Alma Mater Studiorum - Università di Bologna

Scuola di Scienze
Corso di Laurea in Informatica per il Management

Data Locality in Serverless Computing

Tesi di laurea

Relatore:
Chiar.mo Prof.
Gianluigi Zavattaro

Presentata da:
Andrea Fenati

Correlatori:
Dott. Saverio Giallorenzo
Prof. Jacopo Mauro

III Sessione
Anno Accademico 2018-2019

Introduzione

Negli ultimi anni il *serverless computing*, un nuovo paradigma cloud, ha sperimentato una rapida crescita. Questo modello, chiamato anche Function as a Service (FaaS), permette l'esecuzione di funzioni stateless in risposta ad eventi asincroni.
Il suo incremento di popolarità è derivato dalla semplicità di utilizzo. Lo sviluppatore si preoccupa solamente di scrivere il codice delle funzioni e di specificare i requisiti in termini di risorse all'interno della console del provider utilizzato. Tutto il resto, compreso il dimensionamento delle risorse, è gestito in modo automatico dal gestore cloud in base al carico di lavoro richiesto. Inoltre, FaaS offre modalità originali di design e di sviluppo software unite ad una maggior flessibilità nell'uso e nel calcolo dei costi.

Questo elaborato è stato inserito in un contesto più ampio, al quale ha partecipato un laureando della Magistrale di Informatica dell'Università di Bologna e due correlatori della *University of Southern Denmark*. Il progetto, partendo dalla piattaforma serverless open-source Apache OpenWhisk, è volto a dimostrare l'importanza della *data locality* durante la fase di scheduling delle funzioni.
La *data locality* è importante per ridurre i tempi di esecuzione nel caso in cui le funzioni necessitino di interagire con basi di dati.
Come dimostrato in questa tesi, eseguire le *cloud functions* il più vicino possibile ai dati utilizzati riduce considerevolmente la latenza.

Nel Capitolo 1 viene presentata una breve e sintetica panoramica sul cloud computing, descrivendo vantaggi, svantaggi e i principali modelli di servizio esistenti.
Nel capitolo 2 si analizza ed approfondisce il modello del Function as a Service.
Nel capitolo 3 si tratta la piattaforma serverless Apache OpenWhisk, si analizzano le sue componenti di utilizzo ed il suo funzionamento interno.
Nel capitolo 4 viene presentato il progetto di cui questa tesi fa parte, si discutono le tecnologie utilizzate e vengono introdotte le *locality*.
Nel capitolo 5 vengono descritti i test eseguiti per verificare la data loclity ed in seguito vengono presentati e discussi i risultati ottenuti.

Indice

Capitolo 1

Cloud Computing

Per molti anni le aziende hanno comprato e costruito internamente i propri server e data center dove poter installare applicazioni web, databases, backends per applicazioni mobile, ecc ...

Una soluzione alternativa è arrivata nell'ultimo decennio con l'avvento di quello che comunemente viene chiamato *cloud*.

Il cloud computing essenzialmente permette di ottenere risorse computazionali, gestite da terze parti, accessibili via Internet e con pagamento in base al consumo e al tipo di servizio.

Questa tecnologia ha rivoluzionato completamente l'industria dell'Information Technology (IT) poiché ha conferito la possibilità ad aziende medie e piccole di poter usufruire di tecnlologie che precedentemente erano accessibili solo a grandi aziende, le quali potevano permettersi l'acquisto e la gestione di un data center di proprietà, che rappresenta un gran esborso in termini economici.

1.1 Vantaggi del cloud

Il cloud computing ha portato diversi benefici, associati al tipo di servizio che si utilizza, ma fondamentalmente permette alle aziende di non dover comprare e manternere proprie infrastrutture, facendo sì che i progetti si possano sviluppare molto più velocemente e senza grandi esborsi economici fissi iniziali. Il beneficio principale del cloud quindi è l'**agilità**, cioè l'abilità di accellerare lo sviluppo di nuovi servizi senza doversi scontrare coi classici problemi di un progetto IT.
Un altro vantaggio è la **scalabilità** perché solitamente i servizi cloud permettono un aumento di risorse in modo molto veloce e sicuro.
Oltre ai benefici già citati possono essere considerati tali anche:

- **Adattabilità**: il servizio è personalizzabile sotto molti aspetti e permette di creare vari tipi di applicazioni; copre un vasto campo di utilizzi.
- **Affidabilità**: il fornitore di servizio garantisce affidabilità e in caso di problemi fornisce supporto.
- **Sicurezza**: il cloud provider predispone misure di sicurezza che difficilmente un'azienda adotterebbe in un proprio server.
- **Multi-tenancy**: architettura software in cui una singola istanza del servizio è utilizzata contemporaneamente da più tenants, ovvero i fruitori del servizio.
- **Riduzione dei costi**: legato al multi-tenancy, grazie al quale il fornitore di servizio sfrutta economie di scala per offrire il servizio a costi contenuti.

1.2 Svantaggi del cloud

Sebbene i vantaggi del cloud siano molteplici, questo approccio ha anche dei difetti:

- **Dipendenza da Internet**: i servizi cloud sono completamente dipendenti da una connessione ad una rete internet e quindi un'impossibilità della connessione può determinare un blackout di ogni attività ad esso collegata.

- **Latenza**: nel caso di grosse moli di dati e strutture di storage dislocate in diversi luoghi, per effettuare comunicazioni c'è il rischio di impiegare troppo tempo.

- **Sicurezza informatica e violazione privacy**: Poiché i dati sono memorizzati in server su macchine virtuali, spesso sono soggetti a interventi del provider e di conseguenza sorge il rischio di manipolazioni per ricerche di mercato, spionaggio industriale ecc ...
  Il rischio sicurezza aumenta con le reti wireless, più esposte a casi di pirateria informatica.

1.3 Tipologie di servizio

I cloud providers offrono diversi modelli di servizo.
Il NIST (National Institute of Technology) [1] suddivide il cloud computing in tre principali tipi di servizio: Infrastructor as a Service (IaaS), Platform as a Service (PaaS) e Software as a Service (SaaS).
Col tempo però si stanno aggiungendo altri tipi di servizi, spesso versioni ibride tra i modelli principali.

Infrastructor as a Service (IaaS)

IaaS mette a diposizione macchine virtuali e non, storage virtuale, infrastrutture virtuali e altre risorse hardware di cui i clienti possono usufruire. Il provider che fornisce questo servizio gestisce tutta l'infrastruttura, mentre il cliente si occupa del suo sviluppo che include il sistema operativo, le applicazioni e le interazioni utente con il sistema.
Questo servizio attrae aziende che vogliono avere il controllo dell'infrastruttura della loro applicazione senza però occuparsi dell'hardware e della manutenzione (fisica).
Esempi di IaaS molto conosciuti sono: Amazon Web Services (AWS), Google Cloud Platform e Microsoft Azure.

Platform as a Service (PaaS)

PaaS fornisce al consumatore una piattaforma che supporta lo sviluppo di applicazioni in ottica cloud computing.
La piattaforma include linguaggi di programmazione, librerie, servizi e strumenti dedicati, sviluppati dal provider. PaaS permette al cliente di concentrarsi puramente sullo sviluppo di un prodotto o di una applicazione senza doversi preoccupare del contesto in cui opera.
Si ha un servizio dinamico e capace di adattarsi alle esigenze dei clienti che, nel caso lo richiedano, possono ottenere una Platform comune per diversi team.

PaaS più utilizzati: Amazon Web Services (AWS) Elastic Beanstalk, Google App Engine e IBM Cloud Platform.

Software as a Service (SaaS)

Saas è un'applicazione completa di interfaccia utente messa a disposizione come servizio al consumatore.
Il cliente deve solo configurare alcuni parametri specifici dell'applicazione e gestire gli utenti.
Il provider si occupa di tutta la gestione dell'infrastruttura, delle logiche dell'app, dei deployments e di tutto ciò che concerne la consegna del servizio o prodotto [2]. Solitamente si accede a questo servizio via browser o API.
SaaS più comuni: Google Apps, Dropbox, Cisco WebEx ...

