lavorano per raggiungere lo stesso obiettivo, solitamente hanno background, esperienza e motivazioni differenti. Il modello DevOps viene preferito per i suoi vantaggi, ma la sua implementazione è difficile, ad esempio membri del dev team difficilmente hanno esperienza nel prendersi cura dell'infrastruttura.

Una soluzione ideale sarebbe quella di mettere in pratica un approccio NoOps, nel quale i dev sviluppano l'applicazione in autonomia senza conoscere niente dell'infrastruttura e senza preoccuparsi dei dettagli della distribuzione dell'applicazione.

*Una soluzione per consentire l'implementazione e la manutenzione automatica del servizio renderebbe fattibile l'implementazione di un approccio DevOps o NoOps.* Si avrebbe ancora bisogno di avere un team ops per gestire l'infrastruttura, ma gli sviluppatori sarebbero autorizzati a distribuire il loro software facilmente

#### 4.4.4 Service deployment and management

Indipendentemente dalle conoscenze richieste per l'implementazione e la gestione delle applicazioni, un'architettura basata su servizi presenta altri svantaggi. Quando un'applicazione consiste di un piccolo numero di componenti, la loro gestione è semplice; quando il numero di componenti aumenta, aumenta la complessità delle operazioni di distribuzione.

La configurazione manuale e la distribuzione sono noiose, soggette ad errori e irrealizzabili quando il numero di servizi raggiunge un certo numero.

*È necessaria una soluzione per automatizzare la distribuzione e la configurazione dei componenti per gestire grandi applicazioni con un gran numero di servizi.*

#### 4.4.5 Service dependencies

Come accennato, i servizi sono sviluppati come componenti indipendenti da diversi team. A causa della loro indipendenza è comune avere diverse squadre che utilizzano diverse librerie. Ogni squadra seleziona la propria libreria, quando la libreria è la stessa, potrebbe accadere che diverse versioni siano selezionate

Oltre a questo, diverse squadre vogliono avere la libertà di sostituire la libreria ogni volta che si presenta la necessità implementare diversi servizi/applicazioni con dipendenze incrociate può essere un incubo per il team ops.

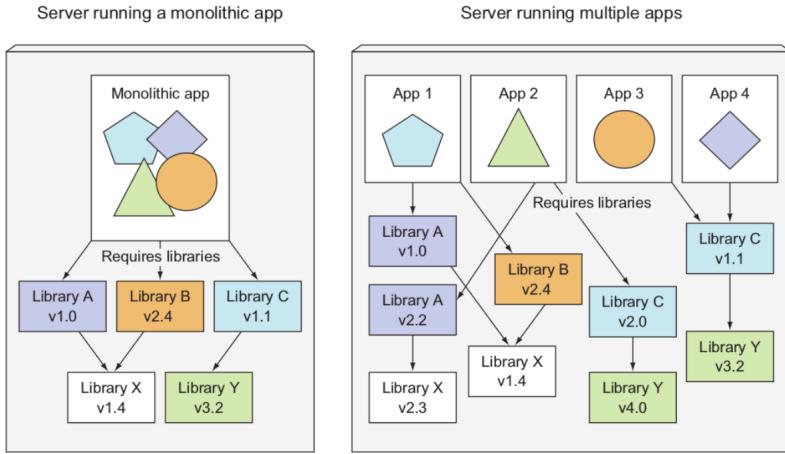


Figure 4.28: Example of library dependencies.

## Consistent Application Environment

Indipendentemente dal numero di componenti uno dei più grandi problemi che i dev e ops team devono gestire sono le differenze nell'ambiente in cui le applicazioni vengono eseguite.

*Un metodo per semplificare tali operazioni è quello di distribuire applicazioni in un ambiente confinato ideale in cui tutte le librerie sono pre-installate e l'hardware reale su cui il software è in esecuzione è in qualche modo astratto.*

Questo ambiente può essere creato tramite una full virtualization, ad esempio facendo girare il software in una VM. Comunque questa soluzione è costosa in termini di overhead (virtualizzazione, guestOS, etc). In aggiunta, usando macchine virtuali c'è overhead anche in termini di configurazione e installazione delle librerie della VM e del guestOS.

### 4.4.6 Lightweight Virtualization

Il meccanismo di lightweight virtualization, come i container, potrebbe essere una soluzione all'alto overhead richiesto dalla full virtualization. I container possono essere usati per preparare a virtualizzare un ambiente nel quale librerie e dipendenze sono preinstallate e configurate per eseguire una certa applicazione o servizio, che è parte di un'applicazione. In aggiunta i container possono essere inviati e installati su macchine diverse usando un sistema di distribuzione ad hoc che si preoccupa di distribuire e installare le immagini su macchine differenti, un esempio è **Docker**.

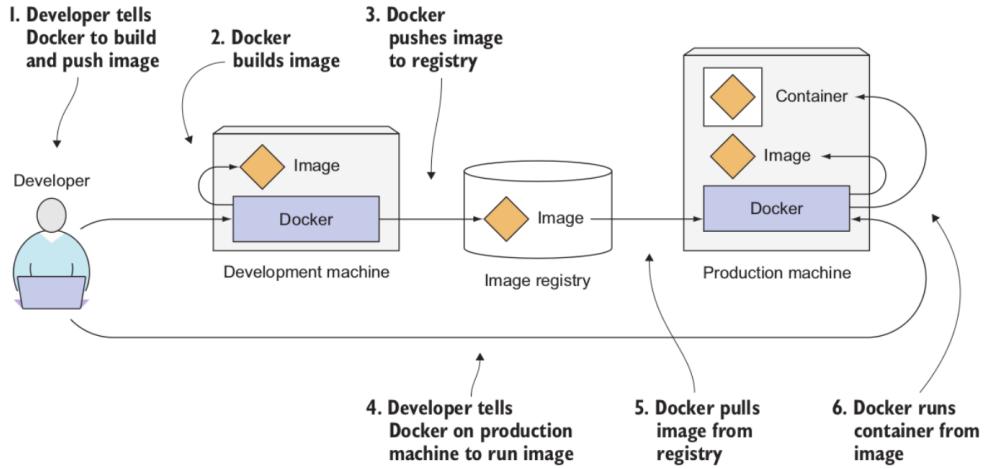


Figure 4.29: How Docker works.

#### 4.4.7 Kubernetes

Kubernetes è un sistema software sviluppato da google che permette di distribuire e gestire facilmente applicazioni containerizzate su di esso. Un'applicazione containerizzata è un'applicazione i cui componenti (servizi indipendenti) sono distribuiti e spediti come container diversi.

Kubernetes mira a semplificare completamente il processo complessivo di distribuzione e gestione dei container che compongono l'applicazione, occupandosi di aspetti riguardanti scalabilità, tolleranza al fallimento, gestione delle risorse, ecc. L'obiettivo è quello di offrire **un'infrastruttura IaaS leggera per distribuire applicazioni in modo semplice ed efficace**, in modo che gli sviluppatori possano prendersi cura di esso senza dover affrontare molti aspetti.

Il team operativo è ancora necessario, in questo caso, tuttavia, è responsabile per mantenere ed eseguire l'infrastruttura Kubernetes.

#### Kubernetes core operation

Kubernetes è una piattaforma distribuita composta da un nodo **master** e un numero variabile di nodi **worker**. I nodi di kubernetes possono essere installati direttamente su server fisici oppure su una piattaforma cloud sopra una VM.

Il nodo *master* espone un interfaccia ai dev che possono richiedere di eseguire una specifica applicazione. Lo sviluppatore deve presentare un descrittore app che include solo l'elenco dei componenti che compongono l'applicazione, per ogni componente viene specificata la configurazione. Il master di Kubernetes si occupa di distribuire i componenti delle app sui nodi di lavoro in base alla descrizione fornita dallo sviluppatore in modo completamente automatico.

