Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Studio ed Analisi di tecniche di container migration con supporto alla persistenza in scenari di Edge Computing

Relatore:
Chiar.mo Prof.
Marco Di Felice

Presentata da:
Alessandro Rappini

Correlatore:
Dott.
Lorenzo Gigli

Seconda Sessione
2019/2020

Dedico questo percorso a mia Madre e mio Padre,
ai miei nonni,
e a Giulia . . .

Introduzione

Il numero di dispositivi IoT connessi alla rete é in costante aumento e con loro i dati da essi prodotti; ne consegue che la maggior parte dei dati é prodotta ai margini della rete. I dati prodotti da dispositivi terminali, come sensori, devono essere condotti ed elaborati in ambienti terminali come Cloud. Le tradizionali soluzioni di elaborazione e archiviazione dati non saranno piú sufficientemente performanti in quanto la mole di dati crescerá in maniera esponenziale ed i nuovi approcci spingono all'introduzione di uno strato intermedio tra il creatore del dato e il luogo dove il dato deve essere depositato, uno di questi approcci é l'Edge computing. Secondo stime di mercato i dati globali raggiungeranno i 180 zettabyte (ZB) e il 70% dei dati prodotti sará elaborato ai margini della rete entro il 2025; parte dei servizi deve essere quindi spostato dal Cloud ad architetture piú vicine a dove il dato viene prodotto. L'Edge computing presenta una valida soluzione ma la gestione dei servizi in esecuzione al suo interno rappresenta una nuova sfida per il mondo scientifico. Nuovi approcci nella distribuzione di servizi si stanno evolvendo, i quali prevedono che lo stesso non venga piú eseguito direttamente sul calcolatore, ma da un'ambiente ospitato dal kernel che ne emula uno; questo nuovo approccio é offerto dai container. Vari tool di contenerizzazione sono in commercio come Docker, il quale é stato preso come caso di studio per questo lavoro di tesi. I container tra i vari vantaggi che apportano offrono la possibilitá di essere migrati tra calcolatori con una facilitá maggiore rispetto un'applicazione in quanto ad essere spostato non é solo l'applicazione ma tutto il container con annesse le librerie e dipendenze della quale l'applicazione necessita per funzionare. L'intento della migrazione é spostare un container da una macchina a un altra riducendo al minino il tempo di inattivitá e offrendo per l'utilizzatore la possibilitá di riprendere l'esecuzione senza perdita di dati.

L'obbiettivo di questo lavoro di tesi é quello di creare un'applicazione che possa gestire e spostare container Docker di un cluster di nodi Edge. Lo sviluppo della stessa é stato impostato in un'ottica che prevede che vari nodi Edge connessi in rete, utilizzando una tecnica di service discovery, possano creare in autonomia un cluster dove al suo interno sia presente un nodo Leader e diversi nodi Follower. Il protocollo di service discovery implementato nell'applicazione é uPnP il quale permette la creazione di una rete Peer-to-peer tra nodi; uPnP é un'architettura aperta basata sui protocolli standard come TCP, UDP e HTTP. L'elezione del Leader avviene dopo la creazione del cluster, tutti i partecipanti del suddetto si scambiato utilizzando il service discovery un numero, ed a vincere l'elezione é quel nodo col valore maggiore, tale valore viene chiamato indice di fitness, ed é un numero che oscilla tra 0 e 1 ed indica quanto il nodo é performante. Piú il valore di fitness si avvicina ad 1 piú il nodo é performante ed atto a eseguire nuovi processi, viceversa piú si avvicina allo 0 piú il nodo é gravato da operazioni interne che si riperquotono sulle performance della scheda e dei sui servizi in esecuzione. Il valore di fitness é ricavato da un calcolo che prende in input dei pesi, settati in fase di configurazione, e degli indicatori di utilizzo della scheda fisica che ospita l'applicazione come il valore di utilizzo della CPU. Il design dell'architettura é stato studiato per essere fault tolerance nei confronti del Leader e dei nodi Follower in caso di temporanea indisponibilitá di un nodo, lo studio é stato rivolto anche a risolvere problematiche di inserimento di un nodo in un cluster giá formato. Il Leader ha il compito principale di raccogliere i valori di fitness dai nodi Follower del cluster e indire la migrazione di container Docker sulla base di policy studiate ad hoc, trattate nella presente, le quali tengono conto anche di valori fitness specifici su ogni singolo container. Il motivo che ha spinto la creazione dell'applicazione é quello di creare un servizio, che dato un cluster di nodi, indiga varie migrazioni al fine di uniformare i valori di fitness dei vari nodi del cluster per evitare che un nodo sia piú gravato di lavoro rispetto ad altri. La migrazione é stata orientata in un ottica che tenga presente anche dello stato del container, ed in particolar modo della sua persistenza, varie tecniche sono state studiate e implementate per migrare oltre al container anche i sui dati fisici depositati all'interno del filesystem. La migrazione della persistenza ha richiesto un notevole impegno in quanto Docker non ne fornisce uno standard ma solo varie modalitá.

I capitoli all'interno di questo elaborato trattano:

Capitolo 1: Edge Computing, cos'é, perché esiste, i vantaggi che apporta e le sfide che deve affrontare.

Capitolo 2: Cos'e un'ambiente virtuale, cos'é una macchina virtuale e cosa sono i container ed i benefici che apportano.

Capitolo 3: Docker, ne viene discusso cos'é, la sua architettura e come si interfaccia con gli elementi del sistema dove viene eseguito come la rete o il filesystem.

Capitolo 4: Confronto Docker Swarm e Kubernetes, i due principali tool per orchestrare container Docker.

Capitolo 5: Spiegazione del lavoro di tesi, il motivo per la quale é stato svolto e gli intenti.

Capitolo 6: Spiegazione in maniera dettagliata delle principali scelte implementative e come sono state sviluppate ed adattate nell'applicazione.

Capitolo 7: Illustrazione dei risultati ottenuti a fronte di varie casistiche, per studiare in un'ambiente i comportamenti e i cambiamenti dello stesso con le scelte implementative sviluppate.

Indice

Introduzione **i**

I Stato dell'Arte xiii

1 Edge Comupting **1**

1.1 I problemi del Cloud Computing 1
1.2 Edge Computing 3
1.3 L'importanza dell'Edge Computing 4
1.4 Fog computing 6
1.5 Vantaggi dell'Edge Computing 8
1.6 Edge Computing and Cloud Computing 9
1.7 Problematiche aperte e sviluppi futuri del Edge Computing 10
1.8 Virtualizzazione nei nodi Edge 11

2 Ambienti Virtuali e Container **13**

2.1 Ambienti Virtuali 13
2.2 Metodi di Virtualizzazione 14
2.2.1 Virtualizzazione a livello Hardware 15
2.2.2 Virtualizzazione OS level, o container level 15
2.3 Container 15
2.4 Gli Albori dei Container 16
2.5 Confronto fra virtualizzazione e container 17
2.6 Vantaggi dell'uso dei software container 19

3 Docker **21**
3.1 Docker Engine 21
3.2 Docker Registries 22
3.3 Docker Image e Docker Container 23
3.4 DockerFile 24
3.5 Network 25
3.6 Gestione della persistenza 26
3.7 Control Group 27
3.8 Potenzialitá di Docker 27

4 Orchestrazione di container **29**
4.1 Docker Swarm 30
4.1.1 Architettura Docker Swarm 31
4.1.2 Servizi 32
4.1.3 Pianificazione dei task 33
4.1.4 Swarmkit 35
4.1.5 Algoritmo Raft 37
4.2 Kubernetes 39
4.2.1 Architettura di Kubernetes 39
4.2.2 Pod 41
4.2.3 Nodo Master 41
4.2.4 Node 43
4.2.5 Scalabilitá 44
4.2.6 Fault Tolerance 45
4.3 Confronto Docker Swarm e Kubernetes 46

II Obiettivo della tesi **49**

5 Progettazione **51**
5.1 Architettura del sistema 52
5.2 Scelte progettuali 55
5.2.1 com.github.docker-java 55

5.2.2 org.fourthline.cling 56
5.3 Service Discovery 56
5.3.1 UpNp 57
5.3.2 Fasi di UpNp 58

6 Implementazione 61
6.1 Migrazione di un'immagine 61
6.1.1 Migrazione di un volume 64
6.2 Funzione di Fitness 66
6.3 Creazione del cluster ed elezione del Leader 67
6.3.1 Aggiunta di un nuovo nodo 68
6.3.2 Fault Tolerance dell'applicazione 69
6.4 Algoritmi di Migrazione 70
6.4.1 Algoritmo Random 71
6.4.2 Algoritmo Lazy 71
6.4.3 Algoritmo dello Zaino 72

7 Validazione 73
7.1 Elementi dei test 73
7.2 Analisi degli algoritmi con ambiente statico 76
7.2.1 Test Alfa 78
7.2.2 Test Beta 79
7.2.3 Test Gamma 80
7.2.4 Test Combined 81
7.2.5 Analisi sui risultati 82
7.3 Analisi degli algoritmi con ambiente dinamico 84
7.3.1 Test Alfa 85
7.3.2 Test Combined 86
7.3.3 Analisi sui risultati 87
7.4 Analisi sulla resilienza del sistema 88
7.4.1 Casistica Combined 89
7.4.2 Casistica Alfa 89

7.4.3 Analisi sui risultati 90

Conclusioni **93**

A Prima Appendice **97**

A.1 Descrittore XML dell'applicazione UpNP 97

Bibliografia **98**

Elenco delle figure

1.1 I tre divelli dell'Edge Computing 6
1.2 Architettura con Cloud, Fog ed Edge layer 7

3.1 Componenti principali di Docker 23

4.1 Architettura di Docker Swarm 32
4.2 Servizio Scalato su tre nodi 34
4.3 Servizio Scalato su tre nodi 35
4.4 Organizzazione a moduli di Kubernetes 40

5.1 Diagramma dei componenti 54
5.2 Diagramma di Flusso di UpNp 59

6.1 Diagramma di Flusso descrivente la migrazione di un'immagine da un nodo A a un nodo B 64
6.2 Diagramma di Flusso descrivente l'elezione del Leader 69
6.3 Chiamata Docker stat 71

7.1 Grafico valori di fitness con alfa a 1 78
7.2 Grafico valori di discrepanza con alfa a 1 78
7.3 Grafico valori di fitness con beta a 1 79
7.4 Grafico valori di discrepanza con beta a 1 79
7.5 Grafico valori di fitness con gamma a 1 80
7.6 Grafico valori di discrepanza con gamma a 1 80
7.7 Grafico valori di fitness con combinazione allo 0.33 81

7.8 Grafico valori di discrepanza con combinazione allo 0.33 81
7.9 Decremento della funzione di fitness con un aumento di dati sul filesystem 82
7.10 Valori di fitness in un ambiente dinamico con casistica alfa 85
7.11 Valori di fitness del nodo A in un ambiente dinamico con casistica alfa . 85
7.12 Valori di fitness in un ambiente dinamico con casistica combined 86
7.13 Valori di fitness del nodo A in un ambiente dinamico con casistica combined 86
7.14 Analisi sulla resilienza degli algoritmi con un approccio combined 89
7.15 Analisi sulla resilienza degli algoritmi con un approccio alfa 89

Elenco delle tabelle

2.1 Confronto tra container e macchine virtuali 18

4.1 Confronto tra container e macchine virtuali 47

Parte I

Stato dell'Arte

Capitolo 1

Edge Comupting

Negli ultimi anni, con la proliferazione dell'Internet of Things (IoT) e l'ampia diffusione delle reti wireless, il numero di dispositivi connessi alla rete e i dati creati sono cresciuti rapidamente. Secondo la previsione di International Data Corporation (IDC) [19], i dati globali raggiungeranno i 180 zettabyte (ZB) e il 70% dei dati prodotti sará elaborato ai margini della rete entro il 2025. IDC prevede inoltre che oltre 150 miliardi di dispositivi saranno collegati in tutto il mondo entro il 2025. Le tradizionali modalitá di elaborazione come il Cloud o le tipiche infrastrutture client-server non sono abbastanza performanti per gestire la mole di dati generati dai dispositivi visto che il modello di elaborazione centralizzata carica tutti i dati nel data center attraverso la rete e sfrutta tutta la potenza computazionale per risolvere i problemi di elaborazione e archiviazione [18]. Focalizzando l'attenzione nel contesto dell'IoT, il Cloud Computing tradizionale presenta diverse carenze.

1.1 I problemi del Cloud Computing

Una delle consegunze della crescita dei dispositivi IoT é l'aumento di dati che devono essere memorizzati e valutati in tempo reale, che difficilmente puó essere realizzato con soluzioni Cloud. Gli ostacoli principali sono soprattutto il lento ritmo di ampliamento della banda larga e i ritardi nella trasmissione dati tra il Cloud e i dispositivi finali. Nello

specifico le principali difficoltá che l'IoT incontra nell'approccio con il Cloud Computing sono le seguenti:

- **Latenza**: Le applicazioni nello scenario IoT tra i vari requisiti richiedono una comunicazione real time. Nel tradizionale modello di Cloud Computing, le applicazioni inviano i dati al data center e ottengono una risposta che aumenta la latenza del sistema. Ad esempio, i veicoli a guida autonoma ad alta velocitá richiedono una risposta in pochi millisecondi;

- **Larghezza di banda**: La trasmissione di grandi quantitá di dati, generati dai dispositivi al Cloud in tempo reale, provoca una forte pressione sulla larghezza di banda della rete. Ad esempio,un Boeing 787 genera piú di 5 GB / s di dati, ma la larghezza di banda tra un aereo e i satelliti é insufficiente per supportare la trasmissione in tempo reale [10];

- **Prossimitá**: I servizi di elaborazione dati sono deployati all'interno di un'architettura Cloud, questo potrebbe essere un serio problema per dispositivi che operano ai margini della rete, in quanto i protocolli di comunicazione potrebbero non essere compatibili e questo renderebbe impossibile far comunicare una scheda con un servizio remoto;

- **Disponibilitá**: Sempre piú servizi Internet vengono implementati sul Cloud, la disponibilitá di questi servizi é diventata parte integrante della vita quotidiana. Ad esempio, gli utenti di smartphone che si abituano ai servizi vocali, come per esempio Siri o Alexa, si sentiranno frustrati se il servizio non é disponibile per un breve periodo di tempo. Pertanto, una grande sfida per i fornitori di servizi Cloud é quella di mantenere i servizi attivi 365 giorni l'anno 24 ore al giorno;

- **Energia**: I data center consumano molta energia. Con la crescente quantitá di calcolo e trasmissione, il consumo di energia diventerá un collo di bottiglia limitando lo sviluppo dei centri di Cloud Computing.

1.2 Edge Computing

L'Edge Computing cerca di affrontare molti dei problemi legati al Cloud Computing introducendo nuovi paradigmi sulla predisposizione dei sistemi. L'Edge Computing fornisce risorse come la capacitá di memorizzazione dati e offre potenza di calcolo il piú vicino possibile ai dispositivi e sensori che generano dati e rappresenta un'alternativa alle classiche soluzioni Cloud. Il termine Edge dall'inglese significa angolo, estremitá o margine, un'allusione al fatto che con questo approccio l'elaborazione dati non avviene in maniera centralizzata nel Cloud, ma in modo decentralizzato ai margini della rete. Le radici del Edge Computing risalgono alla fine degli anni 90, quando Akamai introdusse le reti di distribuzione dei contenuti (CDN) per accelerare le prestazioni del web [17]. Una CDN utilizza nodi ai bordi di un'infrastruttura, vicini quindi agli utenti per precaricare e memorizzare nella cache i contenuti web. Le CDN sono particolarmente utili per i contenuti video, poiché i risparmi sulla larghezza di banda derivanti dalla memorizzazione nella cache possono essere notevoli. Edge Computing generalizza ed estende il concetto di CDN sfruttando l'infrastruttura di Cloud Computing; Come per le CDN, la vicinanza dei nodi Edge agli utenti finali é cruciale, tuttavia, invece di limitarsi alla memorizzazione nella cache del contenuto Web, un nodo Edge puó eseguire un servizio proprio come nel Cloud Computing. Un Edge device é un device che svolge la funzione di elaborare dati al margine della rete, dati che solitamente vengono generati da board connesse a sensori specifici. L'obiettivo di un nodo Edge é aumentare il tempo di risposta delle applicazioni in esecuzione su dispositivi IoT utilizzando principalmente connettivitá senza fili a bassa latenza e larghezza di banda elevata e ospitando risorse di Cloud Computing, come le macchine virtuali , fisicamente piú vicine ai dispositivi che accedono ad esse. Questo ha lo scopo di eliminare i ritardi di latenza della rete WAN che possono verificarsi nei modelli di Cloud Computing tradizionali. I nodi Edge, in ultima analisi, posso essere banalmente considerati come dei nodi che effettuano una pre-elaborazione dei dati prima che questi vengano inviati al Cloud.

1.3 L'importanza dell'Edge Computing

L'Edge Computing ha lo scopo di fornire ció che il Cloud non é ancora in grado di offrire, un server in grado di gestire a tutto tondo grandi quantitá di dati provenienti da fabbriche intelligenti, reti o sistemi di trasporto senza ritardi e intervenire immediatamente in caso di problemi. Il caso d'uso tipico é quello dei dispositivi IoT, che spesso devono fronteggiare problemi di latenza, mancanza di banda, affidabilitá, non implementabili attraverso il modello Cloud convenzionale. L'architettura di Edge Computing é in grado di ridurre la mole di dati da inviare nel Cloud elaborando i dati critici sensibili alla latenza, nel punto di origine, oppure inviandoli a un server intermedio, localizzato in prossimitá. I dati meno time-sensitive, possono invece essere trasmessi all'infrastruttura Cloud o al data center, per consentire elaborazioni piú complesse, come l'analisi di big data. Un ulteriore aspetto che sottolinea l'importanza di utilizzare nodi Edge é la prossimitá di un nodo con un dispositivo che produce dati. Mentre vengono esplorate nuove applicazioni e vengono utilizzati casi sia per il mobile computing che per l'IoT, i vantaggi della prossimitá stanno diventando sempre piú evidenti. Poiché la vicinanza della rete logica é interamente caratterizzata da bassa latenza, basso jitter e larghezza di banda elevata, la domanda del mondo scientifico si sta concentrando su quanto é fisicamente abbastanza vicina ad una risorsa che produce dati a un'altra risorsa abile a raccoglierli e ad elaborarli. La risposta non é conosciuta, e il tema non é banale, in quanto non é possibile rispondere in astratto, dipende da fattori come le tecnologie di rete utilizzate, la contesa di rete, le caratteristiche dell'applicazione e la tolleranza dell'utente per una scarsa risposta interattiva. La vicinanza fisica influisce sulla latenza end-to-end, sull'ampiezza di banda economicamente praticabile e sull'affidabilitá. Con sforzi e investimenti sufficienti, la mancanza di prossimitá puó essere parzialmente mascherata. Ad esempio, una connessione in fibra diretta puó ottenere bassa latenza e larghezza di banda elevata tra punti distanti. Tuttavia, ci sono limiti a questo approccio, come la velocitá della luce é un evidente limite fisico alla latenza. La necessitá di utilizzare una strategia di rete multihop per coprire una vasta area geografica con molti punti di accesso, impone un limite economico sia sulla latenza che sulla larghezza di banda. Ogni hop introduce un ritardo nell'accodamento e nell'instradamento. La vicinanza di un nodo Edge a un nodo che produce i dati offre molti benefici. L'Edge Computing é in grado di ridurre

larghezza di banda di ingresso nel Cloud, particolarmente rilevante, per esempio, nel contesto della guida autonoma, poiché un automobile in movimento produce moltissimi dati. I nodi Egde potrebbero eseguire analisi in tempo reale di tutti i dati provenienti dal mezzo, come i sensori connessi col motore e altre fonti, per avvisare il conducente di guasti imminenti o della necessitá di manutenzione preventiva [17]. Una volta elaborati e catalogati, le informazioni potrebbero essere trasmesse al Cloud per l'integrazione nel database del costruttore del veicolo. Cosi facendo il quantitativo di dati che il Cloud riceve é molto minore e i dati hanno una valenza piú significativa. L'Edge computing é in grado di eseguire l'archiviazione dei dati, la memorizzazione nella cache ed elaborazione, nonché distribuire richieste e fornire servizi dal Cloud all'utente. Il modello di Edge Computing puó essere definito a tre livelli come si puó vedere nella figura 1.1. Il primo livello é formato dai dispositivi IoT, come per esempio droni, sensori , dispositivi ed elettrodomestici domotici o apparecchiature appartenente al campo industriale. Protocolli di comunicazione multipli sono usati per connettere dispositivi IoT e il secondo livello, Edge. Ad esempio, i droni possono connettersi a una torre cellulare tramite 4G / LTE e i sensori nella casa intelligente possono comunicare con il gateway di casa tramite MQTT. I nodi Edge richiedono enormi capacitá di elaborazione e archiviazione del Cloud per completare attivitá complesse. I protocolli tra IoT e Edge hanno generalmente le caratteristiche di basso consumo energetico e breve distanza, mentre i protocolli tra Edge e Cloud hanno un throughput elevato e un'alta velocitá. Ethernet, fibre ottiche e il prossimo 5G sono le modalitá di comunicazione tra Edge e Cloud piú preformanti.

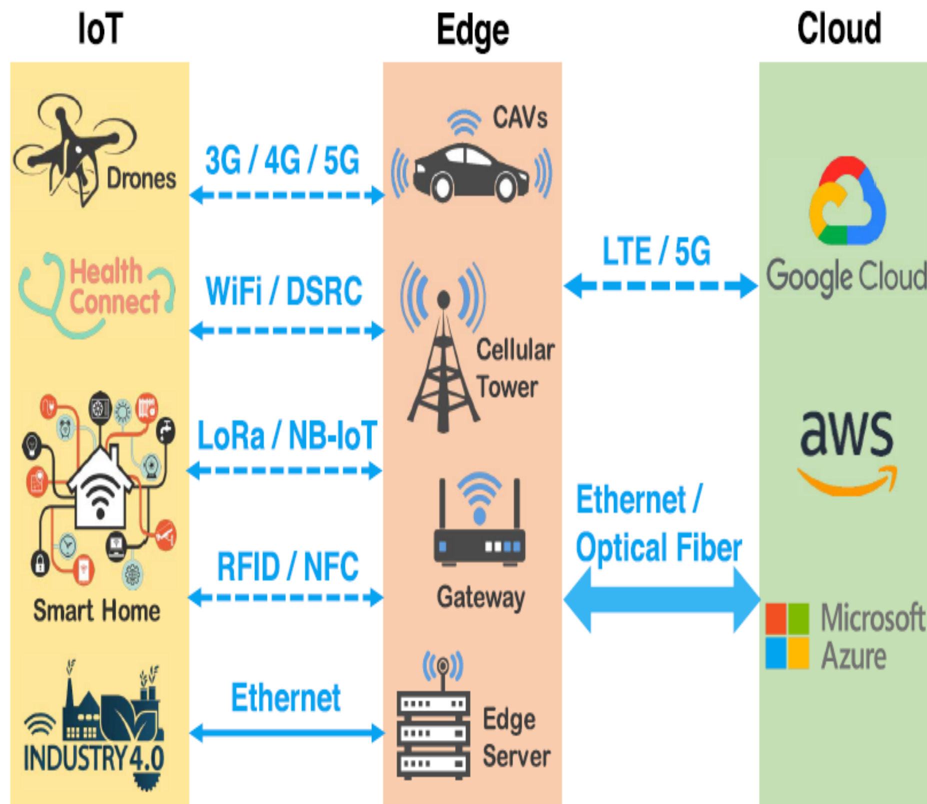

Figura 1.1: I tre divelli dell'Edge Computing

1.4 Fog computing

Il Fog Computing é stato introdotto per la prima volta da Cisco nel 2012 come un concetto che "estende il paradigma del Cloud computing ai confini della rete "[2]. I nodi Fog sono dei mini datacenter che sono messi davanti al Cloud e rappresentano cosí un livello intermedio nella rete. Il suo intento principale é quello di aggiungere storage distribuito e strutture di calcolo al fine di avvicinare i servizi basati su Cloud ai dispositivi finali. In altre parole, il Fog computing realizza una distribuzione di funzionalitá fondamentali come archiviazione, calcolo e processo decisionale, avvicinandolo al luogo in cui vengono prodotti i dati di input, ed i dati di output vengono consumati. I dati generati negli ambienti IoT quindi non vengono piú trasferiti direttamente al Cloud, ma vengono dapprima riuniti in nodi Fog, valutati e selezionati per le successive fasi di elaborazione. L'

Edge Computing é una parte del Fog computing dove risorse come la potenza di calcolo e la capacitá di memorizzazione si avvicinano ancora di piú ai dispositivi IoT ai margini della rete. Mentre l'elaborazione dei dati viene eseguita dal Nodo Edge. Nell'immagine 1.2 é illustrata un'architettura con Cloud, Fog ed Edge layer.