Mobile Backend as a Service (BaaS o mBaaS)

BaaS fornisce i servizi necessari per la parte server side (backend) di un'applicazione web o mobile. Lo scopo di BaaS è di occuparsi di tutte le infrastrutture che stanno dietro al giusto funzionamento di un applicazione, come data storage, object storage, messaggistica ecc...
I servizi più conosicuti sono: FireBase, Amazon Web Services (AWS) Amplify, Back4App e Azure Mobile Apps.

Container as a Service (CaaS)

CaaS è un servizio cloud, relativamente nuovo, che permette agli sviluppatori di caricare, lanciare, controllare e stoppare containers usando chiamate API o una pagina web.
CaaS risolve i problemi di applicazioni sviluppate per una certa piattaforma, la cui esecuzione è ristretta alle sole specifiche proprietarie di questa PaaS.
CaaS rende libere le applicazioni dalle piattaforme aumentandone la portabilità [3].

I servizi più conosciuti sono: Google Container Engine (GKE), Amazon EC2 Container Service (ECS) e Azure Container Service (ACS).

Function as a Service (FaaS)

FaaS è un modello di cloud computing recente che fornisce al cliente una piattaforma su cui eseguire applicazioni e funzioni.
In questo modello vi è un'astrazione completa dei livelli sottostanti. FaaS permette di caricare moduli di funzionalità sul cloud che verranno eseguiti indipendentemente senza occuparsi né delle infrastrutture né dell'ambiente di esecuzione.
Questo servizio, spesso denominato anche "serverless", verrà ampiamente presentato e discusso nel prossimo capitolo.
Servizi esponenti di questo modello sono: Amazon Web Services (AWS) Lambda, Apache Open Whisk e Google Cloud Function.

Capitolo 2

Serverless Computing

In questo capitolo si andranno ad approfondire i concetti di *Function as a Service* (FaaS) e *serverless computing*.
Il servizio Function as a Service permette che l'utente si occupi soltanto di scrivere funzioni, dette *cloud functions*, senza doversi preoccupare dell'ambiente di esecuzione e che queste siano immediatamente eseguibili.
Function as a Service fa parte del paradigma "serverless computing", il quale è un termine coniato dall'industria IT per descrivere un modello di architettura e di programmazione dove applicazioni vengono eseguite sul cloud senza nessun controllo delle risorse sulle quali girano [4].
In questo caso il termine "serverless" è un ossimoro poiché anche se letteralmente significa "senza server", i server sono utilizzati lo stesso per eseguire le funzioni degli utenti, i quali lasciano ogni tipo di amministrazione dell'infrastruttura al cloud provider.
Le *cloud functions*, impacchettate in offerte di tipo FaaS, rappresentano il core del serverless computing, il quale deve parte della sua popolarità anche alle offerte di tipo Backend as a Service (BaaS) che forniscono all'utente database, object storage, messaggistica ecc...
Si può quindi dire che applicazioni create in ottica serverless computing sono composte da funzioni cloud eseguite da servizi FaaS e, se necessaria memorizzazione di dati, da servizi BaaS [5].

Il primo servizio FaaS a prendere piede fu Amazon AWS Lambda a fine 2014 e da allora l'interesse verso il serverless computing, come mostra l'immagine 2.1, è notevolmente aumentato sia da parte delle aziende che da parte degli studi accademici, i quali fino ad ora non avevano mostrato grande attenzione [6].

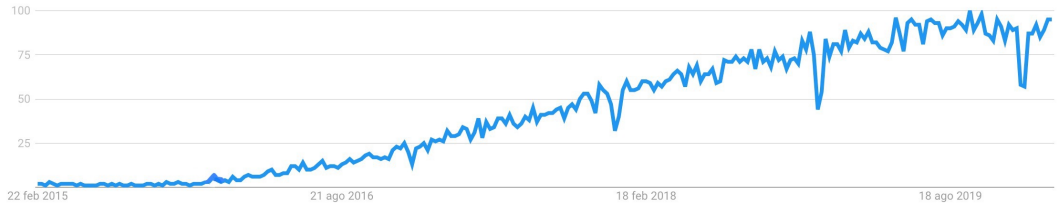

Figura 2.1: Grafico che mostra la popolarità dell'argomento "Serverless" su Google Trends negli ultimi 5 anni.

2.1 Caratteristiche

Come già menzionato, una delle caratteristiche fondamentali delle cloud functions è il poter eseguire funzioni su un ambiente senza preoccuparsi della configurazione di esso. Ci sono anche altre caratteristiche importanti di questo modello, di seguito presentate.

Semplicità

Le funzioni cloud vengono apprezzate dagli utenti perché per eseguirle non serve capire o saper utilizzare l'infrastruttura cloud sottostante l'ambiente di esecuzione. L'utente si limita a caricare la propria funzione e sa che verrà eseguita in qualche data center in giro per il mondo.

Auto-scaling

I servizi serverless per funzioni cloud sono efficienti sia quando vengono richiamate periodicamente con poca frequenza sia quando c'è un utilizzo intensivo parallelo. Il sistema è pronto a scalare in qualsiasi momento e a fornire tutte le risorse computazionali necessarie.

Varietà di Linguaggi

FaaS, di norma, supporta un'ampia varietà di linguaggi di programmazione come Javascript, Java, Python, Go, C# e Swift. Alcune piattaforme supportano meccanismi di estensione per codici scritti in altri linguaggi purché siano impacchettati in un'immagine Docker che li supporti.

Costo

L'utilizzo viene misurato e l'utente paga solo per il tempo e per le risorse utilizzate per l'esecuzione delle funzioni serverless.
Il sistema ha la possibilità di creare centinaia di istanze diverse qualora necessario oppure di scalare fino a zero se non richiesto e in questo caso l'utente non paga nulla.
Gli utenti delle cloud functions potrebbero quindi risparmiare denaro perché le funzioni vengono eseguite solo quando accadono degli eventi specifici (approfondiremo in seguito) e di conseguenza pagano solo per il tempo che queste rimangono in esecuzione.
Di solito c'è un salario che prevede un importo ogni 100ms di esecuzione in base alla memoria massima scelta.

2.2 Funzionamento

Il modello serverless funziona in modo differente: è event-driven. Un evento che viene scatenato esternamente attiva l'esecuzione del codice applicativo attraverso un *trigger*.
Le funzioni vengono eseguite su *containers* che possono essere su macchine virtuali (VM) e server vicini o anche molto lontani.
I container utilizzati sono:

- Stateless: non memorizzano lo stato di un'esecuzione precedente.
- Temporanei: possono essere eseguiti per un periodo di tempo breve e definito dal provider.
- Event-driven: vengono istanziati ed eseguono le funzioni automaticamente quando attivati da eventi.
- Totalmente gestiti da un provider cloud: l'utente non può interferire in alcun modo sull'infrastruttura di esecuzione della funzione.

Il processo che porta all'esecuzione di una funzione serverless viene mostrato 2.2 e si può riassumere con un livello di astrazione elevata coi seguenti passaggi:

1. Il Cloud provider viene notificato o rileva un evento.
2. Un container viene creato e carica il codice della funzione.
3. Quando il container è pronto, gli viene inviato l'evento.
4. Il container esegue la funzione.
5. Finita l'esecuzione la funzione ritorna il risultato dell'elaborazione.
6. Dopo un tempo predefinito di inutilizzo il container viene terminato.