Attraverso l'interfaccia lo sviluppatore può specificare che alcuni servizi devono funzionare insieme, di conseguenza sono sviluppati sullo stesso nodo *worker*. Altri saranno distribuiti intorno al cluster in base allo stato di ciascun nodo *worker*.

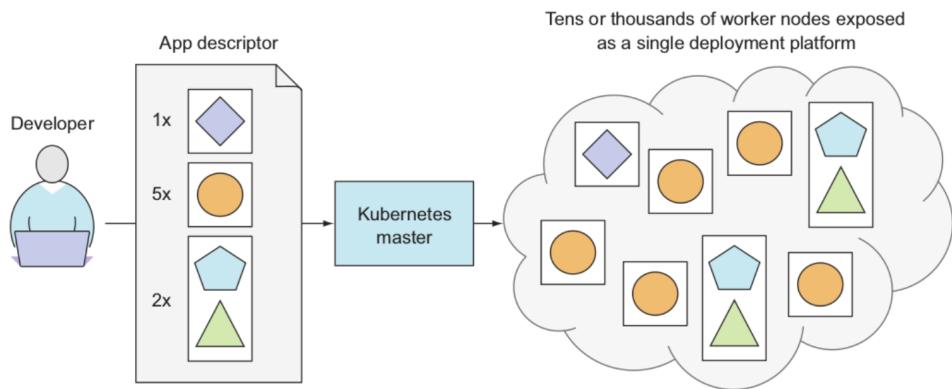


Figure 4.30: App descriptor.

Oltre alla configurazione di ogni componente lo sviluppatore deve specificare il numero di istanze desiderato per garantire la scalabilità. La piattaforma si occuperà di replicare ogni componente e bilanciare il traffico tra di loro al fine di garantire la scalabilità. La piattaforma si occupa di monitorare lo stato di ogni componente e gestire i guasti: nel caso in cui una istanza fallisca (o fallisca il nodo di lavoro su cui viene eseguita) la piattaforma si occupa di sostituire l'istanza con una nuova.

## Advantages

Kubernetes solleva i dev di applicazioni dalla necessità di implementare le funzionalità relative all'infrastruttura nelle loro applicazioni.

La piattaforma si occupa di implementare funzionalità come ridimensionamento, bilanciamento del carico, auto-healing (ripresa dai guasti). Di conseguenza i dev di applicazioni possono concentrarsi sullo sviluppo di applicazioni senza perdere tempo a capire come integrare l'applicazione all'interno dell'infrastruttura.

I compiti della squadra operativa sono semplificati: sono sollevati dal prendersi cura di distribuire i componenti, configurarli e risolvere i guasti, tutte queste funzionalità sono gestite dalla piattaforma stessa.

In aggiunta a questo Kubernetes può ottenere un migliore utilizzo delle risorse disponibili nell'infrastruttura

#### 4.4.8 Cluster Architecture

Il nodo master implementa il **pannello di controllo**, è responsabile del controllo del cluster. Ogni componente del pannello di controllo può essere installato su un singolo master node, può essere spartito tra nodi multipli oppure può essere replicato per garantire alta disponibilità.

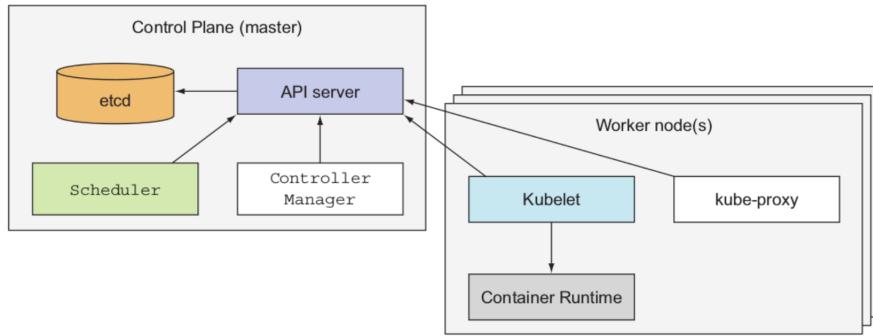


Figure 4.31: Control plane.

Il **pannello di controllo** ha i seguenti componenti:

- **API server**: espone un interfaccia per i dev
- **Scheduler**: schedula le app
- **Controller manager**: Compie funzioni a livello di cluster come replica di componenti e gestione dei fallimenti dei nodi
- **Etdc**: un affidabile e distribuito data store

I nodi **worker** invece includono i seguenti componenti:

- **Container runtime**: un ambiente (come Docker o altri) che si prende cura dell'esecuzione dei container
- **Kublet**: che comunica con l'API server e gestisce i container sul nodo
- **Kubernetes Service Proxy**: si prende cura di bilanciare il traffico sulla rete tra i componenti delle applicazioni e l'inoltro delle richieste, mentre i container migrano tra i working nodes.

#### 4.4.9 Application Deployment

La descrizione dell'applicazione fornita dallo sviluppatore elenca i container che compongono l'applicazione. I contenitori sono raggruppati in **pod**. Una pod è un gruppo di container che devono funzionare sugli stessi nodi worker, in altre parole devono essere distribuiti insieme e non devono essere isolati (un pod può essere un container) Per ogni pod/container singolo lo sviluppatore specifica la configurazione (es. da quale immagine deve essere istanziato) e il numero di repliche da distribuire.

#### 4.4.10 Management and Maintenance

Una volta che l'applicazione è in esecuzione, Kubernetes si assicura continuamente che lo stato dell'applicazione combaci sempre con la descrizione fornita dai dev. Se uno o più istanze smettono di funzionare a dovere (es un crash dell'app) Kubernetes si preoccuperà di farla ripartire. Se l'istanza, o un intero nodo worker, diventa inaccessibile, Kubernetes selezionerà un nuovo nodo per tutti i contenitori che erano in esecuzione su di esso. Mentre l'applicazione è in esecuzione lo sviluppatore può decidere di aumentare/ diminuire il numero di istanze, in questo caso la piattaforma si prenderà cura di aggiungere/ rimuovere istanze a runtime. Lo sviluppatore può anche lasciare che la piattaforma decida il numero ottimale di istanze di un determinato contenitore, in questo caso la piattaforma regolerà il numero di istanze in esecuzione in base a metriche in tempo reale come carico della CPU, consumo di memoria, query per secondo, ecc.

#### 4.4.11 Container Migration

La piattaforma si prende automaticamente cura di istanziare i container e di migrarli quando richiesto (es. quando un nodo worker fallisce). Quando un container fornisce un servizio ad un cliente esterno, la piattaforma deve prendersi cura di inoltrare le richieste quando il container viene spostato tra la piattaforma. Ciò che viene solitamente eseguito è che la piattaforma espone un servizio specifico con un **unico indirizzo IP pubblico**, indipendentemente dal numero di container e la loro posizione. La piattaforma con componente Kube-proxy si occupa di connettersi ai container e inoltrare il traffico in modo corretto, inoltre, il componente si occupa di implementare politiche di load-balancing per inoltrare il traffico in modo equilibrato al fine di garantire la scalabilità.

#### 4.4.12 Exposing Applications

I pod sono connessi tramite una rete privata locale in modo da permettere la comunicazione tra di essi. *Ogni pod ha il proprio indirizzo IP all'interno della rete locale*, non è quindi accessibile dall'esterno. Questa rete privata permette la comunicazione tra i container in esecuzione sullo stesso o su diversi worker.

Se un servizio necessita di esporsi sulla rete esterna (es. perchè espone un sito web o REST API dell'applicazione) il dev ha bisogno di definire l'oggetto servizio. Esso istruisce la piattaforma ad associare al pod un IP e una porta esterni, in questo modo il pod è raggiungibile da host esterni. Il servizio si prende cura di inoltrare la richiesta proveniente da IP esterno a quello interno associato all'istanza del pod.