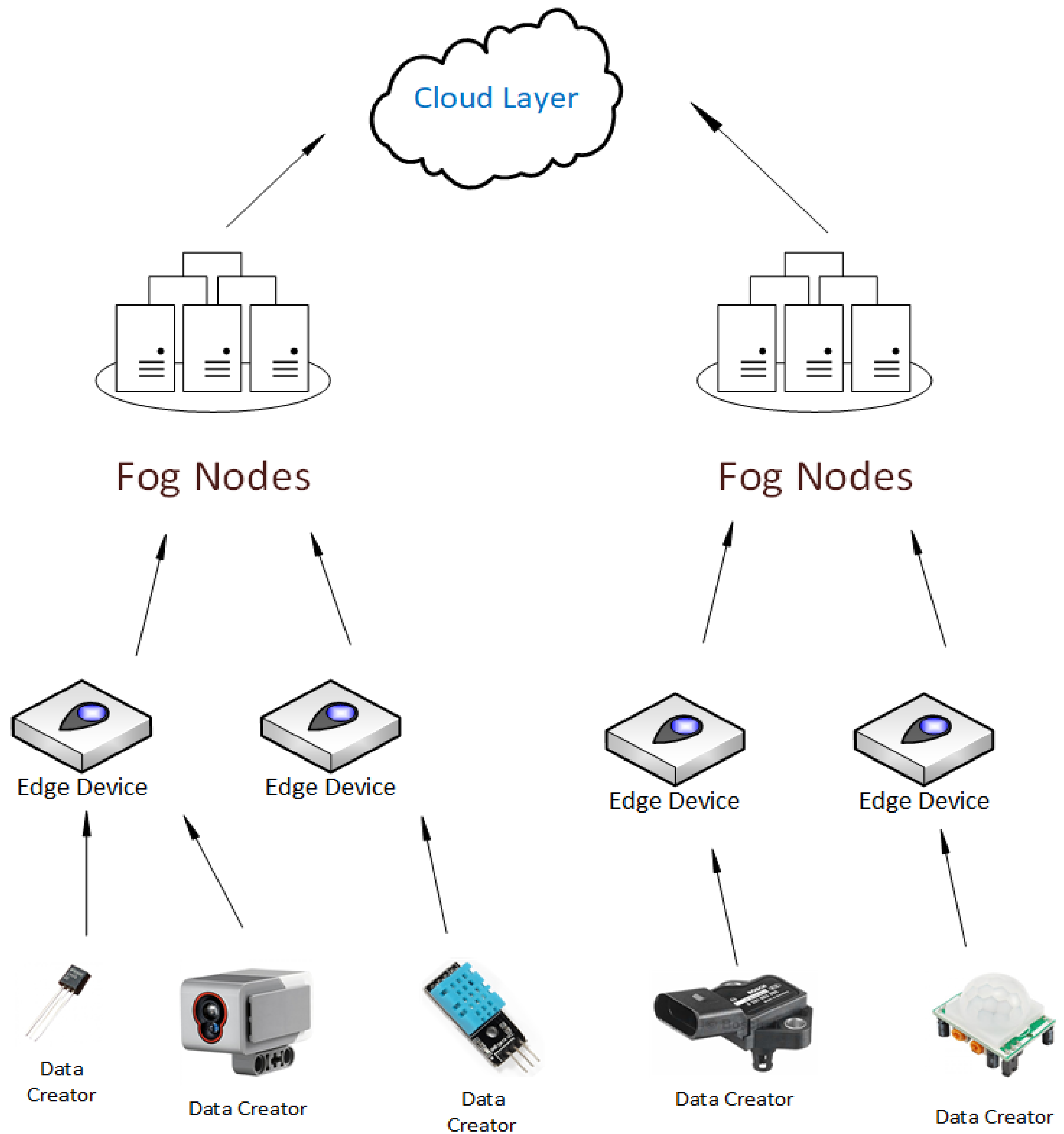

Figura 1.2: Architettura con Cloud, Fog ed Edge layer

1.5 Vantaggi dell'Edge Computing

I vantaggi dell'Edge Computing sono i seguenti:

1. **Prossimitá**: uno degli aspetti piú utili dell'utilizzo del Edge computing é il vantaggio di avere una risorsa di rete abile a raccogliere i dati molto vicino a dove gli stessi sono stati prodotti, questo rende piú veloce la comunicazione. Ad esempio, un nodo Edge potrebbe essere connesso al router che fornisce la connessione wireless a uno smartphone, ne consegue che la latenza per l'erogazione del servizio puó essere notevolmente ridotta, con notevoli vantaggi per le applicazioni in tempo reale, ad esempio la Realtá aumentata. Inoltre é possibile eseguire calcoli locali per quelle applicazioni che generano dati rilevanti solo nelle vicinanze;

2. **Raccolta e aggregazione di dati**: mentre le sorgenti di dati nelle classiche architetture Cloud trasferiscono tutti i dati a un'unitá di calcolo centrale al Cloud per una valutazione centralizzata, l'Edge Computing si basa sulla raccolta di dati in prossimitá dell'origine. A tale scopo, i microcontrollori sono collegati direttamente al nodo Edge. I nodi Edge combinano dati provenienti da dispositivi diversi e consentono la pre elaborazione e l'archiviazione di dati. Il caricamento nel Cloud avviene dopo che le informazioni sono state valutate e aggregate localmente;

3. **Memorizzazione locale dei dati**: l'Edge Computing é particolarmente utile quando bisogna fornire dati a banda larga a livello locale. Con grosse quantitá di dati, la trasmissione in tempo reale dall'unitá di calcolo centrale al Cloud di solito non é possibile. Questo problema puó essere superato memorizzando i dati corrispondenti in maniera decentrata ai margini della rete. In un contesto del genere, gli Edge device fungono da server di replica;

4. **Monitoraggio basato su intelligenza artificiale**: le unitá di calcolo decentralizzate in un ambiente di Edge Computing ricevono e valutano i dati, consentendo un monitoraggio continuo dei dispositivi collegati. Con gli algoritmi di Machine Learning é possibile monitorare in tempo reale lo stato delle macchine, ad esempio per controllare e ottimizzare i processi nelle fabbriche intelligenti;

5. **Comunicazione M2M**: l'abbreviazione "M2M" sta per "Machine-to-Machine", che indica lo scambio automatico di informazioni tra terminali per mezzo di qualsiasi standard di comunicazione. In ambienti IoT, come una fabbrica intelligente, la comunicazione M2M potrebbe essere utilizzata per il monitoraggio remoto di macchine e impianti. Nell'ambito del controllo di un processo avviene sia la comunicazione tra terminali sia la comunicazione con un'unitá centralizzata, che funge da istanza di controllo.

1.6 Edge Computing and Cloud Computing

Edge Computing e Cloud Computing sono soluzioni complementari nell'attuale modello dell'IoT. L'ubiquitá dei dispositivi intelligenti e il rapido sviluppo della moderna virtualizzazione e tecnologie Cloud, hanno portato in primo piano l'Edge Computing, definendo una nuova era per il Cloud Computing. Il Edge Computing ha bisogno di un'elevata potenza di elaborazione e di un enorme supporto di archiviazione come un centro di Cloud Computing, e il centro di Cloud Computing ha bisogno del modello di Edge Computing per elaborare dati di grandi dimensioni e snellirli quando molti nodi sono connessi. Edge Computing presenta numerosi vantaggi nei confronti del Cloud Computing, poiché, una grande quantitá di dati temporanei viene elaborata ai margini della rete, e non tutti i dati vengono caricati sul Cloud, il che riduce notevolmente la pressione sulla larghezza di banda della rete e il consumo energetico del data center. Un ulteriore punto di nota é che, l'elaborazione dati presso chi li produce non richiede la risposta del centro di Cloud Computing attraverso la rete, il che riduce notevolmente la latenza del sistema e migliora la capacitá di risposta del servizio. Un ulteriore vantaggio dell'utilizzo del modello Edge Computing in relazione con un datacenter Cloud é quello di mascherare le interruzioni del Cloud. Con l'aumentare della dipendenza dal Cloud, aumenta anche la vulnerabilitá delle sue interruzioni. Ci sono contesti di utilizzo in cui l'accesso al Cloud deve essere visto come un lusso occasionale piuttosto che una necessitá di base. Un esempio é un paese in via di sviluppo con una debole infrastruttura di rete. I nodi Edge possono alleviare le interruzioni del Cloud, a causa della loro vicinanza fisica, le caratteristiche di sopravvivenza di un nodo Edge sono maggiori quando sono piú vicini

ai dispositivi associati che al Cloud. Ció apre le porte agli approcci in cui un servizio di fallback su un nodo Edge puó temporaneamente mascherare l'inaccessibilitá del Cloud [17].

1.7 Problematiche aperte e sviluppi futuri del Edge Computing

Edge Computing offre chiaramente molti vantaggi, ma allo stesso tempo, deve affrontare anche molte sfide tecniche e non. Dal punto di vista tecnico, ci sono molte incognite relative ai meccanismi software, agli algoritmi necessari per il controllo collettivo e la condivisione nel calcolo distribuito. Fattori come l'ambiente di esecuzione, i requisiti dell'applicazione e i modelli di mobilitá degli utenti devono essere presi in considerazione quando si valutano i possibili luoghi di distribuzione. Anche fornire componenti di rete con strutture computazionali generiche potrebbe essere un compito difficile. Ci sono anche ostacoli sostanziali nella gestione dell'infrastruttura Edge distribuita. Una delle forze trainanti del Cloud Computing é il minor costo di gestione dell'infrastruttura centralizzata, la dispersione inerente al Edge Computing aumenta notevolmente la complessitá della gestione, lo sviluppo di soluzioni tecniche innovative per ridurre questa complessitá é una prioritá di ricerca per l'Edge Computing. Un'altra importante area di studio é lo sviluppo di meccanismi per compensare la debole sicurezza dei nodi, rispetto ai data Center Cloud. I nodi progettati per attivitá piú semplici potrebbero diventare vulnerabili ad attacchi dannosi, la privacy e l'autenticazione in un ambiente cosí distribuito sarebbe difficile da garantire senza indurre un degrado significativo delle prestazioni. Un'altra sfida riguarda il fatto che l'infrastruttura verrá probabilmente implementata in un ambiente con risorse limitate, lo sviluppo di applicazioni in un ambiente con risorse limitate é piú difficile rispetto ai data center convenzionali. L'emergere del Edge Computing coincide con tre importanti tendenze nel panorama dell'informatica e della comunicazione che, nonostante siano guidate da forze distinte, sono convergenti. Una tendenza é la rete definita dal software (SDN) e il concetto associato di virtualizzazione delle funzioni di rete (NFV), che deve essere supportato da alcune delle stesse infrastrutture virtualizzate del Edge Computing. Una seconda tendenza é l'interesse crescente per le reti wireless a

bassissima latenza. La latenza ultraleggera é uno degli attributi proposti per le future reti 5G. Edge Computing é un partner naturale delle reti 5G. Una terza tendenza é il continuo miglioramento delle capacitá di elaborazione di dispositivi indossabili, smartphone e altri dispositivi mobili che rappresentano il limite estremo di Internet. Sebbene questi dispositivi stiano effettivamente aumentando la potenza di calcolo, i loro miglioramenti sono attenuati dalle sfide fondamentali della mobilitá come peso, dimensioni, durata della batteria e dissipazione del calore.

1.8 Virtualizzazione nei nodi Edge

La virtualizzazione ha lo scopo di consentire l'esecuzione simultanea di piú sistemi operativi i quali condividono una piattaforma hardware comune. Questo disaccoppiamento é realizzato da un componente software chiamato Virtual Machine Monitor (VMM) o hypervisor, il quale verrá spiegato meglio nel prossimo capitolo. L'hypervisor funge da mediatore nelle interazioni tra le macchine virtuali e le macchine sottostanti. Il VMM fornisce un'astrazione dell'hardware fisico e la sua responsabilitá é fornire alle macchine virtuali tutte le risorse virtuali necessarie per il funzionamento (ad esempio CPU, memoria e dispositivi I / O). L'adozione di macchine virtuali nel Edge computing puó portare tutti i vantaggi di lavorare con ambienti virtuali, come la possibilitá di implementare un servizio insieme al suo ambiente di esecuzione, il che semplifica notevolmente il processo di avvio. Poiché ogni macchina virtuale é gestita dal proprio sistema operativo, é anche possibile distribuire applicazioni progettate per piattaforme distinte. Le macchine virtuali forniscono anche eccellenti proprietá di isolamento. In caso di guasto o attacco informatico, ad esempio, il resto dei servizi non verrebbe influenzato. Nonostante i loro vantaggi, le macchine virtuali mostrano anche limitazioni sostanziali in particolar modo nel mondo dell'Edge Computing. Uno dei principali svantaggi delle macchine virtuali é il degrado delle prestazioni. Lo strato intermedio introdotto dalla tecnica di astrazione genera un notevole overhead. L'impatto, che é ben evidente sulle macchine desktop, sarebbe anche peggiore nei dispositivi con risorse limitate come i nodi Edge. In secondo luogo, le dimensioni di una macchina virtuale sono generalmente piuttosto grandi. In effetti, il file, che contiene tutte le informazioni relative a una macchina

specifica, puó richiedere fino a diversi gigabyte di spazio su disco, ed é probabile che le capacitá di archiviazione di un semplice punto di presenza non siano in grado di far fronte a tali requisiti. A causa delle loro dimensioni, il trasferimento di macchine virtuali tra host fisici puó essere un'operazione molto lenta. Poiché i nodi di un'infrastruttura Edge il piú delle volte non dispongono di una rete dedicata ad alta velocitá né di hardware performante, l'implementazione di un servizio puó subire ritardi inaccettabili. Nonostante non siano presenti degli standard all'interno nei nodi Edge predilige l'utilizzo di tecniche di virtualizzazione piú leggere, come i container.

Capitolo 2

Ambienti Virtuali e Container

Gli Ambienti Virtuali sono una pratica di sviluppo e di rilascio nota come virtualizzazione la quale permette di creare degli spazi indipendenti dal resto del sistema. La grande popolaritá degli Ambienti Virtuali, é dovuta al fatto che consentono di lavorare con piú applicativi contemporaneamente nonostante questi utilizzino una versione diversa degli stessi moduli o addirittura del kernel, senza che essi abbiano conflitti.

2.1 Ambienti Virtuali

Le origini delle prime macchine virtuali risalgono all'IBM negli anni '60, dove il concetto di virtualizzazione fu originariamente introdotto per partizionare le risorse hardware in modo piú efficiente. Questo avveniva ben prima dell'inizio della maggior parte dei sistemi operativi nei primi anni '70, che furono inizialmente introdotti come alternativa software per partizionare ed esporre le risorse hardware a piú utenti. Alla fine degli anni '80 e '90, gli hypervisor software hanno permesso il multiplexing dell'hardware dei calcolatori e hanno consentito l'uso di macchine virtuali, che negli anni 2000 sono diventate una merce che doveva essere orchestrata in modo efficiente e approvvigionata in blocco. Un hypervisor permette a un computer host di supportare piú VM guest attraverso la condivisione virtuale delle risorse come quelle di elaborazione e memoria, sostanzialmente é software adibito alla gestione delle risorse necessarie. Conosciuto anche come Virtual Machine Monitor (VMM), assegna memoria, spazio su disco fisso, componenti di rete o

potenza di elaborazione all'interno del sistema, in questo modo, diverse macchine virtuali possono funzionare su un unico sistema host, poiché l'hypervisor assicura che esse non interferiscano tra loro e che tutte abbiano a disposizione le capacitá necessarie. Esistono due tipi di hypervisor, le quali sono:

Hypervisor di tipo 1

Il primo tipo di hypervisor é chiamato Bare Metal Hypervisor o Native Hypervisor. Questa forma di virtualizzazione é implementata direttamente sull'hardware fisico e non é collegata al sistema operativo ospitante. Il consumo di risorse di un hypervisor di tipo 1 é relativamente basso perché la potenza di calcolo non deve necessariamente passare attraverso il sistema operativo host. Questo tipo di hypervisor é indicato principalmente per gli use case che richiedono la creazione di un server per la virtualizzazione, mentre é troppo complesso e oneroso per progetti piú piccoli.

Hypervisor di tipo 2

La seconda variante, nota anche come Hosted Hypervisor, richiede un sistema operativo esistente, che a sua volta si basa sull'hardware fisico. L'hypervisor di tipo 2 viene quindi eseguito come un banale programma. I driver dei dispositivi non devono essere installati nell'hypervisor, poiché il sistema operativo reale puó semplicemente trasmetterli al software. Tuttavia, questo vantaggio va a discapito delle prestazioni. Una gran parte delle risorse viene persa nel sistema operativo host. Grazie alla loro semplice installazione e configurazione, gli hypervisor di tipo 2 si adattano perfettamente per progetti di piccole dimensioni.

2.2 Metodi di Virtualizzazione

I metodi di virtualizzazione principalmente utilizzati sono due: virtualizzazione a livello hardware e a livello di sistema operativo.

2.2.1 Virtualizzazione a livello Hardware

Questo approccio si preoccupa di virtualizzare il sistema fisico sul quale eseguire il servizio. Per applicare questo metodo si ricorre all'uso delle macchine virtuali. Una macchina virtuale é un software che simula un sistema operativo, chiamato sistema guest, dando possibilitá all'utente di ricorrere a tutte le funzionalitá offerte dal sistema operativo simulato. Questo sistema é eseguito sul kernel ed il virtualizzatore é installato su di esso, il quale garantisce indipendenza fra i due, garantendo che ció che viene eseguito nel sistema virtuale sia confinato al suo interno. Per fare ció il sistema operativo host dedica una parte di risorse hardware al sistema guest, anche se questo non ne disporrá in maniera completa. É possibile eseguire piú sistemi virtuali su una stessa macchina fisica piuttosto che su macchine diverse, permettendo di abbattere cosí i costi legati all'hardware che altrimenti un'azienda dovrebbe sostenere.

2.2.2 Virtualizzazione OS level, o container level

Questo approccio permette di eseguire applicazioni e servizi in un ambiente isolato, differente dal modello di virtualizzazione della macchina virtuale, la quale virtualizza fisicamente un intero sistema. Questo metodo di virtualizzazione si dice che virtualizza il software, in pratica si tratta di creare ambienti isolati chiamati container nel quale eseguire le applicazioni. I container istanziati non sfruttano un sistema operativo ospite, ma sfruttano direttamente il kernel del sistema operativo sottostante, con chiamate di sistema. É possibile eseguire piú container nello stesso host, i quali condivideranno le risorse hardware disponibili, ed utilizzandole solo all'occorrenza.

2.3 Container

Un container é uno spazio software in cui viene eseguita un'applicazione in modo isolato ed indipendente, la quale dispone di proprie risorse software in condivisione con quelle presenti sul host dove il container viene eseguito. Un container nasce da un'immagine, che definisce il servizio che deve essere eseguito, con la relativa configurazione dell'ambiente d'esecuzione. L'approccio di eseguire le applicazioni in questo genere di

ambienti mira a soddisfare e risolvere determinate necessitá e problematiche legate soprattutto alla gestione, distribuzione e manutenzione dei software. Un container esegue istruzioni native della CPU, eliminando la necessitá di emulazione del livello hardware di istruzione, consentendo di risparmiare sul consumo di risorse senza il sovraccarico della virtualizzazione, fornendo allo stesso tempo isolamento e guadagnano molto in termini di velocitá. La differenza principale tra la virtualizzazione e l'uso dei container é la seguente. La virtualizzazione consente di eseguire piú sistemi operativi (Windows o Linux) contemporaneamente su un singolo sistema hardware, mentre i container condividono lo stesso kernel del sistema operativo e isolano i processi applicativi dal resto del sistema. La virtualizzazione utilizza un hypervisor per emulare l'hardware che consente di eseguire piú sistemi operativi in modalitá affiancata, procedura non ottimale come l'uso dei container. Disponendo di risorse limitate con capacitá limitate, sono necessarie applicazioni leggere che possano essere distribuite su larga scala. I container Linux rappresentano un passo avanti nella modalitá di sviluppo, deployment e gestione delle applicazioni.

2.4 Gli Albori dei Container

Nel 1979 fu introdotta nei sistemi Unix-like, la chiamata di sistema chroot (change root), la quale é in grado di cambiare la directory di riferimento dei processi in esecuzione e di quelli generati da questi ultimi. Con questa chiamata era possibile associare ai processi degli spazi isolati nello storage in uso [10]. Nel 2000 fu introdotto FreeBDS jail, questa soluzione, aggiungeva funzionalitá di sandboxing per l'isolamento dei file di sistema, degli utenti e della rete consentendo di assegnare determinate configurazioni software ed IP a ciascuna jail. Nel 2001 uscii Linux VServer , tecnologia simile alla jail ma permetteva inoltre la suddivisione delle risorse di un computer come il file di sistema, indirizzi di rete, memoria, CPU time in partizioni chiamate security context [10]. Nel 2004 Sun Microsystem rilasció Solaris Container, e per la prima volta si parló di container. I container della Sun Microsystem combinavano il controllo delle risorse di sistema con ambienti isolati in esecuzione su un'unica istanza del sistema operativo. Nel 2006 Google sviluppa Process container, il quale sfrutta i Control groups "cgroups", i quali verranno discussi in questo capitolo. Di base sono una funzionalitá che permette

di limitare, registrare ed isolare l'utilizzo delle risorse (CPU, memoria, disco I/O, ecc) di gruppi di processi, fornendo cosí un accesso controllato all'hardware da parte dei processi. Nel 2008 sono presentati i LinuX Container, la prima soluzione che offre un completo sistema di container dato che sfruttano i cgroups e i namespace di Linux, e non richiedono l'installazione di componenti aggiuntivi al kernel. Nel 2011 Warden propose una piattaforma che sfruttasse inizialmente container LXC, per poi utilizzare qualche tempo dopo una sua versione, la quale architettura si basa sul modello client server, dove il demone, server, agisce sui container soddisfando le richieste inviate dal client, interfacciandosi con delle API. LXC é una tecnologia di virtualizzazione a livello di sistema operativo che crea un ambiente virtuale completamente sandbox in Linux senza l'overhead di una macchina virtuale a tutti gli effetti [3]. Nel 2013 fú lanciato il progetto LMCTFY (Let Me Contain That For You) versione open source di un progetto di Google, utilizza cgroups, come molte soluzioni container, ma la sua particolaritá é che ogni applicazione é container aware, significa che le applicazioni eseguite in questi contesti sono in grado di eseguire e gestire dei propri sotto container. Il progetto termina nel 2015, quando Google decide di contribuire alla nuova libreria di Docker libcontainer, inserendo in questa alcune parti appartenenti al progetto ormai chiuso.

2.5 Confronto fra virtualizzazione e container

I containers si differenziano dalle macchine virtuali soprattutto per la loro "leggerezza", infatti una macchina virtuale, necessita di un sistema operativo ospite che viene eseguito sul sistema operativo host, e di un controllore chiamato hypervisor per il suo funzionamento, mentre un container sfrutta il kernel e le librerie del sistema operativo sottostante per virtualizzare un ambiente privato di esecuzione, condividendo le risorse (CPU, Ram, rete, memorie di massa..) con il resto delle applicazioni/container. L'articolo [8] offre un confronto tra le macchine virtuali e i container sotto diversi punti di vista come prestazioni, sicurezza di isolamento, rete, la tabella é la seguente 2.1.

Paramentro	Macchina Virtuale	Container
Guest OS	Ogni VM viene eseguita su hardware virtuale e il kernel viene caricato nella propria regione di memoria	Tutti i guest condividono lo stesso sistema operativo e kernel. L' immagine del kernel viene caricata nella memoria fisica
Sicurezza	Dipende dall'implementazione di Hypervisor	É possibile sfruttare il controllo degli accessi obbligatorio
Performance	Le macchine virtuali soffrono di un piccolo sovraccarico poiché le istruzioni della macchina vengono tradotte dal sistema operativo guest al sistema operativo host	I container forniscono prestazioni quasi native rispetto al sistema operativo host sottostante.
Isolation	Non é possibile condividere librerie , file ecc. Tra ospiti e tra ospiti ospiti.	Le sottodirectory possono essere montate in modo trasparente e possono essere condivise.
Start-up time	Le VM impiegano alcuni minuti per avviarsi	I container possono essere avviati in pochi secondi rispetto alle VM.
Storage	Le VM richiedono molto piú spazio di archiviazione poiché l'intero kernel del sistema operativo e i suoi programmi associati devono essere installati ed eseguiti	I container richiedono una quantitá inferiore di spazio di archiviazione poiché il sistema operativo di base é condiviso

Tabella 2.1: Confronto tra container e macchine virtuali

2.6 Vantaggi dell'uso dei software container

Vengono di seguito analizzati alcuni dei principali vantaggi dei container.

- **Leggerezza**: Un container puó essere eseguito su una vasta gamma di dispositivi, anche se non dotati di grosse risorse (come ad esempio RaspberryPi), in quanto nel container é contenuto solo lo stretto necessario per il funzionamento dell'applicazione, cosa che non avviene con le macchine virtuali, le quali necessitano di un sistema operativo ospite, e quindi di risorse hardware da virtualizzare, inoltre il numero di container che si possono eseguire in un server, é piú alto del numero di macchine virtuali eseguibili nello stesso server;

- **Isolamento**: Ogni servizio viene eseguito in un proprio ambiente, in maniera indipendente da altri container sfruttando determinate funzionalitá del kernel. Si puó limitare per fare in modo che il container possa interagire con altre risorse, come la rete, o i volumi;

- **Rapido avvio**: L'assenza di una macchina virtuale, implica giá l'assenza del tempo di avvio dell'OS ospite della macchina stessa, mentre l'avvio di un container puó essere effettuato mediante una riga di comando ed é praticamente istantaneo;

- **Portabilitá**: Proprio come i container dei camion, i container godono di un approccio standardizzato, ció significa che possono essere distribuiti su qualunque risorsa di calcolo indipendentemente da configurazioni, sistema operativo o hardware. Grazie a questa caratterista, una volta completato il deployment, il rilascio e la distribuzione dell'applicativo sono molto veloci;

- **Condivisione delle risorse**: Ogni container vede le risorse come se fossero dedicate, anche se in realtá sono condivise con gli altri processi in esecuzione. In questa maniera le risorse verranno utilizzate solo quando vi é un effettivo carico di lavoro da svolgere, rilasciandole quando non sono utilizzate, permettendo perció ad altri container di potervi accedere e utilizzarle; ció tuttavia non viola l'isolamento del contesto di esecuzione del container;

- **Granularitá e flessibilitá**: i container permettono di organizzare le risorse computazionali in micro-servizi migliorando le prestazioni del sistema, che potrá adattarsi in modo estremamente flessibile alle esigenze dell'azienda;

- **Scalabilitá**: Un sistema implementato tramite container, é scalabile. La scalabilitá é data dal fatto che i servizi che sono inclusi nei container, quindi spesso per aggiungere potenza, basta aggiungere un container, o un insieme di container che la implementa; oltre a ció é possibile allocare o distruggere container al fine di gestire al meglio il carico di lavoro a cui far fronte in momenti diversi (scalabilitá orizzontale) .Inoltre ad un aumento delle prestazioni hardware, puó spesso corrispondere un proporzionale incremento della performance del sistema, ad esempio in termini di velocitá (scalabilitá verticale);

Capitolo 3

Docker

Docker é un software open-source con il quale pacchettizzare e deployare applicazioni all'interno di container. Ció garantisce alle applicazioni di funzionare esattamente allo stesso modo in qualunque ambiente di runtime poiché tutto ció di cui l'applicazione necessita é racchiuso nel container. Docker é in grado di orchestrare infrastrutture basate su container finalizzate alla realizzazione di applicazioni distribuite. Il progetto Docker nacque nell'azienda DotCloud, e fu pubblicato come progetto open source a marzo 2013. Il successo del progetto fu tale che nell'ottobre dello stesso anno la societá fu rinominata con il nome dell'omonima tecnologia. Sin dagli albori Docker ha riscosso un enorme successo posizionandosi tra le principali tecnologie di virtualizzazione.

