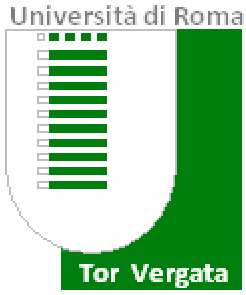
**Università degli Studi di Roma "Tor Vergata"**



**MACROAREA DI INGEGNERIA**

**DIPARTIMENTO DI INGEGNERIA DELL’IMPRESA**

**Tesi di Laurea in Ingegneria Gestionale**

Analisi delle prestazioni di un sistema di virtualizzazione per telecomunicazioni basato su container

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Relatore |  |  |  |  |  | Candidato |
| Prof. Franco Mazzenga  Correlatore  Prof. Romeo Giuliano |  |  |  |  |  | Giordano Galli |
|  |  |  |  |  |  |  |

Anno accademico 2017/2018

Sommario

[Introduzione 4](#_Toc5982601)

[CAPITOLO 1 6](#_Toc5982602)

[La virtualizzazione software 6](#_Toc5982603)

[1.1 Virtualizzazione e macchine virtuali 6](#_Toc5982604)

[1.2 Software container definizione e comparazione con VM 9](#_Toc5982605)

[1.3 Funzionamento di Docker 12](#_Toc5982606)

[1.3 Virtualizzazione nelle reti di telecomunicazione 16](#_Toc5982607)

[1.2.1 Network Function Virtualization 16](#_Toc5982608)

[1.2.2 Software Defined Networking 18](#_Toc5982609)

[CAPITOLO 2 18](#_Toc5982610)

[Sperimentazioni sull’uso di container 18](#_Toc5982611)

[2.1 Set-Up Hardware e Software 19](#_Toc5982612)

[2.2 Testing e analisi dei dati 20](#_Toc5982613)

[2.2.1 Primo Test 20](#_Toc5982614)

[2.2.2 Secondo Test 29](#_Toc5982615)

[CAPITOLO 3 Conclusioni e sviluppi futuri dell’elaborato 39](#_Toc5982616)

[3.1 Kubernetes e Pokemon Go! 39](#_Toc5982617)

[3.1.2 Architettura e funzionamento di kubernetes 41](#_Toc5982618)

[3.1.3 Possibili Sviluppi futuri dell’elaborato 44](#_Toc5982619)

[Bibliografia 46](#_Toc5982620)

# Introduzione

La quantità di dati digitalizzati raddoppia ogni anno, oltre il 98% di tutte le informazioni del mondo è disponibile in formato digitale e in parte accessibile da chiunque. Bernard Marr in un articolo scritto per Forbes “*How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read*” (Marr, 2018) riassume una serie di statistiche interessanti per capire il fenomeno e il volume indescrivibile di dati che collezioniamo continuamente, come comunità mondiale, in formato digitale. Di seguito alcuni dati significativi:

* Circa 3.7 miliardi di persone hanno accesso a internet sull’intero pianeta
* Ogni minuto circa 4.200.000 video vengono visualizzati su You Tube
* Sono previsti 200 miliardi di dispositivi connessi entro il 2020 per le connessioni Internet Of Things (IoT)
* Google arriva a processare 40.000 ricerche al secondo, circa 3.5 miliardi di ricerche al giorno

Il processo di trasformazione di questi dati in informazioni utili e utilizzabili è definito datizzazione e include tutte quelle attività che permettono di collezionare, identificare, analizzare e utilizzare questa mole disordinata e eterogenea di dati. Tutte queste informazioni viaggiano in rete, una rete sempre più complessa e strutturata che deve garantire prestazioni eccezionali, per accontentare un’utenza “viziata” da servizi ad alto carico trasmissivo e con caratteristiche di latenza prioritarie ad esempio app di audio streaming e video streaming quali Spotify, Netflix o Dazn.   
Nella prima parte di questa tesi si ripercorre la storia delle tecnologie di virtualizzazione in particolare quella dei software container, di cui si tratterà anche il funzionamento prendendo in esame nello specifico il sistema Docker. Nella seconda parte dell’elaborato si effettueranno dei test con l’obiettivo di analizzare le prestazioni di un sistema di calcolo che implementa software container per la distribuzione di micro servizi. Una tecnologia dirompente soprattutto per i nuovi processi di cloud computing e di mobile edge computing (MEC), che permettono di spostare il carico computazionale del singolo dispositivo in house a sistemi di calcolo remoti connessi in rete. Servizi come Amazon Web Service, Google Cloud Platform o Microsoft Azure Cloud permettono di focalizzare l’attenzione sull’applicativo in sviluppo, oggetto della produzione per le software house, sgravando le stesse di tutti quei compiti che non sono a valore aggiunto, come la gestione dei server, la gestione della scalabilità, la manutenzione e l’aggiornamento hardware. Questo abbatte le barriere all’entrata per i piccoli attori permettendo a chiunque di imporsi a livello globale senza l’impiego di investimenti iniziali limitanti.

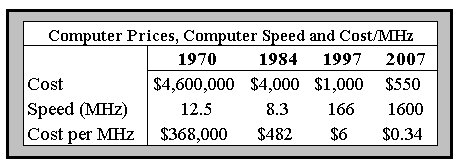
# CAPITOLO 1

# La virtualizzazione software

Questo capitolo presenta un breve excursus sulle tecnologie di virtualizzazione e in particolare sull’utilizzo dei software container, mettendo in luce gli aspetti che contraddistinguono gli stessi dalle macchine virtuali, studiandone le principali applicazioni. Si andrà poi a riassumere in breve la storia di Docker, il sistema open source più diffuso per la creazione e gestione di container, quello utilizzato anche per effettuare i test nel secondo capitolo.

## 1.1 Virtualizzazione e macchine virtuali

La virtualizzazione nasce a cavallo tra gli anni ’60 e ’70 del 900, quando la IBM sviluppa, sotto commissione del MIT, CP-40, una soluzione mai commercializzata sul mercato ed utilizzata solo per le necessità di MIT e Bell lab. Una soluzione di utilizzo multi-batch “time-sharing” dei sistemi di calcolo per le aziende, permetteva di fatti, di utilizzare lo stesso sistema a più utenti contemporaneamente mettendo in atto una politica a divisione in tempo delle risorse, nella storia dell’informatica diede una svolta epocale agli eventi proprio perché permise di sviluppare soluzioni senza però possedere fisicamente un calcolatore che al tempo significava un investimento a cui solo le aziende di grandissime dimensioni o i centri di ricerca finanziati dal governo potevano far fronte. (Conroy, 2018)



*Figura 1: Costi dei computer nel tempo*

Con la moderna definizione di virtualizzazione si indica la capacità di installare diversi sistemi operativi andando a virtualizzarne l’hardware (CPU, RAM, Hard Disk, Scheda di rete); questo set di componenti virtuali viene chiamato macchina virtuale. I vantaggi nell’utilizzo di sistemi virtuali sono molteplici, essi permettono infatti di:

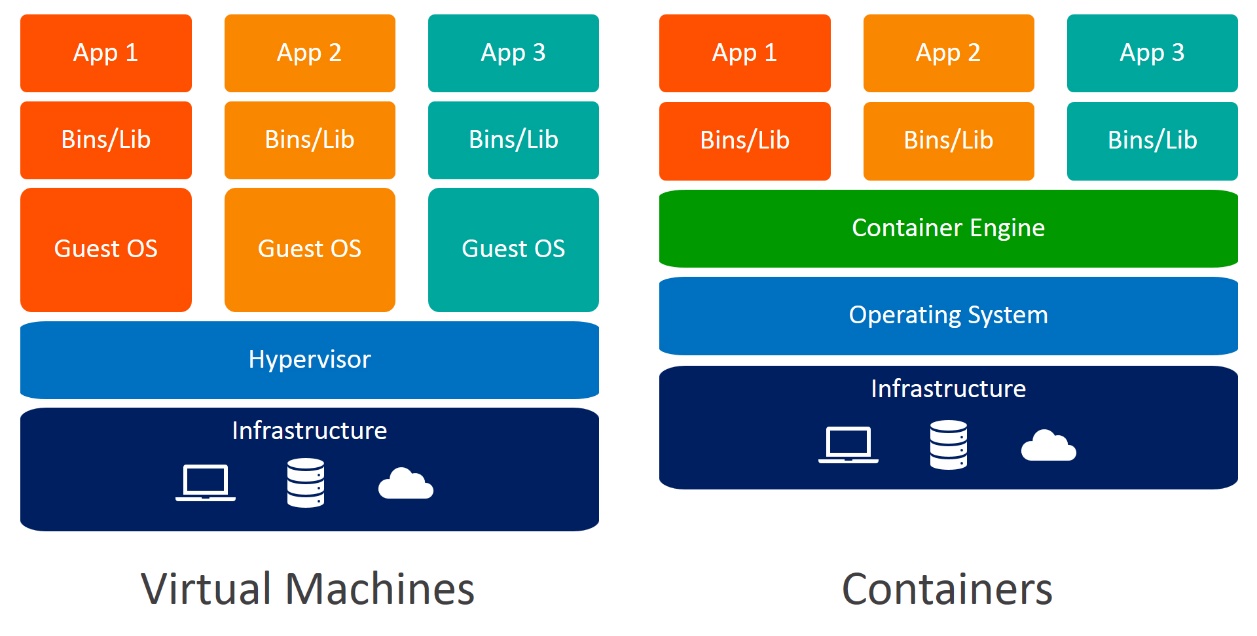
* **Ridurre i costi**. Grazie alla virtualizzazione si può limitare il numero di risorse hardware, riducendo di gran lunga i costi energetici e di manutenzione, andando a razionalizzare, saturando, le risorse attive.
* **Provisioning più veloce.** La virtualizzazione mette a disposizione le risorse e le applicazioni molto più velocemente rispetto ad un’infrastruttura fisica e grazie a questa tecnologia il server è in grado di rispondere a qualsiasi richiesta in modo quasi istantaneo.
* **Procedure di backup migliorate.** La virtualizzazione semplifica le procedure di backup, rendendole estremamente più veloci e permette anche di eseguire backup non solo dei server ma anche delle macchine virtuali.
* **Isolare le risorse.** Tutte le macchine virtuali operano l’isolamento delle risorse per non incorrere in problematiche di compatibilità tra applicativi e per aumentare l’efficienza dei server andando a fornire alle macchine virtuali la giusta quantità di risorse.
* **Sviluppo e testing più semplice.** Con la virtualizzazione e grazie all’isolamento delle risorse, si possono utilizzare le macchine virtuali anche in fase di testing, difatti, non è difficile, in caso di errore, effettuare downgrading.
* **Business continuity e disaster recovery.** Grazie ai backup costanti, è molto facile, in caso di disastro, andare a rimettere in piedi il data center, evitando di compromettere l’operatività aziendale. Grazie a questa possibilità è anche ammissibile effettuare testing del piano di recupero valutandone l’effettivo funzionamento.
* **Gestione semplificata.** Tutte le operazioni di gestione (configurazione di sistema, aggiornamenti, rimozione guasti) possono essere eseguite da un solo centro di controllo e non necessitano di risorse distribuite sull’intero sistema.

Il software virtualizzante simula il funzionamento di un computer fisico, ma il carico di lavoro ricade direttamente sull'hardware reale del computer in uso. La gestione delle risorse e le varie richieste di accesso a queste risorse effettuate da programmi installati all'interno della macchina virtuale vengono prese in carico dalla macchina stessa. La comunicazione tra il sistema operativo ospitante (installato sulla macchina fisica) e quello ospitato (installato sulla macchina virtuale) avviene attraverso un layer di virtualizzazione, che permette di "tradurre" i comandi in arrivo alla, e dalla, macchina virtuale nel "linguaggio" del sistema operativo ospitante. La gestione delle risorse è affidata invece all’hypervisor.

Un hypervisor è un processo che separa il sistema operativo e le applicazioni di un computer dall’hardware fisico sottostante. L’hypervisor guida il concetto di virtualizzazione consentendo alla macchina host fisica di utilizzare più macchine virtuali come ospiti per aiutare a massimizzare l’uso efficace delle risorse di calcolo come la memoria, la larghezza di banda della rete e i cicli della CPU, esempi comuni di hypervisor son VMWare ESX, HyperV monitor di Microsoft.  
L’hypervisor svolge anche un ruolo di difesa, comportandosi da strato di connessione tra le macchine virtuali e le macchine fisiche su cui sono istallate. Questo evita che attacchi inferti alla macchina virtuale ricadano sulla macchina fisica evitando di comprometterne il funzionamento.

## 1.2 Software container definizione e comparazione con VM

Nell’ambito degli [approcci alla virtualizzazione](http://www.cwi.it/labc-della-virtualizzazione_128/), un container è una forma di server virtualizzato a livello del sistema operativo. Invece di creare una istanza virtuale di tutto un server fisico (processore, storage, connessioni di rete, sistema operativo…) come accade per le macchine virtuali, nell’approccio IT a container si attiva una istanza virtuale solo dello spazio utente, quindi essenzialmente dell’ambiente di esecuzione delle applicazioni.

Tutto quello che supporta tale ambiente, quindi dal sistema operativo “in giù” verso l’hardware, non è virtuale ma reale e condiviso fra tutti i container in esecuzione.  
È utile confrontare la struttura di una macchina virtuale con la struttura di un container.