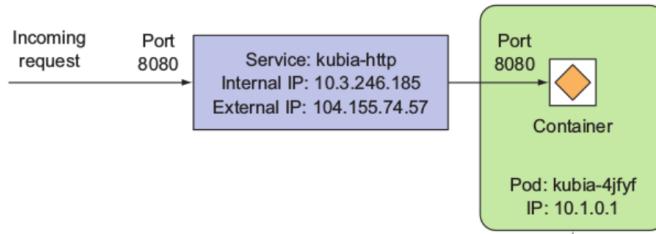


Figure 4.32: Exposes to external network.

#### 4.4.13 Load Balancing

Un servizio può essere connesso ad un pod che richiede istanze multiple e ridondanti per bilanciare il carico. In questo caso il servizio distribuisce le richieste tra le differenti istanze. Ad ogni pod la piattaforma associa un **replication controller** per assicurare che siano in esecuzione il corretto numero di istanze.

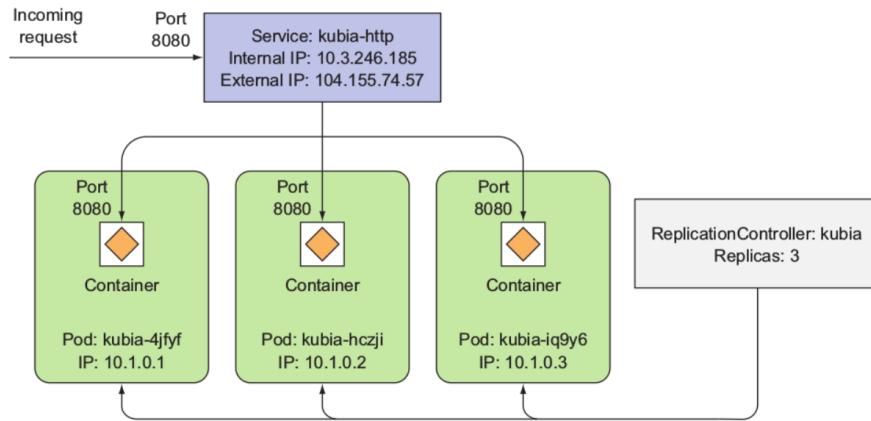


Figure 4.33: Replication controller to ensure load balancing.

#### 4.4.14 Failure handling

Il replication controller è responsabile anche di controllare monitorare se ci sono fallimenti nelle istanze del pod. Quando un fallimento viene individuato il replication controller reagisce creando una nuova istanza del pod.

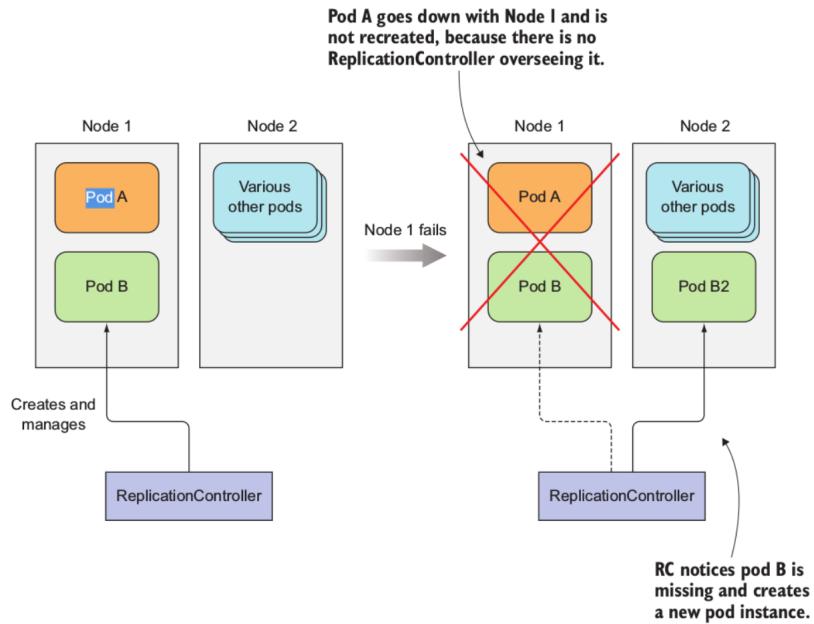


Figure 4.34: Failure handling.

#### 4.4.15 Pod design

La regola generale per raggruppare i container dentro pod è quella che i dev dovrebbero sempre eseguire container in pod separati a meno di specifici motivi che richiedono che siano eseguiti nello stesso pod, in tal caso vengono messi nello stesso nodo worker.

Ad esempio, un applicazione potrebbe richiedere una certa latenza nello scambio di dati tra due container, in questo caso è conveniente raggrupparli nello stesso pod così non cominceranno sulla rete.

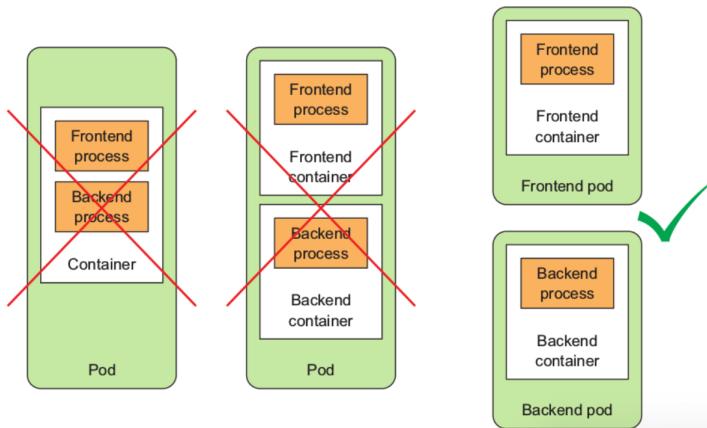


Figure 4.35: Pod design.

# Chapter 5

## Cloud Storage and Distributed File System

Nello scorso decennio le tecnologie per il salvataggio dei dati si sono evolute a ritmo sostenuto, è aumentato il volume di dati che possono essere salvati e il tempo di accesso ad essi si è drasticamente abbassato. Gli Hard Disk si sono evoluti dai Megabytes ai Terabytes e in più anche i prezzi si sono abbassati. La transizione ai Solid State Drive (SSD) ha permesso l'eliminazione delle parti mobili di un HDD tradizionali portando ad un ulteriore abbassamento dei tempi di accesso ai dati.

Parameter	1956	2016
Capacity	3,75MB	10TB
Average Access Time	~600ms	2,5-10 ms
Average Life Span	~2000 Hours	~22500 Hours
Price	9200\$/MB	0,032\$/MB
Physical Volume	1,9m <sup>3</sup>	34cm <sup>3</sup>

Figure 5.1: Table with the evolution of HDD

### 5.1 Block storage and File System

Il tipo di storage che può essere implementato con un hard drive è il *Block Storage* che offre la possibilità di salvare blocchi continui di dati non strutturati. Un blocco, solitamente chiamato *record fisico*, è una sequenza di bytes o bits e i blocchi possono essere salvati sui settori fisici dell'hard drive. Su una serie di blocchi viene costruito un **File System** che è responsabile dell'organizzazione dei dati in file e gestisce i metadati necessari affinchè i file possano essere salvati in blocchi differenti. Il file system è utilizzato dagli OS per salvare i dati degli utenti e quelli del sistema.

## 5.2 Object Storage

Recentemente è stata creata un nuovo tipo di storage chiamato **Object Storage**. Gli oggetti sono bundle di dati, ad esempio un file, con i rispettivi metadata. Ogni oggetto ha associato un ID univoco che è calcolato in base al contenuto dei metadata, le applicazioni possono accedere agli oggetti tramite l'ID, il set dei metadata non è definito a priori e può anche essere esteso. Ogni oggetto è immutabile, quindi se il metadata viene cambiato allora verrà generato un nuovo oggetto contenente la nuova versione. Può essere introdotto un *incremental change system* in modo da minimizzare la replicazione dei dati nel sistema.