3.1 Docker Engine

Docker utilizza un'architettura client-server. Il Docker client comunica con il Docker deamon, chiamato Dockerd il quale fa il grosso del lavoro mandare in esecuzione o arrestare i container. Il client e il demone Docker possono essere eseguiti nella stessa macchina oppure un Docker client puó cominicare con un Docker deamon remoto. Il Docker deamon ascolta le richieste e gestisce gli oggetti Docker, quali le immagini, i container, le porte, i volumi e la rete. Il Docker client é il modo standard per interfacciarsi con il sistema Docker, dato che interpreta i comandi da inviare al Docker deamon quale li processerá.

Gli attori sono tre:

- **Docker API REST**: protocollo di comunicazione tra client e Docker Daemon.
- **Docker Client**: implementa la Command Line Interface (CLI) che traduce gli input da console in richieste REST compatibili col Docker Demon. Inoltre Docker Client puó comunicare con piú Daemon.
- **Docker Daemon**: gestisce gli oggetti Docker come immagini, container, reti e volumi. Un Daemon puó anche comunicare con altri Daemon per soddisfare le richieste.

Questa scelta implementativa permette una buona suddivisione delle responsabilitá ma di fatto abbatte le barriere della localitá, facendo sí che lo stesso Engine possa essere distribuito su host differenti.

3.2 Docker Registries

Una delle caratteristiche principali di Docker é la capacitá di trovare, scaricare e avviare rapidamente immagini create da altri sviluppatori. Il luogo standard in cui vengono archiviate le immagini é chiamato Docker Hub o Registry `https://hub.docker.com/`. Oltre a varie immagini di base, utilizzabili per creare ulteriori immagini, il registry Docker pubblico include immagini di software pronto per l'esecuzione, inclusi database, sistemi di gestione dei contenuti, ambienti di sviluppo, server Web e cosí via. L'offrire immagini gratuitamente é uno dei punti di forza di Docker, consultabile da tutti in maniera gratuita, ha dato una grossa mano nella diffusione del progetto [12]. La presenza di queste immagini giá pronte é essenziale per la comunitá Docker, in quanto permette di avere delle basi di partenza di un servizio giá testate e in costante manutenzione e aggiornamento, liberando cosí gli sviluppatori delle fasi preliminari dello sviluppo dei container consentendogli di concentrarsi solo sulla customizzazione dell'applicativo. Nonostante il Docker Registries sia gratuito e accessibile da tutti é possibile creare un registries locale tramite un immagine liberamente scaricabile. Nel Registries locale é possibile svolgere le stesse operazioni che si possono svolgere nel Registries pubblico.

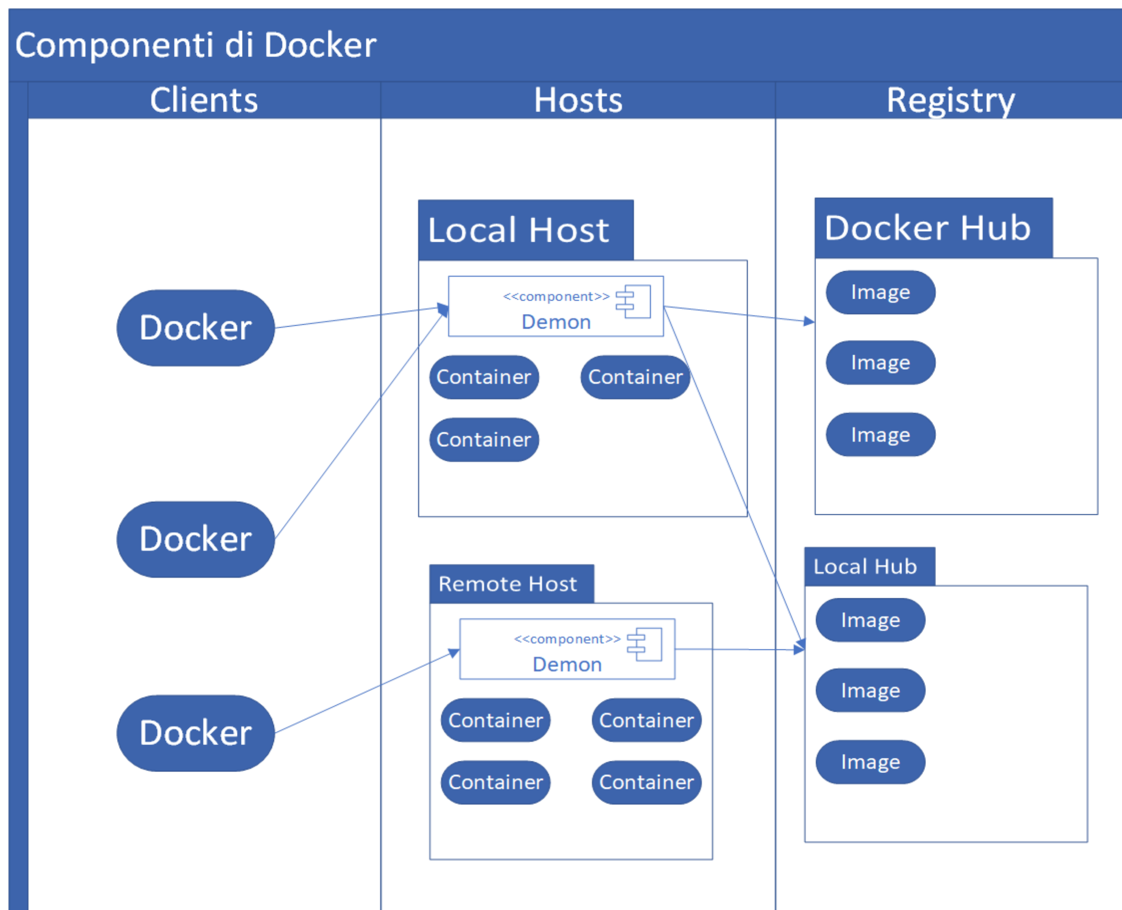

Figura 3.1: Componenti principali di Docker

3.3 Docker Image e Docker Container

Un'immagine é un file di sola lettura contenente istruzioni che hanno lo scopo di creare container. Solitamente un'immagine si basa su una o piú immagini giá esistenti, e a partire da questa si susseguono le istruzioni che servono a personalizzare l'immagine. Tutto questo viene gestito tramite il Dockerfile, un file il quale passo a passo dichiara tutte le istruzioni da eseguire per creare l'immagine ed eseguirla. Ogni istruzione del Dockerfile si traduce in uno strato dell'immagine. Se un'istruzione viene modificata, alla ricompilazione dell'immagine le modifiche saranno apportate solo allo strato corrispondente all'istruzione modificata. L'interpretazione di un Dockerfile produce un'immagine, la quale puó essere trasformata a sua volta in un Container. Un container é un'istan-

za eseguibile dell'immagine. Tramite le API di Docker, é possibile gestire le operazioni sui container come l'esecuzione, fermarli, metterli in pausa, riavviarli e cancellarli. Un container quindi é definito dalla sua immagine esattamente dalle configurazioni che gli vengono fornite alla sua creazione e al suo avvio. Da ció ne consegue una grande caratteristica di Docker, il fatto che da un immagine possano nascere piú container, e che dalla stessa immagine possono nascere piú immagini.

3.4 DockerFile

Il DockerFile é una modalitá che permette di costruire una nuova immagine descrivendone il suo contenuto in un file chiamato per l'appunto Dockerfile, dove al suo interno sono presenti una serie di istruzioni e man mano che vengono letti producono delle azioni che servono per costruire l'immagine. Le principali istruzioni sono le seguenti:

- **FROM** la prima istruzione presente nel file, é obbligatoria, e va a definire l'immagine di base su cui si va a costruire la propria, per esempio ubuntu per l'appunto;
- **COPY** permette di copiare dei file locali nell'immagine, specificando la posizione desiderata;
- **RUN** esegue nel Docker Engine un comando di shell. Questo comando é molto utile per l'installazione di software e package aggiuntivi necessari per l'esecuzione dell'applicazione desiderata;
- **ENV** offre la possibilitá di definire delle variabili d'ambiente utilizzabili all'interno del container;
- **VOLUME** crea una relazione tra una directory presente all'interno dell'immagine e una directory presente nel filesystem dove il container verrá eseguito;
- **CMD** permette di eseguire un comando di shell a runtime nel container. Lo scopo principale di questa istruzione é quella di fornire un comando che il container dovrá eseguire, come per esempio avviare un pacchetto java con il comando:

  ```
  java -jar app.jar
  ```

- **EXPOSE** dichiara le porte che devono essere raggiungibili dall'esterno, ad esempio se il container ospita un server web é opportuno lasciare raggiungibili la porta 80.