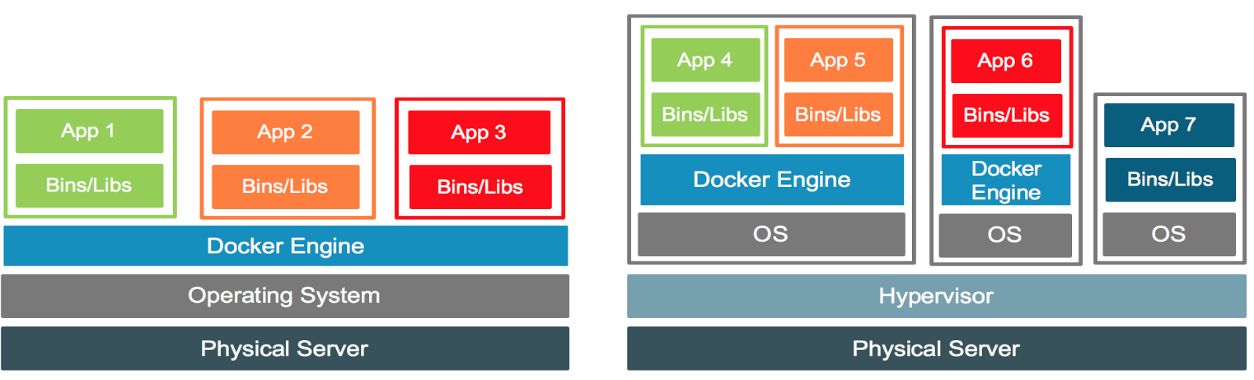
*Figura 2: Confronto VM – Container*

Si può osservare dall’immagine che le macchine virtuali per operare su una infrastruttura fisica necessitano dell’installazione di un sistema operativo per singola macchina virtuale, il tutto coordinato dall’hypervisor che, come detto in precedenza, si occupa di gestire le risorse assegnate alle singole macchine, di risolvere eventuali anomalie e conflitti. Con il sistema di software container invece, sull’infrastruttura fisica si installa un unico sistema operativo, nella gran parte dei casi applicativi kernel Linux, e sullo stesso vengono installati dei container, un’unità standard di software che contiene in sé il codice dell’applicativo e tutti i file dipendenti; gli applicativi in questa maniera sono molto veloci alla partenza e possono essere spostati agilmente da un sistema ad un altro.

La container engine che si prende in considerazione su questa tesi è DOCKER, leader nel mercato della containerizzazione. Nasce nel marzo 2013 quando al PyCon, Solomon Hykes. la introduce al pubblico. Nel 2015 la stessa Docker fonda OCI (open containers initiative) (Torvald, s.d.)per creare degli standard open per la tecnologia dei container. Al momento la OCI contiene due specifiche:

* The Runtime Specification: mira a specificare la configurazione, l'ambiente di esecuzione e il ciclo di vita di un container. La configurazione di un container viene specificata come config.json e delinea i campi che consentono la creazione di un container. L'ambiente di esecuzione viene specificato per garantire che le applicazioni in esecuzione all'interno di un container dispongano di un ambiente coerente tra i runtime e le funzioni definite per il ciclo di vita del container.
* The Image Specification: delinea invece le caratteristiche di un’immagine per container, in maniera tale che possa essere riconosciuta come standard e utilizzata sulle varie piattaforme

In definitiva i container sono un’astrazione dello strato applicativo che incapsula insieme codice e file dipendenti; più container possono girare sulla stessa macchina e condividere lo stesso kernel di sistema operativo, gestendo i processi in un ambiente isolato. Occupano meno spazio delle Virtual Machine (VM) e possono gestire più applicativi. Le virtual Machines, astraggono lo strato fisico, trasformando un’unica macchina reale in più macchine virtuali, l’hypervisor permette di far girare più macchine virtuali sulla stessa macchina fisica. Ogni VM include una copia completa del sistema operativo, gli applicativi, le librerie necessarie e i binari; andando ad occupare vari Gb di memoria, possono essere molto lente all’accensione. Un sistema molto usato prevede un utilizzo ibrido delle due soluzioni, andando così ad ottenere i vantaggi di entrambe le soluzioni, senza contare che ormai le VM sono una tecnologia consolidata, garantiscono quindi una sicurezza intrinseca data dall’esperienza di anni di utilizzo.

*Figura 3: Container e VM insieme*

Per comprendere meglio anche la filosofia con cui sviluppare applicativi su container, bisogna iniziare a non pensarli come delle macchine virtuali, perché queste sono progettate per durare anni, montate su qualche server, e modificate ben poco. I Container invece, possono durare anche pochi minuti per poi venire distrutti, essere attivati per un brevisssimo lasso di tempo e spenti poco dopo.

Di seguito vengono riassunte alcune caratteristiche che permettono di comprendere meglio le metodologie d’uso dei software container:

1. **Isolamento limitato**

I container sono isolati tra di loro ma di default utilizzano le risorse del sistema fisico su cui girano. Se i container non vengono limitati nell’utilizzo delle risorse possono incorrere in attività di conflitto tra loro per contendersi le risorse computazionali. Con le virtual machines la configurazione delle risorse è chiesta al momento dell’installazione così da non concorrere in simili situazioni. I processi containerizzati sono processi che girano nelle stesse istanze del kernel Linux cui gira il sistema operativo il che rende il sistema più che vulnerabile, mentre nelle VM questa separazione è effettuata dalle istanze del sistema operativo per le singole macchine. Per questo un connubio tra le due tecnologie è per il momento la soluzione migliore, che offre si, la velocità, l’elasticità e la semplicità dei container, ma permette anche di difendere il kernel da attacchi o da errori del sistema.

1. **Leggerezza**

Creare un container occupa davvero poco spazio. Il container pesa così poco perché è solo il riferimento ad un’immagine e alcuni metadata sulla configurazione. Non ci sono copie di dati all’interno del container. I container si utilizzano in quelle situazioni in cui eseguire una virtual machine risulterebbe davvero troppo sovradimensionato mentre un semplice container risolverebbe con leggerezza e semplicità il task richiesto.

1. **Microservizi**

I container permettono di gestire microservizi isolati, prendere le grandi applicazioni monolitiche, scomporle in piccole parti e gestirle come un unico sistema composto da tanti piccoli applicativi. (Matthias, 2018)

## 1.3 Funzionamento di Docker

Docker è il sistema leader nel mercato della containerizzazione, fin dal suo lancio nel 2013 docker si è imposto sul mercato dei container distribuendo soluzioni che ora sono diventate dei veri e propri standard per la virtualizzazione al livello applicativo. Docker può essere un ottimo sostituto, ma anche lavorare in sintonia con molti strumenti che sono di largo uso tra le imprese di tutti i tipi tra cui, piattaforme cloud, piattaforma di virtualizzazione per impese, i sistemi di gestione del carico e gli ambienti di sviluppo.

Docker permette di creare e gestire container, per far girare le proprie applicazioni in maniera isolata e sicura su un qualsiasi host. I container sono molto leggeri perché non richiedono l’utilizzo di un hypervisor ma comunicano direttamente con il kernel della macchina su cui girano.

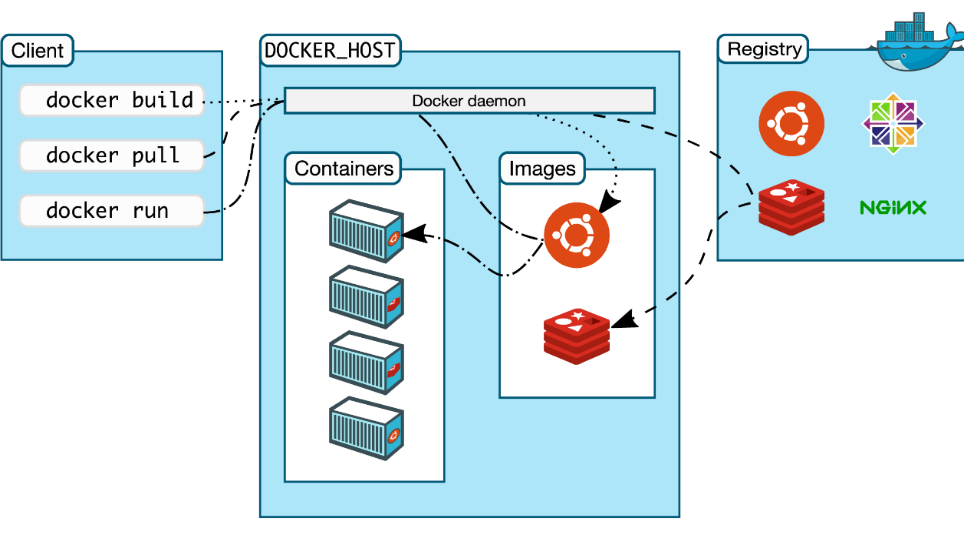
La docker engine è un applicativo client-server con tre componenti principali:

* Un processo demone (Docker + Demon = Dockerd) che funge da  
  server e che svolge la maggior parte del lavoro; creando, gestendo  
  ed eliminando tutti gli oggetti Docker (immagini, container, reti  
  e volumi);
* Una REST API che specifica l’interfaccia attraverso la quale si  
  può comunicare con Dockerd;
* Un client con interfaccia a linea di comando (CLI) che utilizza  
  Docker REST API per controllare e interagire con Dockerd.



*Figura 4: Architettura Docker*

Quindi, il client docker parla con il DockerD, che svolge tutto il lavoro di costruire, attivare, e distribuire i container. Il client docker e il Daemon possono girare sullo stesso sistema, oppure si può connettere un client ad un daemon remoto. Il client e il daemon comunicano utilizzando una REST API su un socket UNIX o su una interfaccia di rete.

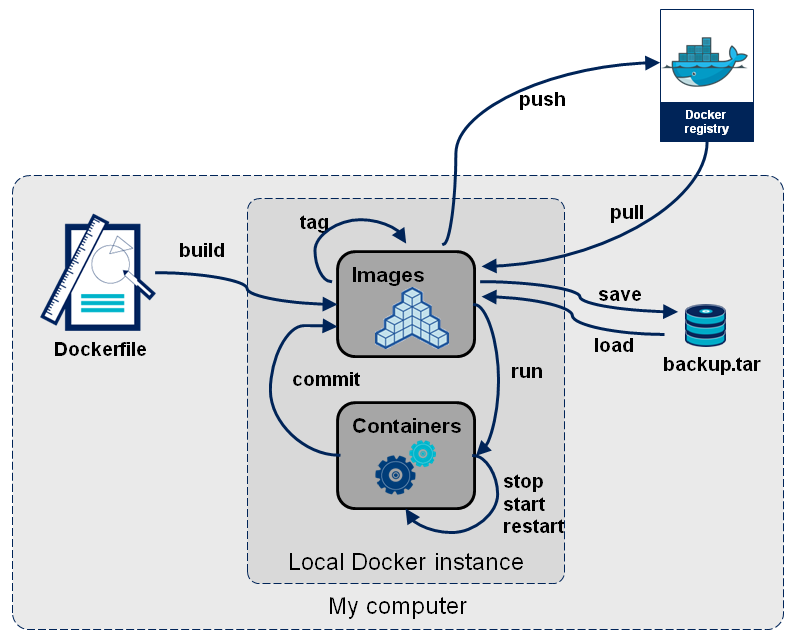


*Figura 5: Stream del processo di comunicazione in Docker*

Nell’immagine si possono notare anche i docker registry, sono database di docker image che vengono utilizzati per la creazione dei container. Docker hub, per esempio, è una registry pubblica che ognuno può utilizzare per condividere le proprie immagini. I comandi docker pull e docker run necessitano che le immagini siano memorizzate nei propri registri mentre con il comando docker push l’immagine viene inserita all’interno della propria registry.

Ogni volta che si utilizza docker in realtà si sta lavorando con una serie di oggetti detti appunto docker Objects, questi includono immagini, container, reti, volumi e plugin. Di seguito una piccola descrizione dei due principali, le immagini e i container.

* **Images**: L’immagine è un template di sola lettura con le istruzioni per creare un container. Molto spesso le immagini sono basate su altre immagini con delle aggiunte che le personalizzano in base alle proprie necessità. Si possono creare le proprie immagini o utilizzare quelle create da altri e condivise pubblicamente, su docker hub o su piattaforma di code sharing come Git Hub. Per creare la propria immagine si deve scrivere un docker file, un file di testo con tutte le informazioni per far si che l’immagine giri senza problemi. Ogni istruzione nel docker file crea uno strato nell’immagine, e se si modifica il dockerfile e si ricostruisce l’immagine solo gli strati che sono stati cambiati verranno ricostruiti.



*Figura 6: Creazione di una docker image*

* **Container**: Il container è l’istanza eseguibile di un’immagine. Si può creare, attivare, fermare, spostare o cancellare un container tramite la command line interface (CLI). Di default un container è ben isolato ma docker permette di controllare e scegliere quanto i sottosistemi del container come memoria e rete sono isolati da altri container o dalla macchina Host.

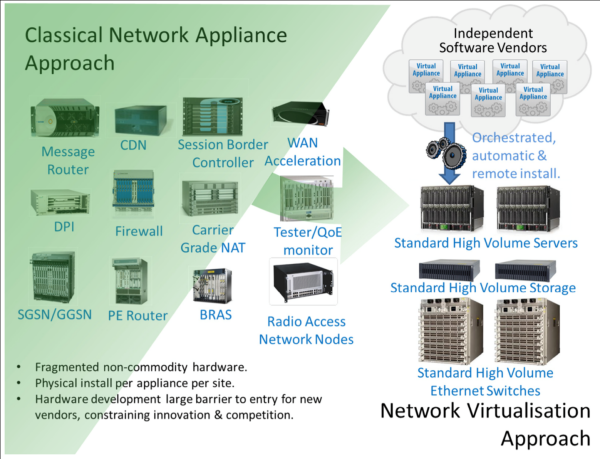
Da quanto scritto in precedenza si comprende che i Container sono la risposta migliore alle esigenze delle aziende che sviluppano servizi software e alle aziende che operano del campo della distribuzione in data center, permettono infatti di: migliorare il time to market eliminando i possibili conflitti tra l’applicazione e l’ambiente in cui andrà ad operare, ridurre i costi di infrastruttura aumentando la densità di calcolo all’interno dei propri server e i costi di licenza dei software, facilitare la gestione delle singole applicazioni andando a gestire il singolo container, di gestire, infine, la scalabilità in maniera facile e sicura.

## 1.3 Virtualizzazione nelle reti di telecomunicazione

### 1.2.1 Network Function Virtualization

Si inizia a parlare di NFV al *“SDN and OpenFlow congress”* del 22 ottobre 2012 a Darmstadt in Germania. Da quel congresso viene pubblicato un white paper (Network Function Virtualization, an introduction, benefits, enablers, challenges and call for action, 2012) con i primi obiettivi che ci si ponevano andando a sviluppare questa tecnologia, la pubblicazione si chiude infatti con una Call To Action alle Telco e alle aziende IT ai fini di collaborare per raggiungere rapidamente gli obiettivi sperati, e con la formazione, incoraggiata dall’ european telecommunication standard institute (ETSI), di un Industry Specification Group (ISG) che gestisse lo sviluppo delle tecnologie atte all’implementazione a casi reali della NFV.