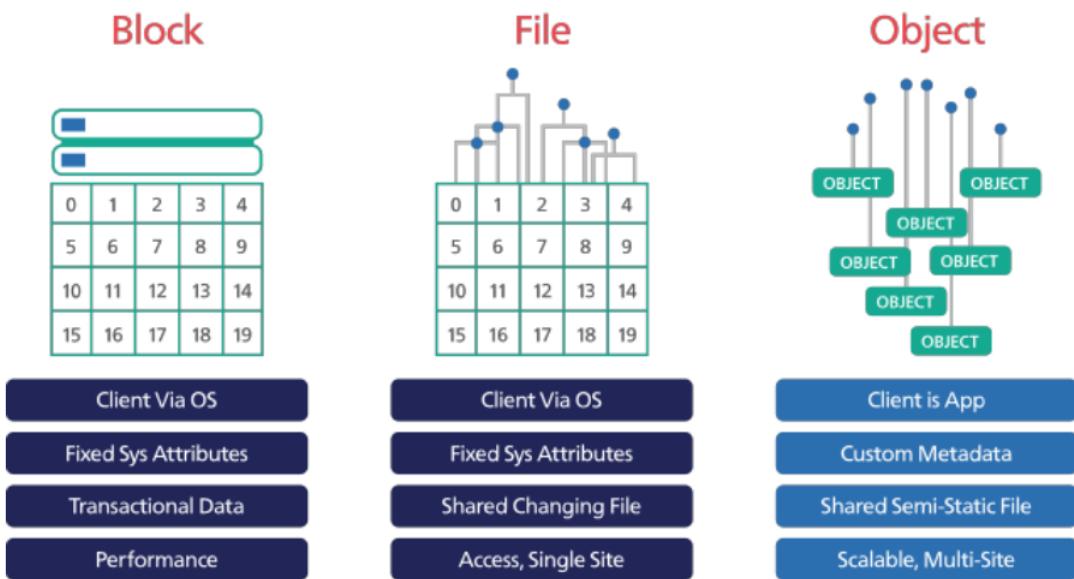


Figure 5.2: Comparison between different approaches.

## 5.3 Single Server Storage Model

Lo storage del modello di computing tradizionale è composto da uno o più hard drive connessi **ad un solo server**. Gli hard drive possono rompersi, il loro tempo di vita varia in base al carico al quale sono sottoposti, anche se generalmente il tempo di vita medio è di 5 anni. Per poter gestire il caso in cui un hard drive si rompa nel singolo server viene adottata la tecnologia **RAID** che permette di essere tolleranti alle rotture e permette anche la replica dei dati. Lo storage massimo disponibile è, però, dettato al numero fisico di slot che un certo server possiede per accogliere degli hard drive, questo problema può essere superato adottando file system *distribuiti*.

### 5.3.1 RAID

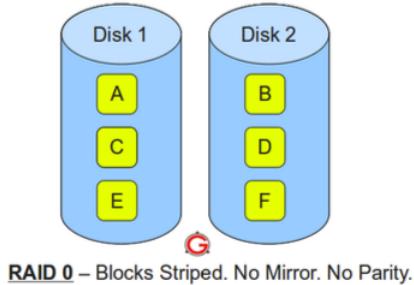
La tecnologia RAID (Redundant Array of Independent Disks) permette di combinare più hard drive fisici in uno o più hard drive logici che sono però virtuali. Sono disponibili vari schemi

di RAID in base allo scopo che si vuole ottenere: ridondanza dei dati, incremento delle performance oppure entrambi. Le funzionalità del RAID possono essere implementate in *hardware* tramite un controllore al quale sono attaccati gli hard drive o via *software*. Nel primo caso il controllore si occupa della gestione degli hard drive, di conseguenza le funzionalità RAID sono nascoste all'OS che accede direttamente all'hard drive virtuale. Nel secondo caso, quindi via software, le funzionalità raid sono implementate direttamente dall'OS che si occupa quindi di accedere direttamente agli hard drive fisici e di creare quelli virtuali.

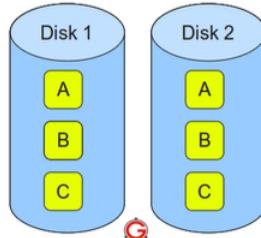
### RAID schema

A seconda di che schema si decide di adottare cambierà il modo in cui i blocchi verranno salvati negli hard drive fisici. Vediamo i 3 principali:

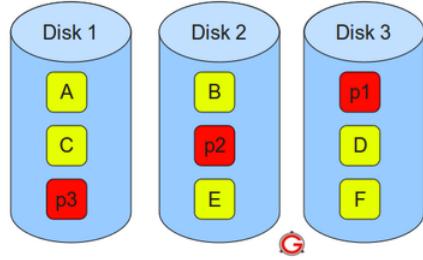
- **RAID 0:** viene usato per l'aumento delle performance, i blocchi sono sparsi per i vari hard drive fisici.
- **RAID 1:** viene usato se si vuole garantire la massima ridondanza dei dati. I blocchi sono replicati fra i vari hard drive fisici.
- **RAID 5:** è un ottimo trade-off fra i primi due casi elencati. I blocchi sono sparsi su diversi hard drive fisici, tuttavia ad ogni set di blocchi viene affiancato un blocco di parità ottenuto con lo XOR, questo per garantire la tolleranza ai guasti e di conseguenza recuperare i dati.



RAID 0 – Blocks Striped. No Mirror. No Parity.



RAID 1 – Blocks Mirrored. No Stripe. No parity.



RAID 5 – Blocks Striped. Distributed Parity.

Figure 5.3: Exemples of different RAID schemas

## 5.4 Distributed File System

Per superare le limitazioni fisiche imposte dalla soluzione a server singolo, si è giunti alla definizione dei *Distributed File System (DFS)*. Il DFS aumenta le capacità di storage distribuendo il file system su più server, il trasferimento dei dati e la sincronizzazione sono possibili sfruttando la rete LAN locale. Un DFS può essere usato per creare un vasto file system che combina insieme tutto lo storage disponibile di diversi server o può essere usato per allargare la capacità di un singolo server usando lo storage di un altro. Sono stati definiti diversi DFS, uno dei primi che sono stati definiti è il *Network File System (NFS)* che è ancora oggi molto utilizzato.

### 5.4.1 NFS

NFS adotta un approccio client-server: un server centrale permette l'accesso al proprio file system a più client, viene inoltre definito un protocollo per lo scambio dei messaggi che sono sincroni in modo da garantire la consistenza e semplificare l'implementazione. Il file system viene acceduto dalle applicazioni che girano sul client nella stessa maniera in cui i file locali vengono acceduti. Tutte le funzionalità di NFS sono implementate nel kernel di Linux, in modo che esse siano nascoste alle applicazioni.

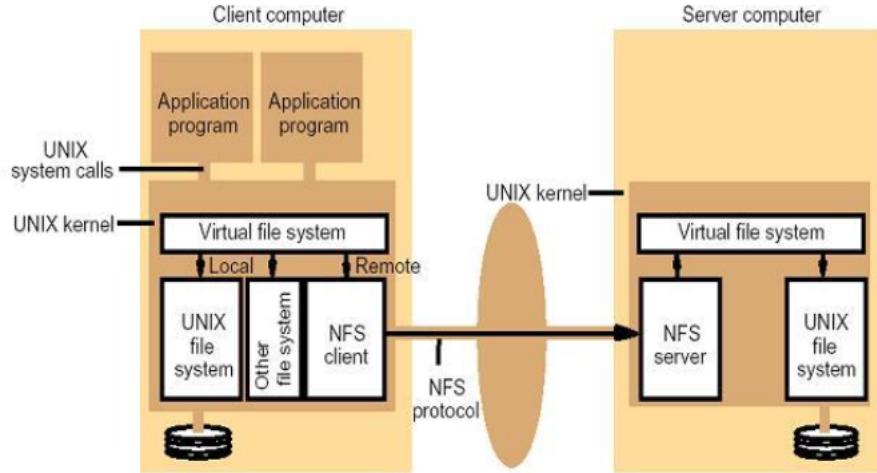


Figure 5.4: NFS client-server schema

### 5.4.2 GlusterFS

Anche se NFS ha uno storage condiviso la sua **architettura centralizzata** frena il suo utilizzo nella distribuzione pratica. Solitamente delle alternative **distribuite** sono utilizzate per aumentare la resilienza ai guasti e garantire la scalabilità anche grazie allo storage disponibile in locale. **GlusterFS** è un esempio di DFS che crea un file system distribuito mettendo insieme tutto lo storage disponibile su tutti i nodi. GlusterFS può essere usato in locale nello stesso modo in cui viene configurato NFS, tuttavia non c'è distinzione fra client e server, tutti i nodi che partecipano mettono a disposizione parte dello storage che hanno a disposizione.