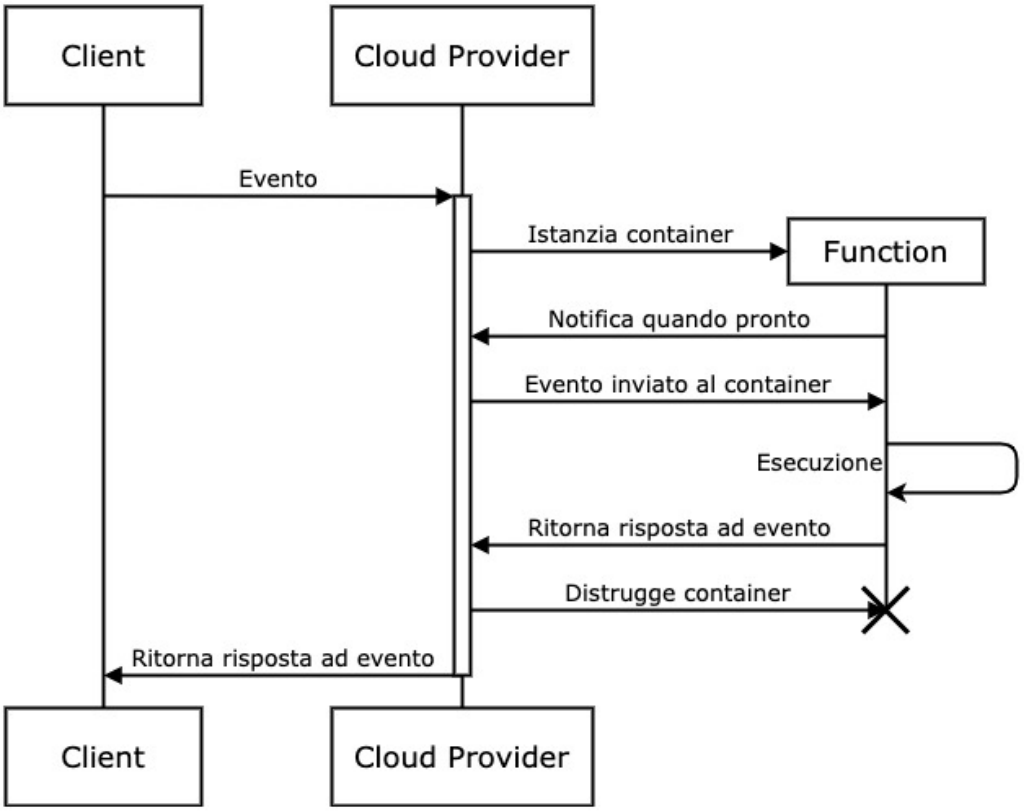

Figura 2.2: Diagramma di sequenza di una computazione FaaS

2.3 Limitazioni

L'architettura serverless ha anche delle limitazioni, dovute alle tecnlologie del momento e anche all'architettura stessa.

- **Esecuzioni lunghe**: le funzioni cloud hanno tempi massimi di esecuzioni predefiniti (es. AWS Lambda 900 secondi, IBM Cloud Functions 600 secondi, Azure functions 300 secondi ...) e questo rende difficile utilizzarle per compiti che necessitano tempi di esecuzioni più lunghi, come upload di grandi moli di dati, machine learning, ecc ...

  Per applicazioni che necessitano operazioni lunghe, al momento sarebbero meglio soluzioni "serverful".

- **Controllo Hardware**: essendo l'hardware gestito interamente dal cloud provider qualora il cliente volesse modificare l'ambiente di esecuzione o l'infrastruttura risulterebbe impossibile in un'architettura serverless.

- **Costi**: il costo è stato anche inserito nei benefici del serverless computing ma si trova anche nei limiti poiché in base alla quantità di funzioni che verranno eseguite il totale potrebbe superare quello di un'altra soluzione cloud come PaaS o IaaS.
  Una soluzione FaaS potrebbe risultare imprevedibile sotto il punto di vista economico qualora non ci fosse certezza sulla quantità di computazione necessaria.

- **Vendor lock-in**: l'applicazione è fortemente dipendente dal fornitore di servizi cloud e ciò comporta significativi cambiamenti qualora si volesse cambiare gestore. Questo problema è noto con il nome di "vendor lock-in".

- **Tempi maggiori alle prime esecuzioni**: l'operazione nella quale viene istanziato un nuovo container per eseguire una funzione viene detta *cold start*. Questo richiede un tempo maggiore di esecuzione rispetto ad un container, non ancora distrutto, nel quale è già stata invocata una stessa funzione almeno una volta. Questo caso viene chiamato *warm start*.

2.4 Use Case

Al momento i maggiori utilizzatori di questa tecnologia cloud sono le startup che cercano di scalare il proprio business limitando le barriere d'entrata (un servizio FaaS non ha alcun costo iniziale).
Serverless è un'ottima scelta anche per tutte quelle applicazioni che non necessitano di esecuzioni continue ma hanno dei picchi periodici di traffico.
La tabella 2.1 mostra le percentuali dei principali utilizzi di serverless nel 2018.
Di seguito vediamo qualche use case che sfrutta il modello serverless.

Percentuale	Use Case
32%	Servizi Web ed API
21%	Data Processing
17%	Integrating 3rd Party Services
16%	Internal tooling
8%	Assistenti virtuali e chatbot
6%	Internet of Things

Tabella 2.1: Tabella che riporta le percentuali di utilizzo del serverless computing in base ad un sondaggio del 2018.[1]

Internet of Things

La natura real-time dell'approccio serverless si sposa benissimo con gli use case dell'Internet of Things (IoT). Spesso i dispositivi come videocamere di sicurezza, sensori del meteo, di temperatura, sentori biometrici non hanno bisogno di comunicare 24 ore al giorno con il server perché è sufficiente comunicare in base a certi eventi e questo è perfetto per un servizio serverless.

[1] https://serverless.com/blog/2018-serverless-community-survey-huge-growth-usage/

Assistenti virtuali e chatbot

I servizi di messaggistica automatizzata (chatbot) e gli assistenti virtuali necessitano di risposte immediate per l'utente. Un sistema di questo genere sviluppato in FaaS permette di ottenere risposte veloci sia per poche decine che per alcune migliaia di utenti collegati. Questo è possibile grazie alla scalabilità automatica fornita da questo servizio cloud.

Creazione di thumbnail

Le applicazioni cloud che contengono molte immagini necessitano di una elaborazione per diminuire drasticamente i tempi di caricamento: la creazione di thumbnails. Una thumbnail è un'immagine di dimensioni ridotte rispetto all'originale adattata al tipo di display sulla quale viene visualizzata.
Attraverso un servizio serverless è possibile automatizzare la creazione ad hoc di queste immagini ridimensionate in base alle caratteristiche del dispositivo.

Capitolo 3

OpenWhisk

L'introduzione da parte di Amazon Web Services di Lambda nel novembre 2014 ha cambiato il panorama del cloud computing e da allora tutti i maggiori cloud provider hanno iniziato a sviluppare un servizio FaaS. Tra questi c'è IBM che ad inizio 2015 ha cominciato un progetto denominato Whisk[1].
Circa un anno dopo IBM ha reso disponibile il codice del progetto, che ha preso il nome di OpenWhisk. Ad oggi il codice è disponibile sulla pagina GitHub di Apache sotto licenza una open-source Apache-License 2.0.
Dal 2016 OpenWhisk è la principale alternativa open-source serverless a AWS Lambda che però necessita di essere installata su un cluster di macchine o VM.

3.1 Utilizzo

OpenWhisk è una piattaforma serverless che esegue funzioni in risposta ad eventi. Gli eventi che possono scatenare l'invocazione di una funzione su OpenWhisk possono essere di vario genere: un cambiamento su un Database, una chiamata API, un evento di un sensore IoT, una richiesta HTTP, ecc...
Gli *eventi* sono collegati ad un *feed* che si occupa di far scattare il *trigger* quando rileva un evento. Il trigger in base alle *rules* fa eseguire una *action*.
Ora si analizzeranno nel dettaglio i principali componenti nel workflow di OpenWhisk.

[1] in inglese significa frusta e quindi rappresenta qualcosa che si muove molto velocemente.

Evento

Un evento sorgente sta alla base, fa partire tutto il processo di elaborazione. Un evento su OpenWhisk viene scatenato esternamente, esempi possono essere:

- Un nuovo messaggio nelle Message Queues
- Cambiamenti sui Databases o Document Stores
- Interazioni con siti o applicazioni web
- Invocazioni di servizi API
- Inoltro di dati da sensori IoT

Feed

Un feed è un flusso di eventi esterni che appartengono ad un trigger. I feed possono essere contenuti in pacchetti preinstallati, pacchetti installabili e in pacchetti personalizzati.
Un feed è controllato da un'action feed che gestisce la creazione, l'eliminazione, la pausa e il ripristino del flusso di eventi che formano un feed. L'action feed interagisce con i servizi esterni che generano gli eventi, utilizzando un'API REST che gestisce le notifiche.

Trigger

I trigger sono nomi associati a classi di eventi. I trigger vengono attivati dagli eventi che li compongono. Ogni volta che viene richiamato un trigger si crea un *activation ID* in corrispondenza dell'action invocata.