L’ISG NFV conta ora 71 membri e 32 partecipanti tra le più importanti aziende ICT e IT del mondo. La definizione che il gruppo dà della NFV parte da un’analisi critica delle presenti reti di telecomunicazioni, si segnala che è sempre più difficile per gli operatori lanciare nuovi servizi di rete poiché questi richiedono nuovi spazi fisici ed energia, sostanziosi investimenti e figure molto rare per la configurazione di nuovi apparati di rete. In più le “hardware based application” raggiungono in fretta l’obsolescenza e richiedono quindi anche grandi spese di manutenzione andando a diminuire sempre di più le entrate. La NFV si prefigge l’obiettivo di trasformare il modo in cui le architetture di rete vengono progettate, fondendo molti elementi di rete in grandi server industriali standardizzati, switch e storages potranno essere localizzati in Data center, nei nodi della rete e nei dispositivi d’utente. Questo permetterebbe di installare nuova componentistica software senza l’utilizzo di altri componenti hardware andando a tagliare costi e rigidità alle vecchie reti di telecomunicazione. Di seguito un’utile figura per comprendere meglio il concetto di rete virtualizzata.



*Figura 7: Vision per la Network Function Virtualization*

Nel primo white paper le sfide che si ponevano erano relative a:   
-**Portabilità e interoperabilità.** La capacità cioè di eseguire applicazioni virtuali in differenti ma standardizzati data center, forniti da differenti vendor per differenti operatori. Definire quindi un’unica interfaccia che disaccoppi chiaramente le istanze software dall’hardware sottostante. La portabilità permette anche agli operatori la libertà di ottimizzare l’allocazione delle risorse delle applicazioni virtuali senza vincoli.  
-**Migrazione, coesistenza e compatibilità con piattaforme esistenti.** L’NFV deve coesistere con gli apparati di rete gia esistenti, deve essere compatibile con i sistemi di management della rete, con gli operation support system, con i sistemi di stazioni base e con i sistemi informatici di orchestrazione gia esistenti. Deve riuscire a lavorare in un ambiente ibrido che contempla sia applicativi basati su rete fisica sia applicativi su rete virtuale.

-**Gestione e orchestrazione**. Deve essere garantita un’architettura per la gestione e l’orchestrazione che permetta di sfruttare l’opportunità data dalla virtualizzazione in termini di flessibilità assicurata dall’ambiente virtuale standardizzato. Questo permette di ridurre i costi per implementare le tecnologie all’interno di una rete operativa gia sviluppata.  
-**Automation.** I processi virtuali devono essere automatizzati per garantire il successo dell’implementazione.  
-**Sicurezza e resilienza.** La sicurezza di una rete virtualizzata deve garantire almeno la sicurezza di una rete fisica agli operatori di rete, dando agli stessi gli strumenti per gestire il proprio Hypervisor. La resilienza invece deve essere garantita da un processo automatico, che permetta di re-instaurare la comunicazione dopo un guasto.  
-**Stabilità.   
-Semplicità.** Deve essere garantita semplicità di gestione della rete, la possibilità di semplificare le piattaforme di rete troppo complesse e i sistemi di supporto.

La struttura di NFV si compone di tre componenti principali.

* Virtualized network functions (VNFs): sono le implementazioni software di funzioni di rete che possono essere rilasciate su un'infrastruttura NFV (NFVI)
* Network functions virtualization infrastructure (NFVI): è il complesso di tutte le componenti hardware e software che costituiscono l'ambiente in cui le VNFs sono rilasciate. L'infrastruttura NFV può essere implementata su postazioni fisicamente disgiunte; la rete che fornisce la connettività tra tali postazioni è essa stessa parte dell'Infrastruttura NFV.
* Network functions virtualization management and orchestration architectural framework (NFV-MANO Architectural Framework): è l'insieme di tutti i blocchi funzionali, dei repository usati da tali blocchi, e dei riferimenti e delle interfacce attraverso cui i blocchi funzionali scambiano informazioni al fine di gestire ed orchestrare NFVI e VNFs.

I blocchi costitutivi sia di NFVI che di NFV-MANO rappresentano la piattaforma NFV. Nel ruolo di NFVI, essa consiste nelle risorse fisiche e virtuali di elaborazione e memorizzazione e nel software di virtualizzazione che opera su un controller hardware. La piattaforma NFV implementa funzionalità carrier grade per rispondere alle necessità delle reti pubbliche: gestire e monitorare i servizi della piattaforma, porre rimedio ai guasti e garantire requisiti di sicurezza.

### 1.2.2 Software Defined Networking

La tecnologia Software-defined networking (SDN) costituisce un nuovo approccio alle architetture di rete in ottica cloud computing. I suoi punti di forza principali sono la facilità nell'amministrazione e configurazione al fine di migliorarne performance e facilitarne il monitoring. La sua evoluzione può essere contestualizzata nel periodo in cui avviene la separazione dei layer di controllo e dati utilizzati in precedenza nella rete telefonica pubblica per semplificare il provisioning e la gestione prima che questa architettura iniziasse ad essere utilizzata nelle reti dati.

L'utilizzo di software [open source](https://it.wikipedia.org/wiki/Open_source) in architetture che prevedono la suddivisione dei layer sopracitati, risale al Progetto Ethane del dipartimento di informatica di Stanford. La semplicità delle funzioni di switch in Ethane ha portato alla creazione del protocollo di comunicazione OpenFlow che permette di definire il percorso dei pacchetti attraverso la rete. Al 2008 risalgono la realizzazione della prima API e la creazione di NOX, un sistema operativo per le reti. Il lavoro sul protocollo OpenFlow ha avuto poi seguito portando alla creazione di banchi di prova per testarne il suo utilizzo all'interno di un campus, e attraverso la WAN come backbone per connettere più campus.

L'architettura SDN nasce con lo scopo di essere gestibile, economicamente efficiente e adattabile; inoltre deve essere dinamica per la natura mutevole e ad alto consumo di banda delle applicazioni odierne. Le architetture SDN disaccoppiano il controllo di rete e le funzioni di forwarding, dando la possibilità al controllo di rete di divenire direttamente programmabile e alla sottostante infrastruttura di essere astratta dalle applicazioni e dai servizi di rete.

L'architettura SDN è:

* **Direttamente programmabile**: perché il controllo della rete è disaccoppiato dalle funzioni di forwarding.
* **Agile**: separando il controllo dal forwarding gli amministratori possono modificare dinamicamente il flusso di traffico nell'intera rete per adattarsi alle necessità di cambiamento.
* **Gestita centralmente**: l'intelligenza della rete è (logicamente) centralizzata all'interno dei controller SDN che mantengono una vista globale della rete, la quale appare alle applicazioni ed ai motori di policy come un unico switch logico.
* **Programmaticamente configurata***:* SDN permette ai gestori della rete di configurare, gestire, rendere sicure e ottimizzare le risorse di rete in maniera molto veloce attraverso script automatizzati che essi stessi possono scrivere, in quanto non devono basarsi su software proprietario.
* **Basata su standard aperti e indipendente dai vendor**: Se implementata attraverso standard aperti, SDN semplifica il disegno e manutenzione della rete poiché le istruzioni sono fornite dai controller SDN invece che da molteplici dispositivi e protocolli proprietari.

Le applicazioni SDN sono programmi che in maniera esplicita comunicano i propri requisiti di rete e il comportamento desiderato della rete ai controller SDN attraverso la Northbound Interface (NBI). Inoltre, essi possono mantenere una vista astratta della rete per i propri scopi. Una applicazione SDN è formata da una SDN Application Logic e da uno o più driver NBI. Le applicazioni SDN possono esse stesse esporre un altro layer di controllo astratto di rete, mostrando una o più interfacce NBI di alto livello attraverso i rispettivi agenti NBI.

Il controller SDN è un'entità logicamente centralizzata con il compito di

1. Tradurre i requisiti provenienti dall'Application Layer SDN verso il Datapath SDN.
2. Fornire alle Applicazioni SDN una vista astratta della rete (che può includere statistiche ed eventi).

Un controller SDN è formato da uno o più agenti NBI, la SDN Control Logic, ed i driver di Interfaccia Control to Data-Plan (CDPI). La definizione di Controller SDN come un'entità logicamente centralizzata non prescrive né preclude dettagli di implementazione come la connessione gerarchica tra controller, interfacce di comunicazione tra controller, né la virtualizzazione di risorse di rete.

Il Datapath SDN è un dispositivo logico di rete che espone visibilità e controllo sulle sue capacità offerte di forwarding ed elaborazione dati. Un Datapath SDN comprende un agent CDPI ed un insieme di uno o più dispositivi di forwarding del traffico e nessuna o più funzioni di elaborazione del traffico. Questi possono includere il semplice forwarding verso le interfacce esterne, l'elaborazione interna del traffico o funzioni di terminazione. Uno o più SDN Datapath possono essere contenuti, fisicamente, all'interno di un unico elemento di rete, una combinazione fisica integrata di risorse di comunicazione.

La SDN CDPI è l'interfaccia definita tra un controller SDN ed un SDN Datapath; questa interfaccia offre:

1. Controllo programmatico di tutte le operazioni di forwarding
2. Esposizione delle capacità sottostanti
3. Reporting statistico
4. Notifica di eventi

Uno dei principi di SDN è l'aspettativa che questa interfaccia sia implementata in maniera aperta, neutrale rispetto ai vendor ed inter-operabile.

# CAPITOLO 2

# Sperimentazioni sull’uso di container

In questo capitolo sono descritte le attività di sperimentazione eseguite al fine di analizzare l’effettivo miglioramento apportato dall’uso dei container rispetto alle macchine virtuali. Nella prima parte sarà descritto il set up utilizzato ai fini del test, che purtroppo è diverso da quello previsto nei requisiti consigliati da Open Cord per l’implementazione del sistema. In un primo esperimento di benchmark saranno analizzati il tempo di overhead, di cui Docker ha bisogno per gestire i container e quindi gli applicativi, andando ad implementare un container in cui gira un servizio A, un secondo container in cui gira un servizio B e un terzo container C in cui si implementa il servizio A+B. Si determinerà poi la differenza tra i tempi di esecuzione di C nel suo complesso, con i tempi di esecuzione dei singoli processi A e B.

In seconda fase si effettuerà invece un esperimento legato alle latenze di scala, ossia, si aumenterà il numero dei container contenenti il processo C e si analizzeranno le latenze che si osservano all’aumentare di questo numero. I container girano in contemporanea sulla nostra macchina. È interessante osservare l’impatto della funzione docker-compose per far partire più container diversi nello stesso istante e farli interagire tra di loro, sia sui singoli processi A e B che sul processo completo C.   
Questi risultati sono utili ai fini di un’analisi quantitativa dell’utilizzo di container per la gestione ottimale di applicativi ospitati su server.

Tutti i codici utilizzati, i test fatti e i risultati di questi test, sono liberamente consultabili, migliorabili e criticabili, sulla piattaforma GitHub all’indirizzo

(INSERIRE INDIRIZZO GIT HUB)

## 2.1 Set-Up Hardware e Software

La macchina utilizzata per effettuare i test è una Work Station Fujitsu, nello specifico una Celsius R940n, che ha caratteristiche differenti da quelle consigliate da Open Cord ma che comunque ha permesso di ottenere utili risultati per trarre delle conclusioni di interesse. Nella tabella sottostante sono confrontate le caratteristiche tecniche della macchina utilizzata nei test con quella dei server consigliati da ONF (Open Cord, s.d.) per la configurazione di un data center modello, in maniera da contestualizzare i test eseguiti.

|  |  |  |
| --- | --- | --- |
| Macchina | Work Station Fujitsu  Celsius R940n | Server OCP Inspired™  QuantaGrid D51B-1U |
| CPU | Intel® Xeon® CPU E5-2637  v3 3.5 GHz | 2x Intel E5-2630 v4  10C 2.2GHz 85W |
| RAM | 16GB 2133Mbps DDR4 | 64GB 2133Mbps DDR4 |
| HD | 1X 240 GB Kingston v300 SSD  1X 2TB Seagate Desktop  HDD ST2000DM001 | 2x 500GB HDD |

*Tabella 1: Paragone macchina utilizzata VS macchina consigliata da ONF*

Il sistema operativo montato sulla macchina è Windows 10 Pro. Sarebbe stato meglio sicuramente sviluppare i test su sistema Linux, proprio perché nelle applicazioni reali è il più diffuso, in quanto permette di personalizzare le funzioni del sistema al fine di gestire le operazioni necessarie al lavoro da svolgere, eliminando quelle funzioni che, ai fini dell’utilizzo specifico che se ne deve fare, non restituiscono nessun valore aggiunto ma anzi appesantiscono la macchina e rallentano i sistemi di calcolo. Per la creazione, configurazione e gestione dei container si è utilizzato il software Docker, capofila tra i sistemi per lo sviluppo su container. La specifica e il codice di runtime di Docker sono ora proprietà della Open Container Initiative (OCI), una comunità open source con l’esplicito obiettivo di creare uno standard aperto per i container in ambiente Unix. Dal 2016 è possibile anche far girare Docker in ambiente Windows in maniera nativa senza dover ricorrere all’uso di macchine virtuali.

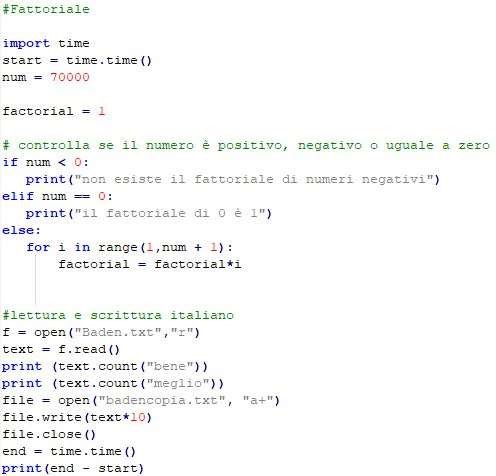
## 2.2 Testing e analisi dei dati

Le sperimentazioni eseguite si possono dividere in due parti. Nella prima sperimentazione si prende in considerazione solamente l’utilizzo del container per far girare gli script, senza chiamare in causa docker-compose. In tale fase non si necessita di far girare più container contemporaneamente. Mentre nella seconda esperienza ci si concentra completamente sull’utilizzo di docker-compose per far girare il singolo container e scalarlo fino a 100 volte.