Un esempio di DockerFile offerto dalla documentazione ufficiale di Docker é il seguente:

```
FROM ubuntu:18.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

Sono presenti quattro comandi, ognuno dei quali crea un livello. L'istruzione FROM inizia creando un livello dall'immagine ubuntu: 18.04. Il comando COPY aggiunge alcuni file dalla directory corrente del client Docker. Il comando RUN crea l'applicazione utilizzando il comando make. Infine, l'ultimo livello specifica quale comando eseguire all'interno del container. Un alternativa all'utilizzo del Dockerfile é la creazione di un immagine in maniera interattiva. Quando si costruisce un container in modo interattivo si é in grado di installare librerie e configurare l'ambiente da una shell proprio come in un tipico ambiente Linux [3].

3.5 Network

Le principali modalitá di network di Docker sono tre.

- **Bridge** Una rete Bridge sfrutta l'interfaccia virtuale di tipo Linux Bridge che consente la comunicazione tra container collegati allo stesso device. Bridge é la rete di default alla quale Docker assegna tutti i container che non ne specificano un'altra.

- **Host** Questa modalitá fa si che lo stack di rete del container non sia isolato dal sistema. Ad esempio, se si esegue un container sulla porta 80 con la rete host, l'applicazione sará disponibile sulla porta 80 all'indirizzo IP dell'host.

- **None** I containers appartenenti a questa rete non avranno nessuna interfaccia di rete se non quella di loopback.

3.6 Gestione della persistenza

La persistenza di Docker ha lo scopo di esportare i dati dal conainer in un altro punto, questo permette la condivisione di dati tra container diversi e garantisce l'esistenza dei dati anche dopo l'eliminazione del container. Docker supporta due principali tecniche per la gestione della persistenza.

Bind Mounts

Con l'utilizzo dei Bind Mounts una directory sulla macchina host viene montata in un container. La directory o il file é referenziato dal suo percorso sulla macchina host. Non é necessario che il file o la directory esista giá sull'host Docker, viene creato se non esiste. I mount bind sono molto performanti, ma si basano sul filesystem della macchina host che ha una specifica struttura di directory la quale potrebbe non essere la stessa del container.

Volumi

I volumi sono il meccanismo preferito per la persistenza dei dati generati e utilizzati dai container Docker. Mentre i montaggi bind dipendono dalla struttura della directory della macchina host, i volumi sono completamente gestiti da Docker. I volumi hanno diversi vantaggi rispetto ai montaggi bind:

- É possibile gestire i volumi utilizzando i comandi della CLI Docker o l'API Docker;
- I volumi funzionano su container Linux e Windows;
- I volumi possono essere condivisi in modo piú sicuro tra piú container, sullo stesso host;
- I driver di volume consentono di archiviare volumi su host remoti o provider cloud, per crittografare il contenuto dei volumi o per aggiungere altre funzionalitá;
- I nuovi volumi possono avere il contenuto precompilato da un contenitore;

- I volumi sono spesso una scelta migliore rispetto alla persistenza dei dati nel livello scrivibile di un contenitore, poiché un volume non aumenta le dimensioni dei container che lo utilizzano e il contenuto del volume esiste al di fuori del ciclo di vita di un determinato contenitore.

3.7 Control Group

Docker si basa sulla funzionalitá di linux che sono i control groups (cgroups). Cgourps limita l'utilizzo da parte del container alle risorse hardware. I Cgroups consentendo a Docker engine di condividere/concedere le risorse hardware disponibili ai container, e di limitarne e vincolarne l'utilizzo, come ad esempio settare il numero di processori che un container puó utilizzare.

3.8 Potenzialitá di Docker

Le potenzialitá di Docker emergono della gestione di due o piú container eseguiti contemporaneamente e fatti comunicare l'uno con l'altro tramite la rete, questa modalitá di chiama Swarm. La modalitá Swarm offre la possibilitá di costruire una rete di nodi con annessa la loro gestione. Un nodo é un'istanza di Docker che insieme ad altri nodi crea un cluster. Le modalitá di nodi sono due il worker e il manager, il worker riceve la richiesta di un task e la esegue, invece il nodo manager riceve i compiti dalle API invocate da un agente, ed ha il compito di rilascio di un servizio e divide esso in una serie di compiti da assegnare ai worker, gestisce cluster e suoi nodi. Il concetto di nodo in Docker che comunica con altri nodi introduce il concetto di gestione di un claster la quale puó essere gestita in due modi:

- **Orchestrazione**: intesa come un modello per la gestione del lavoro degli host sulla rete. In particolare sará un host (direttore d'orchestra) a gestire il flusso dei dati tra gli altri host sulla rete;

- **Coreografia**: non é presente un nodo particolare che si occupa di organizzare il lavoro degli altri, ogni host sulla rete é autonomo nei propri compiti e conosce con chi comunicare.

La comunicazione tra container ha come potenzialitá quella di permettere di comporre container per realizzare un unico servizio o applicazione. Per composizione si intende la possibilitá di definire ed eseguire applicazioni multi-container. Tali applicazioni possono essere viste come uno stack, costituito dall'insieme dei servizi, conteiner, che la compongono. Le funzionalitá di base di Docker permettono di gestire la composizione solo nei casi piú semplici, per composizioni piú complesse ci sono vari tool per Docker che ne permettono il realizzo e la gestione. L'orchestrazione in Docker é possibile tramite delle estensioni native che ne permettono il realizzo, ma oltre a quelli standard ci sono molti tool per l'orchestrazione di container Docker, tra i piú diffusi Amazon EC2 Container Service, Google Container Engine, Kubernetes, Apache mesos. I principali tool di orchestrazione verranno trattati nel prossimo capitolo.

Capitolo 4

Orchestrazione di container

L'orchestrazione di container é un processo che permette di distribuire una molteplicitá di container per poter implementare un'applicazione. Al giorno d'oggi le applicazioni non sono piú monolitiche, ma sono invece composte da dozzine o centinaia di componenti containerizzati e accoppiati che devono lavorare insieme per consentire a una determinata applicazione di funzionare come progettata. L'orchestrazione dei container fa riferimento al processo di organizzare il lavoro dei singoli componenti e dei livelli dell'applicazione. Tool di orchestrazione di container, quali Kubernetes e Docker Swarm, consentono agli utenti di guidare il deployment, gli aggiornamenti e il monitoraggio di container. Il vantaggio dell'uso di orchestratori é una gestione automatica capace di controllare la grande quantitá di richieste e di gestire le potenzialitá che i container offrono, come per esempio la scalabilitá [14]. I vantaggi di usare un tool di orchestrazione sono:

- Gestione efficiente delle risorse;
- Scalabilitá continua dei servizi;
- Alta disponibilitá;
- Basso sovraccarico operativo su larga scala;
- Un modello dichiarativo per la maggior parte degli strumenti di orchestrazione, riducendo l'attrito per una gestione piú autonoma;
- Esperienza operativa coerente tra provider di cloud pubblici e locali.

Le caratteristiche base di una tipica piattaforma di orchestrazione di conteiner includono:

- Pianificazione delle attivitá;
- Gestione delle risorse;
- Rilevamento dei servizi;
- Controlli sanitari;
- Scalabilitá automatica;
- Aggiornamenti e upgrade.

L'orchestrazione dei contenitori incoraggia l'uso del modello di architettura dei microservizi, in cui un'applicazione é composta da servizi piú piccoli, atomici e indipendenti, ognuno progettato per una singola attivitá. Ogni microservizio é confezionato come container e piú microservizi appartenenti logicamente alla stessa applicazione vengono orchestrati in fase di esecuzione [14].

4.1 Docker Swarm

Docker Swarm é un tool che permette il clustering e orchestrazione di container, il tool é integrato in Docker, nasce dagli stessi sviluppatori di Docker e raggruppa un qualsiasi numero di container Docker in un unico cluster, chiamato Swarm (sciame), permettendone la gestione centralizzata e l'orchestrazione. Docker Swarm si occupa di ripristinare anche i servizi non piú disponibili a causa di errori o guasti improvvisi. L'utilizzo di un tool di orchestrazzione porta come principale vantaggio il mantimento di uno stato ottimale di un servizio [6]. Quando viene creato un servizio, ne viene definito lo stato ottimale, quality of service (QoS), il quale comprende il numero di repliche, risorse di rete e di archiviazione disponibili, porte che il servizio espone al mondo esterno e altro. Docker Swarm lavora per mantenere lo stato desiderato. Ad esempio, se un nodo diventa non disponibile, Docker Swarm pianifica i task di quel nodo su altri nodi. Un task é un conteiner in esecuzione, che é parte di un cluster gestito da Doker Swarm, al

contrario di un container autonomo [7]. Uno dei principali vantaggi nell'utilizzo di Docker Swarm rispetto ai container autonomi é la possibilitá di modificare la configurazione di un servizio, comprese le reti e i volumi a cui é connesso, senza la necessitá di riavviare manualmente il servizio. Un fattore chiave che spinge l'utilizzo di tool come Docker Swarm é la ripartizione del carico dato che dispone di funzioni integrate di load balancing, ovvero bilanciamento del carico di lavoro. Se per esempio viene eseguito un server web NGINX con 4 istanze, Docker distribuisce in modo intelligente le richieste in arrivo alle istanze dei server disponibili, Docker Swarm utilizza il bilanciamento del carico in ingresso, ingress load balancing, per esporre i servizi che si desidera rendere disponibili esternamente.

4.1.1 Architettura Docker Swarm

L'architettura di Docker Swarm, visibile in Figura 4.1, presenta una tipologia master-slave. Le funzionalitá di gestione e orchestrazione dei cluster incorporate in Docker vengono create utilizzando Swarmkit . Swarmkit é un progetto separato che implementa il livello di orchestrazione di Docker e viene utilizzato direttamente all'interno di Docker, ne verrá discusso nel prossimo sotto capitolo. Uno Swarm é costituito da piú host Docker che vengono eseguiti in modalitá Swarm e agiscono come manager o worker. Un determinato host Docker puó essere un master, un worker o svolgere entrambi i ruoli. Un cluster puó avere piú worker, ma é presente sempre e solo un Master, scelto in base ad un algoritmo di elezione, algoritmo Raft, anche di questo ne verrá discusso nei prossimi sotto capitoli. I nodi worker ricevono il lavoro da eseguire direttamente dal manager, il quale, oltre a decidere il worker da assegnare la richiesta mantiene stabile lo stato del cluster. Un nodo di Docker Swarm puó quindi fungere da Manager o Worker.

Worker

Un Worker, noto principalmente come slave o semplicemente come nodo, é un'istanza Docker che partecipa al cluster. É possibile eseguire uno o piú nodi su un singolo calcolatore fisico o server cloud, ma le distribuzioni di Swarm degli ambienti di produzione in genere includono nodi Docker distribuiti su piú macchine fisiche e cloud.

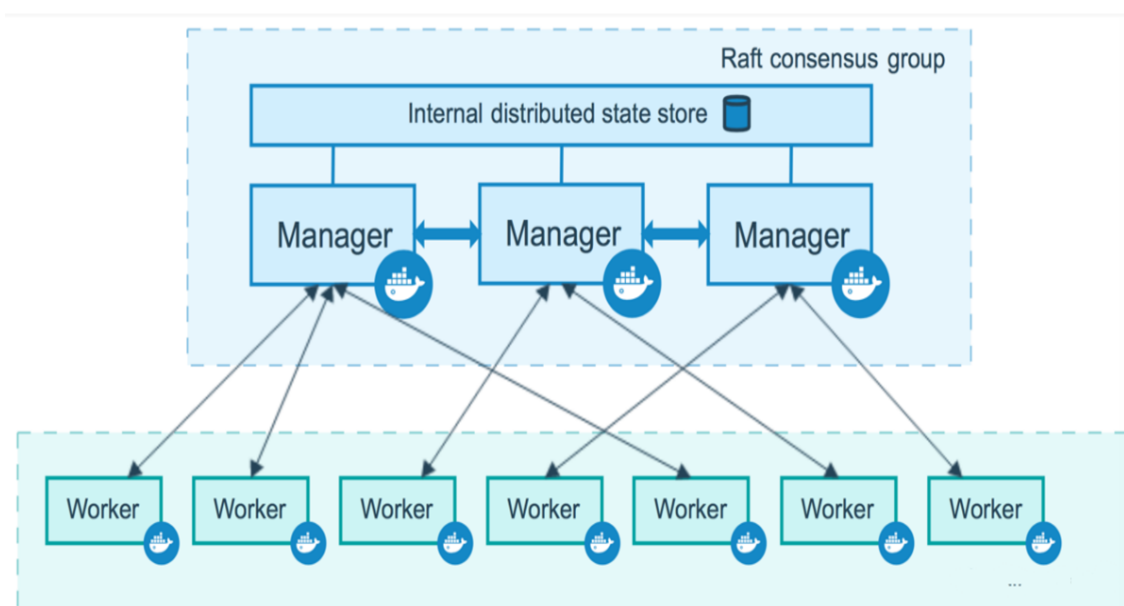

Figura 4.1: Architettura di Docker Swarm

Manager

I nodi manager, o master, una volta eletti come tali mediante l'algoritmo di consenso Raft, si occupano di gestire lo stato del cluster. Uno dei principali compiti del master é il load balancing dei container, per ogni applicazione, é possibile dichiarare il numero di attivtá che possono essere eseguite. Per ripartire il carico il nodo Manager monitora costantemente lo stato del cluster e riconcilia eventuali differenze tra lo stato effettivo e lo stato desiderato. Ad esempio, se viene settato un servizio per eseguire 10 repliche di un container e una macchina worker che ospita due di queste repliche si arresta in modo anomalo, il manager sposta le due repliche del nodo che si é arrestato su di un altro nodo.

4.1.2 Servizi

Il deploy di un'immagine, quando Docker, é in modalitá Swarm, produce la creazione di un servizio o service. Per service si intende un gruppo di container che si basano sulla stessa immagine. Quando viene creato un servizio, deve essere specificato quale immagine del container utilizzare e quali comandi eseguire all'interno dei container in esecuzione.

Ogni servizio é composto da un set di task singoli, che vengono elaborati ciascuno su un conteiner. Quando viene deployato il servizio, il manager dello swarm accetta la definizione del servizio cosí come é stato settato per il servizio e di conseguenza pianifica lo stesso sui nodi dello Swarm come una o piú repliche. I task sui vari conteiner vengono eseguite indipendentemente l'una dall'altra sui nodi. Ad esempio, il bilanciamento del carico tra tre istanze di un listener HTTP, come mostrato nell'immagine 4.2. Ciascuna delle tre istanze é un istanza dello stesso servizio ma replicata. Docker Swarm supporta due modalitá per definire un servizio:

Servizi replicati

Un servizio replicato é un task che viene eseguito in un numero di repliche definite dall'utente. Ogni servizio replicato é un'istanza del container. I servizi replicati vengono scalati all'aumentare della computazione ed hanno come risultato finale la creazione di piú repliche.

Servizi globali

Eseguendo un servizio in modalitá globale, ogni nodo disponibile nel cluster inizia un task per il relativo servizio. Se viene aggiunto un nuovo nodo al cluster, il manager gli assegna immediatamente un'attivitá del service globale.

4.1.3 Pianificazione dei task

Un task é un istanza di un container, l'esecuzione di un servizio dipende dall'esecuzione di un certo numero di task relativi a quel servizio. Quando si dichiara uno stato del servizio desiderato creando o aggiornando un servizio, il manager analizza lo stato desiderato pianificando le attivitá che dovranno essere svolte per manenere lo stato globale come da impostazione in input. Ad esempio, si definisce un servizio che deve mantenere in esecuzione tre istanze di un listener HTTP in ogni momento, il manager quindi risponde creando tre task. Se un task listener HTTP successivamente non supera il controllo di integritá o si arresta in modo anomalo, il manager crea un nuovo task di replica che genera un nuovo container. Un task transita attraverso una serie di stati, i

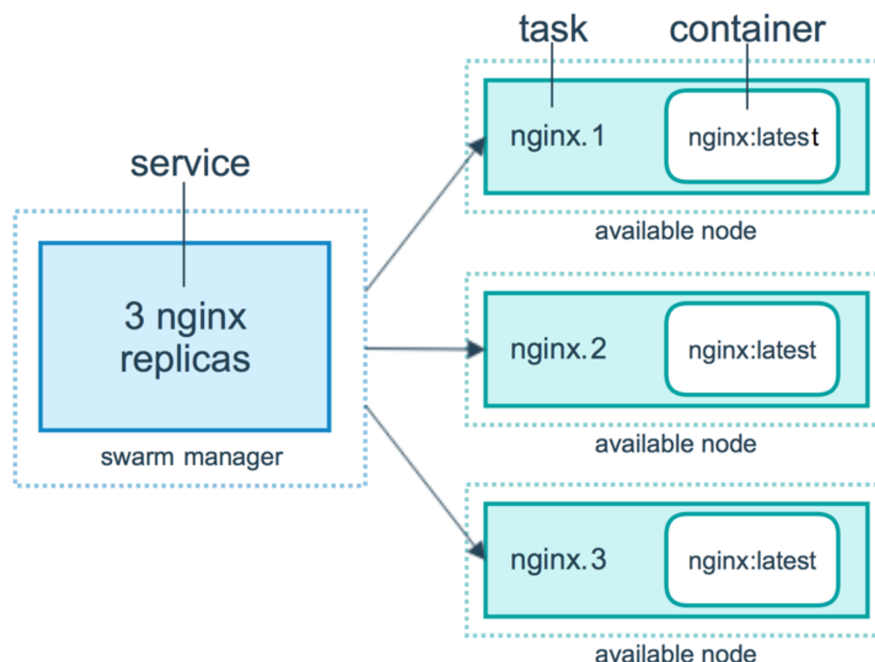

Figura 4.2: Servizio Scalato su tre nodi

quali sono assegnati, preparati ed in esecuzione. Se un task fallisce, il manager rimuove il task e crea un nuovo container per sostituirlo in base allo stato desiderato specificato dal servizio. L'immagine 4.3 mostra come la modalitá Swarm accetta le richieste di creazione del servizio e pianifica le task per i nodi worker.

Un servizio puó anche essere configurato in modo che nessun nodo attualmente possa eseguire le sue attivitá. In questo caso, il servizio rimane in stato *pending* [5]. Questo é utile quando é necessario riservare una quantitá specifica di memoria per un servizio e nessun nodo ha la quantitá di memoria richiesta; il servizio rimane in uno stato in pending finché non é disponibile un nodo che puó eseguire le sue attivitá. Se si specifica un valore molto grande, ad esempio 500 GB, l'attivitá rimane in sospeso per sempre, a meno che non si disponga di un nodo in grado di soddisfarla.

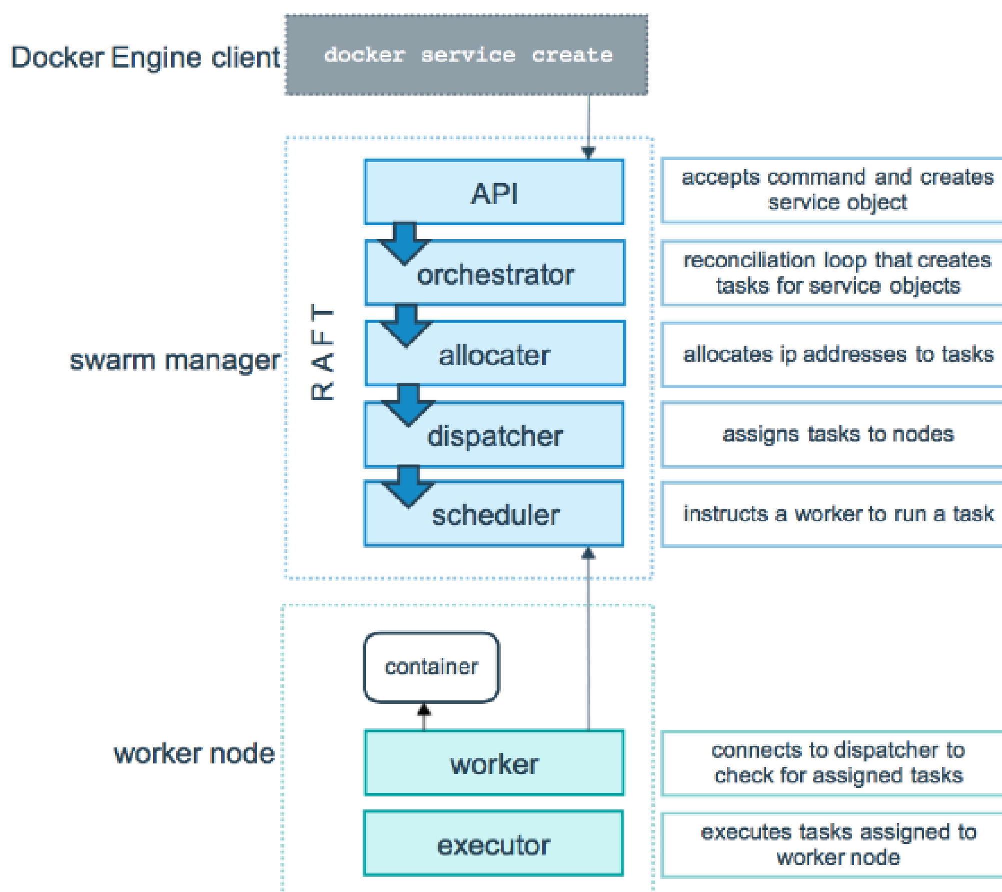

Figura 4.3: Servizio Scalato su tre nodi

4.1.4 Swarmkit

Docker Swarm per orchestrare i propri container utilizza SwarmKit che é un tool di orchestrazione di sistemi distribuiti open source. I principali compiti di Swarmkit sono la scoperta dei nodi, il consenso basato sull'algoritmo Raft e la pianificazione delle attivitá. Di seguito sono analizzate le principali attivitá di SwarmKit.

L'orchestrazione dei nodi

SwarmKit confronta costantemente lo stato desiderato con lo stato corrente del cluster e riconcilia i due se necessario. Ad esempio, se un nodo si guasta, SwarmKit ripianifica le sue attivitá su un nodo diverso, gestendolo in base al fatto che si basi su di un servizio

replicato o un servizio globale. In qualsiasi momento é possibile modificare il valore di uno o piú campi per un servizio. Dopo aver effettuato l'aggiornamento, SwarmKit riconcilia lo stato desiderato assicurandosi che tutte le attivitá utilizzino le impostazioni desiderate. Per impostazione predefinita, esegue un aggiornamento lockstep, ovvero aggiorna tutte le attivitá contemporaneamente. Per ultimo, ma non meno importante, Swarmkit monitora le attivitá e reagisce agli errori in base al criterio specificato. L'operatore puó definire condizioni di riavvio, ritardi e limiti come il numero massimo di tentativi in un range temporale. SwarmKit puó decidere di riavviare un'attivitá su una macchina diversa.Ció significa che i nodi difettosi verranno gradualmente svuotati dei loro compiti.

Scheduling dei compiti

Lo Scheduling dei compiti avviene mediante l'analisi delle risorse. SwarmKit é a conoscenza delle risorse disponibili e posizionerá le attivitá di conseguenza, utilizzando una strategia di diffusione che tenterá di pianificare le attivitá sui nodi meno caricati, a condizione che soddisfino i vincoli e le risorse richieste.

La Gestion del Cluster

Cosí come la "scelta" della tipologia dei ruoli dei nodi se worker o slave, che possono essere modificati dinamicamente tramite chiamate API o CLI, un altro compito di Swarmkit é la gestione dei nodi. Il master puó modificare la disponibilitá desiderata di un nodo, per esempio lo puó impostare su Paused il quale ne impedirá la pianificazione di ulteriori attivitá.

L'analisi di sicurezza

Tutti i nodi comunicano tra loro utilizzando Mutual TLS. Il manager dello Swarm agiscono come autoritá di certificazione, emettendo certificati a nuovi nodi. Quando dei nodi vogliono unirsi allo Swarm essi richiedono un token crittografico, che definisce il ruolo di quel nodo. I certificati TLS vengono ruotati e ricaricati in modo trasparente su ogni nodo, consentendo all'utente di impostare la frequenza con cui deve avvenire la rotazione.

4.1.5 Algoritmo Raft

Raft é un algoritmo di consenso il quale consenso é un problema fondamentale nei sistemi distribuiti [16]. Quando Docker viene eseguito in modalitá Swarm, i nodi manager implementano l' algoritmo di consenso Raft per gestire lo stato del cluster globale. Il motivo per cui la modalitá swarm Docker utilizza un algoritmo di consenso é assicurarsi che tutti i nodi del gestore che sono responsabili della gestione e della pianificazione delle attivitá nel cluster, memorizzino lo stesso stato coerente. Avere lo stesso stato coerente in tutto il cluster significa che in caso di errore, qualsiasi nodo Manager puó raccogliere le attivitá e ripristinare i servizi a uno stato stabile. Ad esempio, se il Manager responsabile della pianificazione delle attivitá nel cluster muore inaspettatamente, qualsiasi altro Manager puó riprendere l'attivitá di pianificazione e riequilibrare le attivitá in modo che corrispondano allo stato desiderato. I sistemi che utilizzano algoritmi di consenso per replicare i registri in sistemi distribuiti richiedono un'attenzione speciale dato che devono garantire che lo stato del cluster rimanga coerente in presenza di errori, richiedendo alla maggioranza dei nodi di concordare i valori. Il consenso implica che piú calcolatori concordino su di un valore, in questo caso chi ricopre il ruolo di Leader. Il consenso sorge tipicamente nel contesto delle macchine replicate, un approccio generale alla creazione di sistemi a tolleranza di errore [15]. Nell'algoritmo Raft, ogni nodo puó transitare in tre stadi diversi, Leader, Follower e Candidato. Normalmente esiste un Leader e tutti gli altri nodi sono suoi Followers. Raft suddivide il tempo in termini di lunghezza arbitraria i quali vengono indicati da un intero crescente. Ogni temine inizia con un'elezione nella quale i Candidati cercano di diventare il Leader. Se nessun candidato ha la maggioranza assoluta dei voti, il termine passa vacante, si inizia un nuovo termine e si procede ad una nuova elezione.

Elezione del Leader

Quando l'algoritmo inizia, tutti i nodi si trovano nello stato di Follower. Un nodo rimane nello stato di Follower finché non riceve comunicazioni da un Leader o dai Candidati. Un Leader invia delle comunicazioni periodiche a tutti i nodi per mantenere la propria leadership. Se un Follower non riceve comunicazioni entro un tempo casuale chiamato Election Timeout, il nodo assume che il Leader sia caduto, diventa un Candidato

e inizia una nuova elezione. Per iniziare una nuova elezione, il candidato incrementa il proprio valore corrente, che inizialmente sará uno, vota per sé stesso e invia in parallelo a tutti i nodi la richiesta di voto. Un Candidato rimane tale finché:

1. il candidato vince l'elezione;
2. un altro Candidato vince l'elezione;
3. l'elezione si conclude senza vincitore.

Nel primo caso, un Candidato vince l'elezione se riceve la maggioranza assoluta dei voti dai nodi nella stessa elezione. Ogni nodo puó votare per un solo candidato. Il voto avviene con politica first-come-first-served, che implica che la prima candidatura ricevuta eleggibile viene votata. Quando un Candidato vince l'elezione, manda a tutti i nodi la comunicazione di vittoria, prevenendo l'avvio di una nuova tornata elettorale. Nel secondo caso, un candidato ha richiesto una votazione, ma riceve una comunicazione che un altro nodo é un leader, il nodo corrente, riconosce il nuovo Leader e ne diventa Follower. Nell'ultimo caso, a causa di una dispersione di voti non é stato possibile eleggere un candidato. In questo caso i candidati ripetono la procedura di richiesta voti. Per evitare il possibile ripetersi infinito di termini vacanti, Raft usa degli Election Timeout casuali entro un intervallo fisso, questo approccio incrementa le probabilitá che i timeout avvengano scaglionati, permettendo ad un solo nodo di diventare candidato, richiedere le elezioni e diventare Leader prima che un altro nodo vada in timeout e richieda le elezioni a sua volta.

4.2 Kubernetes

Kubernetes, noto anche con l'acronimo K8S, é un altro sistema per l'orchestrazione di applicazioni containerizzate il quale fornisce un meccanismo per il deployment, manutenzione e lo scaling di applicazioni. É stato sviluppato dal team di Google, per poi passare nel 2015 sotto il controllo della Cloud Native Computing Foundation (CNCF), che attualmente lo supporta [9]. I principali compiti di Kubernetes sono: deployment, scaling e management di applicazioni. In dettaglio:

- **Deployment**: gestisce la distribuzione di applicazioni assegnando ai nodi del cluster ciascuna istanza dell'applicazione. Il deployment in Kubernetes puó essere eseguito in una varietá di ambienti con pattern differenti;

- **Scaling**: permette di ridimensionare l'applicazione a seconda delle esigenze dell'utente, andando a modificare le dimensioni del cluster e il numero di repliche dei Pod. Un Pod é il piú piccolo oggetto deployabile nel modello a oggetti di Kubernetes e puó incapsulare un singolo container o piú container che necessitano di lavorare insieme;

- **Management**: fornisce un'interfaccia per la gestione dei cluster e delle applicazioni containerizzate.

4.2.1 Architettura di Kubernetes

Kubernetes é organizzato in livelli di moduli in base alle funzionalitá, la panoramica generale della partizione dei moduli é mostrata nella Figura 2.1.

Di seguito viene spiegato ciascun livello [11].

- **Nucleo - API ed Esecuzione** Il nucleo contiene il set minimo di funzionalitá necessarie per costruire i livelli superiori del sistema ed é composto da API REST e i moduli di esecuzione responsabili dell'esecuzione dell'applicazione all'interno dei container. Il modulo di esecuzione piú importante in Kubernetes é *Kubelet*.

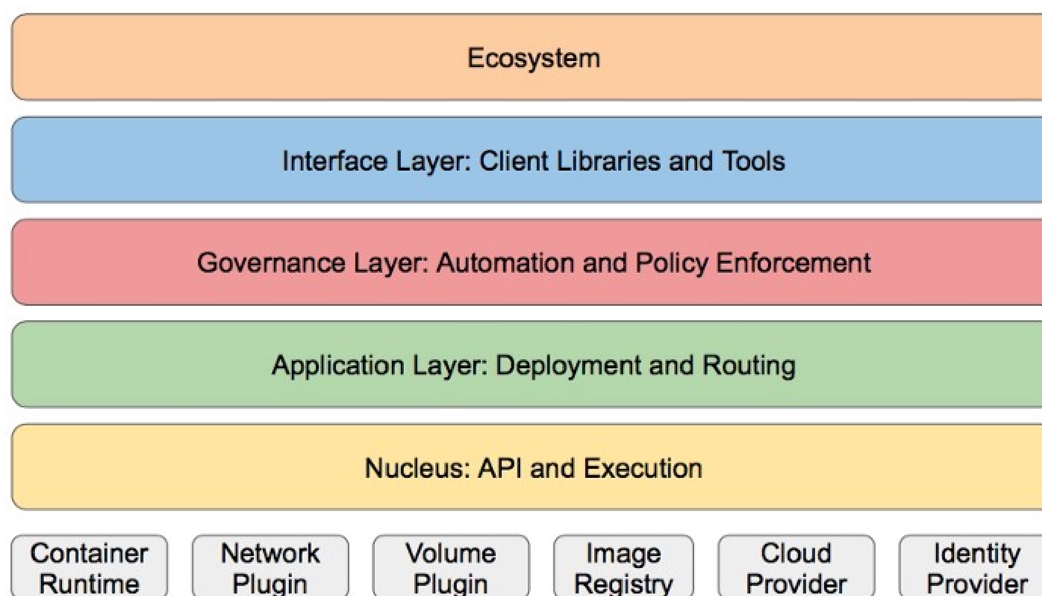

Figura 4.4: Organizzazione a moduli di Kubernetes

- **Livello Applicazione - Deployment e Routing** Il livello dell'applicazione fornisce auto-healing, scaling, gestione del ciclo di vita delle applicazioni, individuazione dei servizi, bilanciamento del carico e routing. Inoltre, il modulo di deployment fornisce metodi container-centric e controller del ciclo di vita per supportare l'orchestrazione, mentre il modulo di routing fornisce il servizio di scheduling.

- **Livello Governance - Automazione e Policy Enforcement** Questo livello contiene il modulo di automazione che fornisce lo scaling automatico del cluster e il provisioning automatico dei nodi. Il modulo di policy enforcement invece fornisce i mezzi per configurare e scoprire le politiche predefinite per il cluster.

- **Livello Interface - Librerie e Tools** Questo livello contiene librerie, tools, sistemi e interfacce utente comunemente usate nel progetto Kubernetes. Uno degli strumenti piú importanti in questo livello é kubectl.

- **Ecosistema** Questo livello fornisce le funzionalitá necessarie per distribuire e gestire applicazioni containerizzate. L'esempio piú popolare di questo livello di supporto

é Docker, che esegue il container reale nei nodi. I moduli utilizzati per comunicare con il provider di cloud appartengono a questo livello.

4.2.2 Pod

Un Pod é una collezione di uno o piú container che funge da unitá principale di Kubernetes per la gestione del carico di lavoro. Nella fase di runtime, i Pod possono essere ridimensionati creando set di repliche, che garantiscono che la distribuzione ne esegua sempre il numero desiderato. Come giá anticipato ogni Pod é una raccolta di piú container e per comunicare tra loro utilizzano le remote procedure calls (RPC), condividendo lo stack di archiviazione e di rete. Negli scenari in cui i container devono essere accoppiati e co-localizzati, ad esempio un container del server Web e un container della cache, possono essere facilmente raccolti in un singolo Pod. Un Pod puó essere scalato manualmente o tramite una feature chiamata Horizontal Pod Autoscaling (HPA). Attraverso questo metodo il numero dei container al'interno di un Pod viene aumentato proporzionalmente. I Pods consentono inoltre una separazione funzionale tra development e deployment. Mentre gli sviluppatori si concentrano sul loro codice, gli operatori possono concentrarsi su una visione piú ampia di come i container possono essere uniti insieme in un'unitá funzionale. Il risultato ottenuto é una quantitá ottimale di portabilitá, dal momento in cui un Pod non é altro che la rappresentazione di piú immagini di container gestiti insieme.

4.2.3 Nodo Master

Kubernetes, come la maggior parte delle piattaforme distribuite, é composto da almeno un nodo master e nodi di lavoro multipli. Il nodo master é responsabile per l'esposizione delle API, per la pianificazione del deployment e la gestione dell'intero cluster. Attraverso l'esposizione delle API é possibile fornire un'interfaccia per poter interagire con il sistema. All'interno del nodo master non ci sono container in esecuzione. In alcuni casi, per garantire alta disponibilitá, i nodi master possono essere anche piú di uno. La definizione degli oggetti in Kubernetes, cosí come i Pods, i Replica sets e i servizi sono un compito del nodo master. Sono progettati per essere liberamente accopiati, estendibili e

adattabili a un'ampia varietá di carichi di lavoro. L'API fornisce questa estensibilitá ai componenti interni, nonché estensioni e container in esecuzione su Kubernetes. Il nodo master si compone di piú elementi, essi sono:

- **API Server**: Il server API forniscono un interfaccia per gli endpoint per interagire con altri componenti del sistema. Impiega etcd-storage come componente di archiviazione persistente. La CLI, l'interfaccia utente web o un altro strumento possono inviare una richiesta al server API. Il server elabora e convalida la richiesta, quindi aggiorna lo stato degli oggetti API in etcd e infine istruisce il servizio appropriato per eseguirla. Ció consente ai client di configurare carichi di lavoro e container tra i nodi worker.

- **etcd-storage**: etcd é un database di CoreOS chiave-valore distribuito leggero il quale agisce come unica fonte per tutti i componenti del cluster di Kubernetes, infatti altri componenti e servizi guardano le modifiche a etcd per mantenere lo stato desiderato di un'applicazione. Il nodo master utilizza *etcd* per memorizzare i suoi stati, ad esempio i job pianificati, creati e dstribuiti; dettagli e stato dei Pod; informazioni sulle repliche, e altro.

- **Scheduler**: Lo scheduler é responsabile di trovare un nodo corretto al quale assegnare un Pod o un servizio, in base ad alcuni vincoli, ad esempio cerca di bilanciare l'utilizzo delle risorse tra i vari nodi e assicura che i Pod si trovino su nodi che abbiano risorse libere a sufficienza, tenendo traccia dell'utilizzo delle risorse per garantire che il Pod non superi l'allocazione consentita. Oltre a tenere traccia dei fabbisogni di risorse e della disponibilitá delle risorse, possiede una varietá di altri vincoli e direttive forniti dall'utente; ad esempio, qualitá del servizio (QoS), requisiti di affinitá / anti-affinitá e localizzazione dei dati.

- **Controller**: Il controller é un loop che periodicamente controlla lo stato del cluster attraverso *kube-apiserver* ed esegue le opportune modifiche per far si che il cluster sia nello stato desiderato. Per stato desiderato s'intende il bilancio delle risorse dichiarate, utilizzate e richieste dal file di configurazione dei Pods, in base alle richieste e vincoli del sistema. Il controller mantiene lo stato stabile di nodi e Pod,

monitorando costantemente lo stato del cluster e i carichi di lavoro distribuiti sul cluster. Ad esempio, quando un nodo diventa non sano, i Pod in esecuzione su quel nodo potrebbero diventare inaccessibili. In tal caso, é compito del controller programmare lo stesso numero di nuovi Pod in un nodo diverso. Questa attivitá garantisce che il cluster mantenga lo stato previsto in qualsiasi momento.

4.2.4 Node

Il Node é l'elemento principale di Kubernetes, responsabile dell'esecuzione di workloads containerizzati. Il suo scopo é di esporre le risorse di elaborazione, di rete e di archiviazione alle applicazioni. Un nodo puó essere una macchina virtuale (VM) in esecuzione in un cloud o un server base metal all'interno del data center. I nodi workers possono essere molteplici all'interno di un cluster Kubernetes, e ciascuno di essi puó avere un numero variabile di Pod, mentre tutti possiedono tre principali componenti che permettono la corretta esecuzione dei container all'interno del nodo worker. Questi componenti sono:

- **kubelet**: Kubelet é il principale servizio di un nodo worker, responsabile della comunicazione con il nodo master. Garantisce che tutti i container su un nodo siano in buona salute, inoltre interagisce con il servizio di container runtime per eseguire operazione quali l'avvio, l'arresto, e la manutenzione di containers. Kubelet ottiene la descrizione della configurazione di un Pod dal kube-apiserver tramite API, per poi ordinare al servizio di container runtime di eseguire i container appropriati. Questo componente riporta anche al master lo stato di salute dell'host su cui é in esecuzione. Lo stato del nodo viene trasmesso al master ogni pochi secondi tramite messaggi heartbeat. Se il master rileva un errore del nodo, il controller di replica osserva questo cambiamento di stato e pianifica i Pod su altri nodi sani.

- **kube-proxy**: Kube-proxy é implementato come proxy di rete e bilanciamento del carico. Instrada il traffico al container appropriato in base al suo indirizzo IP e al numero di porta di una richiesta in arrivo. Esegue l'inoltro delle richieste ai relativi Pod/containers tra le varie reti isolate all'interno del cluster.

- **Container runtime**: Container runtime é il software responsabile dell'esecuzione dei container all'interno del nodo. Dopo che un Pod é schedulato sul nodo, il runtime estrae le immagini specificate dal Pod dal registro. Quando un Pod termina, il runtime elimina i container che appartengono al Pod.

4.2.5 Scalabilitá

Le applicazioni all'interno di Kubernetes sono eseguite come microservizi, che sono composti da piú container raggruppati in Pods. Ciascun container é progettato per eseguire solamente un unico task. I Pods possono essere composti da container con o senza stato (stateless/stateful). I Pods senza stato possono essere scalati facilmente quando necessario anche attraverso l'auto-scaling dinamico. Kubernetes supporta l'auto-scaling orizzontale dei Pod, che scala automaticamente il numero di Pod in repliche controllate basate sull'utilizzo della CPU. Quando un nuovo Pod viene scalato tra tutti i nodi disponibili, Kubernetes coordina l'infrastruttura sottostante per far si che ulteriori nodi vengano aggiunti al cluster. L'auto-scaling orizzontale é implementato come un loop di controllo, in cui periodicamente, ogni 30 secondi, il controller manager interroga l'utilizzo delle risorse rispetto alle metriche specificate. Nello specifico, il controller manager ottiene le metriche, come per esempio l'utilizzo della CPU, attraverso un API apposita per tale metrica, quindi se é settato un valore target di utilizzo, il controller calcola il valore di utilizzo come percentuale delle richieste alle risorse equivalenti contenute nei container di ciascun Pod. Se invece é impostato un valore grezzo, il controller utilizza direttamente tale valore. Il controller quindi assume o la media di utilizzo o il valore grezzo, per produrre poi un rapporto usato per scalare il numero di repliche desiderate. Da sottolineare il fatto che se alcuni container all'interno dei Pods non hanno il set di richieste di risorse rilevanti, l'utilizzo di CPU per quel dato il Pod non verrá definito e l'autoscaler non intraprenderá alcuna azione per tale metrica.

4.2.6 Fault Tolerance

In kubernetes i carichi di lavoro richiedono la disponibilitá sia a livello di infrastruttra che applicazione, nei cluster a larga scala, tutto é soggetto ai guasti, il che rende necessaria un'elevata attenzione nella gestione della fault tolerance. Kubernetes assicura un'alta disponibilitá per mezzo di *ReplicaSet*, *Replication Controller* e *StatefulSets*. Un ReplicaSet garantisce che un numero specificato di repliche di Pod siano in esecuzione in qualsiasi momento. Quindi l'utilizzatore di Kubernetes puó dichiarare il numero minimo di Pods che devono essere eseguiti in uno specifico istante di tempo. Se un container o un Pod si arresta in modo anomalo a causa di un errore, la policy dichiarativa puó riportare il deployment alla configurazione desiderata. Analizzando la disponibilitá rispetto all'infrastruttura, Kubernetes supporta un'ampia gamma di back-end di archiviazione, quali file systems distribuiti, gli storage persistente a livello di blocco, e infine container di storage. L'aggiunta di un livello di archiviazione affidabile e disponibile a Kubernetes garantisce un'elevata disponibilitá di workloads statici. Ogni componente del cluster di Kubernetes puó essere configurato per garantire un alta disponibilitá. Le applicazioni hanno quindi il vantaggio di usufruire di un bilanciamento del carico di lavoro e di checks di controllo per garantire disponibilitá.

4.3 Confronto Docker Swarm e Kubernetes

Docker Swarm e Kubernetes sono i principali tool di container orchestration, ed entrambi questi strumenti consentono di gestire un cluster di server su cui sono in esecuzione uno o piú servizi, ma sebbene entrambe le piattaforme di orchestrazione open source forniscano molte delle stesse funzionalitá, ci sono alcune differenze fondamentali tra il modo in cui queste due operano. Kubernetes supporta richieste piú elevate con maggiore complessitá, mentre Docker Swarm offre una soluzione semplice con cui é facile iniziare. Docker Swarm é stato molto popolare tra gli sviluppatori che preferiscono implementazioni rapide e semplicitá. Kubernetes é un tool molto piú efficiente, in quanto offre una maggiore scalabilitá e automazione, inoltre possiede delle proprie librerie per il monitoraggio e i logging, e ció viene a mancare in Docker Swarm, che deve ricorrere ad applicazioni di terze parti per sopperire a queste features. La seguente tabella mostra un confronto tra i due principali tool.

	Kubernetes	Docker Swarm
Scalabilitá	Supporta cluster con un massimo di 100 nodi , il 99% di tutte le chiamate API viene restituito in meno di 1 secondo e il il 99% dei pod e i relativi container hanno un tempo d'avvio entro 5 secondi.	Testato per supportare 1000 nodi e 30.000 container senza alcuna differenza evidente tra il tempo di avvio del primo o dell'ultimo nodo.
Disponibilitá	Altamente disponibile. I pod sono distribuiti tra i nodi worker. I nodi guasti vengono rilevati automaticamente e gestiti per evitare tempi di inattivitá.	Altamente disponibile. I container sono distribuiti tra i nodi Swarm per ridondanza e alta disponibilitá. Lo Swarm Manager gestisce le risorse su larga scala e viene replicato per garantire che siano altamente disponibili.
Load balancing	Kubernetes consente ai pod di essere definiti come un servizio. é quindi possibile impostare un bilanciamento del carico per accedere a questi servizi.	Lo swarm manager utilizza un bilanciamento del carico in ingresso per per esporre i servizi che vuole rendere visibili esternamente allo swarm. Inoltre utilizza anche il load-balancing internamente.
Auto scaling	Eccellente. Target come numero di pod e CPU-utilizzo-per-pod sono disponibili direttamente tramite l'API ed é suffciente impostare questi parametri e K8 assegna automaticamente le risorse per mantenerli.	Non supporta l'auto-scaling, ma é possibile dichiarare il numero di attivitá che si desidera eseguire e Swarm Manager si adatterá per mantenere questo stato quando si scala in su o in giú.

Tabella 4.1: Confronto tra Docker Swarm e Kubernetes virtuali

	Kubernetes	Docker Swarm
Performance	Kubernetes ha un'architettura piú complessa e flessibile di Docker e offre maggiori garanzie. Putroppo ancora queste cose rallentano le prestazioni	Docker ha un'architettura piú semplice ed in termini di prestazioni é piú veloce. Secondo alcuni test é 5X piú veloce di K8s in termini di start di un cluster di container e scalabilitá.
Easy to use	Kubernetes é piú difficile da configurare e da utilizzare in quanto piú complesso.	L'API di Docker Swarm condivide la CLI di Docker, inoltre é piú indicato per implementazioni rapide e semplici

Parte II

Obiettivo della tesi

Capitolo 5

Progettazione

L'aumento di dispositivi IoT nell'ambito delle applicazioni in tempo reale, domotica e l'Industria 4.0, necessitano di sistemi backend capaci di ottenere basse latenze per fornire un'interazione continua con il dispositivo finale, imponendo requisiti di qualitá del servizio molto rigorosi. Le applicazioni multimediali in tempo reale basate su cloud richiedono latenze end-to-end inferiori a 60 ms e valori molto piú bassi in contesti specifici come quello industriale [1]. L'unico modo per soddisfare tali requisiti é spostare i servizi che solitamente vengo eseguiti sul cloud o all'interno di un server centrale ai margini della rete, in modo da ridurre i ritardi altrimenti inaccettabili fornendo localmente le risorse necessarie. Per fronteggiare il problema sono stati proposti vari modelli, come l'Edge computing, la quale idea principale é quella di avere servizi in prossimitá dei dispositivi che ne fanno uso in modo da migliorare l'esperienza di servizio in termini di qualitá del servizio. L'Egde Computing, come giá descritto all'inizio di questo elaborato, fornisce risorse di archiviazione, calcolo e rete offrendo la possibilitá di ospitare servizi ai margini della rete per diminuire la latenza. Una delle principali problematiche di questo paradigma é che i nodi Edge possono essere dispositivi con risorse limitate rispetto alle tradizioni soluzioni, Un dispositivo Edge potrebbe essere per esempio un Raspberry Pi; l'Edge Computing, di conseguenza, oltre ad apportare numerosi vantaggi ha introdotto nuove problematiche, una di queste é la migrazione dei servizi e dei loro dati tra i nodi di un sistema che opera ai margini della rete. Allo stato attuale ci sono diverse tecnologie di virtualizzazione eterogenee e strategie di migrazione e alcune di esse potrebbero non

essere pratiche se utilizzate con dispositivi Edge poveri di risorse; ció rende la gestione della migrazione dei servizi un'attivitá molto complessa. Concentrandosi sulle soluzioni esistenti, la maggior parte degli sforzi fondamentali si é concentrata sul concetto di Live Migration of Virtual Machines (VM) per garantire il minor tempo di inattivitá possibile del servizio [4] . La migrazione in tempo reale definisce la migrazione del servizio, come la migrazione del servizio stesso, dei suoi dati e delle sue variabili d'ambiente al fine di ottenere il servizio migrato nelle stesse condizioni che si trovava prima della migrazione. Per garantire ció, sono stati proposti meccanismi complessi che includono le due principali varianti chiamate pre-copia e post-copia. La pre-copia invia la maggior parte dei dati all'host di destinazione prima di arrestare e migrare la VM . La post-copia estrae la maggior parte dei dati dall'host di origine dopo la ripresa della VM nell'host di destinazione [4]. Successivamente, con la diffusione delle tecnologie dei container come Docker, la maggior parte degli sforzi di ricerca si é concentrata sulla migrazione dei container di servizi.
L'obbiettivo di questa tesi é quello di creare un'applicazione capace di interagire con i container Docker presenti in un host e spostarli, con annessi i dati e le variabili di stato, in base a determinate metriche, su di un altro host per garantire che il servizio venga sempre eseguito all'interno di un nodo capace di soddisfare in termini computazionali le richieste per la quale il servizio é stato creato.

5.1 Architettura del sistema

L'applicazione creata, é un servizio che viene eseguito all'interno di piú nodi presenti in una rete al fine di poter creare un cluster. All'interno del cluster é presente un nodo Leader e vari nodi Follower, l'elezione del Leader avviene periodicamente all'interno del cluster e a vincere l'elezione é il nodo che risulta in quel momento piú performante secondo un analisi svolta da una funzione chiamata funzione di fitness, la quale restituisce un valore compreso tra 0 e 1, piú il valore é vicino all'1 piú il nodo é performante, viceversa piú il valore si avvicina allo 0, piú il nodo non é performante. Il compito principale del Leader é quello di raccogliere le metriche dei nodi Follower della rete e ordinare la migrazione di un servizio da un nodo a un altro per garantire l'equa distribuzione dei

container. I compiti dell'applicazione sono i seguenti:

- Gestione completa di Docker per ogni nodo nodo;
- Analisi locale per determinare la funzione di fitness del nodo;
- Creare un cluster di nodi;
- Indigere elezioni periodiche.
- Interazione con il Leader qualora sia un Follower;
- Raccolta delle metriche dei nodi del cluster qualora il nodo é un Leader;
- Gestire il processo di migrazione di un contaier da un nodo a un altro piú preformate qualora il nodo é un Leader.

La creazione del cluster avviene in maniera automatica con l'ausilio di un protocollo di service discovery il quale permette di scoprire applicazioni di una rete, una volta che il cluster é stato creato la comunicazione tra le varie applicazioni avviene mediante Socket. I componenti dell'applicazione sono visibili nell'immagine 5.1; essi sono:

- **Docker Manager**: É il componente che si occupa della gestione completa di Docker, dal Run di un container al Build di un immagine, incluso l'esportazione di un immagine, utile per la migrazione, processo che verrá descritto in maniera dettagliata nel prossimo capitolo, riceve i compiti da svolgere dall'Interface Operazion.
- **Fitness Analayzer**: Si occupa di produrre un risultato della funzione di fitness che viene utilizzata nel sistema per capire se un nodo é libero da carichi di lavoro e quindi atto a riceverne dei nuovi, o viceversa se in quel momento é gravato di lavoro e quindi sarebbe una buona scelta ripartire il suo lavoro tra piú nodi.
- **Fitness Data Sender**: É quel componente che si occupa di inviare il valore di fitness precedentemente analizzato sia tra i componenti interni dell'applicazione che ai nodi del cluster qualora esso sia un Follower.

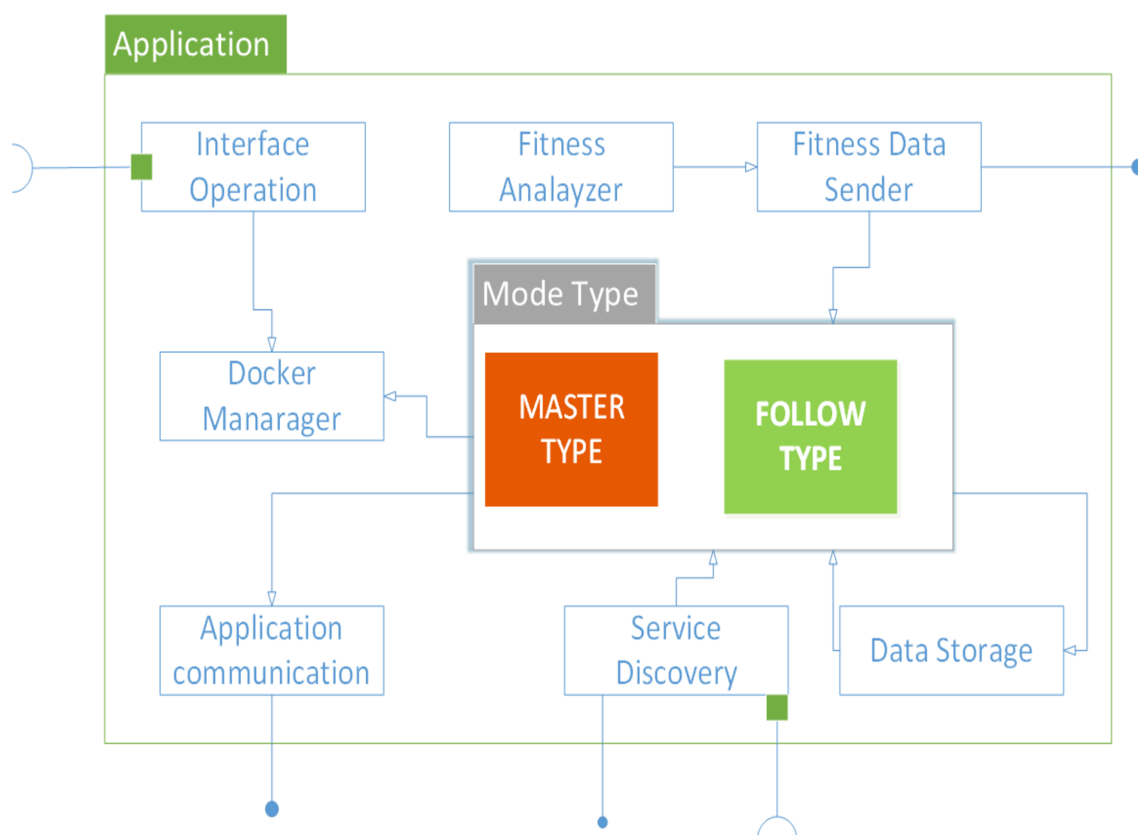

Figura 5.1: Diagramma dei componenti

- **Interface Operation**: É quel componente che espone l'interfaccia dell'applicazione, utile al Master, per esempio, qualora debba commissionare al nodo un operazione, una volta ricevuto un ordine lo invia al Docker Manager che lo eseguirá.

- **Application Comunication**: Componente che utilizza l'interfaccia esposta dall'Interface Operation component per comunicare con altre applicazioni.

- **Mode Type**: É quel pacchetto che si occupa di settare l'applicazione come Master o come Follower, al suo interno sono presenti due moduli, Master Type e Follow Type, solo uno dei due puó essere attivo. Qualora sia attivo il Master Type sará in eseguzione l'algoritmo che si occupa di analizzare le migrazioni del cluster e produce le operazioni da svolgere; Se ad essere attivo é il Follow Type saranno in

esecuzione i Thread che si occupano di inviare periodicamente le informazioni al Leader e di analisi sulla continua disponibilitá del Leader.

- **Service Discovery**: É quel modulo che si occupa di scoprire nuove applicazioni al fine di creare o ampliare il cluster, oltre ad utilizzare un'interfaccia ne espone una, l'interfaccia esposta serve per essere scoperto da altri nodi sulla rete.

- **Data Storage**: É quel componente che si occupa di salvare le informazioni che l'applicazione necessita, se l'applicazione é un Leader allora salverá, per esempio, la lista dei nodi del cluster, qualora sia un Follower, salverá i riferimenti del leader.

5.2 Scelte progettuali

L'applicazione é sviluppata in Java, versione 1.8, ed é stata sviluppata per essere eseguita all'interno di kernel linux. Il motivo per la quale é stato optato un approccio distribuito, piuttosto che uno centralizzato é dato dal fatto che uno degli obbiettivi é far in modo che il Leader ragnante differisca nel tempo. Due sono le principali librerie utilizzate per la creazione dell'applicazione e sono com.github.docker-java e org.fourthline.cling

5.2.1 com.github.docker-java

La dichiarazione della libreria dal gestore delle dipendenze di Java é la seguente:

```
<dependency>
        <groupId>com.github.docker-java</groupId>
        <artifactId>docker-java</artifactId>
        <version>3.1.5</version>
</dependency>
```

La libreria com.github.docker-java offre la possibilitá di far interloquire un'applicazione Java direttamente col Docker Demon su una specifica porta mediante TPC/IP; questo porta il vantaggio che non solo la libreria puó comunicare col Docker Daemon in localhost ma un ipotetico nodo A puó comunicare col Docker Daemon di un nodo B a patto che i due nodi possano interloquire mediante una rete. La libreria rende molto

semplice la comunicazione tra un'applicazione Java e un Docker Daemon, per esempio, per creare un client é sufficente il seguente eseguire il seguente codice:

```
DefaultDockerClientConfig defaultDockerClientConfig = DefaultDockerClientConfig.
createDefaultConfigBuilder().withDockerHost("tcp://0.0.0.0:5555").build();

DockerClient dockerClient = DockerClientBuilder.
getInstance(defaultDockerClientConfig).build();
```

L'esempio precedere utilizza la porta 5555, per essere fruibile é necessario attivarla, la configurazione della porta é possibile andandola a settare nel file di docker */lib/systemd/system/docker.service*; La configurazione del file fatta é la seguente:

```
ExecStart=/usr/bin/dockerd -H fd:// -H=tcp://0.0.0.0:5555
--containerd=/run/con
```

5.2.2 org.fourthline.cling

La dichiarazione della libreria dal gestore delle dipendenze di Java é la seguente:

```
<dependency>
       <groupId>org.fourthline.cling</groupId>
       <artifactId>cling-core</artifactId>
       <version>2.1.2</version>
</dependency>
```

La libreria org.fourthline.cling é necessaria per poter effetturare il service discovery con UpNp. La libreria garantisce di poter individuare altre applicazioni che si espongono in UpNp e di poter comunicare con loro.

5.3 Service Discovery

Service discovery é un termine usato per descrivere i protocolli e meccanismi attraverso cui un'applicazione o componente software diventa parte della rete a cui é connesso

e ne scopre ulteriori servizi disponibili. Le tecnologie di service discovery sono state sviluppate per semplificare l'uso di dispositivi mobili in una rete permettendo loro di essere scoperti, configurati ed usati da altri dispositivi. La fase di service discovery termina con l'indicazione della disponibilitá di uno o piú servizi. Un protocollo di discovery é caratterizzato in generale da diversi elementi, essi sono:

- Service: é il servizo in se;
- Client agent: é il dispositivo che cerca i servizi di cui ha bisogno;
- Service agent: é il componente software che implementa un servizioe lo mette a disposizione dei client agent.

Al fine di ricercare un servizio, o una risorsa, é necessario definire un meccanismo unico di identificazione, ovvero un insieme di identificatori utilizzati per definire univocamente un'entitá. Ogni servizio ha un nome e tramite questo viene ricercato, invocato ed infine utilizzato dal client agent. Il protocollo di discovery restituisce la lista dei servizi che rispondono ai service attributes selezionati dal client nella fase di discovery.

5.3.1 UpNp

UpNp é un protoccolo di servie discovery, ed é quello utilizzato nel progetto di questo lavoro di tesi, é stato scelto principalmente per la sua universalitá, dato che non necessita di driver specifici, i dispositivi che lo utilizzano possono, quindi, comunicare direttamente fra loro, dando vita a reti peer-to-peer, questo é il secondo motivo per la quale é stato scelto in quanto la tolopogia della rete che si desidera creare é per l'appunto peer-to-peer. I fruitori principali di una rete UpNp sono dei client i quali sono alla ricerca di dei servizi, ed ad ognuno di esso é associato un XML di descrizione del servizio. un'applicazione mette a disposizione i proprio servizi e il punto di controllo, PoC (Point of Control), che é un controller in grado di rilevare e controllare i servizi dopo il rilevamento. La tecnologia UpNp si basa su una serie di protocolli IP standard aperti, quali TCP/IP, HTTP e XML. UpNp utilizza SSDP (Simple Service Discovery Protocol) il quale definisce le modalitá di rilevazione dei servizi di una rete. Il protocollo SSDP si basa sui protocolli HTTPU e HTTPMU e definisce i metodi che consentono sia a un punto di controllo di individuare

le risorse di interesse nella rete, sia di comunicare la loro disponibilitá nella rete. Dopo l'avvio, un punto di controllo UpNp puó inviare una richiesta di ricerca SSDP per rilevare i servizi disponibili nella rete, tale richiesta é contenuta in un messaggio multicast di discovery all'indirizzo: 239.255.255.250:1900. Il punto di controllo puó impostare criteri di ricerca piú restrittivi, in modo da trovare solo servizi di un certo tipo, utlizzando delle query specifiche le quali filtrano per il nome del servizio per esempio. I servizi UpNp ascoltano la porta multicast e dopo aver ricevuto una richiesta di ricerca, il servizio lo esamina i criteri di ricerca per stabilire se soddisfa i propri criteri. In caso affermativo, al punto di controllo viene inviata una risposta in un messaggio contenente il puntamento che rimanda all'XML dove é descritto il dispositivo stesso e i suoi servizi. Quando un servizio UpNp viene collegato in rete, invia dei messaggi di presenza SSDP che rendono noti i servizi offerti, questi annunci sono contenuti all'interno di messaggi multicast di advertisement. Sia gli annunci di presenza e i messaggi di risposta contengono un puntatore alla posizione del documento di descrizione.

5.3.2 Fasi di UpNp

Il processo attraverso il quale qualunque dispositivo si collega alla rete é costituito da diverse fasi, le fasi sono le seguenti.

Rilevazione

Dopo che il servizio si é connesso alla rete avviene la rilevazione gestita dal protocollo SSDP. Quando si aggiunge un'applicazione alla rete, il protocollo SSDP consente a tale applicazione di rendere pubblici i propri servizi ai punti di controllo della rete. La rilevazione avviene con un un messaggio che contiene poche specifiche essenziali relative all'applicazione, come l'ID e un puntatore al documento di descrizione della applicazione. Il descittore XML utilizzato in questo lavoro di tesi é visionabile nell'appendice A.1.

Descrizione

La fase successiva é la descrizione dei servizi offerti. Dopo la rilevazione, un punto di controllo per ottenere informazioni sui servizi che l'applicazione offre per interagire con

essa, per farlo é necessario ottenere il puntamento al descrittore dei servizi, la modalitá é la stessa del descrittore di un'applicazione.

Controllo

Per controllare un servizio, é necessario ottenere la descrizione UpNp dettagliata di ciascun servizio. Anche la descrizione di un servizio é in formato XML e include l'elenco dei comandi o delle azioni ai quali il servizio risponde e degli argomenti per ogni azione. Per controllare un servizio, un punto di controllo invia una richiesta di azione a un servizio specifico. A tal scopo, il punto di controllo invia un messaggio di controllo all'URL del servizio. I messaggi di controllo sono anch'essi espressi in formato XML e la comunicazione avviene utilizzando il protocollo SOAP.

Il diagramma di sequenza del processo é il seguente:

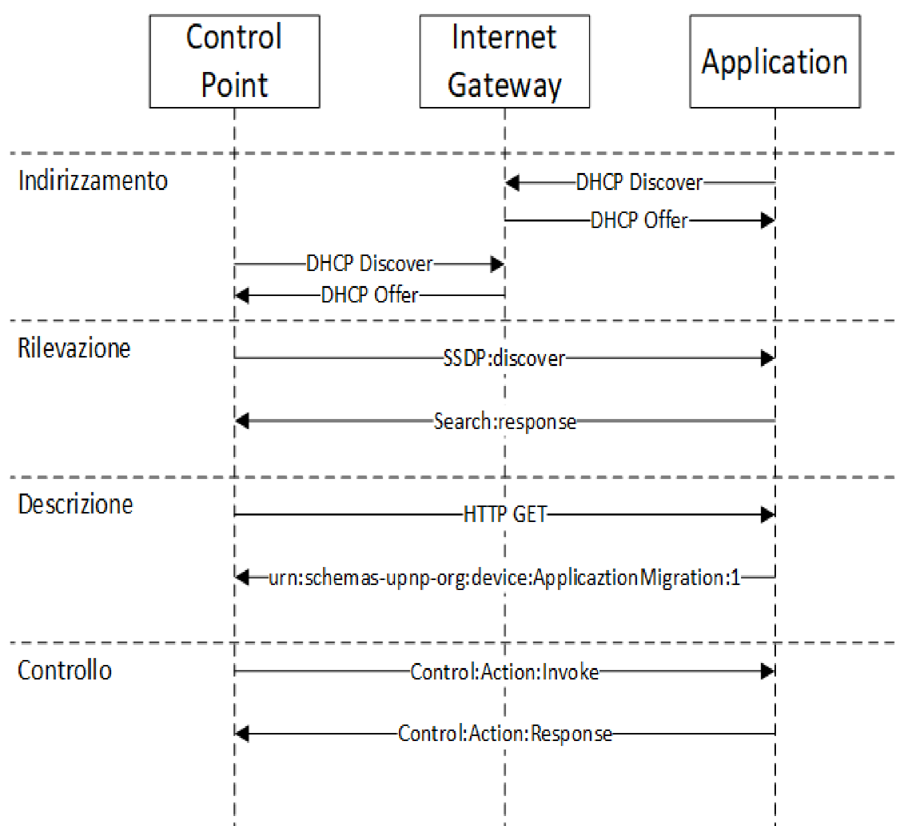

Figura 5.2: Diagramma di Flusso di UpNp

Capitolo 6

Implementazione

In questo capitolo vengono spiegate le principali implementazioni svolte nello sviluppo dell'applicazione, le quali riguardano, la migrazione di un'immagine, la creazione del cluster, la funzione di fitness e gli algoritmi di bilanciamento del cluster. Durante questo capitolo si faranno esempi concreti utilizzando il termine nodo che durante la fase di sviluppo, test e validazione, per rimanere nel contesto dell'Edge computing, sono stati utilizzati diversi Raspberry pi 3 b+, i quali hanno un processore quad-core a 64bit, 1 GB di ram e 32 GB di disco, ed erano in comunicazione gli uni con gli altri mediante cavo ethernet collegato direttamente a un modem.

6.1 Migrazione di un'immagine

Il primo step implementativo ha riguardato lo studio delle varie tipologie di migrazione di un container. Per migrazione si intende una tecnica che, data una situazione iniziale dove un ipotetico nodo A esegue un container, si ottiene una situazione finale dove un secondo ipotetico nodo B esegue la stessa immagine. Le immagini si dividono principalmente in due macro categorie nel contesto dell'applicazione, immagini data-less e data-full, ovvero immagini che vengono eseguite senza l'ausilio di volumi, un esempio é un'immagine nginx, o nel secondo caso un'immagine mysql la quale richiede un volume dove persistere i propri dati. La migrazione di un'immagine, indipendentemente se legata a una persistenza puó avvenire con l'utilizzo del registry di Docker, ma in questo

caso non si tratta di migrazione, ma del fatto che un nodo A ordina a un nodo B di scaricarsi un'immagine e di eseguirla. Questa modalitá presenta due grossi problemi, il primo é che il nodo B, il ricevitore del comando di pull, potrebbe non essere collegato a una rete abile a connettersi col registry di Docker, e in secondo luogo presenta un problema legato alla latenza della rete. La tipologia appena descritta é stata analizzata nel corso dell'implementazione e il tempo medio di 10 esperimenti per scaricare un'immagine tobi312/rpi-nginx e di eseguirla é stata di 115,4 secondi. Per risolvere il problema della latenza é stato implementato un Docker registry sulla stessa LAN dove i nodi A e B sono in esecuzione, all'interno di un computer desktop di potenza computazionale superiore a quella dei nodi. Quello emerso é che il tempo medio di pull e run di un'immagine é stato di 47,4 secondi, anche in questo caso la stima é stata fatta con 10 esperimenti. Il problema ancora presente utilizzando questa tipologia é che il registry potrebbe non essere disponibile per determinati quanti di tempo, oppure le richieste potrebbero crescere nel tempo e far si che il tempo stimato di pull cresca notevolmente. Per entrambi gli scenari precedentemente analizzati, la migrazione avveniva mediante l'utilizzo di un registry e al fine del lavoro di tesi svolto si voleva utilizzare un tipologia piú sofisticata. Per risolvere il problema si é optato per il save e il load di un'immagine. Il comando save produce un'immagine in un archivio .tar, il comando load carica un'immagine da un archivio tar all'interno di Docker. I passaggi che i nodi compiono solo i seguenti:

- Il nodo A crea l'archivio .tar dell'immagine utilizzando la libreria, il comando é il seguente:

```
InputStream inputStreamImage = DockerClient.
saveImageCmd("id dell'immagine da migrare").exec();
```

- Ottenuto l'InputStream dell'immagine essa viene inviata mediante socket al nodo B il quale lo deposita all'interno del proprio file system in una cartella accessibile all'applicazione;

- Una volta terminato il processo di spostamento del file il nodo A invia al nodo B il comando di load dell'immagine precedentemente inviata, nel messaggio sono presenti anche tutte le configuarazioni con la quale l'immagine é stata avviata

all'interno del nodo A, come ad esempio la porta con la quale il container si espone, o nel caso specifico di un container mysql, la password di sistema;

- una volta che il nodo B ha ricevuto il comando di load, carica l'immagine e ne esegue il run con i paramentri precedentemente ricevuti;
- quando il nodo ha terminato il processo che produce il run dell'immagine invia un messaggio di successo al nodo A il quale termina tutti i processi del container migrato.

Utilizzando questo approccio il container sul nodo A rimane attivo fino a quando lo stesso non é in esecuzione nel nodo B. Anche in questa casististica é stato analizzato il tempo medio di migrazione facendo una media di 10 migrazioni, in questo caso il nodo A per salvare e inviare un contaier tobi312/rpi-nginx impega 24,7 secondi, il nodo B per termianre i sui compiti impega 46,7 secondi per un totale di 71,4 secondi. Il risultato finale potrebbere essere abbassato a 46,7 qualora il nodo A avesse giá depositato nel proprio file-system il file .tar dell'immagine, a questo punto il suo unico compito sarebbe quello di inviarlo al nodo B. Il diagramma di flusso UML che raffigura il processo é visibile nell'immagine 6.1.

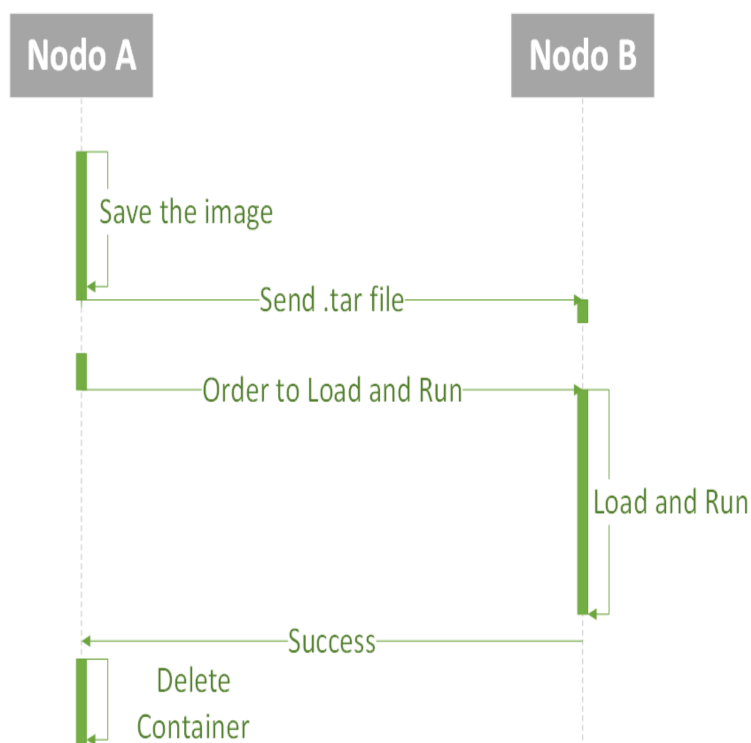

Figura 6.1: Diagramma di Flusso descrivente la migrazione di un'immagine da un nodo A a un nodo B

6.1.1 Migrazione di un volume

La migrazione di un volume é parte integrante della migrazione di un container tra due nodi. Per migrazione di volume si intende spostare tra un nodo a un altro tutte le cartelle, chiamate volumi, che il container Docker necessita durante la propria fase di run; per esempio un container mysql utilizza una cartella dove salva all'interno tutti i dati criptati che utilizza per creare gli schemi e salvarne i contenuti. La prima modalitá analizzata é stata quella di utilizzare un file-system distributo, ed in particolar modo NFS dato che Docker offre un'integrazione diretta per la creazione di volumi distribuiti NFS, un esempio é il seguente:

```
sudo docker volume create --driver local \
--opt type=NFS \
--opt o=addr=10.0.0.5,NFSvers=4 \
--opt device=:/export/rednode \
noderedvolume
```

All'interno del precedente esempio, il server 10.0.0.5 espone una cartella dove é possibile creare un volume, ed in particolar modo il volume é denominato noderedvolume, in questo esempio il container preso in analisi é nodered; per poter utilizzare il volume é sufficente utilizzare il comando di run settandone il volume, per esempio:

```
sudo docker run -it -p 1880:1880 -v noderedvolume:/data
--name mynodered -d nodered/node-red
```

Questo approccio ha due tipi di problemi, il primo é legato alla fault tollerance del calcolatore che ospita il server NFS, il secondo, di origine piú pratica, non tutti i container sono in grado di utilizzare volumi NFS, un esempio pratico é mysql. Le modalitá utilizzate al fine del lavoro di tesi svolto sono due, la prima riguarda la migrazione di una cartella fisica utilizzata da un container Docker come volume, la seconda comprende la migrazione di un volume gestito direttamente da Docker. Nel primo caso é sufficiente creare uno zip della cartella, inviarlo nel nodo di destinazione, e avviare il container facendogli puntare la cartella come volume. Per analizzare le performance é stato analizzato lo spostamento di una cartella utilizzata come volume da un'immagine di mysql la quale aveva persistito un database con all'interno 40000 righe, sono stati effettuati 10 test e vi é stata calcolata la media. Il nodo A per stoppare il container, zippare l'immagine e inviarla al nodo B impiega di media 1,1 secondi; Il nodo B per estrarre il contenuto dallo zip, e avviare il container che punti a quella direcotry impiega di media 6,8 secondi. Questo approccio é molto veloce ma presenta il problema di limitazione del file-system, il file-system del nodo A potrebbe non essere lo stesso del Nodo B; Docker suggerisce l'utilizzo dei propri volumi piuttosto che utilizzare una cartella e farla diventare un volume di Docker. Per risolvere il problema é stata effettuata la migrazione dei dati di un volume gestito la Docker. La casistica iniziale é un nodo A, che esegue un container che utilizza un volume creato da Docker col comando:

```
docker volume create my-volume-2
```

Al fine dell'analisi sulle performace il volume é utilizzato da un container mysql che esegue all'interno uno schema con 40000 righe. Il secondo nodo, é privo del volume. Il primo passaggio da svolgere é l'export in formato .tar del contenuto di quel volume utilizzando un container ubuntu. Il comando é il seguente:

```
sudo docker run --rm -v my-volume-2:/recover -v ~/backup:/backup
ubuntu bash -c "cd /recover && tar xvf /backup/some-mysql.tar"
```

Il comando esposto utilizza un container ubuntu connesso al volume dove é presente il db e mediante un comando bash crea un file .tar di tutto il contenuto. A questo punto il file compresso viene mosso dal nodo A al nodo B. Il nodo B per prima cosa deve creare il volume, e per scompattare il contenuto del file .tar all'interno del volume appena creato deve utilizzare il seguente comando:

```
sudo docker run --rm --volumes-from some-mysql -v
/home/pi/migrationFile/tarData:/backup ubuntu bash -c
"cd /var/lib/mysql && tar cvf /backup/some-mysql.tar .
```

Eseguendo gli step implementativi appena descritti si ottiene che gli stessi dati presenti nel volume del nodo A sono presente nel nodo B col vantaggio che la creazione del volume e la sua persistenza sul file-system siano direttamente gestiti da Docker. Il nodo A per stoppare l'immagine, compattare il volume e inviarlo impiega 8,1 secondi, il nodo B per scopattarlo ed eseguire l'immagine impiega 13,1 secondi; anche in questo caso i valori sono una media di 10 esecuzioni.

6.2 Funzione di Fitness

La funzione di fitness é una funzione che viene calcolata per ciascun nodo, la quale restituisce un valore compreso tra 0 e 1 e indica quanto il nodo stesso sia performante; piú il valore si avvicina a 1, piú la board é performante, ovvero libera da processi che occupano molte risorse, viceversa, se il valore della funzione tende allo 0 allora significa che il nodo possiede molti processi e le risosrse fruibili sono limitate. La funzione di fitness, come verrá spiegato nel dettaglo nei prossimi sotto capitoli serve per l'elezione

del Leader e come metrica principale per la scelta del container da migrare. La funzione di fitness viene calcolata utilizzando tre metriche, la percentuale di non utilizzo della cpu, la percentuale di non utilizzo della ram e sempre la percentuale di non utilizzo del disco. Ad ogni metrica é associato un peso che viene letto dal file di configurazione all'avvio dell'applicazione con all'interno il valore da associare a ciascuna metrica, i valori sono espressi da tre variabili: alfa, beta e gamma, e riguardano rispettivamente la cpu, ram e il disco. La somma dei pesi deve essere 1. Un esempio é:

- alfa: 0.5
- beta: 0.2
- gamma: 0.3

Un esempio di valori computazionali estrapolati sono:

- percentuale di utilizzo della cpu: 22%; percentuale di non utlizzo: 78%;
- percentuale di utilizzo della ram: 60%; percentuale di non utlizzo: 40%;
- percentuale di utilizzo del disco: 30%; percentuale di non utlizzo: 70%;

Il motivo per la quale si usano le percentuali di non utilizzo é motivato dal fatto che ad interessare la funzione di fitness sono le risorse disponibili piuttosto di quelle utilizzate. Il valore, per esempio della cpu, che é espresso in percentuale, viene trasformato e ridotto con una proporzione andando a sostituitre alla base 100 il valore corrispondente, nel caso della cpu, alfa. Riprendendo l'esempio precedente, la percentuale della cpu é 78%, alfa é impostato a 0.5 allora la "sotto funzione" di fitnes sará 0.39, il valora é dato dalla seguente proporzione:

$78 : 100 = X : 0.5$

La funzione di fitness é data dalla somma delle tre "sotto funzioni", riprendendo l'esempio la funzione di fitness finale sará: 0.68.

6.3 Creazione del cluster ed elezione del Leader

Alla base della creazione del cluster c'é il protocollo di service discovery UpNp. Quando un nodo si avvia, la prima operazione che svolge é calcolare la propria funzione di

fitness, il valore viene utilizzato come countdown per indire l'elezione, se il valore della funzione di fitness del nodo equivale a 0, caso pessimo, il nodo prima d'indire l'elezione dovrá aspettare 5 minuti, nel caso in cui il nodo restituisca 1 puó indire immediatamente le elezioni, nel caso dell'esempio precedente, dove la funzione di fitness equivaleva a 0.68, il nodo prima d'indire l'elezione dovrá aspettare (100 - 68%) di 5 minuti, ovvero 1 minuto e 36 secondi. Nel momento in cui il primo nodo indice l'elezione, invia tramite UpNp a tutti i nodi l'ordine d'indire l'elezione, ogni nodo a questo punto invia in broadcast il proprio ip e la propria funzione di fitness. I nodi rimangono in ascolto di ricevere i valori dagli altri nodi per 15 secondi, quando il tempo é scaduto ogni nodo ordina, in ordine decrescente i valori di fitness che hanno ricevuto. Terminato l'ordinamento la prima tupla sará la coppia fitness e ip del Leader, ogni nodo analizza l'ip al primo posto della lista col prorio, se corrisponde invia a tutti gli altri nodi la notifica di essere il Leader. Quando un nodo riceve la notifica di presentazione di un Leader, aggiorna la propria variabile di stato interna settandola con il valore del Leader, e arresta il proprio timer di contdown per indigere l'elezione. Tutti i nodi escluso il Leader ogni minuto inviano il proprio valore di fitness al Leader. Dopo un'elezione ogni nodo, incluso il Leader avviano un thread con un timer, il timer é settato sempre come (100 - percentuale di fitness) di 60 minuti, al termine del timer vengono indette delle nuove elezioni, questo garantisce che il Leader differisca nel tempo.

Il diagramma di flusso 6.2 mostra le fasi d'elezione di un Leader, per semplicitá si assume che i tre nodi vengano avviati nello stesso istante temporale. Nella fase 1 ci sono tre nodi, il nodo con il valore di fitness maggiore é il nodo A, con valore 0.852, il che significa che dopo l'avvio saranno aspettati 42 secondi prima di indigere le elezioni. Nella fase 2 il nodo A indice l'elezione e nella fase 3 tutti i nodi si scambiano il loro valore di fitness. Nella fase 4 i valori sono analizzati e nella fase 5, il nodo con la funzione di fitness maggiore si presenta come Leader.

6.3.1 Aggiunta di un nuovo nodo

Quando un nodo si avvia, la prima chiamata UpNp che viene fatta é quella per scoprire se é presente un cluster con un Leader; qualora non sia presente, il nodo si mette in attesa di indire l'elezione o di farla indire. Nell'ipotesi che il Leader sia giá

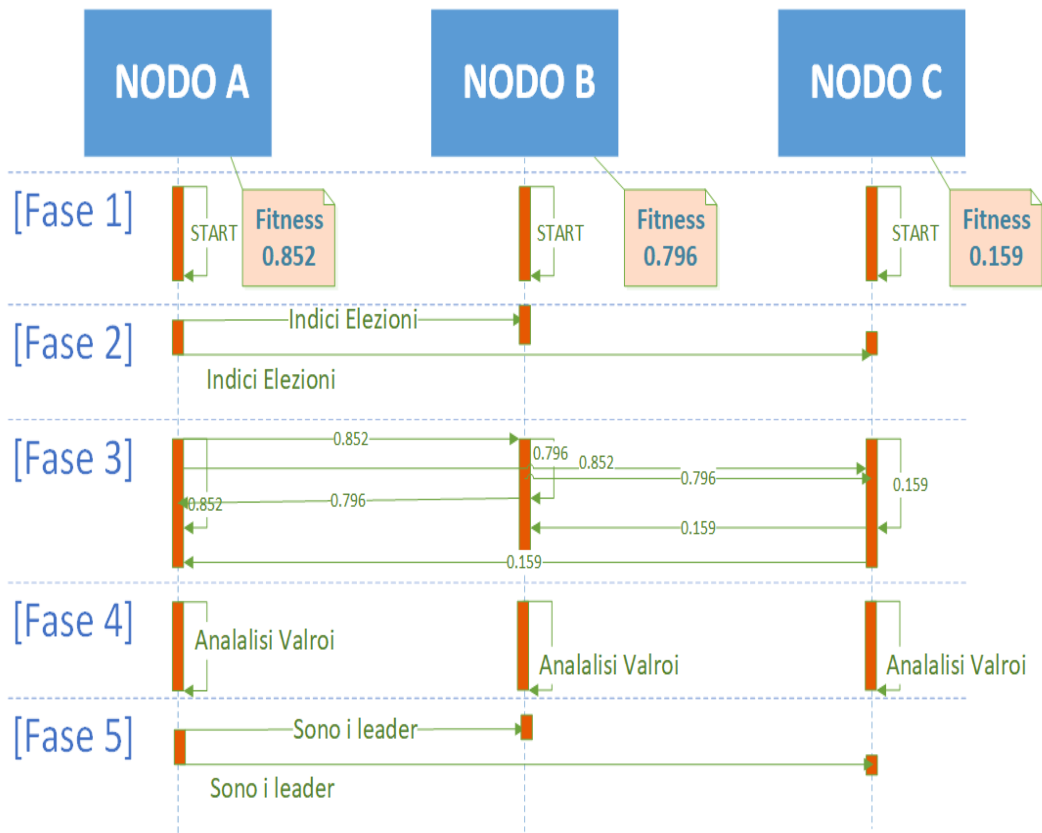

Figura 6.2: Diagramma di Flusso descrivente l'elezione del Leader

presente avendo effettuato una chiamata in broadcast tutti i nodi gli risponderanno con l'ip del Leader, a questo punto il nuovo nodo inizia a prendere parte al cluster ed a inviare i dati al Leader designato.

6.3.2 Fault Tolerance dell'applicazione

Uno dei grandi vantaggi di UpNp é dato dal fatto dopo che due applicazioni quando si sono riconosciute e sono entrate in contatto, esse si scambiano dei messaggi periodici per determinarne lo stato; la libreria utilizzata, in maniera totalmente trasparente, notifica qualora un nodo non fosse piú disponibile sulla rete. Se la libreria notifica la caduta di un ip che corrisponde al valore salvato nella variabile di stato come Leader comporta al fatto che il Leader non sia piú disponibile sulla rete. Il primo nodo che riconosce la

caduta del Leader notifica tutte le altre con una chiamata che comporta l'immediata elezione di un nuovo Leader.

6.4 Algoritmi di Migrazione

Il compito principale del Leader nel contesto di questo lavoro di tesi é quello di migrare i container tra un nodo del cluster e un altro al fine di mantenere una funzione di fitness paritaria tra i nodi; durante la fase di avvio, oltre a leggere i parametri necessari per la funzione di fitness viene caricato il parametro Omicron. Omicron é quel valore che, dati due nodi, indica la differenza fitness massima che due nodi devono tenere, se la differenza supera Omicron é necessario effettuare una migrazione di un container. Per esempio, se Omicron é settato come 0.10, e il nodo A ha una funzione di fitness pari a 0.20 e il nodo B ha una funzione di fitness pari a 0.35, la differenza tra i due nodi é 0.15 che eccede Omicron di 0.05, in questo caso é necessaria una migrazione. La migrazione avviene come precedentemente descritto, e a scegliere il container da migrare é un algoritmo di migrazione. Durante questo lavoro di tesi sono stati sviluppati tre tipi di algoritmi, Random, Lazy e il problema dello Zaino. Prima di introdurre gli algoritmi é necessario introdurre come viene valutato un container; per ogni container viene calcolata una funzione di fitness, che ha la stessa logica di quella precedentemente descritta, solo che in questa viene analizzato l'utilizzo di CPU e RAM di ogni singolo contaier, i pesi vengono letti dal fail di configurazione, essi sono fitnessContainerRam e fitnessContainerCpu. Per ottenere le metriche sulla quale applicare i pesi di ogni singolo container viene effettuata la seguente chiamata:

```
Docker stats --no-stream --format "table
{{.Name}}\t{{.Container}}\t{{.CPUPerc}}\t{{.MemUsage}}"
```

La quale restituisce per ogni container la percentuale di cpu e ram utilizzata. Un esempio di chiamata é la seguente:

```
pi@raspberrypi:~/Desktop/dev/ssh $ docker stats --no-stream --format "table {{.Name}
}\t{{.Container}}\t{{.CPUPerc}}\t{{.MemUsage}}"
NAME                CONTAINER           CPU %               MEM USAGE / LIMIT
somma4              44010c15f965        98.79%              24.36MiB / 926.1MiB
somma3              50192ca48180        93.22%              25.63MiB / 926.1MiB
somma2              40e917569482        93.79%              26.21MiB / 926.1MiB
somma1              e128d0829501        94.15%              26.93MiB / 926.1MiB
pi@raspberrypi:~/Desktop/dev/ssh $
```

Figura 6.3: Chiamata Docker stat

Di seguito vengono descritti gli algortimi di migrazione sviluppati, la loro validazione é descritta nel prossimo capitolo.

6.4.1 Algoritmo Random

L'agoritmo Random inizia con l'ottenimento del valore di fitness da ogni nodo, una volta ottenuti, il nodi vengono ordinati in ordine decrescente; se la differenza del valore di fitness del primo elemento della listra e dell'ultimo elemento della lista é maggiore di Omicron allora é necessario effettuare la migrazione di un container dal nodo meno performante al nodo piú performante. L'algoritmo Random determina il container da spostare in maniera randomica e ne viene effettuata lo spostamento. Una volta effettuata la migrazione viene richiamata l'analisi in maniera ricorsiva; l'algoritmo termina quando la differenza tra i valori di fitness é minore di Omicron.

6.4.2 Algoritmo Lazy

Come l'algoritmo Random, l'algormitmo Lazy inizia il proprio ciclo raccogliendo i valori di fitness dai nodi e analizzandoli, solo che al posto di migrare un container random, viene mosso il container che occupa meno risorse tra la lista dei container presenti, ovvero il container con il valore di fitness maggiore. L'obbiettivo di questo algoritmo é effettuare tanti piccoli passi per bilanciare il cluster. Anche l'algoritmo Lazy termina quando la differenza tra i valori di fitness é minore di Omicron e come l'algoritmo Random progredisce in maniera ricorsiva.

6.4.3 Algoritmo dello Zaino

L'algoritmo Knapsack o problema dello Zaino si differenzia dall'algoritmo Random o Lazy perché prima viene creata una coda di operazioni che successivamente verrano svolte, a differenza dagli algoritmi precedentemente spiegati i quali prima analizzano la funzione fitness, effettuano la migrazione, e richiamano l'analisi in maniera ricorsiva finché la differenza tra le funzioni fitness é inferiore di Omicron, l'algoritmo dello Zaino non utilizza il parametro Omicron. L'algoritmo in questione inizia il proprio processo richiedendo ad ogni nodo del cluster la lista dei container in esecuzione, e su ogni conteiner ne calcola la funzione di fitness. L'algoritmo dello zaino richiede la capienza di uno zaino e una lista di elementi dove per ogni elemento é necessario conoscere peso e valore; gli elementi sono i container, e il loro peso e il loro valore é dato da (1 - funzione di fitness del singolo container), cosi facendo un container con una funzione di fitness pari a 0.28, il che vuol dire molto utilizzato, avrá un peso e un valore pari a 0.72, un elemento molto pesante ma molto importante, viceversa un container con una valore di fitness pari a 0.98, avrá un peso e un valore pari a 0.02, un elemento leggero ma futile all'interno del problema dello zaino. La capienza dello zaino é dato dalla somma dei pesi diviso il numero di container. Questo algoritmo ha il fine di distribuire in maniera solidale il peso computazionale dei singoli container trovando, nel caso peggiore, una soluzione per la quale un container molto pesante venga eseguito all'interno di un singolo nodo.

Capitolo 7

Validazione

In questo capitolo viene tratta la validazione dell'applicazione creata nei confronti di diverse casistiche per testarne l'affidabilitá, la capacitá di gestire il carico computazione di un cluster e di resistere nei confronti della fault tolerance del sistema o nell'aggiunta di un nuovo nodo Follower. Le casistiche analizzate sono:

- La capacitá di bilanciare il fitness dei nodi del cluster;
- La capacitá di saper gestire il carico computazionale dei confronti di un sistema dinamico;
- La resilenza del sistema nei confronti della caduta del Leader.

7.1 Elementi dei test

I test sono stati effettuati utilizzando vari container provenienti dalla stessa immagine; la quale é stata creata ad hoc per emulare un sistema reale. Il Dockerfile che costruisce l'immagine é il seguente:

```
FROM openjdk:8-jdk-alpine
ARG JAR_FILE=target/*.jar
COPY gs-spring-boot-docker-0.1.0.jar app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

La quale build produce un'immagine chiamata *somma*, la build viene effetuata con il seguente comando, *docker build -t somma .*, e produce un'ambiente java con all'interno un'applicazione, tale "gs-spring-boot-docker-0.1.0.jar", che é un progetto Spring-Boot alla quale occorre la porta interna 8080 per esporre un servizio REST che richiamato in GET passandogli due valori interi restituisce la somma. Il metodo che effettua la somma é il seguente:

```
@GetMapping("/somma/{a}/{b}")
int somma(@PathVariable int a, @PathVariable int b) {
        int somma = a + b;
        return somma;
}
```

Un esempio di richiesta GET é la seguente: `http://10.0.0.4:38081/somma/6/5`, che restitusice il valore 9.
Durante gli esperimenti condotti all'interno del sistema erano presenti vari container, i quali si distinguevano gli uni dagli altri mediante il nome e la porta con la quale si esponevano, per esempio nella casistica in cui il sistema doveva bilanciare 8 container, quest'ultimi erano portati nella fase di run con la seguente chiamata:

```
docker run -p 38081:8080 -d --name somma1 somma &&
docker run -p 38082:8080 -d --name somma2 somma &&
docker run -p 38083:8080 -d --name somma3 somma &&
docker run -p 38084:8080 -d --name somma4 somma &&
docker run -p 38085:8080 -d --name somma5 somma &&
docker run -p 38086:8080 -d --name somma6 somma &&
docker run -p 38087:8080 -d --name somma7 somma &&
docker run -p 38088:8080 -d --name somma8 somma
```

L'immagine somma ha un peso di 104.2MB e il tempo di trasferimento tra nodi é pari a 39 secondi, 22 per inviare l'immagine e 19 per essere caricata nel nodo di destinazione e avviata. I container erano stressati da un'applicazione che costantemente effettuava richieste REST per ottenere la somma di numeri generati in maniera Random. I test

sono stati effettuati in un cluster formato da Raspberry pi 3 b+; le loro caratteristiche tecniche rilevanti al fine dei test sono:

- OS: Raspbian GNU/Linux 10 (buster);
- CPU: Cortex-A53 (Armv8) 64-bit SoC-1.4 GHz, 4-core;
- RAM: 1 GB LPDDR2 SDRAM;
- DISCO: Samsung MB-ME32GA EVO Plus, microSD 32 GB;
- SCHEDA DI RETE RETE: Gigabit Ethernet over USB 2.0 (maximum throughput 300 Mbps).

Tutti i nodi del cluster sono connessi gli uni con gli altri mediante un cavo ethernet.

7.2 Analisi degli algoritmi con ambiente statico

L'analisi in un sistema statico va a validare l'effettivo funzionamento degli algoritmi. Con il termine sistema statico si va a indicare un esperimento dove il numero di container é fisso e non cambia nel tempo, a differenza del sistema dinamico dove il numero di container cresce col progredire del tempo. All'inizio delle validazioni il sistema statico, prima dell'esecuzione di ogni algoritmo, viene caricato con un numero di container, i numeri di container utilizzati sono: 3, 4, 6, 8, 10 e 12; é importante evidenziare che il nodo dove vengono inseriti i container é sempre lo stesso, questo fa si che il nodo indicato, tale nodo A, sará sbilanciato rispetto un ipotetico nodo X, se il caricamento fosse stato effettuato in maniera round robin il bilanciamento sarebbe giá stato effettuato in parte, prima dell'esecuzione dell'algoritmo. Dopo il caricamento dei container vengono eseguiti gli algoritmi e al loro termine ne viene analizzata la funzione di fitness. Il valore di fitness viene analizzato anche nell'istante di tempo dopo il caricamento dei container e prima dell'esecuzione dell'algoritmo, questo passaggio offre un prima e un dopo la computazione dell'algoritmo. Le analisi sono state effettuate per ogni algoritmo varie volte e il risultato presentato é frutto della media. Oltre a calcolarne la funzione di fitness per nodo, viene analizzata la differenza tra il minimo e il massimo valore di fitness, questo parametro é chiamato discrepanza. Se all'inizio dell'algoritmo é presente un'alta discrepanza, vuol dire che un nodo é molto piú carico degli altri, mentre se al termine dell'algoritmo la discrepanza diminuisce, significa che il sistema é riuscito a spalmare il peso computazionale dei contaier tra i vari nodi del sistema. Tutti gli algoritmi sono stati testati andando a valorizzare i paramentri della funzione di fitness, alfa, beta e gamma, spiegati nel capitolo 6.2, nei seguenti modi:

- "alpha":1.0,"beta":0.0,"gamma":0.0,"omicron":0.1, denominato test alfa
- "alpha":0.0,"beta":1.0,"gamma":0.0,"omicron":0.1, denominato test beta
- "alpha":0.0,"beta":0.0,"gamma":1.0,"omicron":0.1, denominato test gamma
- "alpha":0.33,"beta":0.33,"gamma":0.33,"omicron":0.1, denominato test combined

I risultati graficati sono presenti nei seguenti sotto capitoli, ogni colore delle linee verticali rappresenta il numero di container nel sistema, la guida che indica il colore a

quale numero di container viene associato é presente in basso ad ogni grafico.
Per ogni esperimento é presente il valore iniziare sulla sinistra e il valore finale sulla destra, questo produce sei gruppi di barre verticali, tre per la situazione iniziale e tre per la situazione finale.