Rule

Le rules servono per associare un trigger ad una action. Una volta creata una rule tra un trigger T e una action A quando arriva un evento del trigger T verrà richiamata la action A.

Action

In OpenWhisk le "funzioni cloud" vengono chiamate actions.
Una action può essere una funzione stateless scritta in un determinato linguaggio tra quelli supportati (JavaScript, Python, Java, Scala, Go, C#, Ruby, Swift e .NET) oppure un contenitore Docker eseguibile. Una action viene richiamata quando c'è un evento esterno oppure quando viene esplicitamente invocata da un utente.

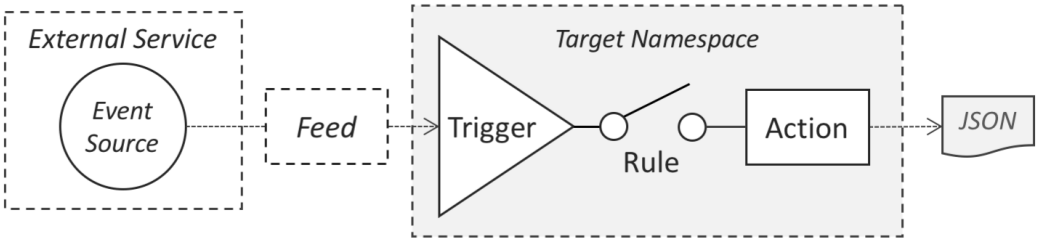

Figura 3.1: Modello di programmazione di OpenWhisk

3.2 Gestione di un'invocazione

In questa sezione viene presentato brevemente il funzionamento interno di OpenWhisk, alla ricezione di un evento o invocazione, con tutti i suoi componenti. L'immagine 3.1 mostra come viene gestita internamente un'invocazione.

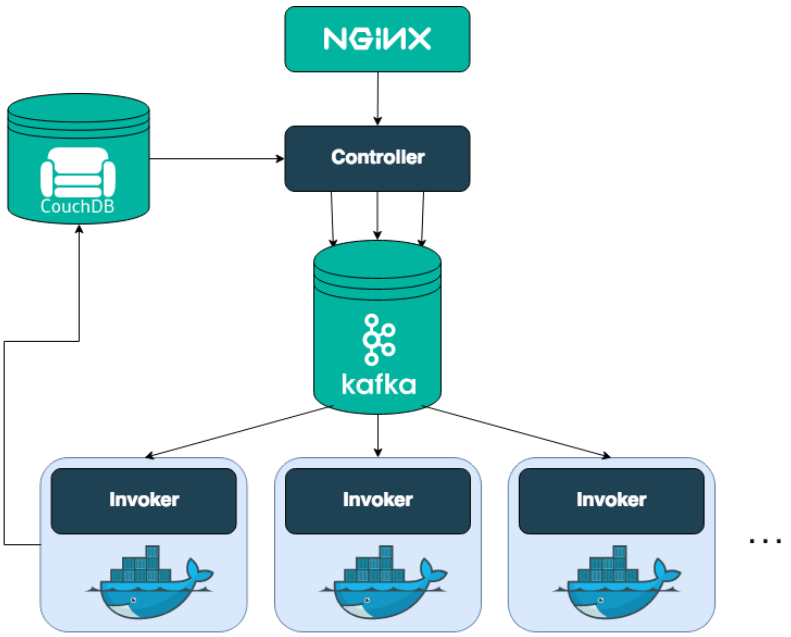

Figura 3.2: Modello astratto dell'architettura di OpenWhisk

OpenWhisk è un sistema restful perciò ogni invocazione viene convertita in una richiesta HTTPS al nodo "edge". L'edge è il web server e reverse proxy **nginx**. *Nginx* implementa il supporto per il protocollo HTTPS, quindi fornisce tutti i certificati richiesti. *Nginx* poi inoltra la richiesta al **controller**.
Prima di eseguire l'action, il *controller* verifica se è possibile inizializzarla ed eseguirla correttamente:

1. È necessario autenticare la richiesta;

2. Verificare se la richiesta è autorizzata.

3. La richiesta contenere alcuni parametri aggiuntivi forniti come parte della configurazione dell'action.

Per eseguire tutti questi passaggi il *controller* consulta il database **CouchDB** dove l'action è contenuta. Quando l'azione viene validata è pronta per essere eseguita e viene inviata al **load balancer**. Il compito del *load balancer* è di bilanciare il carico tra i vari esecutori nel sistema, chiamati "**invokers**". Le actions vengono eseguite runtime.
Il *load balancer* mantiene le istanze disponibili di una action runtime, riutilizzandole se disponibili o creandone di nuove se necessario.
Ora il sistema è pronto per invocare la action, ma non può semplicemente inviarlo all'*invoker* scelto perché potrebbe essere occupato con un'altra azione oppure potrebbe essersi bloccato. Si deve quindi considerare che potrebbero non esserci le risorse necessarie per eseguire immediatamente la action. In questo caso si devono "bufferizzare" le invocazioni.
OpenWhisk utilizza **Kafka**, un sistema di messaggistica ad alte prestazioni in grado di archiviare le richieste fino a quando non sono pronte per essere eseguite. La richiesta viene convertita in un messaggio indirizzato all'*invoker* che il *load balancer* ha scelto per l'esecuzione.
Un'invocazione di una action viene effettivamente trasformata in una richiesta HTTPS a *nginx* che poi internamente diventa un messaggio per *Kafka*. Ogni messaggio inviato ad un *invoker* ha un identificatore chiamato **activation ID**.
Una volta che il messaggio è stato messo in coda su *Kafka* ci sono due possibili invocazioni: una bloccante e una non-bloccante.
Per un'invocazione non-bloccante, l'*activation ID* viene inviato come risposta finale della richiesta al client. In questo caso il client potrà trovare il risultato dell'invocazione una volta che l'action viene ultimata.
Invece per una chiamata bloccante, la connessione rimane aperta: il *controller* attende il risultato dall'action e invia direttamente il risultato al client.
In OpenWhisk l'*invoker* esegue le actions in ambienti isolati forniti dal contenitore **Docker**.

Docker utilizza immagini per creare i container che eseguiranno le actions. OpenWhisk fornisce una serie di immagini *Docker* che includono il supporto per i vari linguaggi. Una volta terminata l'elaborazione dell'action, i risultati vengono poi salvati di nuovo sul database *CouchDB*.

Capitolo 4

Progetto

Il lavoro di questa Tesi ha contribuito alla Tesi di Giuseppe De Palma, studente della Laurea Magistrale in Informatica presso l'Università di Bologna. Entrambi gli elaborati sono stati seguiti dal Professore ordinario Gianluigi Zavattaro del dipartimento di *Informatica* dell'*Università di Bologna*, relatore delle Tesi.
Al progetto hanno collaborato anche il Professor Jacopo Mauro e il Dottor Saverio Giallorenzo, correlatori degli elaborati, entrambi parte del dipartimento di "*Mathematics and Computer Science*" presso la *University of Southern Denmark* (SDU).

4.1 Obiettivo

Il progetto è volto a dimostrare come potrebbe migliorare l'efficienza delle attuali piattaforme serverless tenendo conto delle *locality* durante lo scheduling delle funzioni.
La piattaforma scelta su cui operare è stata Apache OpenWhisk, per via della sua natura open-source.
Lo scopo del progetto è dimostrare l'importanza delle locality durante la fase di scheduling delle actions di OpenWhisk e proporre uno scheduler (load balancer) configurabile e alternativo a quello di default, che permetta di definire

delle preferenze tra gli invokers del cluster per determinate actions.
Lo scheduler è configurabile attraverso un apposito file che permette di specificare delle politiche di preferenza degli invokers per determinate actions.
Questa soluzione è adatta ad utenti che vogliono costituire un cluster ibrido con diversi nodi in data center differenti.
L'utilizzo di questa soluzione però differisce dal classico utente target di un servizio serverless, che spesso è un utente che non vuole pensare a configurazioni e manutenzioni dei server. Tuttavia, in questo caso deve eseguire il deploy di OpenWhisk sulle sue macchine e deve essere in grado di capire e saper scrivere un file di configurazione adatto alla propria situazione.
Lo sviluppo della mia tesi, oltre al supporto fornito a Giuseppe, si è concentrata soprattutto nel testing della *data locality*.