### 2.2.1 Primo Test

Il primo test, è molto semplice, consta nel creare l’immagine per uno script, farla girare in un container e valutare il tempo di esecuzione. Si opererà nello stesso modo per un secondo script simile e poi per un terzo script che svolge entrambi i task dei due script precedenti. Gli script sono stati scritti in Python per sfruttare la capacità di calcolo della CPU ed effettuare anche funzioni di lettura e scrittura su disco.

Si analizzano in dettaglio i due script



*Figura 8: Script 1*

Per caricare la CPU il primo script (ferretti-script.py) calcola il fattoriale di un valore assegnato, nel caso specifico 70000, per far sì che i tempi di calcolo siano abbastanza lunghi da poter essere osservati; come secondo task prende un file di testo (Baden.txt) lo legge, conta quante volte si ripete la parola “bene” e la parola “meglio”, lo stampa a schermo, poi copia l’intero testo e lo incolla in un altro file chiamato badencopia.txt per 10 volte. I task selezionati sono inutili da un punto di vista pratico ma sono utili ai nostri fini, ossia per valutare il tempo di esecuzione dei container stessi.   
Questo tempo è infatti calcolato dallo script stesso con le funzioni di

**import time**

**start = time.time()**

**[code]**

**end = time.time()**

**print( end – start)**

Queste funzioni segnano il tempo di inizio e fine del processo e restituiscono il dato “tempo di calcolo”, come differenza dei due tempi, fondamentale per il nostro obiettivo.  
Nello specifico il file Baden è composto da una frase di Baden Powell, fondatore del movimento scoutistico mondiale, “essere buoni è qualcosa, fare del bene è molto meglio” ripetuta varie volte. Con il codice si calcolano infatti il numero di volte in cui la parola bene e la parola meglio vengono ripetute.

Si analizzano ora i comandi utilizzati su docker da PowerShell di Windows per creare e far partire l’immagine del primo script.

*Figura 9: Creazione immagine per lo script 1*

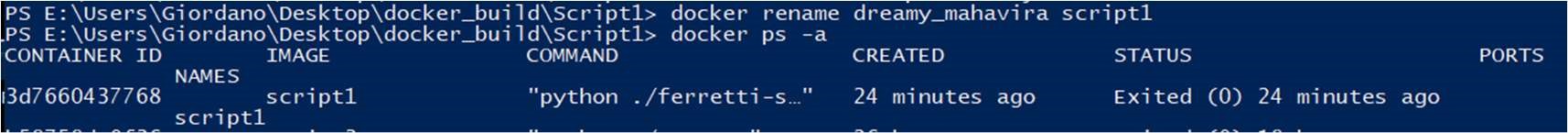


Per prima cosa si considera la cartella contenente il dockerfile, lo script e i file utilizzati nello script, in questo caso Baden.txt.

Il comando docker build -t script1 . crea l’immagine dal dockerfile, il -t consente di dare un nome alla immagine così da poterla identificare e il “.” indica che il dockerfile si trova proprio nella cartella dove si è aperta la shell, questo luogo è chiamato “Context” ed è il luogo in cui si trova la serie di file utili alla configurazione della docker image.

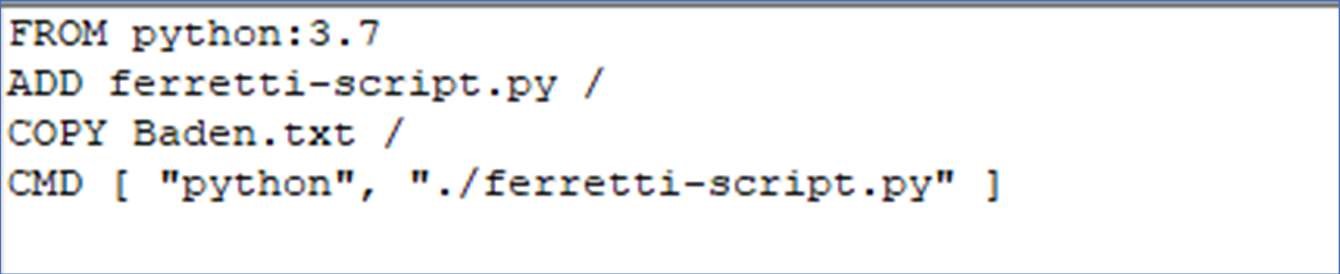
Una volta creata si utilizza il comando docker run script1 che fa partire l’immagine all’interno di un container che andrà a creare autonomamente e a cui darà un nome di fantasia, questo comando restituisce tre valori che sono appunto: il numero di volte che si ripete la parola “bene”, il numero di volte che si ripete la parola “meglio” e il tempo di esecuzione dello script. Nel caso specifico le due parole si ripetono in quantità identiche perché la frase copiata le contiene entrambe una sola volta. Tutti i container creati sono identificati da un ID e da un nome leggibile, ogni qualvolta si crea un container e non gli viene dato un nome specifico, docker in automatico gli affida un nome di fantasia.

Con il comando docker ps -a si visualizza la lista dei container, con il tag -a si visualizzano sia gli attivi che i non attivi, e con il comando docker rename [VN] [NN], dove VN sta per vecchio nome e NN per nuovo nome, si modifica il nome del container.

*Figura 10: Rinominare la docker image*

Si analizzerà ora il file più importante, quello che permette di creare la docker image dello script per farla girare poi all’interno di un container, il dockerfile. Il dockerfile è un file di testo senza estensione in cui sono contenute tutte le istruzioni per creare la docker image, contenente tutti i comandi che un utente può richiamare sulla command line per assemblare un’immagine.

*Figura 11: Docker File script 1*

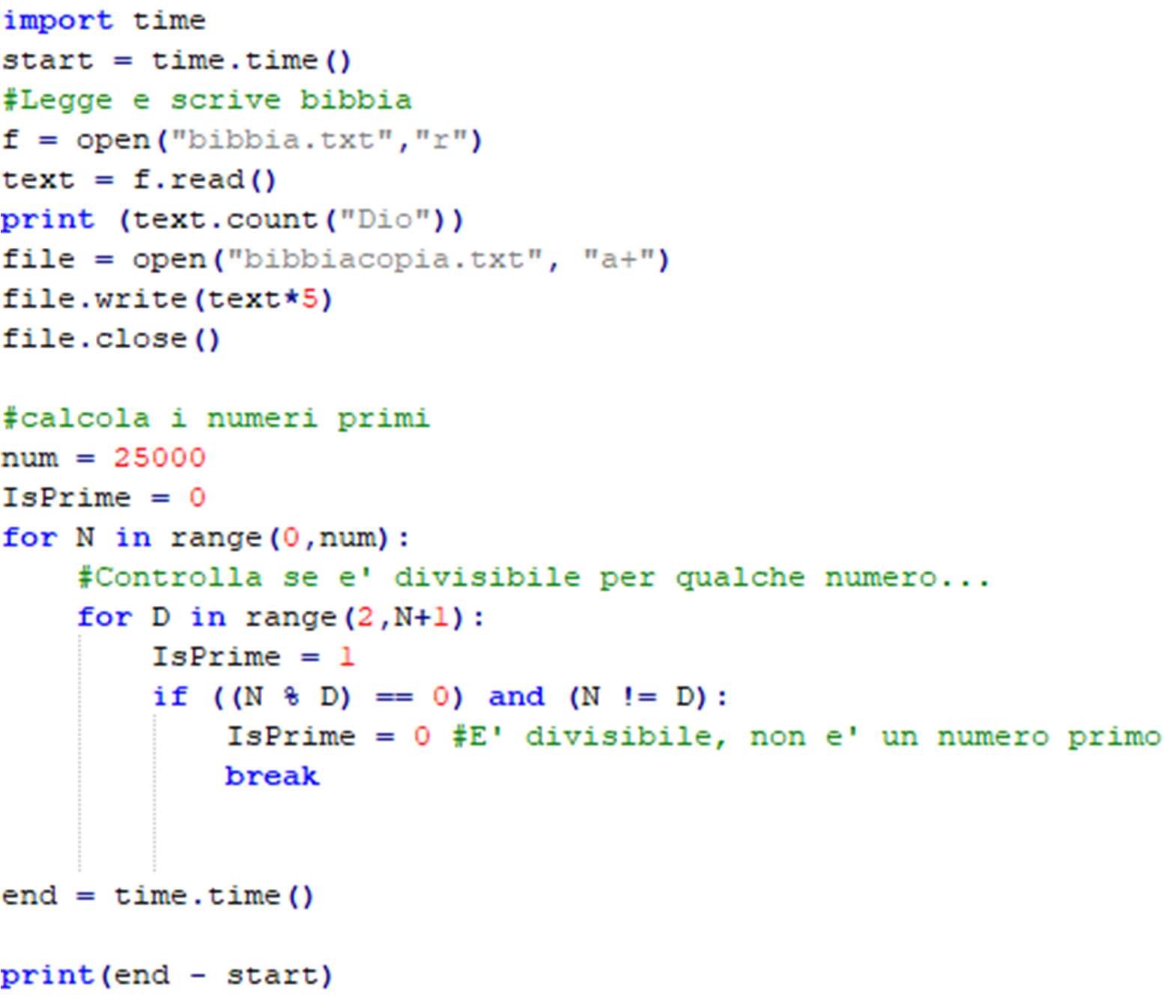


Il dockerfile per lo script1 è molto semplice, con l’istruzione FROM si inizializza un nuovo build e si crea un’immagine di base per istruzioni seguenti, nel caso in esame l’immagine di python nella versione 3.7 poiché gli script sono in python.

Con l’istruzione ADD si copiano nuovi file, cartelle o file remoti da un certo indirizzo <src> e si aggiungono al filesystem della nostra immagine, in questo caso il file .py del nostro script.

L’istruzione COPY effettua le stesse azioni di ADD ma solo con file in locale, non permette cioè di accedere a file in remoto tramite URL.   
Per ultima si osservi l’istruzione CMD (command) che permette di utilizzare la funzione ferretti-scipt.py come funzione di Entrypoint, cioè funzione di default al lancio della nostra immagine. In questo caso si lancia python e ferretti-script.py perché sono le uniche funzioni che occorre far girare.

Si osservi ora lo script2 e si proceda nello stesso modo.

*Figura 12: Script 2*

Lo script2 esegue sempre un calcolo di lettura/scrittura e un calcolo numerico, in questo caso si considera una frase della bibbia “in principio era il Verbo, e il Verbo era presso Dio e il Verbo era Dio” copiata e incollata svariate volte come nello script precedente. Per il calcolo numerico si occupa di trovare i primi 25000 numeri primi. Anche qui con i comandi start e end viene restituito il tempo di esecuzione dello script. Ci si posiziona a questo punto nella cartella in cui sono stati salvati gli elementi utili per creare l’immagine dello script2 (iacino.py) quindi dockerfile, script2, bibbia.txt e si utilizzano gli stessi comandi utilizzati in precedenza

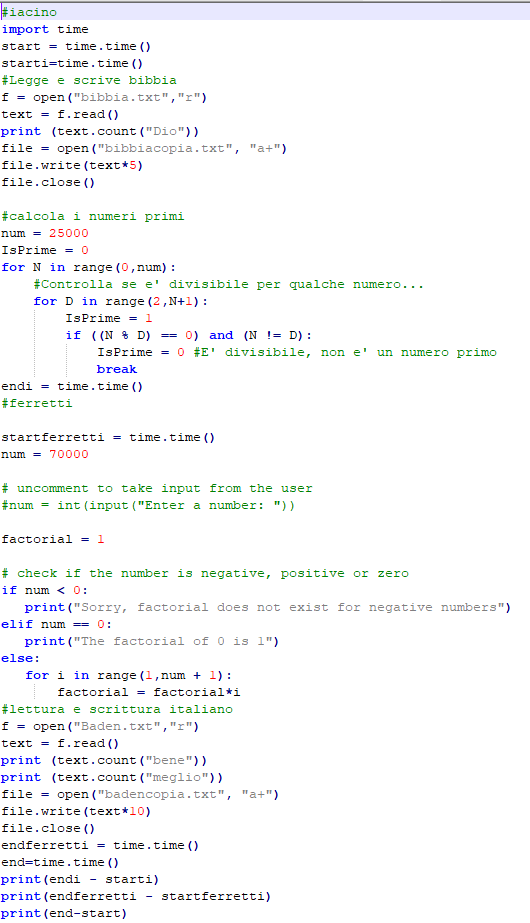
docker build -t script2 . 🡪 docker run script2.

*Figura 13: Creazione immagine per lo script 2*

Ovviamente i dati restituiti saranno differenti, e si otterrà infatti il numero delle volte in cui si ripete la parola “Dio” e il tempo di esecuzione dello script.