7.2.1 Test Alfa

Valori di fitness:

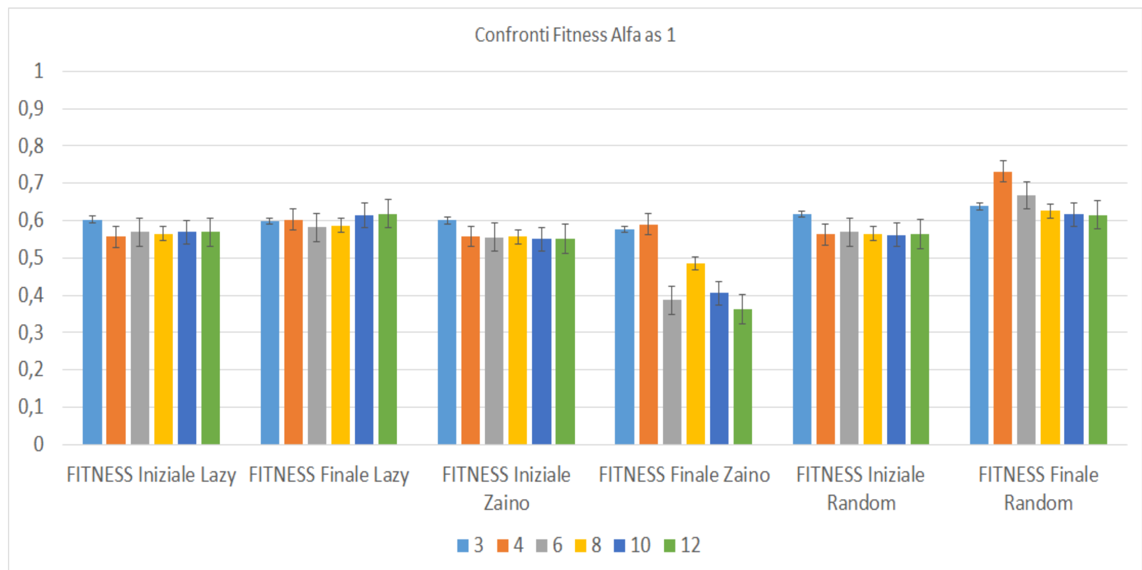

Figura 7.1: Grafico valori di fitness con alfa a 1

Valori discrepanza:

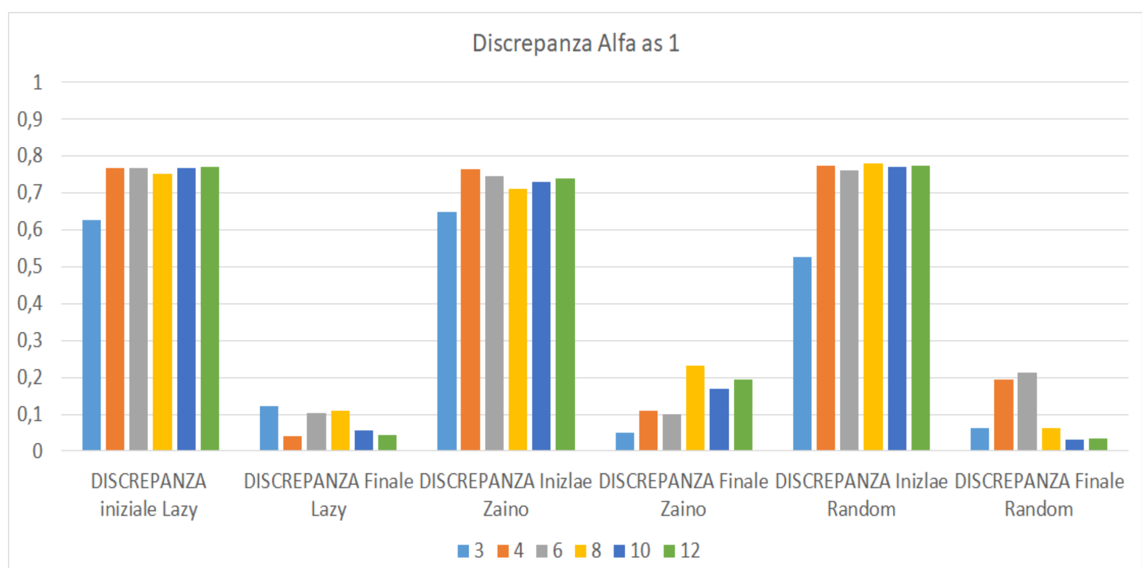

Figura 7.2: Grafico valori di discrepanza con alfa a 1

7.2.2 Test Beta

Valori di fitness:

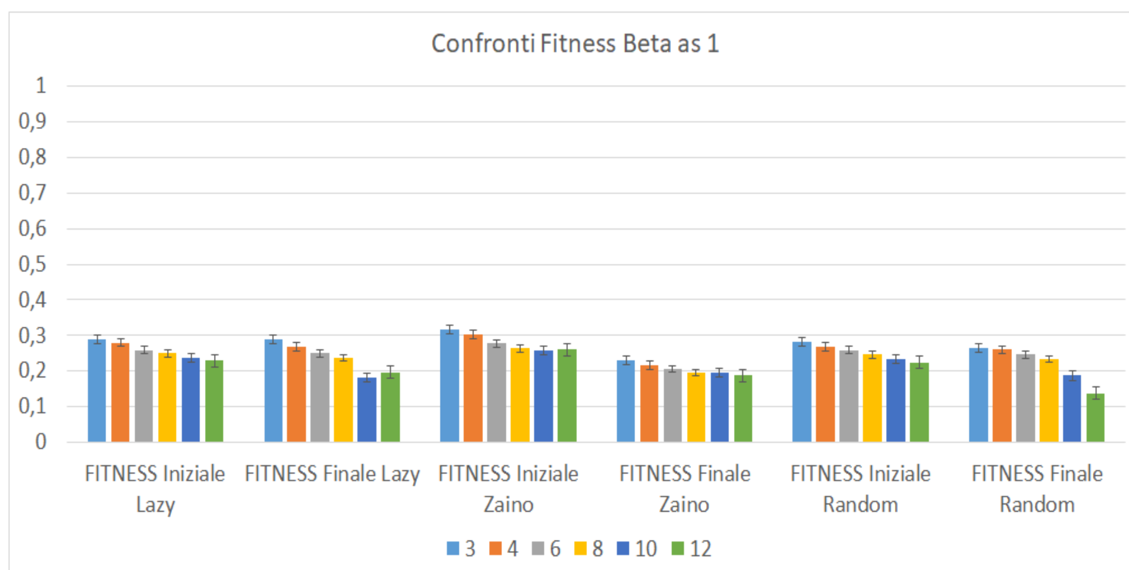

Figura 7.3: Grafico valori di fitness con beta a 1

Valori discrepanza:

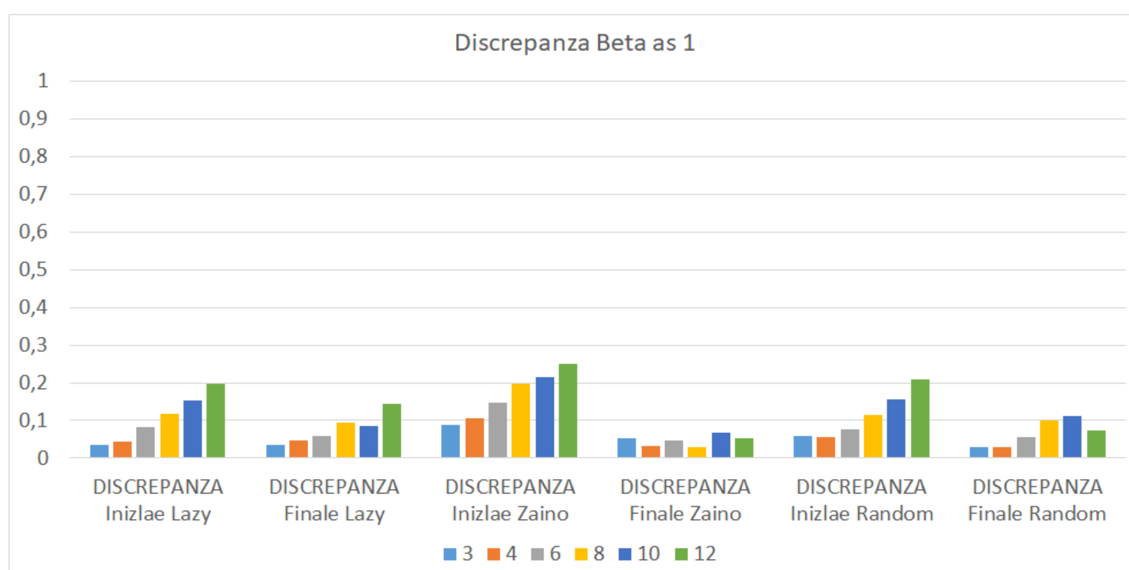

Figura 7.4: Grafico valori di discrepanza con beta a 1

7.2.3 Test Gamma

Valori di fitness:

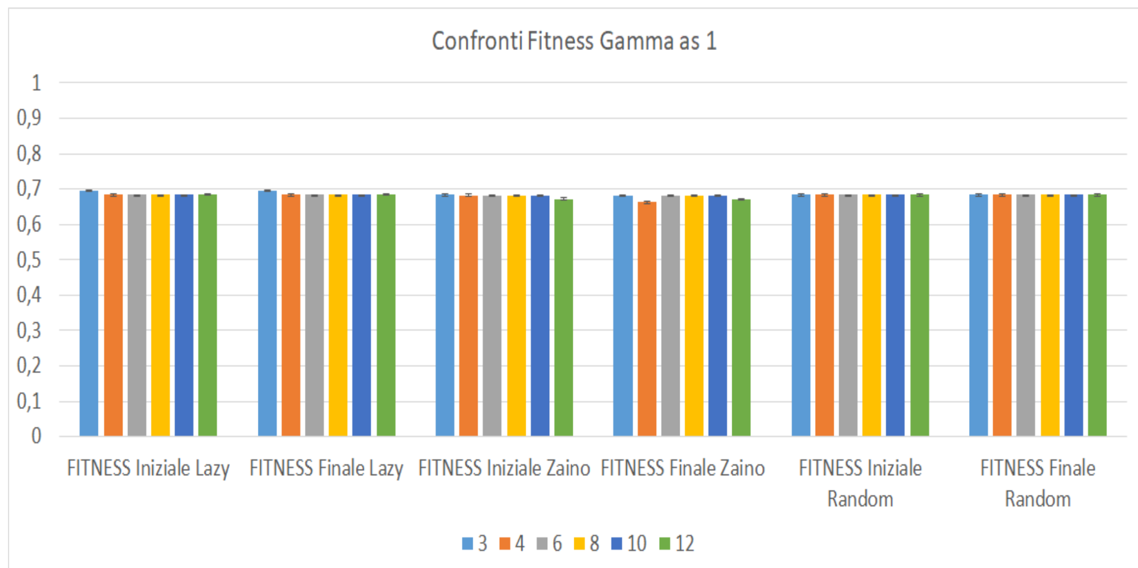

Figura 7.5: Grafico valori di fitness con gamma a 1

Valori discrepanza:

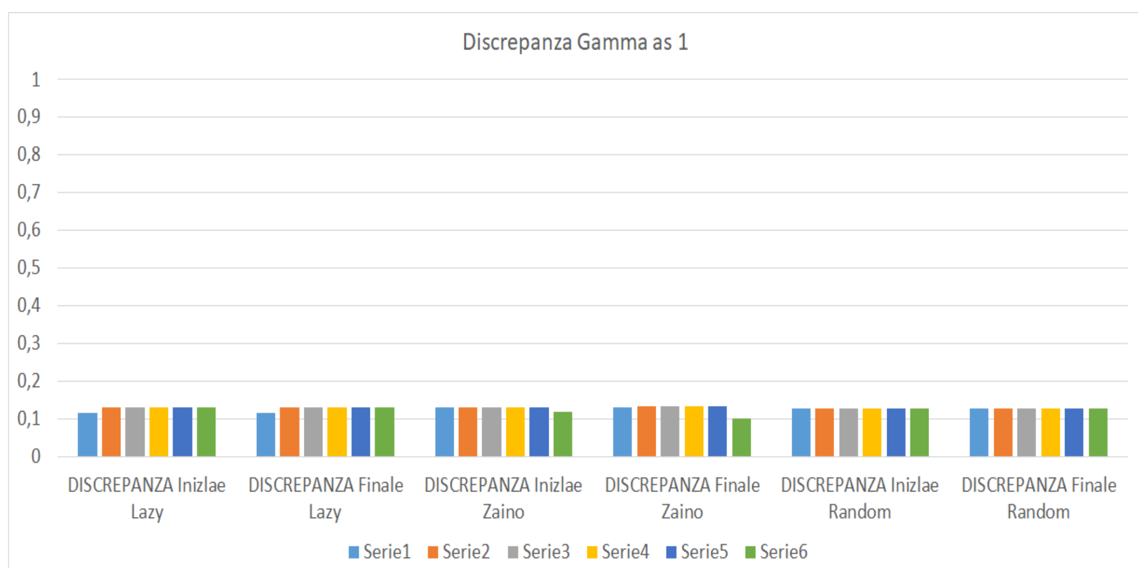

Figura 7.6: Grafico valori di discrepanza con gamma a 1

7.2.4 Test Combined

Valori di fitness:

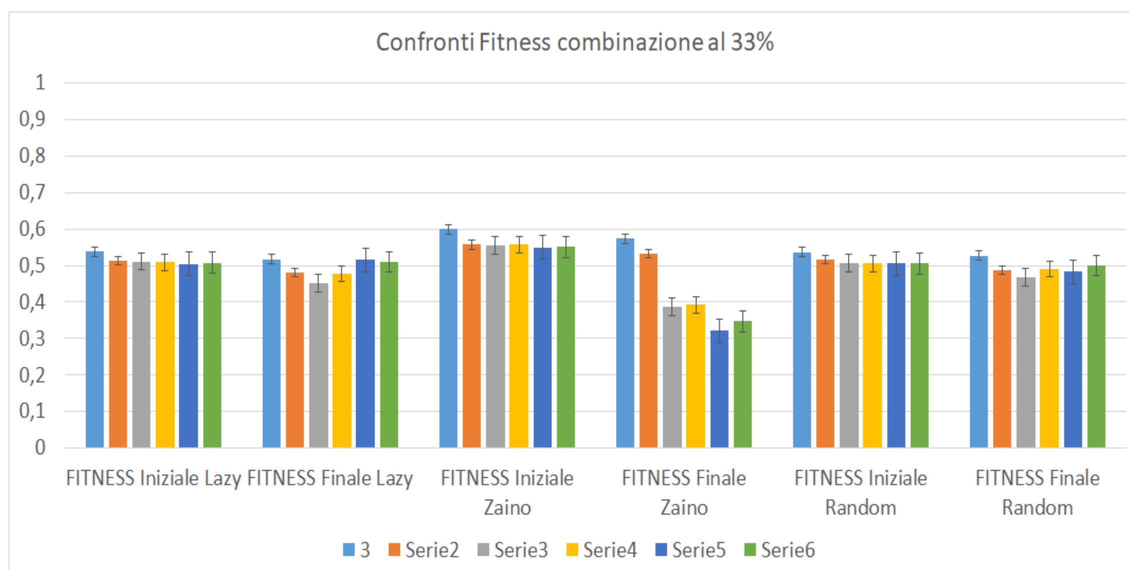

Figura 7.7: Grafico valori di fitness con combinazione allo 0.33

Valori discrepanza:

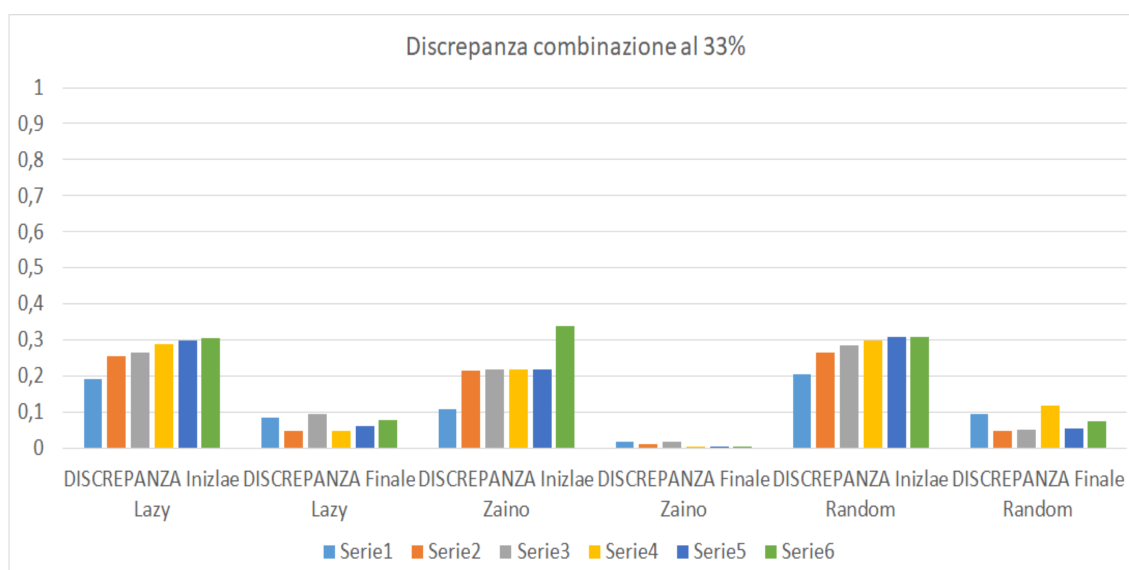

Figura 7.8: Grafico valori di discrepanza con combinazione allo 0.33

7.2.5 Analisi sui risultati

L'analisi alfa, che si basa solo sui dati provenienti dalla CPU, é soggetta a varianza in quanto il processore deve gestire anche processi non provenienti da Docker o dall'applicazione stessa, un Thread di sistema, che esula dal caso d'uso in analisi, potrebbe compromettere negativamente il valore di fitness. Una metrica molto convincente é quella beta, anche se la differenza in punti di fitness é minima, essa dimostra un chiaro andamento del sistema e mostra miglioramenti evidenti tra l'inizio di un algoritmo e il suo rapporto finale, nonostante i container in esecuzione non siano container che saturino molto la RAM. Il diminuire di container all'interno di un nodo provocano un decremento della percentuale di RAM utilizzata da Docker per allocare le risorse necessarie al funzionamento dell'architettura. Il valore di gamma prima e dopo la computazione rimane invariato se non di un'impercettibile differenza nella discrepanza dell'algoritmo dello Zaino all'aumentare del numero di conteiner, nonostante i dati prodotti, il valore gamma é molto importante perché restituisce un chiaro indicatore sulla saturazione del disco. Il successivo grafico rappresenta il decremento del valore di fitness all'aggiunta di file di dimensione 1.8GB, all'aumentare dei file il valore ovviamente diminuisce.

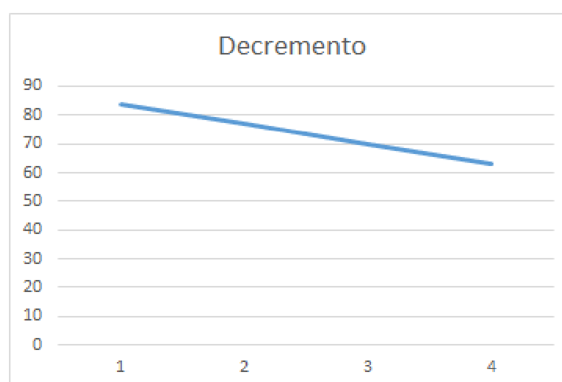

Figura 7.9: Decremento della funzione di fitness con un aumento di dati sul filesystem

Tale grafico porta l'analisi su un punto di vista differente, esso evidenzia che se un nodo ha pochi processi in esecuzione, quindi un alto valore di CPU e di RAM, ma ha il disco saturato al 95%, deve essere considerato come un nodo meno performante rispetto ad uno con valori di CPU e RAM peggiori ma un valore di disco migliore, é importante

preservare il disco di un nodo al fine di non saturarlo con la migrazione di un'immagine. La combinazione piú performante, dopo l'analisi fatta sulle altre tre casistiche, é quella combined. La combinazione delle tre metriche restituisce un valore che va a correggere le inesattezze di utilizzo delle tre metriche prese saparatamente, essa riesce ad attenuare la varianza della CPU, e dá il giusto peso alla RAM e al disco.
Focalizzando l'attenzione sul grafico 7.7, l'algoritmo piú performante é quello dello Zaino, in quanto ha una capacitá di "spalmare" i container sui nodi migliore dell'approccio Random e di quello Lazy che, nonostante i risultati di discrepanza molto soddisfacenti, procedono a step guidati dal fattore di Omicron. L'algoritmo dello Zaino, nonostante impieghi piú risorse degli altri due, é riuscito a diminuire il fattore di fitness medio dopo la sua computazione. L'algoritmo Lazy ha il vantaggio, a differenza dell'appoccio dello Zaino, di arrivare piú velocemente alla soluzione, ovviamente la velocitá di risoluzione é inversamente proporzionale alla dimensione di Omicron; piú omicron é piccolo, piú sono necessari molti passaggi per calibrare il sistema. L'approccio Random, nel grafico 7.1, ha prodotto un aumento della funzione di fitness media, nonostante il valore di discrepanza sia molto soddisfacente, questo perché l'approccio Random, per sua natura, sceglie il container da spostare in maniera Randomica senza un analisi sul "peso" del container, questo produce scelte inesatte; nonostante l'analisi fatta, l'algoritmo Randomico soddisfa parzialmente l'obbiettivo, spostare un qualcosa, da un nodo meno performante a uno migliore.

7.3 Analisi degli algoritmi con ambiente dinamico

A differenza dell'analisi precedemente svolta, nella quale l'ambiente era statico, in questa analisi il sistema é dinamico, ovvero il numero di container aumenta con l'aumentare del tempo. Il test dura 30 minuti, al minuto 0 nel cluster non sono presenti container, al minuto 2 verranno eseguiti 3 container, fino ad arrivare al minuto 17 dove ve ne saranno 12, dal minuto 17 al 30 il sistema si adatta al cambiamento. Nello specifico i container sono eseguiti ai seguenti minuti:

- minuto 0, numero container totali nel cluster 0;
- minuto 2, numero container totali nel cluster 3;
- minuto 5, numero container totali nel cluster 4;
- minuto 8, numero container totali nel cluster 6;
- minuto 11, numero container totali nel cluster 8;
- minuto 14, numero container totali nel cluster 10;
- minuto 17, numero container totali nel cluster 12;
- minuto 30, numero container totali nel cluster 12.

La seguente analisi é stata redatta con la casistica alfa e combined. I grafici che riportano i risultati sono i seguenti, per ogni casistica viene mostrata la fitness dei tre algoritmi e il valore di fitness del nodo A, in quanto i container vengono aggiunti sempre nel nodo A, questo é stato fatto per dimostrare che, all'aggiunta di un contaienr il suo valore di fitness scende, ma col progredire del tempo lo stesso tende a salire. Le acqusizioni dei valori di fitness vengono fatte per ogni nodo ogni 30 secondi, ne consegue che per la seguente analisi sono state fatte 60 rilevazioni per nodo. Nei seguenti grafici le barre gialle indicano il momento in cui vengono aggiunti i nodi al sistema.