4.2 Contributo

Oltre ai test sulla data locality, focus di questa tesi, ho fornito supporto a Giuseppe De Palma nella realizzazione delle sue prove.
In particolare:

- inizialmente ho realizzato diverse actions (in JavaScript) per testare il sistema ed effettuare le prime prove;
- ho realizzato uno script Python che passati in input il nome dell'action e i suoi parametri (se presenti) e il numero di invocazioni esegue l'action per il numero di volte specificato e raccoglie tutti i tempi di esecuzione (salvandoli in un file) e restituisce la media dei tempi;
- ho inserito e preparato i documenti per i test sul database CouchDB installato in una VM di Giuseppe;
- ho realizzato le actions sviluppate in JavaScript per richiedere i documenti e gli allegati su CouchDB.

4.3 Locality

Per ridurre la latenza ed i tempi di esecuzione delle cloud functions un load balancer dovrebbe prendere decisioni su dove schedulare le funzioni in base a diversi tipi di *locality* [7].

4.3.1 Session Locality

Innanzitutto, lo scheduler deve considerare la session locality: se un'invocazione ad una Lambda function fa parte di una sessione di lunga durata con connessioni TCP aperte, sarà utile eseguire l'action sull'invoker in cui vengono mantenute le connessioni TCP in modo tale che il traffico non debba essere deviato attraverso un proxy.
Durante una singola sessione vi è spesso uno scambio bidirezionale di dati tra client e server; questo scambio è in genere facilitato da un WebSocket, o da lunghi sondaggi. Questi protocolli sono difficili per le cloud functions perché si basano su connessioni TCP di lunga durata.
Se le connessioni TCP sono gestite all'interno di un invoker serverless e questo è inattivo, i costi per il cliente dovrebbero tener conto del fatto che il provider incorrerà in un sovraccarico di memoria ma non consumerà CPU.
In alternativa, se la piattaforma fornisce la gestione delle connessioni TCP al di fuori degli invoker, è necessario prestare attenzione a fornire una nuova chiamata a funzione con le connessioni necessarie avviate da precedenti invocazioni.

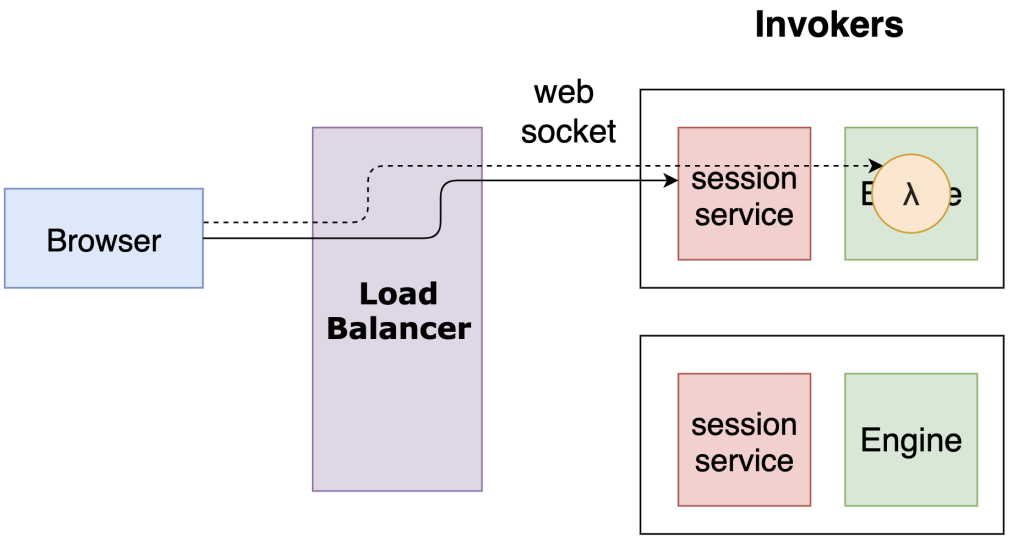

Figura 4.1: Grafico che mostra il principio della session locality. Un browser che ha già invocato una action λ, qualora la rieseguisse dovrebbe comunicare direttamente con l'engine dell'action, invece che ripassare per il proxy.

4.3.2 Code Locality

In secondo luogo, la code locality diventa più difficile. Uno scheduler consapevole del fatto che due diversi invoker utilizzano fortemente gli stessi pacchetti, può prendere decisioni migliori sul posizionamento delle actions. Inoltre, uno scheduler potrebbe voler indirizzare le richieste in base ai vari gradi di ottimizzazione dinamica raggiunti su vari lavoratori.
Schedulare un'action, che utilizza un determinato pacchetto o libreria, su un'invoker che ha già caricato quella libreria o pacchetto in memoria permette di ridurre i tempi d'esecuzione e riduce la latenza.
Una soluzione è quella di mantenere in memoria i pacchetti nel worker node (nodo che esegue la funzione) invece di impacchettarlo insieme alla funzione. Un algoritmo di schedulazione "sensibile" al pacchetto che tenta di assegnare funzioni che richiedono il pacchetto al nodo su cui è già stato caricato, è proposto nel paper [8]. Questo algoritmo aumenta la frequenza di presenze del

pacchetto nella cache e, di conseguenza, riduce la latenza delle funzioni cloud.

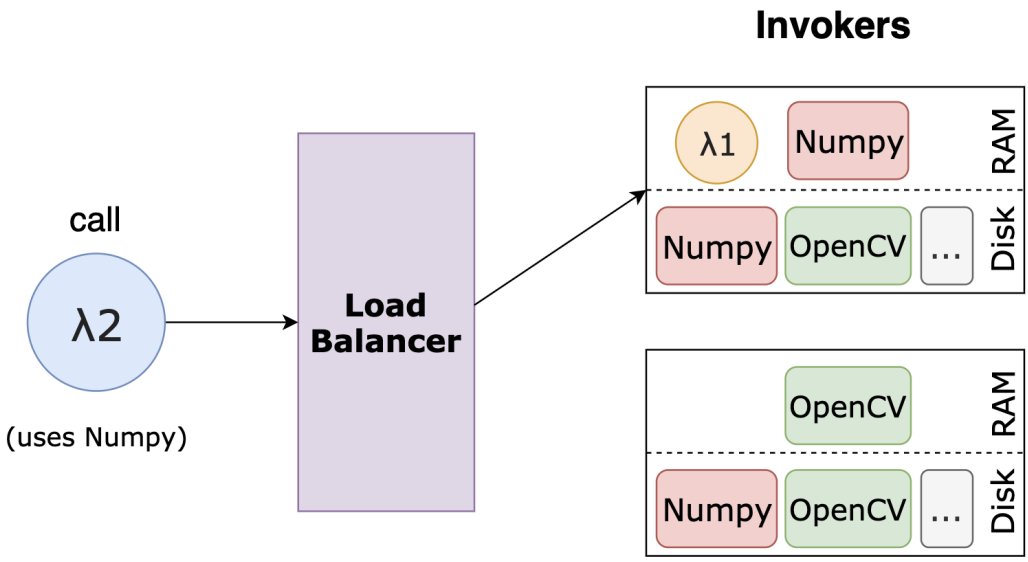

Figura 4.2: Grafico che mostra il principio della code locality: una action $\lambda 2$ che utilizza una libreria "Numpy" dovrebbe essere schedulato sull'invoker che ha già caricato in memoria quel pacchetto.

4.3.3 Data Locality

In terzo luogo, la data locality è importante per eseguire le funzioni che usano database e indici di grandi dimensioni.
Lo scheduler dovrà anticipare quale query o database utilizzerà una particolare invocazione Lambda o quali dati leggerà.

