CLOUD COMPUTING

Corso A

Autore
Giuseppe Acocella
2024/25

Ultima Compilazione - March 3, 2025

Contents

1 Introduzione

Il Cloud Computing ha rivoluzionato l'utilizzo dei calcolatori, permettendo di astrarre sull'effettiva implementazione e fornendo numerosi servizi agli utenti. Formalmente il Cloud Computing è un modello che permette di avere accesso alla rete su richiesta di risorse di calcolo configurabili.

1.1 Servizio vs Bene

Questo nuovo modo di utilizzare i calcolatori si basa su uno specifico schema, ossia quello del **servizio** in sostituzione al **bene** fisico. Un esempio è **Spotify**: Prima di un applicazione che forniva l'accesso alla musica (**servizio**) esisteva il **bene** fisico, ad esempio il vinile. Questo ci permette di astrarre dal concetto di possedimento, lasciando all'utente la possibilità di accedere ad un servizio che non dipende da lui.

1.2 Service Level Agreement (SLA)

Ogni rapporto va stabilito formalmente con un contratto. Lo **SLA** si occupa di questo, permette una formulazione del rapporto tra user e servizio. All'interno si trovano tutti i dettagli su cosa un user deve aspettarsi dal servizio, anche in caso di malfunzionamento ed indisponibilità del servizio stesso.

1.3 Quality of Matter (QoS)

La qualità di un servizio offerto dipende fortemente dalle tecnologie su cui si basa, come ad esempio il servizio di hosting. Lo sviluppo dunque di un app ha bisogno di operazioni di **previsioning** che calcolino le risorse necessarie ad una buona esecuzione del servizio in questione.

Under/Overprevisioning Immaginando di stabilire una capacità massima per il nostro servizio:

1. **Underprevisioning**: La capacità massima è inferiore al picco massimo di richiesta al servizio, questo porta a situazioni di stallo e rallentamenti.
2. **Overprevisioning**: La capacità massima è inutilmente più grande della media delle richieste, di conseguenza si causa un forte spreco di risorse.

CapEx e Cloud Computing In passato, per avviare un attività era sempre necessario un primo capitale iniziale, che permettesse il corretto avvio. Molti dei servizi più in voga (IaaS,Paas,SaaS) del cloud computing permettono l'avvio di applicazioni e piccole aziende senza nessuna difficoltà iniziale.

Elasticità e Risk Transfer Il Cloud Computing offre anche altri vantaggi, risulta infatti molto semplice scalare ad esempio da uno storage di $1Mb$ ad uno di $1Tb$ (**Elasticità**) dato che l'intera responsabilità della reallocazione viene lasciata alla piattaforma. Lo stesso vale per i backup dei dati, che vengono autonomamente effettuati dalla piattaforma (**Risk Transfer**).

1.4 Model Service (IaaS, PaaS, SaaS)

Tutti i servizi definiti prima si basano su un **modello**, ossia quanto il cloud è "coinvolto", elenchiamo tutti i modelli, nello stack rappresentiamo tutte le **responsabilità** date all'**user**:

1. **IaaS**: Infrastracture as a Service:

Sistema Operativo
Applicazione
Dati
Impostazione Dati

2. **PaaS**: Platform as a Service:

Applicazione
Dati
Impostazione Dati

3. **SaaS**: Software as a Service:

Dati
Impostazione Dati

Cloud Privato/Ibrido/Pubblico Ogni azienda nel corso del proprio sviluppo ha la necessità di chiedersi come impostare la propria rete di calcolatori. Illustriamo tre modi:

1. **Cloud Privato**: L'azienda ha la necessità di costruire il proprio data center, prendendosi tutte le responsabilità circostanti come la gestione dei backup, la manutenzione ecc... Questo permette però all'azienda stessa di avere un controllo diretto sui propri server, e di conseguenza dei propri dati.
2. **Cloud Pubblico**: L'azienda sceglie di hostare tutti i propri servizi su server offerti da un provider (Amazon, Google, ...). Questo scaricherà tutte le responsabilità sul provider, causando un costo all'azienda.
3. **Cloud Ibrido**: L'azienda sceglie di suddividere i propri dati, in modo tale da utilizzare sia server locali sia servizi a pagamento di provider esterni.