#### GlusterFS - Basic Modes

GlusterFS garantisce diversi livelli di configurazione con diversi livelli di replica e di ridondanza. Le configurazioni base includono: *replicated volumes*, *distributed volumes* e *striped volumes*.

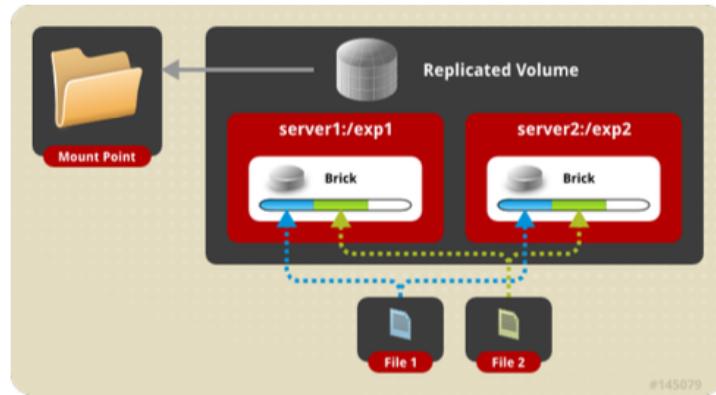


Figure 5.5: GlusterFS basic modes



Figure 5.6: GlusterFS basic modes

## GlusterFS - Advanced Modes

Per poter soddisfare diversi requisiti vengono permesse diverse combinazioni di configurazioni base, ad esempio: *striped replicated* e *distributed replicated*.

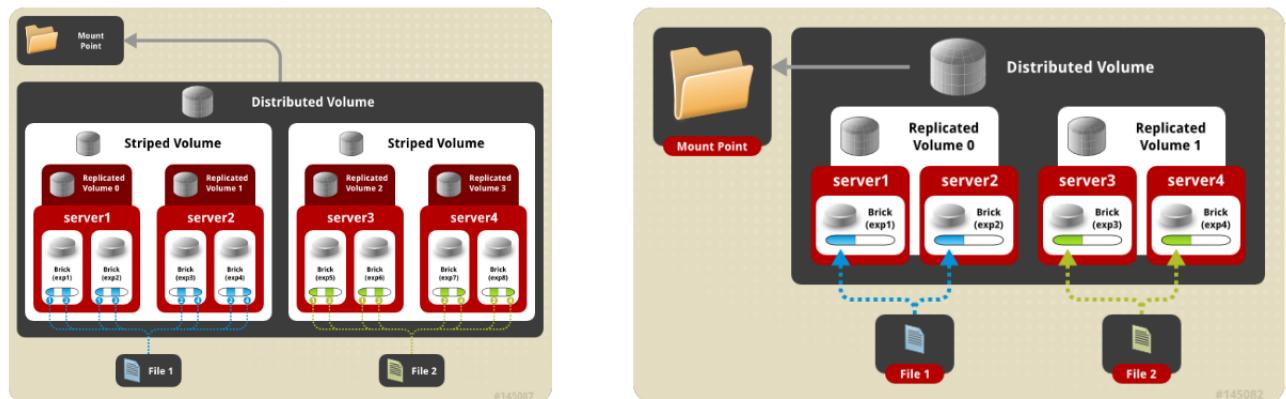


Figure 5.7: GlusterFS advanced modes

## 5.5 Cloud Storage

I requisiti di storage per le piattaforme di cloud computing sono differenti rispetto a quelli delle applicazioni tradizionali. Le applicazioni cloud hanno necessità di storage scalabile, in quanto lo storage scala col numero di VM che la piattaforma deve gestire, scala con il numero e la grandezza dei volumi creati e scala anche con la mole di dati generati da un servizio.

Si puossano anche sfruttare gli hard drive locali nelle piattaforme cloud dal momento che l'hypervisor crea hard drive virtuali per le VM della sua macchina fisica. Tuttavia questa soluzione non è sostenibile in quanto la scalabilità è limitata dai problemi del *traditional computing model*.

### 5.5.1 VM Live Migration

Utilizzando lo storage locale impedisce l'implementazione del *VM live migration*, che permette a una VM di essere spostata dinamicamente da un compute node ad un altro mentre la VM rimane in esecuzione. Com questa tecnica le cloud platform possono minimizzare i down time, è richiesto però che ci sia uno storage condiviso, altrimenti la VM deve essere spenta e l'hard drive virtuale deve essere spostato sul nuovo nodo.

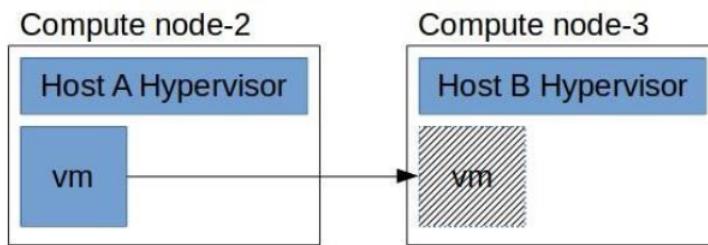


Figure 5.8: Exemple on VM live migration

### 5.5.2 Strorage Area Network

Per evitare di questa limitazione è stata introdotta la Storage Area Network (SAN) che consiste in una infrastruttura ad alta velocità composta da dispositivi con molto storage. I dispositivi di storage appositi sono connessi ai server che compongono la cloud platform e andranno a comporre lo storage a livello di blocchi. Lo storage può essere sfruttato dalla piattaforma per far girare i propri servizi, i dispositivi di storage possono usare nastri o dischi anche se questi ultimi sono i più comuni, utilizzano anche link ad alta velocità come la fibra ottica. Vengono anche sfruttati dei protocolli ad hoc per la comunicazione fra server e i dispositivi di storage, uno di quesri è **iSCSI**. Questo protocollo è basato su quello IP che permette di collegare capannoni adibiti allo storage dei dati, che permettono l'accesso a livello del blocco, ai dispositivi di storage portando i comandi iSCSI attraverso TCP/IP.

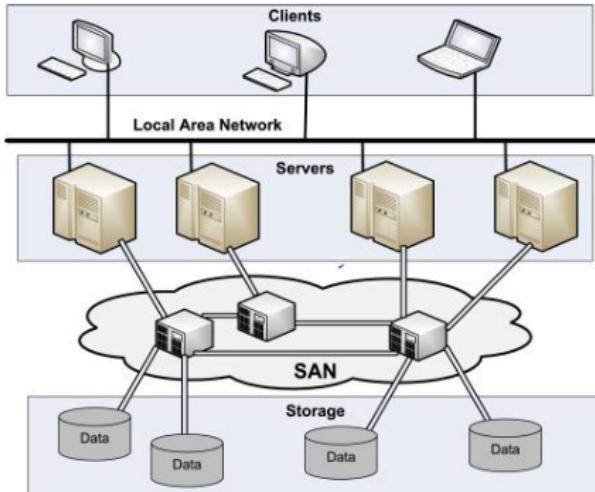


Figure 5.9: SAN architecture

### 5.5.3 Unified Storage Solutions

La SAN richiede soluzioni hardware ad hoc che sono complesse e costose, per questo non tutte le distribuzioni di applicazioni cloud possono giustificare questa strada. Per questo motivo i *file system distribuiti* hanno cominciato a rappresentare un'alternativa più economica rispetto a SAN.