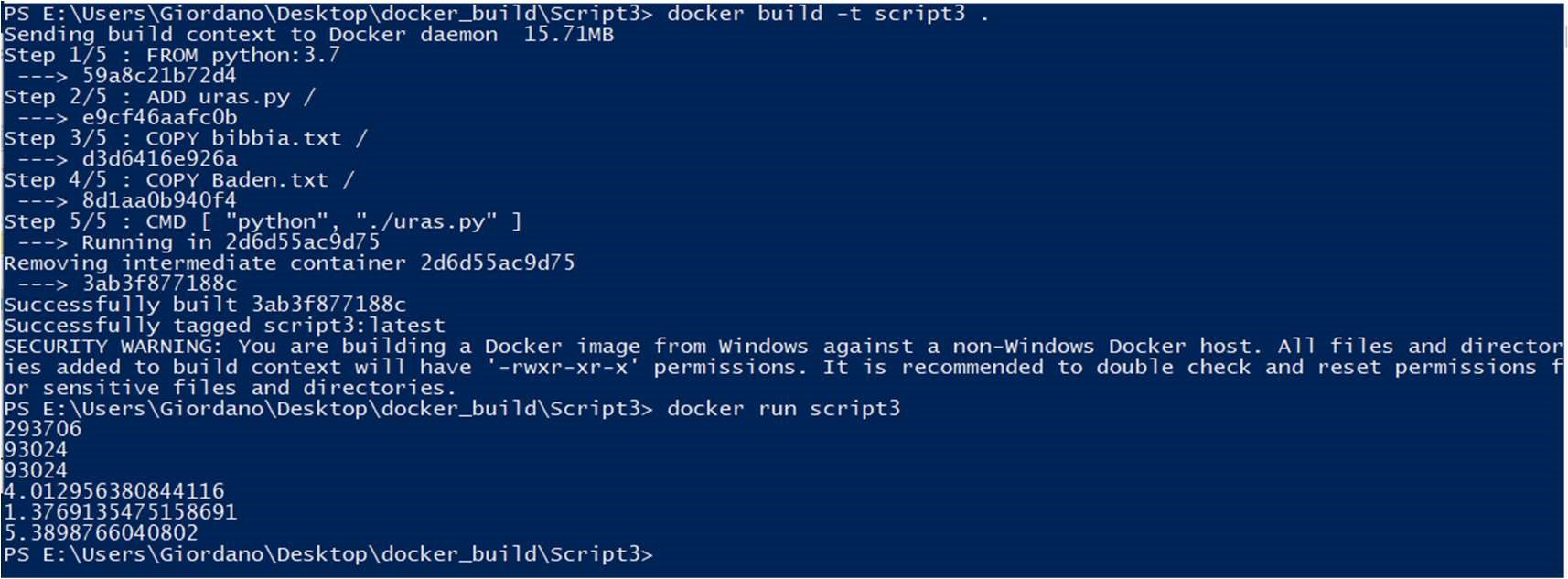
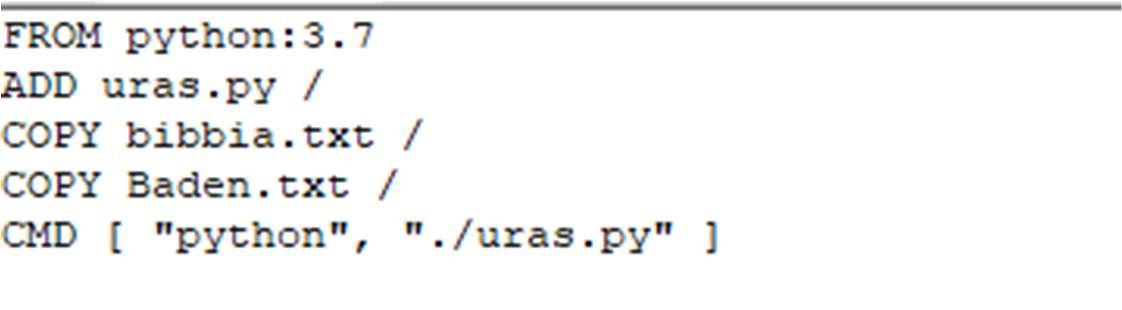
Per concludere si osservi ora lo script3 (uras.py) che è semplicemente l’unione dei 2 script precedenti.

*Figura 14: Script 3*



*Figura 13: Script*

L’unica aggiunta consiste nella possibilità di ottenere i tempi dei singoli script e dello script totale utilizzando sempre lo start e l’end nella maniera coerente. Anche per questo script è stata creata e lanciata l’immagine dal suo dockerfile che in questo caso include i due file bibbia.txt e Baden.txt.



*Figura 15: Docker file, build e run script 3*

Questa volta si osserva che i 6 valori restituiti sono rispettivamente, il numero delle volte in cui si ripete la parola “Dio” nel testo bibbia.txt, il numero delle volte in cui si ripete la parola “bene” e il numero delle volte in cui si ripete la parola “meglio” nel testo Baden.txt, il tempo di esecuzione dello script2, il tempo di esecuzione dello script1 e il tempo di esecuzione totale dei due script.

Una volta impostato l’esperimento questo è stato ripetuto più volte per valutare la media così’ da migliorare la qualità dei risultati; riportati nella tabella seguente, in più, non sono state prese in considerazione le prime risposte del test, perché queste includono tempi di inizializzazione dei Docker, quindi si sono fatti 3 run a “vuoto” per poi iniziare a collezionare risultati.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Script1 |  | Script2 |  |  | Script3 |  |
|  | 1,6281 |  | 4,2646 |  |  | 5,94 |
|  | 1,6942 |  | 4,0355 |  |  | 6,2193 |
|  | 1,6609 |  | 4,1424 |  |  | 6,1177 |
|  | 1,7467 |  | 4,0483 |  |  | 5,7803 |
|  | 1,6781 |  | 4,0527 |  |  | 6,1721 |
|  | 1,6922 |  | 4,0811 |  |  | 6,2048 |
|  | 1,728 |  | 4,2711 |  |  | 6,3185 |
|  | 1.681 |  | 4,7908 |  |  | 5,8977 |
|  | 1,6957 |  | 4,1234 |  |  | 6,0146 |
|  | 1,6627 |  | 4,2917 |  |  | 5,8514 |
| Media |  |  |  |  |  |  |
|  | 1,68676 |  | 4,21016 |  |  | 6,05164 |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| Teoria |  |  | 5,89692 |  |  |  |
| Pratica |  |  | 6,05164 |  |  |  |
| Overhead |  |  | 0,15472 |  |  |  |

*Tabella 2: Tempi di esecuzione degli script e relativo tempo di Overhead (in secondi)*

I risultati riportati in tabella sono espressi in secondi e rappresentano i tempi di esecuzione dello script1, dello script2 e dello script3. Nella cella sottostante i 10 valori in colonna è riportato il valore medio, si può notare con questo esperimento che il valore teorico (,risulta inferiore al tempo di esecuzione dello script 3 pratico. Questo fenomeno è dovuto probabilmente in parte all’aumento del carico computazionale che richiede a Docker di organizzare il lavoro in maniera differente, in parte all’aggiunta dei comandi di lettura del tempo addizionali a quelli iniziali.

Il tempo così definito di overhead risulta del 0,974 % maggiore rispetto al tempo teorico, di poco conto per un calcolo di questo tipo ma che può diventare critico per applicazioni di diversa natura in cui le latenze sono un fattore da tener sotto rigido controllo. Una soluzione a questo problema potrebbe essere l’ottimizzazione del codice e del container stesso.

### 2.2.2 Secondo Test

Il secondo test prende in considerazione invece una delle caratteristiche che hanno permesso ai software container di espandersi così rapidamente, la scalabilità. Per scalabilità si intende la capacità di un sistema di aumentare o diminuire in scala a seconda delle necessità, con i container e in docker questo processo è reso davvero semplice grazie allo strumento docker-compose.

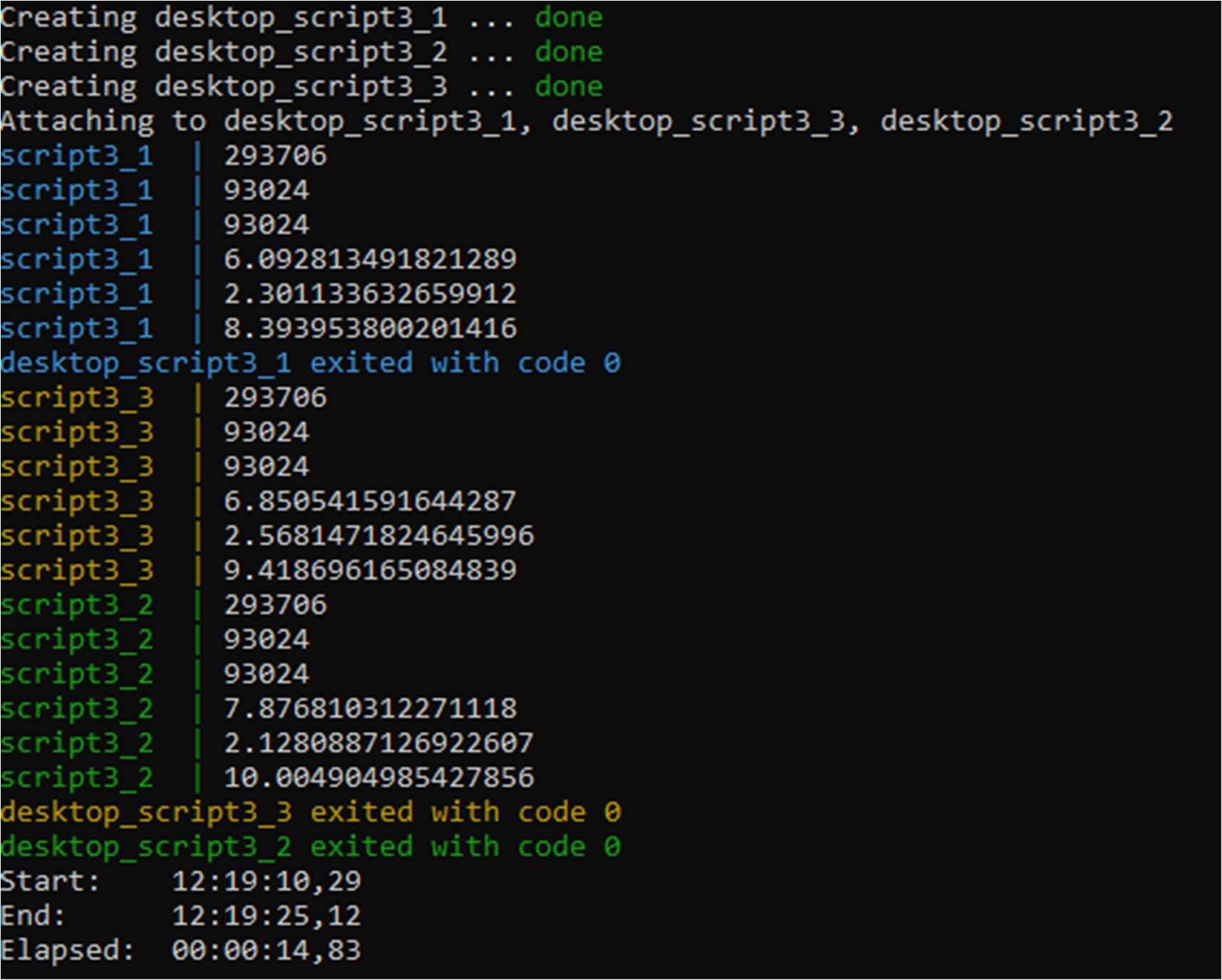
Docker compose è uno strumento utile ogni qual volta si debba far girare un applicativo che richiede l’utilizzo di più container, nel nostro caso si andrà soltanto a riprodurre lo stesso container più volte per osservare la variazione prestazionale della nostra macchina. Per il test si utilizzerà lo script3 ma, per semplificare l’analisi, è stato approntato un altro script, in questo caso un file di tipo batch (script.bat), che permette di impostare il numero di scale, cioè il numero di container che si desidera far girare contemporaneamente per ogni prova.



*Figura 16: script.bat*

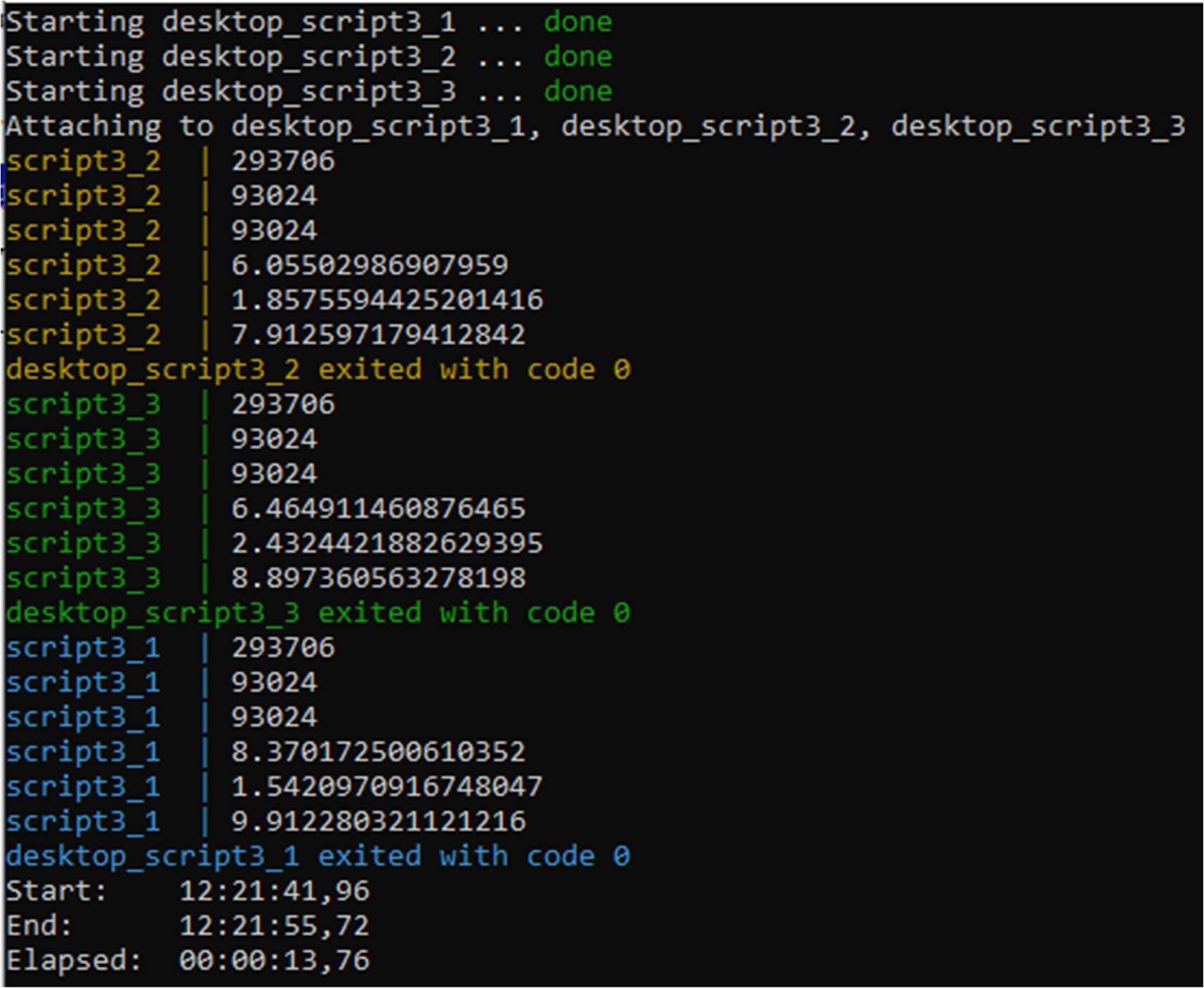
Il comando docker-compose up –scale script3 = N con N numero di container, andrebbe lanciato ogni volta da riga di comando, mentre con questo batch file è più semplice ripetere la prova ed è più semplice tener traccia di tempi d’esecuzione.

Per andare avanti nella trattazione del test occorre però considerare come lavora docker-compose.