Vendor Lock-In La scelta dell'azienda di utilizzare o meno cloud esterni a pagamento potrebbe portare a difficoltà future di migrazione dato che i provider potrebbero scegliere di "rendere difficile" l'uscita ai clienti. Di conseguenza diventa necessario per le aziende sviluppare un **exit plan** per una potenziale migrazione.

2 Infrastracture as a Service (IaaS)

Analizziamo nello specifico l'Infrastruttura come Servizio. Queste si suddividono in due sottocategorie:

1. **Tipo Compute**: Servizio che offre calcolo computazionale virtualizzato, veri e propri calcolatori remoti. Elenchiamo alcune caratteristiche note:
   (a) Solitamente forniscono **istanze**, ossia **server virtuali**.
   (b) Spesso utilizzano preset con Windows/Linux.
   (c) Gamma di istanze basate su CPU, Memoria, GPU, Storage...
   (d) Istanze a pagamento secondo i criteri **on demand**[1] oppure **istanza spot**[2].

   Un esempio di questo tipo di servizio è **Amazon EC2**.

2. **Tipo Storage**: Servizio che offre spazio di archiviazione. Elenchiamo alcune caratteristiche note:
   (a) **Backup** automatici.
   (b) **Versioning** di dati offerto di default.
   (c) **Classi** di memorizzazione in base alla **frequenza** d'**accesso**.
   (d) Configurazioni predefinite di **sicurezza** e **cifratura**.

2.1 Hypervisors

Gli Hypervisors permettono la gestione di ambienti virtuali sulla stessa macchina fisica. Illustriamo lo stack dei due possibili tipi di Hypervisors:

1. **Tipo 1**: Stack caratterizzato dall'assenza del **Host OS**:

App A	App B
Bins/Libs	Bins/Libs
Guest OS	Guest OS
Hypervisor	
Hardware	

2. **Tipo 1**: Stack caratterizzato dalla presenza del **Host OS**:

App A	App B
Bins/Libs	Bins/Libs
Guest OS	Guest OS
Hypervisor	
Host OS	
Hardware	

Solitamente il tipo 1 è utilizzato nei data center, mentre il tipo 2 nei personal computer che hanno la necessità di virtualizzare per qualche specifico motivo.

[1] Pagamento proporzionato all'utilizzo.
[2] Pagamento fisso

2.2 Identity and Access Managment (IaM)

L'**IaM** gestisce l'accesso sicuro ai servizi, e si suddivide in due specifiche operazioni:

1. **Autenticazione**: $User_1$ viene confrontato ad un attesa identità di $User_1$, confrontando quindi identità corrente ad una virtuale.
2. **Autorizzazione**: Dopo aver autenticato $User_1$ è necessario stabilire quali permessi possieda.

L'**IaM** è detta eventually consistent perchè spesso i sistemi dipendono da transazioni distribuite che causano **inconsistenza temporanea**, questa va risolta successivamente, quindi non si riesce ad affermare che il sistema sia sempre **consistente**.

3 Containers e Virtualizzazione

Vogliamo generare degli "ambienti virtuali" come nel caso degli hypervisors senza però portarsi dietro un vero e proprio **Guest OS**. Di conseguenza immaginiamo uno schema del genere:

<table>
<tr><td>App A</td><td>App B</td></tr>
<tr><td>Bins/Libs</td><td>Bins/Libs</td></tr>
<tr><td colspan="2">Container Manager</td></tr>
<tr><td colspan="2">Host OS</td></tr>
<tr><td colspan="2">Hardware</td></tr>
</table>

La **virtualizzazione** di **ambiente** è un processo che esiste da molto tempo, ma lo standard degli ultimi anni è stato definito da Docker.

Docker è una piattaforma che ci permette di lanciare applicazioni in ambienti isolati, questo genera una **completa portabilità** di un applicazione da macchina a macchina grazie alla virtualizzazione di tutte le dipendenze.