Queste soluzioni sono state ispirate dal DFS, infatti il loro obiettivo è quello di offrire una soluzione a storage unificato per offrire scalabilità, performance elevate eliminando anche il *single point of failure*, tutto questo usando però dei pezzi hardware di uso comune. Fra queste soluzioni il più popolare al momento è **Ceph**

## 5.6 Ceph

Ceph garantisce un sistema di storage a livello aziendale, robusto e altamente affidabile, costruito però sopra hardware comune. Al centro di Ceph ci sono le seguenti caratteristiche:

- Ogni componente **deve** essere scalabile.
- Non deve essere presente il single point of failure.
- La soluzione **deve** essere basata sul software, open source e adattabile.
- Il software Ceph dovrebbe essere in grado di girare su hardware di comune utilizzo.
- Tutto dovrebbe essere in grado di auto mantenersi dove possibile.

Il risultato è che Ceph gestisce in maniera autonoma la replicazione dei dati e il recupero dei guasti, conseguentemente significa che non è più necessario implementare tecnologie di supporto come il RAID.

### 5.6.1 Ceph Services

Ceph può essere installato sui servizi canonici in modo da creare un sistema di storage (**Ceph Cluster**) nel quale la capacità totale di tutti gli hard drive del sistema viene messa a disposizione. Ceph inoltre offre diversi tipi di storage: *object storage*, *block storage* e *file system storage*.

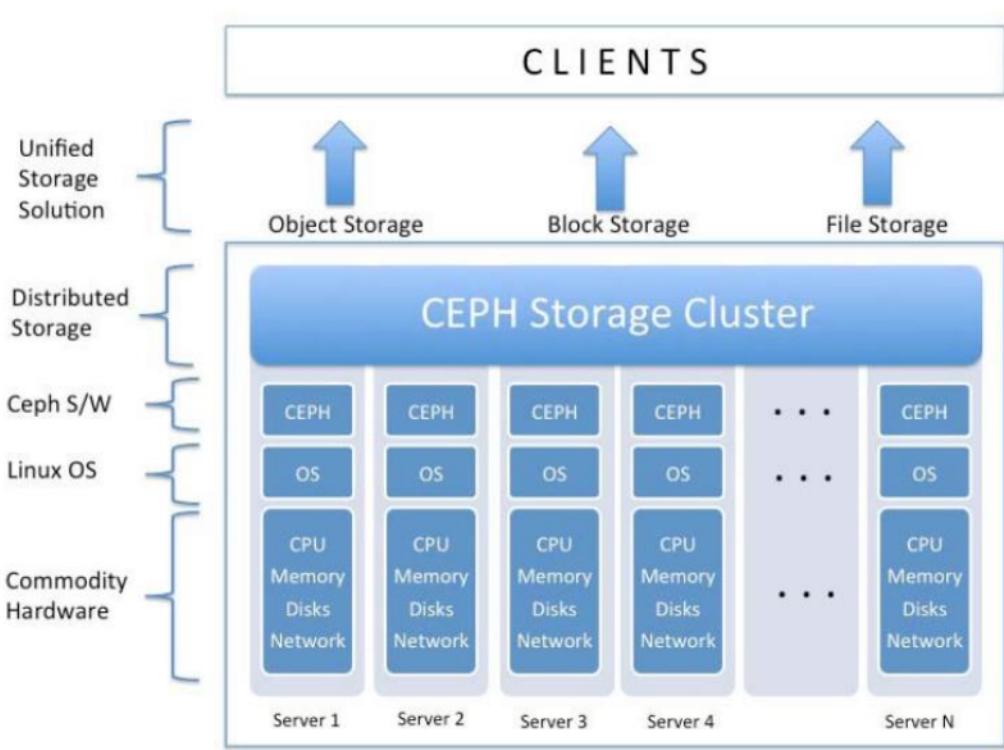


Figure 5.10: Ceph Services

### 5.6.2 Ceph Architecture

Uno storage cluster di Ceph è composto da diversi servizi software ognuno dei quali si occupa di una singola funzionalità di Ceph ed è separato dagli altri. I client interagiscono con uno di questi componenti in base al tipo di storage necessario. Come *core* di Ceph è presente **RADOS (Reliable Autonomic Distributed Object Store)**, tutto in Ceph è salvato come *oggetto* e il RADOS si occupa di salvare questi oggetti indipendentemente dal tipo di dato. Gli altri tipi di storage sono costruiti a partire da questo storage base di oggetti.

Il livello di RADOS si assicura che i dati rimangano in uno stato consistente e in maniera affidabile. Per quanto riguarda la consistenza, il livello RADOS mette in atto la replicazione dei dati, il rilevamento dei guasti e il conseguente recupero, la migrazione dei dati stessi e il ribilanciamento fra tutti i nodi del cluster. Le applicazioni che vogliono al servizio di storage di oggetti interagiscono con il RADOS attraverso **LIBRADOS**, una libreria che offre un interfaccia nativa per RADOS in diversi linguaggi.

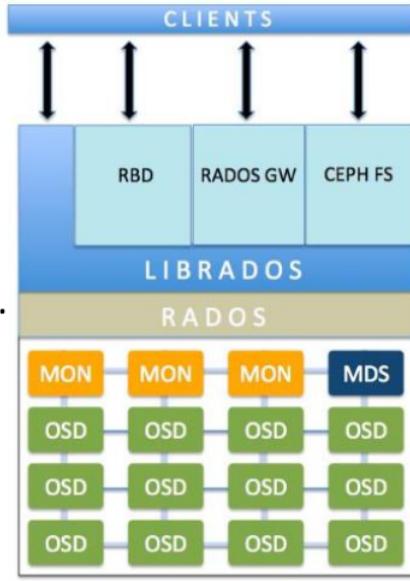


Figure 5.11: Ceph architecture

Vengono creati servizi addizionali per offrire servizi aggiuntivi rispetto allo storage di oggetti.

- **Ceph Block Device, also known as RADOS block device (RBD)**: permette lo storage a blocchi che possono essere formattati e trattati come gli altri dischi presenti nei server. Un dispositivo a blocchi in Ceph è equipaggiato con funzionalità specifiche come *thin provisioning* e *snapshot*. Il dispositivo può essere sfruttato per creare volumi o dischi virtuali per le VM.
- **Ceph File System (CephFS)**: si basa su Ceph MDS (Ceph Metadata Server) che tiene traccia della gerarchia dei file e salva i metadati solo per CephFS. Dal momento che RADOS e i dispositivi a blocchi di Ceph non necessitano di metadata non è necessario affiancargli un daemon di Ceph MDS.

Come detto precedentemente al cuore di Ceph c'è RADOS che fornisce tutte le sue funzionalità tramite l'algoritmo CRASH che implementa funzioni come alta disponibilità, affidabilità e assenza del single point of failure. Le funzionalità del livello RADOS sono implementate in maniera distribuita tramite i servizi OSD e MON:

- **OSD (Ceph object Storage Device)**: si occupa di salvare i dati sugli hard disk fisici di ogni nolo del cluster, solitamente è presente un'istanza di OSD per ogni hard disk fisico.
- **MON (Ceph Monitor)**: è il responsabile di monitorare la salute dell'intero cluster, il monitor si occupa di mantenere lo stato e la configurazione del cluster. Un set di istanze di MON sono presenti per rendere il cluster tollerante ai guasti.

### 5.6.3 Ceph Object

Un oggetto di Ceph è composto dai dati veri e propri e dai relativi metadati che sono uniti insieme, viene inoltre affiancato un ID univoco ad ogni pacchetto di Dati + Metadati. Gra-

zie all'ID viene garantita l'unicità dell'oggetto nell'intero cluster. Inoltre, non c'è limite alla grandezza dei dati all'interno dell'oggetto. Gli oggetti sono salvati nel *Object-based Storage Device* in maniera replicata in modo da garantirne l'elevata disponibilità. Quando il cluster riceve dal client una richiesta di scrittura, esso salva i dati in forma di oggetti, il daemon di OSD salva quindi i dati nel filesystem di OSD.