*Figura 17: Docker Compose con scale=3, si osserva la fase di creating dei container.*

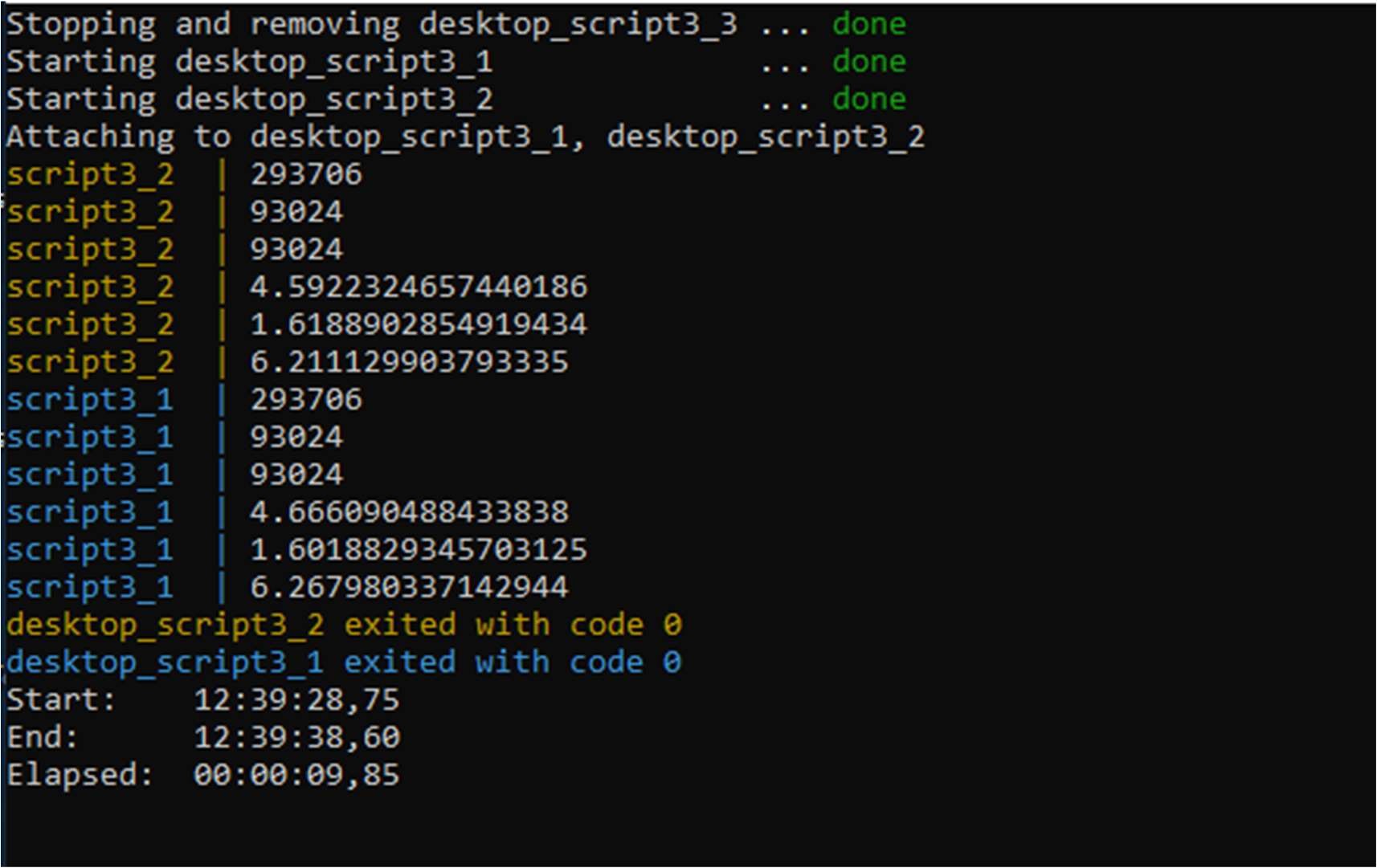
Nell’immagine riportata sopra riferita al caso di 3 container identici che girano contemporaneamente, si osserva una prima latenza dovuta alla creazione e lancio dei container. Al termine di questa fase vengono restituiti i valori richiesti dallo script3 per i tre container. In particolare vengono restituiti il numero delle volte in cui si ripete la parola “Dio” nel testo bibbia.txt, il numero delle volte in cui si ripete la parola “bene” e il numero delle volte in cui si ripete la parola “meglio” nel testo Baden.txt, il tempo di esecuzione dello script2, il tempo di esecuzione dello script1 e il tempo di esecuzione totale dei due script. Già da una prima analisi qualitativa, si può vedere che i tempi dei singoli processi sono in parte aumentati, ma più interessante è l’ultimo valore, quello di Elapsed, che indica il tempo di esecuzione dello script.bat che include tutti i tempi necessari a docker-compose per creare/lanciare i container e chiudersi. Quindi la semplice differenza della media dei tempi di esecuzione dello script3 nei container meno la media dei tempi di esecuzione dello script.bat restituisce proprio il valore dei “tempi di processo” di docker-compose.

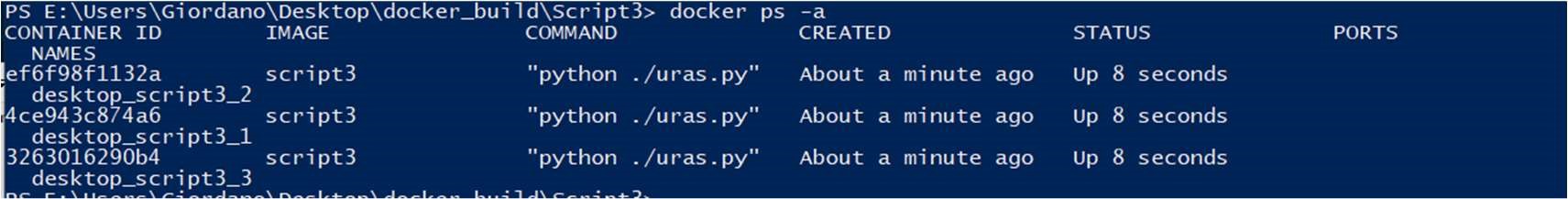
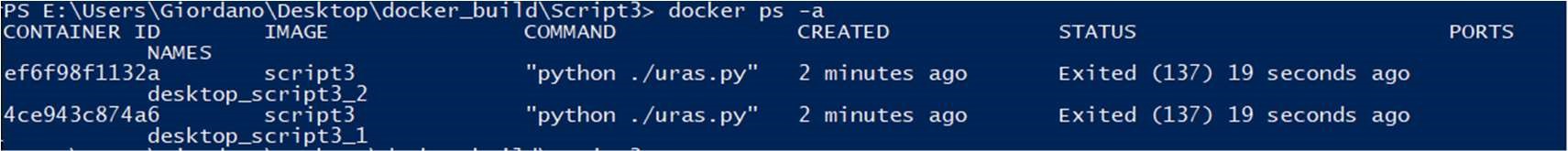
Lanciando nuovamente lo stesso comando senza modificare il numero di scale si osserva che non sarà più presente la fase di creating ma questa verrà sostituita con una fase di starting dei container creati nella prima fase.

*Figura 18: La fase di Creating in docker compose è sostituita dalla fase di Starting*

Si può notare anche in questo caso una diminuzione dei tempi “elapsed” data dall’assenza della fase di creating dei container, docker compose se già ha creato i container nel suo primo run, li tiene dormienti finché non vengono richiamati da una particolare funzione, si vede infatti che non saranno creati ma solamente eseguiti, e si leggerà starting.

Se invece si va a diminuire il numero di container da 3 a 2 si osserva che docker compose effettuerà una funzione di stopping and removing del container desktop\_script3\_3 che era appunto il terzo container che in precedenza aveva creato e utilizzato per il test.

*Figura 19: Stopping and removing da parte di docker compose*

Utilizzando il comando docker ps -a visto in precedenza si può vedere che sono presenti tre container creati nello stesso momento con lo scale impostato a 3 mentre impostando lo scale a 2 si osserva che i container divengono 2 ma effettivamente non sono stati ricreati. Infatti quelli già presenti sono rimasti gli stessi e ciò si dimostra semplicemente andando a confrontare i container ID. *Figura 20: Docker compose ps -a prima e dopo lo scale*

Una volta compreso ciò che accade in docker compose si possono anche qui effettuare le prove più volte.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 container | prova 1 | prova2 | prova3 | prova4 | prova5 | media | compose  per singolo container |
| Processo 1 | 1,4047 | 1,386 | 1,3772 | 1,386 | 1,3911 | 1,389 |  |
| Processo 2 | 3,9913 | 4,2356 | 4,1255 | 3,9602 | 4,1561 | 4,09374 |  |
| processo 1+2 | 5,3961 | 5,6214 | 5,5027 | 5,3463 | 5,5472 | 5,48274 |  |
| processo tot | 7,81 | 8,13 | 7,9 | 7,8 | 8,06 | 7,94 | 7,94 |
| 2 container |  |  |  |  |  |  |  |
| processo 1 | 1,484 | 1,5429 | 1,5624 | 1,508 | 1,4955 | 1,51856 |  |
| Processo 2 | 4,5491 | 4,6537 | 4,6011 | 4,4995 | 4,7006 | 4,6008 |  |
| processo 1+2 | 6,0331 | 6,1967 | 6,1636 | 6,0076 | 6,1971 | 6,11962 |  |
| processo tot | 9,39 | 9,59 | 9,45 | 9,53 | 9,6 | 9,512 | 4,756 |
| 3 container |  |  |  |  |  |  |  |
| Processo 1 | 1,9099 | 1,832 | 2,2179 | 2,0712 | 2,0937 | 2,02494 |  |
| Processo 2 | 6,8717 | 7,2797 | 6,8901 | 6,896 | 6,8852 | 6,96454 |  |
| processo 1+2 | 8,7817 | 9,1118 | 9,1081 | 8,9673 | 8,9789 | 8,98956 |  |
| processo tot | 13,35 | 13,49 | 13,33 | 12,98 | 13,35 | 13,3 | 4,4333 |
| 4 container |  |  |  |  |  |  |  |
| Processo 1 | 3,0424 | 2,9574 | 2,904 | 2,8199 | 2,9977 | 2,94428 |  |
| Processo 2 | 9,194 | 8,9073 | 8,8004 | 8,9941 | 8,7758 | 8,93432 |  |
| processo 1+2 | 12,2365 | 11,8649 | 11,7045 | 11,8141 | 11,7736 | 11,87872 |  |
| processo tot | 17,8 | 17,34 | 16,98 | 16,92 | 16,97 | 17,202 | 4,3005 |
| 5 container |  |  |  |  |  |  |  |
| Processo 1 | 3,5521 | 3,4255 | 3,10167 | 3,4302 | 3,4475 | 3,391394 |  |
| Processo 2 | 11,4432 | 11,6143 | 11,60524 | 11,081 | 11,6664 | 11,482028 |  |
| processo 1+2 | 14,9954 | 15,03988 | 14,7866 | 14,5113 | 15,114 | 14,889436 |  |
| processo tot | 21,24 | 20,89 | 21,07 | 20,51 | 21,47 | 21,036 | 4,2072 |
| 8 container |  |  |  |  |  |  |  |
| Processo 1 | 5,3558 | 5,5466 | 4,8468 | 5,603 | 5,2817 | 5,32678 |  |
| Processo 2 | 17,9632 | 19,4403 | 17,5591 | 18,1129 | 19,0452 | 18,42414 |  |
| processo 1+2 | 23,3191 | 24,987 | 22,601 | 23,715 | 24,0023 | 23,72488 |  |
| processo tot | 32,56 | 33,19 | 31,52 | 32,59 | 32,74 | 32,52 | 4,065 |
| 10 container |  |  |  |  |  |  |  |
| Processo 1 | 6,2061 | 6,8555 | 6,8904 | 6,9278 | 7,0241 | 6,7807 |  |
| Processo 2 | 22,5243 | 24,1814 | 23,538 | 22,5803 | 22,8391 | 23,1326 |  |
| processo 1+2 | 28,8014 | 30,2768 | 30,4285 | 29,5082 | 29,8633 | 29,7756 |  |
| processo tot | 39,72 | 41,37 | 41,41 | 41,07 | 40,4 | 40,794 | 4,0794 |
| 50 container |  |  |  |  |  |  |  |
| Processo 1 | 31,5043524 |  |  |  |  |  |  |
| Processo 2 | 109,3088288 |  |  |  |  |  |  |
| processo 1+2 | 140,223597 |  |  |  |  |  |  |
| processo tot | 191,56 | 191,5 | 194,44 | 193,29 | 196,6 | 193,478 | 3,86956 |
| 100container |  |  |  |  |  |  |  |
| Processo 1 | 54,705 |  |  |  |  |  |  |
| Processo 2 | 199,94 |  |  |  |  |  |  |
| processo 1+2 | 254,68 |  |  |  |  |  |  |
| processo tot | 384,75 | 390,01 | 389,91 | 383,47 | 384,98 | 386,624 | 3,86624 |

*Tabella 3: Tempi di esecuzione con Docker Compose e Tempi operativi di Docker Compose*