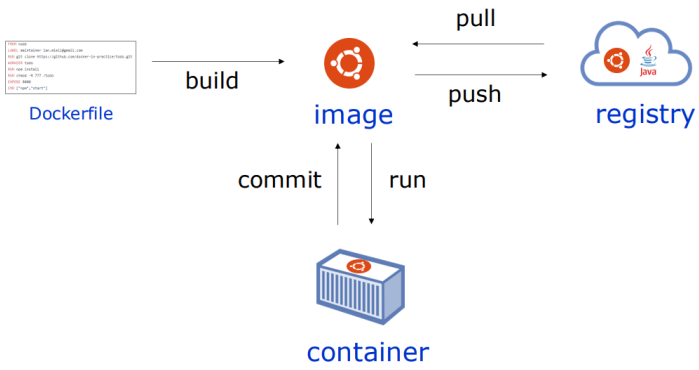
Confronto con Hypervisors Il **Docker Engine** dunque gestisce i container sulla macchina fisica come l'**Hypervisor** gestisce le varie macchine virtuali sulla macchina fisica.

Volume Quando un applicazione richiede la persistenza dei dati anche dopo l'arresto risulta necessario l'utilizzo di un volume, ossia un "disco d'archiviazione virtuale" compatibile nativamente con i container. Questi possono essere utilizzati per rendere condivise risorse tra i container.

3.1 Dockerfile, Immagine, Container

Entriamo nello specifico, descrivendo le fasi di costruzioni di un container:

1. **Dockerfile**: File di testo che segue una specifica sintassi e permette al **Docker Engine** di interpretarlo ed eseguire le istruzioni al suo interno che terminano producendo un immagine.
2. **Registry**: Riferimento ad un immagine online, una sorta di repository, solitamente ottenibile da **Dockerhub**.
3. **Image**: Oggetto **eseguibile** dal Docker Engine, permette di costruire un container sulla base dell'immagine data.
4. **Container**: Effettivo ambiente virtuale generato sulla macchina. Questo può essere mandato in esecuzione permettendo alle applicazioni al suo interno di lavorare in ambienti protetti ed isolati.



3.2 Docker Compose, Docker Swarm e Mapping Porte

Docker Compose Grazie ad un file di configurazione in formato *yaml* (*.yml*) permette la coordinazione di più microservizi su container diversi. In questo modo si permette la comunicazione tra microservizi grazie alla **rete overlay** settata dal Docker Compose.

Docker Swarm Servizio di supporto offerto da Docker che permette varie semplificazioni:

1. Rete di microservizi diversi divisa in nodi **worker** e nodi **manager**.
2. Riorganizzazione del lavoro tra i nodi in caso di fail di specifici nodi.
3. Reboot automatico (detto Self Healing) in caso di fail dei microservizi sui container.

Mapping di Porte dei Container A causa della virtualizzazione si crea inconsistenza sulla gestione delle porte, infatti i container faranno riferimento a delle **porte virtuali** non realmente esistenti sulla macchina, anche essendo "rappresentate dallo stesso numero". Di conseguenza diventa compito di **Docker** e di chi gestisce le applicazioni gestire il map delle **porte virtuali** su **porte reali** della **macchina fisica**.

4 Function as a Service (FaaS)

Questo tipo di modello porta il livello d'astrazione quasi al massimo, dando la possibilità all'user di inserire il proprio codice in una e vera **textbox** ed eseguirlo direttamente. In questo modo si deresponsabilizza del tutto l'utente dalla configurazione e gestione di ambienti/macchine virtuali. Si basa sulla **Event Driven Programming**, ossia programmazione guidata dall'avvenimento di specifici eventi. Gli ambienti forniti dai provider di (FaaS) sono compatibili con la maggior parte dei linguaggi e le relative librerie.

Trigger e API Gateway Spesso queste funzioni si espongono all'esterno grazie a dei **trigger**, come ad esempio un **API Gateway** che ci permette di effettuare delle **GET/-POST** in direzione della funzione stessa.

4.1 AWS Lambda e Definizione di Serverless

AWS offre il proprio servizio di (FaaS), anche il più utilizzato al momento sul mercato. Questo servizio è **AWS Lambda** e permette di caricare funzioni ed interagire con esse con l'impostazione dei trigger definiti prima. La lontananza tra l'user e la configurazione del server su cui sta lavorando caratterizza la definizione di **serverless**. **Ovviamente l'user sta utilizzando un server**, ma lo sta facendo in maniera "**implicita**".