ID	Binary Data	Metadata
1234	010101010101010011010101010010 010110000101010011010101010010 010110000101010011010101010010	name1 value1 name2 value2 nameN valueN

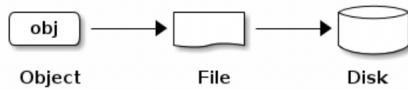


Figure 5.12: Ceph Object Structure

#### 5.6.4 CRUSH Algorithm

Sia i client che i Ceph OSD usano il *CRUSH Algorithm* per calcolare la posizione di un oggetto per scrivere o leggere in maniera indipendente, non c'è quindi bisogno di appoggiarsi ad un tabella centrale di lookup. L'algoritmo permette di avere una gestione dei dati migliore rispetto agli approcci precedenti, permette infatti una grande scalabilità raggiunta distribuendo un maniera chiara il lavoro di ogni componente nel cluster. CRUSH usa un metodo intelligente di replicazioni dei dati in modo da assicurare la resilienza.

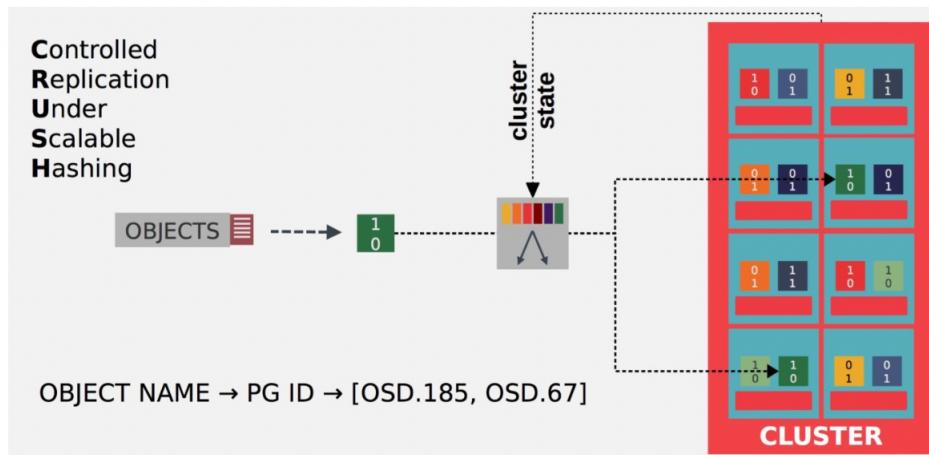


Figure 5.13: CRUSH computes the location based on the cluster status

#### 5.6.5 Cluster Map

Lo stato del cluster è rappresentato da una Cluster Map che può essere ottenuta da un'istanza qualsiasi di MON. La mappa include le seguenti informazioni:

- Monitor Map: contiene indicatori riguardo il tempo corrente, quando la mappa è stata creata e l'ultima volta che è stata modificata.
- OSD Map: contiene la lista dei pool, la grandezza delle repliche, la lista degli OSD e i loro stati.
- PG Map: contiene i dettagli di ogni placement group come il PG ID, Up Set, Acting Set e lo stato del PG.
- Cluster Map: contiene la lista dei dispositivi di storage, la gerarchia dei domini di fallimento e le regole per attraversare la gerarchia quando si salvano i dati.

Un placement group rappresenta l'OSD attuale e quindi l'hard disk fisico dove verranno salvati i dati.

Prima che un client Ceph possa leggere o scrivere devono contattare il Ceph Monitor per ottenere la versione più aggiornata della mappa, successivamente il client calcola il *placement group* semplicemente applicando una funzione hash all'ID dell'oggetto. Per calcolare gli OSD, quindi gli hard disk fisici, per salvare effettivamente l'oggetto si utilizza l'algoritmo CRUSH, esso si occupa di tradurre il placement group in una lista di OSD in base allo stato del cluster. La lista ottenuta consiste in un OSD primario che salva l'oggetto e in una serie di OSD secondari che salvano le repliche dell'oggetto stesso.

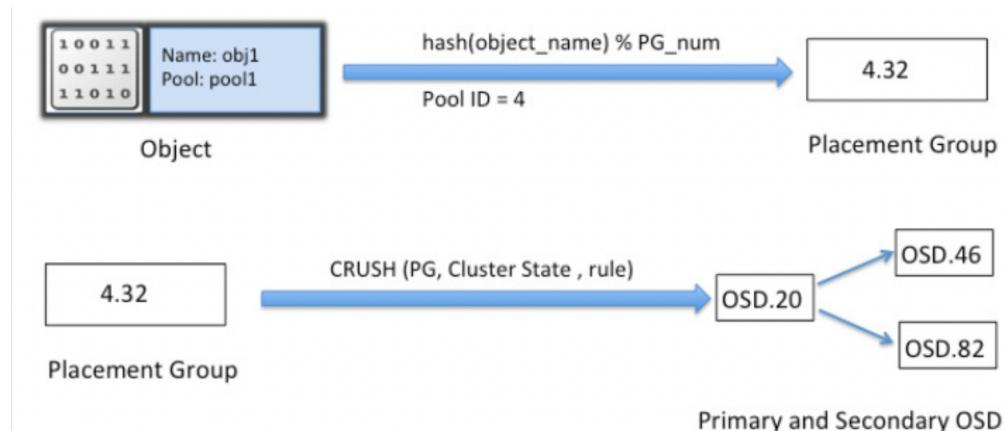


Figure 5.14: Mechanism to obtain the OSDs

### 5.6.6 Data Replication

Il client inserisce l'oggetto nel placement group selezionato nell'OSD principe, esso, tramite la sua copia della mappa CRUSH, può identificare l'OSD secondario e quello terziario per salvare le repliche. Dopo questo passaggio viene replicato l'oggetto nel placement group appropriato negli OSD secondari e terziari. La risposta al client avviene solo dopo aver effettivamente avuto la conferma di aver salvato correttamente l'oggetto. Con questo meccanismo gli OSD di Ceph si occupano di controllare se il salvataggio ha avuto esito positivo, questo assicura una disponibilità dei dati elevata e anche la sicurezza dei dati.

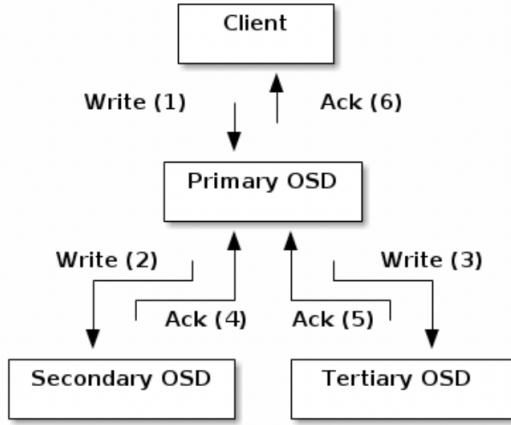


Figure 5.15: Mechanism to replicate data

### 5.6.7 Pools

Il sistema di storage di Ceph supporta i Pool che sono delle partizioni logiche per lo storing di oggetti. I client ricevono la Cluster Map dal Ceph Monitor e scrive l'oggetto nel pool, possono essere istanziati diversi pool in modo da accontentare i dati di diverse applicazioni.