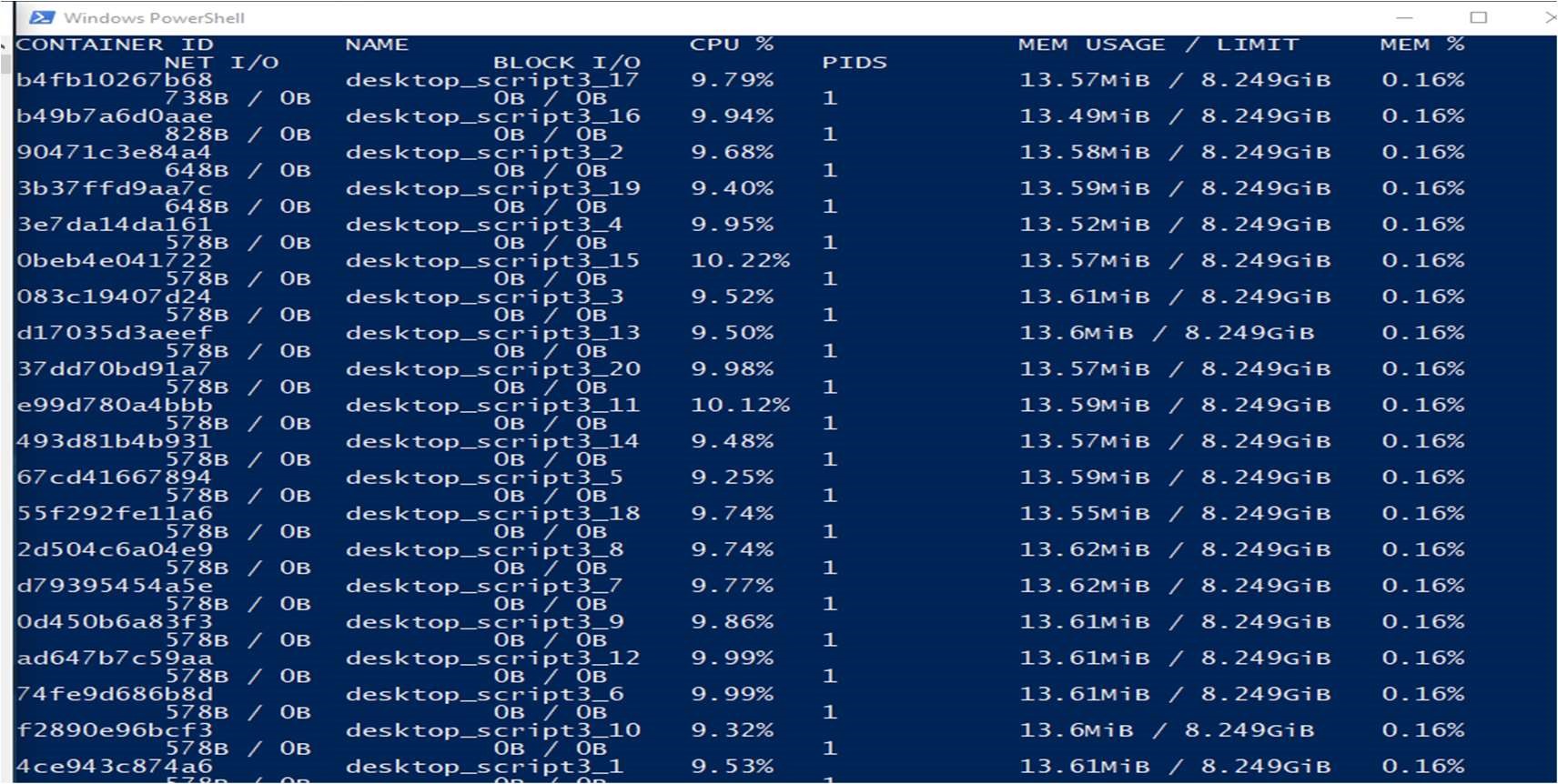
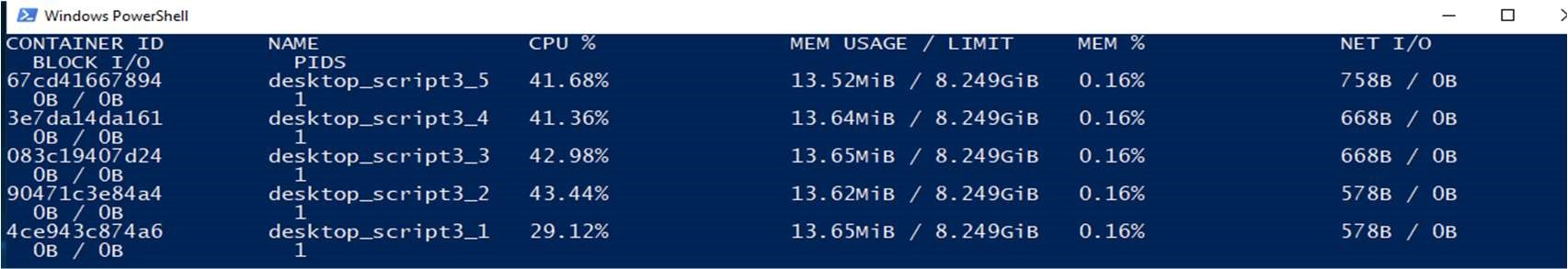
È interessante partire dai tempi di esecuzione di un singolo container (scale=1) con l’utilizzo però di docker compose, si nota che i tempi di esecuzione degli script sono effettivamente coerenti con quelli del test precedente, ma il processo totale, che include apertura e chiusura da parte di docker compose del container, è più lungo, rimanendo coerente con la tesi precedente, docker compose impiega tempo per la gestione dei container.

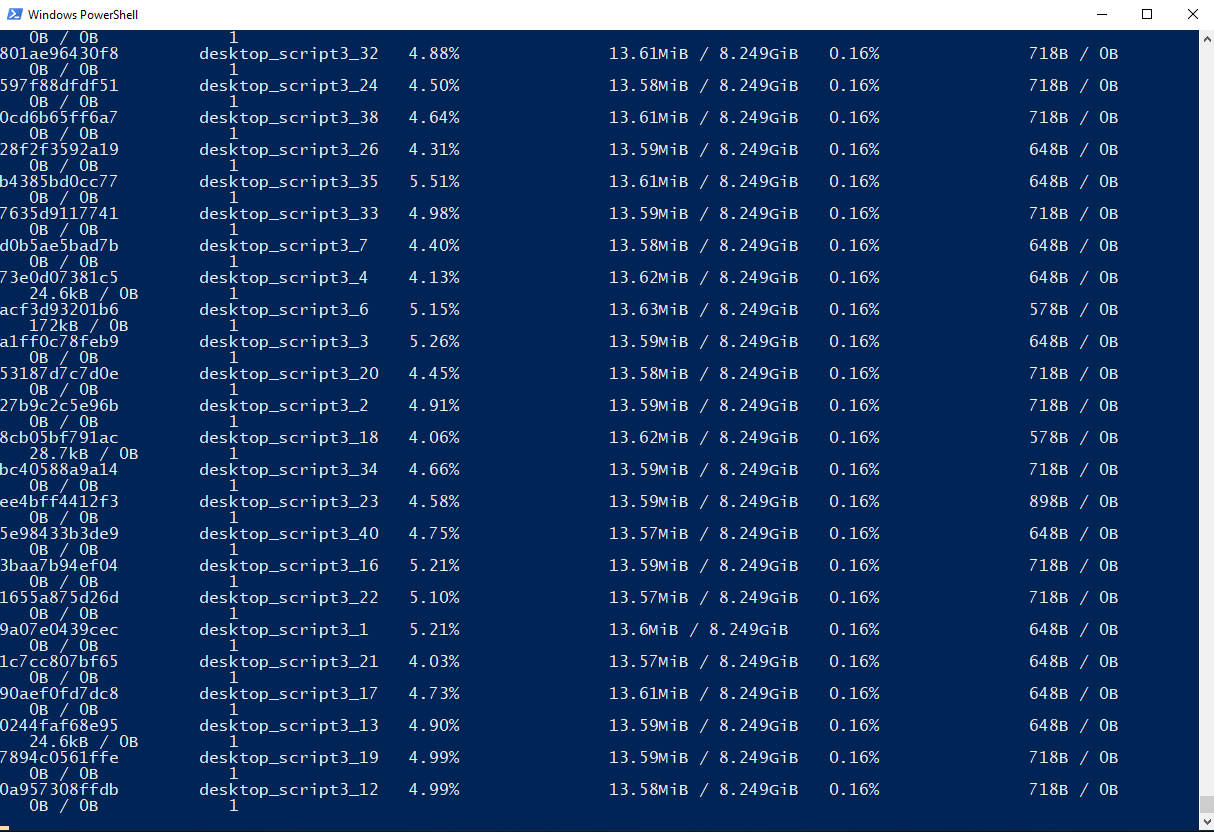
Si osserva ora, attraverso l’utilizzo di un grafico, l’andamento dei tempi di esecuzione all’aumentare del numero di container.

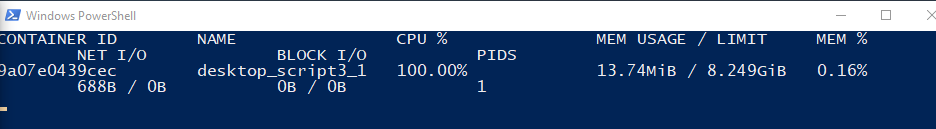
Come si può evincere dalla tabella sono state riprodotte le prove 5 volte, per poi trarre una media dei tempi. (1 solo container \* 5 volte, 2 container \*5 volte e cosi’ via).

*Grafico 1: Andamento dei tempi all'aumentare dei container (dati tabella 3)*

È evidente dal grafico che i tempi di esecuzione si dilatano all’aumentare dei container attivi contemporaneamente, ma che i tempi di esecuzione di docker compose influiscono in modo significativo sul tempo totale di processamento.  
Per comprendere per quale motivo anche i tempi dei singoli processi aumentano si potrebbero analizzare le metriche dei singoli container, utili per comprendere le dinamiche di utilizzo delle infrastrutture di calcolo da parte di docker.  
Se si lancia il comando docker stats -a vengono restituiti i dati per ogni singolo container, identificato proprio dal suo container ID, in merito a uso percentuale della CPU, uso della memoria su uso limite, uso della memoria percentuale, come stream costante di dati nel periodo di attività del container stesso. Di seguito le docker stats per un compose di 5,20 e 40 container.



*Figura 21: Docker stats per un compose di 5,20,40 container*

Si osserva che la CPU percentuale utilizzata dal singolo container diminuisce all’aumentare del numero dei container, si passa infatti da una media del 40% per un compose di 5 applicativi, al 10% per un compose di 20, al 5% per uno di 40, il che comporta un forte aumento dei tempi di calcolo, osservando infatti che il compose di un singolo container porta invece l’utilizzo da parte dello stesso del 100% della capacità di calcolo disponibile, completando in minor tempo il calcolo richiesto.*Figura 22: Docker stats 1 container*

Le percentuali si riferiscono alla CPU del sistema assegnata a Docker. Nel caso specifico sono stati affidati a Docker 2 Core su 4 per questo si vede che la sommatoria delle percentuali di uso della CPU porta al 200% invece che al 100%. Un altro modo comodo per tracciare il flusso delle nostre metriche è attraverso l’utilizzo di cAdvisor, un daemon attivo sviluppato da Google, che colleziona, aggrega, processa e esporta informazioni sui container attivi. Ha supporto nativo su docker e permette di monitorare in tempo reale i container mettendosi in ascolto in localhost sulla porta 8080.

Si osservi infine l’andamento marginale dei tempi totali in rapporto al numero di container () , si può notare un andamento decrescente, proprio dell’effetto di scala, dell’influenza del tempo di gestione di docker compose; diminuisce infatti il “tempo a container” utilizzato dallo strumento compose. Si vede che questo si stabilizza, e descresce molto più lentamente aumentando il numero dei container di molto. Tutte queste considerazioni sono valide, se si prendono in considerazione container che svolgono lo stesso compito e che quindi non sono affetti da variazioni interne dei tempi.

*Grafico 2: tempo totale a container all'aumentare dei container. (dati tabella 3)*

Si osserva in conclusione che le istruzioni di default per la gestione delle risorse da parte dei container sono di tipo greedy, cioè non garantiscono un utilizzo efficiente delle capacità di calcolo ma attuano politiche che tendono a dividere in maniera equivalente tra i container le potenzialità della macchina ospitante.

In docker è possibile limitare le risorse per i singoli container, questo è consigliabile nel caso in cui si incorra nel pericolo di richiamare un’eccezione di Out Of Memory (OOME), perché in quel caso, su Linux, il sistema tenderà a eliminare processi per liberare la memoria, potrebbe però accadere che i processi eliminati siano fondamentali per l’utilizzo della macchina e che quindi anche questa venga spenta e il sistema crolli. Grazie a dei flag inseribili nel comando run di docker è possibili effettuare queste limitazioni.

**-m o – memory** 🡪 permette di settare il limite massimo di memoria utilizzabile dal singolo container

**-- memory-reservation** 🡪 permette di settare un limite più soft del memory, restituisce allarmi nel caso in cui riscontri poca memoria nella macchina ospitante

**--kernel-memory** 🡪 permette di settare il limite massimo di memoria del kernel che il singolo container può utilizzare. Visto che la memoria del kernel non può essere scambiata (to swap), un container che è in attesa sul kernel può effettivamente andare a rallentare l’intero sistema.

Nel prossimo capitolo saranno presentati però metodi più intelligenti per gestire le risorse, che garantiscono un controllo più rigido sui container in gioco.

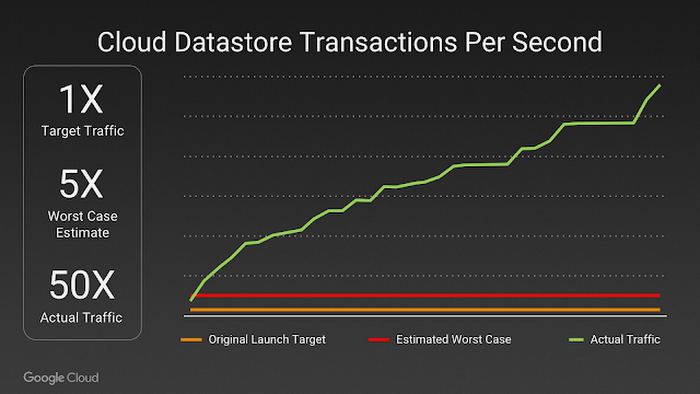
In particolare si descriverà uno dei software più diffusi per la gestione dei cluster di container, molto più densi di quelli visti per i test.

# CAPITOLO 3 Conclusioni e sviluppi futuri dell’elaborato

La tecnologia dei software container sta prendendo piede nelle più diverse situazioni, andando a sostituire in parte, come visto nel primo capitolo, le passate tecnologie di virtualizzazione.  
L’open source è stato anche in questo caso carburante per uno sviluppo rapido e di altissima qualità. Docker in particolare, con una serie di accordi negli anni, ha creato un campo fertile per tutte quelle aziende che volevano portare soluzioni per migliorare la tecnologia dei container, e renderli il nuovo strumento standardizzato per sviluppare gli applicativi di nuova generazione. La gestione di cluster di container per esempio, è uno degli ambiti in cui molte soluzioni sono state proposte. Si pensi, per esempio, al caso in cui si devono gestire centinaia di processi isolati, che si attivano e vengono spenti in lassi di tempi molto brevi e che usano differenti risorse di sistema, tenendo sotto controllo che la qualità del servizio, unico KPI per il cliente, rimanga nei parametri. Una delle soluzioni che si è imposta con il tempo è quella di Google denominata Kubernetes. Di seguito si effettuerà una breve descrizione di questo strumento, pensando che potrebbe essere oggetto di una sperimentazione successiva, tentando di migliorare le performance ottenute senza strumenti di gestione delle risorse.