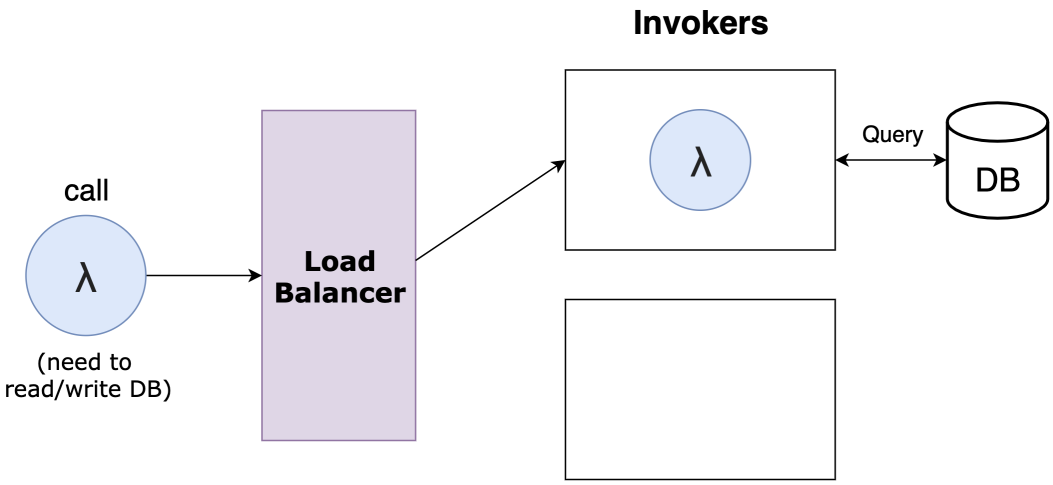

Figura 4.3: Grafico che mostra il principio della data locality: una action λ che durante l'esecuzione necessita di operazione in lettura o scrittura sul database DB dovrebbe essere schedulato nel primo invoker, poiché si trova fisicamente più vicino del secondo e in questo modo si riduce la latenza. DB potrebbe essere nello stesso data center del primo invoker.

Anche quando lo scheduler sa a quali dati accederà una Lambda function e dove risiedono le repliche dei dati, un'ulteriore comunicazione con il database può essere utile per scegliere la migliore replica, nel caso di database replicati o distribuiti.
Quando un'action deve comunicare con un database durante la sua esecuzione per il principio di data locality si dovrebbe cercare di eseguire la funzione vicino ai dati.

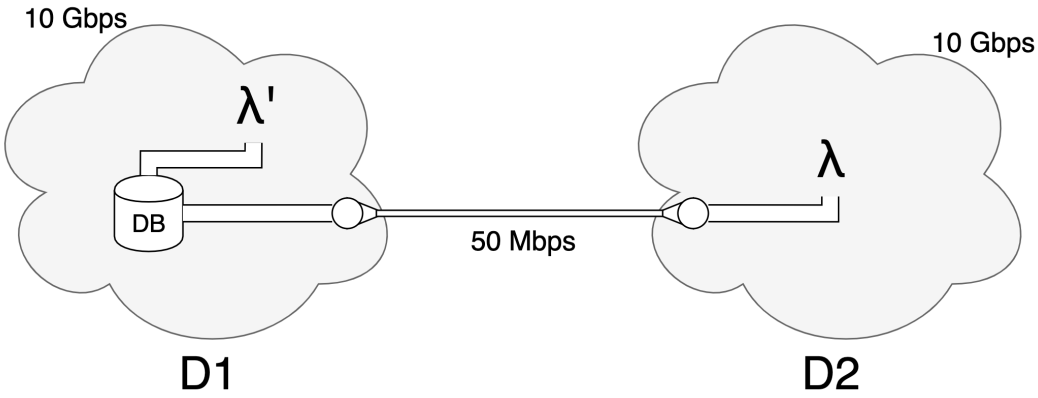

Figura 4.4: Modello che simula l'esecuzione di una function λ su due invoker in data center diversi che devono accedere ad uno stesso database DB. λ' avrà la possibilità di comunicare con velocità e banda molto superiore rispetto a λ poiché si trova nello stesso data center.

La vicinanza spaziale tra il database (o la replica con cui si comunica) e l'invoker dell'action può portare una riduzione dei tempi di esecuzione importante perché più i dati sono vicini minor latenza avrà il sistema. L'incremento massimo di efficienza si potrebbe avere nel caso in cui il database risiedesse nello stesso data center dell'invoker dell'action (figura 4.4).

4.4 Ambiente utilizzato

Per eseguire le attività si è utilizzato **IBM Cloud Functions** come piattaforma serverless, poiché è un servizio completamente implementato su OpenWhisk, con la differenza che l'infrastruttura è gestita da IBM e non va realizzato un proprio cluster di macchine. Sono disponibili numero di invocazioni e potenza limitate nella versione gratuita utilizzata.
Come servizio per una base di dati si è utilizzato **Cloudant**, database noSQL proprietario di IBM sviluppato sulla base del progetto Apache CouchDB.

4.4.1 IBM Cloud Functions

IBM Cloud functions è una piattaforma Function as a Service proprietaria completamente basata su OpenWhisk, che prima di diventare un progetto open-source era di proprietà proprio di IBM, la quale lo donò ad Apache Foundation nel 2016.
IBM Cloud Functions impremenata OpenWhisk e si occupa della gestione delle infrastrutture e dell'offerta di altri servizi. Essendo basato su OpenWhisk, l'utilizzo delle funzioni (chiamate *actions* in questo contesto), eventi e trigger è analogo.

Configurazione

Per iniziare ad utilizzare IBM Cloud Functions è stato necessario eseguire alcuni passaggi, tra cui:

- creazione di un IBM Cloud account;
- download ed installazione della CLI di IBM Cloud Functions;
- login IBM Cloud da terminale (Mac OS nel mio caso);
- installazione dei plug-in necessari ad IBM Cloud Functions.

Durante la creazione dell'ambiente IBM offre la possibilità di scegliere la locazione del "*Master node*" dove eseguire le action tra le seguenti città:

- Londra, Regno Unito;
- Francoforte, Germania;
- Washington DC, Stati Uniti;
- Dallas, Stati Uniti;
- Chennai, India;
- Seul, Corea del Sud;

- Tokio, Giappone;
- Sidney, Australia.

Una volta configurato l'ambiente ed effettuato gli accessi necessari, è possibile iniziare ad usare i comandi di OpenWhisk per creare *actions*, *triggers* e *rules.*

Utilizzo

Per iniziare ad utilizzare le actions è necessario creare un *package* (pacchetto) in cui poterle inserire. Successivamente è possibile creare una action, inserendola in un pacchetto e caricando il file che corrisponderà ad essa.
Alla creazione è possibile associare dei parametri, alcuni validi come impostazione della action (–web per esempio fornisce un URL valido per invocare l'action), altri come parametri di esecuzione della stessa (parametri in input alla funzione, preceduti da -p).

Limiti di utilizzo

IBM Cloud Functions fornisce un determinato numero di invocazioni e memoria gratuiti ogni mese[1] e in più 400'000 GB di allocazione memoria *una tantum.*
La tariffa di base è 0,000017$ al secondo di esecuzione, per ogni GB di memoria allocata.
Quando si crea una action viene impostato di default 60 secondi di timeout (tempo dopo il quale viene arrestata l'action) e 256MB di memoria massima allocata. Il tempo di timeout è espandibile al massimo fino a 600 secondi e la memoria è espandibile sino a 2048MB, ovvero 2GB, di RAM massima allocata per action.

[1] con un tempo di esecuzione medio di 250ms e memoria massima di 256MB le invocazioni gratuite ammontano a circa 5000 al mese

4.4.2 Cloudant

Cloudant è un prodotto software IBM, principalmente fornito come servizio cloud. Fornisce un servizio di database distribuito non relazionale che si basa sul progetto CouchDB, supportato da Apache, e sul progetto open source BigCouch.
Cloudant scala i database sul framework CouchDB e fornisce supporto agli strumenti di hosting, amministrativi, analitici e commerciali per CouchDB e BigCouch.
Il servizio CouchDB distribuito di Cloudant viene utilizzato allo stesso modo di CouchDB, con l'ulteriore vantaggio di distribuire i dati in modo ridondante su più macchine.
Ho scelto Cloudant come database perché la sua struttura non relazionale si adatta bene ad applicazioni veloci come quelle create secondo il paradigma serverless, per la sua natura di struttura già gestita che non necessita di configurazioni particolarmente lunghe e per la possibilità di scelta della locazione della struttura dati.
I documenti memorizzati sono in formato JSON e c'è la possibilità di allegare dei file ad ogni documento.
Le città risultano le stesse delle possibili scelte del master node di Cloud Functions e ciò fornisce un'ottima opportunità per poter verificare ed eseguire test sulla data locality.