7.3.1 Test Alfa

Media fitness dei nodi

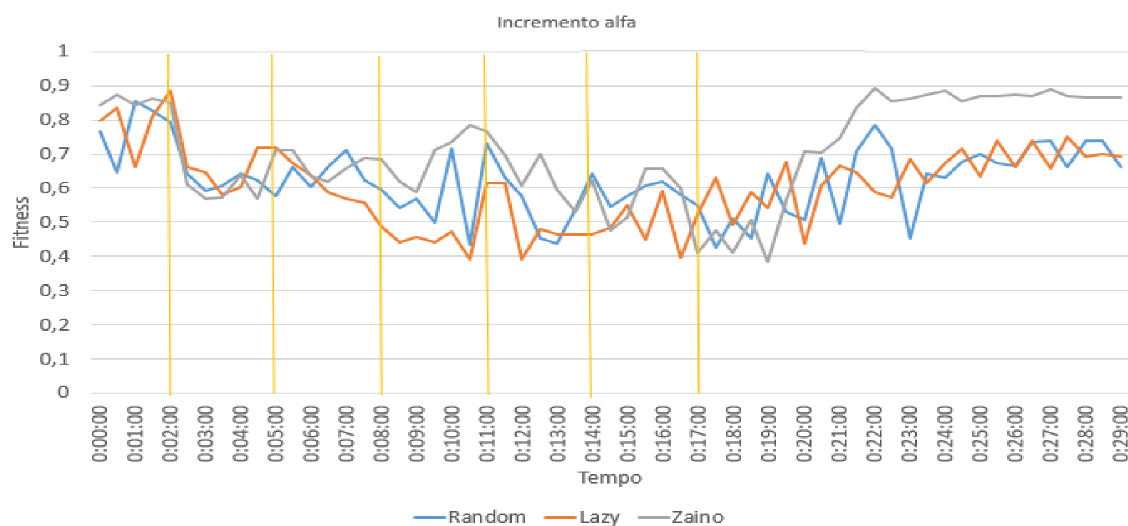

Figura 7.10: Valori di fitness in un ambiente dinamico con casistica alfa

Valori di fitness del nodo A

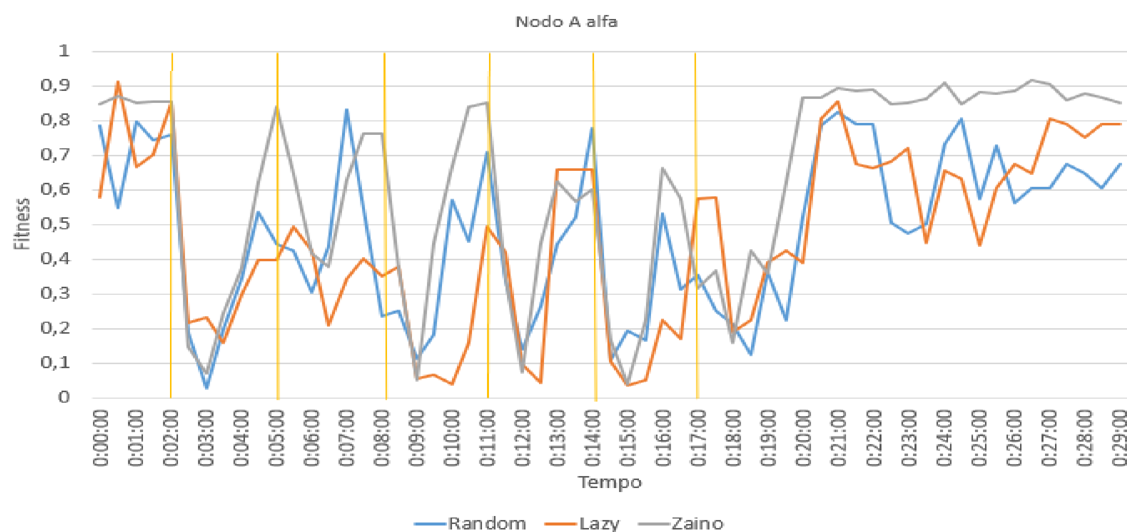

Figura 7.11: Valori di fitness del nodo A in un ambiente dinamico con casistica alfa

7.3.2 Test Combined

Media fitness dei nodi

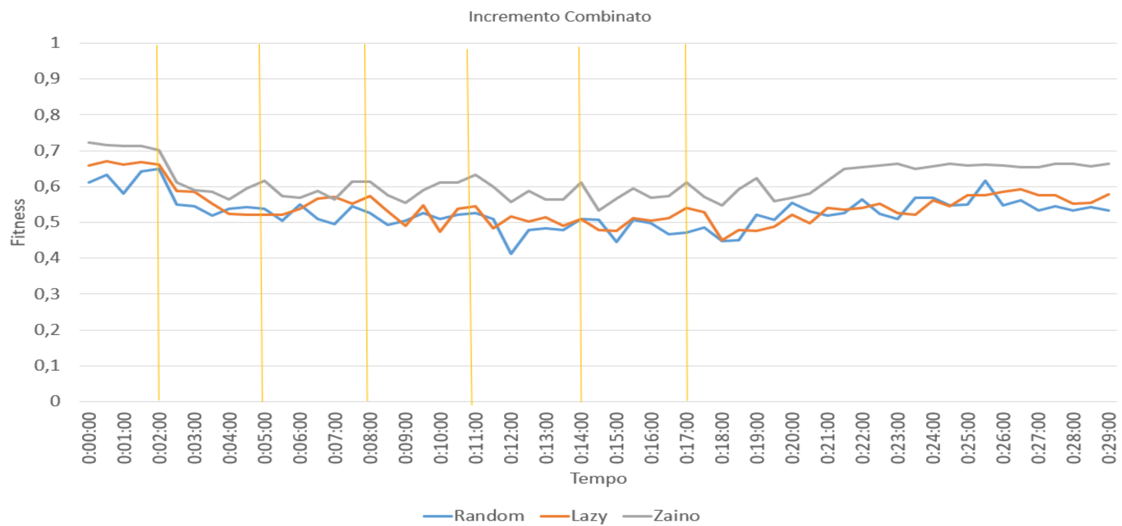

Figura 7.12: Valori di fitness in un ambiente dinamico con casistica combined

Valori di fitness del nodo A

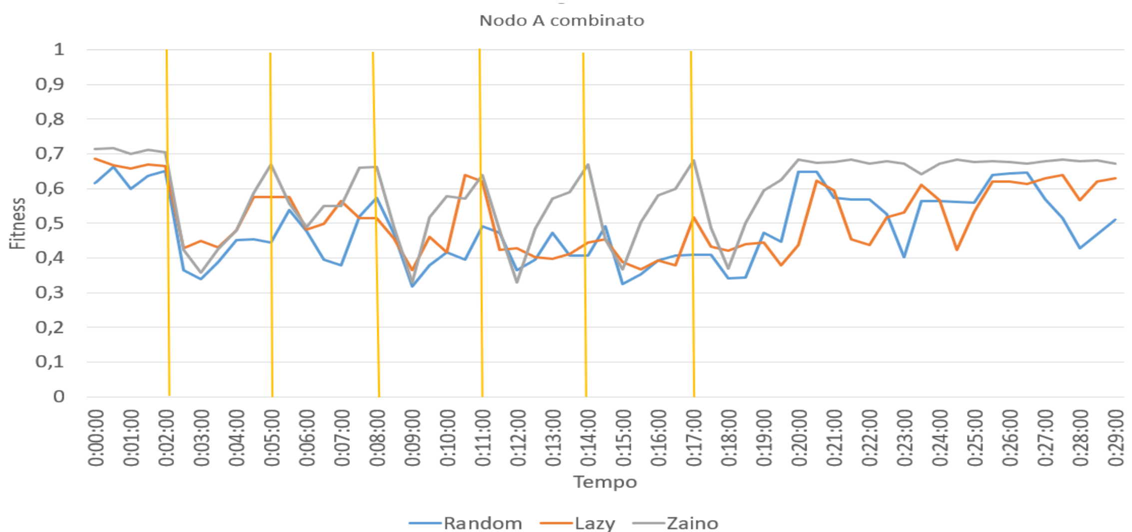

Figura 7.13: Valori di fitness del nodo A in un ambiente dinamico con casistica combined

7.3.3 Analisi sui risultati

Concentrando principalmente l'attenzione sulla figura 7.13, nella quale é impostato un approccio combined, quello che emerge é che all'aggiunta di n container al nodo A il suo valore di fitness scende drasticamente per poi risalire; in particolar modo l'algoritmo dello Zaino ha un andamento sinusoidale quasi costante. Il motivo di questo andamento é nella prontezza nel ridistribuire i container all'interno del cluster, infatti appena vengono inseriti i container nel sistema, esso raccoglieva i paramentri dai vari nodi del cluster e ne ridistribuiva i container. L'algoritmo Lazy, concentrandosi sul grafico 7.10, mostra un decremento non appena vengono aggiunti dei container nell'ambiente, ma dal minuto 17 in avanti mostra miglioramenti crescenti dovuti al fatto che esegue sempre delle piccole mosse per arrivare al proprio Goal. L'algoritmo Random invece ha un andamento imprevedibile, ma come l'approccio Lazy, tende a stabilizzarsi nel tempo se al sistema non vengono aggiunti nodi. Osservando 7.12 dal minuto 2 al minuto 8 quello che si puó notare é che l'algoritmo Lazy si comporta molto bene e con pochi container da analizzare ha un andamento migliore di quello Random il quale andamento é imprevedibile. Nello stesso grafico si puó osservare il calo di performance dello Zaino ogni qual volta sono aggiunti container al sistema, questo comportamento é motivato dal fatto che ogni volta che lo Zaino deve creare le migrazioni quest'ultimo inizia il proprio flusso con un analisi sullo stato dei nodi del cluster, questa analisi richiede parecchie risorse e questo fa diminuire per un breve lasso di tempo di valore di fitness.

7.4 Analisi sulla resilienza del sistema

Nel sistema descritto vi é un Leader e diversi Follower, l'analisi della resilienza mira a verificare che se nel cluster il Leader non sia piú presente a fronte di un evento inaspettato, il sistema si riconfiguri eleggendone un altro che porti avanti il lavoro di quello precedente. Anche in questo caso ad essere sotto analisi sono tutti e tre gli algoritmi e le modalitá coinvolte sono quella combined e quella alfa. Il test si sviluppa nel seguente modo, la durata é di 30 minuti, al minuto 0 non sono presenti container nel cluster, al minuto 2 ne vengono aggiunti 10. Al minuto 10, dopo che il sistema é a un buon livello di bilanciamento viene spento il Leader, a questo punto il sistema lo percepisce e ne elegge un altro, questa operazione potrebbe richiedere anche 2 minuti, successivamente, il Leader controlla lo stato del sistema andando a capire quali container non sono piú disponibili nel sistema, perché erano in stato di run nel nodo caduto. Una volta che il nuovo Leader ha la lista dei container non piú disponibili, li ricarica al suo interno e ricomincia l'analisi per distribuire i container. Al minuto 20 il nodo che prima era stato spento viene riacceso, il quale capisce che é giá presente un cluster, vi si aggiunge e ne entra a far parte. Anche in questa analisi come in quelle precenti é bene sottolineare che i container presenti nel cluster sono stressati da un'applicazione che costantemente ne effettua richieste e di conseguenza ne crea della computazione interna al cluster che si ripercuote sulle performace del nodo. I grafici ottenuti sono i seguenti. Le acqusizioni dei valori di fitness vengono fatte per ogni nodo ogni 30 secondi, ne consegue che per la seguente analisi sono state fatte 60 rilevazioni per nodo.

7.4.1 Casistica Combined

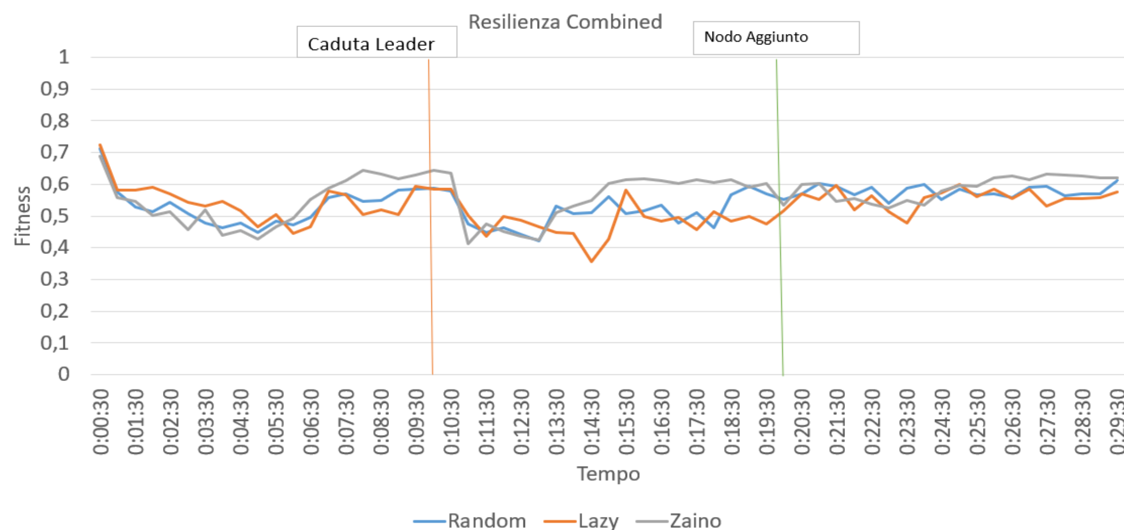

Figura 7.14: Analisi sulla resilienza degli algoritmi con un approccio combined

7.4.2 Casistica Alfa

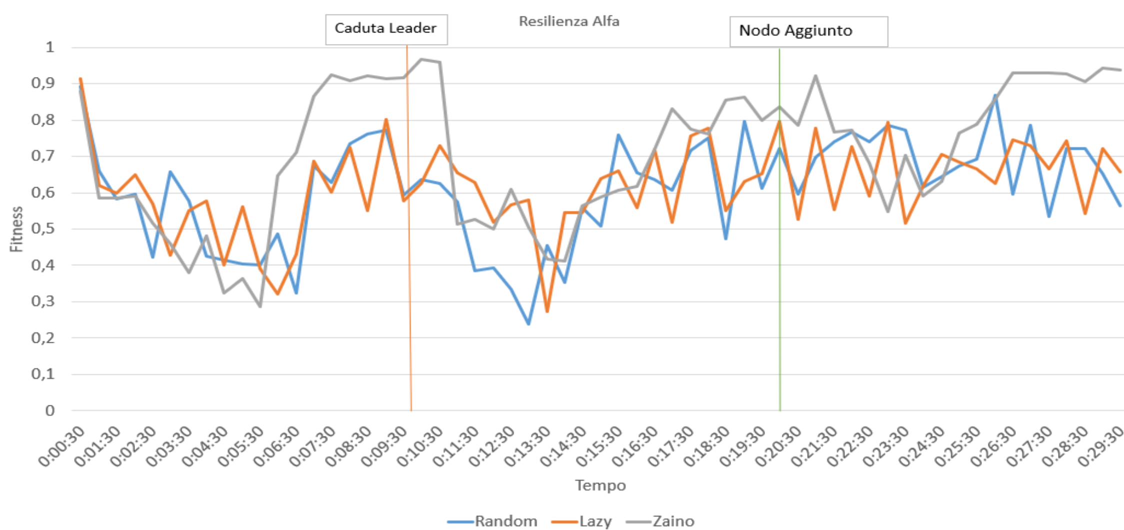

Figura 7.15: Analisi sulla resilienza degli algoritmi con un approccio alfa

7.4.3 Analisi sui risultati

Concentrando l'attenzione in particolar modo su 7.15, ed in particolar modo sull'algoritmo dello Zaino quello che emerge é che, dopo la caduta del Leader, il sistema vive un momento di incertezza e la funzione di fitness sembra quasi aumentare, per poi crollare drasticamente. Questo andamento é motivato dal fatto che il sistema prima del minuto 10 era giá bilanciato, e la perdita del Leader non ha comportato un cambiamento nel sistema in quanto i nodi avevano giá valori di fitness simili e la media per il numero di nodi o per il (numero di nodi - 1) non produce un cambiamento visibile. Il comportamento non é lo stesso se si osserva l'andamento degli algoritmo Random e Lazy che dopo la caduta del Leader hanno un primo crollo di fitness, questo comportamento é motivato dal fatto che il cluster prima della caduta del Leader non era bilanciato, e il nodo con il valore di fitness maggiore era il Leader, la sua mancanza nel sistema ha portato a calcolare la media di fitness con un valore in meno, il piú alto. Quello che accomuna i tre algoritmi é che tra il minuto 11 e il minuto 15 gli algoritmi vivono un momento in cui il valore fitness decresce molto, perché in quel lasso di tempo i Leader hanno caricato i contaier che non erano piú presenti nel sistema e da quel punto in avanti iniziano la distribuzione nel cluster. Osservando il grafico 7.14 sul minuto 11, quello che si nota é che per un lasso di tempo di 1 minuto la funzione dello Zaino si comporta peggio rispetto gli altri due algoritmi, questo comportamento é motivato dal fatto che il nuovo Leader dopo che ha fatto ripartire i container scomparsi ha dovuto acquisire le metriche dei nodi rimanenti e creare per ognuno di essi un nuovo Zaino. Questo passaggio necessita di un fattore temporale non presente negli altri due nodi dove hanno un approccio piú diretto spostando un elemento da un nodo peggiore a uno migliore. Sempre l'algoritmo dello Zaino nel grafico 7.14 al minuto 13:30, dopo che ha creato gli zaini inizia un incremento quasi costante verso il suo massimo di fitness.
Al minuto 20, nel cluster riappare il nodo precedentemente caduto, il quale si unirá al sistema come Follower in quanto é giá presente un Leader. In entrambi i grafici per gli algoritmi Lazy e Random questo rappresenta un picco per il valore di fitness, perché a fare la media tra i valori saturi di container si aggiunge un nodo privo di essi, con un valore di fitness alto, che porta di conseguenza un aumento della media. Lo stesso comportamento non accade per l'algoritmo dello Zaino che dopo l'aggiunta di un nodo

"libero" al cluster vive un decremento medio, tale andamento é motivato dal fatto che prima dell'aggiunta il sistema era bilanciato, con l'aggiunta di un nuovo nodo, il Leader deve ottenere le informazioni da tutti i nodi e ridistribuire i nodi, o per meglio dire, far inviare dai nodi del cluster dei container al nuovo nodo, questa operazione richiede un grande apporto di lavoro e come si puó vedere da 7.15 provoca un crollo del valore di fitness, ripagato da un incremento nei minuti successivi. L'algoritmo Lazy vive l'aggiunta di un nuovo nodo come un momento di incertezza, il valore di alfa oscilla tra 0.5 e 0.7, l'andamento é motivato dal fatto che il sistema si deve adattare a una nuova configurazione e il valore alfa, che rispecchia in maniera diretta quanto la CPU sta lavorando é estremamente sensibile al lavoro che deve compiere, l'aggiunta di un nuovo nodo al sistema comporta lavoro extra per tutti i nodi che devono inviare o caricare dei container, lo stesso andamento, piú lieve é visibile anche in un approccio combinato. L'algoritmo Random, con alfa a 1 arriva a toccare il minimo del sistema, con 0,238, questo perché scegliendo i container da muovere senza un criterio preciso puó capire di effettuare la mossa sbagliata, questo peró puó avvenire anche se la metrica di decisione é piú accurata come si puó vedere al minuto 14 dell'algoritmo Lazy con alfa impostato a 1. Il fatto che un l'algoritmo Random o Lazy effettuino una mossa sbagliata non é da vivere come un grande problema perché se la mossa sbagliata comporta che il nodo X diventi il peggiore, questo implica che nello step successivo il nodo X verra scelto per inviare un container all'ipotetico nodo Y che il quel momento risulta essere il migliore.

Conclusioni

In questo elaborato di tesi viene presentato com'é stata sviluppata un'applicazione che possa gestire container Docker all'interno di un cluster, col fine di ridurre il carico di lavoro che si creerebbe su un nodo se molti container venissero eseguiti al suo interno.
Quello che é stato sviluppato é una soluzione che offre una reale soluzione ad un problema; l'Edge computing dalle piccole aziende a quelle di maggiore rilievo rappresenta un vero apporto di computazione disponibile il piú vicino possibile a dove il servizio viene richiesto, oltre che rappresentare una tecnica di pre-elaborazione del dato con l'ausilio di appositi servizi. La presenza di servizi comporta peró anche la necessitá che questi ultimi vengano gestiti ed eseguiti all'interno di un calcolatore capace di garantire l'esecuzione del container in un nodo libero da processi che potrebbero gravare sulla performance.
Sicuramente un caposaldo dello sviluppo é dato dal fatto che l'applicazione sia distribuita e che il Leader possa variare nel tempo, questo offre anche il vantaggio che cambiando nel tempo, é meno probabile che il Leader si trovi in situazioni in cui il proprio valore di fitness sia peggiore rispetto ad altri nodi; l'idea di utilizzare un approccio centralizzato piuttosto che uno distribuito é stato preso in considerazione in fase di ideazione, ma non scelto per garantire la massima fault tolerance.
Quello che ha guidato lo sviluppo durante i mesi di creazione dell'applicazione é stato quello di utilizzare approcci totalmente indipendenti dal linguaggio di programmazione utilizzato, la stessa applicazione potrebbe venir sviluppata in altri linguaggi. Questo garantisce la libertá in futuro di migrare ad uno con caratteristiche diverse che potrebbero adattarsi meglio al caso d'uso richiesto. Un esempio concreto é stata la scelta di utilizzare come protocollo di Service Discovery uPnP, il quale é un'architettura aperta basata sui protocolli Internet standard come TCP, UDP e HTTP, le stesse chiamate che

vengono effettuate a Docker potrebbero essere implementate da altri linguaggi di programmazione.
Il lavoro ha richiesto molti mesi di analisi, implementazione e validazioni, ma é stato appagato dai risultati finali. La prima analisi effettuata é stata quella di capire come la run di un container potesse essere mossa tra due nodi, studio non semplice che ha richiesto parecchio tempo in quanto non sono presenti soluzioni commerciali o approcci standard che permettessero tale procedura, lo stesso é stato complicato ancor di piú quando si é iniziato a studiare le possibilitá offerte da Docker per quanto riguarda la migrazione di un volume. Una volta che si é stati certi che la migrazione funzionasse lo step successivo é stato quello di capire le varie modalitá di Service Discovery ed analizzare quella che meglio si poteva adattare a quello che era stato precedentemente sviluppato; anche questo step ha richiesto parecchio tempo e prima di scegliere UpNp ne sono state testate altre come mDNS-SD. Gli ultimi steps sono stati quelli sulle policy di migrazione, la creazione degli algoritmi e la loro validazione.
Analizzando i risultati presentati si puó concludere come l'algoritmo dello Zaino riesca a bilanciare il carico computazionale in maniera sofisticata e a diminuire l'effettvo valore di fitness di tutto il cluster, nonostante ció gli altri algoritmi ottengono ottimi risultati in particolar modo quando si tratta dell'analisi sulla discrepanza. É stato analizzato nel corso delle validazioni come tutti e tre gli algoritmi riescano a gestire in maniera ottimale l'aumento del numero di container su di un nodo nel tempo e della loro effettiva capacitá di rialzare il valore di fitness. Oltre a ció é stato studiato come a fronte di un evento imprevisto come la caduta del Leader il sistema sia realmente in grado di riconfigurasi e di ri-eleggerne un secondo e di continuare il proprio lavoro, in particolar modo anche in questo caso l'algoritmo dello Zaino, una volta appreso la non piú presenza del coordinatore nel cluster, ed aver avuto un picco negativo di fitness, é riuscito a ri-incrementare il proprio valore.
Per quanto riguarda gli sviluppi futuri sicuramente due sono le principali strade: la prima riguarda ampliare la gamma di tool di container gestiti, in questo lavoro di tesi é stato preso in analisi Docker, principalmente per la sua popolaritá, ma altri sistemi sono altrettanto famosi come per esempio Vagrant. La seconda oppurtunitá di sviluppo potrebbe essere quella di utilizzare algoritmi di machine learning per determinare pre-

ventivamente la migrazione di un container prima che esso stesso rappresenti un peso per il nodo che lo esegue.

Appendice A

Prima Appendice

A.1 Descrittore XML dell'applicazione UpNP

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<root xmlns="urn:schemas-upnp-org:device-1-0">
<specVersion>
<major>1</major>
<minor>0</minor>
</specVersion>
<device>
<deviceType>urn:schemas-upnp-org:device:dockerMigration:1</deviceType>
<manufacturer>ARUBO</manufacturer>
<modelDescription>An applicatio to migrate service.</modelDescription>
<modelName>0.0.0.8</modelName>
<modelNumber>v1</modelNumber>
<UDN>uuid:dae615e2-6395-0592-0000-00001eccef34</UDN>
<serviceList>
<service>
<serviceType>urn:schemas-upnp-org:service:MigratePower:1</serviceType>
<serviceId>urn:upnp-org:serviceId:MigratePower</serviceId>
<SCPDURL>/dev/dae615e2-6395-0592-0000-00001eccef34/svc
```

```
/upnp-org/MigratePower/desc</SCPDURL>
<controlURL>/dev/dae615e2-6395-0592-0000-00001eccef34/svc
/upnp-org/MigratePower/action</controlURL>
<eventSubURL>/dev/dae615e2-6395-0592-0000-00001eccef34/svc
/upnp-org/MigratePower/event</eventSubURL>
</service>
</serviceList>
</device>
</root>
```

Bibliografia

[1] P. Bellavista, A. Corradi, L. Foschini, and D. Scotece. Differentiated service/data migration for edge services leveraging container characteristics. *IEEE Access*, 7:139746–139758, 2019.

[2] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16, 2012.

[3] R. Chamberlain and J. Schommer. Using docker to support reproducible research. *DOI: https://doi. org/10.6084/m9. figshare*, 1101910:44, 2014.

[4] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286, 2005.

[5] Docker. How services work. [ultima vista 05.09.2020].

[6] Docker. Introduction to docker swarm architecture. [ultima vista 02.09.2020].

[7] Docker. Swarm mode key concepts. [ultima vista 04.09.2020].

[8] R. Dua, A. R. Raja, and D. Kakadia. Virtualization vs containerization to support paas. In *2014 IEEE International Conference on Cloud Engineering*, pages 610–614. IEEE, 2014.

[9] H. S. Enreina Annisa Rizkiasri, Francisco Morales and M. Riftadi. Kubernetes - container orchestration. [ultima vista 13.09.2020].

[10] K. A. Henderson. An oral life history perspective on the containers in which american farm women experienced leisure. *Leisure Studies*, 9(2):121–133, 1990.

[11] Medium. Learning kubernetes, design consepts. [ultima vista 12.09.2020].

[12] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[13] R. A. Meyer and L. H. Seawright. A virtual machine time-sharing system. *IBM Systems Journal*, 9(3):199–218, 1970.

[14] J. MSV and L. Hecht. *The State of the Kubernetes Ecosystem, Second Edition*. The New Stack, 2020.

[15] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.

[16] Raft. The raft consensus algorithm. [ultima vista 05.09.2020].

[17] M. Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[18] W. Shi, G. Pallis, and Z. Xu. Edge computing [scanning the issue]. *Proceedings of the IEEE*, 107(8):1474–1481, 2019.

[19] M. Zwolenski, L. Weatherill, et al. The digital universe: Rich data and the increasing value of the internet of things. *Journal of Telecommunications and the Digital Economy*, 2(3):47, 2014.

Ringraziamenti

Voglio ringraziare mia Madre Paola e mio Padre Giancarlo per esseci sempre stati nei momenti di difficoltá e per avermi sempre dato la forza di non mollare quando i problemi rendevano la fine di questo percorso molto lontana.

Voglio ringraziare i miei nonni Andrea e Gabriele e le mie nonne Emma, Pupi ed Ettorina per avermi sempre trattato come un figlio e non avermi mai fatto mancare niente. Sono molto grato a tutti voi.

Ringrazio Giulia, per esserci sempre stata e per avermi sempre allungato la mano nei momenti di difficoltá. Sei davvero molto importante per me.

Ringrazio i miei migliori Amici Andrea ”Guio” e Mattia ”Gaet”, per essere come dei fratelli per me e per essere sempre stati presenti nei momenti belli e in quelli meno belli. Vi voglio davvero molto bene.

Ringrazio Riccardo ”Covi”, Luca ”Dona” e Michele ”Miki” per essere dei buon amici e di farmi star bene.

Voglio ringraziare i miei compagni di viaggio Sara e Stefano, con i quali io e Giulia condividiamo dei momenti molto belli ed ai quali tengo molto.

Ringrazio i miei compagni dell’universitá Mattia e Jean perché abbiamo affrontato insieme dei bellissimi momenti e perché soprattutto ci siamo divertiti molto.

Ringrazio la CDM per avermi formato in questi anni ed in particolar modo Andrea perché oltre ad essere un responsabile é anche un amico.

Per ultimi ma non per importanza ringrazio il Professor Marco Di Felice che in questi anni mi ha seguito come studente e mi ha guidato in questo progetto di tesi insegnandomi davvero tanto e il Dott. Lorenzo Gigli per avermi seguito molto meticolosamente in questo percorso finale, vi ringrazio davvero molto, grazie e voi ho capito che la strada che ho intrapreso é quella giusta.