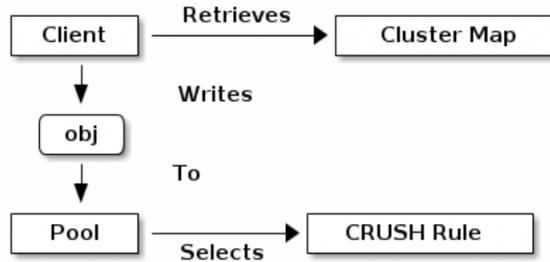


Figure 5.16: Mechanism to write in pools

### 5.6.8 MON Redundancy

Un cluster di Ceph può essere operativo con un singolo monitor, tuttavia questo rappresenta un single point of failure perché in caso di guasto al monitor i client non sarebbero più in grado di leggere o scrivere. Per aggiungere affidabilità e tolleranza ai gusasti si aggiunge un cluster di monitor, tuttavia con questo metodo la latenza e altri inconvenienti possono portare i monitor a non essere più al passo con lo stato corrente del cluster. Ceph deve avere un accordo tra le varie istanze del monitor riguardo lo stato del cluster, viene sempre utilizzata la maggioranza dei monitor (ad esempio: 1, 2/3, 3/5) e l'algoritmo Paxos per stabilire il consenso fra tutti i monitor.

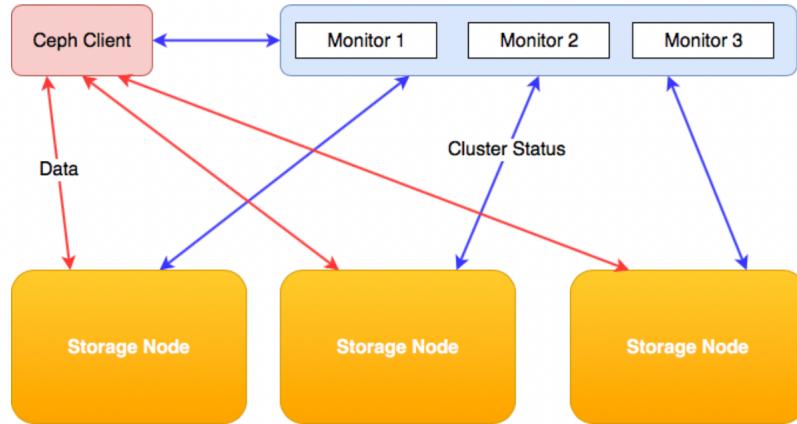


Figure 5.17: Ceph with a cluster of monitor

### 5.6.9 Placement groups computation

Ogni *Pool* ha un numero prestabilito di *Placement Group*, l'algoritmo CRUSH si occupa di mappare dinamicamente i PG in uno o più OSD. Quando un client salva un oggetto, la funzione hash lo mappa in uno specifico placement group, come già detto CRUSH pensa alla mappatura in OSD. La mappatura è decisa basandosi sulle zone con guasti nel cluster in maniera tale da garantire dati sicuri anche in presenza di componenti non funzionanti. Le zone con guasti non prendono in considerazione solo guasti agli hard disk o ai server, ma può prendere in considerazione anche l'architettura della rete o eventuali problemi con la corrente elettrica.

Inoltre, i gruppi sono mappati in modo da garantire bilanciamento fra gli OSD, quindi proporzionalmente alla grandezza del disco di ogni OSD.

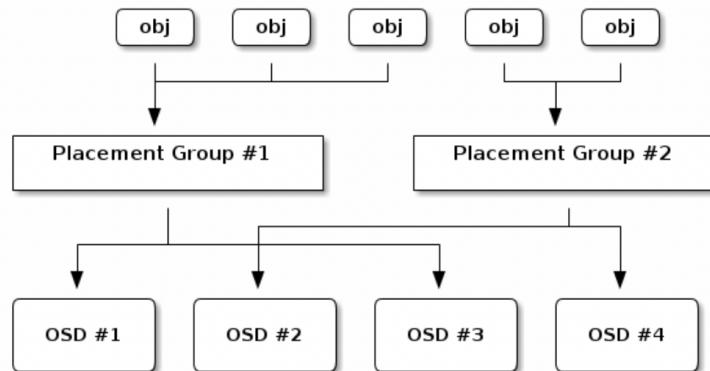


Figure 5.18: Mapping of different components

### 5.6.10 Recovering and Rebalancing

Mappando gli oggetti ai placement group si crea un livello indiretto fra il Ceph OSD Daemon e il Ceph client, grazie a questo livello il cluster può scalare senza problemi e ribilanciare dinamicamente il punto di storage degli oggetti. Ad esempio se viene aggiunto un OSD Daemon al

cluster, la mappa del cluster viene aggiornata e conseguentemente viene ricalcolato dove i placement group vengono mappati, questo porta a un cambiamento nel luogo di salvataggio dell'oggetto risultando quindi in un ribilanciamento generale.

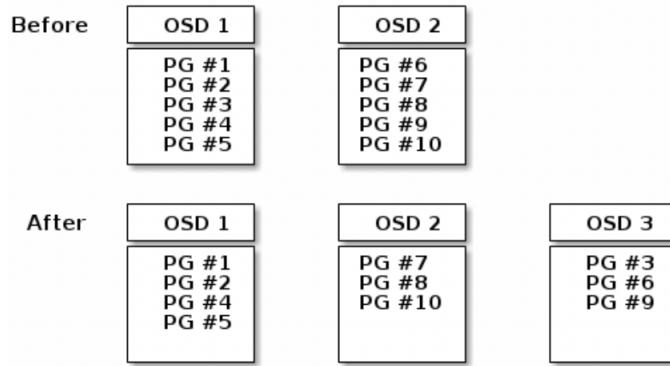


Figure 5.19: Rebalancing Exemple

### 5.6.11 Ceph and OpenStack

Molti servizi di OpenStack utilizzano Ceph come tecnologia di cloud storage. Ecco alcuni esempi:

- **Cinder** può sfruttare Ceph per salvare i volumi delle VM, in questo caso viene usato il Ceph block storage.
- **Swift** può sfruttare Ceph per salvare i suoi oggetti, in questo caso viene utilizzato il servizio di storage di default.
- **Ceilometer** usa Ceph per salvare i dati Telemetrici.
- **Glance** usa Ceph per salvare i template degli OS per le istanze future delle VM.
- **Nova** usa Ceph per salvare gli hard drive virtuali delle VM.

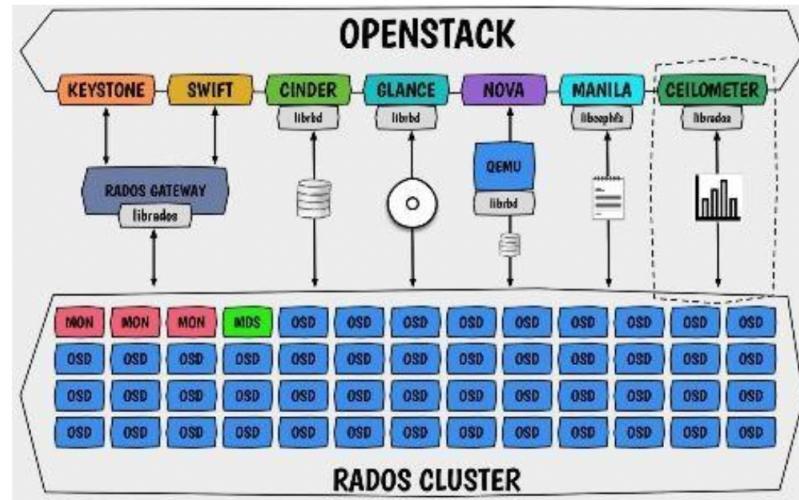


Figure 5.20: Various OpenStack services with Ceph

### 5.6.12 Storage as a Service

Tra i vari servizi che i cloud provider possono fornire c'è anche lo **Storage as a Service (STaaS)**. Questo servizio offre l'accesso al cloud storage attraverso internet, in altre parole viene offerto l'accesso alla loro infrastruttura di cloud storage. Spesso i servizi STaaS adattano uno storage a oggetti, per salvare o leggere gli oggetti si utilizza un'interfaccia di tipo REST tramite delle apposite API, questo può essere usato per salvare i backup o storage a lungo termine di grandi dati. Un esempio è il servizio Swift di OpenStack che può essere usato per esporre un Ceph cloud storage.