## 3.1 Kubernetes e Pokemon Go!

Kubernetes è uno strumento open source di orchestrazione per container, sviluppato da Google e rilasciato nella sua versione 1 il 21 luglio 2015. Kubernetes è frutto di anni di sviluppo sul sistema Borg di google per la gestione di cluster di applicativi. L’obbiettivo è di creare una piattaforma che garantisca il lancio, lo scaling e la gestione di container tra cluster di Host. Per comprendere di che numeri si parla quando si lavora con cluster ho voluto prendere in esame un caso applicativo e utilizzarlo per descrivere le potenzialità di questo sistema. Il caso Pokemon GO.

 *Figura 23: Traffico utenti sull'applicativo Pokemon Go*

L’immagine sovrastante rappresenta il grafico con cui inizia il report (Stone, 2016)stilato da google sul caso Pokemon go, commissionato dall’azienda Niantic. Vediamo che il traffico generato dall’applicativo ha superato di 50 volte il traffico target e di 10 il traffico nel worst case stimato dalla stessa Niantic.

Pokemon Go è un mobile game in realtà aumentata, permette, tramite la funzione di localizzazione di scovare, combattere e catturare creature virtuali che popolano il nostro mondo, interagire poi con altri utenti per lottare e ottenere ricompense. Il gioco è stato rilasciato negli stati uniti il 6 luglio 2016 e in una settimana ha raggiunto i 10 milioni di download; solo il 12 luglio aveva ben 21 milioni di utenti attivi divenendo il gioco con più utenti connessi di sempre, l’8 agosto raggiunge i 100 milioni di download in tutto il mondo dopo poco meno di 33 giorni sul mercato. Un vero caso di cluster di applicativi. Tutto questo è stato reso possibile proprio dalla tecnologia dei container, in particolare dall’engine Google Container Engine (GKE), e dallo strumento Kubernetes. Una delle sfide più grandi, racconta Luke Stone direttore del Google Customer Reliability Engineering (CRE), è stato lo sviluppo di una nuova GKE che permettesse l’aggiunta di un migliaio di altri nodi al cluster di container, in preparazione al lancio che si avrebbe avuto in Giappone, non è facile, bisogna non andare ad influire sulle prestazioni dei giocatori che già sono dentro al gioco. In cima a questi upgrade figurava l’aggiornamento del Network Load Balancer, andando a sostituirlo con il nuovo HTTP/S load balancer, che offre più controllo, maggiore throughput, e connessioni più veloci.

Il CRE è un team di ingegneri che si occupa dei clienti della piattaforma cloud andando a risolvere in maniera efficiente ed efficace i problemi legati allo sviluppo di nuovi applicativi che richiedono l’utilizzo della stessa.

Si capisce con questo caso di studio che i container sono la tecnologia abilitatrice ai nuovi servizi digitali globali, servizi che necessitano altissime prestazioni, costi ridotti e elasticità costruttiva.

Si entra ora nel merito dello strumento Kubernetes, facente parte dei progetti della Cloud Native Computing Foundation (CNCF). Kubernetes è una piattaforma open source per la gestione di carico su container, che semplifica sia la configurazione che l’automazione di strumenti virtualizzati, è una piattaforma che orchestra le risorse di calcolo, rete e storage per ottimizzarle sulle necessità degli utenti. Garantisce la semplicità di un Platform as a Service (PaaS) e la flessibilità di una Infrastrutrure as a Service (IaaS).

### 3.1.2 Architettura e funzionamento di kubernetes

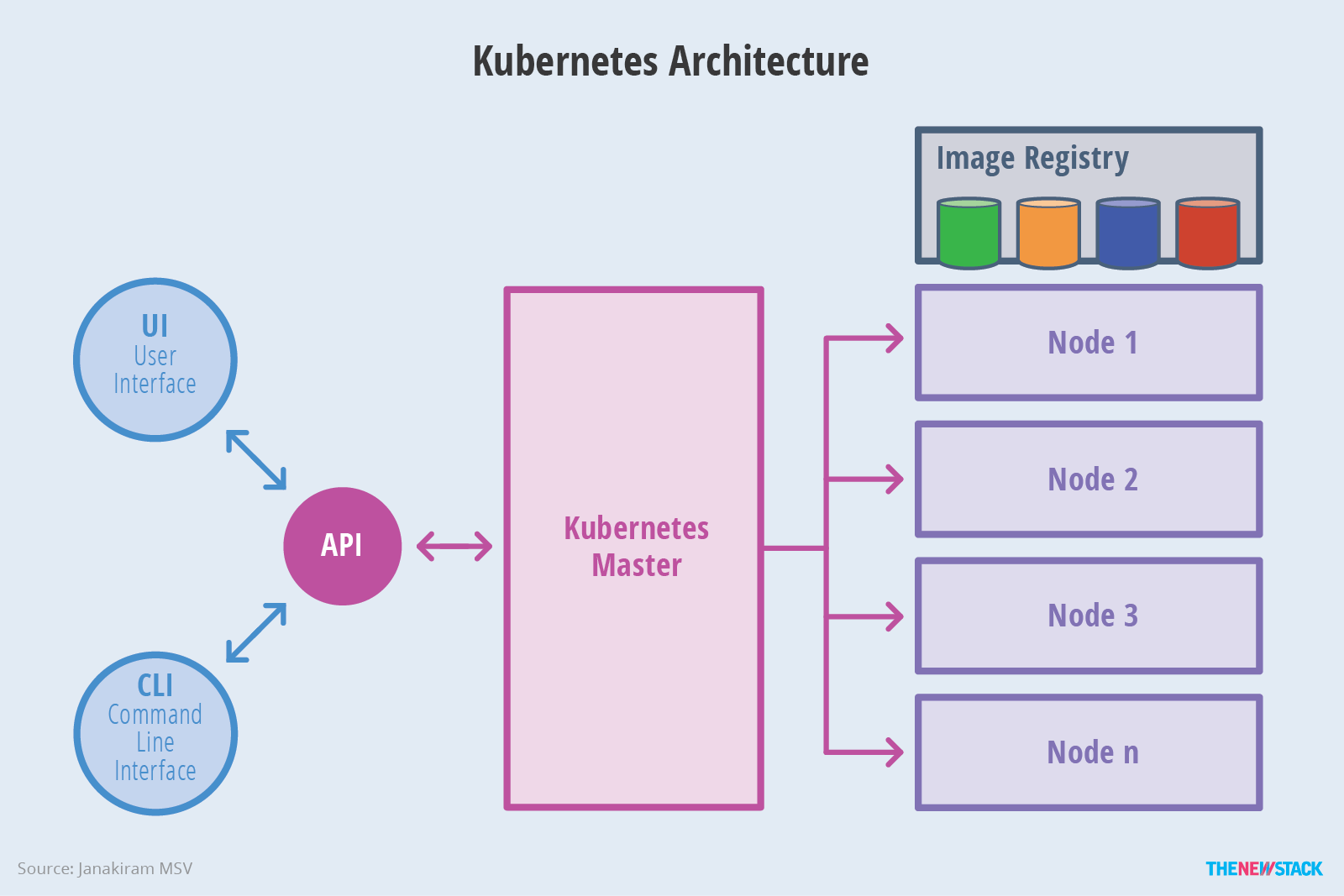
Alla base Kubernetes incapsula insieme macchine fisiche o virtuali isolate in un cluster usando una rete condivisa per metterle in comunicazione, questi cluster sono la piattaforma fisica dove tutti i componenti di kubernetes, le capacità e i carichi sono configurati. Nel sistema, ad ogni server (o a piccoli gruppi se si tratta di un sistema molto grande), viene dato un ruolo. Esistono 2 categorie:

* **Master:** Questi server agiscono come controllori del cluster, espongono API per users e client, controllano lo stato di salute degli altri server e del sistema, effettuano lo scheduling del lavoro, andando ad effettuare il load balancing tra i server e gestiscono le comunicazioni tra gli altri server. I server master, utilizzano servizi per effettuare i vari compiti a loro assegnati. Vengono elencati di seguito

1. **Etcd:** è uno storage di configurazione accessibile a tutti i server nodi, è necessario ai server per riconfigurarsi in maniera coerente.
2. **Kube-apiserver:** è uno degli strumenti più importanti, permette agli utenti di gestire il carico, è responsabile anche di controllare che l’etcd sia aggiornato e accessibile a tutti i nodi.
3. **Kube-controller-manager:** è una funzione generica, controlla che le modifiche apportate all’etcd siano coerenti con il numero di nodi effettivamente in uso, e in caso contrario effettua l’aumento o la diminuzione degli stessi.
4. **Kube-scheduler:** è il processo che effettivamente assegna carico nel cluster, legge il carico richiesto e gestisce le risorse in maniera che non vengano sovracaricate o sottoutilizzate. Tiene traccia delle capacità dell’infrastruttura sottostante per essere pronto a utilizzarla nel migliore dei modi.
5. **Cloud controller manager:** funge da collante tra il sistema generico che gira all’interno della macchina e l’infrastruttura fisica sulla quale effettivamente va ad operare.

* **Nodes:** server preposti ad accettare carico lavorativo utilizzando risorse locali o esterne. Ogni nodo è equipaggiato con container runtime come Docker e quindi ogni servizio è containerizzato e gestito dal master. Ad ogni nodo vengono inviati comandi ed ogni container viene creato, distrutto, reindirizzato, in base a questi. Anche nel caso dei Nodi ci sono dei servizi che vengono utilizzati affinché questi possano lavorare nel migliore dei nodi:

1. **Container runtime:** il componente principale che ogni nodo deve avere, tipicamente il runtime utilizzato è Docker. È responsabile di eseguire e fermare container. Ogni servizio, come detto in precedenza, è sviluppato utilizzando uno o più container contemporaneamente. Il runtime permette di far girare effettivamente il servizio sviluppato.
2. **Kubelet:** è un piccolo servizio che ha il ruolo di punto di contatto tra nodi e il cluster. Ha il compito di comunicare con il piano di controllo e monitorare l’etcd per visualizzare la configurazione del cluster.
3. **Kube-proxy:** è installato su ogni nodo e permette il reindirizzamento del servizio sul nodo esatto, e garantisce che l’ambiente di rete sia accessibile a tutti i nodi e isolato dove serve.

Figura 24: Architettura di Kubernetes

Per concludere con una trattazione introduttiva su Kubernetes si descrivono, di seguito, altre istanze utilizzate all’interno della piattaforma:

* **Pods:** un pod è un insieme di container che vanno controllati come un'unica applicazione, sono container che operano a stretto contatto, condividono il ciclo di vita e devono essere sempre allocati su un unico nodo.
* **Replication controller:** è lo strumento che definisce un template per un determinato pod, andando ad analizzare le sue configurazioni, e permette di scalarlo orizzontalmente per effettuare una distribuzione controllata. Se un pod crolla il replication controller sarà pronto a sostituirlo con un pod identico per garantire il numero necessario alla dimensione del servizio. Permette anche di effettuare upgrade ad applicativi uno alla volta non incidendo sulla disponibilità del servizio in upgrading.
* **Replication set:** svolge gli stessi compiti del replication controller, non permette però di effettuare upgrade. Andrà a sostituire il replication controller poiché garantisce prestazioni migliori.
* **Deployments:** permette di gestire più facilmente il ciclo di vita dei pod replicati. Permette di tenere traccia della storia di un pod così da permettere downgrading facilitati o risoluzione d’errore.
* **Stateful set:** crea un identificatore per ogni pod, permettendo così di individuarlo nella rete anche se questo ha cambiato nodo. È utile per applicativi che utilizzano database in quanto hanno bisogno di un accesso garantito sempre allo stesso set di dati.
* **Daemon set:** permette di attivare uno specifico pod su tutti i nodi o su un sotto insieme degli stessi. È utile quando si vogliono distribuire pods che permettono di mantenere le performance del nodo stesso. Per esempio, analisi delle metriche dei pod, servizi che aumentano le capacità del nodo, sono ottimi candidati per il Daemon set. (Ellingwood, 2018)

### 3.1.3 Possibili Sviluppi futuri dell’elaborato

Nel paragrafo precedente si è visto come la piattaforma Kubernetes, sia stata sviluppata proprio per la gestione di cluster di applicativi. Uno strumento potentissimo per la distribuzione di servizi su scala mondiale.

In uno sviluppo futuro dell’elaborato si potrebbero analizzare gli algoritmi di scheduling utilizzati dal Kube Scheduler per eseguire load balancing, e trovare soluzioni migliorative.

Con questo elaborato si è cercato di dare una panoramica sulle moderne tecnologie di virtualizzazione andando a descrivere i nuovi modelli di gestione dei servizi. I container sono di vastissimo uso ormai in tutte le aree, si potrebbe pensare anche di sviluppare una soluzione per un particolare caso aziendale.

# Bibliografia

Conroy, S. (2018, Gennaio 25). *History of Virtualization*. Retrieved from i don't know read the manual: https://www.idkrtm.com/history-of-virtualization/

Ellingwood, J. (2018, Maggio 2). *An introduction to kubernetes*. Retrieved from Digital Ocean: https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes

Marr, B. (2018, Maggio 21). *How Much Data Do We Create Every Day? The Mind-Blowing Stats Everyone Should Read*. Retrieved from Forbes: https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/#50d4c5d160ba

Matthias, S. P. (2018). *Docker Up & Running.* O'reilly.

Network Function Virtualization, an introduction, benefits, enablers, challenges and call for action. (2012). *SDN and OpenFlow World Congress*, (p. 16). Darmstand-Germany.

Open Cord. (n.d.). *Prerequisites*. Retrieved from open cord: https://guide.opencord.org/prereqs/

Stone, L. (2016, September 29). *Bringing Pokemon GO to life on Google Cloud*. Retrieved from google cloud: https://cloud.google.com/blog/products/gcp/bringing-pokemon-go-to-life-on-google-cloud

Torvald, L. (n.d.). *open containers*. Retrieved from opencontainers: https://www.opencontainers.org/