Capitolo 5

Test su data locality

In questo capitolo vengono descritti i test svolti al fine di dimostrare l'importanza della data locality e si presentano i relativi risultati.

5.1 Descrizione

Nel capitolo precedente è stato descritto l'ambiente in cui si sono svolti i test: IBM Cloud Function come piattaforma serverless e Cloudant come database non relazionale.
Alla creazione del profilo per IBM Cloud Function è stata scelta come località di esecuzione delle actions **Londra**.
Potendo avere solo un database gratuito nello stesso momento, se ne è creato uno per volta, utilizzando tutte le locazioni disponibili che offre IBM e si sono eseguiti tutti i test necessari.
Per i test è stata utilizzata sempre la stessa action, scritta in Node.js, che ritorna una Promise facendo una richiesta di un documento ad un database, entrambi passati come parametri. L'action si conclude quando riceve il documento richiesto, ovvero un documento JSON di piccole dimensioni (circa 2 KB) con in allegato due file, uno da 5 MB e uno da 10 MB (limite massimo per un allegato nella versione gratuita di Cloudant), da poter richiedere separatamente.

Per ottenere test veritieri le action sono state invocate 200 volte[1] per ogni documento e per ogni database.
Per ottenere i tempi di esecuzione è stato eseguito uno script Python che invoca una action per un numero di volte specificato (200 per i miei test), raccoglie tutti gli *activation ID* e ottiene tutti i tempi desiderati.
Ultimate tutte le richieste, lo script restituisce una media dei tempi di esecuzione e una media che rimuove il 5% dei peggiori valori ottenuti[2].
Ricapitolando, le action sono state eseguite nei data center di Londra di IBM, è stato poi allocato un database per ogni località disponibile al fine di poter verificare quanto la vicinanza tra il database e la action potesse influire sulla latenza ed i tempi d'esecuzione della stessa.
I test eseguiti sono stati i seguenti:

- 200 action che richiedono un documento JSON da circa 2KB;
- 200 action che richiedono un file da 5MB;
- 200 action che richiedono un file da 10MB;

ognuno eseguito per ogni database nelle seguenti città del mondo:

- Londra, Regno Unito;
- Francoforte, Germania;
- Washington DC, Stati Uniti;
- Dallas, Stati Uniti;
- Chennai, India;
- Seul, Corea del Sud;

[1]Numeri di invocazioni più alte non modificavano i risultati dei test.

[2]Una rimozione di una percentuale dei peggiori valori ottenuti permette di eliminare i casi in cui ci sia un *cold start* del container di esecuzione e ottenere una media fatta di valori con casi simili.

- Tokio, Giappone;
- Sidney, Australia.

5.2 Risultati

In questa sezione vengono mostrati e discussi i risultati ottenuti.

5.2.1 Documento JSON

Di seguito la tabella dei risultati dei test eseguiti richiedendo un documento JSON da 2KB su vari database Cloudant.

Locazione DB	distanza	n richieste	media	media del 95%
Londra	0 km	200	**17.51 ms**	**16.91 ms**
Francoforte	641 km	200	66.17 ms	58.68 ms
Washington DC	5896 km	200	299.06 ms	297.99 ms
Dallas	7635 km	200	454.02 ms	449.07 ms
Chennai	8198 km	200	554.00 ms	548.66 ms
Seul	8796 km	200	903.62 ms	898.24 ms
Tokio	9557 km	200	928.23 ms	924.25 ms
Sidney	16961 km	200	1006.80 ms	1002.72 ms

Tabella 5.1: Tabella con i tempi di esecuzione delle actions che richiedono un documento JSON

I risultati della tabella 5.1 con un file di piccole dimensioni si possono considerare molto soddisfacenti.
All'aumentare della distanza tra il worker node e il database, la latenza aumenta sensibilmente.
Il tempo medio di lettura del documento a Sidney rispetto al tempo di lettura

dello stesso documento a Londra è cinquantanove volte superiore.
L'aumento più vertiginoso si ha tra il database locato a Chennai e quello di Seul, che ad un aumentare di soli 700 km da Londra, si ha quasi un raddoppio di latenza, come si nota anche dal grafico sottostante.

5.2.2 5MB

Di seguito la tabella dei risultati dei test eseguiti richiedendo un file da 5MB su vari database Cloudant.

Locazione DB	distanza	richieste	media	media del 95%
Londra	0 km	200	**376.27 ms**	**358.02 ms**
Francoforte	641 km	200	421.72 ms	411.25 ms
Washington DC	5896 km	200	1116.93 ms	1109.18 ms
Dallas	7635 km	200	1590.17 ms	1563.08 ms
Chennai	8198 km	200	1817.13 ms	1772.49 ms
Seul	8796 km	200	3079.10 ms	3013.76 ms
Tokio	9557 km	200	3082.01 ms	3063.39 ms
Sidney	16961 km	200	3260.02 ms	3218.12 ms

Tabella 5.2: Tabella con i tempi di esecuzione delle actions che richiedono un file da 5MB dai diversi database.

I risultati ottenuti nel richiedere un file di dimensioni intermedie (5MB) riflettono in proporzione minore i risultati precedentemente analizzati. La differenza di latenza tra il database Cloudant locato a Londra e a Sidney è di circa nove volte superiore, che si traduce in una differenza di due secondi e mezzo.
Per distanze minori, come Francoforte, invece rimane una latenza maggiore di circa $50ms$, ovvero il 15%.
Il grafico sottostante mantiene le stesse proporzioni del grafico ottenuto dalle richieste del documento JSON.

5.2.3 10MB

Analizzando la tabella 5.3 si può notare come, rispetto alla richiesta del file da 5MB (tabella 5.2), ci sia stato un aumento nei tempi di circa 300 *ms* (probabilmente il tempo necessario ad inviare un file di dimensione doppia rispetto al precedente attraverso una comunicazione già aperta).
In questo test è presente l'unico caso in cui un database più lontano (Tokio) ha una latenza minore rispetto ad uno più vicino (Seul). Una possibile causa di questo fenomeno è l'instradamento dei pacchetti utilizzato lungo le pipeline internet mondiali.

Locazione DB	distanza	richieste	media	media del 95%
Londra	0 km	200	**696.42 ms**	**676.18 ms**
Francoforte	641 km	200	722.53 ms	700.59 ms
Washington DC	5896 km	200	1494.47 ms	1463.82 ms
Dallas	7635 km	200	1959.38 ms	1929.02 ms
Chennai	8198 km	200	2025.46 ms	2015.96 ms
Seul	8796 km	200	3526.36 ms	3472.56 ms
Tokio	9557 km	200	3474.86 ms	3423.72 ms
Sidney	16961 km	200	3706.87 ms	3623.89 ms

Tabella 5.3: Tabella con i tempi di esecuzione delle actions che richiedono un file da 10MB dai diversi database.

Di seguito sono mostrate le rappresentazioni grafiche dei risultati ottenuti e si può facilmente notare una similitudine di "forma" tra di essi.

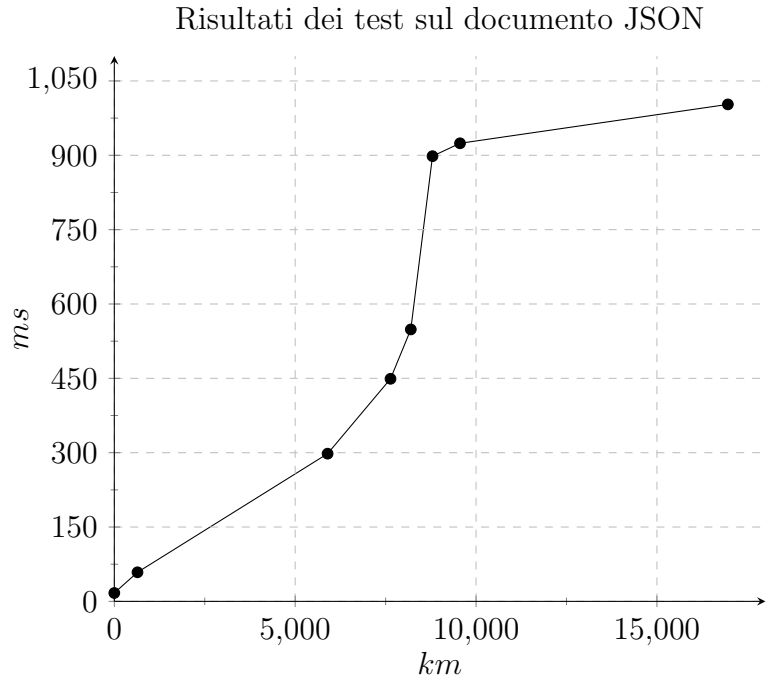

Figura 5.1: Grafico che mostra i risultati della tabella 5.1 con la distanza da Londra in km nell'asse x e la latenza in ms nell'asse y.