AWS Step Function Lo sviluppo su AWS Lambda non si limita a singole funzioni atomiche, ma si estende anche a veri e propri sistemi di microfunzioni che interagiscono tra loro secondo uno specifico diagramma di flusso, realizzando intere applicazioni. Le Step Function si basano dunque sulla combinazione di λfunzioni che seguono un workflow specifico. Il cosiddetto servizio di orchestrazione permette di parallelizzare, mettere in serie, in loop ecc... una quantità arbitraria di funzioni.

Pricing di AWS Lambda Una caratteristica fondamentale di questo servizio è quella di definire il proprio costo in base all'utilizzo delle chiamate alla funzione lambda in questione. Questo comporta costi modici anche sull'ordine dei milioni di chiamate. Chiaramente viene però effettuato uno **scaling** anche in base alle risorse utilizzate (memoria e tempo di computazione) dalla chiamata alla funzione.

4.1.1 Pesante Vendor Lock In delle FaaS

Immaginiamo di aver costruito un intera applicazione grazie alla composizione di funzioni lambda. Se per qualche motivo dovesse risultare necessario cambiare provider del servizio (FaaS) allora avremmo un gran problema. Questo perchè i **trigger** delle funzioni della nostra applicazione **sono proprietari**. Di conseguenza non è impossibile migrare verso un nuovo provider ma risulta **molto difficile**, dovendo modificare il codice sorgente dell'applicazione. Dunque ogni tipo di semplificazione sulla configurazione e sulla manutenzione del ambiente la pago in potenziale libertà di movimento.

5 Platform as a Service (PaaS)

Questo modello basato sulla piattaforma come servizio deresponsabilizza ancora di più l'utente rispetto all'infrastruttura come servizio, l'user infatti non dovrà occuparsi della **gestione dell'ambiente** e questo modello risulta anche facilmente adattabile a nuove tecnologie. Nello specifico l'user non sarà più il responsabile della gestione del **Setup**, dello **Scale** e del **Managment**. Solitamente l'IaaS viene utilizzato in contesti di sviluppo, debug, testing o deployment.

Differenze tra PaaS e IaaS Spesso il confine tra questi due tipi di modelli non è ben definito, infatti molti provider forniscono contemporaneamente sia PaaS sia Iaas. Formalmente però il PaaS è più ad alto livello, liberando l'user da ancora più responsabilità. Un esempio di piattaforma come servizio può essere l'utilizzo di un IDE online. Solitamente nei PaaS ci vengono fornite direttamente le piattaforme d'utilizzo dell'user e non l'accesso diretto alla macchina host.

Add-On e Lock In dei PaaS Una delle migliori funzionalità di questo modello è la presenza di Add Ons che offrono la possibilità di aggiungere estensioni quasi in modalità "drag and drop". Questo tipo di aggiunte però lega l'applicazione al corrente provider di (PaaS), dato che tutti i plug in aggiunti sono proprietari. In tale modo si definisce un tipo intrinseco di **lock in** del provider sull'applicazione.

5.1 Heroku

Servizio cloud (PaaS) basato su containerizzazione di ambienti. La caratteristica di questi container è la loro facile scalabilità. I container di questo servizio sono denominati **dynos**, e la loro gestione automatica genera grande scalabilità del servizio.

Scalabilità dei Dynos L'user non dovrà mai occuparsi della gestione manuale delle dynos, ma queste verranno scalate automaticamente in base alle richieste effettuate al servizio hostato sulle varie dynos. Sono quindi dei container basati su Linux autogestiti sotto il punto di vista dello scaling.

Build/Run Time Durante il tempo di build, avendo a disposizione il codice sorgente da deployare e tutta la lista di dipendenze, si compone un **Procfile** che contiene il comando da eseguire all'avvio. L'esecuzione del procfile è detta **build** e produce un oggetto detto **slug**. Il tempo di run invece acquisendo uno **slug** permette la costruzione di uno stack e "l'engine" saprà quante dynos dedicare all'esecuzione dello slug in questione. La grande scalabilità si basa anche sullo schema simil "produttore/consumatore" utilizzato spesso tra i vari tipi di dynos.