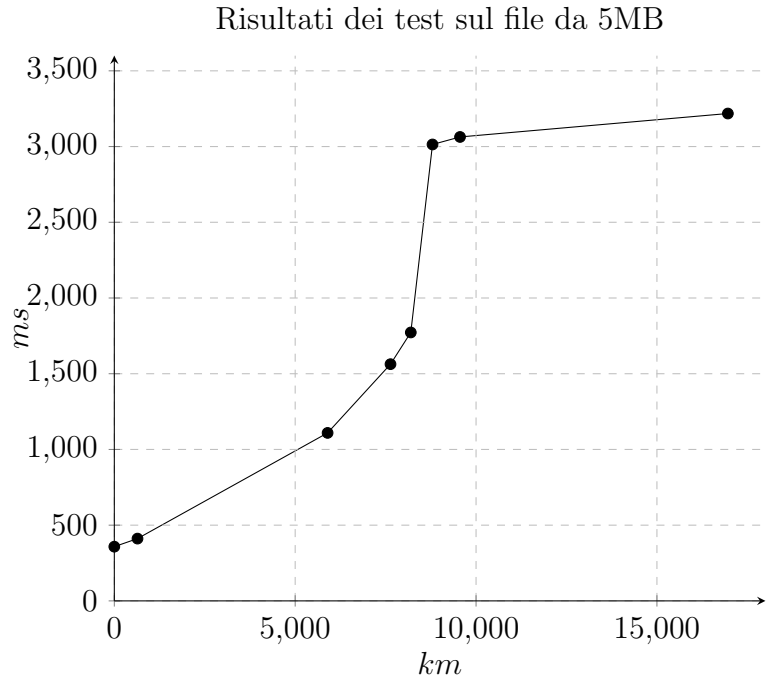

Figura 5.2: Grafico che mostra i risultati della tabella 5.2 con la distanza da Londra in km nell'asse x e la latenza in ms nell'asse y.

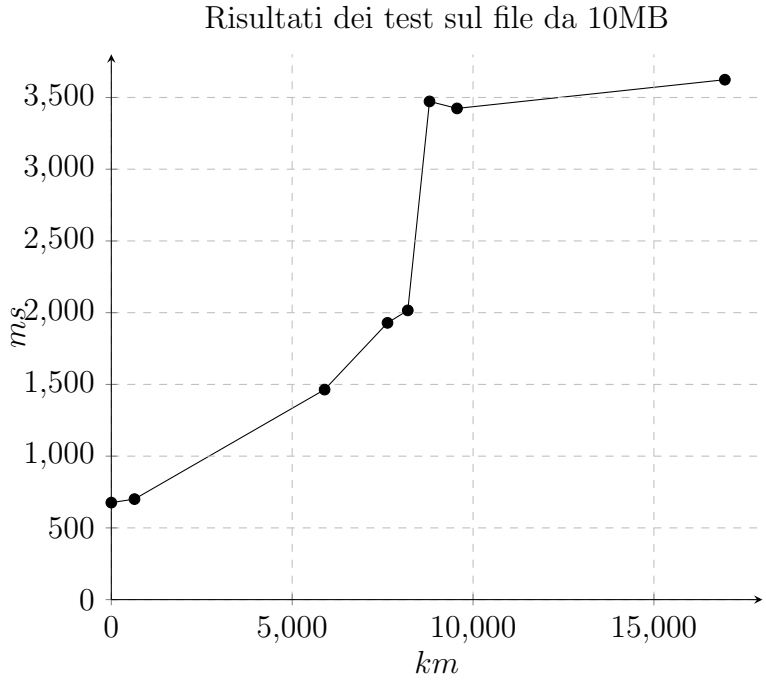

Figura 5.3: Grafico che mostra i risultati della tabella 5.3 con la distanza da Londra in km nell'asse x e la latenza in ms nell'asse y.

Conclusioni

Il serverless computing è una tecnologia non ancora largamente utilizzata ma che sta acquisendo sempre più popolarità e ciò comporta anche un maggior interesse da parte di aziende e comunità scientifica.
I risultati ottenuti in questo elaborato, per verificare la data locality in un ambiente basato da Open Whisk, si sono rilevati molto soddisfacenti.
Dall'esito dei test si evince un possibile importante aumento di prestazioni derivato dal poter spostare la funzione nella stessa zona dei dati.
Un load balancer in grado di tener conto della data locality e di ottimizzare l'esecuzione delle funzioni in base alla vicinanza dei database permette di ridurre sensibilmente la latenza.
Integrare uno *smart* scheduler in un deployment di OpenWhisk su proprie infrastrutture potrebbe non essere semplice, ma porta ottimi risultati, come mostrato da Giuseppe De Palma nella sua tesi.
Riguardo ai provider terzi, come AWS, IBM, Google e altri, l'inserimento di uno scheduler in grado di tener conto delle locality potrebbe produrre grossi benefici qualora decidessero di integrare la tecnologia serverless con i servizi di storage che già offrono, schedulando le funzioni in *edge node* il più vicino possibile ai dati che utilizzano. In questo modo i gestori potrebbero accrescere l'interesse da parte degli utenti ancora scettici verso questo genere di servizi.
Infine, un approccio di questo tipo, in grado di ridurre sensibilmente i ritardi ed i tempi di esecuzione, permette di ottenere un risparmio economico derivante dal tipo di tariffazione tipicamente utilizzato dai privider.
Oltre alla data locality sono importanti anche code e session locality.

Per quanto riguarda la code locality, un algoritmo è già stato proposto in soluzione al problema [8] mentre nessuna soluzione è stata ancora presentata per la session locality.
Interessante capire quanto l'integrazione nello scheduler di politiche *smart*, che tengono conto di tutte le locality, possa impattare questa tecologia e fornire miglioramenti realmente tangibili.

Bibliografia

[1] Fang Liu, Jin Tong, Jian Mao, Robert Bohn, John Messina, Lee Badger, and Dawn Leaf. Nist cloud computing reference architecture. *NIST special publication*, 2011.

[2] Michael J Kavis. *Architecting the cloud: design decisions for cloud computing service models (SaaS, PaaS, and IaaS)*. John Wiley & Sons, 2014.

[3] Mohamed K Hussein, Mohamed H Mousa, and Mohamed A Alqarni. A placement architecture for a container as a service (caas) in a cloud environment. *Journal of Cloud Computing*, 8(1):7, 2019.

[4] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. *Serverless Computing: Current Trends and Open Problems*, pages 1–20. 12 2017.

[5] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley, Feb 2019.

[6] Joseph M. Hellerstein, Jose M. Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang

Wu. Serverless computing: One step forward, two steps back. *CoRR*, abs/1812.03651, 2018.

[7] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, June 2016. USENIX Association.

[8] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

[9] Michele Sciabarrà. *Learning Apache OpenWhisk*. O'Reilly, 2019.

Ringraziamenti

Ringrazio innanzitutto il Prof. Zavattaro per avermi dato la possibilità di realizzare questo elaborato riguardante un'interessante tecnologia in rapida espansione. Un sentito ringraziamento anche al Dott. Saverio Giallorenzo e al Prof. Jacopo Mauro, correlatori di questa tesi, per il supporto fornitomi.
Infine ringrazio la mia famiglia, i miei amici e tutti coloro che mi hanno sostenuto in questo percorso di studi accademici.

Andrea